Southampton Solent University

SCHOOL OF MEDIA ARTS AND TECHNOLOGY

**BSc Computer Games (Software Development)**

**2017/2018**

**James Andrew Moran**

***“Balanced First-Person Shooter(FPS) Level Generator Engine Plugin”***

Supervisor : Mark Bennett

Date of submission : May 2018

Abstract

This report details the process of development, for an engine-plugin (for Unreal Engine 4 (UE4)), that generates a level for a First-Person Shooter, by considering the coefficients of Defensiveness, Flanking and Dispersion, for the placement of Zones (sub-divisions of a level), in a manner that would not favour one side or the other.

This tool was able to be developed for UE4, with access via a custom menu option, allowing the User to define the starting point and dimensions, of the level to generate. Only the Defensiveness and Dispersion coefficients are considered by the Level Generator, when it is assembling the level. Although levels generated by the tool, are mostly traversable, balance for one side or the other, is not clear in the levels generated.

Contents

[Abstract ii](#_Toc512943742)

[Contents iii](#_Toc512943743)

[List of Tables vii](#_Toc512943744)

[List of Figures vii](#_Toc512943745)

[Overview 0](#_Toc512943746)

[Project Specification 3](#_Toc512943747)

[Potential Solutions 3](#_Toc512943748)

[Filling Space 3](#_Toc512943749)

[Random Walks 3](#_Toc512943750)

[Positive Aspects 3](#_Toc512943751)

[Negative Aspects 5](#_Toc512943752)

[Cellular Automata 5](#_Toc512943753)

[Positive Aspects 8](#_Toc512943754)

[Negative Aspects 8](#_Toc512943755)

[Settling 8](#_Toc512943756)

[Positive Aspects 8](#_Toc512943757)

[Negative Aspects 9](#_Toc512943758)

[Wang Tiles 9](#_Toc512943759)

[Positive Aspects 10](#_Toc512943760)

[Negative Aspects 11](#_Toc512943761)

[Considered Development Tools 11](#_Toc512943762)

[Unreal Engine 4 (UE4) 12](#_Toc512943763)

[Positive Aspects 12](#_Toc512943764)

[Negative Aspects 12](#_Toc512943765)

[Unity 13](#_Toc512943766)

[Positive Aspects 13](#_Toc512943767)

[Negative Aspects 13](#_Toc512943768)

[Native C++ Implementation 14](#_Toc512943769)

[Positive Aspects 14](#_Toc512943770)

[Negative Aspects 14](#_Toc512943771)

[Development Tool Comparison 15](#_Toc512943772)

[Native C++ Implementation 15](#_Toc512943773)

[Unreal Engine 4 15](#_Toc512943774)

[Unity 15](#_Toc512943775)

[Overall Aim(s) 17](#_Toc512943776)

[Initial Objectives 18](#_Toc512943777)

[Project Management Approach 19](#_Toc512943778)

[Initial Plan 21](#_Toc512943779)

[Risk Assessment and Evaluation 21](#_Toc512943780)

[Task List 24](#_Toc512943781)

[Work Breakdown Structure (WBS) 25](#_Toc512943782)

[Gantt Chart 30](#_Toc512943783)

[High Level Implementation Planning 31](#_Toc512943784)

[Class Overview 31](#_Toc512943785)

[High Level Pseudocode 33](#_Toc512943786)

[Resource Implications 34](#_Toc512943787)

[Implementation 35](#_Toc512943788)

[Considered Methods for Balancing the Space Filling Algorithm 35](#_Toc512943789)

[Setting-Up a Plugin in Unreal Engine 4 (UE4) 38](#_Toc512943790)

[Balanced FPS Level Generation System 41](#_Toc512943791)

[First Row 45](#_Toc512943792)

[Second Row 46](#_Toc512943793)

[Third Row 47](#_Toc512943794)

[Screenshots from the Implementation 49](#_Toc512943795)

[Improvements to the First Prototype 51](#_Toc512943796)

[Level Generation Heuristics: First Phase 52](#_Toc512943797)

[Phase Change 1 55](#_Toc512943798)

[Phase Change 2 56](#_Toc512943799)

[First Row 57](#_Toc512943800)

[Second Row 58](#_Toc512943801)

[Third Row 59](#_Toc512943802)

[Adding New Wang Tiles to the Set: Second Phase 61](#_Toc512943803)

[Phase Change 1 63](#_Toc512943804)

[Phase Three: Balancing the Placement of Zones 65](#_Toc512943805)

[Considering Zone Defensiveness, Flanking and Dispersion coefficients 65](#_Toc512943806)

[Comparison Between the coefficients of Adjacent Zones 66](#_Toc512943807)

[Phase Change 1 68](#_Toc512943808)

[Software Development Analysis of Classes for the Method Detailed In: ‘Procedural Generation of Balanced Levels for a 3D Paintball Game’ 69](#_Toc512943809)

[Software Development Analysis Change 1 70](#_Toc512943810)

[Reflections and Conclusions 71](#_Toc512943811)

[Evaluation 71](#_Toc512943812)

[Reflections 73](#_Toc512943813)

[First Phase Conclusions 73](#_Toc512943814)

[Second Phase Conclusions 74](#_Toc512943815)

[Third Phase Conclusions 75](#_Toc512943816)

[Overall Reflections 76](#_Toc512943817)

[Future Development 77](#_Toc512943818)

[References 78](#_Toc512943819)

[Bibliography 81](#_Toc512943820)

[Appendix A: Stretch Goals 82](#_Toc512943821)

[Appendix B: Literature Review 84](#_Toc512943822)

[Appendix C: Past Coefficient Descriptions/Example Usage 86](#_Toc512943823)

[MS Paint Pixel-Based Calculations 90](#_Toc512943824)

List of Tables

Table 1: The rules for Conway’s Game of Life 6

Table 2: Risk Assessment and Evaluation of the Project. 21

Table 3: WBS Dictionary for the Project. 25

Table 4: Purchased Literature for the Project. 34

Table 5: How the edge-colours are to match-up against each other, on a percentage-chance basis. 55

List of Figures

Figure 1: 1D-Random Walk. 4

Figure 2: 2D-Random Walk with 2500 Steps 4

Figure 3: Specific Cellular Automata example. 7

Figure 4: An example set of aperiodic Wang Tiles. 10

Figure 5: An example of a pattern created with a different set of Wang Tiles. 10

Figure 6: The top level WBS Diagram. 25

Figure 7: Project Gantt Chart, from MSProject 2016. 30

Figure 8: UML Class Diagram for the project (top-level, initial diagram). 31

Figure 9: Project Top-Level Program Flow Diagram 32

Figure 10: Screenshot of the 'Procedural' menu-option, added to the Edit menu of the UE4-Editor. 38

Figure 11: First implementation of tool's options for generation, shown via a Property Editor dialog-window. 39

Figure 12: Level Generation Bounds Exterior, Unlit. Screenshot from the UE4-Editor. 40

Figure 13: Level Generation Bounds Interior, Lit. Screenshot from the UE4-Editor. 40

Figure 14: Initial version of the encapsulation structure, that the level is to be assembled within. 41

Figure 15: Empty level-grid to be used for the representation of the placement of 'Wang Tiles' (Zones). 42

Figure 16: 11 Wang Tile set screenshot from the UE4-Editor. 43

Figure 17: 11 Wang Tile set screenshot from the UE4-Editor, with numbers and edge-colours for each Wang Tile. 44

Figure 18: Grid with the selected Wang Tiles for the first row. 45

Figure 19: Grid with the selected Wang Tiles for the first and second rows. 46

Figure 20: Grid with the selected Wang Tiles for the first, second and third rows. The traversable areas of the level are shown with green arrows. 47

Figure 21: Screenshot of a generated level in the UE4-Editor. This is without the floor and ceiling of the level present. 49

Figure 22: Another screenshot of a generated level in the UE4-Editor. This is also without the floor and ceiling of the level present. 50

Figure 23: 11 Wang Tile set screenshot from the UE4-Editor, with edge-colours as per the heuristics defined for this phase. 54

Figure 24: A Screenshot of a level generated in the UE4-Editor, with these improvements in place (as well as the use of additional Wang Tiles, for the edges of the level-generation area). 56

Figure 25: The first row of results for a dry-run of this phase's heuristics. 58

Figure 26: The first and second rows of results for a dry run of this phase's heuristics. 59

Figure 27: All 3 rows for the dry run of these heuristics. The traversable paths in the level are shown with the arrows. 60

Figure 28: Screenshot of the 18 Wang Tile set, from the UE4-Editor. 61

Figure 29: Screenshot of the 18 Wang Tile set, from the UE4-Editor. The edge-colours are also shown 62

Figure 30: Screenshot of the set of 22 Wang Tiles, from the UE4-Editor. 63

Figure 31: Screenshot of the set of 22 Wang Tiles, from the UE4-Editor. The edge-colours are also shown. 64

Figure 32: Screenshot of a level generated in the UE4-Editor, considering the Defensiveness and Dispersion coefficients. 67

Figure 33: The UML Class Diagram, that considers Zones as an Area, as well as the Edges of these Zones. 69

Figure 34: Revised UML Class Diagram, for only a Zone class, that interacts with the logic of the Level Generator, without any other classes. 70

Figure 35: A screenshot of a level generated in the UE4-Editor, taking into account the above changes. 73

Figure 36: Altered equation taken from the respective paper, for use in the Level Generator. 87

Figure 37: Example diagram, constructed in paint, keeping to-scale, considering how many pixels scale to the width and height in cm (100cm = 1 UU). 89

# Overview

Considering the competitive play of recent First-Person Shooters (FPS) titles (such as Counter Strike: Global Offensive (CS: GO)). (Valve Corporation, 2012)

It has become apparent, that creating balanced levels for FPS titles, could be considered a project in of itself. That is, levels that would not favour one side or the other, (with advantages and disadvantages for each team, given where they start in a level, that the Players must overcome). For the creation of balanced levels, having a team dedicated to such a purpose seems necessary, with them using a substantial quantity of the overall project’s resources (the game’s), to create balanced levels. (LevelCapGaming, 2014)

This leads to Players having feelings of unfairness, in relation to how they are at a disadvantage from the start of a game (irrespective of Player skill-level). This is in respect to the paths Players can take through the level, along with where they would have to look to find enemies at certain points in a path, or from entrances/exits to/from a path, as well as entrances to other paths. (LevelCapGaming, 2014)

The scope of the project, will be broken down into one main phase (with four Bonus Phases, see Appendix A: Stretch Goals, for these bonus phases):

* Greybox Phase: In this phase, the initial framework for the level generator will be implemented, so that it can produce the geometry for the level (such as the walls, entry and exit points, as well as obstacles to provide cover). The generator will also have to consider other factors too, such as where the chokepoints in the level will be and how props will affect lines of sight within the level. It will also have to consider the vantage points on the level and the routes of navigation, through the level

For the first phase, to generate a balanced level, it is important for the generator to consider various aspects. This is because one would not want the level generator to generator levels, that are unfair to one side or the other, reducing the enjoyment of the game, when played on a level generated by this generator.

