University of Derby
Department of Electronics, Computing & Mathematics

A project completed as part of the requirements for
BSc (Hons) Computer Games Programming

entitled

Architectural Intelligence: Balancing team-based first person shooters using reactive real-time procedural level generation

by
Samuel Peter Kinnett
100320071
s.kinnett1@unimail.derby.ac.uk

May 2018

1 Abstract

Like most competitive multiplayer games, first person shooters have the potential to suffer from large imbalances in terms of team skill. While approaches such as intelligent matchmaking and handicaps exist, this paper presents a novel approach to balancing in which the level geometry is modified to confer advantages to the weaker team. This technique is demonstrated in a game developed alongside the study which was used in a series of experiments to show that procedural real-time level generation has the potential to balance team based first person shooters.

2 Acknowledgements

I would like to thank my dissertation supervisor Amanda Whitbrook for providing support and feedback throughout the dissertation process. I would also like to thank the numerous people who took part in the experiments in this study, and my friends who helped proof read this paper.

Table of Contents

3 Introduction

In first person shooters, as in many competitive games, it is possible for situations to arise in which a team of highly skilled individuals may face off against a team which is comparatively less skilled. For those in the highly skilled team this can prove boring as the gameplay becomes less challenging. Those in the less skilled team also have a sub-optimal experience as they are unable to win due to the difference in skill. While solutions to this problem exist, such as the matchmaking system in Counter-Strike: Global Offensive (Valve Corporation, 2012) which attempts to create matches where players all have similar skill (Counter-Strike Wiki, 2018), or handicap systems in games like Worms Armageddon (Team 17, 1999) where teams can be given more or less energy prior to beginning a game, this study aims to propose a novel approach to match balancing where matches are not only balanced in real time but also balanced by changing the level geometry to confer an advantage to the weaker team.

3.1 Hypothesis

Continuous, reactive procedural level generation is an effective method of providing real-time balance in a multiplayer team-based first person shooter.

3.2 Aims and Objectives

The aim of this project is to develop a team-based multiplayer first person shooter in which the environment is re-generated during play to provide a proportional advantage to the weaker team and vice versa. This outcome will be achieved through the following objectives:

1. A literature review will be performed into the areas of procedural FPS level generation, artificial intelligence directors, and methods of creating metrics for measuring player performance.
2. A simple client-server multiplayer first person shooter incorporating the aforementioned level generation will be developed.
3. Experiments will be conducted using volunteers with varying levels of skill and experience in the FPS genre in which they will play the game against each other, with both static and dynamically generated levels. Individual and team performance will be recorded throughout and a questionnaire provided to glean participant opinions to supplement the hard data.
4. The gathered data will be used to evaluate the effectiveness of the dynamic generation in balancing the teams, by comparing the results with dynamic generation to the control experiments without.

4 Literature Review

The work in this thesis is heavily focused on procedural content generation and the concept of actively directing this generation, in order to bring balance between teams of potentially disparate skill levels. While the concept of real time level modification is seemingly unexplored, a great deal of research has already been conducted in the more general area of procedural content generation for first person shooters. In this chapter, existing research will be examined in order to asses current solutions and approaches. These in turn will then guide the development within this thesis.

The strategy for this literature review was a keyword search in each of the key areas in both academic portals and more general purpose search engines. Where relevant material was found, the references within were then investigated further.

4.1 Procedural Content Generation

Procedural Content Generation (henceforth referred to as PCG) forms one of the core focuses of the research in this paper. Initial research will be focused on the core principles of generating multiplayer first person shooter levels, before moving on to examine methods of evolving maps and adjusting level layout for balance.

4.1.1 Search-based Procedural Generation

The first paper that will be examined is one that presents the core ideas behind many of the other research in this section. In this paper, Togelius et al. (2011) introduce the field of Search Based Procedural Content Generation (SBPCG) and identify some of the key challenges facing this area of research.

Search-based procedural generation is similar to the generate-and-test approach to procedural content generation. In a generate-and-test system, content is generated and then a series of tests are performed on it to ensure it meets a required standard. If it does not, some or all of the content is discarded and the process repeats itself until the standard is met. SBPCG differs from the generate-and-test approach in two ways: the tests are used to evaluate the content in some way and assign it a fitness value, rather than simply accepting or rejecting it, and future content generation is performed with the intent of getting higher fitness values than the previous generation (Togelius et al., 2011).

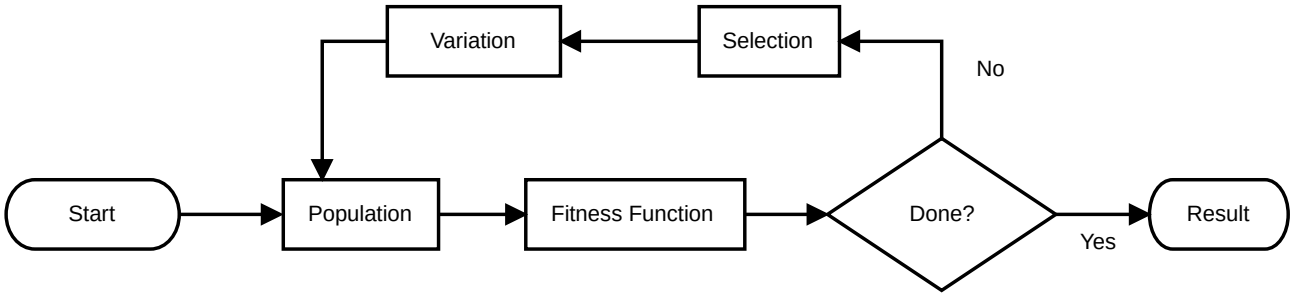

Figure 4.1: A flowchart detailing the general process of a search based procedural generation system, adapted from the original diagram by Togelius et al. (2011)

A key consideration in a SBPCG system is the representation of content within the search space. Togelius et al. (2011) use terminology from evolutionary computation to better describe the

concepts around representation. Data structures that are handled by the evolutionary algorithm used to generate content are referred to as genotypes, while the data structures that are evaluated by the fitness function and form the output of the system are known as phenotypes. In terms of real world analogues, the authors compare genotypes to blueprints and phenotypes to the physical buildings produced from them.

Genotypes can be represented in a number of different ways. In the examples given in this paper, some represent the genotype as a vector of real numbers, some as expression trees and as matrices. The genotype could even be represented as a random number seed. The two main factors governing the design of the genotype are dimensionality (the size of the vector) and locality (how much a change in the genotype affects the resulting phenotype). The issue of dimensionality is a complex decision; if the vector is too short it will be incapable of properly representing the phenotype in search space, but if it's too long then it becomes more difficult for search algorithms to handle it. Locality, on the other hand, should be kept as high as possible. That is, a small change in the genotype should ideally result in a small change in the phenotype. As locality decreases, so does the usefulness of the search algorithm, since two genotypes that differ only slightly may produce wildly different phenotypes.

The fitness function, or evaluation function, is another core principal of the SBPCG approach. The authors propose three classes of evaluation function:

1. Direct Evaluation Functions: Features in the content are directly mapped to the fitness value, such as number of exits in a maze or firing rate of a weapon.
2. Simulation-Based Evaluation Functions: The generated content is played with by an artificial agent and the observed gameplay is used to create the fitness value.
3. Interactive Evaluation Functions: Content is scored in real time based on interaction with the player.

Finally, the choice of search/heuristic algorithm (responsible for the "Selection" and "Variation" processes in Figure 4.1) is an important consideration. Most existing examples of SBPCG systems use genetic algorithms, although other heuristic and search algorithms can be used instead. More specifically, this paper proposes simulated annealing and particle swarm optimisation as possible alternatives. These are covered in more detail below.

Genetic Algorithms

In nature, individuals within a population compete for resources and mating opportunities. The most successful individuals survive and mate, passing on their genes to future generations, while weaker individuals die off. Over time, the population evolves to become increasingly more suited to their environment. Genetic algorithms mimic this natural process of "survival of the fittest" in order to evolve solutions to problems (Beasley et al., 1993).

A genetic algorithm maintains a population of solutions to a particular problem. Each solution is represented as a set of parameters called a chromosome, and each parameter in the chromosome is referred to as a gene (Beasley et al., 1993). The set of parameters represented by the chromosome is called the genotype, and the result of using a genotype to solve the problem is called the phenotype (Beasley et al., 1993). Beasley et al. (1993) use the example of building a bridge as the problem. In this example, the genotype could be the dimensions of the beams used in the bridge and the phenotype would be the constructed bridge. As previously mentioned, these terms are used by Togelius et al. (2011) to describe content representation. To evolve solutions, individual solutions are selected from the population and then re-combined (Beasley et al., 1993). Many different selection methods exist, some of which have been summarised below.

Proportionate Roulette Wheel Selection

In this selection method, the probability of selecting a solution is proportional to the solution's fitness value relative to the total fitness value of the population. This is similar in concept to a roulette wheel, where the probability of selecting a sector on a spin is proportional to the central angle of the sector (Shukla et al., 2015). For example, a sector of 40 degrees has an approximately 11% chance of being landed on, while a sector of 90 degrees would have a 25% chance. Applying this to a genetic algorithm, if the population had a total fitness of 20.1 and a particular solution had a fitness value of 0.43 then that solution would have an approximately 2.14% chance of being selected.

Truncation Selection

In truncation selection, individuals are first ordered by their fitness. A threshold representing a fraction of the population is defined and the best individuals within this threshold are randomly

selected from with equal selection probability (Blickle and Thiele, 1995). For example, if the threshold value was 20% and the population contained 100 individuals, the 20 fittest individuals would be chosen and one would be selected from this set at random.

Tournament Selection

A specific number of individuals are chosen from the population at random and the one with the highest fitness out of this set is selected. The number of individuals chosen is known as the tournament size (Shukla et al., 2015).

When two or more individuals (henceforth referred to as the parents) have been selected, the next step is to re-combine them to produce offspring that form the population for the next generation. This is done using several different types of operator. One such type is Crossover (Beasley et al., 1993). A simple example of a crossover operator is single point crossover. In single point crossover, two parents have their chromosomes split at a random point known as the crossover point. The "head" and "tail" sections of each chromosome are then swapped (Beaslet et al., 1993; Umbarkar and Sheth, 2015). An example of single point crossover can be seen in figure 4.2, with chromosomes represented as coloured squares to show how offspring are formed.

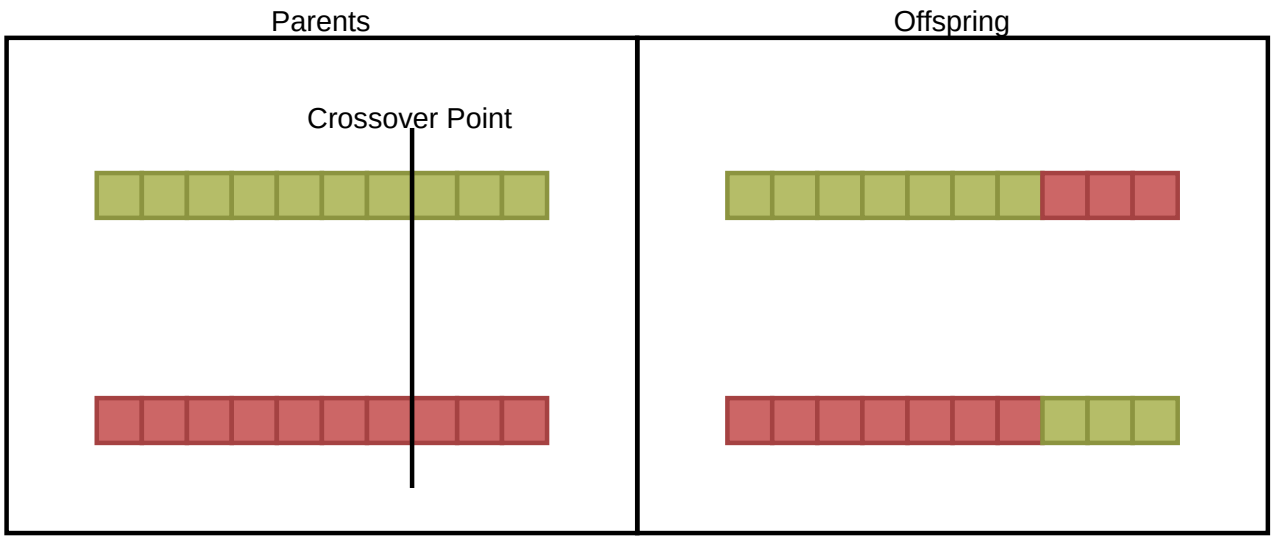

Figure 4.2: A diagram showing an example of how single point crossover can create new offspring

Other types of crossover operators also exist, such as average crossover (where a single child is produced with each gene being an average of the parents' genes) and uniform crossover, which works similarly to single point crossover in that the two offspring are made by swapping sections of the parents. Rather than using a crossover point, however, uniform crossover randomly chooses for each gene whether to swap or not (Umbarkar and Sheth, 2015). Another example of a type of operator that can be applied are mutation operators. The uniform mutation operator, for instance, selects a gene and changes its value to a random value between an upper and lower bound (Soni and Kumar, 2014). Another mutation operator is swap mutation, where two genes are selected at random in a chromosome and their positions are swapped. (Soni and Kumar, 2014).

Simulated Annealing

Annealing is the process in which metal is heated to below its melting point and then gradually cooled, lowering its hardness (Collins English Dictionary, n.d.). Simulated annealing applies this concept to the search space. With each iteration of the simulated annealing algorithm, a new random solution is generated that neighbours the current solution. The cost of this solution is then evaluated and, if it is less than the current solution, the current solution is replaced by it. If, however, the new solution has a higher cost than the current one, the algorithm calculates the acceptance probability to determine if the new solution should be accepted or not (Geltman, 2014). This probability is given by equation 4.1 (Geltman, 2014; Carr, 2018?), where C_{new} and C_{old} represent the "cost" of the new and old solution respectively and where T represents the "temperature". The temperature is a variable that starts at 1 and is then multiplied by a constant (typically between 0.8 and 0.99) to decrease it with each iteration, mimicking the cooling of metal seen in annealing (Geltman, 2014).

$$a = e^{-\frac{C_{new} - C_{old}}{T}} \quad (4.1)$$

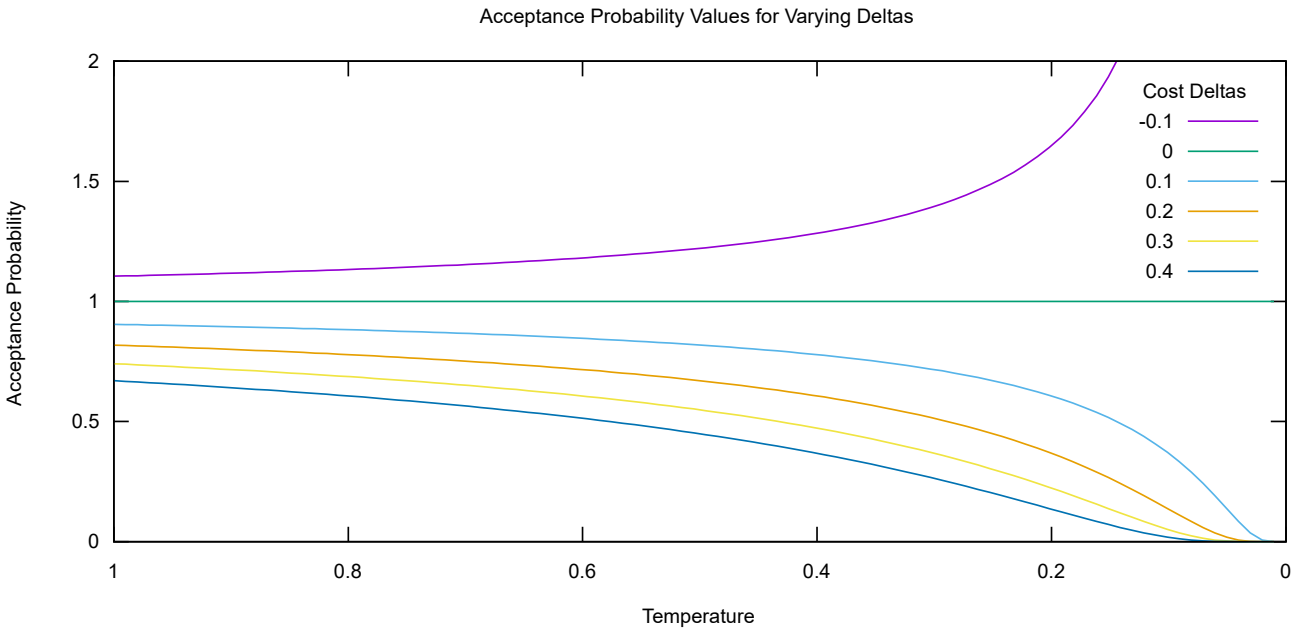

Figure 4.3: A graph demonstrating how different cost deltas create different acceptance probability values with respect to temperature

As seen in figure 4.3, when the new cost is equal to or less than the old cost the acceptance probability will be greater than or equal to 1, in which case the new solution will always be accepted. Otherwise, a random value between 0 and 1 is compared to the acceptance probability and the new solution is accepted if this value is smaller (Carr, 2018?). As the temperature decreases, so too does the probability that worse solutions will be considered; as the algorithm 'cools', it will tend towards a local minimum (Carr, 2018?). Simulated annealing is unlikely to find the optimum solution, but it does have to potential to escape local minima early on and find a good solution regardless of its starting point (Geltman, 2014; Carr, 2018?) While in this example the algorithm attempts to find the cheapest solution, it could also be applied to find the greatest fitness function value, plugging easily into the SBPCG framework.

Particle Swarm Optimisation

Particle swarm optimisation (PSO) is a technique proposed by Kennedy and Eberhart (1995) based on artificial life (A-life) simulations, such as bird flocking and fish schooling. In particular, they based their approach on bird flocking simulations.

PSO shares many concepts with genetic algorithms. For example, PSO maintains a population of solutions, each of which are assigned fitness values, and searches for optimum solutions over many

generations (Hu, 2006). In PSO, solutions are represented by agents which "fly through" an N dimensional search space, where N is the number of parameters (Kennedy and Eberhart, 1995). If the goal was to find, for example, the optimum size of solar panel for a mobile phone charger, the search space might be represented as two dimensional, with the X axis representing the width and the Y axis representing the height of the solar panel respectively. Agents remember their current position in this search space, along with their current velocity, between generations. Agents converge on optimum solutions by moving towards both the position they remember as being the best so far and the position of the fittest agent (Hu, 2006).

$$v[\,] = v[\,] + c1 * rand() * (pbest[\,] - present[\,]) + c2 * rand() * (gbest[\,] - present[\,]) \quad (4.2)$$

The velocity of an agent is calculated using Equation 4.2 (Kennedy and Eberhart, 1995; Hu, 2006). $present[\,]$ is the current position of this agent in the search space. $pbest[\,]$ is the position at which this agent had its highest fitness value, while $gbest[\,]$ is the position at which the highest fitness value across all agents was found (Kennedy and Eberhart, 1995). Kennedy and Eberhart (1995) compared $pbest[\,]$ to "autobiographical memory", the impact on the velocity being that agents tend to return to positions which gave them the highest fitness value. They compare $gbest[\,]$ to "publicized knowledge" or "a group standard" that causes agents to tend to move towards it. $c1$ and $c2$ are called the "learning factors" by Hu (2006) and the "stochastic factor[s]" by Kennedy and Eberhart (1995). These are set at 2 by the latter and cause the agents to "overfly" their targets. This presumably allows for more solutions to be explored rather than agents converging directly on local optima.

The population of the PSO algorithm is first initialised with a number of agents with randomised velocities. Generations are then run in a loop for a specific number of iterations or until a minimum error requirement is met (Hu, 2006). The minimum error requirement refers to the output of the fitness function, specifically the minimum desired fitness value. Hu (2006) gives the example of using PSO to evolve an Artificial Neural Network. In this instance, the fitness value is obtained by feeding a number of patterns into an ANN weighted by the values from the agent and comparing the output to the standard output. The number of misclassified patterns then forms the fitness value of the agent. In this case, Hu (2006) suggests the minimum error value would be one misclassified pattern. When a solution meeting this requirement is found, the PSO algorithm will terminate. In the example of a solar panel from earlier, the fitness function might take in dimensions from the agent

and calculated how many watts of energy this panel would generate. The minimum error could then be set to the desired power. Every generation, the fitness of each agent is calculated and, if this new fitness is better than the agent's $pbest[]$, it is updated (Hu, 2006). The $gbest[]$ position is set to the position of the best fitness value in history of all the particles (Kennedy and Eberhart, 1995). The velocity of each agent is then updated using Equation 4.2 and the position of each agent is updated by adding its current velocity to its position (Hu, 2006).

4.1.2 Design Patterns in FPS Levels

Before advancing further into existing PCG literature, it will be useful to examine the specialist terminology used for describing first person shooter level design. Hullett and Whitehead proposed a common language for describing level-design patterns in first person shooter games in their 2010 paper "Design Patterns in FPS Levels. The pattern definitions in this paper (which is cited by many of the papers examined later in this chapter) shall be used from this point on for the purposes of describing the more unique elements of FPS levels.

Definition 1

The paper describes several level design concepts, which have been summarised below:

- *Pacing: The overall flow of the level resulting from raising or lowering tempo, tension, challenge or difficulty throughout the level.*
- *Tension: The mental strain a game can create in the player as they struggle to survive or complete objectives.*
- *Segmentation: The methods of breaking aspects of the game into smaller elements. Has three different categories.*
    - *Temporal: splitting up the length of time allowed for gameplay*
    - *Spatial: splitting up the game into multiple levels, or splitting up the levels themselves*
    - *Challenge: splitting up the challenges presented to the player*

Definition 2

The design patterns contained in the paper are presented in a condensed form below:

- *Sniper Location: An elevated position overlooking some portion of the level, from which a character can engage others using long range weaponry while still being protected.*
- *Gallery: An elevated area parallel and adjacent to a narrow passageway.*

- *Choke Point: A narrow area with no alternate routes, forcing characters to use it and often exposing them to attack.*
- *Arena: An open area or wide corridor, possibly containing cover.*
- *Stronghold: A confined area with good cover and limited access points. Characters within the stronghold can defend against attackers while remaining protected.*
- *Turret: A special, highly powerful weapon that is fixed in place*
- *Vehicle Section: A form of alternate gameplay where the player drives or rides in a vehicle*
- *Split Level: A corridor with an upper and lower section. Characters can use both sections and switch between them.*
- *Hidden Area: A small room containing a cache of items, usually placed out of the way of the typical route in the level and intended to reward exploration.*
- *Flanking Route: A separate corridor or elements of cover that enable a character to gain a positional advantage in an area with heavy resistance.*

4.1.3 Evolving Interesting Maps for a First Person Shooter

Cardamone et al. (2011) propose an approach to first person shooter level generation that uses the SBPCG technique described in the previous section. This is, according to their paper, the first time that this technique has been used to generate FPS levels and so it forms an ideal starting point to examine. Of key interest is the fitness function and how the genotype and phenotype are modelled.

$$f = -T_f + S \quad (4.3)$$

Cardamone et al. decide on using “fighting time” as the basis for the fitness function, shown in Equation 4.3. That is, the time (in milliseconds) between the player starting to fight an opponent and them dying, represented in equation 4.3 as $-T_f$. Their fitness function also incorporates the free space of the map, represented in equation 4.3 as S. As a result, the fitness function favours maps with longer survival times of the AI agents and with more free cells. The reasoning given for this is to encourage larger maps where there is more space to place weapons and spawn points. A key issue highlighted by the authors here is the difficulty in defining what makes a map interesting, as such a subjective term depends on the preferences of individual players. The phenotype for the

SBPCG system is a 64 by 64 array of cells which can be either a wall or free space and can optionally contain items or a spawn point.

The genotype is where things get interesting. The authors created four different genotypes with four different generation methods, which are summarised below.

Grid

In this representation, the map is divided into a 9 by 9 grid to form 81 cells. Each of these cells has a wall along the top and right face, such that the initial configuration is a grid of walls. The genotype represents each of these cells as an integer between 0 and 3 that dictates the state of the top and right 'walls', demonstrated in Figure 4.4. Figure 4.5 shows one of the maps generated in the experiments conducted by Cardamone et al. (2011) using the Grid method. The genome value of each tile, and the grid on which the map is built, have been overlayed in a light grey. A few assumptions were made when creating this diagram. Firstly, that the border walls on the left and bottom of the map are automatically filled in to "close" the map. Secondly, this diagram assumes that no further processing was performed on the map post-generation; the authors mention that unreachable cells are removed from the map post generation, so the actual genotype of this map may be different from that shown.

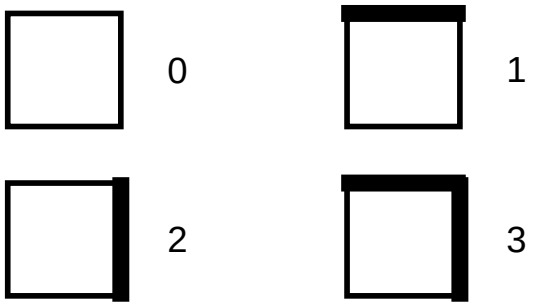

Figure 4.4: The possible genome values for a cell and the corresponding wall configuration

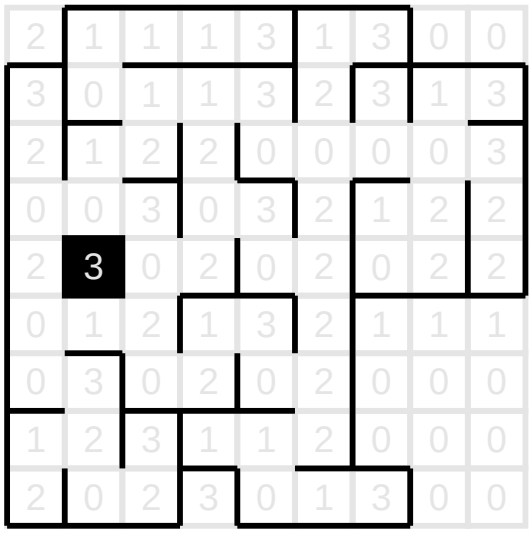

Figure 4.5: One of the maps generated by Cardamone et al. (2011) using the Grid representation. The genome value of each of the larger tiles has been overlayed to demonstrate the encoding.

