Game Artificial Intelligence Coursework – Jack Oswald

Project Proposal

The purpose of this coursework is to investigate, research and implement an AI technique or algorithm. The goal of this coursework is to research and learn as much as possible about the given topic we decide to pick. The goal of this project is not to create a brand-new algorithm or technique in the AI field, but it is to learn how to properly research an algorithm, implement and evaluate it.

For this project, I have chosen to investigate down the Decision-Making path, more specifically, Utility AI. I am researching Utility AI as it forms the core of any AI system in most of today’s games such as Dark Souls. Each enemy in Dark Souls will use Utility AI to determine what course of action they will take at the given moment. Utility AI is focused on an agent which observes and acts upon their current environment. A utility will be assigned to each available action and the agent will then perform the action that provides them with the greatest utility. Each action will be treated separately, and will be scored on a scale that is unique to each algorithm it is implemented in. I hope to gain a base understanding and vital knowledge of how utility based AI systems are implemented and how they compare to the different decision making AI techniques Utility AI is an extremely powerful way to get rich, life-like behaviour and it is good at fulfilling the requirements of any state of the art AI. Utility AI can use some simple values and a handful of weights and from this can add some personality to each individual agent in the game.

This artificial intelligence will be tested in a 2D based game. The game will feature the player and the AI agent. The player and agent will both be able to move around freely and shoot in any direction they wish to do so. There will be also be health packs that the player can pick up. The Utility AI will be combined with Seek and Flee Steering Behaviours. Doing this will ensure the AI is as lifelike as possible and provides a good testing ground for the Utility AI algorithm.

The algorithm will be set up in a way so that the AI will consider 4 basic factors. When to attack, run, heal and reload. The AI will determine what the best course of action would be at that specific situation by plugging in the relevant values of the associated utility of each action. For example, if the AI has more than 20% of their bullets left and more than 35% of their health left, then they will start to attack. If the AI is low on health and is currently attacking with a substantial number of bullets left, then they will start to heal based on how the healing mechanic is set up in game. This utility will be calculated by coming by up with proper functions to generate reasonable utility curves.

Investigation and Research

Rasmussen (2016) explores the realization that Behaviour Trees are a thing of the past and Utility AI is taking over. With developers needing more complex AI that features emergent behaviour and life like decision making, Utility AI is replacing Behaviour Trees as a more advanced method of artificial intelligence. An AI with only 4 actions will create a Behaviour Tree that is rather big and complex for only a small amount of actions. Increasing the actions by 1 or 2 could see a well-designed Behaviour Tree grow out of control. There is also little room for change. If a designer wants to change the input values or some of the rules, a whole redesign of the tree must be undertaken. Utility AI does not require implementation of a tree structure. Instead, it will read in the number of actions and generate a score. The order of the input parameters is irrelevant; this means that there is no need to regularize the AI. This research is implemented in one of the best designed AI systems around, Killzone 2.

The utility system in Killzone 2 and games in general proves that it is a well-working method for several reasons:

1. Simple to Design – Utility AI allows designers to fully converse with programmers about certain parts of the system, without having to know any technical jargon involved
2. Easily Extendable – Rules can be easily added making it easy to extend any function of the AI
3. Better Quality – The ease of use and simplicity of the design improves productivity

When calculating utility, it is extremely subjective between each programmer. Although utility can have the same input, two different programmers will write two different functions that produce separate outputs. It is however important that an understanding of the relationship between the input and output is obtained. The result of this is a utility curve. A utility curve expresses the conversion from a value in the game world to a utility. Value is expressing a concrete number in the game world, while utility is measuring a concept. Converting value to utility uses a formula unique to each situation. (Graham, 2013).