Examples of these aspects are as follows:

* Positional Advantage: Spaces where one Player has an advantage over another
  + Gallery: An elevated area parallel and adjacent to a narrow passageway
  + Choke Point: A narrow area with no alternative routes, causing Players to be exposed to engagement as they move through
* Large-Scale Combat: Areas designed to facilitate combat involving large numbers of Players.
  + Arena: An open area or a wide corridor
  + Stronghold: A Confined area with dense cover and limited access points
* Alternative Gameplay: Introduce new elements that break from the established mechanics of the game
  + Turret: An area with a fixed high-powered weapon, where one side has a clear advantage
* Alternate Routes: Create alternatives for the Players, in how they approach the level
  + Split Level: A corridor with an upper and lower section, where those on the upper section can attack those on the lower section
  + Hidden Area: A small area off the main route that contains secrets (such as special items), that Players can obtain
  + Flanking Route: A path that allows Players to gain a positional advantage

(Kenneth M. Hullett, 2012)

# Project Specification

## Potential Solutions

### Filling Space

After the program has defined the space of the level to generate, this space must be filled. The following algorithms have been considered for filling space:

#### Random Walks

These fill space by starting at a given point and then taking steps in random directions (filling the space that has been traversed, along the way). Random Walks can accurately model certain natural processes, such as molecular motion. They are also useful for generating ‘naturalistic’ paths and features in a game level. (Brian Bucklew, 2017)

##### Positive Aspects

* Varied Sub-Types: As a Random Walk can be executed in as many of the 3 dimensions as one would want to (for 1D, 2D or 3D Random Walks), this offers flexibility. Examples of the results of 1D and 2D walks, are shown below:

Figure 1: 1D-Random Walk.

With steps shown between 0 and ~101 X (inclusive).

(James Moran, 2018) [Image captured from JSim software]

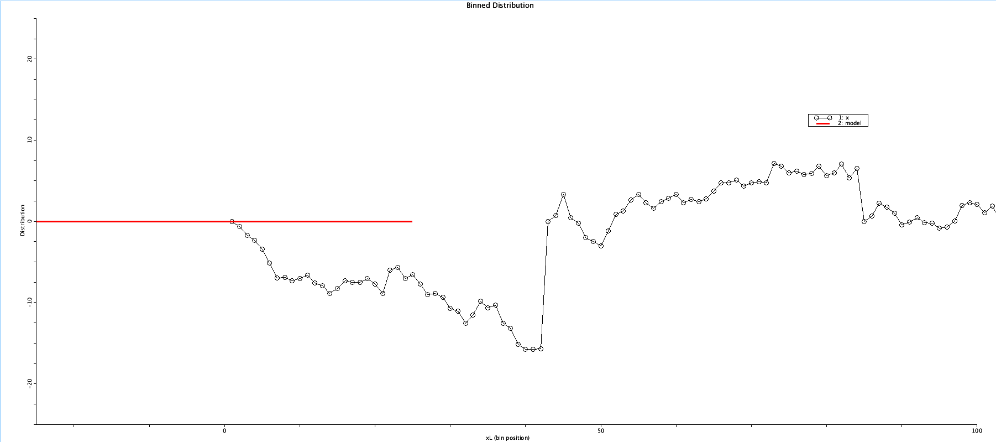


Figure 2: 2D-Random Walk with 2500 Steps

(László Németh, 2013)



* Plausible Randomness: Given the examples provided above, one would find it hard to determine that a procedure was used to generate such output, causing an observer, unbeknownst of the logic for Random Walks, to believe that such an image was created randomly, giving them a sense of discovery (if this algorithm was used as a component in level generation)
* Simple Implementation: The implementation of this algorithm, is relativly simple (given a certain starting point in a generation area, to then move up or down one unit, for 1D-Random Walks (or north, east, south or west for 2D-Random Walks), for a certain number of steps, or until a certain section of the generation area has been filled)

##### Negative Aspects

* Binding to Generation Area (For 2D-Random Walks): As it would make sense to only use this algorithm, to plot a route in a certain area, if ever a step in the algorithm, would move outside of the generation area, that step would have to be discarded. In this case, the direction would have to be determined again, until a step in a direction that is still within the generation area, is taken
* No Consideration of Piece Connectivity: Although this algorithm can establish a route for the ‘pieces’ of a level to be placed on, it would not consider how the pieces are to be connected to each other, or indeed, if a piece connecting to another piece adjacent to it, would be valid (such as a stream length piece, attempting to connect to the back of a corridor wall piece)

#### Cellular Automata

This is a broad category of systems that operate on a graph of discrete cells, where each cell has a state, along with a set of rules that determine how the state of each cell changes, based on the state of adjacent cells (for cardinal directions on the grid). The rules tend to be executed in a step-wise manner (not always though), with each cell computing its current state, based on its neighbouring cells. Then, all the remaining cells, change to their next state as well, one step at a time, following these rules. (Brian Bucklew, 2017)

One example of a cellular automaton, is that of Conway’s Game of Life, which is played out on a 2D grid with two states for each cell, dead or alive. (Brian Bucklew, 2017)

Table 1: The rules for Conway’s Game of Life

(Brian Bucklew, 2017)

|  |  |
| --- | --- |
| **Current State Plus Alive Neighbours** | **New State** |
| Alive, plus 0 or 1 alive neighbours | Dead |
| Alive, plus 2 or 3 alive neighbours | Alive |
| Alive, plus 3 or 4 alive neighbours | Dead |
| Dead, plus 3 alive neighbours | Alive |

After 6 steps through Conway’s Game of Life.

(Michael Cook, 2013)

Figure 3: Specific Cellular Automata example.



##### Positive Aspects

* Simplistic Implementation: As there are only a discrete number of states and rules to account for, which will determine the next state of the cells, on another tick of this algorithm

##### Negative Aspects

* Static Rule Binding: As all the variations of Cellular Automata, use static rules, to determine the result of execution on all of a specific set of cells, this will produce the same outcome each time. Therefore, a system would have to coincide with the chosen Cellular Automata, to generate unique sets of cells, to use in the first instance (to not receive the same result)

#### Settling

This algorithm takes a set of varied shapes that are generated with a certain extent of overlapping. These shapes are then given a simple means to simulate physics, that allows them push away from each other (along with a simple physics rigid body representation). This physics simulation is run, until the shapes are no longer overlapping. (Brian Bucklew, 2017)

This provides a manageable method, to arrange a vast set of pieces, which have different sizes and forms, into a set that is connected, but not overlapping. (Brian Bucklew, 2017)

##### Positive Aspects

* Distribution of Randomly Sized Dungeon Rooms: For a certain set of these rooms, with larger physics collision bounds than the visible extent of the rooms themselves, settling out in an area, then connecting rooms that are not connected to each other uniformly, with a series of hallways. (Brian Bucklew, 2017) This algorithm is well suited to generating a dungeon in this manner.
* Useful for generating a cave system, using randomly and irregularly shaped areas, settling away from each other (starting at the centre point of the area for the level to be generated within) (Brian Bucklew, 2017)
* Useful for using a collection of straight and elbow pieces, settling out from a point, to generate a sewer system (Brian Bucklew, 2017)

##### Negative Aspects

* Uncommon Usage: It appears other space filling algorithms are preferred, as it has not been possible to find a graphical representation (or even a description) of this algorithm, when browsing the World Wide Web (WWW)
* Relatively Complex implementation: It would seem as though there are other algorithms (such as those mentioned here), for filling space, that are simpler to implement

#### Wang Tiles

This is a mechanism for defining a set of tiles, along with the way they connect to each other, side by side, as they are placed one by one, across a plane. One can consider them as squares with a colour for each side, where they are placed so that the colour of each side of the new tile, matches the corresponding colours for the sides of the tiles that have already been placed. (Brian Bucklew, 2017)

It is also possible to put tiles on a plane, via a sub-set of Wang Tiles, along with a specific, carefully selected set of these tiles, so that they will not create a repeating pattern (aperiodic tilling). (Brian Bucklew, 2017)

Figure 4: An example set of aperiodic Wang Tiles.

(Parcly Taxel, 2016)

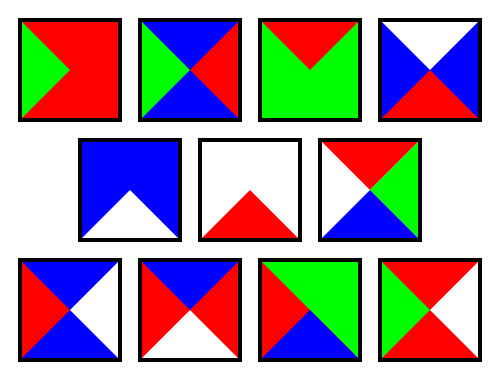
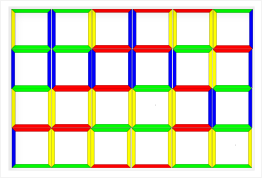


Figure 5: An example of a pattern created with a different set of Wang Tiles.

(Miguel Cepero, 2013)



##### Positive Aspects

* Not Requiring an Undefined Design Process: As Wang Tiles are a defined mathematical process, the use of them is not abstract, versus using ‘typical’ design methods, for image and texture generation, this allows Wang Tiles to also be used in level generation (Sean Barret, 2011)
* Reduction in the Minimum Number of Tiles Required: Via allowing tiles to be rotated or mirrored (rotated 180-degrees), but the edge constraints (colours) on the left and right sides, must be the same as the edge constraints on top and bottom sides (Sean Barret, 2011)

##### Negative Aspects

* Recognisable Patterns: Wang tiles can form long ‘chains’ of repeating colour patterns, even if the overall tiling is irregular (this is due to perception issues with the human eye) (Artificial Mind, 2013)
* Potential Memory Burden: In order to create a plausible pattern of segments in a level using Wang Tiles, one would have to have a vast set of tiles, which could cause the algorithm to use the majority of the computer’s memory (that it was being run on), causing a substantial delay in the generation of the level (Artificial Mind, 2013)

Considering this set of algorithms for filling space, I have decided on using the ‘Wang Tiles’ algorithm, for the generator to fill the predefined space of a level.