All-White

This representation encodes the maps less directly, working with the full 64x64 map rather than the 9x9 map of the previous representation. It assumes the map is initially empty except for a border wall. The genotype is a list of walls (the authors use 30), encoded as tuples. The first two values represent the X and Y position of the top-left position of the wall and the third value represents the wall's length. A positive length means the wall is horizontal, while a negative length means the wall is vertical. Examples of some tuples and their corresponding walls are shown in figure 4.6. The origin point of the maps in each of the examples is the top left. One of the maps generated by the authors using this representation can be seen in figure 4.7, with spawn point and resource markings removed so that the map structure is clearer. Black pixels represent walls, while white pixels represent empty space.

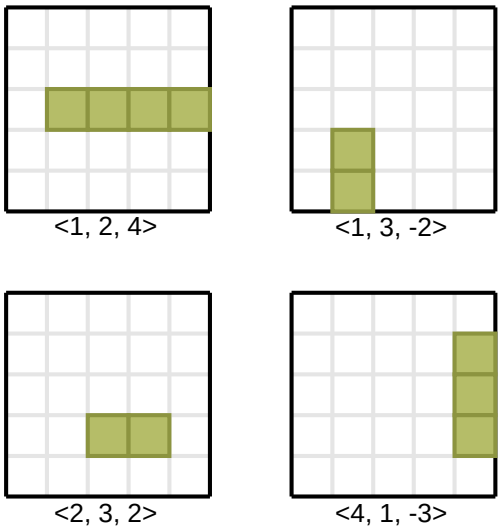

Figure 4.6: Examples of how tuples are converted into walls

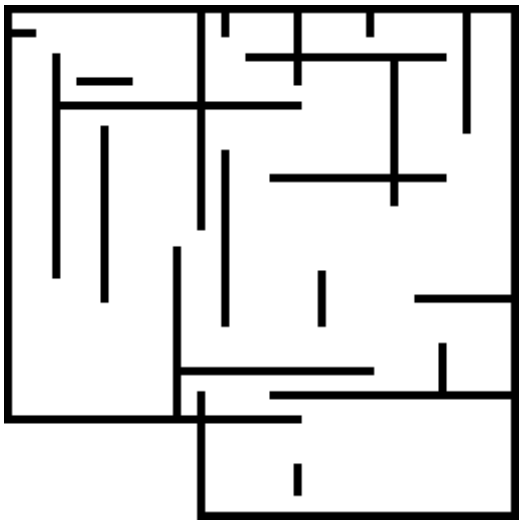

Figure 4.7: One of the All-White maps generated by Cardamone et al. (2011) with resource items and spawn points removed

All-Black

The All-Black representation is similar in concept to the All-White representation, but instead of assuming the map is initially empty, it instead assumes the map is initially entirely ‘wall blocks’. Corridors and arenas are then ‘carved out’ to form empty space. The genotype is once again a list of tuples with a fixed number of corridors and arenas, in this case 30 corridors and 5 arenas. Arenas are encoded with the first two values giving the X and Y co-ordinate of the centre of the arena and the third value giving the size. Since the arenas are squares, the width and height can both be

represented with this single value. Cardamone et al. do not specify how exactly the size is calculated, whether it refers to how much the arena extends out from the centre point or if it refers to the total width/height, from corner to corner. Figure 4.8 shows examples of how both of these encoding methods might work, with the “radius length” examples referring to the length extending from the central tile (highlighted in red) and the “edge length” examples referring to the length representing the total length of an edge. Where the length is even in the latter, it is assumed the length will be biased towards the origin.

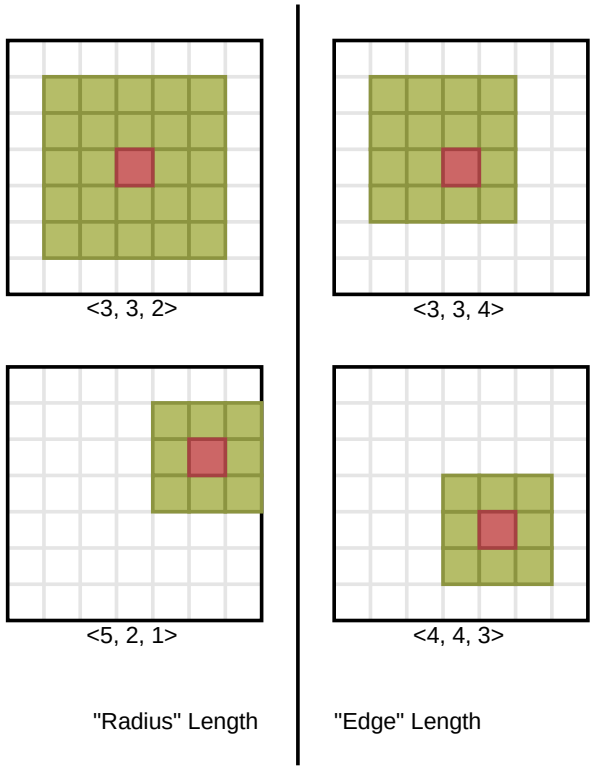

Figure 4.8: Two possible methods of encoding Arenas in the All-Black map representation

Corridors are encoded in the same way as walls in the All-White representation. The only difference is that, while the width of wall in All-White representations is one tile, the width of corridors in All-Black representations is three tiles. Examples of corridors can be seen in figure 4.9.

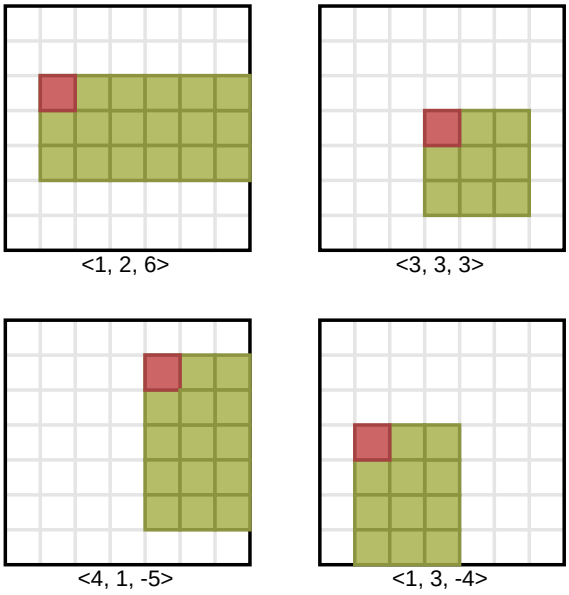

Figure 4.9: Examples of corridors encoded in the All-Black representation and their tuples

Random-Digger

This representation assumes an all wall block starting state like the All-Black representation. The genotype encodes the behaviour of a simple agent that moves around the map and 'carves out' free space as it goes. This is encoded as four probability values representing the probability of going ahead, turning left, turning right and visiting an already visited cell. This agent is initially placed in the middle of the map. From there, it randomly chooses an action based on the probability values, rotating 90 degrees at a time or moving one tile at a time. Each tile the agent visits is made empty. This process is repeated for a fixed number of steps, although this number is not specified by the authors. A map generated by the authors with the spawn points and resource locations removed can be seen in figure 4.10. The authors don't specify exactly how the last probability, the probability that the agent will revisit an already visited cell, is implemented. This could mean that only the first three probabilities are actions that can be taken (i.e. the agent can choose to turn left, turn right or go straight ahead) and the final probability is only used when the agent encounters an already explored tile. When this happens the agent would check to see if it should re-explore the tile and, if not, re-calculate the action for that step. Alternatively, it could mean that there is a chance the agent will "teleport" to a previously explored tile and carry on from there as normal.

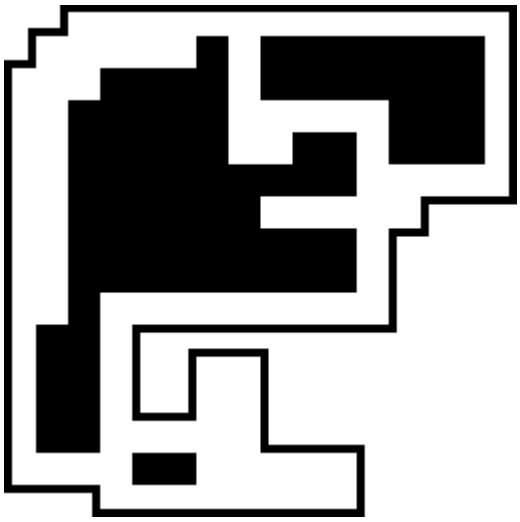

Figure 4.10: One of the Random-Digger maps generated by Cardamone et al. (2011) with resource items and spawn points removed

The authors implemented their system in the Cube 2: Sauerbraten game engine, and used the engine's bots for testing. This involved four bots playing 10 minute matches in the generated maps. The maps themselves were generated using a genetic algorithm with a population of 50 maps and ran for 50 generations. The authors found that, statistically speaking, the All-White representation resulted in the best maps, though they noted that all four representations could generate playable maps. It was also observed that each representation resulted in features that "strongly characterize[d]" the maps.

The research in this paper presents several interesting concepts and ideas that relate to this study. The use of a fitness function that relates to player skill, though admittedly rather loosely, shows the potential for the idea of adapting level design to player ability. The presented map genotypes also provide an ideal proven starting point for further genotype development. An obvious issue with the experiment, pointed out by the authors themselves, is that testing was performed using bots and so the fitness function was dependent on the bot AI rather than real human performance. Justification for this decision was that evolving maps in this way using humans would require them to play several thousand games which would be impractical from a time perspective. This shortcoming can be addressed by this study, as content generation will be occurring constantly as opposed to once per match, giving more potential generations and room to evolve in a shorter space of time.

4.1.4 Evolving maps for match balancing in first person shooters

Lanzi, Loiacono and Stucchi (2014) put forward an approach to procedural map generation for first person shooter levels in which level generation is performed using a genetic algorithm, the fitness function of which is derived from individual player performance. Their work follows a similar approach to the paper examined in the previous section by Cardamone et al. (2011), focusing this time on evolving first person shooter maps with the intention of balancing for "specific combinations of players skills and strategies".

$$f = -\sum_{i=1}^{2} \frac{S_i}{S_1 + S_2} log_2\left(\frac{S_i}{S_1 + S_2}\right) \quad (4.4)$$

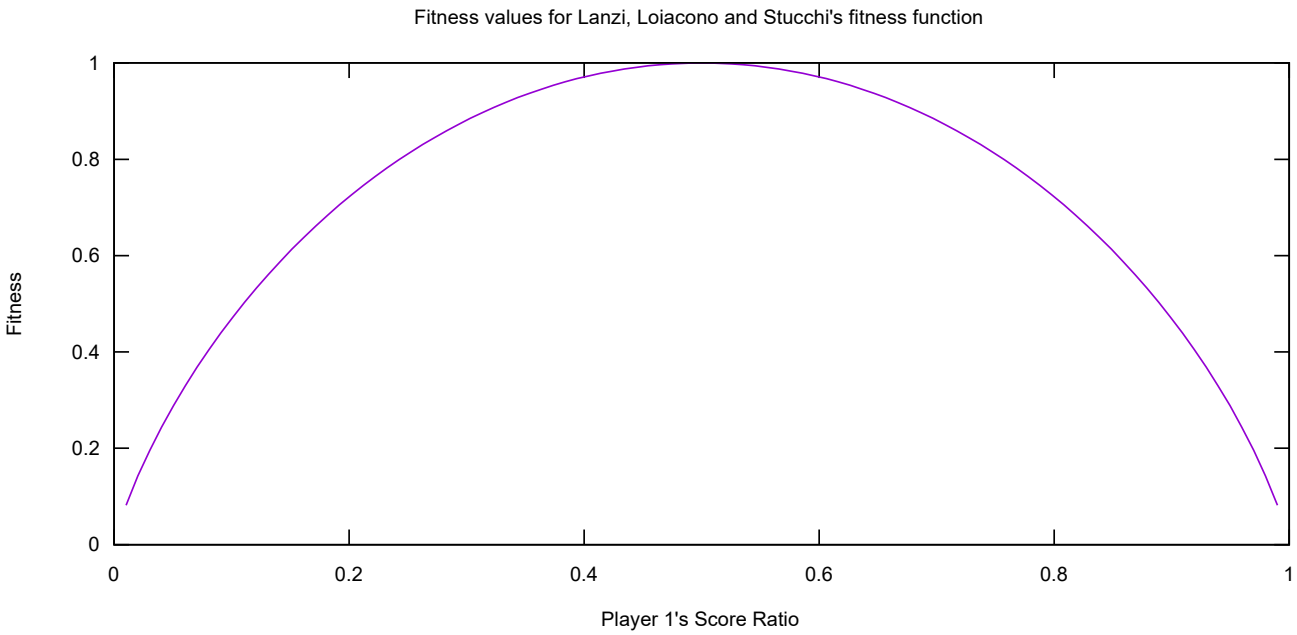

Figure 4.11: Graph showing how the fitness function relates to the player score ratios. Note that the score ratios of the two players are directly related; if player 1's score ratio is 0.2, player 2's will be 0.8 and vice versa.

The fitness function used by Lanzi, Loiacono and Stucchi represents a significant step-up in terms of complexity when compared to the one proposed by Cardamone et al. (2011). In this function, seen in equation 4.4, S_i represents the score of player 'i', which is the number of kills the player made minus the number of times they killed themselves (because of a grenade or falling, for example). This is divided by the total score of both players to give the "score ratio". Figure 4.11

shows the relationship between the score ratio of the first player in a two-player scenario and the fitness value. As can be seen, the fitness function favours maps in which the score ratio of players is equal at 0.5, meaning both players scored equally. In this example, the authors tailored the fitness function and performed the evolution using only two players but make it clear that it could be extended if need be. For the genotype, the authors chose to use the "all-black" representation proposed by Cardamone et al. (2011), discussed in the previous section. Lanzi, Loiacono and Stucchi define a map being "balanced" in this case as having a fitness value greater than around 0.92. In this case, the most skilled player would die at least once for every two kills they made.

Similarly to Cardamone et al. (2011), the maps in this paper are comprised of a 64 by 64 grid of tiles which can either be empty or contain an impassable wall. Empty tiles can contain items and spawn points. In their implementation of the "All-Black" genotype, all arenas are square and all corridors have a fixed width of three squares. Following generation, unreachable empty tiles are removed and spawn points, items and add-ons are placed.

Once again, the Cube 2: Sauerbraten game engine was used for the implementation of the system. The engine's built-in bots were used. Cube 2 allows for bot "skill levels" to be modified from 0 to 101, with a value of 101 making them as skilled as possible (Cube 2 Sauerbraten, 2013?). This feature allowed for matches to be simulated in which the skill of the bots varied. Three experiments were performed, testing not only balancing maps between two bots of different skill levels but also maps where the bots had differing skill levels and weapons. Each run of the genetic algorithm used a population of 100 maps and ran for 30 generations. The fitness value of each map was calculated using the fitness function in equation 4.4, using statistics gathered from a simulated 10 minute match. 10 runs were performed for each experiment. In the first experiment, both bots were armed with rifles with one having a skill of 80 and the other 35. In the second and third experiments, one bot retained the rifle and a skill of 20, while the other bot had a skill of 80 and was armed first with a chainsaw in the second experiment and a grenade launcher in the third.

In the first experiment, the fitness value of the maps rose from around 0.33, to as high as 0.95 (though the final average was 0.85). Further testing using all possible skill value combinations for each bot showed that as the generations progress, balanced outcomes become more likely and unbalanced outcomes less so. Similar results were observed in the second experiment. In this scenario it was found that the bot with the chainsaw had a clear advantage compared to the bot with the rifle, despite the different effective ranges of the weapons. The fitness value rose over the

generations from around 0.3 to a final average of 0.83, though the highest recorded fitness reached 0.96. When testing all of the possible skill combinations the difference between the early and late maps is much more noticeable than in the first experiment. Maps go from a situation in which the rifle player has a score ratio of less than 0.5 until they have a skill of at least 45, to where the rifle player has a 0.5 score ratio from as little as 5 skill. The final experiment presented an interesting scenario in that the rifle is a hit-scan weapon while the grenade launcher is a physics based delayed-hit weapon. The resulting games were almost the opposite of the previous experiment; the bot with the rifle had a clear advantage over the bot with the grenade launcher despite the lower skill. The fitness value rose from around 0.33 to an average of 0.87 with the highest value being 0.97. The skill combination testing revealed a dramatic change from the early to late maps, with the grenade launcher player going from almost always getting a score ratio below 0.2 to a far more balanced situation by the end.

The testing performed in this paper shows the potential for level geometry and its guided evolution to affect the balance of a game. Similarly to Cardamone et al. (2011) the authors elected to use bots for testing purposes to allow for a great many games to be simulated in a short space of time. While this study aims to address this through real-time evolution, consideration will be given to the number of generations required to reach a near-balanced map and how this might be tackled effectively. Another area that requires addressing is that testing was performed with only two players, whereas this study aims to perform testing on many more in a single match. Not only that, but the fitness function shown here is tailored for individual player performance and not the performance of the team. It is therefore unknown how this particular fitness function might scale to the number of players planned for games in this study and will therefore likely need extensive modification should it be used.

4.1.5 Interactive Evolution of Levels for a Competitive Multiplayer FPS

The procedural generation techniques so far have had two major shortcomings in common: generation of levels takes place offline and bots are used for the purposes of evaluation. Peter Thorup Ølsted, Benjamin Ma and Sebastian Risi's paper, "Interactive evolution of levels for a competitive multiplayer FPS" (2015), presents an approach that addresses both. In the paper, they develop a first person shooter in which maps are generated, played and future variations voted on,

all by the players and without gameplay being interrupted, effectively allowing the players to steer the underlying evolutionary search.

While previous papers have been focused on deathmatch levels, where the aim is to get more kills than the opponents, the authors here focused on “bomb defusal” levels, to encourage teamwork and tactical planning. They proposed a new term, “the good engagement”, defined as a level experience “in which the intended strategic and skill-based gameplay shines through” (Ølsted, Ma and Risi, 2015). Following analysis of Counter Strike and Call of Duty maps and gameplay they decided on five rules for maintaining TGE, condensed below:

1. Maps should contain choke points, reachable by both teams at around the same time
2. Bomb locations should be closer to the defenders
3. Bomb and spawn points should never be visible from each other
4. Levels should not contain too many paths, especially four way intersections
5. Spawn points should be at opposite ends of the level

Unlike the previous papers, no fitness function was used in Ølsted, Ma and Risi’s procedural generation. While SBPCG was still employed, player voting was used for the evaluation step. Following a game, six variations of the map just played are presented to players, who vote positively or negatively on each. The map with the most votes is then carried forward and six potential “bomb position” layouts are generated, which are voted for in much the same way as the layout. In both cases, a player can also vote to regenerate the variations (Bunjabin, 2015).

Maps are encoded as variable length lists of “L” shaped corridors, which are represented by a pair of 2D co-ordinates giving the start and end points of the corridor and a variable specifying if the horizontal or vertical line should be placed first. All L shapes are snapped to a grid to ensure connectivity and the size and resolution of this grid are both included in the genotype. Once the L shapes are placed the algorithm then removes dead ends and changes any four way intersections into three way intersections, repeating this process until neither dead ends nor four way intersections remain. Corridors and rooms are then carved out of impassable tiles in a similar manner to the “all-black” map representation (Cardamone et al., 2011) and props and doorways are placed. Spawn points are placed as closely as possible to their encoded positions and bomb sites are chosen based on their distance from the spawns, ensuring the defenders are able to reach them faster. A filter is also applied to generated levels that rejects levels that don’t fit the rules of TGE,

such as levels with too long corridors, spawn points that are immediately visible to each other and spawn points that can't reach each other.

The testing present in this paper focused more on how direct player input shaped map generation and so will not be covered in great detail, due to lack of relevance to this study. One area of interest in the testing however is that the genotype and map representation were capable of effectively creating very different maps.

By far the most interesting part of this paper, at least in the context of this study, is the map genotype and generation process. Through a combination of genetic techniques and "post-processing" filtering and placement of bomb sites, a finer level of control is possible compared to the potential chaos of a purely genetic algorithm based PCG system. For instance, using this technique it can be assured that maps will not be generated with a huge bias towards one team or that areas of the map are impossible to reach.

4.2 AI Directors

A similar concept to the proposal in this study was explored in the game "Left 4 Dead 2" (Valve Corporation, 2009). Left 4 Dead 2 and its predecessor Left 4 Dead (Valve Corporation, 2008) make use of multiple systems to create the "AI Director", intended to promote replayability by adjusting the game's pacing on the fly (Booth, 2009a; 2009b). This is done by estimating the "emotional intensity" of players, reducing the number of threats when the intensity is too high and increasing the number of threats when it is too low (Booth, 2009a; 2009b). The population and placements of enemies are procedurally generated, as are the placement of weapons and items (Booth, 2009a; 2009b). Left 4 Dead 2 also experimented with randomised terrain. In-game developer commentary also reveals that Valve experimented with dynamic level modification in the "Park" and "Cemetery" levels (Valve Corporation, 2009). While the procedural path system was abandoned for the park level, in the cemetery level the AI director has the ability to create one of four different paths for the players by changing the arrangement of gates and crypts (Valve Corporation, 2009).

While not exactly what this study is trying to achieve, Left 4 Dead 2 does represent an example of level modification by an AI in a multiplayer game. Whether or not the AI modifies the level for the purpose of affecting pacing as it does with the placement of enemies and pickups isn't clear but is

definitely a possibility. It also raises a potential problem that will need to be considered when developing the software for this study; playtesters for the “Park” level apparently found the dynamic paths confusing (Valve Corporation, 2009). Care must be taken to make the procedural level generation used in this study as unobtrusive and well integrated as possible to avoid causing confusion.

4.3 Metrics

Another core element of the intended research project of this thesis is the ability to use quantifiable metrics to evaluate both player performance and the characteristics of generated levels, in order to allow the AI director to adjust level geometry in a targeted and effective manner. This section examines research into generating metrics for both of these areas.

4.3.1 Towards a Generic Method of Evaluating Game Levels

Liapis, Yannakakis and Togelius (2013) put forth formulas intended to measure the prevalence of three key game design patterns within procedurally generated levels: area control, exploration and balance. These are intended to be as generic and high level as possible so that they can be applied to a variety of genres and domains. In order to provide this level of abstraction, the algorithms are developed for so called “map sketches”, simple grid-based layouts made up of tiles. These tiles can be passable, impassable or “special”, the latter representing domain specific features such as spawn points or traps.

In this paper, the term balance is used in place of symmetry and refers to “[ensuring] players have equal opportunities”. Area control refers to the control of areas and strategic resources. In this case, the former are the passable tiles of the map sketch and the latter are the special tiles.

$$S_{t,i}(S_N) = \min_{\substack{1 \le j \le N \\ j \neq i}} \left\{ max\left\{0, \frac{d_{t,j} - d_{t,i}}{d_{t,j} + d_{t,i}}\right\}\right\} \quad (4.5)$$

To calculate area control and exploration values, two sets are defined: reference tiles (special tiles such as player bases or spawn points), denoted as S_N, and target tiles (strategic resources, e.g. resources in a RTS game or weapons in a FPS), denoted as S_M. A reference tile is said to "own" a tile if it is closer to them than all other reference tiles. This is represented using a safety value, calculated using Equation 4.5. In this equation $S_{t,i}$ refers to the safety value between any tile t and reference tile i. $d_{t,j}$ refers to the distance between the tile t and the current reference tile j while $d_{t,i}$ refers to the distance between the tile t and the reference tile i. The closest reference tile to the tile t will have a positive safety value, while all others will have a safety value of 0. The paper does not specify how the distance between tiles is calculated, though it can be assumed methods such as A* can be employed due to the simplistic nature of the map sketches.

$$E_i(S_N) = \frac{1}{N-1} \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{E_{i \to j}}{P} \quad (4.6)$$

The exploration value of a reference tile is defined as how large an area of the map must be covered to discover one reference tile, starting from another. Equation 4.6 gives the exploration value E_i required to get from reference tile i to all other reference tiles. $E_{i \to j}$ is the map coverage, in number of tiles, when a four-way flood fill is performed on the map, starting at reference tile i and stopping once reference tile j has been reached. P is the total number of passable tiles.

The safety function and the area control function shown in equations 4.5 and 4.6 are used as part of three functions to calculate the strategic resource control, area control and exploration of maps, shown in Equations 4.7, 4.8 and 4.9 respectively. A_i is the map coverage of safe tiles for element i. For a tile t to be considered safe relative to an element i, $S_{t,i}$ should be less than a defined constant. The authors use a constant of 0.35 for this paper.

$$f_s(S_N, S_M) = \frac{1}{M} \sum_{k=1}^{M} \underset{1 \leq i \leq N}{max} \{S_{k,i}\} \quad (4.7)$$

$$f_a(S_N)=\frac{1}{P}\sum_{i=1}^{N} A_i \quad (4.8)$$

$$f_e(S_N)=\frac{1}{N}\sum_{i=1}^{N} E_i \quad (4.9)$$

As the authors point out, these three functions average or aggregate over reference tiles. If there is a significant number, it is conceivable that one player may have a strategic resource very close to them yet the strategic resource control value may appear balanced, for example. To remedy this, three further functions are proposed that deal directly with balance, shown in Equations 4.10, 4.11 and 4.12.

$$b_s(S_N,S_M)=1-\frac{1}{MN(N-1)}\sum_{k=1}^{M}\sum_{i=1}^{N}\sum_{\substack{j=1\\ j\neq i}}^{N}\left|S_{k,i}-S_{k,j}\right| \quad (4.10)$$

$$b_a(S_N)=1=\frac{1}{N(N-1)}\sum_{i=1}^{N}\sum_{\substack{j=1\\ j\neq i}}^{N}\frac{\left|A_i-A_j\right|}{max\{A_i,A_j\}} \quad (4.11)$$

$$b_e(S_N)=1-\frac{1}{N(N-1)}\sum_{i=1}^{N}\sum_{\substack{j=1\\ j\neq i}}^{N}\frac{\left|E_i-E_j\right|}{max\{E_i,E_j\}} \quad (4.12)$$

The functions proposed in this paper form an ideal basis for the map evaluation functions in this study, and could easily be implemented and expanded upon in order to evaluate generated FPS maps which, in turn, could be used to steer the procedural generation towards a balanced state. While the examples given in this paper relate to real time strategy games and roguelike dungeons, the authors point out that the functions are high level and generic which should facilitate their use in this study.

