Balanced First Person Shooter Level-Generator

James Moran

# Overview

Having played various First-Person Shooters (FPS), since my early teenage years, as well as having seen the competitive play of more recent FPS titles (such as Counter Strike: Global Offensive (CS: GO)), it has become apparent to me, 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, on an overall basis (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. 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

To clarify, the focus of this project is on the first phase.

Looking closer at the first phase then, it is important for the generator to consider the following aspects, to generate a balanced level:

* 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)

## Considered Development Methods

The are many different methods for developing a tool, 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

* Memory Management: Although having to handle memory management, allows for greater levels of optimisation, at the same time, if memory is not managed properly by the application, this could lead to memory leaks and the application terminating suddenly (in this case, not finishing the level generation process)
* 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)

# 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:

* Consider relevant literature, for algorithms that one could utilise for (procedural) level-generation (literature considered for review and initial lines of research, will be listed in the next section)

# 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

## 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

## Gantt Chart

# 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.

# References

1. LevelCapGaming, 2014. FPS Level Design – LevelCap Bashes BF4 Map Design [viewed 02/12/2017]. Available from: <https://www.youtube.com/watch?v=FN7iLKUR8eY>
2. James Moran, 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/>
3. ©2017 Springer International Publishing AG. Procedural Content Generation in Games [viewed on the 05/12/2017]. Available from: <http://www.springer.com/gb/book/9783319427140>
4. ©2017 CRC Press, Taylor & Francis Group. 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>
5. Kenneth M. Hullet, 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>
6. IT Knowledge portal, 2017. Software Development Methodologies [viewed 09/12/2017]. Available from: <http://www.itinfo.am/eng/software-development-methodologies/>
7. 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>

# Bibliography

1. Procedural Generation – How Games Create Infinite Worlds – Extra Credits. <https://www.youtube.com/watch?v=TgbuWfGeG2o>
2. How does one get started with procedural generation? – Stack Overflow. <https://stackoverflow.com/questions/155069/how-does-one-get-started-with-procedural-generation>
3. Procedural Generation in Game Design – Tanya X et al. <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>
4. Algorithms for Procedural Content Generation – PCG.wikidot. <http://pcg.wikidot.com/category-pcg-algorithms>
5. Plugins – Unreal Engine 4. <https://docs.unrealengine.com/latest/INT/Programming/Plugins/>
6. An Introduction to UE4 Plugins – EpicWiki. <https://wiki.unrealengine.com/An_Introduction_to_UE4_Plugins>