This is due to Wang Tiles, putting the level together in a uniform manner, as each tile must connect to its surrounding tiles correctly when placed (considering the edge type/colour of its adjacent tiles). The level will hence, be put together, as if each tile are the pieces of a puzzle, having to correctly match up with their neighbours, to complete the puzzle (I will design the tiles, to allow for this algorithm to generate multiple solutions to generate levels, for a given tile set). (Brian Bucklew, 2017)

## Considered Development Tools

There are many different tools for developing a plugin, that can generate balanced levels for an FPS.

The development tools, with their Positive and negative aspects, are listed below. This is followed by a comparison of these methods:

### Unreal Engine 4 (UE4)

#### Positive Aspects

* Tabbed Interface: With context sensitive segments for each tab, whether that is for the active editor level tab, the Project Settings tab, the Output Log etc. This is used to prepare the project in editor for testing, with output provided during testing, as shown in the Output Log and/or the Message Log
* C++ Backend: For the engine, with a modular hierarchy of classes, for the engine’s features, which can be used as a base for custom classes (in bespoke projects using UE4). One can also tailor a project’s optimisation with C++ (even though the engine can handle certain aspects of memory management, custom classes could overload it)
* Blueprint Visual Scripting System: Used for its utility, to allow for faster implementation then using C++, where it would be deemed suitable to use it (if such implementation in C++, is long-winded, with negligible/no gain in performance over using blueprint, for the same functionality)

#### Negative Aspects

* Blueprint Visual Scripting System: Despite its utility, one must take care not to overuse it, given how it can put an unnecessary strain on the computer trying to run the project. This would then lead to optimisation issues, which in turn, would lead the tool taking longer to generate a level
* Interpretation of Output Messages: Although most messages explain why they have been logged in the Output Log/Message Log, a few messages would seem to be cryptic as per the output they provide, so one must spend time looking into what these messages mean

### Unity

#### Positive Aspects

* Class Hierarchy: Assets inherit from one class at the root level (GameObject), this in turn, inherits from either Monobehavior or ScriptableObject, allowing one to start with an unambiguous base for any given project
* Asset Store: As well as asset packs, one can also find useful tools, that build on the engine’s core functionality, polishing-up a few aspects of the engine
* Community: The community surrounding the use of Unity, are helpful in answering questions about aspects of Unity (if a certain query has not already been answered), along with a robust Application Programming Interface (API) reference

#### Negative Aspects

* Scaling: As projects have additional, as well as more complicated, features added to them, the project management systems scale poorly, finding it harder to manage the project’s hierarchy
* Asset Store: Quite a few asset packages on the store are of sub-par quality, some even falling below that level. One is suggested to vet the packages that one finds on the Asset Store. If a certain asset package is specific to a certain type of game, one should create their own assets for their game (if not matching to the package)

(Mike Prinke, 2016)

### Native C++ Implementation

#### Positive Aspects

* Controlled Implementation: From the start, one can define their own hierarchy of classes, along with interfaces between classes, granting them a custom hierarchy for their project
* Online Community: As C++ has been around for many decades, a vast community has formed around it, with a wide pool of knowledge available for general C++, as well as for certain libraries and any other questions regarding using C++ for a project
* Memory Management: As one must manage memory for a C++ project, this allows for bespoke optimisation, for greater performance of a project (in this case, allowing for levels to be generated faster)

#### Negative Aspects

* Use of Libraries: To implement the same basic functionality as game engines, the standard features of C++, are not able to provide this, causing one to have to learn how to use certain libraries/Software Development Kits(SDK)/API to allow for such features, as is required for this project (as an important aspect of game levels in an FPS, is their visual aspect)
* Time Factor: Even if one knows how to put a solution together, starting from scratch with C++, one should consider the additional time that must be used (versus using a piece of existing software), to implement a solution, with the features required

# Development Tool Comparison

## Native C++ Implementation

After looking over the positive and negative aspects of these tools, using C++ for implementation, has been ruled out. This is primarily due to the time constraint factor, of having to implement this level generator, from the ground-up.

## Unreal Engine 4

Looking at the positive and negative aspects of the aforementioned game engines, although UE4 has been developed and grounded in C++ (with a substantial API, that anyone can look at), allowing for further development of classes and/or modules with C++, a negative (as well as positive) aspect of the engine, is that of the Blueprint Visual Scripting System.

Given the positive and negative aspects of this feature of the engine (noted in the previous section), I would have to take care not to overuse this system, versus C++, for the sake of optimisation.

Other than this and the other noted negative aspect, this engine, would seem a worthy candidate to consider, for implementation.

## Unity

I would not dismiss the use of Unity though, as although C# is used here (given how it is harder to optimise the project with C#), there is also a substantial engine API, providing a base for any project made with it.

As mentioned in the positive and negative aspects section for Unity, I would have to constrain the expanse of the project, given how the structure of Unity, scales poorly to larger projects. But given the scope mentioned for this tool, this should not be an issue (I must simply maintain focus).

Therefore, once again, this is also a worthy candidate to consider, for implementation.

# Overall Aim(s)

Considering this, the overall aim of the project will be that of creating a tool, that generates a level, for an FPS, that has an interior context (such as a cave or an office building), with one main degree of level gradient. In addition, the generator will produce a ‘balanced’ level, factoring in certain aspects for such.

# Initial Objectives

When thinking of some initial objectives for the project, the following come to mind:

* Analyse the project’s goal, to derive requirements (into features), for this Level Generator
* Put together design components for these features, to direct the project (such as UML class diagrams/flow diagrams/pseudocode etc)

# Project Management Approach

The Project will use standard project-management tools and techniques:

* **Work Breakdown Structure (WBS)**
  + **Structured list of tasks**
  + **Organized by phase**
  + **"Living" product backlog**
* **Gantt chart**
  + **Task sequence and progress visualised**
  + **Summary of the current phase**
* **Update regularly and record changes**
* **Supplement with tools you find useful**

(Dave Cobb, 2018)

ADAPTIVE

* **User stories**
* **Product backlog**
* **Iteration planning**
* **Task board**
* **Retrospective**
* **Burndown chart**
* **Velocity chart**
* **Timeboxes**
* **TDD**

PREDICTIVE

* **Task list**
* **WBS**
* **Critical path**
* **Gantt**
* **Dependencies**
* **Baseline**

(Dave Cobb, 2018)

This process is initiated by identifying certain characteristics of the Project, these are:

* The Project’s objective
* When the Project is to have been completed by (scheduling)
* Project Complexity
* Tasks of the Project, the time required to complete these tasks and how one should complete a project task
* Available Resources
* Organisational Structure
* Information and Control Systems

(James Moran, 2017).

# Initial Plan

## Risk Assessment and Evaluation

Table 2: Risk Assessment and Evaluation of the Project.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Risk Description** | **Probability of Occurrence (%)** | **Severity (1-10, 1 = negligible, 10 = catastrophic)** | **Loss Size (Days)** | **Risk Exposure (Probability x Loss Size)** | **Priority (Probability x Severity)** | **Contingency Plan** |
| Following (sample) end-user testing additional features are requested (that were not foreseen). | 70 | 2 | 6 | 4.2 | 1.4 | Put in place a modular system, composed during the initial planning phases, that allows for addition of features to the plugin, on an ad-hoc basis (fluidly). |
| Requirements are found to have not received full definition | 55 | 4 | 10 | 5.5 | 2.2 | Make sure to follow the requirements gathering process thoroughly, to reduce the imprecision of any requirement definitions, if any imprecision is identified. |
| **Risk Description** | **Probability of Occurrence (%)** | **Severity (1-10, 1 = negligible, 10 = catastrophic)** | **Loss Size (Days)** | **Risk Exposure (Probability x Loss Size)** | **Priority (Probability x Severity)** | **Contingency Plan** |
| The project’s deliverables are not finished in the time that was calculated, for how long it should take to finish them. | 50 | 5 | 10 (overtime) | 5 | 2.5 | Making sure to utilise any spare time as effectively as possible (if completing other parts of the project before they are due), as well as allotting suitable leeway, to the time it should take to finish a component of this project, accounting for any delays. |
| Following (sample) end-user testing, more effort on the user guide is required. | 40 | 3 | 4 | 1.6 | 1.2 | Make sure the user guide thoroughly details all aspects of the plugin, as well as the implementation of it in one’s project. |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Risk Description** | **Probability of Occurrence (%)** | **Severity (1-10, 1 = negligible, 10 = catastrophic)** | **Loss Size (Days)** | **Risk Exposure (Probability x Loss Size)** | **Priority (Probability x Severity)** | **Contingency Plan** |
| Software Development Methodology (SDM) deemed insufficient. | 35 | 4 | 20 | 7.0 | 1.4 | Ensure that the SDM utilised meets the expected development practices, for the plugin, considering as many conditions as possible. |
| The project enters an ‘over-budget’ state. | 25 | 6 | 18 | 4.5 | 1.5 | Making sure to accurately identify costs during the planning phases, as well as having an emergency company capital funds account. |
| A power cut occurs during compilation time. | 0.1 | 9 | 40 | 0.04 | 0.009 | Making sure to compile and save as often as possible, as well as backing up the files in multiple locations. |

(Cast Software, 2016).

## Task List

The following tasks come to mind for implementing the level generator:

* Let the user supply bounds for the level (width and height), for the generator to know as to what extent to run the algorithm for
* Implement functionality to the generator, to have it generate a floor for the level (to the bounds)
* Have the generator determine the starting positions, for both teams, on this floor
* Using the starting points as reference, have the generator run the algorithm, to put geometry into the level, for the routes between the two teams and each team’s objective (accounting for choke-points, galleries, sniper positions etc.)
* Following on from this, have the generator put power-up collectibles in place around the level (accounting for what has been generated in the above step)

## Work Breakdown Structure (WBS)

Figure 6: The top level WBS Diagram.