Once the utility has been calculated, picking an action is different across most games. The standard way to pick an action is known as Absolute Utility, which is just to pick the highest scoring action. In today’s modern games, this kind of behaviour can feel robotic and can become predictable to the user. Another method to use is called Relative Utility. This utility involves that each action is weighted and then pick one of the highest weighted actions at random. This ensures that utilities with higher weights will always be picked. However, even though an action can be weighted for that specific situation, in some cases an action can seem completely unreasonable and foolish. The most complex and reasonable method of calculating utility is called Dual Utility. Dual Utility is the combination of both absolute utility and relative utility. Instead of using a single utility score, each action will use rank and weight. The rank will divide each action into separate categories and each category will have its own weight. The category with the highest weight is then chosen and using a random weight basing an action is chosen in this category.

The Sims 4 is a great example of a smart AI that picks an action that is the best for the agent at that specific time and moment. For this system, the agent would first look around the world and figure out what can be done in that area. It will then score all the possibilities based on how beneficial it is in satisfying the agents internal needs. For example, if the agent has a hunger value of 30, this means that the agent would more urgently need to find some food. This process can be broken down into an algorithm that is simple to understand, easy to implement and is extremely efficient in terms of CPU usage and memory. The algorithm pushes scores based on the agents need, onto an action queue.

The AI loop looks something like this:

* While there are actions available, pop the next one off the queue and perform this action
* If there are no actions left, perform action selection based on the agent’s current needs
* If there is still nothing left for the agent to do, perform some fall-back actions

The second step, the action selection point, is where the agent chosen which action to undertake. It decomposes as follows:

1. Examine objects around you, and find out what they advertise
2. Score each advertisement based on the agent’s current needs
3. Pick the best advertisement, get its action sequence
4. Push the action sequence on the queue

(Zubek, 2011).

One problem that occurs in Utility AI is the concept of inertia. If each state is being calculated every frame in the game world, the agent is prone to oscillation. For example, if the agent is scoring shoot the enemy and run away closely, the agent could start uncontrollably performing these states. This can cause extremely uncharacteristic and unlifelike behaviour. There are multiple ways to solve this problem:

1. Introduce a locking mechanism that will force the agent into completing the action before undertaking another action
2. Introduce a cooldown timer
3. Introduce a weighting system to a current weighting system, this weight would be extremely high and only an action that is significantly better would change the action

Implementation

A basic 2D scene was created so the agent could move around and shoot. An ammo box and a health pack were also placed in the scene so that the agent could pick them up whenever their relevant utility values are high enough. A static AI turret was also placed in the scene, this static turret can shoot at the AI agent and damage it.

To start, a script called AIScript was created. This script will handle all the functionality of the AI Agent. It will also be used to calculate the highest utility value from all the utility functions, and then perform the action which is the highest.

The attack method starts off by finding a random static AI in the scene. A method called SelectRandomEnemy() is used. This method will take calculate and pick a random game object from an array of game objects. This method was originally used to calculate a random object between 3 static AI agents in the scene. Due to complications when selecting more than one agent and a general cluster in the scene, 2 of these static AI had to be scrapped and only 1 exists in the scene now. Below is an image of the scene in game view with 3 of the static agents to show why this was changed. After the object is selected, the agent must move towards the static AI. This method takes in the random game object selected and will transform and rotate the agent transform values and move the agent towards the static agent. When rotating the agents transform, to ensure the agent doesn’t rotate in an axis which is irrelevant to the game world, “Space.Self” is used. This solved an earlier problem where the agent was rotating out of game view.

The heal method implements a similar random method as the attack. A random health pack is elected from an array of game objects. The agent will then move towards the selected health pack. A public method is then created called AddHealth() which adds health to the agent based on the parameter in the method. This method is then called by the HealthPackScript. The HealthPackScript will detect if the agent collides with the health pack, and if it does, the AddHealth() method is then called.

The original idea for the health pack was to have constantly spawning health packs. The code was added and implemented to the game, however when run, it seemed like a pointless addition and there were some problems caused. One problem was the health packs were spawning on top of each other, this caused each health pack to stack. When the agent picked up this stack of health packs, it caused unexpected behaviour and the agent started oscillating in place. A simple fix to this was to remove the spawning of the health packs and keep a static health pack in the scene. The agent can now run over and pick up the health pack and health pack won’t be destroyed or changed. Although this method seems less life-like, it fixes the issue of oscillation and ensure that the health pack will always be in the scene when the health utility is the highest.

