Procedurally Populated Environments with use of Steering Behaviours

Final Report for CS39440 Major Project

Author: Jakub Janas (jaj48@aber.ac.uk)

Supervisor: Dr Christine Zarges (chz8@aber.ac.uk)

4th May 2018

Version 1.0 (Draft)

This report is submitted as partial fulfilment of a BSc degree in Computer Science (G400)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Quality and Records Office (AQRO) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work I understand and agree to abide by the University's regulations governing these issues.

Name: Jakub Janas

Date: 4th May 2018

Consent to share this work

By including my name below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name: Jakub Janas

Date: 4th May 2018

Acknowledgements

I would like to thank my supervisor Christine Zarges for helping me in the development of this project and for patience.

I would also like to thank Neil Taylor, the coordinator of the module for good communication.

Abstract

A Game Development project focused on populating the scene with automotive agents. The project will be developed in Unity3D and use resources available in this engine. The game world will be in 3D space, but with 2D objects. Basic gameplay will involve player fighting with enemies, that will be spawned around the map based on the player's location along with couple other determinants.

Contents

1. Background, Analysis & Process

Procedural Content Generation today is widely used in Game Development. In simplicity it means content created following simple procedures – rules – by an agent, which is a computer. The first digital uses that are commonly known are Rogue (A.I. Design, 1980) and Elite (Braben & Bell, 1984). Both of these games used procedural content generation for generating randomized worlds. This type of use can relieve the game designers of a major part of the work, as it is automating the process of creating a world. The only work to be done is to create an algorithm and assets, such as textures and similar tools that can be used by the computer to generate new, unique content. It allows then to create very large worlds. The best example in this case would be No Man's Sky (Hello Games, 2016), in which the programmers created over eighteen quintillion planets. It is a record in this field of development and probably will not be replicated anytime soon. There is a number of reasons to that, but a major one is that, while the game world was enormous, the generated content actually wasn't interesting enough to be well received by the users. It is arguable that simply generating landscapes – or planets in the above example - cannot be enough to create an interesting game world (in het Veld, Kybartas, Bidarra i Meyer, 2015). Important aspect of every game is storytelling. The most successful single player games are winning over the players with brilliant stories. The other important factor is to make the game world feel like it is not empty.

On the other hand there is a game that became a huge success even before its release, due to an alpha and beta access. This game was Minecraft (Mojang, 2011). The game is quite simple in its form, yet what was a main reason for its success is that it gave players freedom, unlike any other game. The goal is to create, shape the world through building, and survive fighting with enemies and gather food. When creating a new game file, a new unique world is generated based on the supplied seed. The generation process is based on Perlin noise (Fingas, 2015). Perlin noise was originally created for procedural generation of textures (Khan Academy). In Minecraft it is used create something similar to an initial map of the world using the seed. If there is no seed applied, the game will generate its own seed for that world. Seed will always produce same results however, so it is not entirely random. Then the system works on this map to smoothen it out and add biomes. Biomes are different clusters of flora and fauna, for example: plains, forests, jungles and so on. The world can often seem empty, but it is the player's input which makes it special. There are two games from a critically acclaimed series The Elder Scrolls that use procedural generation of the worlds. The first instalment of that series is Arena (Bethesda Softworks, 1994). It is not possible to cross the world map without fast travelling to certain cities, because if the player leaves the town the world is procedurally generated in the game time, meaning that the map size is infinite. The second instalment, Daggerfall (Bethesda Softworks, 1996), also uses this method to create the game world. But rather than generating the world without end, there is a set map size. The actual game map size is 62,394 mi^2 (Plunkett, 2016), which is about half of the size of the British Isles (121,684 mi^2). The power of procedural generation can be really powerful, if used correctly. New projects extend this subject in new directions, such as: creating new uses of procedural generation in case of mechanics, letting the player decide how to generate, a larger contribution in multiplayer games, and many more (Smith, 2014).

1.1. Background

In the case of filling the world with entities there is a method for procedural generation called procedural population. The project focuses on that aspect the most as the author was inspired by the game Left 4 Dead (Valve South, 2008). The game is set in the apocalyptic world where 4 co-operating players are fighting to get to a safe zone. It is a multiplayer game against the AI of the enemies. There is a limited amount of maps that are available and that leads to an issue of making the content interesting. Since this is a multiplayer game the storytelling is not important and relatively few maps create a situation in which players, if they want to play it again, they will possibly play in the same setting. In this case the designers have identified that problem and created a system of procedural population to promote replayability. The AI created for this is called the AI Director (Booth, 2009). The system uses several tools to populate the world. The most basic of these tools is the Navigation Mesh. It divides the map into areas, which each of them is traversable. As mentioned above the goal of the game is to get to a safe zone while battling with the enemies. To track the possible paths from start to finish a useful tool is used called Flow Distance. It uses the areas from the Navigation Mesh to track the possible paths and where team of the players is at the every moment of the game. The players can see only certain parts of the Navigation Mesh based on their position on the map. The last but probably most important is Active Area Set. It is a collection of areas that are surrounding the player. In the process of spawning the areas visible by the players are excluded from the Active Area Set and the rest are used for spawning. When the player team is moving, the areas that are left out, have the enemies despawned out of them, which can be reused in the other areas that are still in the Active Area Set. Each area has a random amount of enemies that is determined at start of the game. When it becomes active, the AI Director spawns enemies to fill it until it reaches either the maximum amount for this area or there are no available enemy objects to reuse. If the area becomes inactive the enemies are despawned. There are also special enemies and bosses in the game. Special opponents will be spawned much less than their simpler versions and in more sophisticated places like above the players, or in front of them and at the back. Bosses are more static, since they are not a subject of regular spawning process but rather set in specific time. The placing of them is also under special rules. There is limited amount of bosses in the game and there can be up to three on the map. They are randomly selected, but it is possible to not spawn a boss at the set interval. There is also a special event that happens at a randomized time called “mob”. The mob is a large (randomized amount) group of enemies that starts to chase the players at spawn. The last thing spawned by the AI Director are weapons. They are however placed at static positions created by the designers. It is to balance the placement of weapons and so that the players know where to look. The AI Director takes into consideration each player’s emotional intensity. It is a numeric value, that is affected by most of the actions in the game, for example: being wounded, killing an enemy etc. If the player is not engaged in combat by an enemy this value starts to decrease over time. The main algorithm of the AI Director consists of four phases. Build Up is when the enemies are spawned and maximum population is maintained. When the combined intensity of each player reaches the maximum, the Director will continue maintaining the maximum population for a short time and then starts to decrease the population to minimum. This will continue until the intensity will fade into a lower value. The minimum population will be maintained for about half a second longer and the Build Up phase will start spawning again. The similar system is suspected to be implemented in Warhammer: The End Times - Vermintide (Fatshark, 2015), but no reliable studies where found to confirm that.