This is followed by the WBS Dictionary:

Table 3: WBS Dictionary for the Project.

|  |  |  |  |
| --- | --- | --- | --- |
| **Level** | **WBS Code** | **WBS Node** | **Definition** |
| 1 | 1 | BalancedFPSLevelGenerator | The work required to implement this Balanced FPS Level Generator. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Level** | **WBS Code** | **WBS Node** | **Definition** |
| 2 | 1.1 | Project Initiation | The work for the initiation phases of this project. |
| 3 | 1.1.1 | Project Idea | Initial formation of the top-level idea for this project. |
| 3 | 1.1.2 | Project Proposal | Reformation of the idea, into a defined starting point for this project. |
| 3 | 1.1.3 | Definition Report | A further level of reformation, to add increased levels of detail, for planning of the project. |
| 3 | 1.1.4 | Progress Report | The final version of this report (going into the greatest level of detail), with a highly refined and clear direction for the project. |
| 2 | 1.2 | Analysis and Design | For the Analysis and Design phases of this project (after the initiation and planning). |
| 3 | 1.2.1 | Decide on a Development Platform | After considering the development platforms for the project, choose and justify a certain solution to use for the implementation of this project. |
| **Level** | **WBS Code** | **WBS Node** | **Definition** |
| 3 | 1.2.2 | Assemble Basic Prototypes | Put together basic prototypes, for the key derived features (user stories), of this system. |
| 3 | 1.2.3 | Infer Requirements | From the user stories and also the prototypes put together thus far, to be put into the system during implementation (on top of merging together the prototypes). |
| 3 | 1.2.4 | Design System | Put together diagrams/pseudocode/program flow, for the system, based on the inferred/derived requirements. |
| 2 | 1.3 | Implementation | For the core implementation of the project’s features. |
| 3 | 1.3.1 | Assemble Project Baseline | For the implementation of the core features of the project. This is to be used as a base, to add complimentary features on top of. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Level** | **WBS Code** | **WBS Node** | **Definition** |
| 3 | 1.3.2 | Merge Prototypes | For merging the prototypes put together in the previous stage, on top of the baseline implementation (for most of the project’s features). |
| 3 | 1.3.3 | Refine Prototypes | Refine the implemented prototypes, to suit any derived features, identified in the later sections of the previous phase. |
| 3 | 1.3.4 | Implement Additional Design Aspects | For the addition of design aspects, that would not succinctly fit in with the features provided by the core baseline, or the merged prototypes. |
| 2 | 1.4 | Testing | For testing of the project (to check it still meets the requirements). |
| 3 | 1.4.1 | Conduct Blackbox Testing | For testing certain features of the project, on if they operate as expected (without concerning one’s self on how they operate). |

|  |  |  |  |
| --- | --- | --- | --- |
| **Level** | **WBS Code** | **WBS Node** | **Definition** |
| 3 | 1.4.2 | Conduct Whitebox Testing | For testing certain features of the project, on how they provide the operation, for these features. |
| 3 | 1.4.3 | Conduct Acceptance Testing | For testing certain features of the project, on the values that are accepted from the user, against the values that should be accepted, or for denying transition in the application, if certain parameters have not been provided. |
| 3 | 1.4.4 | Conduct Full System Testing | For testing on all of the features of the project, to make sure they meet the requirements as expected (given the other forms of testing conducted as well). |
| 2 | 1.5 | Maintenance | For maintaining the system after initial testing. |
| 3 | 1.5.1 | Identify System Bugs | Identify any bugs in the system. |
| 3 | 1.5.2 | Resolve System Bugs | Resolve the identified bugs. |

## Gantt Chart

Figure 7: Project Gantt Chart, from MSProject 2016.

# High Level Implementation Planning

## Class Overview

Considering the steps that have been taken into consideration for the Level Generator, the classes shown in diagram below come to mind:

Figure 8: UML Class Diagram for the project (top-level, initial diagram).



The top-level program flow has been derived from the diagram shown below:

Figure 9: Project Top-Level Program Flow Diagram

## High Level Pseudocode

The pseudocode for the two highest level classes in the hierarchy, is noted here, starting with the upper level of functionality, for the InitialisationInterface class:

1. Show Root Menu to user.
2. Get target Level Dimensions.
3. Get Level-Generation Biases.
4. Initialise LevelGenerationManager with the Level Dimensions and Level-Generation Biases, provided by the User.

Then moving onto the LevelGenerationManager:

1. Initialise Level Bounds.
2. Place a floor within these bounds.
3. Initialise team spawn-points on the level (within the bounds).
4. Initialise the corridors of the level (considering where the spawn-points have been placed).
5. Initialise Power-Up positions on the level (considering the above 2 aspects of the level).
6. Initialise Obstacle positions for the level (considering the above 3 aspects of the level).

The above 6 steps for the LevelGenerationManager, will be affected by the LevelGenerationBiases, defined by the user via the InitialisationInterface.

# Resource Implications

First off, comes the consideration of human resources.

The primary human resource in this project, is myself and so, I will have to strike a balance between working on the project (efficiently, using proper time management techniques, such as sticking to the WBS, Gantt Chart, keeping a log of how I have used my time etc.) and keeping myself contented.

Next, comes the consideration of physical resources.

To acquire the literature for this project, I would have to purchase this literature. Although for certain pieces of literature, I can borrow these from the Mountbatten Library.

For the literature I will purchase, the cost is noted in the table below:

Table 4: Purchased Literature for the Project.

|  |  |  |
| --- | --- | --- |
| **Title** | **Cost (£)** | **Item Payed For** |
| Procedural Generation in Game Design | 41.56 | Yes |

Current Total Project Cost (£): 41.56

# Implementation

## Considered Methods for Balancing the Space Filling Algorithm

Firstly, by looking at a paper entitled ‘Procedural Generation of Balanced Levels for a 3D Paintball Game’, I was able to determine that the coefficients for Defensiveness, Flanking and Dispersion, were key factors, that the system used, as described in the paper. (Raul Lara-Cabrera et al, 2017)

These coefficients will be used for balancing the placement of Zones, within the level-generation area as defined by the User. (Raul Lara-Cabrera et al, 2017)

For the Defensiveness coefficient, di, the following equation is used:

(Raul Lara-Cabrera et al, 2017)

Where the density of objects in that zone (Densityi), is calculated with the following equation:

(Raul Lara-Cabrera et al, 2017)

With δi equating to the object count of zone i and δMAX equating to the highest object count. For example, in the Level-Generator, δMAX = 5 (as there are at most, 5 objects in a Zone), then for a given zone (i), there are 2 objects in that zone, so:

.

For the TotalZoneObjectArea, this is simply the sum of each object’s X-Scale value, multiplied by its Y-Scale value:

. Where ZoneObjectCount is the number of objects (components) present in the Zone, ScaleX is the X-Scale of object-j in the Zone and ScaleY is the Y-Scale of object-j in the Zone. Following through with Zone i having two objects (one with an X-Scale value of 1 and a Y-Scale value of 0.1. Whilst the other has an X-Scale value of 0.1 and a Y-Scale value of 0.9):

.

For the density of the paths to and from that zone (Densitypaths), the following equation is used:

(Raul Lara-Cabrera et al, 2017)

Where γi represents the degree of node i (the degree is the number of edges connected to each node, (The MathWorks, ©1994-2018) in this case though, the number of adjacent zones will be used instead) and δj represents the density of edge j (for ease of understanding, this is simply the number of edges connected to j, from other zones, not including the ‘virtual’ edges of the Level-Generation Area).

An example run-through of these calculations are as follows (using δMAX equating to 5, as in the above sample calculation and a bottom-left corner Zone being considered as the node):

For the flanking coefficient (ki), this equation is used:

(Raul Lara-Cabrera et al, 2017)

Where φi is the number of adjacent Zones to the Zone being considered and γi is the number of zones surrounding the Zone being considered (diagonally and horizontally). For example, a zone is in between 8 other zones (with 4 adjacent zones), so:

.

(Raul Lara-Cabrera et al, 2017)

## Setting-Up a Plugin in Unreal Engine 4 (UE4)

To start off, I initiated the creation of a toolbar plugin, providing me with a base menu structure, that could be used to add an option for using the Balanced FPS Level Generator.

After attempting to have a sub-menu display to the user, via the button added to the top tool-bar of the editor, I decided it would make more sense to extend the editor’s menu listings (up to the top left corner of the default editor window).

For this, I followed a Live Training video, entitled ‘C++ Extending the Editor’, which has shown me how to add custom menu options, to the menu items list (for the ‘Edit’ sub-menu). (Noland, 2015)

This menu item will be added to the custom ‘Procedural’ category, on the ‘Edit’ sub-menu, as shown below. Currently a blank box is shown, as there are no children of the UBaseEditorTool class, to be displayed in this category. This is where the Balanced FPS Level Generator option will be shown:

Figure 10: Screenshot of the 'Procedural' menu-option, added to the Edit menu of the UE4-Editor.



This will then show an Edit Properties dialog for the user to choose constraints/other settings, before initiating the level generation (such as the area to generate a level in and bias of the generation for certain aspects of the level). An example of this is shown below:

Figure 11: First implementation of tool's options for generation, shown via a Property Editor dialog-window.



This currently produces a structure to encapsulate the level, for the respective dimensions, with a light source (point light), at the centre of this structure.

An example of this can be seen to the right:

Figure 12: Level Generation Bounds Exterior, Unlit. Screenshot from the UE4-Editor.

Figure 13: Level Generation Bounds Interior, Lit. Screenshot from the UE4-Editor.

This provides a platform, to generate the level on top of. This process is detailed in the next section.

In forming this encapsulation-object, a source from the UE4 documentation and the UE4 AnswerHub, were used for reference. (Epic Games, 2017), (Digi Labs Dan, 2016)

## Balanced FPS Level Generation System

An example of its use, is detailed in this section. Note that no matter which algorithm is used to fill the area, the set-up shown immediately below, is always used, before the space is filled with the respective space-filling algorithm:

The width and depth for this area, are expressed by the user (in cm). This will encapsulate the volume to generate a level within (filling space from the ‘Generation Area Floor’, to the default height of the level (100cm or 1UU).

Figure 14: Initial version of the encapsulation structure, that the level is to be assembled within.

For the use of Wang Tiles, each Wang Tile (Zone class instance) in this implementation, will take up a volume of 1m3 (1x1x1 UU). Specifically, as shown in the diagram immediately to the right, is this set-up for the placement of these Zones (for a 3x3m area (9UU2)):

Figure 15: Empty level-grid to be used for the representation of the placement of 'Wang Tiles' (Zones).

As there are walls encompassing this area, they are presumed to have a red-edge (matching up to the tiles).

A top-down view (to scale), of all the Wang Tiles in the set, is provided below:

There are 11 Wang Tiles in this set used by the level-generator.

The anchor point of each object on the tile is 5 cm from the edge of that object.