5 Research Methodology

Before beginning the design and implementation process, the methods of data collection, processing and analysis must be identified to develop a sound research approach. This will, in turn, guide the development of the game itself.

5.1 Research Strategy

After examining the quantitative approaches employed by Lanzi et al. (2014) and Cardamone et al. (2011), it was decided to also use a quantitative approach for this study in order to identifying balance in games. A game will be developed and used as a test platform to gather data through several experiments, supporting and contrasting this data with player opinion through a multiple choice questionnaire.

5.1.1 Game Mechanics

Gameplay will be based on the mechanics of the “King of the Hill” game-mode from Team Fortress 2 (Team Fortress Wiki, 2017). While many games implement a “King of the Hill” gamemode, Team Fortress 2 was chosen as the base for the capture mechanics due to the authors familiarity with the game and the age of the gamemode. “King of the Hill” has been a part of the game since 2009, with several updates since then adding further maps for the gamemode (Team Fortress Wiki, 2017) making it a relatively mature game mode and therefore a seemingly solid, polished base of mechanics to use. In this game-mode, two teams of players fight over a single “control point” in the map. This point begins in a neutral state but can be captured by either team, at which point their capture timer will begin to count down. Each team must hold the point for a total of three minutes. If the opposing team captures the point, the team that previously held it will have their timer pause at the point it was captured. Team Fortress 2 also implements additional rules regarding captures, such as overtime (Team Fortress Wiki, 2017). For the purposes of this study, such extra features were deemed overly complex and would add extra variables with no perceived benefit. The general mechanics of the single control point with a three minute capture timer for each team will be retained. However, in place of the overtime mechanic and to ensure games do not last forever if a capture is not made, a limit of 10 minutes for each game will be enforced. If 10 minutes expire, the

team with the least time remaining will be declared the winner. Another mechanic that will be borrowed from TF2 is the mechanic wherein the time it takes to capture the control point is dependent upon how many players from the same team are on it (Team Fortress Wiki, 2018), which should encourage and reward team play in the testing sessions.

Of the papers examined in the literature review, only Ølsted et al. (2015) used teams in their gameplay. In their testing, a variety of player numbers were used, from as few as 4 to as many as 13. To make analysis and comparison of results easier, it was decided to use fixed team sizes for this study. Following some early enquiries and advertising to gauge interest in the experiment, a team size of 3 players with 2 teams per game was decided on. This is partly an arbitrary value, though it was also chosen due to concerns that larger team sizes may cause difficulties reliably securing participants, based on the aforementioned gauge of interest.

In order to further reduce the number of variables that could affect the team and game balance, only a single weapon will be implemented. This will be a semi-automatic raycast-based weapon to remove the potential impact of physics and bullet travel time. Reloading will not be a mechanic, with the weapon drawing from an infinite pool of ammunition. Another reduction to simplify gameplay mechanics will be the lack of any method of healing. Player health will be replenished only upon re-spawning after death.

5.2 Data Generation Methods

Data will be generated in two ways. Firstly, gameplay data such as kills, captures and the map layout will be recorded automatically by the game itself. To support this data, users will also be required to fill out a short multiple-choice questionnaire following each match.

The data gathered automatically will include:

- **Teams** – The composition of teams will need to be recorded in order to assess team skill and performance.
- **Maps** – In procedural games, recording how the map changes will allow visualisation of the terrain advantages or disadvantages afforded to each team in the analysis.
- **Kills** – Kills will provide a useful metric when measuring player (and by extension team) skill.

- **Shots** – By recording the shots fired during the game it will be possible to create heatmaps to analyse areas of high activity in the maps.
- **Captures** – Since capturing the control point will be the main victory condition of the game, it will be important to store when the point is captured and by whom to allow later analysis.

These questions concern the game you have just played. For each question, please tick one box for the answer you wish to pick. There are no right or wrong answers to any of these questions, they are merely your opinion, so please answer honestly. If at any point you are unsure of the meaning of a question please ask the experiment supervisor.

Question 1: Which team had better players overall?

My team ☐ Neither, both teams were evenly matched ☐ The opposing team ☐

Question 2: Which team did the level layout tend to favour more?

My team ☐ Neither team ☐ The opposing team ☐

Question 3: Do you feel that the two teams were evenly matched, taking into account the player skill distribution and any handicaps or benefits provided to a team by the level layout?

Yes ☐ Unsure ☐ No ☐

Figure 5.1: The questionnaire presented to each player following the completion of a match

The design of the questionnaire is shown in Figure 5.1. Multiple choice questions were chosen to make the answers discrete and therefore easily analysed using statistical techniques. This in turn will make comparing and contrasting player opinion with hard data from gameplay a much easier and more reliable task as opposed to allowing open ended questions.

5.3 Data Analysis

In order to determine how balanced a game is, it is necessary to define a quantifiable measure of balance. In regards to the previously described game mechanics, it is reasonable to assume that a team that is significantly stronger than the other will capture the point very quickly and have their

capture uncontested for the duration of the game. Conversely, should the teams be relatively equal in terms of skill, it would be expected that the control point would trade ownership many times per game and that the game would last comparatively longer as a result. Thus the two main areas of gameplay likely to be affected by the skills of the teams are the duration of the game and the remaining time on each teams capture timer at the end. A formula to provide a balance value for a given game was developed, shown in Equation 5.1. T_g represents the duration of the game, while T_r and T_b represent the time remaining for the red and blue teams' capture timers respectively at the end of the game.

$$B = \frac{\left(\frac{T_g - 180}{420}\right) + \left(1 - \frac{|T_r - T_b|}{180}\right)}{2} \quad (5.1)$$

The resulting balance value ranges from 0, indicating an unbalanced game, to 1, representing a game in which the two teams performed equally. The balance value is calculated using the time remaining in the game as a percentage of the total time in addition to the delta of the two teams' capture timers. The former is calculated by first subtracting 180 from the duration of the game, then dividing the resulting value by 420. Subtracting 180 from the duration and the divisor eliminates the capture time from this part of the equation. Without this, a game in which the capture point was captured very quickly and then remained uncontested would appear to have a relatively high balance value. For example, suppose a game takes place in which the red team capture the control point after just 5 seconds and hold it uncontested until the end of the game. In this case T_g would be 185, T_r would be 0 and T_b would be 180. If 180 were not subtracted from T_g and the divisor was 600 instead of 420, the balance value would be 0.154. With the subtraction, the balance value would instead be 0.00595. The latter value far more accurately represents the imbalanced nature of this game. The second part of the dividend of the equation is calculated by dividing the absolute value of the difference between the time remaining for the red team timer and the time remaining for the blue team timer by 180, resulting in a positive value between 0 and 1 representing the percentage difference between the two teams' remaining time. This is then subtracted from 1 such that the closer the two teams' capture timers are to each other, the higher the value of the second part of the dividend will be. By dividing the dividend by 2, the resulting value will be between 0 and 1 instead of 0 and 2. In this way, both the game time remaining and the capture time delta are equally as important in determining balance. For the purposes of this study, a game is considered balanced if the balance value is greater than or equal to 0.5. This could be caused by a

number of scenarios, but the simplest is that a game lasting the full 10 minutes will be guaranteed a balance value of at least 0.5. Similarly, a game in which both teams have equal or nearly equal capture times remaining will also have a balance value close to 0.5.

In order to better evaluate map balance, it will be necessary to have a numerical measure of the team balance, allowing the skill disparity between the teams to be compared and contrasted with the map balance. For this purpose, a formula was developed based on the concept of 'score ratios' used by Lanzi et al. (2014). Unlike in the experiments performed by Lanzi et al. (2014) there will be no methods in which a player can kill themselves in this study. The formula, shown in Equation 5.2, is therefore based solely on the score of each team. K_r represents the number of kills made by the red team, K_b represents the number of kills made by the blue team and K_t represents the total number of kills made in the game. T is the team balance value, ranging from 0 to 1. In a similar manner to the balance value equation, a value of 0 indicates a match in which the teams had vastly different levels of skill, while a value of 1 indicates the teams were evenly matched as far as ability is concerned. For the purposes of this study, the teams are considered balanced if the team balance value is greater than or equal to 0.67. This is based on the definition of a game being balanced proposed Lanzi et al. (2014) where the most skilled player will die once for every two kills they make. Applied to the team score ratio function, a value of 0.67 indicates that the more skilled team had two kills for every one kill made by the weaker team.

$$T = 1 - \frac{|K_r - K_b|}{K_t} \quad (5.2)$$

The key difference between the balance value and the team balance value are that the former represents the balance in practice while the latter represents the balance in theory. As an example, if the team balance value was 0.32 it would indicate that the teams are relatively imbalanced in terms of player skill. One team performed much better than the other at an individual level by achieving more kills and thus a higher score than the opposing team. If the balance value is then a low number, such as 0.21, it would indicate that the outcome of the game was also imbalanced and (if the map was procedural for this game) that the attempted methods of balancing using terrain modification were ineffective. If, however, the balance value was relatively high (0.86, for example) it would indicate that although the team skill distribution wasn't balanced the outcome of the game

was and (if the map was once again procedural) that the attempt to balance the game using terrain modification was successful.

5.4 Experiment Structure

Each testing session will consist of four matches. Two of the these will be “Control” games in which the map will remain in a static configuration. These games will provide a baseline for game and team balance for the teams. The other two will be “Procedural” games in which the map will start in the same configuration as the control games, but will change throughout in response to team performance. Participants will first play a control game, then a procedural game, then another control and procedural game in the same order. The intention behind staggering the games is to allow players a chance to become adjusted to the mechanics. If both control games were played first, then the procedural games (or vice versa) players may have improved their skills through practice by the final games, which could impact the results.

Because the focus of this study is on balancing teams, rather than players, participants will be allowed to take part in multiple sessions if they wish. However, no two team compositions will ever be duplicated and will instead be randomised every session and consistent for the games within it.

5.5 Sampling

For this study, a combination of convenience sampling and snowball sampling (Trochim, 2006) will be used. Advertising will be open to any and all volunteers and take place through social media and other public communication channels. Volunteers will also be asked to inform contacts they know may be interested about the study.

5.6 Ethics

Participants will be made aware before consenting to participate what data will be collected and how it will be used. Participants will also be made fully aware of the purpose of the study, the aims and intentions of the research and the purpose of the testing they are about to participate in.

Data gathered will be associated with participant information, such that participants can be linked to their data, for a short period of time following testing to allow participants the right to withdraw after which all data will be anonymised before analysis.

5.7 Limitations

The purely quantitative approach this study will take towards testing means that detailed player feedback will not be possible to measure and analyse. While games can be evaluated for balance and the discrete data from the questionnaires used to support the findings, questions cannot be answered about other aspects of this balancing method, such as if it is fun, if the stronger team felt challenged etc.

Another limitation is that only one method of balancing is being tested. While the effectiveness of this study's proposed balancing method can be compared to control games to determine its effectiveness, it may be that this balancing method confers less benefits and is less effective than other methods such as reducing the damage done by the stronger team, for example.

6 Development Methodology

For this project, a Personal Extreme Programming approach was adopted. Personal Extreme Programming, or PXP, is an adaptation of Extreme Programming tailored for individual programmers rather than teams (Agarwal and Umphress, 2008). PXP is an agile methodology that allows for rapid development and a great deal of flexibility when system requirements change mid-development. Such an approach therefore lent itself well to the implementation of this project, given the relatively tight time allowance and the potential for development to change focus and concepts as a result of new research or insights. The PXP process script proposed in Agarwal and Umphress' (2008) paper was used as a starting point for the planning process.

6.1 System Metaphor

While the system metaphor is typically used to aid communication and understanding between developers and customers, a function not required for this project given the solitary nature of the development, it also helps to "[guide] the mental models that project members have of the system and [shape] a logical architecture for the system" (Khaled et al., 2004?). The system metaphor for this project is that the system is an architect laying tiles within a fixed area, which represents the map. The architect, representing the map system, places and removes tiles to form the map. Certain tiles may have additional "height" to them but from the perspective of the architect what is being dealt with is a two dimensional area of tiles.

6.2 User Stories and Features

The following user stories were created, then broken down into features in order to more clearly define the necessary steps to realise the system.

User Story	Necessary Features

"As a player, I should be able to connect to a server so that I can take part in a team-based multiplayer game"	• Client/server architecture
	• Match system
	  ◦ Timer
	  ◦ Scoreboard
	• Team system
	• King of the hill gamemode

"As a player, I want to be presented with a familiar first person shooter so I can quickly begin playing."	• 3D environment • GUI • Mouse & keyboard controls • Weapons • Health/Damage
"As a player, I want to see the map layout change during gameplay in response to team skill to attempt to balance the game"	• Voxel-based map system • Dynamic map alteration • AI balancing system • Player/team performance evaluator
"As a researcher, I should be able to view collected data and produce graphs or visual aids so I can report on findings."	• Database for saving statistics • Data logging functionality • Data visualisation/export tools

“As a researcher, I need a way of overseeing games in progress.”	• Admin client

6.3 Feature Sets

The following feature sets were formed from the user stories and arranged in order of priority:

Feature Set	Features	Acceptance Tests	Estimated Time to Implement
Base Client/Server Architecture	• Application that can launch as a client or server • Client that can connect to a server using an IP address and port • Server that can specify a port to host on • Initial admin client	• The regular and admin clients should be able to connect to a server both on the local network and over the internet	< 1 day
Base gameplay features	• Simple 3D environment • Mouse and keyboard controls • Match system • Database • Data logging functionality for logging players	• Clients should be able to move around the environment • Clients should be able to connect to a server and see the movement of other clients • The server should run matches for a specified amount of time, then restart the match when the timer runs out • The server should save players to the database with a unique player ID for each device	2 days

Completion of gameplay features	• Weapons • Health/Damage • GUI • King of the hill gamemode • Team system • Logging of shots, hits, kills etc.	• Clients should be able to shoot at and kill each other in-game • Clients should be able to view their health and score • Clients should be able to capture a static capture point • Clients should be randomly assigned teams and be incapable of damaging teammates • The server should save shots, hits and kills made by players to the database	4 days
Base dynamic map features	• Voxel-based map system • Dynamic map alteration	• The server should maintain a voxel map that is synchronised to the clients • The server should be able to modify the map and have the modifications be present on the clients	2 days
Completion of dynamic map features	• Player/team performance evaluator • AI balancing system	• The server should be capable of evaluating team performance • The server should adjust the map layout to provide an advantage to the	2 days

		weaker team	
Implementation of analysis tools	• Data visualisation/export tools	• The export tool should be able to export the database data in formats usable by software such as gnuplot for the purposes of analysis.	1 day

6.4 Iteration Plan

Iteration #	Feature Set / Refactor	Planned Start Date	Planned Duration
1	Base client/server architecture	17/01/2018	< 1 day
2	Base gameplay features	18/01/2018	2 days
3	Completion of gameplay features	22/01/2018	4 days
4	Refactor	26/01/2018	1 day
5	Base dynamic map features	29/01/2018	2 days
6	Completion of dynamic map features	31/01/2018	2 days
7	Implementation of analysis tools	02/02/2018	1 day
8	Refactor	05/02/2018	1 day
9	Final testing	06/02/2018	3 days

7 Technical Implementation

The core game was built using the “Merry Fragmas” (Unity Technologies, 2018) Unity3D tutorial series in the Unity3D game engine as a foundation. It was then extended and modified in order to incorporate the procedural map system, database logging, teams and other features. As a result this section will not cover *all* of the code in the software, but instead focus on what was added to the

“Merry Fragmas” base in order to facilitate the reactive level geometry mechanic required of this study.

The following external libraries were used in the development of this project:

- The Genetic Algorithm Framework (GAF) 2.3.1 (Newcombe, 2016a) – Used for all genetic algorithm related functionality within the program.
- Json.NET (Newtonsoft, 2018) – Used to serialise/deserialise objects for storage in the database, such as shot vectors or map chromosomes.
- System.Data.SQLite (SQLite, 2018?a) – Used by the data export tool to interface with the SQLite database

7.1 Map Controller

The MapController script is one of the most significant additions to the core gameplay code and is responsible for the generation of new maps and synchronising the current map with the clients. This is accomplished using a custom SyncList of GeneTuples which stores the current genes to be used for the map. Whenever this list is changed on the server, the changes are also made on the clients, thanks to Unity3D’s networking framework. The level geometry is stored as a three dimensional byte array. A value of 0 indicates empty space, while values above 0 map to an enum describing different terrain types, allowing multiple different textures for different terrain.

```
void Start() {
  set chunksInitialised to false;
  initialise the mapData array;
  initialise the mapChunks list;
  instantiate the chunks;
  set the capturePoint variable;
  initialise the spawn positions lists;

  if (isServer) {
    update the GeneticAlgorithmHelpers static variables;

    for (int i = 0; i < 31; ++i) {
      add a new gene to the currentGenes list;
    }

    modify the genes in the currentGenes list to create the default map;
    call DelayedInitialMapUpdate() after 2 seconds;
    unpause the game timer;
    create a new chromosome from the currentGenes list;
    set the currentMapChromosome variable to the new chromosome;
    set mapUpdateNeeded to true;
  }
}
```

Figure 7.1: Pseudocode for the Start function

The Start function, seen in Figure 7.1, contains the initialisation code for the MapController script. The main initialisations performed here are the mapData array, which stores the level geometry, and the mapChunks list, which stores a list of references to the MapChunkControllers which themselves are created by a call to InstantiateChunks. This calculates the number of chunks required to contain the map and then instantiates them, adding them to the mapChunks lists for later reference. After the capturePoint script is assigned, using the CapturePointController component on the capture point prefab, the spawn point lists are initialised. These lists are used to store all possible spawn points for each team.

With the shared initialisation done, the function then moves on to server specific initialisation. The mapSketchWidth, mapSketchHeight and timeToCapture static variables in the GeneticAlgorithmHelpers class are updated and the currentGenes list is populated with 31 blank GeneTuples. Several of these tuples are manually modified in order to create the default map, seen in Figure 7.2. The player spawn positions are shown in red and blue for team 1 and 2 respectively and the capture point is shown in purple. Black tiles are impassable, while white tiles are passable. The design of the default map incorporates certain elements of TGE (Ølsted, Ma and Risi, 2015) appropriate to this style of gameplay; the spawn points are located at the opposite ends of the map, and care was taken to ensure the spawn points are not visible to each other or from the control point in the centre. The central arena was left purposefully large and open with the intention of creating a control map in which player skill is the determining factor in capturing the point. In the absence of

any walls or cover of any kind, players will be forced to use mobility to avoid fire while returning fire themselves. After the default map is constructed, the Start function invokes a call to DelayedInitialMapUpdate that will execute after two seconds. This delay is intended to counteract situations where clients may be on a slow connection and join later than expected. Finally, the currentMapChromosome variable is updated and the mapUpdateNeeded flag is set to true.

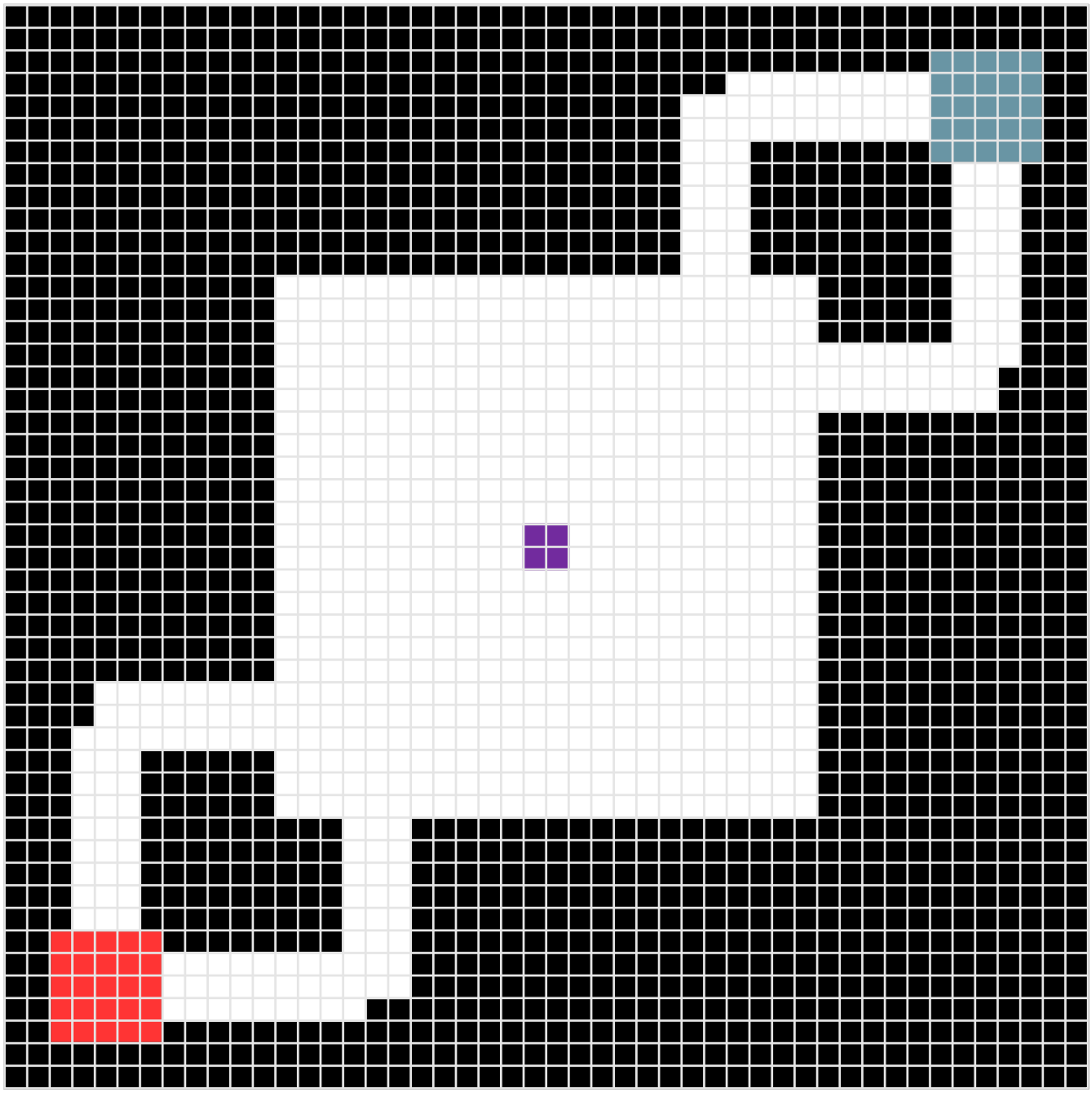

Figure 7.2: The default map layout

```
void Update() {
  if (isServer and mapUpdateNeeded) {
    update the synced gene list;
    update the map on the clients;
    set mapUpdateNeeded to false;
    store the map in the database;

    if this is a dedicated server {
      update the map with the current genes;
    }

    if the game isn't finished and this a procedural game {
      Generate a new world after 15 seconds;
    } else if the game isn't finished and this is a control game {
      Force a map update after 15 seconds;
    }
  }
}
```

Figure 7.3: Pseudocode for the Update function

The Update function of the MapController, seen in Figure 7.3, is called every frame and simply waits for when map generation is finished and the map needs to be updated. No code is run client side but on the server side, when a new map is ready, the SyncList is updated so that all clients have the newest genes locally. To prevent excessive updates if individual genes are modified, and to ensure synchronisation across clients and the server, the map mesh itself isn't updated using these genes until the server makes a remote procedure call to RpcUpdateMap on the clients. The genes are then serialised using Json.NET and this string is saved to the database using the DatabaseManager singleton. If the game is being run in "host and play" mode, the remote procedure call will be executed on the machine acting as the server since it is also a client. However, a dedicated server needs to manually update the map. The final step of the update loop is to call either the GenerateWorld function or the ForceMapUpdate function after 15 seconds. The former will generate a new world before updating the map, while the latter simply sets the mapUpdateNeeded variable to true when called. ForceMapUpdate was added after early testing identified an issue wherein clients might not properly receive the synchronised map genes on the first run. In a procedural game this wasn't a problem as the map would be updated on the second Update loop but in control games this could lock players out of the game, permanently stuck in a loading state. 15 seconds was chosen to prevent the map from updating too frequently and disorienting players, while also being short enough to make a reasonable impact on gameplay as opposed to, for example, a period of 2 minutes. This would result in the map changing only 5 times during a game, which was seen as far too infrequently. A period of 15 seconds results in approximately 40 map iterations during the game.

```
void GenerateWorld() {
  create a new population of map chromosomes;
  populate the population with 100 copies of the current map chromosome;
  add the elite operator;
  add the crossover operator;
  add the mutation operator;
  create the genetic algorithm instance and assign callback functions and operators;
  update the time variables in the GeneticAlgorithmHelpers class;
  start an asynchronous run of the genetic algorithm;
}
```