As mentioned before, procedural generation – very often – uses assets or tools already implemented. A substantial research was made into methods of automating the enemies in this project. The main system that was looked into was steering behaviours (Reynolds, Steering Behaviors For Autonomous Characters, 1999). Steering behaviours are algorithms that – when combined – create a force that steers the autonomous agent. An agent in this example is an enemy. To create a group of enemies, that would act and move as a group different steering behaviours were researched. There are different behaviours, for example: seek – calculates a direction to the target point as a vector and normalizes it, applying the speed to create a steering force. This steering force is applied to the agent to move it in the direction of the target. That is the simplest behaviour and other behaviours are using it in different ways to create different forces. Other interesting behaviours are: flee – which is an opposite of seek, it calculates a force away from the target position. Arrive steering behaviour is similar to seek but decreases speed as the agent draws closer to the target. The main interest however were the group behaviours that are the main steering behaviours for flocking. Flocking is a system that was first described in "Flocks, Herds and Schools: A Distributed Behavioral Model" (Reynolds, 1987). This method was inspired by the nature and was described biologist long before. It is based on a natural grouping in the nature like flocks of birds, schools of fish and herds of land animals. All of these phenomenon are similar. Each animal is only aware of its closest neighbours – the number also depends on the sight of animal, for example birds have a wider line of sight since they can look for the neighbours above and below, fish might be swimming in murky waters, which could potentially limit their vision. When the animal is aware of others around it, it can start to correct their flight. There are three rules that apply to the flocking. First is separation. A member of the group shouldn't collide with others, so it needs to maintain maximum distance from them, while still be in the group. Next is cohesion. Member needs to stay close to the center of the group. Not the actual group, but a center of closest neighbours. Last rule is the alignment. The member needs to head into average direction of other closest members of the group, so that it doesn't steer away from the group and separate. When all of these rules are combined, they create a natural movement within the group. Craig Reynolds describes these rules as a steering behaviour in the first paper that was mentioned above (Reynolds, 1999), and continues on in the next paper "Interaction with Groups of Autonomous Characters" (Reynolds, 2000), where he describes the flocking on example of a flock of pigeons. In this paper he used a finite state machine to differentiate two states: walking and flying. When walking pigeons only consider two dimensions: x and y. But when they take up to the sky a third dimension is added to accommodate height. This also expands their line of sight, because they can see other pigeons above and beyond, and when calculating the steering force there is a much more variety than with only two dimensions.

After calculating the steering forces, there needs to be a way to combine them. Different methods for doing that are described in "Programming Game AI by Example" (Buckland, 2005). These methods apply to both single entities and members of the flock. The most basic and obvious way would be to add all of the steering behaviours together, but that may result in an unrealistic movement. That is because there are some forces that needs to be prioritized over others. For example obstacle avoidance is much more important than seeking. It is not a very good behaviour if an agent would go through a wall in order to reach a certain point. Or if an Enemy is low on health it is more desirable to flee than to continue attacking the player. In order to achieve that level of reality, there needs to be a system to manage these steering forces. One way is not to simply add all of them together, but to introduce weights that each behaviour would have. This is called the weighted truncated sum. The system will

have the weights saved in variables that can be changed during runtime, so they would not be static. After summing the weighted forces, the overall steering force is to truncate to the maximum force available for this agent. This method would improve the realisticity of the movement, but there are other methods which can bring even more results. Next method is basing on the weighted truncated sum and is called “weighted truncated running sum with prioritization”. It is similar to the previous, but builds on it. The system would go through each of the steering behaviour in order of the most prioritized to the least. After each calculation of the force, it is checked if the current weighted sum have not exceeded the maximum. If yes, then the method returns it and it is applied to the agent. If not, then next steering behaviours are calculated. Last method described by Buckland is called “prioritized dithering”. It is similar in concept to the previous method, but more extreme. The system goes through each of the forces, but first it calculated the probability of it being calculated this simulation step. It is usually a random generated number, and there is a hardcoded probability - different for each of the steering behaviours – that it needs to exceed in order to pass the check and the force would be calculated then. This last method, although it is possible that all of the forces can be skipped, is the least memory consuming, so it is favoured when a program has to run in a limited environment. On the other hand simply summing all of the forces or using a weighted truncated sum method will be the most processor time consuming.

Reynolds also highlights leader following steering behaviour which can be accommodated to use in flocking as well. This behaviour designates a leader, which all other members of the flock follow. The arrive steering behaviour is used to make them follow that leader. But there is an issue of blindly following a leader, especially where a member blocks its path. To solve that problem, there is an additional check if the current evaluated member is in front of the leader, in a close proximity. If yes, then instead of following him the member temporarily flees from him, until it will be behind the leader again. Leader can steered with the flocking behaviours as well, but it was decided to explore other options. Instead it was chosen to use finite state machine to accommodate leader’s behaviour in the Unity3D.

Finite State Machines were first used by mathematicians before the creation of computers. The most famous case study was a “Turing machine” first described in paper “On Computable Numbers, with an Application to the Entscheidungsproblem” (Turing, 1937). These machines were used to perform a number of logical operations, such as writing or erasing the symbols. This was very useful, however the implementation on digital machines gave much more possibilities. A finite state machine is a system in which a device – speaking in computer terms it is a program or part of the program – that is always in some sort of a state and is able to transition into a different state given the right conditions. This method is very popular in game development as it may apply to many situations. The AI Director described above uses a very strict state machine, which looks like a cycle since there is only one transition from each state. However it still uses powerful tools also described above to achieve population of the game world. Reynolds in his example with flock of pigeons also used a state machine with only two states to switch between two dimensional and three dimensional flocking (Reynolds, 2000). That system is also very strict, it allows only two states, each of them only having one transition as well. It was also researched that Unity’s animation system is implemented in the form of finite state machine. Each animation is one state, and can have as many transition as the designer will decide. There is an Animator window in Unity in which the user can create transitions between the states – states are created with each animation for this animator – but it is also possible to access the animations inside the

scripts. Finite state machines are very useful and can used in many different situations – as large as the game's core logic or system, or small as one object's internal script. They can also work very well in conjunction with each other, creating complex systems with many capabilities. It possible to nest the machines in each other as well.

There was also decision to make what environment to develop the project in. There are many different possibilities as basic as using a programming language, that has a capability to render graphics or there is a library developed to be able to do so. Examples are C++ with OpenGL library (Hewlett Packard Enterprise), Java with Swing (which is not being developed anymore) (Oracle Corporation), JavaFX (Oracle Corporation) or LWJGL (Lightweight Java Game Library). This is not a common approach in modern days, since all of the core systems like physics, colliders and similar have to be developed from the scratch. It is possible, the game Minecraft (Mojang, 2011) mentioned above was initially developed in Java, as a personal project by Markus Persson, who later founded the studio Mojang. However it was decided against using just the programming language as developing what essentially would be a game engine is not a target of the project.

There are many available game engines to use for free or as part of the paid subscription. There are two that are the most notable and approachable. The first one is Unreal Engine (Epic Games, Inc., 1998)It is considered one of the best engines that are used in the industry. The engine was first developed for the game Unreal (Epic MegaGames, 1998), but was later – in 2009 - released as a free to use version, under the name of Unreal Development Kit. It is implemented in C++, and that is also the language that can used in the editor. There used to be another language available to use in Unreal, called UnrealScript, but was later removed, to make room for C++. Other feature is Blueprints Visual Scripting, which is a tool for designers that allows them to basically create code without actually writing it. The other engine is Unity3D (Unity Technologies, 2005). It has become very popular over the past few years, mostly because of its free to use license and a wide range of learning materials on their website and their scripting API. It commonly regarded as inferior to Unreal, but in reality its capabilities are similar. It was first developed for the game GooBall (Over The Edge Entertainment, 2005), which was not very successful. However the developers decided to continue their work on the engine, and make it accessible and affordable for anyone. The primary programming language is C#, but there are two other – UnityScript (Unity's Javascript) and Boo – which are deprecated. It was decided to use Unity3D as an engine for the game, due to the author's prior experience with it and because of the very good manuals of Unity's website. All the methods of developing games in Unity were researched on their official website, Unity Learn (Unity Technologies).

The author was inspired by the games like Diablo (Blizzard North, 1996) and other from the "hack and slash" genre, to develop this project. "Hack and slash" is a type of game that focuses on combat. It was originally used to describe tabletop games, but has since moved to digital gaming. There are many subtypes of this genre, the mentioned Diablo is a dungeon crawler type, but as long as the game is about fighting – melee style usually – the game fits into the criteria. Other major influence was Don't Starve series (Klei Entertainment, 2013) with its use of 2D texture in a 3D world. It was decided to use that style of graphics in the project. The project also focuses on implementing AI systems rather than actual gameplay, since that is not in the scope of academic work.