The calculations of the edge-colour, for each of the edges of each wang tile, is provided below (starting at the top left of the above diagram, for WangTile1, moving rightwards for the next Wang Tile, on a row by row basis, with the ranges for the edge-colour as:

Where if

(Omitted) The Geometry for Zones 9 and 11, allow the Player to pass through them, by moving from the west or east to the opposite direction, over the obstacles in the Zone.

Figure 16: 11 Wang Tile set screenshot from the UE4-Editor.

These Edge colours, are represented on the diagram of Wang Tiles, as per the calculations, as shown below (along with a number to identify each tile, when mentioned in a text passage of this document):

Figure 17: 11 Wang Tile set screenshot from the UE4-Editor, with numbers and edge-colours for each Wang Tile.

As per the example grid layout (shown earlier in this section), these tiles will now be placed onto the grid, making sure that they match up to the edges correctly.

If there is more than one tile available, in the set of sub-tiles, that match the current grid position, a random number will be used to decide on which tile to use from this sub-set. This has been generated by Random.org. (Random.org, 2018)

Placement of tiles will begin on the lowest row of the grid, in the first column, moving rightwards. The colour of the geometry on the tiles, has been changed to black (to have them stand out).

### First Row

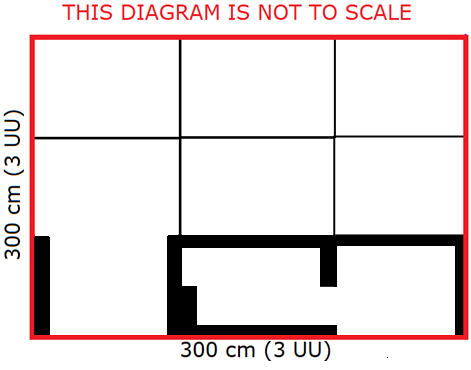
First Column: Valid tiles: 2, 3, 4, 7, 8, 9 and 10. Tile chosen: 10.

Second Column: Valid tiles: 2, 3, 4, 7, 8, 9 and 10. Tile chosen: 8.

Third Column: Valid tiles: 2, 3, 4, 7, 8, 9 and 10. Tile chosen: 4.

The generation area, after selecting tiles for the first row, is shown below:

Figure 18: Grid with the selected Wang Tiles for the first row.



### Second Row

First Column: Valid tiles: 2, 3, 4, 7, 8, 9 and 10. Tile chosen: 7.

Second Column: Valid tiles: 2, 3, 4, 7, 8, 9 and 10. Tile chosen: 4.

Third Column: Valid tiles: 5. Tile chosen: 5.

The generation area, after selecting tiles for the second row, is shown below:

 Figure 19: Grid with the selected Wang Tiles for the first and second rows.

## Third Row

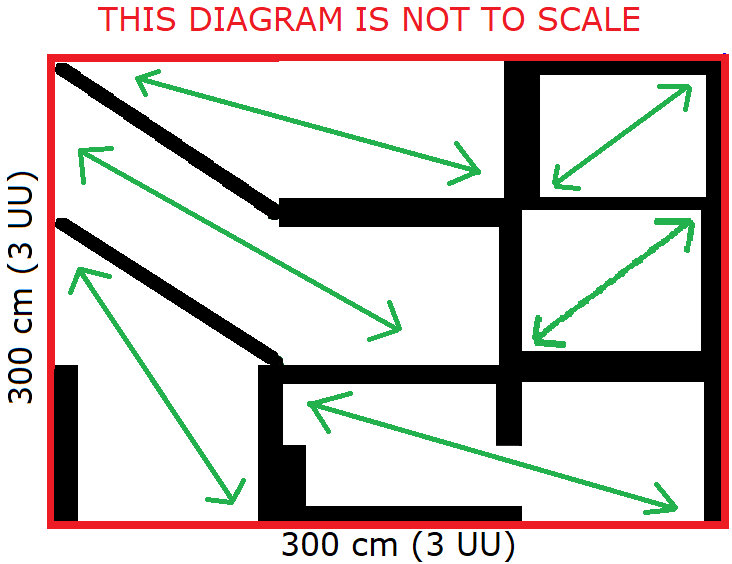
First Column: Valid Tiles: 2, 7, 8, 9 and 10. Tile Chosen: 7.

Second Column: Valid Tiles: 5. Tile Chosen: 5.

Third Column: Valid Tiles: 2, 7, 8, 9 and 10. Tile Chosen: 9.

The generation area, after selecting tiles for the third column, is shown below:

Figure 20: Grid with the selected Wang Tiles for the first, second and third rows. The traversable areas of the level are shown with green arrows.



After completing these sample calculations, it has become clear that the equation for determining the edge-density of a Zone’s Edges, provides inconsistent values. This can result in output, as is shown, in the diagram to the right. This shows a level, where the Player is not able to move between all the Zones, no matter where their Spawn Point is placed in the level. Modifications to the Edge Colour calculations, will have to be undertaken to resolve this issue. Heuristics will be considered for this along with reconsidering the tile design (as in this example, the 11th tile will have never been considered, as there is no configuration of tiles that would allow for the placement of it).

### Screenshots from the Implementation

After resolving certain issues with the plugin’s use of Blueprint assets, these screenshots have been taken of a level. These are to scale (from a top-down perspective, with the floor and ceiling of the level generation area, removed so that the tile geometry can be seen):

Figure 21: Screenshot of a generated level in the UE4-Editor. This is without the floor and ceiling of the level present.



Figure 22: Another screenshot of a generated level in the UE4-Editor. This is also without the floor and ceiling of the level present.



This shows that the steps for the generation of a level, fall into place, but as this is based on the current set of logical steps, it is still subject to the issues identified in the previous section. Rules will have to be put in place, to have the generator put together a level that allows the Player to move between most of the Zones of the level. After this, factors for balancing the level can be considered.

## Improvements to the First Prototype

After reflecting on the improvements, I could add to the first prototype (as detailed in the previous section), I would want to consider the following phases for the next prototype (in this order):

* Using heuristics to determine the edge-colour of a tile, instead of an equation (to make sure the level is mostly traversable)
* Adding new tiles to the set of tiles to use (at least for copies of the tiles currently present in the set, but at a different orientation)
* Considering the Defensiveness, Flanking and Distribution characteristics of a Zone, along with the Heuristics, to ensure a level is balanced for both teams, as well as mostly traversable

To initiate these improvements, will be that of designing with heuristics (for a traversable level, even though, more than likely, not a balanced level).

### Level Generation Heuristics: First Phase

For these heuristics, I will want them to use the Blue edge-colour, to indicate a solid wall in parallel with that edge, that is the closest object in that Zone, to the respective Zone-Edge. This means that the ‘virtual-edges’ of the ‘virtual-zones’ , for the walls that encapsulate the level-generation area (with the respective north, east, south or west ‘virtual-edge’), will have a blue edge-colour, as well as the edges of any Zone, which completely obstruct Player movement (as in, the Player is not able to leave that Zone via that Edge).

This logic will be applied to the previously, ‘lower’ edge-colours (given the range for edge-density values used to determine the edge-colour), for green equating to an edge with chokepoints (either a narrow opening between two Zones, or an obstacle that must be vaulted over), red equating to edge with a wider opening (but not completely clear, or multiple smaller openings and/or minor obstacles that the Player can walk over, experiencing a minor ascent or descent) and grey equating to an edge with no obstructions to Player movement at all (from this Zone to another, via that Edge).

This is presented as the following sequence (for determining the edge-colour, of any edge):

1. Consider not only the extent of objects that are blocking entry into the Zone via that Edge, but also what percentage of the Zone’s total area, a Player is able to access via that Edge.
2. If entry into the Zone via that edge is either completely blocked (with objects taking-up all the area within 1cm to that Edge), and the Player is only able to access up-to 10% of the Zone’s total area, via that Edge, that Edge-Colour will be Blue.
3. If entry into the Zone via that edge is either only accessible via one chokepoint (between 10 and 20cm in width), and the Player is only able to access up-to 25% of the Zone’s total area, via that Edge, that Edge-Colour will be Green.
4. If entry into the Zone via that edge is either accessible via one entry-point 21-80cm in width, or four or more entry points 5-10cm in width, or a barrier high enough so that the Player can just about jump over it and the Player can access 26-75% of the Zone’s total area, via that Edge, that Edge-Colour will be Red.
5. If entry into the Zone via that edge is either completely open (with no objects taking-up the complete area within 1cm to that Edge (so height and width)), or there is an object with a height half or less than half of a Player’s maximum jump height, and the Player is can access greater than 75% of the Zone’s total area, via that Edge, that Edge-Colour will be Grey.

The above heuristics have been applied to the same set of 11 tiles as before (see Figure 16: 11 Wang Tile set screenshot from the UE4-Editor).

This produces the following results:

Figure 23: 11 Wang Tile set screenshot from the UE4-Editor, with edge-colours as per the heuristics defined for this phase.

No calculations have to be performed to determine the Edge colours, as one is simply able to apply the heuristics to tiles, so long as they know the extents of the objects in that zone.

There is also logic for how edge colours match-up to each other, as shown in this table:

Table 5: How the edge-colours are to match-up against each other, on a percentage-chance basis.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Edge Colour to Compare** | **Against Blue Edge (%)** | **Against Green Edge (%)** | **Against Red Edge (%)** | **Against Grey Edge (%)** |
| Blue | 85 | 5 | 5 | 5 |
| Green | 5 | 10 | 75 | 10 |
| Red | 5 | 10 | 75 | 10 |
| Grey | 5 | 5 | 85 | 5 |
| Colourless | 5 | 10 | 75 | 10 |

Colourless, has been added as an Edge-Colour for comparison, as it is used when a Zone will be placed in a position of the level-generation area, that has no adjacent Zone (on the respective side for that Edge), at the moment. These values have been chosen, so that space to move through the level is maximised (comparisons between a Zone’s edge, to a blank slot in the level-generation area, is presumed to be comparing that edge to a grey edge).

#### Phase Change 1

Now for certain Edge-Colours, there is a specific set of tiles that will match up to that set (as follows, north first, clockwise order):

* Blue, Colourless, Colourless and Blue: 1 and 3
* Blue, Colourless, Colourless and Any Colour: 2, 8 and 9
* Blue, Blue, Colourless and Any Colour: 4 and 11
* Any Colour, Colourless, Blue and Blue: 6 and 8

This change was undertaken, so that Wang Tiles placed along the edges and corners of the level-generation area, would not prevent Player access to those parts of the level, instead, providing access to these parts of the level, for the Player.

