Game Artificial Intelligence Coursework – Jack Oswald

Project Proposal

The purpose of this coursework is to investigate, research and implement an AI technique or algorithm on one of the given areas: Steering Behaviours, Searching (Pathfinding), Decision Making or Procedural Content. 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. 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. I am researching decision making as it forms the core of any AI system in most of today’s games. 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.

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.

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.

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 fallback 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

This project will use a locking mechanism for each action. Since there are only 4 actions, and each have clear implications of when they start and end, this will be the most effective method to implement.

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 agent 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.

The attack method starts off by finding a random 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 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. A public method is then created so which adds health to the agent based on the parameter on the method. This method can be called by any script.

The agent’s health and ammo values are displayed at the top left of the screen for visualization purposes.

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 a specific 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. A quick changed 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. The Heal Utility value is based off the Sigmoid Curve, as shown in Figure 1 below. It uses a basic logistic function; this function gives the biggest rate of change in the center 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.

The Reload Utility curve is based on a quadratic curve as shown below in Figure 2. This curve allows an increasing rate of utility based on the agent’s current ammo count.

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.

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.

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]