This project was chosen because of the interest in game development. The author wanted to expand his skills in this area of Computer Science, as both programming skills and actual game development skills. This project will be a valuable asset to the author's portfolio, since it is important to him to work in the game industry after graduating. The AI in game development is one possible approach to achieve that.

1.2. Analysis

The analysis of the researched material brought many conclusions in place. There are number of features that needs to be implemented and the process needs to be decided to correctly carry out all of the actions. Since the game engine, which is the basis for the game, is familiar to the author, it was not a problem to plan how it will be implemented. Unity utilizes a component design pattern in its object oriented approach. This means that every Game Object, as they are called in the engine, has a number of components. These attachments to the original objects are treated by the program as single objects, but in reality they are scripts with many other scripts attached. This also works in the so called hierarchy – each object in the scene is a child of the scene, and objects can be positioned as children of the other objects and so on. All of the objects and components can be accessed in many different ways inside the scripts.

However there were concepts, that the author was not familiar with. Some issues had been caused by the fact that the game is developed as 2D/3D hybrid, and that does not work well as a default in Unity. It is possible though to work around it. Unity has separate colliders for 2D and 3D rendering, but it is possible to accommodate both in one project. Researching a method for that and understanding the processes behind it was crucial, but beneficial. It allowed the author to understand the collision system in Unity in depth, which is a vital knowledge to posses and will be advantageous in future projects. It is also believed all of the concepts used in Unity3D are universal, no matter the actual implementation in this game engine and the skills gained during the research and development of the project can be applied in other systems and areas of computer science.

Important issue was to adapt the systems that were chosen and accommodate them in the Unity's built in systems and settings of the game. As already mentioned above, the 2D/3D hybrid is not an often encountered choice of game developers, so there weren't too many examples on the market to draw inspiration from. There was no games or tutorials found about these type of game that were implemented in Unity3D. The general design of the game have been done based on analysis of the visual data from the "hack and slash" games and Don't Starve series[CITATION KleiEntertainment \l 1045] mentioned before. The procedural population system for Left4Dead[CITATION Valve \l 1045], described in the previous section, was the most adapted system in case of the tools used by the AI Director, as the nature of the game turned out, after analyzing it, to be much different. The general procedure was not changed however as the state machines work well in most environments.

During the analysis and research important features were recognized, that were believed to be crucial to shape the systems within the project. Since the project was developed by only one programmer, the design process of the game very flexible. It was decided to focus on implementing the basic features and the AI systems, rather than creating all of the aspects of the game.

1.3. Process

After analyzing the research outcome for this project, it was time to decide on main features. Functional requirements were created based on the data from the analysis stage.

The most important function is creating a scene environment and a playable character. These two aspects are presented together, because of the setting of the game. To create a 2D/3D game hybrid there is a need for certain rules to applied, such as: if the sprites are used for all of the objects,

the camera needs to be in the right position all the time to correctly capture them. These are also put together because they're not a part of the AI systems and the project focuses on developing these.

The next function is the creation of an Enemy object. It needs a specific set of behaviours that will apply his movement and other functionalities. Choosing the right type of behaviour is very important as it needs to fulfill its duties as an actual enemy of the player. It is also important to bear in mind that this object will be used by other systems in the game, developed at a later stage of the project. There is a question: does this object need to be accessible or does other parts of the game must adapt to the said object. It is crucial to resolve this issues at the planning stage of the project, as it may bear other problems in the future. That said, some problems cannot be predicted and the programmer needs to work around them.

After developing the set of basic systems, it is time to tie together certain parts of them. For example: the collision detection can be implemented now as there are more objects in the game, combat system can be created since there are entities that can fight with the player. All of these aspects need to be adjusted and polished at many stages of the game. The question to ask here is: how these systems will work alongside each other.

Last but not least, the major AI can come into play. Using the tools that were implemented or already built into Unity3D, the AI systems can be developed. It was recognized that the methods chosen for the general AI of the project are not dependent on each other and can be developed in either order.

After describing the features above, it was decided to use the plan driven development for this project. Plan driven development is a waterfall style approach to developing software. Traditionally it is a very strict approach, but as mentioned before since the project is developed only by the author the development style has been modified to allow change of the design at later stages if necessary.

2. Design

2.1. Overall Architecture

This project has been adapted to work in the Unity3D environment. This means that some parts of the project are Unity features. All of the scripts are written in C# and objects can be controlled by them.

All of the objects belong to the scene. Objects can be children of the other objects, and so on. This creates the scene's hierarchy in Unity. All of the objects are called GameObjects in the engine and they inherit from the Object class in C#. GameObjects can have components, which from the programming point of view are separate objects, but in the Unity3D editor they are considered as one GameObject. This allows for creation of complex objects, while maintaining a very tidy architecture. Programmer can also treat children of the GameObject as an addition to it, like a component. However this is just a simplification in the eyes of the developer, because they are treated as separate objects by the software. It also depends on the situation sometimes an object can be there just for the purpose of grouping other GameObjects, but sometimes the design requires objects to be multi level. In this project both of these methods will be used. However, it is worth noting that every object has its transform component. Transform is used to store the position, rotation, scale, parent object and object's children. The parent object is the origin of the local space of the child. If the GameObject is on the top of the hierarchy it is considered in the world space rather than a local space. To summarize, if the object's parent's position will not be on the origin point in the scene, its children will have different local space than the world space.

GameObjects can be saved as assets. When creating an object it is possible to easily reuse it by simply dragging and dropping the object from hierarchy to the assets window. When using this method all of the children will be saved as well. GameObjects saved like this are called prefabs. Prefabs can be anything from single environment objects to the whole buildings with all of the furniture inside. These are very helpful when designing the game world. A designer can create a tree or any other environment object as a prefab. It can later be used by dragging and dropping it from the assets folder into the scene window and allocating it inside that scene. Also, when using a terrain type of object it is possible to "paint" trees and details using prefabs (GameObjects to be specific). They will be automatically added to the scene as a GameObject and positioned at the target position. This method is much easier because it only needs clicking on the terrain instead of the dragging and dropping the prefab from assets.

Assets can be any file that's format is supported by the Unity3D. Most common are textures and scripts, but there are many more available: materials that can be used on mesh filters to apply textures to 3D objects, audio files, animations created in Unity or other external sources. Objects can also be modeled in software like Photoshop[CITATION Adobe \l 1045] or Maya[CITATION Autodesk \l 1045] and used for any 3D gameobject in the scene. It is even possible to create a terrain object in 3D software as a model. These assets can be manually added to the scene or be used in the scripts. For example a texture can be used when instantiating a GameObject or changing materials on a 3D object. This project only uses prefabs inside the scripts.

Unity uses a Canvas system for the UI. Canvas can be rendered in three modes: screen space – overlay, screen space – camera and world space. First two methods are have the user interface rendered on the screen space. Overlay renders it directly on the screen, doesn't matter if there is a camera or not in the scene. Camera rendering mode have the interface rendered in relation to the camera – on a plane parallel to the camera. Last rendering mode is quite unusual as it can be rendered anywhere the designer decides. For example if it was needed part of the interface could be rendered as a plane in any part of the scene as a tablet screen for example. It could even be interactive. This is not a common approach because traditionally the UI is rendered in the screen space. For this project it was decided to use camera render mode, as it is non interactive and is only used to display information for the user.

Unity utilizes many different kinds of input. The input manager in the Unity editor manages it through axes. The designer can create as many axes as it is needed. It is possible then to get the current value of the axis in the script. The value can be anywhere between -1 and 1. Every axis has a negative and positive button. The actual input buttons can be set in the built game, in the configuration screen that pops up when running the game. There are only two built in axes used and these are horizontal and vertical. These are used for movement and by default they are bound to arrow and WASD keys. It is possible to read every button input from the script. There are three methods that can do that: GetButton, GetButtonDown and GetButtonUp. These can read when the button is held down, when it is just pressed and when it is released respectively. These however use the same input as the axes that can be changed by the player. It is possible as well to read keyboard and mouse button and there are separate methods for that. These are used in some scripts in this project.