#### Phase Change 2

There is a new set of heuristics, for the level-generator to use, to produce levels where Edge or corner spaces, will have a pre-defined set of tiles to choose from (for Edges, this set only has one tile in it). An example level, generated with these new rules, is shown below:

Figure 24: A Screenshot of a level generated in the UE4-Editor, with these improvements in place (as well as the use of additional Wang Tiles, for the edges of the level-generation area).



An example of the intended logic, for these heuristics, is shown as a manual calculation (with results), below.

For these sample calculations, Random.org will be used again, along with the set of 11 Wang Tiles. (Random.org, 2018)

The grid as depicted in Figure 15, will also be used.

#### First Row

First Column: Target Edge Colours (for this space, the chosen edge-colours of this space and all other spaces, the values are shown for the north, east, south and west edges respectively):

Blue, Colourless, Colourless and Blue.

Chosen Edge Colours: Red, Red, Red and Grey. Closest matching tile(s): 7. Tile Chosen: 7.

Second Column: Target Edge Colours: Blue, Colourless, Colourless and Red. Chosen Edge Colours: Blue, Red, Red and Grey. Closest Matching Tile(s): 3. Tile Chosen: 3.

Third Column: Target Edge Colours: Blue, Blue, Colourless and Red. Chosen Edge Colours: Blue, Green, Green and Green. Closest Matching Tile(s): 1, 2, 3, 4, 8, 9 and 11. Tile Chosen: 2.

Results for the first row:

Figure 25: The first row of results for a dry-run of this phase's heuristics.



#### Second Row

First Column: Target Edge Colours: Red, Colourless, Colourless and Blue. Chosen Edge Colours: Red, Red, Red and Blue. Closest Matching Tile(s): 3, 6 and 7. Tile Chosen: 7.

Second Column: Target Edge Colours: Red, Colourless, Colourless and Red. Chosen Edge Colours: Red, Red, Green and Red. Closest Matching Tile(s): 7. Tile Chosen: 7.

Third Column: Target Edge Colours: Blue, Blue, Colourless and Red. Chosen Edge Colours: Blue, Blue, Red and Grey. Closest Matching Tile(s): 4 and 11. Tile Chosen: 4.

Results for the second row:

Figure 26: The first and second rows of results for a dry run of this phase's heuristics.



#### Third Row

First Column: Target Edge Colours: Red, Colourless, Colourless and Blue. Chosen Edge Colours: Red, Red, Blue and Grey. Closest Matching Tile(s): 6. Tile Chosen: 6.

Second Column: Target Edge Colours: Red, Colourless, Blue and Red. Chosen Edge Colours: Blue, Red, Blue and Blue. Closest Matching Tile(s): 1. Tile Chosen: 1.

Third Column: Target Edge Colours: Red, Blue, Blue and Red. Chosen Edge Colours: Red, Blue, Blue and Red. Closest Matching Tile(s): 5. Tile Chosen: 5.

The Final Results are shown below. With green arrows showing the accessibility between tiles (for how many tiles one can access from one end of the line, to the other). The lowest accessibility value is 2 and the greatest is 6:

Figure 27: All 3 rows for the dry run of these heuristics. The traversable paths in the level are shown with the arrows.



These results show that with some tweaking, the generator will get to a point quite soon, where it is able to generate a level where Players can traverse most of the level (irrespective of balance). Then with the aspects for a balanced level being taken into consideration, this should allow for a balanced level, that Players are able to traverse most of.

### Adding New Wang Tiles to the Set: Second Phase

For this phase, 7 new Wang Tiles have been added to the set, for alternate orientations of existing Wang Tiles in the set (to account for as many possibilities in traversal that have been foreseen).

The set of Wang Tiles (Zones) is now as follows:

Figure 28: Screenshot of the 18 Wang Tile set, from the UE4-Editor.



With the edge colours as so:

Figure 29: Screenshot of the 18 Wang Tile set, from the UE4-Editor. The edge-colours are also shown



The process of implementing the Blueprints for each of the new Wang Tiles, has been successful and the new Wang Tiles are part of the original set of 11 Wang Tiles, that the system can pick from. Eighteen Tiles are in this new set of Tiles.

#### Phase Change 1

Four new Wang Tiles have been added to the set (as shown below), for wall-tiles lining-up with the Edges of the level-generation area. This increases the size of the set, to twenty-two Wang Tiles:

Figure 30: Screenshot of the set of 22 Wang Tiles, from the UE4-Editor.



With Edge-Colours:

Figure 31: Screenshot of the set of 22 Wang Tiles, from the UE4-Editor. The edge-colours are also shown.



This allows for the Level Generator, to envelop the level-generation area effectively, whilst allow maximum access to the Players, around the edges and corners of the level.

## Phase Three: Balancing the Placement of Zones

### Considering Zone Defensiveness, Flanking and Dispersion coefficients

The equations listed in ‘Considered Methods for Balancing the Space Filling Algorithm’, will be used for this (as described in that section).

For the defensiveness coefficient (di):

. (Raul Lara-Cabrera et al, 2017)

For the density of objects in the zone (Densityi):

. (Raul Lara-Cabrera et al, 2017)

For the density of the paths to and from the Zone (Densitypaths):

(Raul Lara-Cabrera et al, 2017)

For the Flanking coefficient: (Raul Lara-Cabrera et al, 2017)

In that section, Dispersion was not described though, so that will be described here as the median value of the variance and Standard Deviation (σ), for the volume of the objects in the zone. For variance and Standard Deviation, the following equation will be used:

. Where xr is an object’s volume, µ is the Zone’s mean object volume and n is the number of objects in the Zone. If there are n objects in the Zone, with all objects having the same volume, the Dispersion will be 1/n. (Revision Maths, 2018)

An example calculation (for a zone consisting of 3 objects, with volumes (m3) of 0.09, 0.09 and 0.1), is given below:

.

Then as the Dispersion is the median of the Variance and the Standard Deviation (σ),

.

This value will be multiplied by a factor (with a default value of 100), to bring it to a similar level as the defensiveness and flanking values, so the Dispersion in this case, now equals 0.75 (to 2 d. p.). If the Dispersion was 0.013 (to 3 d. p.), then it would be multiplied by 10, instead.

The Defensiveness and Flanking coefficients can only be determined when the Zone is placed, but the Dispersion coefficient can be determined before placement of the Zone (knowing which Zone the Dispersion coefficient is being calculated for).

### Comparison Between the coefficients of Adjacent Zones

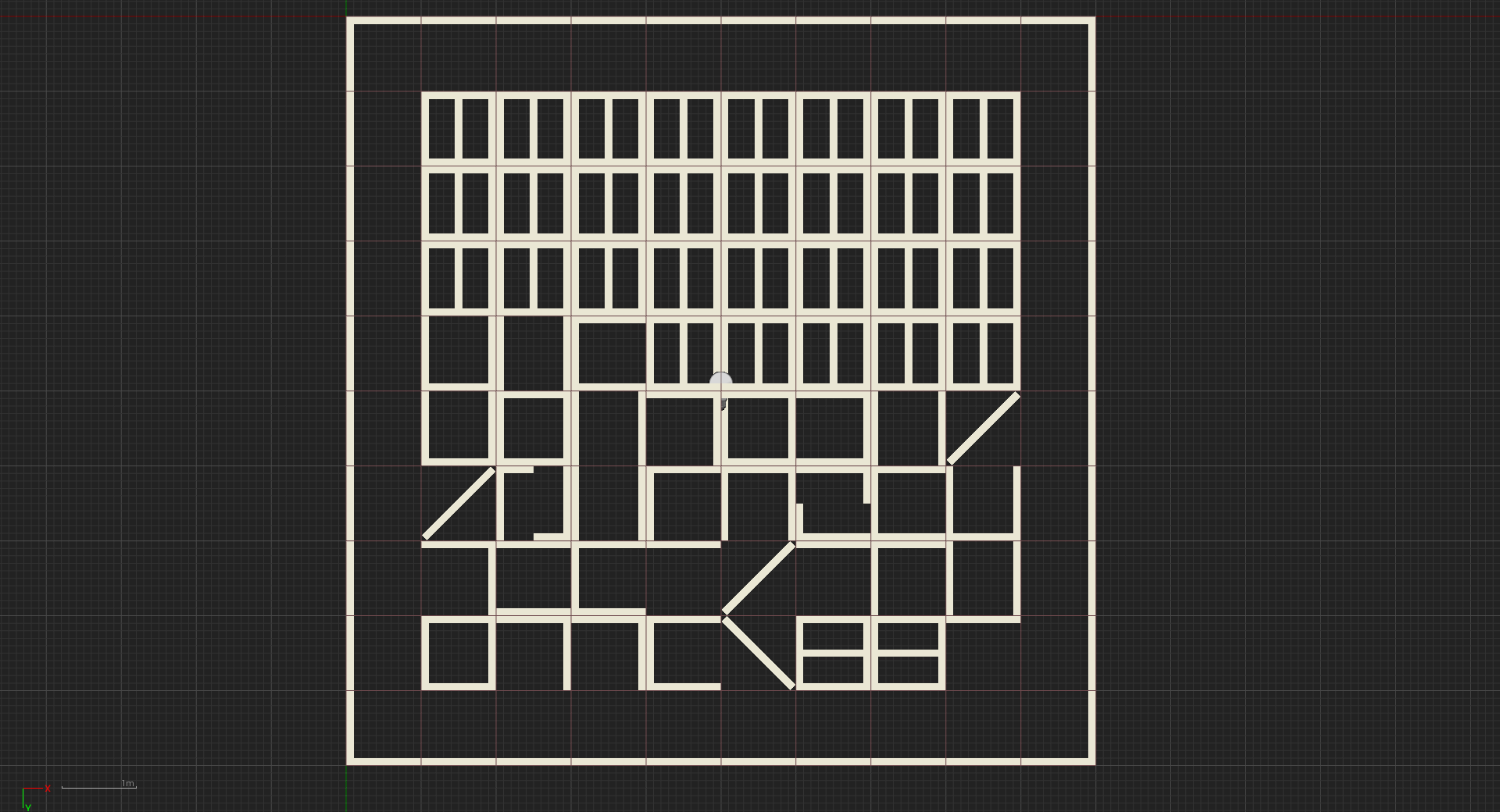
This will start with comparisons between the Dispersion coefficient of the Zones, following this set of rules (first to last):

1. If the placed Zone is one that fits for a corner or edge of the level-generation area, find a Zone to place, that has the lowest possible Dispersion coefficient.
2. If the placed Zone has a Dispersion coefficient of exactly 0.5 find an appropriate Zone to put next to the placed Zone, given that this Zone will be either WangTile2 or WangTile10.
3. If the placed Zone is not a corner, or has a Dispersion coefficient that is not 0.5, find a Zone that will have a lower or higher Defensiveness coefficient, depending on whether the placed Zone’s Defensiveness coefficient is higher or lower than a threshold for balancing the area taken-up, by objects in that Zone.