The reload method will use the same functionality as the attack and heal. A random object will be chosen in the scene, and the agent will move towards the selected ammo box. In this scene, there is only 1 ammo box, so only that one will be chosen. The reload functionality originally had the agent pause for a couple of seconds and reload. This caused an unbalance in the scene as sometimes the agent would be stuck inside the range of the static AI, causing him to take unnecessary damage.

The agent’s health and ammo values are displayed at the top left of the screen for visualization purposes. The static AI’s health variable is also displayed on the bottom right of the screen. These values are assigned to a text variable and then set to an object reference in the hierarchy.

The agent and the static AI both have similar shooting scripts. To start, each have their own tracking script called TrackingScript and TrackingTurretScript. A variable called lastPosition is used, this is first set to a Vector3 of zero. This variable will then be set to the agent/static AI’s current transform position if it isn’t already. The game object will then be rotated towards the agent/static AI’s current position. A public abstract class is then created called BaseProjectileScript. This abstract class contains two abstract member functions called FireProjectile1 and FireProjectile2. These member functions will handle the firing of the projectile from the specific AI in the scene. An example of these functions is shown in Figure 1 below.

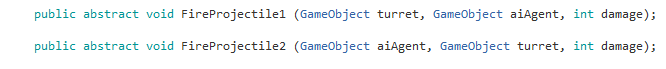


Figure 1

A script called ProjectileScript is then created, this script inherits from the BaseProjectileScript. The FireProjectile functions from Figure 1 are then overridden since it is being implemented from an abstract class. These functions will now set the direction of the projectile to be fired and set the fired Boolean variable to true. Once fired is equal to true, the projectile is then fired out in the direction of the target and is based on the speed variable in the BaseProjectileScript abstract class. An example of one of these functions is shown in Figure 2.

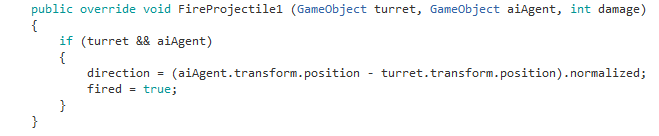


Figure 2

A script called ShootingScript is then created for the static AI to be able to shoot. The FireProjectile2 function is called in the base AIScript for the Attack() method. The FireProjectile1 is called within the ShootingScript. For both functions, the bullet prefab is instantiated first, then the specific Fire Projectile function is called.

To calculate the utility values on attack, heal, reload and run, a script called UtilityAI is created. In this script, all the specific utility functions are created and all these values are then place in an array. The utility values are held by 4 floats specific to each utility.

The 4 float values are placed in their own unique element in the array. Once the array is created and initialized, it is then recreated to resize it. Originally, a list was created and each value was placed in a specific element. However, this list was adding and updating each frame and the list grew uncontrollably after only a couple of seconds. This caused major issues in the game and only after 10 seconds the game became unplayable due to fps drops. A quick change of data structure ensured that the original behaviour that was intended was implemented correctly.

A utility curve is implement for each individual utility value. These curves have their own formula based on different values and stipulations in the game world. One interesting curve is the Heal Utility curve. The Heal Utility value is based off the Sigmoid Curve, as shown in Figure 3 below. It uses a basic logistic function; this function gives the biggest rate of change in the centre of the input and gives the intended behaviour that was chosen at the start of the project. This curve uses Euler’s number, the base of the natural logarithm, to calculate and adjust the steepness of the curve. The formula uses the currentHealth and maxHealth variables from the base AIScript to calculate a reasonable utility value.