The design will make use of the described aspects of the Unity game engine to create the game. The only external components are textures and audio files. All the rest has been created inside the Unity editor or in Visual Studio (scripts). The game only uses the internal Unity3D functionality and only Unity's libraries were used. The project makes the best use of what available to create the desired result.

2.2. Detailed Design

The Unity engine is based on the object oriented aspect of C# programming language. It also utilizes the composite design pattern well, as it was already mentioned. Because of that the project will rely on that as well.

The component system in Unity was already explained, but it is important how it is used in the project. Usually GameObject have multiple components except for the grouping objects. Every object starts with a transform and will always have it as it not possible to remove it. It is part of the design and probably a RequireComponent attribute has been used in the GameObject script to ensure that. That attribute can be used by a programmer as well and it comes very handy. It is frequently used in this project.

```
[RequireComponent (typeof(EnemyStats))]
[RequireComponent (typeof(NavMeshAgent), typeof(EnemyAttack), typeof(EnemyHealth))]
```

Figure 1. Attribute from the EnemyBehaviour script

As it can be seen in the *Figure 1*, EnemyBehaviour script requires four components. If the GameObject does not have these attached to itself, Unity editor forbids attaching this script to it. This feature works very well to ensure that the GameObject are true to the design, but also prevents null exceptions from the script if the desired component has not been found.

2.1.1. Environment design

The environment should consist of a flat terrain. The terrain should use a seamless texture of grass. On the terrain there should be environmental objects which would create a forest.

2.1.2. Physics and collisions

Physics and collisions systems are provided in the Unity. There are issues with that however, that were mentioned before. This game project, being a 2D/3D hybrid, needs to utilize both 2D and 3D components that are built in the Unity3D in order to achieve the nature of the game. Using 3D game world, the objects need to have 3D colliders. Colliders in turn need to have mesh renderers and filters to work correctly. It was decided to take advantage of the hierarchy and spread these components onto separate GameObjects. In some cases there will be two children, one of which will have the SpriteRenderer attached to render graphics and the other will have the colliders and meshes. In the case of environment objects it was decided to attach the SpriteRenderer to the parent instead of creating a new GameObject for a single component. Environment object do not have any other components besides the mandatory transform.

2.1.3. Player Character design

The Player character must have several features:

- Movement – It was initially decided to create two types of movement: one would be with the keyboard and the other with mouse. Keyboard movement would be a standard movement via arrow keys of WASD keys. It would change the position according to the

keys pressed. The other movement would take the input from the mouse, which is a point on the screen convert it to the world point on the terrain and slowly move the player by changing the position each frame by slow amount. Due to the fact that the game should run in high amount of frames per second, because it is not very memory consuming, that would create a smooth movement. To both of these movements a variable of speed should be applied.

It was later decided that having two types of movement can change the pace of the game, or can be confusing to the user and the design was modified to remove this method of movement.

- Combat – Since this is a game that focuses on fighting a combat system must be developed for this purpose. There were couple decisions to be made: how many opponents can hit the player at once, how to calculate the damage and how to apply it. The initial intention was to allow the player to hit all of the enemies that are in range. It was decided otherwise, because of the decision to not make the player's character too powerful. With attacking a single opponent at once there was a question how to choose the target. Again the initial decision was to have a list of enemies in range and attack the first enemy from the list. It was decided against, because of the potential was recognized in the design of players mouse movement. It was possible to get the GameObject that was clicked on. This gives the player more control over the combat and it allows the player to choose the target. Next was how to calculate the damage. Flat amount is not a good decision, because it makes the game too static. It was decided to have a base damage amount that would be added to a random number. Last would be how to apply the damage. It is actually more of a part of the enemy design so it will be described later. The player has a small chance to miss, it is calculated as a probability check.
- Health – To accommodate important functionality for player's character is being able to take damage and ultimately die. In order to allow that there needs to be a Health component. This component would be relative simple. It hold the health points which can be modified by amount. It is also linked to a UI component, a slider which serves as a health bar on the screen and allows the player to monitor its health.

It was later decided to add some sort of healing. There is no item system designed for this game, as it was not the scope of the project, however the author plans to expand the game later on. In the meantime the only form of healing is natural health regeneration when the player is out of combat.

2.1.4. Enemy Design

Following are functionalities for the Enemy object:

- Single enemy behaviour – This would be the AI for the enemy that is not a part of the group. It was decided to use a finite state machine to control the opponents. This machine would consist of four states: Idle – enemy is wandering and player is nowhere to be seen, Chase – when player enters the view distance it starts to chase, Attack – when player is in range it starts to attack it, Flee – when health is low and player is in view distance. *Figure 2* shows the transitions between the states.

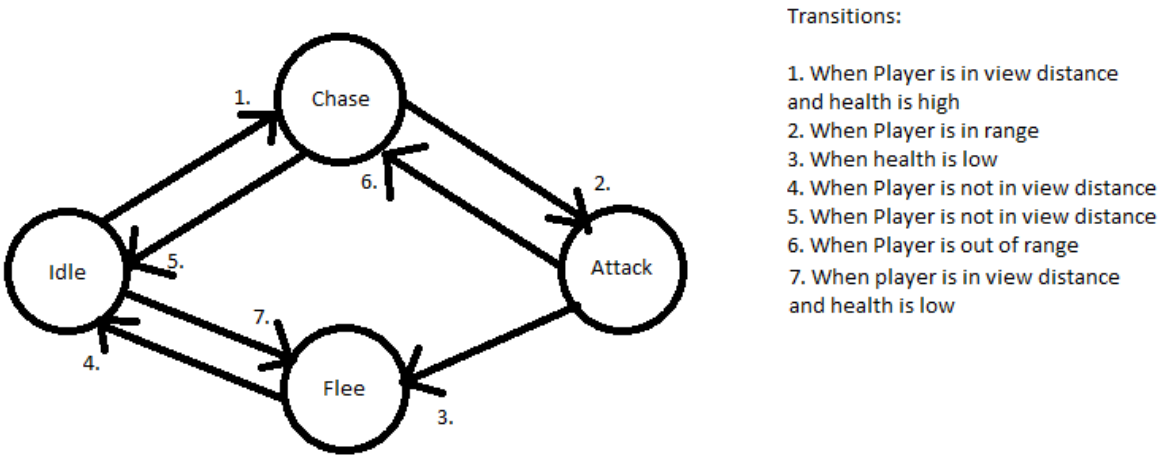

Figure 2. Enemy behaviour state machine

- Flocking – A functionality that will replace the enemy behaviour. It will use the steering behaviours, more specifically: the group behaviours – separation, alignment, cohesion, and interchangeably leader following and flee. The leader following will follow the nominated leader of the group, but when in combat the enemy will follow the player instead and will attempt to attack him. If low health then instead of following it will be steered away from the player. The general steering force will be calculated using weighted truncated sum.

- Grouping – When enemies will meet, they will form a group. Each new encountered enemy without a group it will join the group. If two or more groups will meet, they will merge into one. The enemies will actively scan the view distance for other enemies and they will proceed as described. Also the group will keep track of its members, nominate leaders and if there will be one or less the group will destroy itself. A leader is the only enemy in the group that uses enemy behaviour functionality rather than flocking.

- Health – A necessary component, similar to the player's health. It does not however has the ability to regenerate and does not have a slider to display the health on. It is planned to add these in future development of the game, hovering above the enemy GameObject. This would utilize the world space render mode of the camera. Also it is planned to add some sort of the notification about the damage done to the enemy and also damage received.

