Assessment 1: State Machine

## Dylan Welch-Madgwick 807178229

# Task 1:

**5 basic syntax rules are:**

* Semicolons are used at the end of lines in C# to make structure clearer and more organised. While in Python a semicolon is used to denote different statements on the same line.
* Colons are used in C# to specify that a class implements an interface, while in python a colon is used to “slice” code allowing access to parts of a sequence.
* Curly braces are used in C# as flow control while in Python they are used to define data structure.
* When determining if two variables have the same value two equals (==) signs are used in C#. While in python only one equals sign (=) is used.
* While trying to comment something out in C#, // is needed to comment out a line. While in python the character # needs to be used.

**The main architectural differences between Unity3D (unity) and Unreal Engine 4 (UE4):**

Terminology is a big difference between the two. The same/similar items or tools being called different things. For example, Game Objects in unity being called Actors in UE4 or the Hierarchy Panel in unity being called the World Editor in UE4. The coding language is another main difference between the two. Unity using C# and UE4 using C++. Another main difference would be the different features such as Blueprint (visual coding) being a focus of UE4 whereas in unity it is an extra feature by no where near has focused on.

**5 common programming principles/techniques to improve code:**

* You Ain’t Gonna Need It (YAGNI). Ensuring that only needed code is added/developed can help ensure that a programmer stays on track and doesn’t get overwhelmed.
* Don’t Repeat Yourself (DRY). Ensuring that code isn’t repeated unless needed. Create variables when able to, making it easier and quicker to program and develop things.
* Keep It Simple Stupid (KISS). Ensure that your code doesn’t get more complex than it needs to be, if you can make it simpler, do so. This helps make the code more readable for everyone and makes maintenance easier in the future.
* Naming Conventions. Ensuring that classes/variables/scripts/etc follow a naming convention so that all are consistent. This will help when scripts become more complex and class/objects/etc are being called on.
* Unit Tests. Testing small sections of your code to see if they work as intended. This can help illustrate how you could improve your code or why the code as a whole isn’t working.

**Software Development Lifecycle:**

This is the method/plan that is used in order to develop software applications. Usually divided into 7 categories; Planning, Design, Build, Document, test, Deploy and Maintain. SDLC’s help ensure that the development of a software application is done properly and efficiently whilst also staying on track with the timeframe. There are multiple different SDLC models that are used in the industry with their own pros and cons. Some of those are;

Waterfall Model: is a popular linear sequential approach that focuses on one section after the other.

Requirements>Analysis>Design>Development>Testing>Deployment>Maintenance

Some of the pro of this model are that it is a simplistic SDLC, this model allows for accurate cost estimation and timeframe. This model works well with deadline or Milestone Oriented Team/workplaces.

However, some of the cons are that this model is very restrictive when is comes to changes after initial design, clients are usually excluded from the process and testing for what is being developed comes late in the model.

Agile Model: is based on collaborative making involving requirements and solutions. Focused on producing working portions of the program before moving on to the next part. Usually don in cycles

Planning>Requirements>Designing>Development>Testing>Loop

Some pros of this model are that it can accommodate changes at any time during development, allows testing of code early on and is effective for a dynamic working environment.

Some of the cons of this model are that changes can occur at any time and doesn’t work as well for larger teams,

Test Driven Development Model: is based on developing a program to pass all given tests, improving code until this occurs. Once all tests have been past then the development is done.

Some pros of this method are that bugs, and code are clean up early on in the development and code would be kept to a minimum as to keep maintenance easy.

Some cons of this method are that the programming speed are a lot slower compared to other models. This model doesn’t accommodate for changes well either, if changes are made at any time the test need to be remade/restructured.

DevOps Model: is a model focused on using practices that emphasize collaboration and communication to build, test and release applications rapidly and reliably.

Development>Integration>Testing>Deployment>Monitoring

**AI Implementation**

The basic idea of the game would be to have an ai and player go head to head. Trying to collect resources and build up an army to protect their power generator whilst also attacking the enemy’s power generator. The Ai would have to be designed in a way to make it difficult for the player to win but not impossible. Bad decisions by the player would result in consequences while good decisions should be rewarded.

The Ai would ultimately have different objectives to the player. The players objective would be to destroy the AI’s power generator, the AI’s objective would be to lose, but to not make it obvious that that is its intent.

The AI would be programmed (depending on difficulty) to make certain bad decisions randomly to give the player the upper hand or even just a chance to recover from a bad choice they made. This would keep the player interested and ensure that they are going to lose after one mistake.

Another thing to consider is that the AI should be a good mixture of defensive and offensive. Ensuring that there are several small battles before the finally big one or they are defeated. If the ai just built up and defended the player could get bored, if they ai just constantly attacked it wouldn’t give the player time to build up and they may become frustrated.

Having a chance, the AI would respond to different actions the player made would also keep the battle interesting and not predictable. For example, having the AI send out a small group to attack the players exploration groups or farming groups would make it seem like the AI is trying to stop them, however making it a chance instead of a set decision would give the gameplay some variation and encourage the player to adapt. Another example would be the AI responding to the state of the AI’s power generator. At low health the AI would want to defend more but if it was at full/high health it would explore/attack more.

**Budget and Timeline**

These ideas would affect the timeline and budget. Having the AI respond more and act randomly (within reason) at times would require more advanced code and testing requiring more development time. Meaning a longer timeline and a bigger budget. However, the ideas listed above aren’t anything new/revolutionary, meaning that it would be easily technically feasible and wouldn’t be difficult for the players to understand what is happening.