This moves on to the comparisons between the Defensiveness coefficient against an adjacent placed Zone, using the logic noted below:

1. If the placed Zone has a Defensiveness coefficient greater than the Defensiveness threshold, find a Zone to place, that has a Defensiveness coefficient that is less than the Defensiveness threshold.
2. If the placed Zone has a Defensiveness coefficient less than the Defensiveness threshold, find a Zone to place, that has a Defensiveness coefficient that is greater than the Defensiveness threshold.

An example of a level generated, considering the coefficients, is shown below:

Figure 32: Screenshot of a level generated in the UE4-Editor, considering the Defensiveness and Dispersion coefficients.

Although, the tool would not consider both Zones’ coefficient values (to the west and south of the current position for Zone placement), in Zone placement.

#### Phase Change 1

After attempting to have the Level Generator consider the Zones to the south and west of the current placement position, this results in the Level Generator, choosing the same tile to place in centre of the level-generation area (not along the edges or corners of this area). For an unknown reason, the Level Generator will also skip adding Zones to the south-east corner of the level-generation area, if the level to be generated, has expansive dimensions (starting at 3000x3000cm).

Therefore, the project has been reverted to the previous version, that only considered Zones either to the south or west and not both.

## Software Development Analysis of Classes for the Method Detailed In: ‘Procedural Generation of Balanced Levels for a 3D Paintball Game’

From this considered method for balancing the space filling algorithm, the following classes can be discerned from it:

* Area (with position coordinates, extents and area object density)
* Zone (inheriting from Area, including the zone’s edges)
* Edge (that of the connection point to and from zones, that are close enough to each other)

These can be represented in a Class Diagram (as shown below):

Figure 33: The UML Class Diagram, that considers Zones as an Area, as well as the Edges of these Zones.

#### Software Development Analysis Change 1

There is no requirement for a dedicated Edge and Area class, as Edges are no longer considered by the Level Generator, with coefficients being taken into account instead. Also, the Area class is a redundant base class for Zones, as Zones are the only class that would inherit from that class, when it can simply have what the Area class contains within itself.

This changes the Class Diagram to simply becomes as follows:

Figure 34: Revised UML Class Diagram, for only a Zone class, that interacts with the logic of the Level Generator, without any other classes.

# Reflections and Conclusions

## Evaluation

The project’s overall aims are as follows:

* Creating a tool, that generates a level, for an FPS
* This level has an interior context (such as a cave or an office building)
* This level has one main degree of level gradient
* The generator will produce a ‘balanced’ level, factoring in Defensiveness, Flanking and Dispersion coefficients

These aims will now be considered in turn.

The first aim has been met, as detailed in the implementation section. This section details the creation of a toolbar plugin, in UE4, providing a base menu structure, that could be used to add an option for using the Balanced FPS Level Generator. Selecting this option shows a Property Editor for the Balanced FPS Level Generator tool. After the User has confirmed the options for the generation of the level, they are then able to click on the ‘GenerateLevel’ button, causing a level to be generated at the specified world position to the specified dimensions.

This has been verified by the production of levels using this tool, as is shown in Figures 21, 22, 24, 32 and 35.

All the objectives for this aim have been met.

The second aim has also been met. For example, see Figure 13: A screenshot of the interior of the level generation bounds, lit, from the UE4-Editor. This verifies that this aim has been achieved. All the objectives for this aim have been met.

The third aim has also been met, in that, the level has one main degree of level gradient. Unfortunately, as the screenshots of the level are top-down, this can only be implied. This issue has identified the need for ensuring that time is assigned to create screenshots that show the perspective point of view.

The fourth aim, to produce a ‘balanced’ level, factoring in Defensiveness, Flanking and Dispersion coefficients, has not been met in full. While Defensiveness and Dispersion coefficients are considered, the Flanking coefficient, although calculated, is not taken into consideration. The consequence of this is that a balanced level is produced, using the Defensiveness coefficient, in regard to the amount of cover present in that Zone and, using the Dispersion coefficient, in relation to the distribution of objects in that Zone. But, because the Flanking coefficient has not been taken into consideration, the Level Generator would not factor in the quantity of Zones that will be adjacent to and surrounding the Zone placed at the current position.

The Flanking coefficient was not taken into account by the Level Generator, as my focus was on perfecting the implementation of the other two coefficients.

## Reflections

### First Phase Conclusions

After finishing the implementation for the first of the three phases (for heuristics), it is clear that the chance-values, for how the Edges match up against each other, are too lenient (allowing for combinations of tiles, that result in a level that is not mostly traversable by the Player, with them being cut-off, from what should be a clear path in the level).

This issue has been addressed in Phase Change 1, for this phase, where there are now a specific set of tiles that will be chosen, if a tile is to be placed in a corner or along the edge of the level generation area. This allows for the Player to access more areas of the level.

An example of a generated level using this new system is shown below:

Figure 35: A screenshot of a level generated in the UE4-Editor, taking into account the above changes.

### Second Phase Conclusions

The second phase of the improvements, has turned out as planned, putting together new Wang Tiles for the set of Wang Tiles, in order to accommodate for Zones at a different orientation to some of the pre-existing zones.

Additional Zones for some ‘Filler’ Wang Tiles, have been added to the set, for the pieces of a level to fill in suitable spaces in the level-generation area (such as single wall-piece Zones, for each of the Edges of a Zone, with a wall-piece at the north, east, south and west Edges).

This brings the quantity of the final set of Wang Tiles, to that of 22 Wang Tiles. This is enough Wang Tiles to generate a mostly traversable level for the Player, with as high predictableness, for the combinations of Wang Tiles in a generated level, as is possible.

### Third Phase Conclusions

As the third phase involved the balancing of the generation of the level, using the respective Defensiveness, Flanking and Dispersion coefficients, the use of 2 out of 3 of these coefficients, have been implemented into the Level Generator.

The Flanking coefficient has been left out, due to time limitations. Therefore, only the Defensiveness and Dispersion coefficients are considered by the Level Generator, for the balanced generation of a level.

### Overall Reflections

In general, the project aims have all been achieved, as discussed in the previous section.

For the aim of creating a level that has one main degree of level gradient, although this was achieved, it could not be demonstrated, as the appropriate screenshots were not taken.

For the aim of having a generator that will produce a ‘balanced’ level, factoring in Defensiveness, Flanking and Dispersion coefficients, time constraints meant that the implementation of the Flanking coefficient into the Level Generator, could not be achieved.

The main learning points of this project are:

* Time management should be considered more thoroughly at the start of the project
* The scope of the project could be bounded by the use of appropriate planning tools and methodologies

I could have reviewed more literature prior to commencing the project.

If the project was to be repeated, I would probably choose to use a different game engine for the development of this tool, which would allow for a simpler implementation of the project; such as Unity 2017.

## Future Development

The project has highlighted that there are algorithms available for creating balanced levels, but, implementing these to create an appropriate balanced level for an FPS is not straightforward.

A project such as this, requires understanding of the available technologies and available time; as well as the needs of FPS Players, to enjoy playing an FPS on a level that is balanced for them, no mater which team they are put in.

In its current state, it is my belief that this project is a sufficient foundation that one could use for future development of a plugin for UE4, that could produce balanced levels for an FPS.

# References

AMAZON, 2017. *Procedural Generation in Game Design* [viewed on the 15/02/2018]. Available from: <https://www.amazon.co.uk/gp/product/1498799191/ref=od_aui_detailpages00?ie=UTF8&psc=1>

‘ARTIFICIAL MIND’, 2013. *Introduction to Wang Tiles* [Viewed on the 22/02/2018]. Available from: <http://procworld.blogspot.co.uk/2013/01/introduction-to-wang-tiles.html>

BARRET, S., 2011. *Herringbone Tiles* Silver Spaceship Software: Sean Barret [Viewed on the 22/02/2018], [unfinished report]. Available from: <http://nothings.org/gamedev/herringbone/herringbone_tiles.html>

CAST SOFTWARE, 2016. Software Development Risk Management Plan With Examples [viewed 05/12/2017]. Available from: <http://www.castsoftware.com/research-labs/software-development-risk-management-plan-with-examples>

CEPERO, M., 2013. *Pattern Generated With Wang tiles* [Digital Image] [Viewed on the 22/02/2018]. Available from: <http://3.bp.blogspot.com/-2e9HBeqlGAs/UOXtFXOSWyI/AAAAAAAABYs/8yQglXXZ-Og/s1600/tiles10.png>

COBB, D., 2018. *Computer Games SD Computer Games Indie Final Major Project*. Southampton: Southampton Solent

COOK, M., 2013. Cellular Automata Cave Generation [Digital Image] [Viewed on the 21/02/2018]. Available from: https://cdn.tutsplus.com/gamedev/uploads/2013/07/gdt-sim6.png

CRC PRESS, 2017. *Level Design: Processes and Experiences* [viewed on the 05/12/2017]. Available from: <https://www.crcpress.com/Level-Design-Processes-and-Experiences/Totten/p/book/9781498745055>

HULLET M. K., 2012. *The Science of Level Design: Design Patterns and Analysis of Player Behavior in First-Person Shooter Levels*. University of California

Santa Cruz: University of California [viewed on the 05/12/2017]. Available from: <https://users.soe.ucsc.edu/~ejw/dissertations/Ken-Hullett-dissertation.pdf>

IT KNOWLEDGE PORTAL, 2017. Software Development Methodologies [viewed 09/12/2017]. Available from: <http://www.itinfo.am/eng/software-development-methodologies/>

LEVELCAPGAMING, 2014. *FPS Level Design – LevelCap Bashes BF4 Map Design* [viewed 02/12/2017]. Available from: <https://www.youtube.com/watch?v=FN7iLKUR8eY>

NEMETH, L., 2013. *File:Random walk 2500.svg* [Digital Image] [Viewed on the 21/02/2018]. Available from: <https://commons.wikimedia.org/wiki/File:Random_walk_2500.svg>

MORAN, J., 2017. *CGP504 – Project Development and Project Preparation*. SOUTHAMPTON SOLENT UNIVERSITY, CGP504, BSc Computer Games (Software Development), 2nd Year. Submitted online via Solent Online Learning. Available at: <https://learn.solent.ac.uk/>