- Combat – Enemy's ability to attack and damage the player. When it is in range it will start attacking every set interval and will stop when the player goes out of range. It has a small chance to miss however. The damage is calculated the same way as the player's, but with smaller numbers.

2.1.5. AI Director Design

The AI Director, being the main AI system in the game, uses most of the functionalities mentioned above to achieve its goals:

- Enemy spawning – The AI Director uses the enemy object described above and spawns its copies in the game scene. It is responsible for keeping track of the amount of active enemies in the scene. It will use an object pool design pattern to achieve this. This pattern initializes a set of objects at the start of the program which can later be reused anywhere in the code during runtime. It is a very useful pattern, much more efficient than creating and destroying objects when the program is evaluated. But rather than creating a fixed set of GameObjects, the decision was to create objects when needed and instead of destroying them, Unity provides an alternative: the GameObject can be set to be inactive. It can be done in the editor (*Figure 3* shows an active GameObject and where is it can be set to inactive) or in code. When this happens a particular object will not appear in the scene window nor in the game window. This is also possible for a single components, which is then referred to as enabling or disabling them. So when an enemy dies it will be deactivated rather than destroyed.

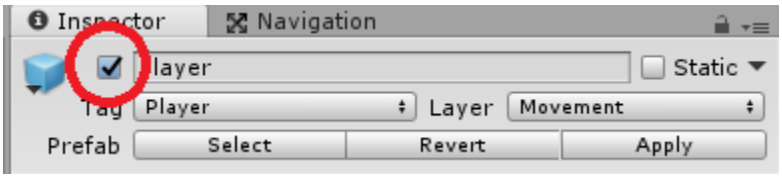

Figure 3. An Active Object in the Unity Editor

- Active Area Set – The active area is a ring around the player. It starts right outside the screen, and continues for fifty more units from the player. It is an area where it is possible to spawn the enemies. This moves with the player, and all of the enemies left out will be despawned (set to inactive for further reuse). When spawning there will be a random position generated inside the ring, it will be then checked if it is traversable on the NavMesh, and if not it will provide the closest valid position.

- Player Intensity – This is a value that AI Director's functionality will be based on. The player starts with zero and various actions can increase it, for example: when the player is hurt, when the player is engaged by an enemy, when the player hurts an enemy. These actions have different values that they can add to the intensity level. This will vary on the type of action. There is a certain maximum that it cannot go over, and when it does the AI Director will change its state. Similar way to how player's health regenerates, it will start to gradually decrease when out of combat. This functionality has been designed to be a basis for the number of enemies that can be spawned. To achieve this, it also keeps track if the player is in combat. While this is part of the player character design, the combat state can be applied by the enemies as they engage or disengage from the fight.

- AI Director States – The main algorithm responsible for controlling the spawning process. It is adapted from the system implemented in Left4Dead [CITATION Booth09 \l 1045]. It takes a form of a finite state machine. Its states are: BuildUp, Sustain, Fade, Relax (*Figure 4* shows the transitions).

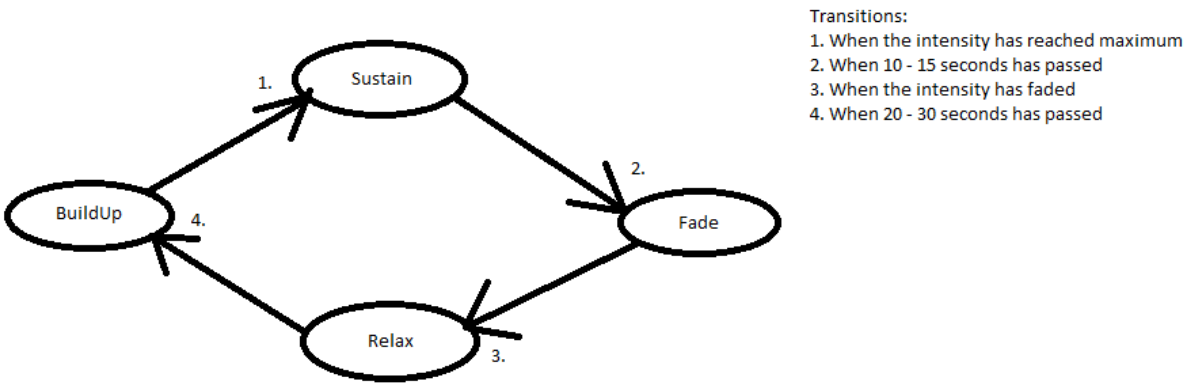

Figure 4. AI Director state machine with transitions

BuildUp creates and maintains the maximum population. The amount is decided each cycle and it is a random number between 50 and 80 entities. It also monitors player's intensity level and when it reaches maximum it transitions into the next state. Sustain continues to maintain the maximum population for the next 10 – 15 seconds. When this time runs out it transitions into Fade. This state starts to despawn enemies until it reaches the minimum population – 50 entities – and maintains it until the intensity has faded. Then the Relax state continues to maintaining minimum population for 20 – 30 seconds. That is the end of the cycle and the transition is made into BuildUp again.

- Statistics Display – To better monitor the AI Director, it was decided to design a display on the screen. This display shows the current state it is in, maximum enemies this cycle, current active enemies in the scene, is the player in combat and player's intensity as a slider.

2.2. General design hierarchy

The following *Figure* represents the design hierarchy:

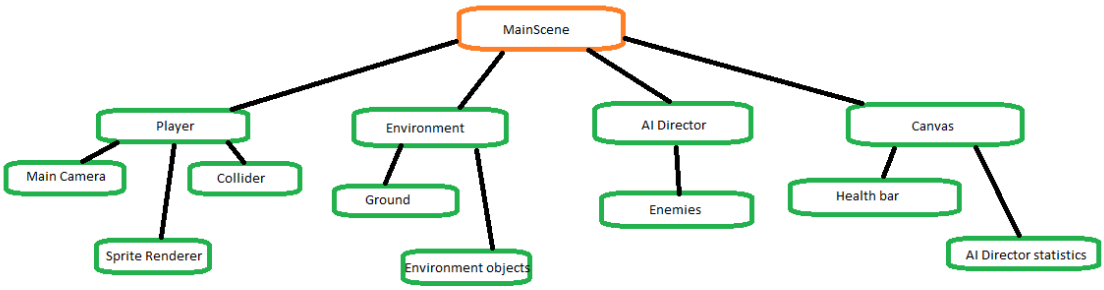

Figure 5. General Design Hierarchy

Figure 5 displays where the objects are placed in the terms of hierarchy. Main Scene holds all of the objects that appear in the scene. On the top of the hierarchy are Player, Environment, AI Director and Canvas. Player also have children: the Main Camera for purposes described in section 2.2.3, Sprite Renderer and Collider as separate objects for purposes described in section 2.2.2. Environment is a parent to the Ground terrain and to the environment objects such as trees or rocks. AI Director will be holding Enemy entities. Canvas is the User Interface in Unity3D and it will be holding the health bar for displaying player's health and AI Director statistics to display information about the AI Director.

3. Implementation

The implementation of this project has mostly met the initial design. However, some issues have been recognized and decisions were made to modify the design accordingly. Most of the issues have been described in the Design section together with an explanation of the choice. The following sections will complete the description if necessary. There were also circumstances which caused the author to question the implementation of this project change it accordingly. These will be described and explained as well.

3.1. Unity Project set up

The project has been created as a standard 3D project in Unity3D. New projects have default MainCamera and Directional Light objects created. The default light object is the only source of light in the scene and it serves its purpose well. The decision was made not to add any more sources of light, however this may change in the future development. MainCamera projection mode has been changed to orthographic to achieve an isometric perspective of the game. It has later been moved in the hierarchy as the child of the Player GameObject.