Figure 7.4: Pseudocode for the GenerateWorld function

World generation is handled by a genetic algorithm, which is initialised and started by the GenerateWorld function. Shown in Figure 7.4, this firstly creates a population of 100 copies of the current map chromosome, which will be evolved using the genetic algorithm. By populating the GA with copies of the current map as opposed to, for example, a blank map chromosome, the map that results from running the GA will be an evolution of the current map which is intended to lead to smoother terrain transitions for the players. Three operators are created for the GA: an elite operator, a crossover operator and a mutation operator. The elite operator uses a value of 5%, while the crossover operator uses a probability of 0.85 and double point crossover. These values were taken from the “Getting Started” page on the GAF wiki (Newcombe, 2016b). The mutation operator is a custom operator written to mutate map chromosomes and is covered here in a later section. The probability of mutation is 0.04, a value once again taken from the “Getting Started” page (Newcombe, 2016b).

```
static void ga_OnRunComplete() {
  get the fittest map chromosme from the population;
  set currentMapChromosome to the fittest map chromosome;
  set mapUdateNeeded to true;
}
```

Figure 7.5: Pseudocode for the ga_OnRunComplete callback function

The callback function passed in to the genetic algorithm, shown in Figure 7.5, is relatively simple. It takes the fittest map chromosome from the population generated by the GA and assigns it to the currentMapChromosome variable. It then sets the mapUpdateNeeded flag to true, which will trigger a map update the next time the Update function is called.

```
void UpdateMapWithCurrentGenes() {
  if (!chunksInitialised) {
    call UpdateMapWithCurrentGenes after 1 second;
  } else {
    create a new chromosome from the genes in currentGenes;
    set currentMapChromosome to this new chromosome;
    convert currentMapChromosome into a new map sketch;
    remove unreachable tiles from the map sketch;
    set currentMapSketch to this new map sketch;
    call UpdateMapWithMapSketch(currentMapSketch);
  }
}
```

Figure 7.6: Pseudocode for the UpdateMapWithCurrentGenes function

UpdateMapWithCurrentGenes is the function responsible for converting the synchronised list of gene tuples into a map sketch, which can then be used by the UpdateMapWithMapSketch to update the map data itself. Seen in Figure 7.6, it does this by first creating a new chromosome from the currentGenes list, which it then converts into a map sketch using a static helper method in the MapSketchHelpers class, which will be covered later. Unreachable tiles are then removed. This prevents players from being trapped in closed off areas and unable to participate in the game. It then calls the UpdateMapWithMapSketch function using the newly created map sketch as the argument. In case this method is called on clients before the map chunks have been instantiated, this function first checks to see if the chunksInitialised flag has been set. If not, it will repeatedly call itself once every second until they have, at which point it will run as normal.

```
void UpdateMapWithMapSketch(TileType[,] mapSketch) {
  calculate mapSketchWidth and mapSketchHeight;
  initialise the spawn point lists;

  for (int curX = 0; curX < mapSketchWidth; ++curX) {
    for (int curY = 0; curY < mapSketchHeight; ++curY) {
      switch (mapSketch[curX, curY]) {
        case Impassable:
          create a wall in mapData;
        case Passable:
          create a floor in mapData;
        case Team1Spawn:
          create a floor in mapData;
          add a spawn point to the red team list;
        case Team2Spawn:
          create a floor in mapData;
          add a spawn point to the blue team list;
        case Barrier:
          create a barrier in map data;
        case CapturePoint:
          create a floor in mapData;
        default:
          create a wall in mapData;
      }
    }
  }

  update the capture point after the map preview finishes;
  update the spawn positions;
  call GenerateMesh();
}
```

Figure 7.7: Pseudocode for the UpdateMapWithMapSketch function

UpdateMapWithMapSketch, seen in Figure 7.7, is called to update the mapData array using a provided map sketch. It iterates through each tile in the map sketch, updating the mapData array according to the type of tile. Two temporary lists containing the locations of the spawn points for each team are also updated during this process as spawn tiles are found. Once the mapData has been updated, the UpdateCapturePoint function is called after a delay determined by the previewTime variable. This variable specifies how long the new map mesh should “fade in” before becoming permanent, giving players time to prepare for the new layout. By delaying the control point update by the same amount, it makes certain that the control point will be moved at the same time as the new map geometry becomes fixed. UpdateCapturePoint itself updates the position of the capture point using the values contained in the capture point gene. The permanent spawn point lists are updated with the temporary lists populated earlier and the GenerateMesh function is called, which simply calls the GenerateMesh function on each of the map chunks.

```
void MovePlayerToNearestSafeTile(Player player) {
  initialise safeTiles list;
  get player position in map-sketch-space;

  for (int x = 0; x < mapDimensions.x / 2; ++x) {
    for (int y = 0; y < mapDimensions.z / 2; ++y) {
      if this tile isn't impassable {
        add the tile to safeTiles;
      }
    }
  }

  sort safeTiles by their distance to the player;
  convert the first tile in safeTiles to a vector3;
  warp the player to this vector;
}
```

Figure 7.8: Pseudocode for the MovePlayerToNearestSafeTile function

Figure 7.8 shows the MovePlayerToNearestSafeTile function, which was created to overcome a perceived problem with the nature of the procedural map. Since it is possible for an arena a player is in to suddenly disappear, leaving them trapped in a solid wall, this function can be called to warp a selected player to the nearest "safe" tile, defined in this case as any tile that isn't impassable. This is done by converting the player position to a co-ordinate in the map sketch, then iterating through the current map sketch to form a list of all safe tiles. These are then sorted by distance to the player using a simple "as the crow flies" distance calculation. The first tile in the list can then be taken as the closest tile to the player. This is converted from a two dimensional mapSketch co-ordinate to a Vector3 game world position, which is passed as an argument to the RpcWarpToPosition function on the player object, which warps the player to the new safe position.

```
bool CheckPlayerInSafeTile(Player player) {
  if (isServer) {
    calculate the player position in map sketch space;
    return true if the player is not in an impassable map tile;
  }

  return true;
}
```

Figure 7.9: Pseudocode for the CheckPlayerInSafeTile function

CheckPlayerInSafeTile, seen in Figure 7.9, is a simple function called from the MapChunkController instances to determine if a specific player is in a safe tile or not. Since this code is only needed on the server, which then handles moving the players if any are in unsafe tiles as discussed above, if this code is called on a client it will simply return true, indicating the player is safe and avoiding wasting time on unnecessary calculations. Otherwise, it will convert the player position into a map sketch co-ordinate and return true if the tile the player is in is not impassable.

7.1.1 Genotype

Maps use the “all-black” system proposed by Cardamone et al. (2011). The genotype, shown in Figure 7.10, encodes a map sketch (Liapis et al., 2013) and is formed of 31 tuples containing three integer values, referred to as X, Y and Z. A custom tuple struct called ‘GeneTuple’ was created owing to incompatibility between Unity3D’s networking code and the default system tuple object. The GeneTuple struct has three public properties: X, Y and Z. It also defines two operator overrides for “!=” and “==” to allow comparisons. 15 genes correspond to arenas, 15 to corridors and one to the capture point. The number of arenas and corridors were taken from the parameters used in Luca Lanzi, Loiacono and Stucchi’s paper (2014). The origin of each map sketch’s co-ordinate system is the bottom left.

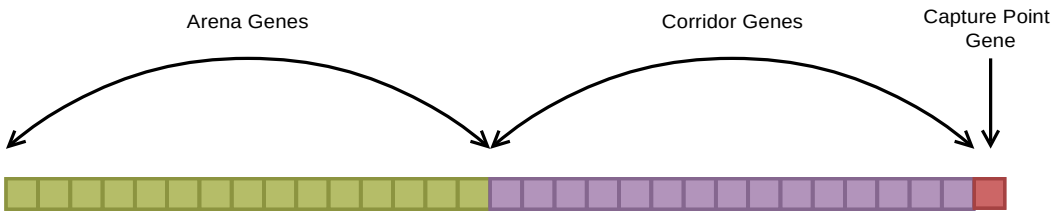

Figure 7.10: The map genotype.

Arenas are encoded such that X and Y store the X and Y co-ordinates of the bottom left corner of the arena. The Z value of the tuple stores the size of the arena, which represents both the width and the height. The capture point is encoded in exactly the same way. Corridors are encoded such that the X and Y values correspond to the centre tile of the start of the corridor. Each corridor has a fixed width of three tiles and can either be horizontal or vertical depending on the sign of their Z value, which is the length of the corridor. A positive length results in a horizontal corridor and a negative length results in a vertical corridor. Figures 7.11 and 7.12 show examples of both arena and corridor tuples. In these examples, impassable tiles are white whilst arena/corridor tiles are light green. The X and Y value of each example has been highlighted in red to show the difference between arenas and corridors as far as the origin points are concerned.

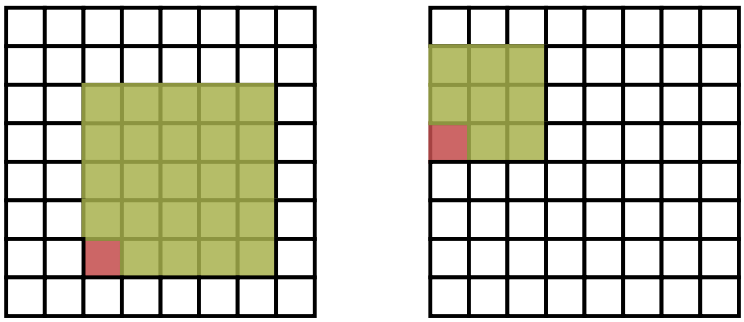

Figure 7.11: Two examples of how arena tuples translate into a map sketch.

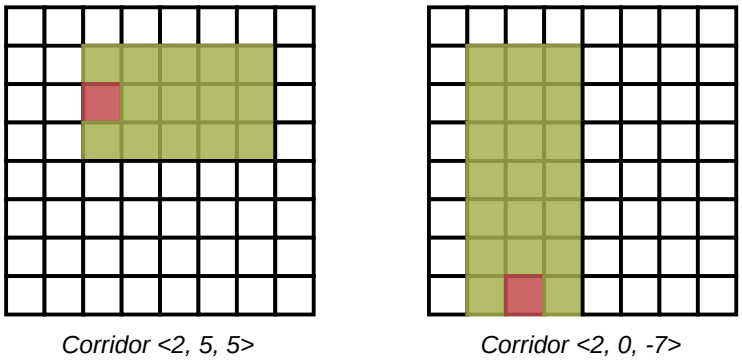

Figure 7.12: Two examples of how corridor tuples translate into a map sketch.

7.1.2 Phenotype

The map data for the in-game map itself is stored in the mapData array. This is a three dimensional byte array in which each element represents a 1m x 1m x 1m cube, referred to as a block. A value of 0 indicates empty space, while any number above that represents a particular type of block, which is used only for changing texture and has no direct impact on gameplay. Since the player measures 2 metres in height and 1 metre in diameter (excluding the players weapon, which extends outwards some distance), the map sketches are converted not directly into blocks but rather so called “large blocks”, which are formed of 8 blocks. When used for walls and floors these large blocks allow ample room for players to fit in compared to the single blocks.

As previously mentioned, the chromosomes encode map sketches (Liapis et al., 2013). These are two dimensional arrays of the TileType enum. The TileType enum has 5 possible values representing 5 different tile types. Two of these are taken directly from Liapis et al. (2013):

- Impassable – The tile cannot be traversed by players
- Passable – The tile can be traversed by players

Where Liapis et al. (2013) defined a third type, special tiles, this has been split into three different types, forming the last three types of the TileType enum:

- CapturePoint – Capture point tiles
- Team1Spawn – Tiles forming the spawn points for the red team
- Team2Spawn – Tiles forming the spawn points for the blue team

The chromosomes are responsible for creating Impassable, Passable and CapturePoint tiles, but Team1Spawn and Team2Spawn tiles are hard coded into every mapsketch as part of a post-processing step to ensure spawn points are always present at the same location. It was decided not to encode the spawn points as part of the genotype in order to reduce the number of variables affecting terrain advantages. By keeping spawn points static, it is only the layout of arenas and corridors and the position of the capture point that can be changed to attempt to impact the balance of the game. Figure 7.13 shows the different tile types and how they appear in game, albeit in a simplified form.

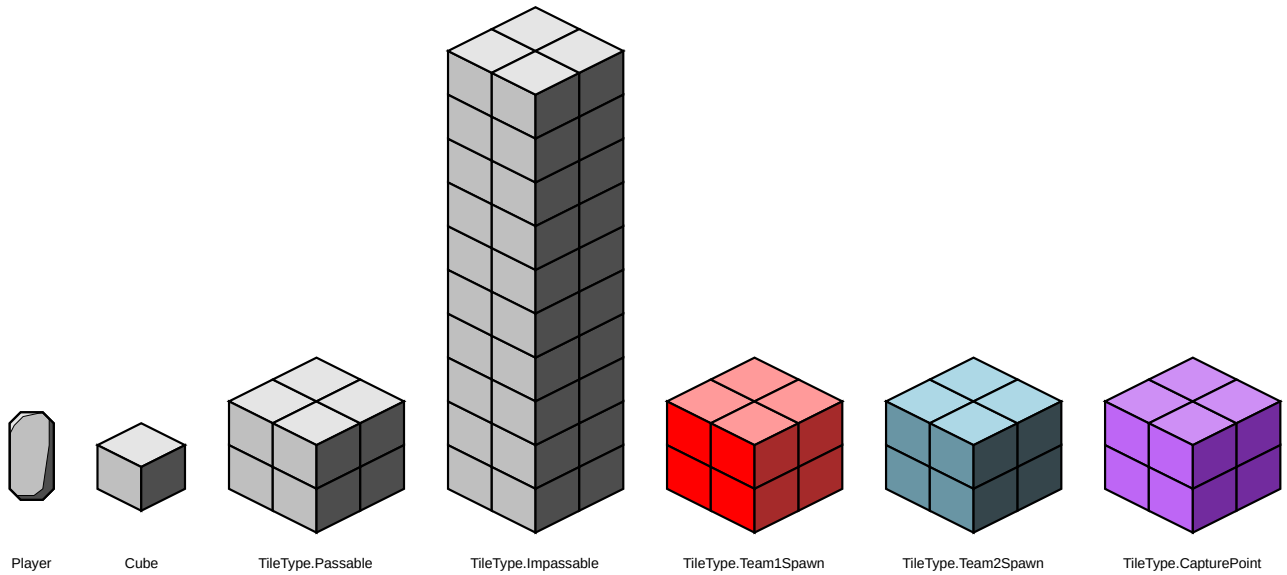

Figure 7.13: Diagram showing the different tile types and their corresponding representations using blocks, along with a rough guide to their size in relation to the player.

7.2 Map Chunk Controller

For performance purposes the map is subdivided into chunks, each of which is a seperate GameObject with an attached MapChunkController script. For this application these chunks measure 16 x 16 x 16 blocks, though these values can be modified prior to running the application. The mesh rendering system was developed using tutorials written by Alexandros Stavrinou (2015), so only the modifications made to Stavrinou's code will be covered in this section. The main change to the original code is the addition of a second mesh to each map chunk, the "preview mesh", which is used to show players upcoming map changes to avoid disorientation. Unlike the regular mesh, this preview mesh has no collision mesh associated with it and uses a transparent material to allow it to slowly fade in.

```
void GenerateMesh(float previewTime, bool isPreview = true) {
  for (int x = chunkX; x < chunkX + chunkWidth; ++x) {
    for (int y = chunkY; y < chunkY + chunkHeight; ++y) {
      for (int z = chunkZ; z < chunkZ + chunkLength; ++z) {
        update the newVertices, newTriangles and newUV lists using the mapData;
      }
    }
  }

  call UpdateMesh(isPreview);
  set timeUntilRegenerateMesh = previewTime;
  set currentTimeUntilRegenerateMesh = previewTime;
}
```

Figure 7.14: Pseudocode for the GenerateMesh function

The GenerateMesh function, shown in Figure 7.14, remains largely unchanged from Stavinrou's (2015) implementation, with the exception of two new arguments specifying the previewTime, the time it should take for the preview mesh to fully "fade in", and a boolean specifying if the preview mesh or the regular mesh should be generated. The isPreview variable is passed into the UpdateMesh call as a new argument. Two variables are also updated here, which are used for fading in the preview mesh.

```
void UpdateMesh(bool isPreview) {
  if (isPreview) {
    clear the preview mesh;
    update the vertices, UVs and triangles of the preview mesh;
    recalculate the preview mesh's normals;
  } else {
    call MovePlayersToNearestSafeTiles;

    clear the regular mesh;
    update the vertices, UVs and triangles of the regular mesh;
    recalculate the regular mesh's normals;

    create the collision mesh;
    clear the preview mesh;
  }

  clear the vertex, UV and triangle buffers;
  set the face count = 0;
}
```

Figure 7.15: Pseudocode for the UpdateMesh function

Seen in Figure 7.15, the UpdateMesh function now takes a boolean argument similar to GenerateMesh that specifies if the preview mesh or regular mesh should be updated. If isPreview is set to true, the function clears the preview mesh then updates it using the vertex, UV and triangle buffers before recalculating the preview mesh's normals. If isPreview is set to false, the function first calls the MovePlayersToNearestSafeTiles method. This ensures that all players will be moved out of the way of the new regular mesh should any of them be in a position where they would be "inside" of the terrain. The function then clears, updates and recalculates the normals of the regular mesh as normal, then creates the collision mesh and clears the preview mesh. Regardless of the state of the isPreview argument, the function will clear the buffers and reset the face count as usual.

```
void Update() {
  if (chunkUpdated) {
    call GenerateMesh(0, false);
    set chunkUpdated = false;
  }

  if (currentTimeUntilRegenerateMesh > 0) {
    subtract Time.deltaTime from currentTimeUntilRegenerateMesh;
    update the preview mesh's material colour's alpha to 1 - currentTimeUntilRegenerateMesh /
      timeUntilRegenerateMesh;

    if (currentTimeUntilRegenerateMesh <= 0) {
      set currentTimeUntilRegenerateMesh = 0;
      call GenerateMesh(0, false);
    }
  }
}
```

Figure 7.16: Pseudocode for the Update function

The Update function, shown in Figure 7.16, retains the ability to automatically update the chunk by setting the chunkUpdated variable. To achieve this with the new preview mesh functionality, the

call to GenerateMesh must specify 0 for the preview time and false for the isPreview argument, which will result in an instantaneous update of the regular mesh. The more notable change is the addition of the preview timer variables, initialised in the GenerateMesh function. If the currentTimeUntilRegenerateMesh variable is greater than 0, it will subtract the delta time each update loop, causing it to act as a count down timer. Every time the timer updates it will also update the alpha colour of the material on the preview mesh to 1 minus the current time until the mesh regenerates divided by the original value of the timer. This has the effect of causing the alpha value to slowly increase from 0 to 1 as the timer approaches 0, causing the preview mesh to appear to "fade in". When the timer reaches 0 or goes past it it is reset to 0 and GenerateMesh is called, once again with arguments of 0 and false to immediately update the regular mesh.

```
void MovePlayersToNearestSafeTiles() {
  foreach (player in Player.players) {
    calculate the player position in map sketch space;

    if (this chunk contains the player) {
      if (mapController.CheckPlayerInSafeTile(player) == false) {
        call mapController.MovePlayerToNearestSafeTile(player);
      }
    }
  }
}
```

Figure 7.17: Pseudocode for the MovePlayersToNearestSafeTiles function

MovePlayersToNearestSafeTiles, seen in Figure 7.17, iterates through each player in the game (accessible from the static players list contained in the Player class) and calculates their position in the map sketch. If the player is contained within the chunk calling this function, the chunk then makes a call to the MapController's CheckPlayerInSafeTile function, with the current player as the argument. Map chunks, unlike the map controller, are not networked instances and hence the check performed to see if the code is executing on the server is contained in the CheckPlayerInSafeTile function. As a result of this, the MovePlayerToNearestSafeTile function will only be called if the map chunk is on the server and the player is in an unsafe tile.

7.3 Capture Point Controller

The CapturePointController script is attached to the capture point gameobject, which it moves and resizes in accordance with the capture point gene in the chromosome. It contains a method to update the position and dimensions of the point, which is called by the map controller.

```
public void OnTriggerEnter(Collider other) {
  if (isServer) {
    if (other is a player and this player is not in the playersInCaptureZone list) {
      add this player to playersInCaptureZone;
    }
  } else {
    if (other is the local player and the mesh is not flipped) {
      flip the capture point mesh;
    }
  }
}
```

Figure 7.18: Pseudocode for the OnTriggerEnter function

```
public void OnTriggerExit(Collider other) {
  if (isServer) {
    if (other is a player and this player is in the playersInCaptureZone list) {
      remove this player from playersInCaptureZone;
    }
  } else {
    if (other is the local player and the mesh is flipped) {
      un-flip the capture point mesh;
    }
  }
}
```

Figure 7.19: Pseudocode for the OnTriggerExit function

As the capture point gameobject has an attached collision mesh, it makes use of the OnTriggerEnter and OnTriggerExit functions, seen in Figures 7.18 and 7.19. These functions are called when an object with a collision mesh attached enters and exits the collision mesh on the capture point. The functions are used on the server to maintain a list of players currently in the capture point area. On the client, these functions are used to flip the capture point mesh for the local player while they are inside it. If this was not done, the capture point would be invisible to players inside it. The mesh is flipped with the FlipMesh function that iterates through each triangle in the mesh and flips them by swapping the first and last indices.

```
void Update() {
  if (isServer and the game timer isn't paused) {
    set redTeamCount = the number of alive red team players in playersInCaptureZone;
    set blueTeamCount = the number of alive blue team players in playersInCaptureZone;

    if (redTeamCount > 0 and blueTeamCount == 0) {
      set captureChangeAmount = Clamp(redTeamCount between 0 and MaxBonus) *
        CapturePercentagePerSecond * Time.deltaTime;

      if (blueTeamCapturePercentage > 0) {
        if (blueTeamCapturePercentage – captureChangeAmount > 0) {
          set blueTeamCapturePercentage -= captureChangeAmount;
        } else {
          set remainder = captureChangeAmount – blueTeamCapturePercentage;
          set blueTeamCapturePercentage = 0;
          set redTeamCapturePercentage += remainder;

          set currentControllingTeam = Team.Random;
          pause the blue team capture timer;
          add the capture to the database;
        }
      } else {
        if (redTeamCapturePercentage < 1.0) {
          set redTeamCapturePercentage = Clamp(redTeamCapturePercentage + captureChangeAmount
            between 0 and 1);

          if (redTeamCapturePercentage == 1.0) {
            set currentControllingTeam = Team.Red;
            un-pause the red team capture timer;
            add the capture to the database;
          }
        }
      }
    } else if (blueTeamCount > 0 and redTeamCount == 0) {
      set captureChangeAmount = Clamp(blueTeamCount between 0 and MaxBonus) *
        CapturePercentagePerSecond * Time.deltaTime;

      if (redTeamCapturePercentage > 0) {
        if (redTeamCapturePercentage – captureChangeAmount > 0) {
          set redTeamCapturePercentage -= captureChangeAmount;
        } else {
          set remainder = captureChangeAmount – redTeamCapturePercentage;
          set redTeamCapturePercentage = 0;
          set blueTeamCapturePercentage += remainder;

          set currentControllingTeam = Team.Random;
          pause the red team capture timer;
          add the capture to the database;
        }
      } else {
        if (blueTeamCapturePercentage < 1.0) {
          set blueTeamCapturePercentage = Clamp(blueTeamCapturePercentage + captureChangeAmount
            between 0 and 1);

          if (blueTeamCapturePercentage == 1.0) {
            set currentControllingTeam = Team.Blue;
            un-pause the blue team capture timer;
            add the capture to the database;
          }
        }
      }
    } else {
      if (currentControllingTeam == Team.Red) {
        set percentageChange = CapturePercentageRecoveryPerSecond * Time.deltaTime;

        if (redTeamCapturePercentage < 1.0) {
          set redTeamCapturePercentage = Clamp(redTeamCapturePercentage + percentageChange
            between 0 and 1);
        }
      } else if (currentControllingTeam == Team.Blue) {
        set percentageChange = CapturePercentageRecoveryPerSecond * Time.deltaTime;

        if (blueTeamCapturePercentage < 1.0) {
          set blueTeamCapturePercentage = Clamp(blueTeamCapturePercentage + percentageChange
            between 0 and 1);
        }
```

```
      } else {
        set percentageChange = CapturePercentageRecoveryPerSecond * Time.deltaTime;
        if (redTeamCapturePercentage > 0) {
          set redTeamCapturePercentage = Clamp(redTeamCapturePercentage – percentageChange
            between 0 and 1);
        } else if (blueTeamCapturePercentage > 0) {
          set blueTeamCapturePercentage = Clamp(blueTeamCapturePercentage – percentageChange
            between 0 and 1);
        }
      }
    }
  }
}
```

Figure 7.20: Pseudocode for the Update function

The important logic for the capture point is contained within the Update function, seen in Figure 7.20. All logic is executed on the server and the state of the capture point is relayed to clients using syncvars. The number of players for both teams in the capture point are first counted, checking that the players are also alive. This is because when players die, they are only disabled and not moved until being respawned. This extra check prevents dead players from capturing and/or contesting the point. If both teams have players on the point, or no-one is on it, it will attempt to reset the capture percentage to a resting state determined by the team that currently owns it. For instance, if the capture point is owned by the red team and nobody is currently on it, if the capture percentage is not entirely red then it will slowly return to 100% red capture percentage at a rate determined by the CaptureRecoveryPerSecond variable. For this study, this variable was set to 0.02, roughly one third the capture rate.

If, however, the capture point has entirely one team on it, that team begins to capture it. If the opposing team owns the point then opposing team's capture percentage will be reduced to 0 before the capturing team's capture percentage begins to increase. For this study, the CapturePercentagePerSecond variable was set to 0.0625 and the max bonus was set to 3, fulfilling the requirements laid out in the research methodology of a point that takes 16 seconds to capture and applies bonus speed for each extra player up to 3.

The Team enum seen here has three possible values: Red, Blue and Random. Random is named as such because the enum was first used for team selection in the lobby. In this case, it represents a neutral team or lack of ownership in the case of the capture point.

