Balanced First Person Shooter Level-Generator

James Moran

# Overview

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

For context on this issue, there are Players who feel as though specific levels of certain FPS titles, favour one side far more greatly than the other. (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 respects 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 goal 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). Not only this though, as the generator will also have to consider where the chokepoints in the level will be, along with the light/dark areas of the level, as well as how props will affect lines of sight within the level. It will also have to consider the vantage points on the level and all the routes that the Player can take to navigate through the level

For the first phase, to generate a balanced level, it is important for the generator to consider aspects, which could favour one side or the other. This is because <insert reasons here>

Examples of these aspects are as follows:

* Positional Advantage: Spaces where one Player has an advantage over another
  + Sniper Location: A Protected, elevated location that overlooks a segment of the level (re-envision this for a closed indoor area)
  + 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 of 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
  + Vehicle Section: Sections of alternative gameplay, where the Player drives or rides in a vehicle (re-envision this for a closed indoor area)
* 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: An alternate path that allows Players to gain a positional advantage

(Kenneth M. Hullett, 2012)

# Project Specification

## Potential Solutions

### Filling Space

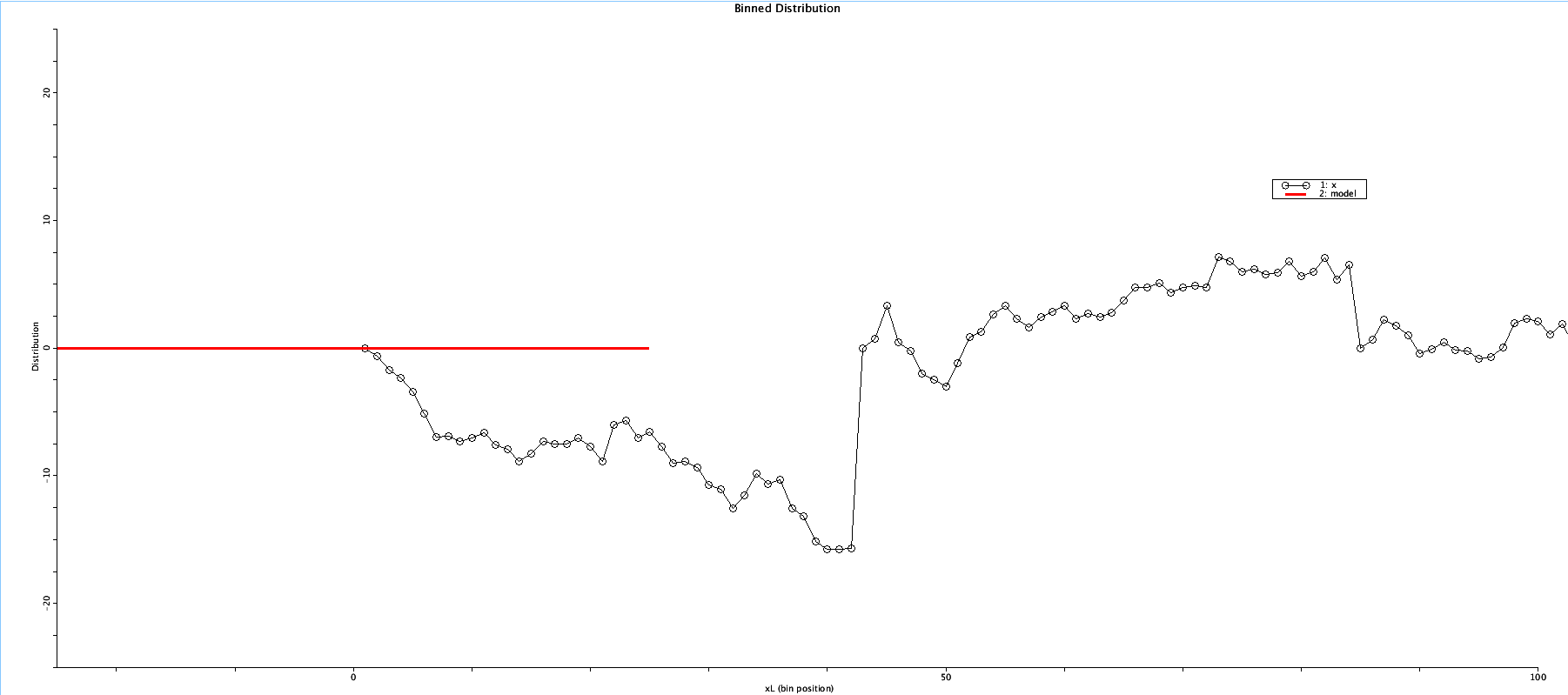
After the program has defined the space of the level to generate, this space must be filled. The following algorithms have been considered for filling space:

#### 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). Randoms 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), examples of the results of 1D and 2D walks, are shown below:

1D-Random Walk, with steps shown between 0 and ~101 X (inclusive). (James Moran, 2018)

2D-Random Walk with 2500 steps. (László Németh, 2013)



* Plausible Randomness: Given the examples provided above, one is not able 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 relative 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 ignored. 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 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, connecting 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)

The rules for Conway’s Game of Life, are as follows:

|  |  |
| --- | --- |
| **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 |

(Brian Bucklew, 2017)

An example, after 6 Cellular Automaton simulation steps. (Michael Cook, 2013)



##### 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, 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 form, 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)
* 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)

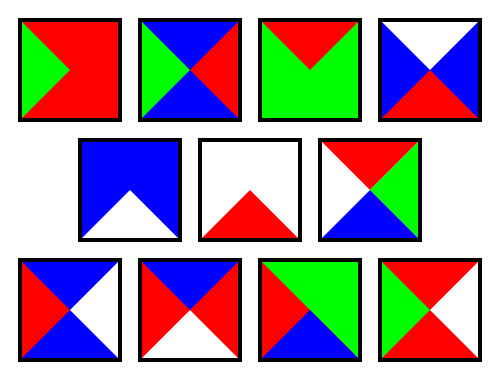
##### Negative Aspects

* 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 through 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 and a specific, carefully selected set of these tiles, so that they will not create a repeating pattern (aperiodic tilling). (Brian Bucklew, 2017)

An example of a set of aperiodic Wang Tiles, is shown below. (Parcly Taxel, 2016)

##### Positive Aspects

##### Negative Aspects

(Brian Bucklew, 2017)

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 will allow the level to be put 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 for a given tile set). (Brain Bucklew, 2017)

<Placeholder>

## Considered Development Tools

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

The development methods, 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 far more time 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 (where certain queries have 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 scales poorly, becoming 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 Method Comparison

## Native C++ Implementation

After looking over the positive and negative aspects on these methods (pros and cons), it is simple enough for me to cancel out, the method for implementing this tool, in C++ as a native C++ application. Although this method can offer greater control over the implementation than using a game engine, one must work from the ground up, even if using a graphical library to aid them (such as DirectX or OpenGL). Therefore, I have removed this method from my consideration of methods to implement this tool, due to the time remaining for this project.

When it comes to the comparing the other two methods (that of using a specific game engine), it appears that no engine of these two, favours the other thus far, for the implementation of this project. I have only noted these two game engines, as these are the game engines I have the most experience in the use of, even though there are other game engines available (e.g. Source Engine, CryEngine, Lumberyard, Construct and GameMaker).

## Unreal Engine 4

Looking at the positive and negative aspects of the aforementioned game engines specifically, 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 from this level), 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 (as this level will take place on, for example, one level of a building, with minor ascent possible via obstacles that the Player can get on top of, if accommodated for in the level-generator). In addition, the generator will produce a ‘balanced’ level, that takes account of various properties for a balanced level, such as movement paths through the level (to objectives or otherwise), choke-points in the level, as well as where Players will want to check for enemies, given the route they have taken through the level.

# 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 techniques. 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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **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. |
| 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. |
| 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

Considering the <chosen> level generation algorithm, 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 (account 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)

The top level WBS Diagram, is shown below:



This is followed by the WBS Dictionary:

|  |  |  |  |
| --- | --- | --- | --- |
| **Level** | **WBS Code** | **WBS Node** | **Definition** |
| 1 | 1 | BalancedFPSLevelGenerator | The work required to implement this Balanced FPS Level Generator. |
| 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. |
| 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. |
| 3 | 1.3.2 | Merge Prototypes | For merging together 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). |
| 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

A screenshot of the Gantt Chart, is shown below:

# 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:

The top-level program flow has been derived from the above 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. Initialise team spawn-points on the level (within the bounds).
3. Initialise the corridors of the level (considering where the spawn-points have been placed).
4. Initialise Power-Up positions on the level (considering the above 2 aspects of the level).
5. Initialise Obstacle positions for the level (considering the above 3 aspects of the level).

The above 5 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 (by taking part in leisure activities, as well as sleeping and eating/drinking sufficient levels of sustenance).

Next, comes the consideration of physical resources.

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

For the literature I have purchased/intend to purchase, the cost is noted in the table below:

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

Current Total Project Cost (£): 41.56

# 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 that I have looked at, as well as that I have not yet looked at.

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).

# 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>

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>

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>

Mike Prinke, 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>

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/>

SHORT, X. T. and T. ADAMS, 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 SOFTWARE,

# 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>