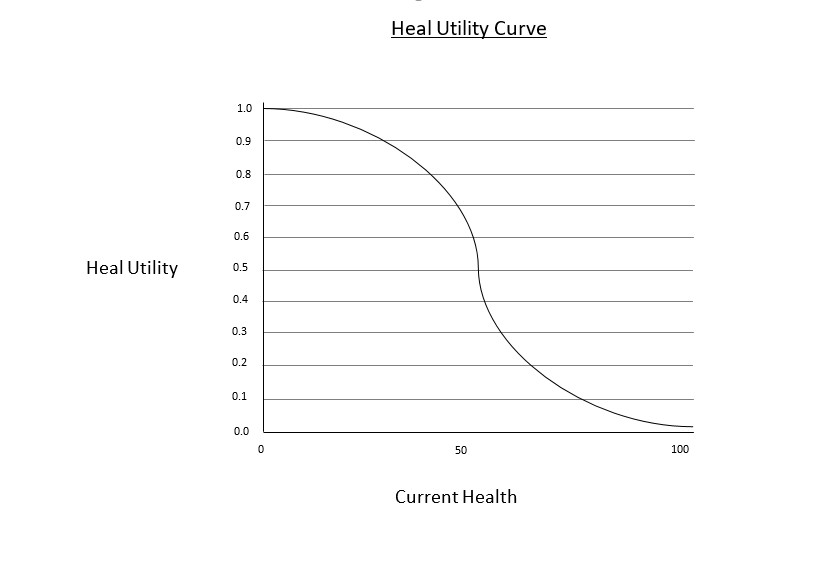


Figure 3

This formula from Figure 4 is taken and based from (Graham, 2013). Graham discusses a solution to the heal desire of an actor based on their current health. This solution gave a base understanding and incredibly useful knowledge of how to implement a utility curve using formulas. These values and variables form the original formula were also changed in regards to this specific solution.

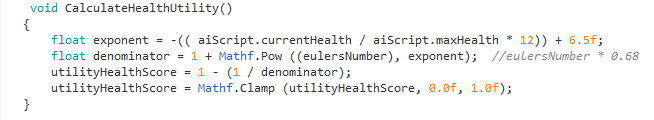


Figure 4

These utility values are normalized within their own formulas, this ensures that they can be compared without having to normalize them using more formulas. These values are also clamped using the Mathf.Clamp function between 0 and 1 to ensure they don’t go over the intended scale to be compared upon.

One each utility has been calculated, these utilities need to map to their specific functions if their utility is the chosen utility. This is done by finding the max value from the array created to hold the utility values. This algorithm will use the Max() method which is an extension method from ‘System.Linq’. This method will return the maximum value in the array. This method is then compared to each specific utility value. If the max value is equal to one of the values in the array, then that utility values function is then called and executed.

One major problem that occurred with the overall algorithm was the run method. Originally, the run method was supposed to give the agent some time to back off and get out of range of the enemy/enemies in the scene. In theory, this was a good state to have in the game. However, this state suffered from never being picked as the highest utility. Since the calculation for this utility is based off how much health and ammo the agent has, the agent would just go and get more health/ammo, thus decreasing the run utility.

Since this solution is using absolute utility, this means that there are no weights being calculated on any action. This could have been an ideal solution to the problem. However, there would have had to been a whole redesign of the system. Since one of the values that the run utility is calculated on is the static AI’s current health, a regeneration method was created. This method will start to regenerate the static AI’s health over a small period of time.

Evaluation

To begin testing of the algorithm, we will look at the built-in statistics window in the game view. This window displays real time statistics, such as FPS and graphic options. The game was left running for approximately 15-30 seconds to test the overall stability of the FPS. In Figure 5, we can see that the FPS is at roughly 80. The FPS averages around 70-80 for most of the time it was left running. Occasionally, it spiked at 200 FPS. This could indicate that the game is either well optimized, or there just aren’t too many heavy calculations or several images/animations to render. To get a more in depth look at this FPS, the Unity Profiler is used.