7.4 Player

The existing player class from the Merry Fragmas tutorial (Unity Technologies, 2018), along with several related classes, had to be modified to support team play since the original tutorial was designed as a free-for-all arena shooter. To allow the player to be warped to the nearest safe tile from the MapController, a simple remote procedure call was added that takes a Vector3 argument and sets this as the player's position. Several new syncvars were added to store the players database id, their team database id, their team, if they're alive and if they've answered the post game questionnaire.

```
private void Start() {
  set mainCamera = Camera.main.gameObject;

  set initialised = !isLocalPlayer;
  if (isLocalPlayer) {
    DisablePlayer();
  } else {
    EnablePlayer();
  }

  if (!isServer) {
    add player to scoreboard;
  }

  update the player's display name and team;
}
```

Figure 7.21: Pseudocode for the Start function

Little has been added to the Start function, seen in Figure 7.21, but enough to warrant a longer explanation. The initialised flag is set here, which is used later for the purposes of spawning. In addition, the player is now only enabled if this is a remote player. The local player remains disabled to the client until they respawn. This change was made due to initial issues with the respawning system and it may no longer be necessary, though further testing would be needed to be sure. The scoreboard is populated on the clients here as well, a simple new addition to allow players to see their performance relative to the other players in-game. Finally, the callback functions to update the player's name label and model colour are called manually here, as the callback hooks were not being called when the syncvars were initialised.

```
private void Update() {
  if (!initialised) {
    if (mapController == null) {
      set mapControllerObject = Find GameObject called “Map”;
      if (the map object was found) {
        mapController = mapControllerObject.GetComponent<MapController>();
      } else {
        set mapControllerObject = Find GameObject with the “Map” tag;
        if (the map object was found) {
          mapController = mapControllerObject.GetComponent<MapController>();
        }
      }
    } else {
      if (the map controller has spawn positions initialised for this player’s team) {
        set initialised = true;
        call Respawn() after mapController.previewTime seconds;
      }
    }
  }

  if (isServer && GameOver) {
    if (all players have answered the questionnaire) {
      return all players back to the lobby;
    }
  }
}
```

Figure 7.22: Pseudocode for the Update function

When the player is spawned, the script initially has no reference to the MapController. As a result, the update loop, shown in Figure 7.22, was changed to search for the MapController in the scene if the initialised variable is not set. It searches by both name and tag and, when the game object is found, updates the mapController variable so that it can then be accessed to spawn the player on the next game tick. The initialised variable is then set to true and a call is made to invoke the Respawn function after a period of seconds defined by the MapController’s previewTime, such that the player spawns after the map preview has finished. The other addition is a small segment of server side code to check that all players have completed the questionnaire at the end of the game. When they have, the existing BackToLobby function is called.

```
private void Respawn() {
  if (the game isn’t paused) {
    if (isLocalPlayer) {
      disable the loading screen on the player’s UI;
      set transform.position = mapController.GetSpawnPositionForTeam(playerTeam);
      set transform.rotation = 0 in the y axis if their team is red, 180 in the y axis if their
        team is blue;
    }

    EnablePlayer();
  } else {
    call Respawn() again after 1 second;
  }
}
```

Figure 7.23: Pseudocode for the Respawn function

The Respawn method, seen in Figure 7.23, was modified to use the MapController's spawn points instead of the original NetworkStartPositions. When Respawn is called, if the game isn't paused then the player will be enabled on the local client, remotes and server. For the local player, their loading screen is disabled if it wasn't already and the player is moved to the location of a spawn point randomly selected from the available positions by the mapController. Red team players are rotated to face 'north', roughly towards the blue team, while the latter are rotated to face 'south'.

```
public void SubmitAnswers(string answers) {
  if (isLocalPlayer) {
    call CmdSubmitAnswers(answers);
  }
}
```

Figure 7.24: Pseudocode for the SubmitAnswers function

```
private void CmdSubmitAnswers(string answers) {
  add the answers to the database for the player;
  set HasAnsweredQuestions = true;
}
```

Figure 7.25: Pseudocode for the CmdSubmitAnswers function

Figures 7.24 and 7.25 show the two functions responsible for submitting questionnaire answers to the server. SubmitAnswers is called on the client by the QuestionsController, which then calls the CmdSubmitAnswers function on the server from the local player. CmdSubmitAnswers stores the answers in the database for the submitting player and sets the player's HasAnsweredQuestions property to true.

7.4.1 Player Health

Two small changes were made to the player health script. Firstly, to support the scoreboard that was added to the UI, Players now have a Death variable that is incremented in the TakeDamage function when the player dies. Secondly, a Start function was added that performs the same function as the existing OnEnable function, setting the health value to the maxHealth value. Initial testing revealed that the health was occasionally not being initialised properly and adding this rectified this issue.

7.4.2 Player Shooting

The biggest change to the PlayerShooting script is that code related to the weapon, such as cooldown, damage etc. was moved into a separate weapon class, theoretically paving the way for switchable weapons, though this feature wasn't used. Aside from that, additional code has been added to only call the TakeDamage function on a player's health if that player is on the opposite team. Since the victory condition is now triggered by the GameTimeManager class and is unrelated to kills, the OnEnable function was removed so the players score is now retained for the duration of the game. Finally, database logging was added whenever a shot is fired or a player is killed.

7.5 Question Controller

The QuestionController script is used to handle the questionnaire displayed to players at the end of a match. It is attached to a gameobject containing the question text and toggle groups for the answers.

```
public void Update() {
  if (all three toggle groups have an answer selected) {
    set allQuestionsAnswered = true;
    set the SubmitButton to be interactable;
  }
}
```

Figure 7.26: Pseudocode for the Update function.

The update function, seen in Figure 7.26, checks to see if all three questions have been answered. If they have, the allQuestionsAnswered variable is set to true and the SubmitButton is set to be interactable, allowing the questions to be submitted for this player.

```
public void SubmitAnswers() {
  if (allQuestionsAnswered) {
    set answers = string.Empty;

    set answers += the name of the active toggle in the first question's toggle groups + ",";
    set answers += the name of the active toggle in the second question's toggle groups + ",";
    set answers += the name of the active toggle in the third question's toggle groups;

    call localPlayer.SubmitAnswers(answers);
    disable the questions panel;
  }
}
```

Figure 7.27: Pseudocode for the SubmitAnswers function

Figure 7.27 shows the SubmitAnswers button, which is called when the SubmitButton is clicked. While the SubmitButton shouldn't be clickable if all questions haven't been answered, an extra check is performed to make sure that the player isn't able to submit partially completed answers. If all the questions have been answered, a comma seperated string is created using the names of the toggles selected for each question, which are always "A", "B" or "C". The resulting string representing all three answers is then passed into the SubmitAnswers function for the local player and the questions panel is then disabled.

```
public void InitialiseQuestions(Player localPlayer) {
  enable the questions panel;
  set the submit button to not be interactable;

  set allQuestionsAnswered = false;

  set this.localPlayer = localPlayer;
}
```

Figure 7.28: Pseudocode for the InitialiseQuestions function

InitialiseQuestions, shown in Figure 7.28, is called to display the questions panel at the end of a match. It enables the questions panel in the UI and sets the SubmitButton to not be interactable. It also initialises the allQuestionsAnswered variable to false and sets the reference to the local player, which will be used when the answers are submitted.

7.6 Database Manager

While not integral to gameplay, the DatabaseManager class is responsible for providing a handy wrapper for logging data to the databases during testing sessions. It interfaces with two SQLite databases, one for the participant information and the other for gameplay data. This separation was

to allow for easier anonymisation of the data at a later date. SQLite was chosen as the most suitable database system for this project largely for simplicity, as it doesn't require a server application to be configured and running (SQLite, 2018?d) and Unity3D includes a Mono library, Mono.Data.Sqlite, that allows easy interfacing with SQLite databases. Since SQLite databases are single files, they are easy to copy, backup and restore (SQLite, 2018?b) which will make backing up copies of the database before testing (to prevent data loss or corruption in the event that something goes wrong) and following testing in the case of accidental data loss much quicker and easier. The amount of data needed to be recorded is also relatively small, far less than the maximum supported SQLite file size of 140 terabytes (SQLite, 2018?b) which far exceeds the hard drive capacity of the server machine. All date and time variables were stored as Unix time stamps as SQLite "does not have a storage class set aside for storing dates and/or times" (SQLite, 2018?c).

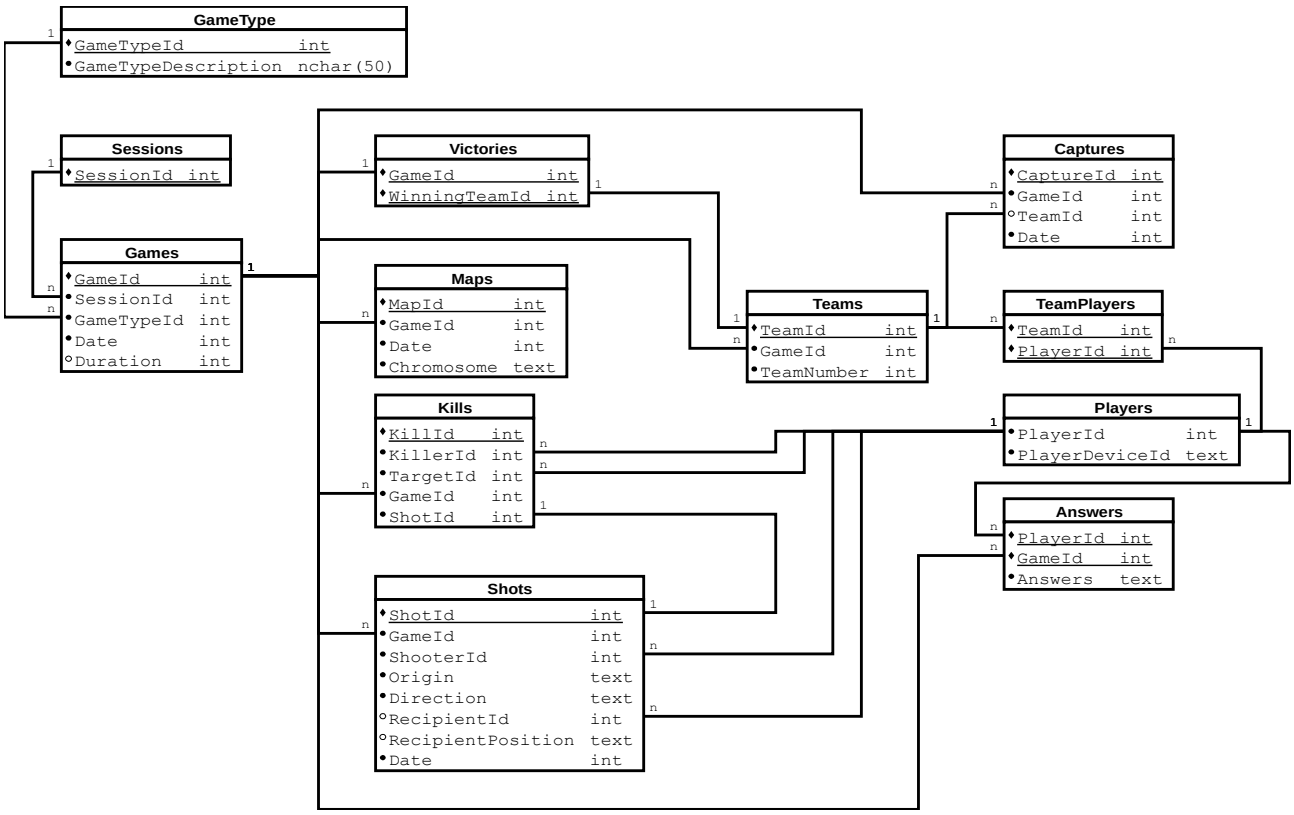

Figure 7.29: The database diagram for this project

Figure 7.29 shows the database design for this study. The twelve tables that form the gameplay database are:

- **GameType** – Stores the GameType enum as a hard coded table in the database. Stores the two enum values of "Control", meaning a game where the map does not change, and "procedural", where the map changes in response to team performance.

- **Sessions** – Each testing session is assigned a unique ID in this table
- **Games** – Stores metadata about each game that takes place. Specifically, it stores the session in which the game took place, the type of game (procedural or control), the date and time the game took place and how long the game took to complete in seconds.
- **Victories** – Each victory is stored in this database using the game id the victory took place in and the id of the winning team.
- **Maps** – Every time the map is updated, the chromosome is saved in this table as a Json string. The date and time at which it was generated is also saved along with the id of the game it was generated in.
- **Kills** – This table stores every kill made by players. Each entry records the id of the player who performed the kill, the id of the player who was killed, the id of the game the kill took place in and the id of the shot responsible for the kill.
- **Shots** – Stores every shot fired. Records the id of the game in which the shot was made, the id of the player firing the shot, the origin and direction vectors (as Json strings) and the date and time when the shot was fired. If the shot hit another player, this will be recorded using the other player's id and the position of the other player when they received the shot, encoded as a Json string.
- **Teams** – Each game has two new teams created for it, which are saved in this table. The id of the game the team was created for is recorded, along with a team number which corresponds to the Teams enum indicating if this was a red or a blue team.
- **Captures** – Every time the capture point is captured or reset it is logged in this table. Stores the id of the game in which the capture was made, the date and time when the capture was made and the team that now owns the point following the capture. If the point was reset to neutral, the team will be NULL.
- **TeamPlayers** – Stores which players belong to which teams. Each entry records a team id and a player id.
- **Players** – Stores players, assigning each an id. The PlayerDeviceId field is unique and is used to make sure return players are identified rather than adding a new player.
- **Answers** – This table stores the answers to the post game questionnaires. Records the id of the player providing the answers, the id of the game for which the answers are being provided and the answers themselves as a CSV string indicating if players picked A, B or C for each of the three multiple choice questions.

The participant information database has a single table named “Participants” which contains four fields: ParticipantId, Name, Email and DeviceId. The device ID is used to link the two databases, with the entry in the Players table in the gameplay database having the same value in PlayerDeviceId as the DeviceId field of the corresponding participant in the participant information database.

7.7 Map Sketch Helpers

MapSketchHelpers is a static class containing several static utility functions related to map sketches.

```
static TileType[,] ConvertChromosomeToMapSketch(Chromosome newChromosome, int mapSketchWidth,
  int mapSketchHeight) {

  set mapSketch = TileType[mapSketchWidth, mapSketchHeight];

  for (int currentGeneIndex = 0; currentGeneIndex < newChromosome.Genes.Count; ++currentGeneIndex)
  {
    set geneTuple = newChromosome.Genes[currentGeneIndex] as GeneTuple;

    if (currentgeneIndex < 15) {
      for (int x = geneTuple.X; x < geneTuple.X + abs(geneTuple.Z); ++x) {
        for (int y = geneTuple.Y; y < geneTuple.Y + abs(geneTuple.Z); ++y) {
          if (co-ordinate [x, y] is in the bounds of mapSketch) {
            set mapSketch[x, y] = TileType.Passable;
          }
        }
      }
    } else if (currentGeneIndex < 30) {
      if (geneTuple.Z > 0) {
        for (int x = geneTuple.X; x < geneTuple.X + geneTuple.Z; ++x) {
          for (int i = -1; i < 2; ++i) {
            if (co-ordinate [x, geneTuple.Y + i] is in the bounds of mapSketch) {
              set mapSketch[x, geneTuple.Y + i] = TileType.Passable;
            }
          }
        }
      } else {
        for (int y = geneTuple.Y; y < geneTuple.Y - geneTuple.Z; ++y) {
          for (int i = -1; i < 2; ++i) {
            if (co-ordinate [geneTuple.X + i, y] is in the bounds of mapSketch) {
              set mapSketch[geneTuple.X + i, y] = TileType.Passable;
            }
          }
        }
      }
    } else {
      for (int x = geneTuple.X; x < geneTuple.X + abs(geneTuple.Z); ++x) {
        for (int y = geneTuple.Y; y < geneTuple.Y + abs(geneTuple.Z); ++y) {
          if (co-ordinate [x, y] is in the bounds of mapSketch) {
            set mapSketch[x, y] = TileType.CapturePoint;
          }
        }
      }
    }
  }

  for (int x = 2; x < 7; ++x) {
    for (int y = 2; y < 7; ++y) {
      set mapSketch[x, y] = TileType.Team1Spawn;
    }
  }

  for (int x = mapSketchWidth – 7; x < mapSketchWidth – 2; ++x) {
    for (int y = mapSketchHeight – 7; y < mapSketchHeight – 2; ++y) {
      set mapSketch[x, y] = TileType.Team2Spawn;
    }
  }

  set the border tiles of the map sketch to TileType.Impassable;

  return mapSketch;
}
```

Figure 7.30: Pseudocode for the ConvertChromosomeToMapSketch function

ConvertChromosomeToMapSketch, shown in Figure 7.30, is responsible for taking a chromosome and map sketch dimensions and returning a two dimensional TileType array representing the map sketch. This is accomplished by iterating over the genes in the chromosome and casting the

ObjectValue of each to a GeneTuple. The first 15 genes are used to construct arenas, taking care to take the absolute value of the Z value of the tuple, since this can be negative. The second 15 genes are converted to corridors. If the Z value of the corridor gene tuples is greater than 0, a horizontal corridor is placed. Otherwise a vertical corridor is placed. The final gene is used to construct the capture point in the same way as the arena. Once the genes have been used to construct the map, the spawn points are placed and the border tiles of the map are made impassable. This is because arenas and corridors can be generated on the edge of the map, which under normal circumstances would then allow players to leave the playing field.

```
static Vector2[] GetReferenceTilePositionsForSpawns(int mapSketchWidth, int mapSketchHeight) {
  create a new array of Vector2s with two elements;

  add the centre of team 1's spawn to the array;
  add the centre of team 2's spawn to the array;

  return the array;
}
```

Figure 7.31: Pseudocode for the GetReferenceTilePositionsForSpawns function

Figure 7.31 shows the GetReferenceTilePositionsForSpawns function, which returns two vectors contained in an array that represent the location of the centre of the red team's spawn area and the location of the centre of the blue team's spawn area.

```
static List<Vector2> GetTargetTiles (Chromosome chromosome, int mapSketchWidth, int
  mapSketchHeight) {

  set geneTuple = chromosome.Genes.Last() as Chromosome;
  set returnList = List<Vector2>();

  for (int x = geneTuple.X; x < geneTuple.X + abs(geneTuple.Z); ++x) {
    for (int y = geneTuple.Y; y < geneTuple.Y + abs(geneTuple.Z); ++y) {
      if (co-ordinate [x, y] is in the bounds of the map sketch) {
        add the co-ordinate as a new Vector2 to the returnList;
      }
    }
  }

  return returnList;
}
```

Figure 7.32: Pseudocode for the GetTargetTiles function

GetTargetTiles, seen in Figure 7.32, is similar to GetReferenceTilePositionsForSpawns, but because the number of capture point tiles can vary it instead returns a list. It uses the last gene in the chromosome to add the co-ordinates of the capture point tiles to the list which it then returns.

```
static int[,] FloodFillMapSketch (TileType[,] mapSketch, int mapSketchWidth, int mapSketchHeight,
  Vector2 startingTile, Vector2? TargetTile = null) {

  set tiles = Queue<Tuple<int, int, int>>();

  set reachableTiles = int[mapSketchWidth, mapSketchHeight];
  fill reachableTiles with -1s;

  set visitedTiles = bool[mapSketchWidth, mapSketchHeight];

  tiles.Enqueue((startingTile.X, startingTile.Y, 0));
  set visitedTiles[startingTile.X, startingTile.Y] = true;

  set exitEarly = false;

  while (tiles.Count > 0 and exitEarly is false) {
    set currentTile = tiles.Dequeue();

    if (target tile isn't null) {
      if (the current tile is the target tile) {
        set exitEarly = true;
      }
    }

    if (mapSketch[currentTile.X, currentTile.Y] is not impassable) {
      set reachableTiles[currentTile.X, currentTile.Y] = currentTile.Z;

      if (currentTile.X + 1 < mapSketchWidth - 1) {
        if (visitedTiles[currentTile.X + 1, currentTile.Y] is false) {
          tiles.enqueue((currentTile.X + 1, currentTile.Y, currentTile.Z + 1));
          set visitedTiles[currentTile.X + 1, currentTile.Y] = true;
        }
      }

      if (currentTile.X - 1 > 0) {
        if (visitedTiles[currentTile.X - 1, currentTile.Y] is false) {
          tiles.enqueue((currentTile.X - 1, currentTile.Y, currentTile.Z + 1));
          set visitedTiles[currentTile.X - 1, currentTile.Y] = true;
        }
      }

      if (currentTile.Y + 1 < mapSketchHeight - 1) {
        if (visitedTiles[currentTile.X, currentTile.Y + 1] is false) {
          tiles.enqueue((currentTile.X, currentTile.Y + 1, currentTile.Z + 1));
          set visitedTiles[currentTile.X, currentTile.Y + 1] = true;
        }
      }

      if (currentTile.Y - 1 > 0) {
        if (visitedTiles[currentTile.X, currentTile.Y - 1] is false) {
          tiles.enqueue((currentTile.X, currentTile.Y - 1, currentTile.Z + 1));
          set visitedTiles[currentTile.X, currentTile.Y - 1] = true;
        }
      }
    }
  }

  return reachableTiles;
}
```

Figure 7.33: Pseudocode for the FloodFillMapSketch function

The FloodFillMapSketch function, shown in Figure 7.33, serves two main purposes. Firstly, it allows other algorithms to check if one tile is reachable from another tile, which is useful both for the fitness function and for the MapController functions concerning warping players to safe tiles. In this regard it also fulfils the requirement of the four way flood fill algorithm needed to implement the equations proposed by Liapis et al. (2013). Secondly, it calculates the distance from the starting

tile to every other tile on the map as it does so. Figure 7.34 shows a high level diagram of the flood fill process.

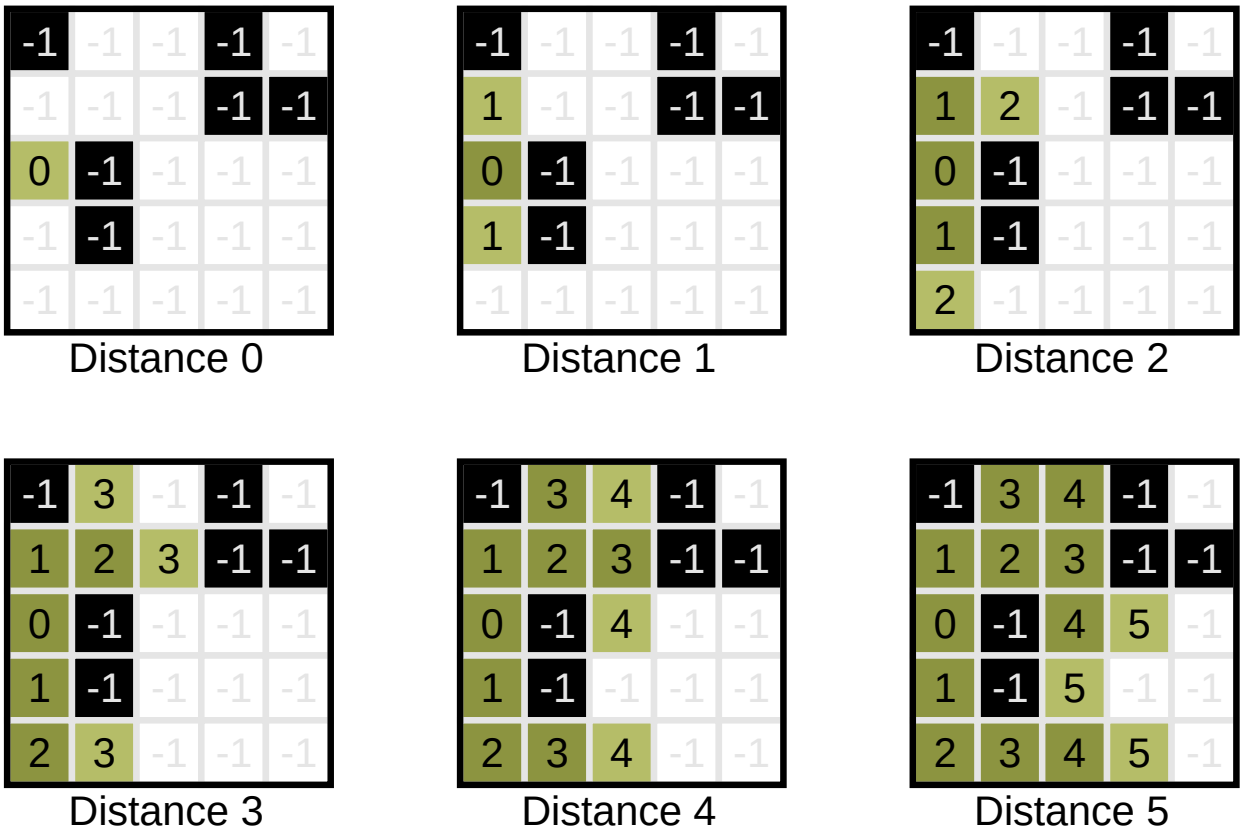

Figure 7.34: Diagram showing how the FloodFillMapSketch function fills the reachableTiles array. Impassable tiles are shown here in black and the numbers on each square represent the corresponding value in the reachableTiles array.

The reachableTiles array is initialised using the map sketch dimensions. This array stores a -1 for every unreachable tile and a positive integer representing the distance from the start tile for all other tiles. A queue of tuples is used to store the tiles still to examine. These tuples store the X and Y position of the tile in the first two element, while the third element is used to store the distance from the starting tile. The main loop repeats until either the queue is empty, which occurs when all reachable tiles have been explored, or if the exitEarly flag is set to true. Each iteration, a tile is dequeued and checked to see if it's the target. If it is, exitEarly is set to true, causing the loop to terminate on the next iteration. Otherwise, the corresponding tile in reachableTiles has its distance set to the current tile's distance and each of the four neighbouring tiles are examined to see if they're within the dimensions of the map sketch and haven't been visited already. Tiles that pass these conditions are enqueued and their corresponding tile in the visitedTiles array is set to true,

preventing this tile from being added again before it is examined. When the loop completes, the reachableTiles array is returned.