MORAN, J., 2018. 1D-Random Walk Sample Output.png [Captured with Snipping Tool]

PRINKE, M., 2016. *What are the pros and cons of Unity?* [Viewed on the 03/02/2018]. Available from: <https://www.quora.com/What-are-the-pros-and-cons-of-Unity>

SHORT, X. T., T. ADAMS, B. Bucklew et al, 2017. *Procedural Generation in Game Design*. Miami: CRC Press

SPRINGER INTERNATIONAL PUBLISHING AG., 2017. *Procedural Content Generation in Games* [viewed on the 05/12/2017]. Available from: <http://www.springer.com/gb/book/9783319427140>

TAXEL, P., 2016. *File: Wang 11 tiles.svg* [Digital Image] [Viewed on the 22/02/2018]. Available from: <https://en.wikipedia.org/wiki/File:Wang_11_tiles.svg>

VALVE CORPORATION, 2012. *Counter Strike: Global Offensive*. Worldwide [Steam]: Valve Corporation

# Bibliography

EPIC GAMES, 2015. *An Introduction to UE4 Plugins* [Viewed on the 20/02/2018]. Available from: <https://wiki.unrealengine.com/An_Introduction_to_UE4_Plugins>

EPIC GAMES, 2017. *Plugins* [Viewed on the 20/02/2018]. Available from: <https://docs.unrealengine.com/latest/INT/Programming/Plugins/>

FLOYD D. et al, 2015. *Procedural Generation – How Games Create Infinite Worlds – Extra Credits*. [Viewed on the 20/02/2018]. Available from: <https://www.youtube.com/watch?v=TgbuWfGeG2o>

PCG.WIKIDOT.COM, 2014. *Algorithms for Procedural Content Generation* [Viewed on the 20/02/2018]. Available from: <http://pcg.wikidot.com/category-pcg-algorithms>

PERIMOSOCORDIAE, 2008. *How does one get started with procedural generation?* [Viewed on the 20/02/2018]. Available from: <https://stackoverflow.com/questions/155069/how-does-one-get-started-with-procedural-generation>

SHORT, X. T. and T. ADAMS, 2017. *Procedural Generation in Game Design*.[Viewed on the 20/02/2018]. Available from: <https://books.google.co.uk/books?hl=en&lr=&id=-ZcnDwAAQBAJ&oi=fnd&pg=PT17&dq=Procedural+Generation+in+Game+Design&ots=3uMJCt1DOH&sig=-jK1igYqlidrEe3lRVBy2FuS0mw#v=onepage&q&f=false>

# Appendix A: Stretch Goals

The bonus phases (stretch goals), are listed below. These are only to be implemented into the project if there is time left at the end, starting with the first:

* Bonus Phase One: Collision Bounds Phase: This is a bonus phase (a stretch goal), as it is not critical to the purpose of the project and will only receive implementation, if there is suitable time for such, after completing the first phase of the project. After the geometry for the level has been generated, one could import this level, as a mesh into a game project. The problem with that is, the project handling system (e.g. a game-engine), would consider the level asset as one asset, applying a collision box or sphere, that envelops the whole level. For this phase then, the Level-Generator would have to create collision bounds for each piece of geometry it has generated in the first phase (for walls, obstacles, doorways and other entry/exit points to name a few)
* Bonus Phase Two: Texture Phase: This is a bonus phase (a stretch goal), as it is not critical to the purpose of the project and will only receive implementation, if there is suitable time for such, after completing the first phase of the project, as well as the first Bonus Phase. After a level with suitable geometry and collision bounds has been generated, comes that of applying appropriate textures to the geometry, as per the setting of the game’s level (such as clinical, office related textures, for the corridors/cubicles and walls, of an office building)
* Bonus Phase Three: Prop Phase: This is a bonus phase (a stretch goal), as it is not critical to the purpose of the project and will only receive implementation, if there is suitable time for such, after completing the first
* phase of the project, along with the first two Bonus Phases. In this phase, props (either dynamic or static) will be added to the level accordingly (such as chairs, desks, stationary equipment, water-coolers, given an office setting). These can either be destroyed, moved or broken through, to remove them as additional obstacles in the level.
* Bonus Phase Four: Lighting Phase: This is a bonus phase (a stretch goal), as it is not critical to the purpose of the project and will only receive implementation, if there is suitable time for such, after completing the first phase of the project, along with the first three bonus phases. If this phase is not undertaken in the project’s development timeline, an ambient (global) light source, will be applied evenly to the whole level, but if this phase is undertaken, then appropriate lighting for each section of the level, will be generated by the Level-Generator (coming from ceiling lights, lamps, torches, external sources (such as from windows), as well as from certain other props in the level (such as a mobile-light source, provided to one or both teams)

# Appendix B: Literature Review

This is a review of the literature, used in the project to a certain extent.

Procedural Content Generation in Games (Computational Synthesis and Creative Systems) – Noor Shaker, Julian Togelius and Mark J Nelson.

This book covers procedural content generation for games, specifically that of levels (as well as items, quests and other types of content). This book is noted as suitable for undergraduate students, as ‘The authors are active academic researchers and game developers’.

(Springer International Publishing AG, © 2017)

This source will provide me with the relevant theories on procedural level generation, which I can then use as a basis for level generation in this project.

Level design: Processes and experiences – Christopher W. Totten.

This book details the experience of game developers, academics, journalists (as well as others), for their take on level design. Each of these sets of people, provide their perspective on the steps for level design, to create the gamespace for the Player (whether that is a horror environment or a computer-generated level).

(CRC Press, ©2017)

This source will offer me the prerequisite knowledge, for developing an engaging level, that I can then use as a basis, for the properties of a level, that this level-generator must adhere to.

The Science of Level Design: Design Patterns and Analysis of Player Behaviour in First-person Shooter levels – Kenneth Hullett.

This dissertation provides a series of guide lines, for developers to use in the design of First-Person Shooter Levels. This was put together, as the author feels as

though there is no common design pattern, that level design could fit into or be described by effectively. This piece of work also lays the foundation to allow further research into this area of gameplay.

(Kenneth M. Hullett, 2012)

This source will provide me with an in-depth level of detail, into specific design patterns for the levels of an FPS, given extensive detail on the many components to consider in the level of an FPS.

Procedural Generation in Game Design – Tanya X. Short and Tarn Adams.

This book provides a high and low level of depth, into the use of procedural generation in computer/video games. This includes information on the implementation and enactment of procedural generation algorithms in games, including for levels in games.

(Tanya X. Short and Tarn Adams, 2017)

This source will provide me with the knowledge of algorithms, used in procedural generation, to then compare against one another, to find the most suitable for this level generator. It will also provide insight into the implementation, for these algorithms (even for maps/levels in games, as noted).

# Appendix C: Past Coefficient Descriptions/Example Usage

In this paper, the idea of using zones, with edges between the zones (up to a certain threshold) will be utilised. A vector representation, holding three specific groups of values will be used for each zone. First, the coordinates of the zone, then the density of obstacles in that zone and finally, the density of obstacles for edges, are taken into consideration. (Raul Lara-Cabrera et al, 2017)

In order to maintain a balanced map (level), with no clear advantage for either team, Coefficients for defensiveness, flanking and dispersion are used, along with the mean and Standard Deviation of the defensiveness and flanking values of the zones. (Raul Lara-Cabrera et al, 2017)

The defensiveness of a zone is determined by the following factors:

* The density of the obstacles within that zone
* The density of the obstacles between that zone and the nearest zones (on the edges)

(Raul Lara-Cabrera et al, 2017)

Considerations for the flanking Coefficient, are shown on the next page.

To alter how the level is generated, the generator will use a mutation-operator. The mutation operator applies pseudo-random permutations to the values of an individual vector (as noted in the 2nd paragraph of this section), adding to or multiplying by a respective pseudo-random value.

The decision of adding or multiplying is also decided upon by chance, with the same probability. If an individual vector is mutated to such an extent, that it becomes invalid for the FPSLevelGenerator’s requirements, the algorithm will not

consider this vector for the map. The map graph is then recalculated after this mutation, to include new edges between zones, if they become close enough because of it. (Raul Lara-Cabrera et al, 2017)

This ties in with Wang Tiles quite well, as a tile and its edges can represent a zone and its edges. A unit measurement for this project is one Unreal Unit (UU), which equates to 1 metre.

Using the Defensiveness, Flanking and Dispersion values, in an equation, to calculate an ‘Overall Compatibility’ value, comes out as follows:

Figure 36: Altered equation taken from the respective paper, for use in the Level Generator.



(Raul Lara-Cabrera et al, 2017)

Where α, β and Γ are the Coefficients for defensiveness, flanking and dispersion respectively. (µd µk) is the mean and (σd, σk) is the Standard Deviation, of the defensive and flanking values respectively. (Raul Lara-Cabrera et al, 2017)

For the flanking Coefficient (ki), this equation is used:

(Raul Lara-Cabrera et al, 2017)

Where φi is the number of connected zones in the sub graph, created from the adjacent zones to the node being considered, not including that zone itself and γi is the number of zones connected to the node being considered. If this zone has no

connected zones, its flanking Coefficient is zero (for example, a zone is in between 5 other zones, with the top 2 zones only being connected to each other and the centre zone, with the bottom 3 zones only being connected to each other and the centre zone. After taking the centre zone out of consideration, there are two sets of connected zones, so ki = 1 – 2/5 = 3/5 = 0.6). (Raul Lara-Cabrera et al, 2017)

For the edges, edge density (δj) is used to represent the colour of an edge. This is calculated from the volume taken up by objects in the zone (in cm3), divided by the mean distance (in cm) of these objects, from the centre point of the edge. These equations are used for such:

δj is then checked against the range of values used to determine the colour of an edge. Where if

The following diagram demonstrates the use of these equations:

Figure 37: Example diagram, constructed in paint, keeping to-scale, considering how many pixels scale to the width and height in cm (100cm = 1 UU).



## MS Paint Pixel-Based Calculation

749: Depth (80 cm), 551: Width (60 cm)

(Average pixels to cm = ((749 / 80) + (551 / 60)) / 2 = (9.3625 + 9.1833) / 2 = 9.273 (to 3 d. p.).

(308, 21): Centre Edge Point.

Distance ((462, 175), (308, 21)) = .

Distance ((220, 379), (308, 21)) = .

Distance ((214, 606), (308, 21)) =.

Distance ((389, 606), (308, 21)) =

As this value lies between 2500 and 5000, the edge colour for the top edge of this zone, is red.