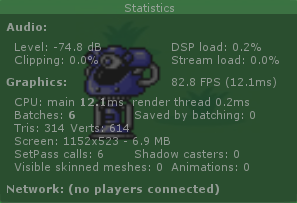


Figure 5

The unity profiler will help with the optimization of the game. It allows us to see how much time is spent in the various areas of the game. This will give a good idea of what processes are being run and how much time they are taking up in regards to the CPU, GPU etc. The game was run for 15-20 seconds and the results are shown in Figure 6 below.

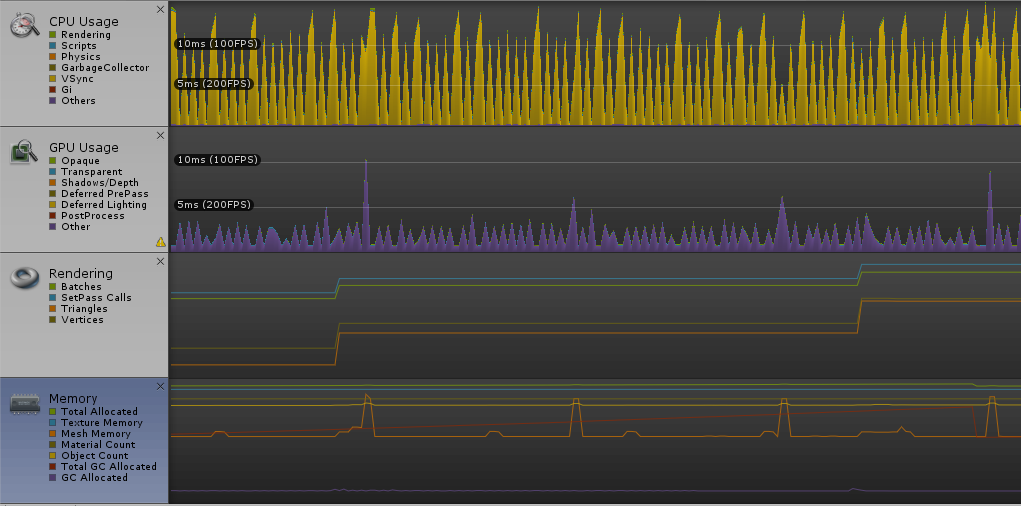


Figure 6

The results from Figure 6 show that the orange parts of the graph in the CPU usage are constantly spiking up and down at an average of 100 FPs. Upon further inspection, this process known as “WaitForFPS” is limiting the FPS of the project. This is a standard procedure in most games and is known as Vsync. Vsync limits the FPS so that the CPU isn’t wasting time on more frames. In a small game like this, limiting FPS won’t give any overall to the project since there isn’t much for the CPU to do. In a larger game, limiting FPS could be crucial to allow other processes to flourish and let the CPU handle them instead of wasting time on frames. Since this process is above an 80% load for most of the game, this ensure that more frames are compressed and the CPU can use more resources elsewhere.

The small spikes on the GPU can be caused by small things such as uploading a graphic to a texture, shading, or perhaps texturing a game object in the scene. Since this specific game is very light on the graphics side, this is not a worrying problem as the graph still shows. If this game was to introduce animations on the agent and turret or even more textures/game objects, this could causer a problem after a while. Each game object would have to be correctly textured in proportion to each other to be consistent

One item that is present in every frame is the “BehaviourUpdate”. This update handles all the scripts and calculations. The time spent in this function itself seems to fluctuate between 5% one frame, and around 0.5% the next. Since this function doesn’t do much work itself, it is relatively light on processing. Inside the function though, is where all the calls are made to handle to scripts update functions. Figure 7 shows what processes are inside the function and how much time they all spend in their update functions. Around 25% of time is spent in this function one frame, and 0.5% the next. Most of the time is spent inside the AIScript. This is obvious as this script handles all the functions are to be performed by the agent itself.