7.8 Map Sketch Extensions

```
static TileType[,] RemoveUnreachableTiles (this TileType[,] mapSketch, int mapSketchWidth,
  int mapSketchHeight) {

  set newMapSketch = TileType[mapSketchWidth, mapSketchHeight];
  set spawnPositions = MapSketchHelpers.GetReferenceTilePositionsForSpawns(mapSketchWidth,
    mapSketchHeight);

  set mapReachableForTeam1 = MapSketchHelpers.FloodFillMapSketch(mapSketch, mapSketchWidth,
    mapSketchHeight, spawnPositions[0]);
  set mapReachableForTeam2 = MapSketchHelpers.FloodFillMapSketch(mapSketch, mapSketchWidth,
    mapSketchHeight, spawnPositions[1]);

  for (int x = 0; x < mapSketchWidth; ++x) {
    for (int y = 0; y < mapSketchHeight; ++y) {
      if (co-ordinate[x, y] isn't reachable for one of the teams) {
        set newMapSketch[x, y] = TileType.Impassable;
      } else {
        set newMapSketch[x, y] = mapSketch[x, y];
      }
    }
  }

  return newMapSketch;
}
```

Figure 7.35: Pseudocode for the RemoveUnreachableTiles function

The MapSketchExtensions class provides a single extension method for removing unreachable tiles. Seen in Figure 7.35, this method allows a copy of any map sketch to be created with all unreachable tiles removed. This is accomplished by creating a temporary map sketch and then performing a flood fill on the original map sketch for both teams. Every tile that is reachable by both teams is copied to the new map sketch. Tiles that aren't reachable by one or more teams are set to be impassable in the new map sketch. This new map sketch is then returned.

7.9 Genetic Algorithm Helpers

The GeneticAlgorithmHelpers class provides several static functions to assist the genetic algorithm. It contains five static variables, which are initialised elsewhere in the program:

- mapSketchWidth (int) – the width of the map sketch in tiles
- mapSketchHeight (int) – the height of the map sketch in tiles
- team1TimeRemaining (float) – how much time team 1 has left on their capture timer

- team2TimeRemaining (float) – how much time team 2 has left on their capture timer
- timeToCapture (float) – how long a team must hold the capture point for to win

```
static double FitnessFunction (Chromosome chromosome) {
  set mapSketch = MapSketchHelpers.ConvertChromosomeToMapSketch(chromosome, mapSketchWidth,
    mapSketchHeight);

  if (mapSketch doesn't contain any capture point tiles) {
    return 0;
  }

  set referenceTiles = MapSketchHelpers.GetReferenceTilePositionsForSpawns(mapSketchWidth,
    mapSketchHeight);
  set targetTiles = MapSketchHelpers.GetTargetTiles(chromosome, mapSketchWidth, mapSketchHeight);

  set mapReachableForTeam1 = MapSketchHelpers.FloodFillMapSketch(mapSketch, mapSketchWidth,
    mapSketchHeight, referenceTiles[0]);
  set mapReachableForTeam2 = MapSketchHelpers.FloodFillMapSketch(mapSketch, mapSketchWidth,
    mapSketchHeight, referenceTiles[1]);

  if (the capture point isn't reachable by both of the teams) {
    return 0;
  }

  set strategicResourceControlForTeam1 = GetStrategicResourceControlValue(0, referenceTiles,
    mapReachableForTeam1, mapReachableForTeam2);
  set strategicResourceControlForTeam2 = GetStrategicResourceControlValue(1, referenceTiles,
    mapReachableForTeam2, mapReachableForTeam1);

  set team1CapturePercentage = 1 – (team1TimeRemaining / timeToCapture);
  set team2CapturePercentage = 1 – (team2TimeRemaining / timeToCapture);

  set percentageDelta = team1CapturePercentage – team2CapturePercentage;
  set strategicResourceControlDelta =
    strategicResourceControlForTeam1 – strategicResourceControlForTeam2;

  set fitness = 1 – abs(strategicResourceControlDelta + percentageDelta) / 2.0f;
  return fitness;
}
```

Figure 7.36: Pseudocode for the FitnessFunction function

The FitnessFunction function, seen in Figure 7.36, is used by the genetic algorithm as its fitness function. To evaluate the fitness of a chromosome, it is first converted to a map sketch. If this map sketch doesn't contain any capture point tiles, or the capture point is not reachable by both teams, a fitness of 0 is returned. Otherwise, the fitness is calculated using Equation 7.1, where ΔS is the difference in strategic resource control between the two teams (calculated using the strategic resource control function proposed by Liapis et al. (2013), seen in Equation 4.7) and ΔC is the difference in capture percentage between the two teams. Both ΔS and ΔC can be between -1 and 1 inclusive. The graph of the fitness function is show in Figure 7.37. If team 1 has a strategic resource control advantage, their ΔS value will be positive. If they also have a capture percentage advantage, their ΔC value will also be positive. In this scenario, the map is unbalanced as team 1 is evidently performing better *and* has better strategic resource control. If team 2 had the strategic resource control advantage instead, team 1's ΔS value would be negative, causing a

correspondingly higher fitness value as team 2 have a strategic resource control advantage and are the weaker team. In this way, the fitness function rewards chromosomes that result in maps in which the weaker team (capture-wise) has an advantage in terms of strategic resource control.

$$f = 1 - \frac{|\Delta S + \Delta C|}{2}$$

Equation 7.1: The fitness function for the genetic algorithm

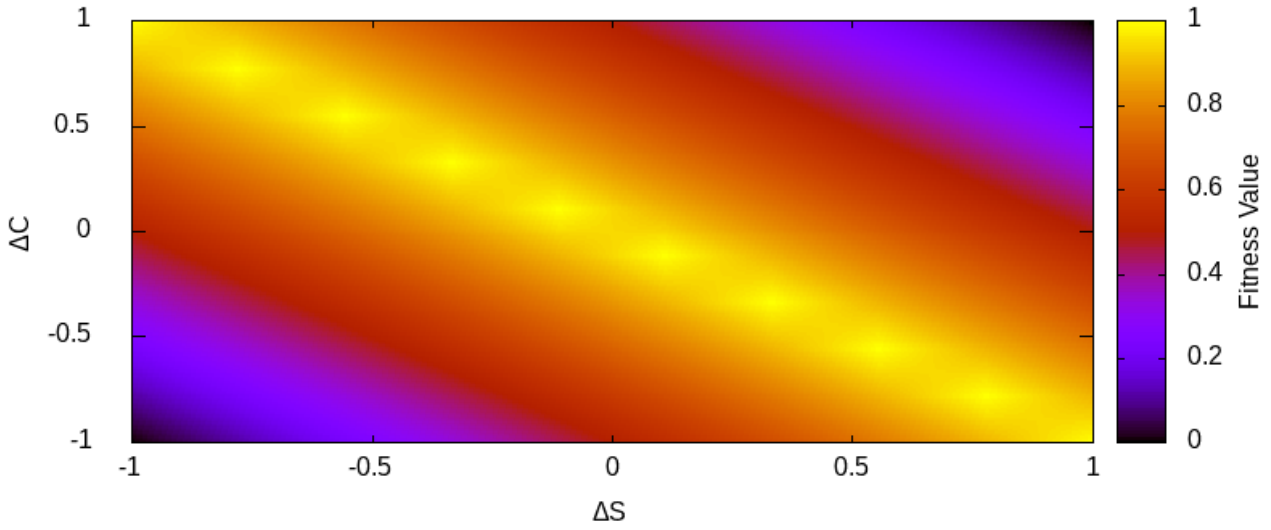

Figure 7.37: The graph of the fitness function for the genetic algorithm

```
static float GetSafetyValue (Vector2 tile, int i, List<Vector2> referenceTiles,
  int[,] iDistanceMap, int[,] jDistanceMap) {

  set currentLowestSafetyValue = 1;

  for (int j = 0; j < referenceTiles.Count; ++j) {
    if (j does not equal i) {
      set distanceToJ = jDistanceMap[tile.X, tile.Y];
      set distanceToI = iDistanceMap[tile.X, tile.Y];

      set safetyValue = max(0, (distanceToJ – distanceToI) / (distanceToJ + distanceToI));
      if (safetyValue is less than the currentLowestSafetyValue) {
        currentLowestSafetyValue = safetyValue;
      }
    }
  }

  return currentLowestSafetyValue;
}
```

Figure 7.38: Pseudocode for the GetSafetyValue function

The GetSafetyValue function seen in Figure 7.38 is an implementation of the safety value function proposed by Liapis et al. (2013), seen in Equation 4.5.

```
static float GetStrategicResourceControlValue (int I, List<Vector2> referenceTiles,
  List<Vector2> targetTiles, int[,] iDistanceMap, int[,] jDistanceMap) {

  set totalSafetyForReferenceTile = 0;

  for (int k = 0; k < targetTiles.Count; ++k) {
    set totalSafetyForReferenceTile = totalSafetyForReferenceTile + GetSafetyValue(targetTiles[k],
      i, referenceTiles, iDistanceMap, jDistanceMap);
  }

  return totalSafetyForReferenceTile / targetTiles.Count;
}
```

Figure 7.39: Pseudocode for the GetStrategicResourceControlValue function

Figure 7.39 shows the GetStrategicResourceControlValue function, a modified version of the strategic resource control function proposed by Liapis et al. (2013), seen in Equation 4.7. The modifications allow the implementation to specify a specific reference tile (given as an index by the 'I' argument) to calculate the strategic resource value for.

7.10 Map Mutate Operator

The MapMutate class inherits from the MutateBase class and provides a custom mutate operator that works with the Genetic Algorithm Framework. It contains two variables initialised through the constructor: mapWidth and mapHeight.

```
override void MutateGene(Gene gene) {
  set newX = random integer between 0 and mapWidth – 1 inclusive;
  set newY = random integer between 0 and mapHeight – 1 inclusive;

  if (a newly generated random integer between 0 and 1 inclusive is 0) {
    set newZ = random integer between 0 and mapWidth – newX – 1 inclusive;
  } else {
    set newZ = random integer between -mapHeight + newY and 0 inclusive;
  }

  set gene.ObjectValue = new GeneTuple(newX, newY, newZ);
}
```

Figure 7.40: Pseudocode for the MutateGene function

The MutateGene function, seen in Figure 7.40, is responsible for mutating an individual gene. This is done by randomising all three values of the new geneTuple. The new X and Y positions are set to

a random integer within the bounds of the map width and map height. Based on the result of a “coin toss”, accomplished using a random integer that can either be 0 or 1, the new Z value is either set to a positive or negative value. If positive, this value is between 0 and the mapWidth minus the new X position of the tuple. If negative, the value is instead between -mapHeight plus the new Y position of the tuple and 0. The gene’s object value is then set to a new GeneTuple created using the three new values.

7.11 Game Time Manager

The GameTimeManager class is a singleton class used for managing the game time and the capture timers for each team. It has simple functions to start and stop the timers and is updated only on the server. Each timer is a syncvar and so is automatically synchronised with the timer on the GameTimeManager on the client.

```
void Update() {
  if (isServer) {
    if (the game timer isn't paused) {
      set GameTimerPaused = true;
      set GameTimeRemaining -= Time.deltaTime;

      if (GameTimeRemaining <= 0) {
        pause all timers;
        if (RedTeamCaptureTimeRemaining < BlueTeamCaptureTimeRemaining) {
          call Won(Team.Red) on the first player in the players static list;
          finish the game on the database;
        } else if (BlueTeamCaptureTimeRemaining < RedTeamCaptureTimeRemaining) {
          call Won(Team.Blue) on the first player in the players static list;
          finish the game on the database;
        } else {
          call Won(Team.Random) on the first player in the players static list;
          finish the game on the database;
        }
      }
    }
    if (the red team capture timer isn't paused) {
      set RedTeamCaptureTimeRemaining -= Time.deltaTime;

      if (RedTeamCaptureTimeRemaining <= 0) {
        set RedTeamCaptureTimeRemaining = 0;
        pause all timers;
        call Won(Team.Red) on the first player in the players static list;
        finish the game on the database;
      }
    }
    if (the blue team capture timer isn't paused) {
      set BlueTeamCaptureTimeRemaining -= Time.deltaTime;

      if (BlueTeamCaptureTimeRemaining <= 0) {
        set BlueTeamCaptureTimeRemaining = 0;
        pause all timers;
        call Won(Team.Blue) on the first player in the players static list;
        finish the game on the database;
      }
    }
  }
}
```

Figure 7.41: Pseudocode for the Update function

Figure 7.41 shows the Update function for the GameTimeManager. This function is where the timers are updated and where the logic for winning the game has been moved to. The Won() function is called on the first player in the static player list contained in the Player class with the winning team as the argument. As discussed previously, Team.Random indicates a neutral team and in this case indicates a draw. If the game timer runs out, the team with the least remaining time is said to have won. If either of the two team capture timers run out, that team is said to have won. The game is finished on the database here by calling the FinishGame function on the databaseManager, which updates the game duration in the database and adds a new entry in the Victories table if appropriate.

7.12 Lobby Modifications

The existing lobby code had to be modified to support the new team mechanic and also the logging of participant information to the database.

7.12.1 Lobby Player

Three new syncvars were added to the LobbyPlayer class to store the player's team, team id and player id. A new button was also added to the LobbyPlayer prefab that displays the current team of the player. Methods were added to support changing teams between Red, Blue or Random but these were removed for testing purposes.

```
void SetupLocalPlayer() {
  …

  using (var simpleAES = new SimpleAES()) {
    set encryptedName = simpleAES.Encrypt(PlayerData.Instance.Name);
  }
  using (var simpleAES = new SimpleAES()) {
    set encryptedEmail = simpleAES.Encrypt(PlayerData.Instance.EmailAddress);
  }
  using (var simpleAES = new SimpleAES()) {
    set encryptedDeviceId = simpleAES.Encrypt(PlayerData.Instance.DeviceId);
  }

  call CmdAddPlayerToDatabase(encryptedName, encryptedEmail, encryptedDeviceId);

  …
}
```

Figure 7.42: Pseudocode of the section of the SetupLocalPlayer function responsible for encrypting and sending participant info to the server

```
public void CmdAddPlayerToDatabase(string encryptedName, string encryptedEmail,
  string encryptedDeviceId) {

  using (simpleAES = new SimpleAES()) {
    set name = simpleAES.Decrypt(encryptedName);
    set emailAddress = simpleAES.Decrypt(encryptedEmail);
    set deviceId = simpleAES.Decrypt(encryptedDeviceId);

    add a new participant to the database using the name, emailAddress and deviceId values;

    set PlayerId = the playerID returned from adding a new player to the database;
    set PlayerTeam = the previous team the player was in this session, otherwise Team.Random;
  }
}
```

Figure 7.43: Pseudocode showing the CmdAddPlayerToDatabase function

A new server command was added, CmdAddPlayerToDatabase (shown in Figure 7.43), which is called in the SetupLocalPlayer function (seen in Figure 7.43). When a player joins, their participant

information is encrypted and sent to the server by calling CmdAddPlayerToDatabase, which subsequently decrypts the information and adds the participant to the participant database. It then calls the AddPlayer function in the database manager, which takes the device Id as an argument. If this participant has previously participated, their existing player id will be returned. Otherwise, a new player entry will be added for them and the new id will be returned. PlayerId is then set to the returned value. After this the GetPlayerTeamForSession function is called on the database manager and the value returned is assigned to the PlayerTeam variable. This function checks to see if the player has already played a game in this testing session. If they have, their team is assigned to their previous team. This is useful in case a player loses connection or accidentally quits, since upon rejoining their old team value will be assigned to them and the team arrangement will be preserved.

7.12.2 Game Lobby Hook

The GameLobbyHook class was modified slightly from the original version in the Merry Fragmas tutorial. The player's colour is no longer assigned, since this is determined by the team. Instead, the player's PlayerId, PlayerTeam and PlayerTeamId variables are set using the variables from the lobby player.

7.12.3 Security

Since participant data was being sent over the network, it was deemed necessary to implement encryption to secure said data against potential packet sniffing attempts or other malicious activity on the participant's local network. This was accomplished using a new class called SimpleAES. Code by Stack Overflow user Angularsen (2015) was used as a starting point, then modified to improve security. The original implementation used a hard coded encryption key which presented a security risk if a malicious user were to get hold of the application and reverse engineer it. To solve this problem a T4 template was created that, when run, generates a static class called EncryptionKey that contains a single static byte array of 32 elements, generated using the RNGCryptoServiceProvider class in the System.Security.Cryptography namespace. This value is used as the encryption key in the SimpleAES class. By using this template, new encryption keys can be quickly generated for new builds, allowing each session to have a different encryption key and reducing the likelihood of the encryption being cracked.

7.12.4 Lobby Manager

To allow the game type to be changed mid-session, a button was added to the lobby, visible only to the server, allowing the game type to be switched from Control to Procedural. A variable to track the game type was added, which is flipped between GameType.Procedural and GameType.Control by a function called when the button is clicked.

```
public void StartGame() {
  start a new game in the database;
  set the game type in the game instance data;

  add a new red team and a new blue team and store the new ids;
  set the red team id in the game instance data to the id of the new red team;
  set the blue team id in the game instance data to the id of the new blue team;

  call ResolvePlayers(redTeamId, blueTeamId);
  call StartCoroutine(ServerCountdownCoroutine());
}
```

Figure 7.44: Pseudocode for the StartGame function

Figure 7.44 shows the changes made to the StartGame function. When this function is called, a new game is started in the database using a call to the DatabaseManager and the GameType variable in the GameInstanceData class is set to the gameType variable. A new red team and blue team are added to the database using the DatabaseManager and the ids of the new teams are used to first update the RedTeamId and BlueTeamId variables in the GameInstanceData class before being passed as arguments to the ResolvePlayers function. Finally, the coroutine to start the server countdown is called as normal.

```
private void ResolvePlayers(int redTeamId, int blueTeamId) {
  set redCount = 0;
  set blueCount = 0;
  set playerCount = 0;
  set unplacedIndices = new List<int>();

  for (int i = 0; i < lobbySlots.Length; ++i) {
    if (lobbySlots[i] != null) {
      ++playerCount;

      switch (lobbySlots[i].PlayerTeam) {
        case Team.Random:
          unplacedIndices.Add(i);
          break;
        case Team.Red:
          add this player to the red team in the database;
          set lobbySlots[i].PlayerTeamId = redTeamId;
          ++redCount;
          break;
        case Team.Blue:
          add this player to the blue team in the database;
          set lobbySlots[i].PlayerTeamId = blueTeamId;
          ++blueCount;
          break;
      }
    }
  }

  set remainingRed = Abs(playerCount / 2 – redCount);
  set remainingBlue = Abs(playerCount / 2 – blueCount);

  foreach (int unplacedIndex in unplacedIndices) {
    if(remainingRed > 0 and remainingBlue > 0) {
      set newTeam = Team.Red or Team.Blue with a 50/50 chance of each;
    } else {
      set newTeam = the team that has remaining slots;
    }

    if (newTeam == Team.Red) {
      add the player at lobbySlots[unplacedIndex] to the red team in the database;
      set lobbySlots[unplacedIndex].PlayerTeamId = redTeamId;
      --remainingRed;
    } else {
      add the player at lobbySlots[unplacedIndex] to the blue team in the database;
      set lobbySlots[unplacedIndex].PlayerTeamId = blueTeamId;
      --remainingBlue;
    }

    set lobbySlots[unplacedIndex].PlayerTeam = newTeam;
  }
}
```

Figure 7.45: Pseudocode for the ResolvePlayers function

The ResolvePlayers function, shown in Figure 7.45, is used to assign any players without a team such that the red team and the blue team have as close to equal numbers of players as possible. This is accomplished by iterating through the players in the lobby and incrementing the playerCount, redCount and blueCount variables. The latter two are incremented only when a player with the corresponding team is found, but the playerCount is incremented for every player found. Any players who are in the “Random” team have their index in the lobbyPlayers list added to the unplacedIndices list. When players are found that are part of a team they are added to that team in the database using a call to the DatabaseManager and their PlayerTeamId property is set to the id of

their team. Once the initial loop has completed, the number of red and blue slots that need to be filled to create a 50/50 split in the teams are calculated. The function then iterates through each unplaced index. If there are still remaining red and blue slots to be filled, the team that the player at the unplaced index will be assigned is determined randomly. Otherwise, if there is only one team with slots to be filled, the player will be assigned to that team. Once the team has been decided, the player at the unplaced index is added to their team in the database and their PlayerTeam and PlayerTeamId properties are set.

7.12.5 Lobby Main Menu

Two new functions were added to the LobbyMainMenu script, OnClickDedicatedControl and OnClickDedicatedProcedural. These are used in place of the original OnClickDedicated function and are called by two new buttons on the lobby menu. Each starts the server the same way as the OnClickDedicated function but with the added step of assigning the lobbyManager's gameType property to either GameType.Control or GameType.Procedural, depending on the button clicked.

7.13 Game Instance Data

GameInstanceData is a singleton class that is attached to a gameobject that persists between scene changes. It is used to store the IDs of the red and blue teams along with the current game type. These variables are initialised in the LobbyManager before switching to the game scene. By storing them in this persistent singleton class they can be easily accessed from within the game scene by various other classes.

7.14 Player Data

PlayerData is another persistent singleton class. In this case, it is used to store participant information, specifically the participant's name, email address and unique device id. These are initialised in the briefing controller but are required every time the player connects to the server. As a result they are stored in this class to allow them to be easily accessed for the lifetime of the program execution.

7.15 Vector3 Extensions

The Vector3Extensions static class provides a single extension method for Vector3 objects called ToTuple. This simply returns a new 3 dimensional Tuple representing the vector to enable easier serialisation and storing of vectors in the database.

7.16 Briefing Controller

To store the participant information entered by the players when consenting to take part in the experiment, and to prevent the consent button from being clicked until the briefing text has been scrolled all the way to the bottom, the BriefingController script was created. The BriefingController script is also responsible for setting the values in the PlayerData instance when the consent button is clicked and the briefing is closed.

8 Findings and Analysis

Six testing sessions were conducted, with a total of 20 unique players and 27 games. 3 games encountered connection problems, meaning they had to be repeated and the originals excluded from the analysis. These were:

- Game 16 – A network synchronisation error caused several players to get stuck in the loading state, unable to play.
- Game 19 – The same issue as game 16
- Game 20 – One player encountered local connection problems and was disconnected. They reconnected the following game with no problems.

8.1 Problems and Workarounds

Two timing issues were identified only upon conclusion of testing. The first issue was that the time games began was being saved to the database too early, namely before the lobby countdown timer began. This meant that when the game duration was calculated by subtracting the time the game started from the time the game finished, extra time was added on by the countdown timer and the time taken to load the game scene on the server. This was identified due to the durations of the five games that ended due to the game timer running out, then confirmed upon analysis of the code. The five games were:

- Game 6 – 603 seconds
- Game 12 – 603 seconds
- Game 17 – 603 seconds
- Game 18 – 608 seconds
- Game 25 – 604 seconds

All five obviously exceed the intended maximum game time of 600 seconds. This has no impact on the logging of captures, kills, maps etc. but does affect the balance value equation. In order to mitigate this, these five games had their durations capped at 600 when calculating the balance values for the games. All other games had the average of the additional time (4 seconds, to the nearest second) subtracted from their durations when calculating the balance value.

The second timing issue was caused by the way datetimes were saved to the database as UNIX timestamps. These were saved in seconds rather than milliseconds as this was deemed to be sufficiently accurate at the time. Unfortunately, the team capture timers were floating point variables and their remaining time was not saved directly to the database at the end of the game. Instead, captures were saved using timestamps and the remaining time for each team at the end of a game must be calculated using the saved captures and, in effect, simulating the capture point. Several games were identified where, although a team had won by capturing, the winning team was shown to have 1 second remaining on their capture timer due to the loss in precision. The workaround for this was, thankfully, relatively simple. Where games had a duration of less than 600 seconds, indicating the game was won through capturing rather than the game timer expiring, if the winning team's capture timer was calculated to be 1 it was instead set to 0.

It was found when performing analysis on the answers to the post game questionnaire that there was a single instance in which a player's answers had not been recorded. More specifically, game 10 which took place during test session 3 recorded only five sets of answers instead of the expected 6.

8.2 Supporting Software

An application named “Map Exporter” was developed to assist with data analysis, allowing map chromosomes to be exported as images for visual analysis and calculating balance values for games, among other things. This application also incorporated the workarounds described in the previous section.

8.3 Analysis

The graphs in this section use an alternating grey and white background to indicate which games belong to which of the six sessions. Analysis of the data has been split into two sections for convenience, first covering the results of the control games and then the results of the procedural games. Unless otherwise stated, all figures have been rounded to three significant figures.