3.2. Environment implementation

The environment in the game is considered every GameObject that is not a Player, Enemy nor AIDirector and is situated in the scene. In the current stage the environment is very limited, but it is planned to be expanded in future development by adding a variety of objects and creating new types of environment. For now it resembles flat grasslands.

3.2.1. Ground

The Ground is flat terrain object. It measures 500 by 500 world units. The terrains are very useful in Unity game engine because there are many ways to modify them. It can be raised and lowered, painted with a texture or have another GameObject placed on top of them. The texture used to paint the Ground with was *grass.jpg*. It is a seamless texture which allows to be tiled on the terrain creating a visually atractive grassy effect.

3.2.2. Environment objects

There are only three environment objects for now. Two of them are rocks with different textures and the other is a tree. All of the use the same structure which is a SpriteRenderer on the

parent object along with a NavMeshObstacle and a child object with a BoxCollider, MeshRenderer and Filter. The child object has been added for collision purposes in 3D space only.

3.3. Player character implementation

Player character has undergone some design changes both in the scripts and as a GameObject in the Unity editor. The following section will explain this in detail.

3.3.1. Player object in the Editor

Initially Player object consisted of only one object with a MainCamera attached to it. It held the sprite renderer with a placeholder texture that has since been removed, Collider, Rigidbody and the scripts. However it was later recognized that this will not work in terms of collisions. The reason was described in the design section of this document. So the Collider was moved to the child object which also held the MeshRenderer and Filter. This solved the issue of objects not colliding with the Player. Later when the actual textures were obtained, they were sectioned into three parts. A character rig has been created in order to use these textures correctly. A child object was created to group other objects, each of them holding one sprite. With the rig it was possible to create animations in the Unity Editor by simply modifying the position and rotation over time. There are only three animations implemented at the moment: Idle, Walk and Attack.

3.3.2. PlayerMovement

This is implemented in a single script. It makes use of the Animator and Rigidbody components attached to the Player GameObject, and cannot be applied to a GameObject that does not have these two (RequireComponent attribute has been used to ensure that). Initially two methods of movement were created. The keyboard movement uses Unity's horizontal and vertical axes. It takes the value as raw (-1, 0, 1) instead of the float from the range, to avoid a quick acceleration when the movement starts. It flips the sprite if the Player is moving in a different direction than before to make the sprite look like it's facing the direction it is heading. There is also an Animator value isMoving changed to true. That causes the Animator to transition into a Walking animation. After that the vector created using horizontal and vertical axes is multiplied by speed variable and used as a parameter in Rigidbody's MovePosition method to apply movement to the GameObject.

There was also a method of moving by clicking the right mouse button. It would then create a Ray object from the screen (Camera) and it would return a position on the first object it would hit. This position would be saved as a destination and with each frame Player would be moved in that direction by changing the position slowly. It has been decided to use the NavMesh as a form of pathfinding and obstacle avoidance. This was implemented successfully with use of a NavMeshAgent component, which was feed with a destination point. Both of these methods would be stopped by a keyboard movement when the axes would return values other than zero.

Each frame the position would also be limited within the constraints of the terrain object and have a set height to be just on the surface of the Ground.

3.3.3. PlayerAttack

This is a combat functionality of the Player. When the left mouse button is pressed – which is not used anymore for setting Player's destination – the Attack animation is played. A Ray object is also created, the same as it was described for the now obsolete mouse movement. But instead of getting an intersection point with the GameObject's collider it returns the GameObject. If this GameObject has an "Enemy" tag there will be an attempt to attack it. Tags are a very useful functionality in the Unity as it can provide means to find an object or check if the object is of desired type. They can be set in the Unity Editor (*Figure 6*) or in the script.

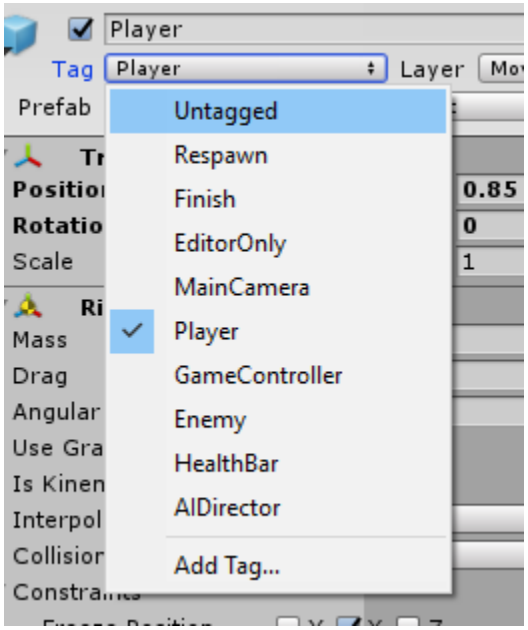

Figure 6. GameObject's tag

The Player then will attempt to attack the Enemy. It has a probability of 1 in 8 to miss though. If it landed a hit, then the damage is calculated by adding a random number from the range between 3 and 7 and applied to the EnemyHealth. The player is also limited to attack once every half a second.

Both the PlayerAttack and PlayerMovement scripts are implemented in a way that can be easily accessed or extended. All of the Player statistics are variables which in the future development can be changed to a new script which would hold the Player statistics and be changed in the runtime, when for example an item is equipped.

3.3.4. PlayerHealth

PlayerHealth is a simple script that holds Player's health points. They are set to the maximum value of 100 at the beginning of the game. When the Player is out of combat, health points are gradually regenerating at a rate of one point per second. HealthPoints can be reduced when the Player is wounded. It is done by simply adding the value of the parameter in the ChangeHealthPoints method (if the health is meant to be reduced then the parameter must be negative). During each method call the healthPoints are checked if they have dropped to zero or below. If it's true then the Player GameObject is deactivated and the game quits. This will definitely change in the future development when the menu is created.

3.4. Enemy implementation

The Enemy can have to types of behaviour: EnemyBehaviour script when it is not part of the group or is a leader of the group and Flocking when it is part of the group but not a leader. Both of these use the EnemyStats, EnemyAttack and EnemyHealth scripts.

3.4.1. EnemyBehaviour

This script is an implementation of the finite state machine described in section 2.2.5 and in *Figure 2*. It makes use of EnemyStats, NavMeshAgent, EnemyAttack and EnemyHealth components, and similar to PlayerMovement script it uses RequireComponent attribute for these. The enumeration State has been created for the states described before. Each of the state has a different method that is called in the Update method every frame. To call the correct method for the current state the switch is used. Idle state gets a random position in a circle of 10 units around the Enemy and sets the NavMeshAgent destination to it. This repeats every 5 seconds. If the Player happens to be in the viewDistance of 10 units and the Enemy has more than 15 healthPoints it transitions to the Chase state. Otherwise it will transitions to the Flee state. Chase state changes the destination to the Player's current position. It also changes speed to RunningSpeed from the EnemyStats script. If the Player moves to a position that is larger than 20 world units than the Enemy transitions to Idle gain. If the Player comes into range of 3 units it transitions to Attack state. This state makes use of the EnemyAttack to attack the Player. It can transition back to Chase or Flee depending on range and healthPoints. Flee state calculates a position 10 units away from the Player and sets the destination to it. If it successfully moves 20 units away from the Player, it transitions to Idle again.

3.4.2. Flocking

Flocking script implements the steering behaviours described in section 2.2.5. This script uses EnemyStats. Every second there is a check for all the colliders in the viewDistance. All of the Enemies that are detected are added to the list of neighbours. Every frame flocking is calculated based on the steering forces. Steering behaviours that are implemented are all added to the overall flocking force. Separation sums the directions away from each neighbour. Alignment calculates the average direction of all the neighbours. Cohesion calculates the direction towards the centre of the neighbour group. Then there is also calculated leader following. This calculates the arrival steering behaviour to the current position of the leader unless it is in front of the leader, then it returns a direction away from it. If the Enemy is in combat it will instead follow the Player. If it is in combat and its healthPoints are low then it will calculate fleeing steering behaviour away from the Player.