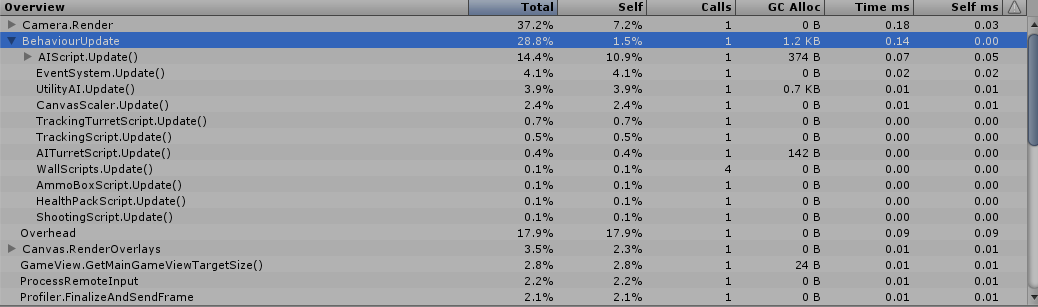


Figure 7

The GC Alloc column shows that 1.2KB in memory is being allocated to the current frame. This is then collected by the garbage collector later. To avoid frame rate issues in the game, this number should be at zero to avoid any problems. This number is extremely small in regards to zero, so this would not cause any noticeable frame rate issues at the current value. If this number was to increase, a serious optimization of the code would have to be done.

In the AIScript, when selecting a random object for the agent to move towards to, it uses 3 separate methods. These methods are extremely similar in their process. There is also only one object in the scene for each of these methods, so this piece of code is generating excess memory for no reason and is generally inefficient. This could be directly attributed to why this script is allocating memory to the current frame. It could also be an issue of how the highest utility value is calculated. Right now, the array is being accessed but it doesn’t belong to this script, so the array data would have to be stored and a copy of the data would have to be provided. The Linq function is also being used to find the maximum value in the array. This function has been known to allocate immediate buffers to the garbage collector. So, limiting the amount of Linq commands should be a priority if this project was to be undertaken any further. Since this project is relatively small, and won’t be used for publishing, using this command 4 times is fine.

Given that the algorithm must constantly update whether the agent is colliding with objects in the scene, it seems logical that the call to the physics render engine would be high. Since there are not many objects in the scene, there isn’t lot of time spent in the “Physics2D.FixedUpdate” call. This means the current algorithm can handle the colliders of bullets and the agent smoothly. Following this, adding in an extra turret or two wouldn’t cause too many problems at this sort of level of game. If this current algorithm was implemented into a full-scale game a few issues could arise.

One issue could that arise would be the communication of scripts on how the game object is detected. When an object collides with another game object in the scene, the tag of the colliding game object is checked. This is done using ‘col.gameObject.tag == “Agent”". Instead of using this, ‘col.CompareTag(“Agent”)’ seems to be a more efficient option. Calling the tag property on the object allocates more memory to it, this can start to cause problems if this is inside of a loop or some other function that repeats itself frequently.

Another issue could be the shooting of the game objects in the scene. The agent and turret are instantiating and destroying bullets for most of the playtime in the scene. To ensure these bullets don’t fly in around in the scene forever, 4 invisible walls are placed around the scene. Although this solution can fix memory leak by not allowing game objects to exist in the scene forever, the bullets instantiating could cause a problem. Instead of instantiating and destroying the bullets every frame, object pooling could have been used. This method pools all the objects so that they are never allocated dynamically during runtime. If this project was to handle more turrets and more shooting, the scene could get out of control very fast if each bullet is being processed in runtime.

Comparing this algorithm to the one discussed in the Investigation by Zubek (2011), they seem to have a lot of similarities. The key structure from that algorithm seems to be scoring an action from the data based on the environment, then picking that specific action. This algorithm, follows the same theory. The data from the environment can be mapped as the agent’s health, or ammo count. Scoring this can be mapped to the method which calculates the highest utility, and picking the action can be depicted as performing the specific action. Although the complexity of the algorithms is significantly different, the common methods seem to hold true.