8.3.1 Control

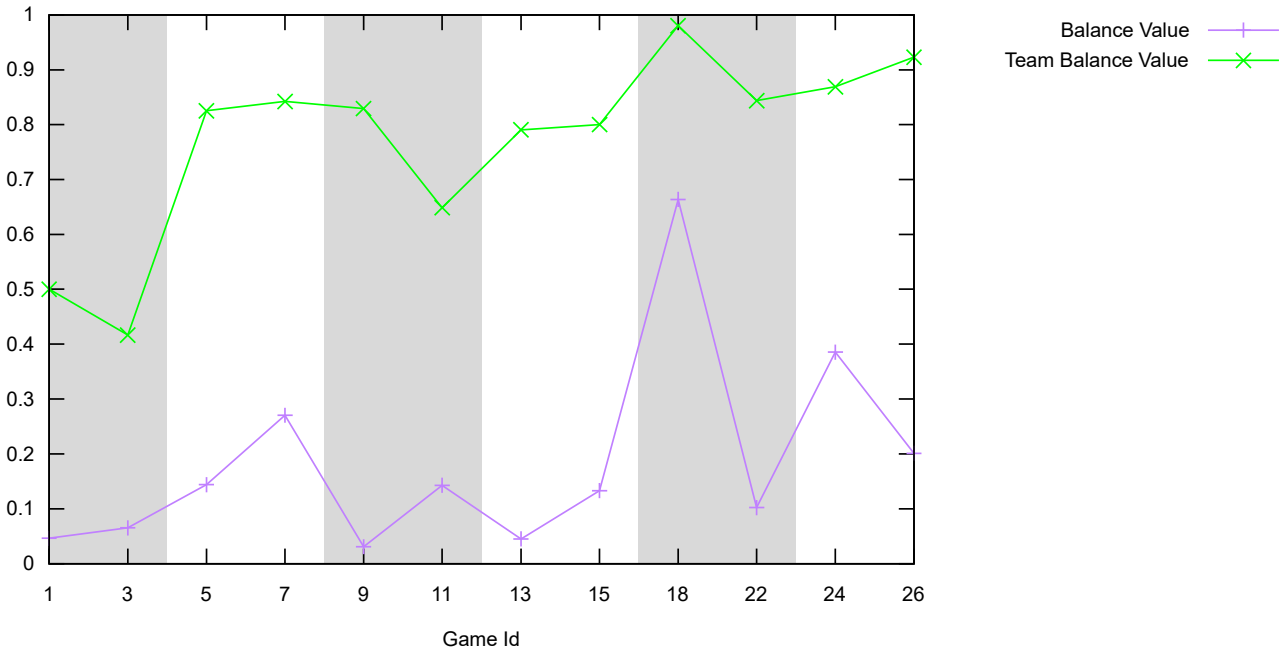

Figure 8.1: A graph showing the balance values and team balance values for the control games

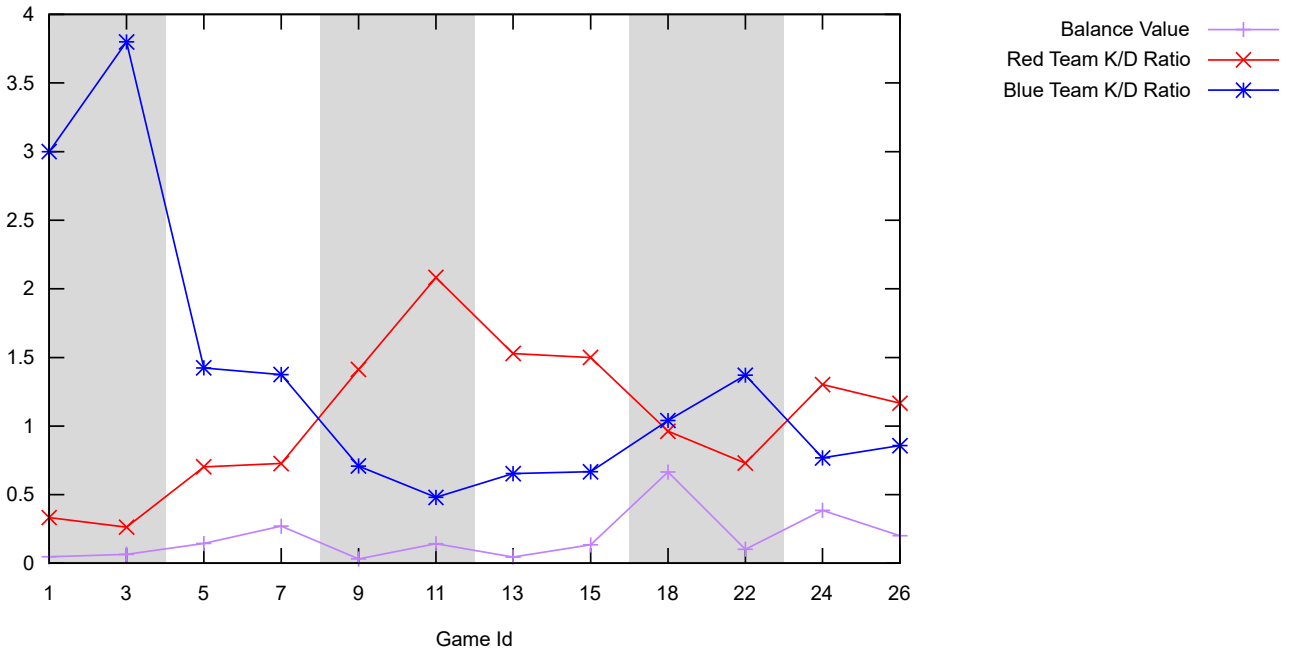

Figure 8.2: A graph showing the balance values and the Kill/Death ratios for the teams for the control games

	Balance Value	Team Balance Value	Red Team K/D Ratio	Blue Team K/D Ratio
Mean	0.186	0.772	1.06	1.35
Median	0.138	0.827	1.06	0.949
Maximum	0.664	0.980	2.08	3.80
Minimum	0.0310	0.417	0.263	0.480
Standard Deviation	0.183	0.167	0.537	1.02

Table 8.1: Statistics for the balance value, team balance value, red team kill/death ratio and blue team kill/death ratio for the control games

On average, the control games had a balance value of 0.186, which is far below the definition of balance set out in Section 5.3. Only a single game, game 18, could be considered balanced according to this definition with its balance value of 0.664. The team balance values for the control games appear balanced on average, according to the definition of team balance set out in Section 5.3. However, three games fell below the team balance threshold: game 1 with a team balance value of 0.5, game 3 with a team balance value of 0.417 and game 7 with a team balance value of 0.649. With regards to the team kill/death ratios, the average values indicate that the blue teams tended to perform better than the red teams on average, though the exceptionally large K/D ratios achieved by the blue team in session 1 may account for most of this advantage, with the blue teams having a maximum recorded K/D ratio of 3.8 compared to the red teams' 2.08. Given the random composition of the teams for each session, it would be expected that given enough different testing

sessions this disparity would disappear. It is worth noting that the average values for the K/D ratios are based on the ratios themselves and not the total kills and deaths for the teams across all sessions.

A correlation coefficient of 0.571 between the team balance value and the balance value seems to suggest that, as would be expected in the control games, more balanced teams lead to a more balanced game outcome. However, sessions 1, 3 and 6 go against this notion, with the second game in each of these sessions having a lower team balance value but a higher balance value compared to the first game (in the case of sessions 1 and 3) or a higher team balance value but a lower balance value (in the case of session 6).

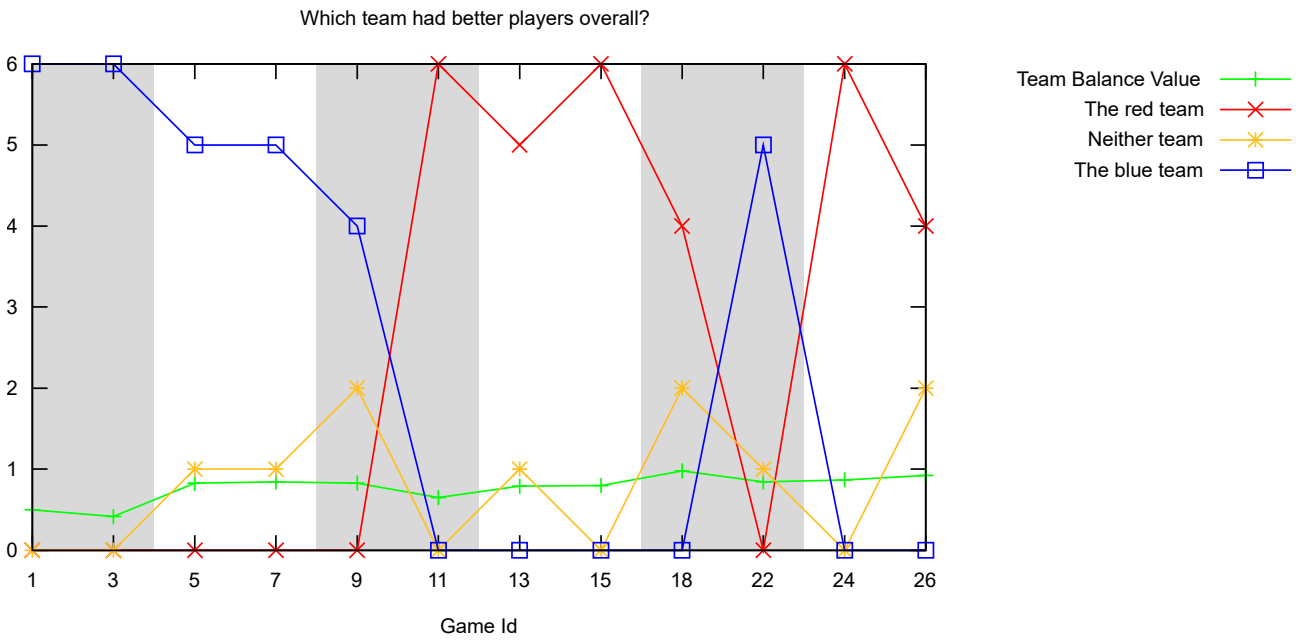

Figure 8.3: A graph showing the answers given to the first question in the questionnaire and the team balance value for the control games

	Team Balance Value	The Red Team	Neither Team	The Blue Team
Mean	0.772	2.58	0.833	2.58
Median	0.827	2	1	2
Maximum	0.980	6	2	6
Minimum	0.417	0	0	0
Standard Deviation	0.167	2.78	0.835	2.75

Table 8.2: Statistics for the team balance value and the answers given for the first question for the control games

The answers to the first question (shown in Figure 8.3 and Table 8.2) indicate that, on average, 43% of players felt the red teams were more skilled, 43% felt the blue teams were more skilled and 14% felt that the teams were evenly matched in each respective game. The median, maximum, minimum and standard deviation values of the responses suggest that the player opinion of team skill was almost perfectly divided between one team or the other being the strongest in each game, with no tendency towards one particular team. This was expected given the random composition of the teams. The player opinion appears to have a strong positive correlation with the observed K/D ratios, with a correlation coefficient of 0.780 between the number of answers for the red team being more skilled and the red team's K/D ratio, and a correlation coefficient of 0.737 between the number of answers for the blue team being more skilled and the blue team's K/D ratio. In terms of the team balance values, there appears to be a very weak positive correlation between the answers given for the red team being more skilled and the team balance value, with a correlation coefficient of 0.300. The correlation co-efficient between the number of answers for neither team and the team balance value is a more expected 0.679. Finally, the correlation coefficient between the answers given for the blue team being more skilled and the team balance value is -0.508.

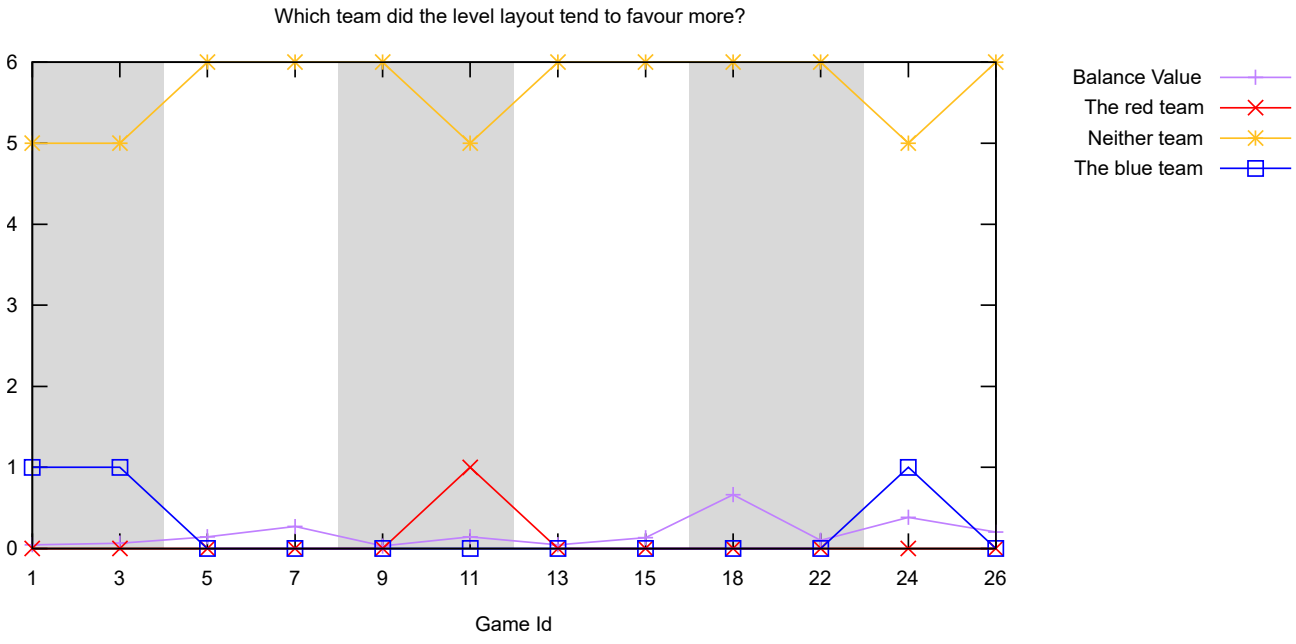

Figure 8.4: A graph showing the answers given to the second question in the questionnaire and the balance value for the control games

	Balance Value	The Red Team	Neither Team	The Blue Team
Mean	0.186	0.0833	5.67	0.25
Median	0.138	0	6	0

Maximum	0.664	1	6	1
Minimum	0.0310	0	5	0
Standard Deviation	0.183	0.289	0.492	0.452

Table 8.3: Statistics for the balance value and the answers given for the second question for the control games

The answers for the second question (shown in Figure 8.4 and Table 8.3) show that, on average, the majority of players felt that neither team was favoured by the level layout; rounding to two decimal places, 94.44% of players felt that the level layout favoured neither team, 4.17% of players felt that the level layout favoured the blue team and 1.39% of players felt that the level layout favoured the red team. These results are expected given that in the control games the map remained static and so, in theory, should have conferred no extra benefits to either team.

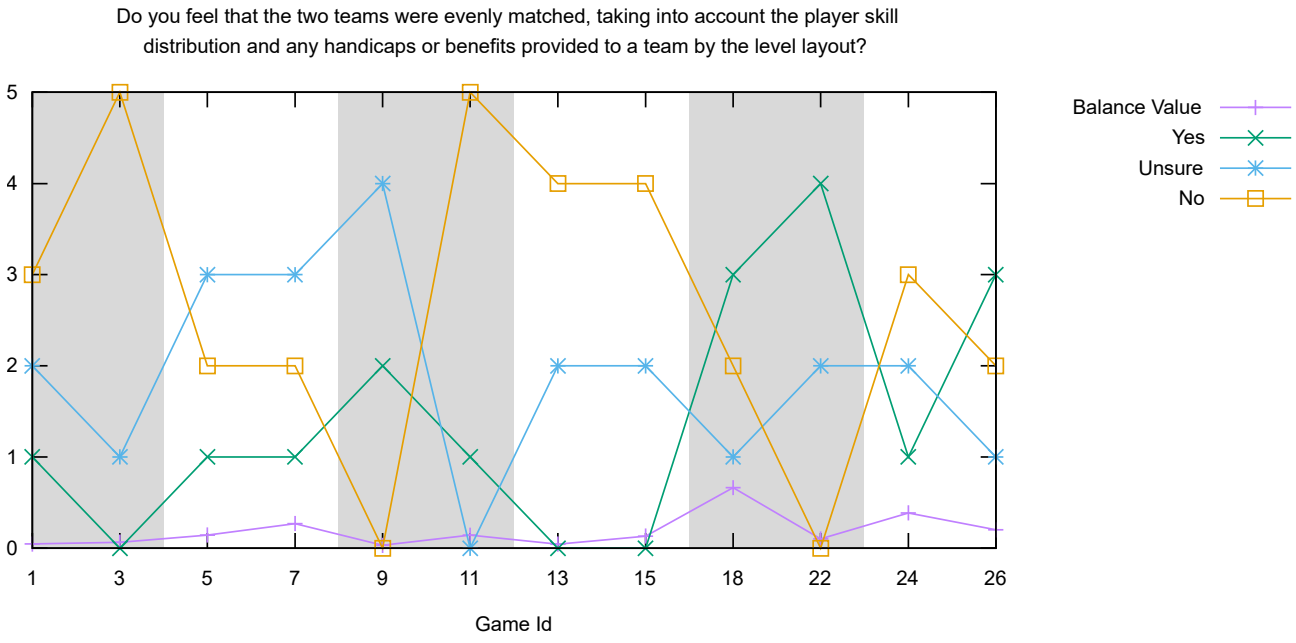

Figure 8.5: A graph showing the answers given to the third question in the questionnaire and the balance value for the control games

	Balance Value	Yes	Unsure	No
Mean	0.186	1.42	1.92	2.7
Median	0.138	1	2	2.5
Maximum	0.664	4	4	5
Minimum	0.0310	0	0	0

Standard Deviation	0.183	1.31	1.08	1.7

Table 8.4: Statistics for the balance value and the answers given for the third question for the control games

The answers to the third and final question for the control games (shown in Figure 8.5 and Table 8.4) show that players found the games to be unbalanced on average. Rounded to two decimal places, 44.44% of players felt the games were unbalanced, 23.61% felt the games were balanced and 31.94% of players said they were unsure (the missing 0.1% is due to rounding inaccuracy).

The correlation coefficient between the number of answers that the game was balanced and the balance value is 0.339, indicating a weak positive correlation. The correlation coefficient between the number of answers that the game was not balanced and the balance value indicates a weak negative correlation at -0.102. A much stronger correlation was observed between the answers for this question and the team balance value. Between the number of answers that the game was balanced and the team balance value, the correlation coefficient is 0.537 and between the number of answers that the game is not balanced and the team balance value the correlation coefficient is -0.578.

8.3.2 Procedural

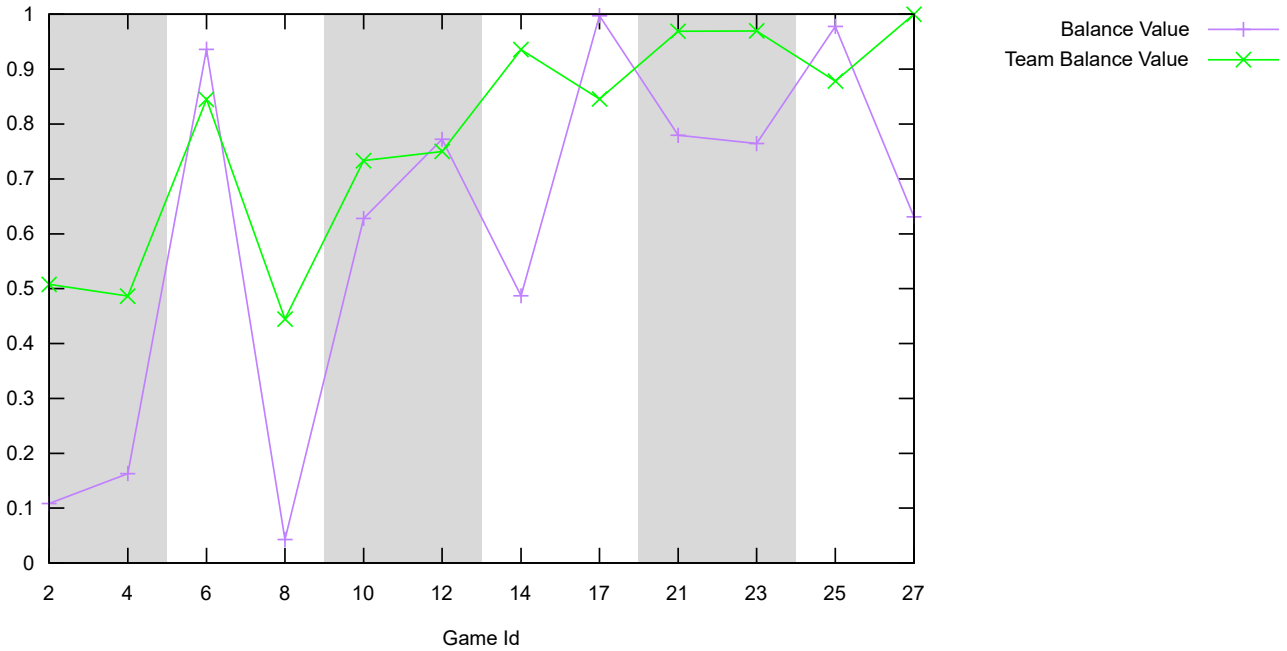

Figure 8.6: A graph showing the balance values and team balance values for the procedural games

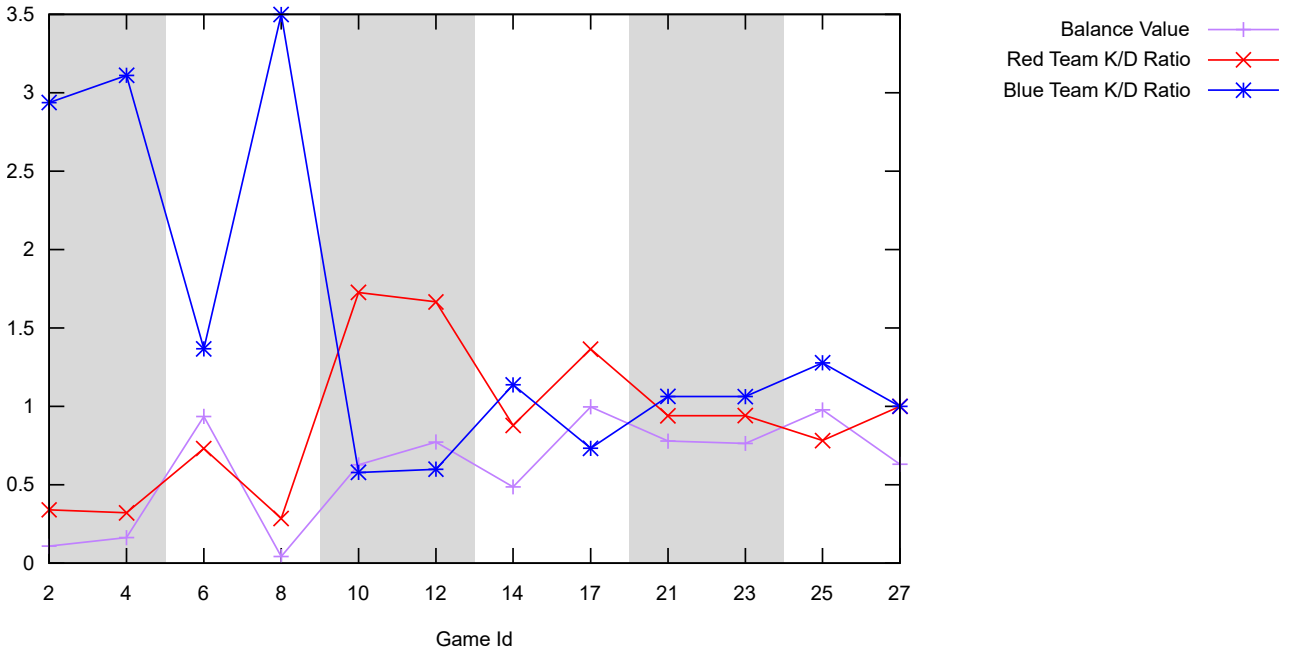

Figure 8.7: A graph showing the balance values and the Kill/Death ratios for the teams for the procedural games

	Balance Value	Team Balance Value	Red Team K/D Ratio	Blue Team K/D Ratio
Mean	0.607	0.780	0.915	1.53
Median	0.698	0.845	0.909	1.10
Maximum	0.997	1	1.73	3.5
Minimum	0.0429	0.444	0.286	0.579
Standard Deviation	0.338	0.200	0.483	1.03

Table 8.5: Statistics for the balance value, team balance value, red team kill/death ratio and blue team kill/death ratio for the procedural games

On average, the procedural games (shown in Figures 8.6 and 8.7 and Table 8.5) had a balance value of 0.607, above the threshold for balance defined in Section 5.3. Four games fell below the balance threshold: 2, 4, 8 and 14, with balance values of 0.108, 0.163, 0.0429 and 0.487 respectively. Despite this, the procedural games had higher average, median, maximum and minimum values than the control games, along with a higher standard deviation. The team values are very similar to the team values of the control games, with an average team balance value of 0.780, slightly higher than the 0.772 of the control games. The other statistical values shown in Table 8.5 are also marginally higher. There appears to be a larger difference between the K/D ratios of the teams in the procedural games, with a lower red team K/D ratio average and higher blue team K/D ratio average than in the control games by 0.145 and 0.18 respectively. The other statistical values for the two team K/D ratios differ very little from those of the control game, which is to be expected given that

the team compositions remain the same on both the control games and the procedural games. One possible explanation for the slight difference between the figures in the control and procedural games is that the procedural games take place on the 2nd and 4th games, meaning players will theoretically have more experience and therefore skill in the game for the procedural games than the control games, which will be played 1st and 3rd.

The correlation coefficient between the team balance value and the balance value is 0.783, much higher than the correlation coefficient between the two values in the control games. As in the control games however, certain sessions show second games which seem to show a negative correlation. For example, between games 25 and 27 (which both took place in session 6), the team balance value rose from 0.878 to a perfect value of 1 while the balance value fell from 0.978 to just 0.631.