Steering forces are summed after being multiplied by weights assigned to them. After that the flocking force is smoothed out to avoid unnaturally looking movement. This is done by averaging up to last five movements. After all the calculations are complete, the force is applied to the Rigidbody of the Enemy through AddForce method, applying the speed as well.

3.4.3. Grouping

Groups have a GroupManager script to keep track of its member and nominate leaders. When two Enemies with Grouping scripts – which are only enabled if EnemyBehaviour is enabled - encounter each other, a Group is created and both of them are added to this group and leader a is nominated. GroupManager enables and disables the scripts accordingly. When a lonely Enemy encounters another Enemy who is a part of a Group, it will then be added to that Group. Both of the above situations are evaluated inside the Grouping script. When two Enemies, who are of different groups, meet, they will start adding the each other to the group. This will cause one of these groups to

become empty which will cause the GroupManager to destroy the Group GameObject. As a result the groups will merge into one.

A Group is a GameObject that exists in the scene and serves as a grouping object for the Enemies, therefore all the Enemies, while part of the group will be moved under the Group in the hierarchy.

3.4.4. EnemyStats

This is a short script, which is only used as a variable holder. These variables are available for other Enemy scripts, through the use of properties. The variables are healthPoints, baseDamage, attackSpeed, walkingSpeed, runningSpeed and viewDistance.

3.4.5. EnemyAttack

This script serves for Enemy combat purposes. When Player is in range of the Enemy, both EnemyBehaviour and Flocking scripts will call the method StartAttacking, which in turn will be calling Attack method every AttackSpeed seconds. The Enemy have a probability of 1 in 4 to miss. If it didn't miss then a random number between 1 and 5 is added to the baseDamage and applied to the Player's healthPoints.

3.4.6. EnemyHealth

This script is similar to PlayerHealth, it holds a value of Enemy's health points. There is a method that can modify the healthPoints. It also checks if the Enemy has less than zero health, and if that is true then it is deactivated.

3.5. AI Director

3.5.1. PlayerIntensity

This script, attached to the Player GameObject, keeps track of the Player's intensity level and if it is in combat. Intensity can be modified similar to healthPoints. When Increase method is called then the intensity level is modified by the amount. It also makes sure if the intensity doesn't go over the maximum value. If the Player is out combat, intensity will start to decrease, 1 point per second.

3.5.2. Spawning

To spawn the Enemy there needs to be a valid position generated for it. Active Area Set is implemented in the form of a ring around the Player. A position is randomly generated using Unity's Random library's insideUnitCircle which is expanded by the size of the Active Area Set. It is also centred on the Player's current position. If the position is within distance of 20 units to the Player, then a new position is generated. This prevents the enemies from unexpectedly spawning right on the screen. Then the position is also checked on the NavMesh using the SamplePosition method. If the position is not traversable then a nearest available position is returned. Then when a spawnPoint is finally ready, it is checked if there are any available inactive Enemy GameObjects in the despawned list. If yes, then it is

reused, if not then a new Enemy GameObject is instantiated at the spawnPoint. Spawned Enemy is added to the spawned list.

Despawning is a simple process. The despawned enemy is deactivated and moved from spawned list to the despawned for later reuse.

3.5.3. AIDirector state machine

The AI Director state machine is implemented based on the design described in section 2.1.5 and in *Figure 4.* It is similar to the EnemyBehaviour state machine. There are four state, each having its own method.

In the BuildUp state the spawning starts and the maximum population is maintained. When intensity level reaches maximum intensity, it generates a random number between 10 and 15 which will serve as time for Sustain state. In Sustain the maximum is upheld until the time runs out. Then a random number is generated, in the range between 15 and 20, this time for a target value to which the intensity must fade. In the Fade state the Enemies are despawned until a minimum population is reached. When the intensity fades below the target amount, another random number is generated, between 20 and 30, which will be a length of the next state. Relax time still does not spawn any enemies. After the time has passed, the cycle is complete and the AI Director transitions into BuildUp state again.

3.5.4. AIDirector information display

The display is located in the top right corner of the screen. It consists of a couple of elements. First is the title, that informs the Player that it can be toggled on and off by clicking the F12 key. The current state of the AI Director is displayed below. Further down there is an information about the maximum enemies value and the current enemies count. Player's intensity level is displayed at the bottom as a slider along with the Boolean value signifying if the Player is in combat. All of these are updated each frame.

Figure 6. AI Director information display

This is only for the purposes of the project. In the future development this will be removed to make room for more User Interface components.

4. Testing

4.1. Manual testing

Unity Editor does a very good job of visualizing the game in the process of development. Most of the changes that are done in the script can be seen in the Play mode. Play mode is basically simulating the game like it was already built. It is very crucial for manual testing. After each change in the code it was immediately tested in the Play mode to see the results. If these were unsatisfactory, it usually could be seen why and what was needed to be changed. Unity editor have also a console in which there are errors, warnings and messages displayed. Messages can be written in the code using Debug.Log method. This method was very helpful in debugging the code. In some cases thought it

wasn't enough, when Unity3D was not giving too much information. Luckily VisualStudio provides a step by step debugger, which helped resolved harder issues.

When implementing AI Director a simple display was created on the screen, which is described in section 3.5.4. This display also proved valuable for catching out bugs, when using the Play mode.

4.2. Automated Testing

Unity3D provides a built in testing framework since version 2017.1. It is Unity's implementation of NUnit framework. In the Unity Editor there is a Test Runner window which will run tests. There are two modes available: EditMode and PlayMode.

EditMode can only be used for unit testing. It is a hard method though, because of the implementation of components in Unity. All components inherit from MonoBehaviour library and this library prevents creation of components that are not attached to a GameObjects. For this reason the only components that can be used in this mode are ones, that are very decoupled and have methods that do not use Unity API. For this reason it was decided to not do EditMode test.

PlayMode tests can be used for both unit and integration tests. To create unit test in this mode an attribute Test or UnityTest is used. PlayMode allows loading scenes, which takes care of the problem described for the EditMode. It is possible then to use the existing GameObjects and their components, or instantiate new ones. These tests are run when the game is not playing. To do integration testing IntegrationTest attribute with the name of the method being tested is used. When the game is playing the test in the Test Runner can be runned and when that method is called, the test will be evaluated.

In this project only unit tests test were carried out and due to the limited time there are not too many of them. Due to the coupling of the scripts it was hard to extract the logic that could be then tested. With this in mind it is planned to refactor the scripts in the future development to allow more successful testing, that could be run regularly.

4.3. User testing

There was also a user testing conducted. Only three volunteers were found and provided feedback.

All of the testers agreed that the game has potential to be further developed. Also they agreed on that the game is interesting at first, but without a clear goal it becomes boring quickly. One of testers claimed that the enemies did not die or run away, instead sticking to the enemy. Since this only happened to one of the testers a conclusion is drawn that this may have happened due to a technical failure of the game. One of the testers was of opinion that a group of enemies is too much for one player. Because of that it is planned to adjust enemy's health and other statistics so that it does not pose that big of a threat when in group. It was also agreed that there should be a menu and some sort of a introduction of the game, as well a "Game Over" screen. This will be implemented in the future development.

Reflecting of the method used when carrying out the testing, a conclusion has been made that in the future possibility of doing this again, that it is important to make everything clear with the testers, as some of the communication was not very good.

5. Critical Evaluation and Insight

In my opinion the project main goals has been successfully implemented, however if more time was spend on the project there could be much more done. Especially the esthetics side of the project is very limited as there are animations left to be implemented. Also there could be more tests done, however as they were done at the latest stage of development, there was not enough time to finish it.