Overall, this algorithm seemed to be a success. In most parts, the structure of the algorithm seemed to work out and each individual component seemed to come together well. The data from the algorithm seems to flow well and doesn’t cause any major issues computationally. More emphasis on not running calculations every frame could be improved on in the future, having 4 calculations constantly being updated could cause problems if this game was to grow in magnitude.

Reflection

I felt I researched and implemented an algorithm from scratch well. Since Utility AI is a common used method in most modern games today, there is very little source code available online. This made it harder to base my project on and made it harder to get a base understanding of the actual code. I generally struggle on where to get started with projects and how to begin coding, so I felt taking on this project would be suitable to improve my skillset.

I struggled with the logic behind my Run Utility state. At the start and up until the extreme late stage of implementation, the Run state seemed like a well thought out state that would add some varying behaviour to the agent. However, I failed to find a suitable formula that would combine several different variables into a reasonable utility curve. I feel this could have been because of the limited amount of time I had to complete the project in and with the several other projects I had to complete, so I had to find a balance.

Even though the run utility seemed to fail for the most part, I feel I manage to implement the other curves well. At the start of the project I was extremely uncertain about how these would be implemented and if they would fail or not. All of them seemed to work out great apart from the run, which didn’t even fail by a considerate amount. These curves could use a little tweaking in certain parts, but I could have spent at least half of the project tweaking them until they were at the perfect amount.

The game I have implemented seems robotic at first glance. It currently moves from one state, and moves to the next highest. This makes the agent easy to predict. At first I felt I had very little time to implement my solution, so I picked easy states. Doing some initial research allowed me to see that there was no source code, so I tried to keep the game simple. To avoid this robotic like AI, I could have added more states, or planned my game differently. I could have made it a 3D game instead of a 2D, or I could have used a user instead of two AI agents. Originally, I wanted to implement some steering behaviours to help with this, but due to the project constraints I thought I would struggle enough with the project without adding more AI to it.

I really enjoyed the insight into the general field of artificial intelligence and how such a simple programmable AI, could result in a multitude of different behaviours. I specifically enjoyed Utility AI and Procedural Content Generation (PCG). When deciding what to do my project I was torn between these two topics.

On one hand, PCG provided a great way to create new and unique content. In a game like Borderlands or Minecraft, these methods are ever present as they make sure the player can have a multitude of experiences every time when playing the game. I chose Utility AI over PCG because I felt more in tune with it. Although a decent amount of games use PCG nowadays, more modern games seem to use Utility AI. I also felt like I wouldn’t have been able to create a PCG game that I would have been proud of in the given time frame of the project.

In the future, I can take a lot of lessons from this project forward with me. The ability to come up with code purely from scratch is a skill I have been lacking for a while now. This project gave a chance to take this skill head on and not have to worry about if the project was a success or not. The room for failure here helped me understand that the researching and studying about the different authors who have implemented Utility AI, was more important than building a state of the art Utility AI.

When talking a project on like this in the future, I would structure my overall design of the code better. Right now, my main AI script is overloaded with methods that could maybe be put in other classes. Currently, I am using a script organiser (#region) to organise certain parts of the code, but overall it could be more efficient. In the future, I will write out how I want the design of my code to be structured on paper first. This gives a great visual aspect of how each system is connected without having to make it up as I go along.

I wold als

References

Rasmussen, J. 2016. *Are Behaviour Trees a Thing of the Past?* [Online] Available at: <http://www.gamasutra.com/blogs/JakobRasmussen/20160427/271188/Are_Behavior_Trees_a_Thing_of_the_Past.php> [Accessed 25/10/2016]

Graham, D. 2013. An Introduction to Utility Theory. In: Rabin, S. ed. *Game AI Pro.* Florida: A K Peters/CRC Press, pp. 113-126

Zubek, R. 2011. *Needs-Based AI.* [Online] Available at: <http://www.zubek.net/robert/publications/Needs-based-AI-draft.pdf> [Accessed 27/10/2016]