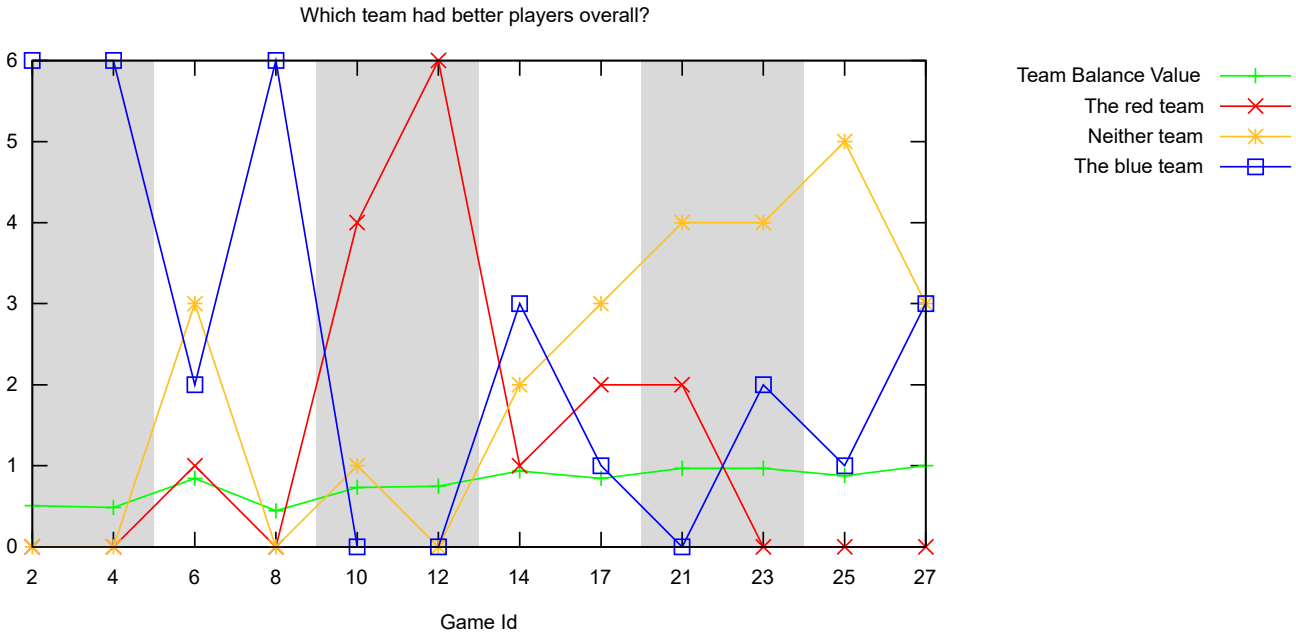

Figure 8.8: A graph showing the answers given to the first question in the questionnaire and the team balance value for the procedural games

	Team Balance Value	The Red Team	Neither Team	The Blue Team
Mean	0.780	1.33	2.08	2.5
Median	0.845	0.5	2.5	2
Maximum	1	6	5	6
Minimum	0.444	0	0	0
Standard Deviation	0.200	1.92	1.83	2.35

Table 8.6: Statistics for the team balance value and the answers given for the first question for the procedural games

The answers to the first question for the procedural games (shown in Figure 8.8 and Table 8.6) show that, rounded to two decimal places, 22.22% of players felt the red teams were more skilled, 41.67% of players felt the blue teams were more skilled and 34.72% of players felt that the teams were evenly matched in each respective game. Due to the missing answer issue mentioned in section 8.1, 1.39% is left unallocated to any answer.

Compared to the answers for the same question following the control games, the average percentage of players that felt the red teams had better players is 20.78% less, the average percentage of players that felt the blue teams had better players is 1.33% less and the average number of players that felt neither teams had better players increased by 20.72%.

When examining the correlation coefficients, there is once again a positive correlation between the player opinion on the stronger team and the team's average K/D ratio, even higher than in the control games. The correlation coefficient between the number of answers for the red team being more skilled and the red team's K/D ratio is 0.831, up 0.051 from the same coefficient in the control games, while the correlation coefficient between the number of answers for the blue team being more skilled and the blue team's K/D ratio is 0.922, up 0.185 from the same coefficient in the control games.

One interesting observation about these results is that there is a difference of only 0.008 between the average team balance value of the control games and the average team balance value of the procedural games, despite the sizeable increase in the number of players that felt neither team was more skilled. Interestingly as well, the correlation between player opinion on team skill and the K/D ratios of the teams increased, quite significantly in the case of the blue team seemingly indicating players were better at identifying the relative skills of the team. This could be because of the order

of the games making players more experienced with the game in the procedural games, as proposed earlier in regards to the correlation between the balance value and team balance value. It could also be that player opinion on teams was being influenced by the game experience as a whole, with the balancing provided by the map making players feel as though the teams were more balanced when in fact they remained the same.

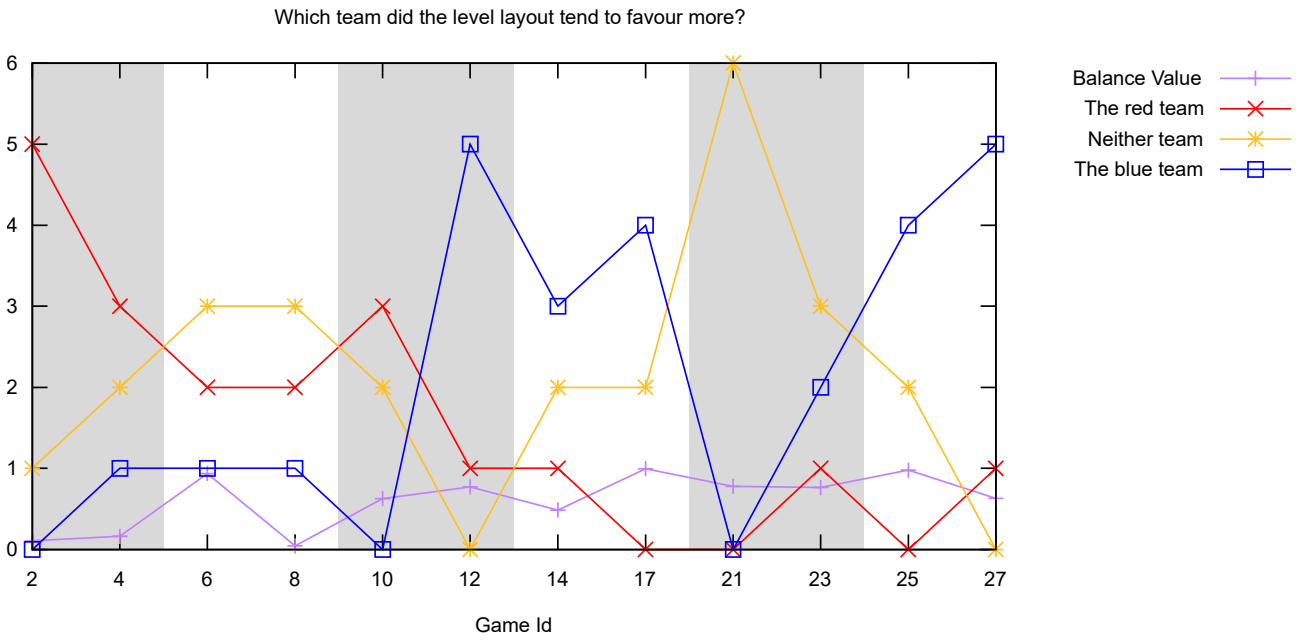

Figure 8.9: A graph showing the answers given to the second question in the questionnaire and the balance value for the procedural games

	Balance Value	The Red Team	Neither Team	The Blue Team
Mean	0.607	1.58	2.17	2.17
Median	0.698	1	2	1.5
Maximum	0.997	5	6	5
Minimum	0.0429	0	0	0
Standard Deviation	0.338	1.51	1.59	1.95

Table 8.7: Statistics for the balance value and the answers given for the second question for the procedural games

The answers given for the second question (shown in Figure 8.9 and Table 8.7) show that, on average, 26.39% of players felt the level layout favoured the red teams, 36.11% of players felt the level layout favoured the blue teams and 36.11% of players felt the level layout favoured neither of the teams. Once again, these figures were rounded to two decimal places and the 1.39% left

unallocated to any of the answers is due to the issue mentioned in Section 8.1. Compared to the answers to the same question in the control games, the answers here are much more “spread out” with a much higher standard deviation for each answer. In addition the “red team” and “blue team” answers have much higher medians and maximums, while the “neither team” answer has a much lower median and minimum. The percentage of players that felt the level layout favoured the red teams is 25% higher than in the control games, the percentage of players that felt the level layout favoured the blue teams is 31.94% higher than in the control games and the percentage of players that felt the level layout favoured neither of the teams is 58.33% lower than in the control games. This shift in opinion on which team the level layout favoured is expected given that the procedural games directly attempt to bias the level layout towards a particular team.

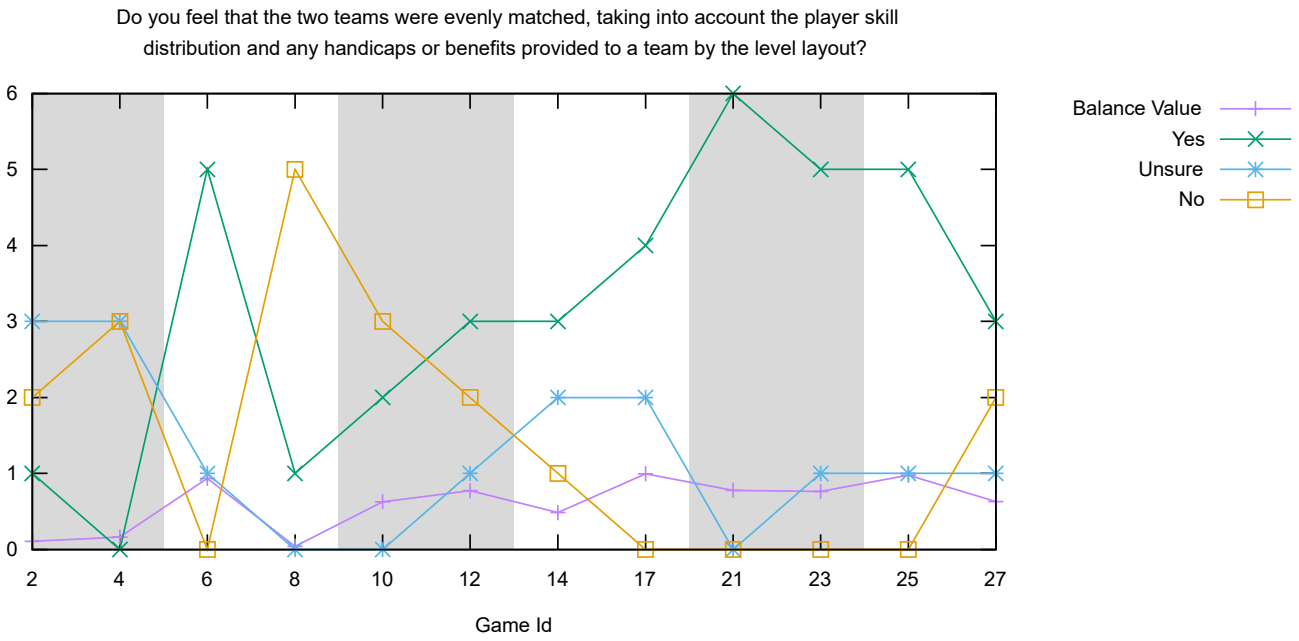

Figure 8.10: A graph showing the answers given to the third question in the questionnaire and the balance value for the procedural games

	Balance Value	Yes	Unsure	No
Mean	0.607	3.17	1.25	1.5
Median	0.698	3	1	1.5
Maximum	0.997	6	3	5
Minimum	0.0429	0	0	0
Standard Deviation	0.338	1.90	1.06	1.62

Table 8.8: Statistics for the balance value and the answers given for the third question for the procedural games

The answers to the final question for the procedural games (shown in Figure 8.10 and Table 8.8) showed that the majority of players felt that the games were balanced. On average, 52.78% of players felt that the games were balanced, 25% of players felt that the games were not balanced and 20.83% of players were unsure. Again, percentages were rounded to two decimal places and the missing 1.39% from the total answers is due to the issue mentioned in section 8.1. Compared to the same question in the control games, on average 29.17% more players felt the games were balanced, 19.44% less players felt the games were not balanced and 11.11% less players said they were unsure.

A much stronger correlation exists between the answers given here and the balance value. The correlation coefficient between the number of answers that the game was balanced and the balance value is 0.857, up 0.518 from the same correlation in the control games, while the correlation coefficient between the number of answers that the game was not balanced and the balance value is -0.8, down 0.698 from the same correlation in the control games. The correlation is also a lot stronger between the answers for this question and the team balance value. Between the number of answers that game was balanced and the team balance value, the correlation coefficient is 0.835, up 0.298 from the same correlation in the control games and between the number of answers that the game was not balanced and the team balance value, the correlation coefficient is -0.761, down 0.183 from the same correlation in the control games.

8.4 Conclusions

The procedural games achieved notably higher balance values than the control games, with an average increase of 0.421. 8 out of 12 of the procedural games were balanced according to the definition set out in Section 5.3, compared to just 1 out of 12 of the control games. Player opinion also seems to support the procedural games were more balanced, with 29.17% more players on average believing that the procedural games were balanced compared to the control games and 19.44% less players believing that the games were not balanced.

9 Discussion

9.1 Spawn Killing

An oversight during the implementation meant that players were not only able to be killed as soon as they spawned, but that there was also nothing stopping the opposing team from approaching and/ or entering the team's spawn point to do so.

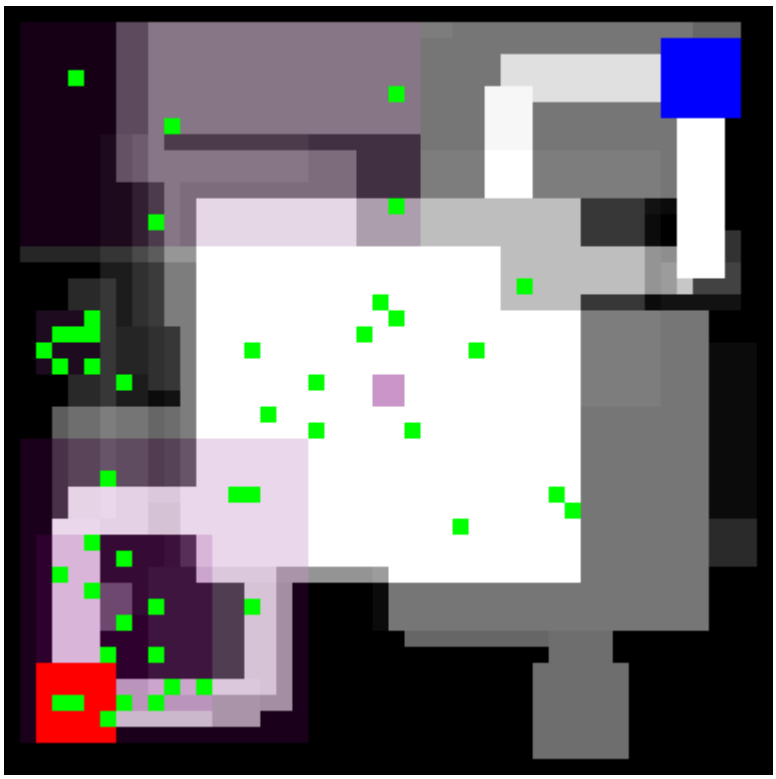

Figure 9.1: The procedural map for Game 2 with the locations where red team players died marked in green

Figure 9.2: The procedural map for Game 12 with the locations where blue team players died marked in green

Examining game 2, a game in which the blue team had a significant skill advantage with an average K/D ratio of 2.94 compared to the red team's 0.34, several deaths of the red team occurred within close proximity to, or even within, the red team's spawn area. This can be seen in Figure 9.1, a composite image made from each map generated during the game overlaid on top of each other. Such kills can become disorienting for the player, as they are randomly spawned within the area only to immediately come under fire. Figure 9.2 shows the same problem occurring for the blue team.

Another flaw with the procedural generation visible in Figure 9.1 is that, due to the fitness function being based solely around the strategic resource value function, where significant skill differences between the teams were present maps would often be generated where the capture point was located on top of the base of the weaker team. Not only did this exacerbate the issue of team killing, it also made it incredible difficult for the stronger team to capture the point. While this did have the effect of providing an advantage to the weaker team, it could be argued that such an advantage was too heavy handed and unfair on the stronger team.

10 Conclusions and Recommendations

This study set out to prove or disprove the hypothesis that continuous reactive procedural level generation is an effective method of providing real time balance in a multiplayer team-based first person shooter. As the results of the testing show, the procedural games showed a marked improvement over the control games in terms of their balance value, with 66% of the procedural games being balanced compared to just 8.3% of the control games. Direct player feedback also indicates that players found the procedural games more balanced, with an average of 52.78% of players saying they felt the two teams were evenly matched in the procedural games and just 25% saying they felt the two teams were not evenly matched, compared to 23.61% of players saying they felt the two teams were evenly matched in the control games and 44.44% saying they felt the two teams were not evenly matched.

Two of the key issues faced during this study were development time and network problems. Development took a lot longer than initially expected due in part to other commitments and also due to underestimation of the work involved in producing the game. While the initial framework was relatively quick to set up, a lot of bugs regarding network synchronisation and the procedural generation itself had to be fixed. Several network bugs were not identified until testing was supposed to begin, which pushed back the testing schedule in order to fix them. A wiser approach in future might be to run the experiments over a local area network rather than over the internet to reduce the likelihood of disconnections and latency related synchronisation issues, which as mentioned previously caused three games to have to be repeated during testing.

A problem in this study that has been previously touched upon is the team balance equation used to evaluate the relative skills of the team. It only considers kills as a measure of player skill when there are a plethora of other metrics that could be used, such as accuracy, kill assists, captures etc. In addition, player skill was calculated on a per-game basis when it may have been wiser to track the player skill across multiple games and sessions to build a better picture of each player’s skill. Doing this would also then allow selective matchmaking to purposefully make teams of certain skill levels, which could provide interesting research data.

The work in this study represents a minimal, stripped down first person shooter in which many features common to the genre, such as multiple weapons, item pickups and healing were removed.

Adding these features in could have a large impact on the effectiveness of this study's approach and, indeed, steer it in new directions. For example, if weapons with different ranges were added to the game, the fitness function could be modified to also attempt to balance the engagement distances. If the stronger team was using exclusively long range weapons, the levels could be steered towards having fewer open areas and more confined corridors.

To conclude, though the work presented in this study has its limitations and the game developed is a very minimalistic example of the genre, it does provide evidence that continuous reactive real-time procedural level generation can have a noticeable impact on the balance of games.

11 Sources

Agarwal, R. and Umphress, D., 2008. Extreme programming for a single person team. *Proceedings of the 46th Annual Southeast Regional Conference on XX,* pp.82-87. Auburn, CA, USA, 28-29 March 2008. New York: ACM New York.

Angularsen, 2015. *Simple insecure two-way "obfuscation" for C#.* [stack overflow] Available at: <https://stackoverflow.com/revisions/26518496/4> [Accessed 02 May 2018]

Beasley, D., Bull, D. and Martin, R., 1993. An Overview of Genetic Algorithms: Part 1, Fundamentals. *University Computing*, 15(2), pp.56-69.

Blickle, T. and Thiele, L., 1995. A Comparison of Selection Schemes used in Genetic Algorithms. [online] Available at: <http://www.tik.ee.ethz.ch/file/6c0e384dceb283cd4301339a895b72b8/TIK-Report11.pdf> [Accessed 06 April 2018]

Booth, M., 2009a. *COUNTER-STRIKE to LEFT 4 DEAD: Creating replayable Cooperative Experiences.* [online video] Available at: <http://www.gdcvault.com/play/1422/From-COUNTER-STRIKE-to-LEFT> [Accessed 13 October 2017]

Booth, M., 2009b. *The AI Systems of Left 4 Dead.* [online presentation] Available at: <http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf> [Accessed 13 October 2017]

Bunjabin, 2015. *Thesis FPS - Voting and Gameplay*. [video online] Available at: <https://www.youtube.com/watch?v=V6eMktY1Sso&feature=youtu.be> [Accessed 24 November 2017]

Cardamone, L., Yannakakis, G., Togelius, J. and Luca Lanzi, P., 2011. *Evolving Interesting Maps for a First Person Shooter.* [online] Available at: <http://julian.togelius.com/Cardamone2011Evolving.pdf> [Accessed 31 October 2017]

Carr, R., 2018?. *Simulated Annealing.* [online] Available at: <http://mathworld.wolfram.com/SimulatedAnnealing.html> [Accessed 18 February 2018]

Collins English Dictionary, n.d.. *Definition of 'annealing'*. [online] Available at: <https://www.collinsdictionary.com/dictionary/english/annealing> [Accessed 18 February 2018]

Counter-Strike Wiki, 2018. *Matchmaking.* [online] Available at: <http://counterstrike.wikia.com/wiki/Matchmaking#Skill_groups> [Accessed 04 May 2018]

Cube 2 Sauerbraten, 2013?. *Cube 2: Sauerbraten – Command Reference*. [online] Available at: <http://sauerbraten.org/docs/game.html#bots> [Accessed 12 January 2018]

Geltman, K., 2014. *The Simulated Annealing Algorithm.* [online] Available at: <http://katrinaeg.com/simulated-annealing.html> [Accessed 18 February 2018]

Hu, X., 2006. *PSO Tutorial.* [online] Available at: <http://www.swarmintelligence.org/tutorials.php>

Hullett, K. and Whitehead, J., 2010. Design Patterns in FPS Levels. *Proceedings of the Fifth International Conference on the Foundations of Digital Games,* [e-journal], pp.78-85. Available at: <https://dl.acm.org/citation.cfm?id=1822359> [Accessed 13 October 2017]

Karavolos, D., Liapis, A. and Yannakakis, G., 2017. Learning the patterns of balance in a multi-player shooter game. *Proceedings of the 12th International Conference on the Foundations of Digital Games,* [e-journal]. Available at: <https://dl.acm.org/citation.cfm?id=3110568> [Accessed 13 October 2017]

Kennedy, J. and Eberhart, R., 1995. Particle Swarm Optimization. In: IEEE (Institute of Electrical and Electronics Engineers), *1995 IEEE International Conference on Neural Networks*. Perth, WA, Australia, 27 November – 1 December 1995. s.l.: IEEE

Khaled, R., Barr, P., Noble, J. and Biddle, R., 2004?. *System Metaphor in "Extreme Programming": A Semiotic Approach.* [online] Available at: <http://www.orgsem.org/papers/13.pdf> [Accessed 14 January 2018]

Liapis, A., Yannakakis, G., Togelius, J., 2013. Towards a Generic Method of Evaluating Game Levels. In: AAAI (Association for the Advancement of Artificial Intelligence), *The Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Boston, Massachusetts, USA, 14-18 October 2013. Palo Alto, California: The AAAI Press.

Luca Lanzi, P., Loiacono, D. and Stucchi, R., 2014. Evolving maps for match balancing in first person shooters. In: IEEE (Institute of Electrical and Electronics Engineers), *2014 IEEE Conference on Computational Intelligence and Games (CIG).* Dortmund, Germany, 26-29 August 2014. s.l.: IEEE

Newcombe, J., 2016a. *The Genetic Algorithm Framework for .Net.* [online] Available at: <https://gaframework.org/wiki/index.php/The_Genetic_Algorithm_Framework_for_.Net> [Accessed 02 May 2018]

Newcombe, J., 2016b. *Getting Started.* [Online] Available at: <https://gaframework.org/wiki/index.php/Getting_Started> [Accessed 21 April 2018]

Newtonsoft, 2018. *Json.NET.* [online] Available at: <https://www.newtonsoft.com/json> [Accessed 02 May 2018]

Shukla, A., Pandey, H. and Mehrotra, D., 2015. Comparative Review of Selection Techniques in Genetic Algorithm. In: , *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*. Noida, India, 25-27 February 2015. s.l.: IEEE

Soni, N. and Kumar, T., 2014. Study of Various Mutation Operators in Genetic Algorithms. *(IJCSIT) International Journal of Computer Science and Information Technologies*, 5(3), pp.4519-4521

SQLite, 2018?a. *System.Data.SQLite.* [online] Available at: <https://system.data.sqlite.org/index.html/doc/trunk/www/index.wiki> [Accessed 02 May 2018]

SQLite, 2018?b. *Appropriate Uses For SQLite.* [online] Available at: <https://www.sqlite.org/whentouse.html> [Accessed 20 April 2018]

SQLite, 2018?c. *Datatypes In SQLite Version 3.* [online] Available at: <https://www.sqlite.org/datatype3.html> [Accessed 24 April 2018]

SQLite, 2018?d. *About SQLite.* [online] Available at: <https://www.sqlite.org/about.html> [Accessed 04 May 2018]

Stavrinou, A., 2015. *[Tutorial] Procedural meshes and voxel terrain C#.* [online] Available at: <https://forum.unity.com/threads/tutorial-procedural-meshes-and-voxel-terrain-c.198651/> [Accessed 31 March 2018]

Team Fortress Wiki, 2017. *King of the Hill.* [Online] Available at: <https://wiki.teamfortress.com/wiki/King_of_the_Hill> [Accessed 20 April 2018]

Team Fortress Wiki, 2018. Control point (objective). [Online] Available at: <https://wiki.teamfortress.com/wiki/Control_point_(objective)> [Accessed 29 April 2018]

Thorup Ølsted, P., Ma, B. and Risi, S., 2015. Interactive evolution of levels for a competitive multiplayer FPS. In: IEEE (Institute of Electrical and Electronics Engineers), *2015 IEEE Congress on Evolutionary Computation (CEC).* Sendai, Japan, 25-28 May 2015. s.l.: IEEE

Togelius, J., Yannakakis, G., Stanley, K. and Browne, C., 2011. Search-based Procedural Content Generation: A Taxonomy and Survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3), pp.172-186

Trochim, W., 2006. *Nonprobability Sampling.* [online] Available at: <http://www.socialresearchmethods.net/kb/sampnon.php> [Accessed 30 April 2018]

Umbarkar, A. and Sheth, P., 2015. Crossover Operators in Genetic Algorithms: A Review. *ICTACT Journal on Soft Computing*, 6(1), pp.1083-1092

Unity Technologies, 2018. *Merry Fragmas 3.0: Multiplayer FPS Foundation.* [online] Available at: <https://unity3d.com/learn/tutorials/topics/multiplayer-networking/merry-fragmas-30-multiplayer-fps-foundation> [Accessed 20 April 2018]

Valve Corporation, 2008. *Left 4 Dead.* [video game]

Valve Corporation, 2009. *Left 4 Dead 2*. [video game]

Valve Corporation, 2012. *Counter-Strike: Global Offensive.* [video game]

Team 17, 1999. *Worms Armageddon.* [video game]