A lesson to be learned from this experience is that the work should be more systematic from the early part of the project and not in the last month before the deadline. The author is aware of that and will definitely not repeat that.

Appendices

A. Third-Party Code and Libraries

Unity3D – The project was built in Unity game engine. It has handled all of the graphics rendering, physics rendering and building and releasing the game. Version used was 2017.3.1f1 for the 64 bit machines, which is the newest release at the moment of writing this report. The license used was Unity Personal free license, which includes all features that are available in this engine, only limited to number of users in the multiplayer (Unity Technologies). No changes were made to the engine as well as the libraries that come with it.

B. Third-Party Assets

All of the textures used in the project were made by Antoni Janas and are used with his permission.

All of the sound files used in the project were made by Piotr Maleszewski and are used with his permission.

C. Ethics Submission

Ethics Application Number: 9778

AU Status Undergraduate or PG Taught
Your aber.ac.uk email address jaj48@aber.ac.uk
Full Name Jakub Janas
Please enter the name of the person responsible for reviewing your assessment. Reyer Zwiggelaar
Please enter the aber.ac.uk email address of the person responsible for reviewing your assessment rrz@aber.ac.uk
Supervisor or Institute Director of Research Department cs
Module code (Only enter if you have been asked to do so) CS39440
Proposed Study Title Procedurally Populated Environments with use of Steering Behaviours
Proposed Start Date 29/01/2018
Proposed Completion Date 04/05/2018
Are you conducting a quantitative or qualitative research project? Mixed Methods
Does your research require external ethical approval under the Health Research Authority? No
Does your research involve animals? No
Are you completing this form for your own research? Yes
Does your research involve human participants? Yes
Institute IMPACS
Please provide a brief summary of your project (150 word max) A Game Development project focused on populating the scene with automotive agents. The project will be developed in Unity3D and use resources available in this engine. The game world will be in 3D space, but with 2D objects. Basic gameplay will involve player fighting with enemies, that will be spawned around the map based on the player's location along with couple other determinants.
I can confirm that the study does not involve vulnerable participants including participants under the age of 18, those with learning/communication or associated difficulties or those that are otherwise

unable to provide informed consent? Yes
I can confirm that the participants will not be asked to take part in the study without their consent or knowledge at the time and participants will be fully informed of the purpose of the research (including
what data will be gathered and how it shall be used during and after the study). Participants will also be
given time to consider whether they wish to take part in the study and be given the right to withdraw at
any given time. Yes
I can confirm that there is no risk that the nature of the research topic might lead to disclosures from the participant concerning their own involvement in illegal activities or other activities that represent a
risk to themselves or others (e.g. sexual activity, drug use or professional misconduct). Should a disclosure be made, you should be aware of your responsibilities and boundaries as a researcher and be
aware of whom to contact should the need arise (i.e. your supervisor). Yes
I can confirm that the study will not induce stress, anxiety, lead to humiliation or cause harm or any other negative consequences beyond the risks encountered in the participant's day-to-day lives. Yes
Please include any further relevant information for this section here: I want participants to conduct user testing
Where appropriate, do you have consent for the publication, reproduction or use of any unpublished material? Yes
Will appropriate measures be put in place for the secure and confidential storage of data? Yes
Does the research pose more than minimal and predictable risk to the researcher? No
Will you be travelling, as a foreign national, in to any areas that the UK Foreign and Commonwealth Office advise against travel to? No
Please include any further relevant information for this section here:
If you are to be working alone with vulnerable people or children, you may need a DBS (CRB) check. Tick to confirm that you will ensure you comply with this requirement should you identify that you require one. Yes
Declaration: Please tick to confirm that you have completed this form to the best of your knowledge and
that you will inform your department should the proposal significantly change. Yes
Please include any further relevant information for this section here:

Annotated Bibliography

A.I. Design. (1980). Rogue. Epyx.

Adobe Systems. (n.d.). Photoshop. Adobe Systems.

Autodesk. (n.d.). Maya. Autodesk.

Bethesda Softworks. (1994, March 25). The Elder Scrolls: Arena. Bethesda Softworks.

Bethesda Softworks. (1996, August 31). The Elder Scrolls II: Daggerfall. Bethesda Softworks.

Blizzard North. (1996, December 31). Diablo. Blizzard Entertainment.

Booth, M. (2009). The AI Systems of Left 4 Dead. Stanford: Artificial Intelligence and Interactive Digital Entertainment Conference (Stanford University).

Braben, D., & Bell, I. (1984, September 20). Elite. Acornsoft.

Buckland, M. (2005). *Programming Game AI by Example.* Wordware Publishing Inc.

Epic Games, Inc. (1998, May). Unreal Engine. Epic Games.

Epic MegaGames. (1998, May 22). Unreal. GT Interactive.

Fatshark. (2015, October 23). Warhammer: End Times – Vermintide. Fatshark.

Fingas, J. (2015, April 3). *Engadget.* Retrieved from https://www.engadget.com/2015/03/04/how-minecraft-worlds-are-made/

Hello Games. (2016, August 9). No Man's Sky. Hello Games.

Hewlett Packard Enterprise. (n.d.). *OpenGL*. Retrieved from https://www.opengl.org/

Horswill, I., & Foged, L. (2012). Fast Procedural Level Population with Playability Constraints. In *AIIDE'12 Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (pp. 20-25). AAAI Press.

in het Veld, B., Kybartas, B., Bidarra, R., & Meyer, J.-J. C. (2015). Procedural generation of populations for storytelling. In *Proceedings of PCG 2015 - Workshop on Procedural Content Generation for Games, co-located with the Tenth International Conference on the Foundations of Digital Games.*

Khan Academy. (n.d.). *Khan Academy.* Retrieved from https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise

Klei Entertainment. (2013, April 23). Don't Starve. 505 Games.

Lightweight Java Game Library. (n.d.). *LWJGL*. Retrieved from https://www.lwjgl.org/

Mojang. (2011, November 18). Minecraft. Mojang.

Oracle Corporation. (n.d.). Java Swing. Oracle Corporation.

Oracle Corporation. (n.d.). *JavaFX*. Retrieved from http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html

Over The Edge Entertainment. (2005, March). GooBall. Ambrosia Software.

Plunkett, L. (2016, November 13). *Kotaku.* Retrieved from https://kotaku.com/old-elder-scrolls-game-takes-over-60-hours-to-walk-acro-1788931074

Reynolds, C. W. (1987). Flocks, Herds, and Schools: A Distributed Behavioral Model. Anaheim: ACM SIGGRAPH.

Reynolds, C. W. (1999). Steering Behaviors For Autonomous Characters. Foster City: Sony Computer Entertainment America.

Reynolds, C. W. (2000). Interaction with Groups of Autonomous Characters. Foster City: Sony Computer Entertainment America.

Smith, G. (2014). The Future of Procedural Content Generation in Games. In *Proceedings of the AIIDE Workshop on Experimental AI in Games.* Raleigh, NC: AIIDE.

Smith, G. (2015). An Analog History of Procedural Content Generation. In *Proceedings of the 2015 Conference on the Foundations of Digital Games (FDG 2015).* Monterey, CA: FDG.

Turing, A. M. (1937). On Computable Numbers, with an Application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society* (pp. 230-265). London Mathematical Society.

Unity Technologies. (2005, June 8). Unity3D. Unity Technologies.

Unity Technologies. (n.d.). *Unity Learn - Tutorials*. Retrieved from https://unity3d.com/learn/tutorials

Unity Technologies. (n.d.). *Unity Personal*. Retrieved from Unity: https://store.unity.com/products/unity-personal

Valve South. (2008, November 17). Left 4 Dead. Valve Corporation.