TASK 1:

Syntax rules:

- C# Ignores whitespace, Python does not.
- C# uses a semicolon ';' to end a line of code, Python uses a semicolon to separate code (Instead of a comma like C#) and forms a newline with a backslash \.
- C# uses a class system for object orientated programming, JavaScript does not have a class system and therefore does not include things like inheritance.
- C# uses a capital letter when programming a method, Python uses a lower case letter.
- C# uses curly braces {} to indicated a code block, Python uses indentation.

Architectural differences of Unity 3D:

- Unity 3D can target pretty much any platform including mobile, Unreal engine is mainly console and PC.
- Unity 3D uses 'normal' scripts to code, Unreal Engine uses visual scripting as a basis.
- Unity 3D uses C# as its programming language, Unreal Engine uses C++
- Unreal Engine has better overall graphical capabilities which includes: Physically-Based Rendering (more on Unreal Engine Rendering here), Global Illumination, Volumetric lights out of the box, Post Processing, Material Editor. Unity 3D has Physically-Based Rendering, Global Illumination, Volumetric lights after a plugin installed, Post Processing. It is still possible to achieve a good looking game with Unity 3D, but there is more work involved.

Coding Principles to improve:

- Keep it simple, if something can be achieved in a single line of code, then write a single line of code. Do not overcomplicate things for the sake of it.
- Don't Repeat Yourself (DRY) if a block of code is using duplicating lines taking up a lot of space, you may be better off programing an algorithm that does the same thing.
- Single Responsibility, do not overload classes. Each Class should provide a specific function and should be broken up into smaller classes if the main class becomes complicated.
- Leave comments, often we forget what a function does, especially if there is complicated math involved, keeping a detailed comment section helps when going back on 'old' code.
- Write clean code, don't overcomplicate, don't try and fit a ton of logic into a single line, comment often and ensure your code is easy to read.

What is an SDLC:

An SDLC is a method for creating high quality software. An SDLC has defined processes to follow, such as: Requirement Analysis, Planning, Software Design, Software Development, Testing, Deployment.

Waterfall: This SDLC has its processes 'flow' downwards in order, using the above example it would follow in this order: Requirement Analysis, Planning, Software Design, Software Development, Testing, Deployment.

- Pros – Easy to use and understand, Easy to manage as each ‘phase’ has a specific function and review process, works well for smaller projects.
- Cons – Doesn’t allow for much revision, once in testing phase going back is difficult, not advised for complex or OOP projects. Because Testing is done last, problems not foreseen earlier can become problematic.

Agile: Like the name implies, this SDLC is designed for small rapid development, testing and deploying cycles, where the software builds up from each successful deployment.

- Pros – Higher Customer satisfaction with rapid continuous, tested deployment, regular review / adaptation to changing circumstances or requirement, can adopt late changes easily.
- Cons – Difficult to assess deliverables if they are large projects at the beginning of the SDLC, hard to implement if you are not a senior dev, can easily get off track if the customer isn’t clear on what they want.

Test Driven Development: this SDLC is the idea that you write and run tests BEFORE writing the code, having the test fail and then write enough code to have it pass.

- Pros – As you test in small increments early, code is written to be more modular, TDD promotes better code architecture by writing tests first the architectural problems surface earlier, makes collaboration easier as the tests will let colleagues know if their implemented code will behave in unexpected ways.
- Cons – Initially slows down development, Tests may be difficult to write, Is not easy to learn, is difficult to apply to existing legacy code, any early state refactoring requires the test classes to be refactored as well, unless the whole team maintains their tests the system will fall apart.

DevOps: This unites the Development Team and the Operations team together, rather than having them separate. Instead of Devs developing, testing and deploying systems then Ops using and reporting errors, Devs and Ops now collaborate.

- Pros – Higher productivity, shortened production cycles, clear product vision within the team, better team efficiency, reduced chance of product failure.
- Cons – Requires the right mindset across the team, dealing with any legacy systems is hard, low security if the ops are outsourced, if communication is poor this SDLC doesn’t work.

TASK 2:

How the AI should react: AI should send out small scouting parties to each strategic location, if no player soldiers are found they should hang around, if destroyed by the player, another slightly larger wave of NPC soldiers should be sent to this location, if buildings are found in this location, the response should be increased again slightly to challenge for the position.

The AI should slowly ramp up all attacking force sizes on the generator giving the player time to react and understand how the AI is going to continue to throw more at them over time.

A state machine should be implemented to increase the aggression level, e.g if the NPC scouts find no player soldiers at a strategic location, the state stays at minimal aggression, the AI builds gathering structures and assigns a small task force to protect the assets. Once player soldiers arrive the scouting party / task force can do a check to see how many soldiers the player has and have the state machine react accordingly, for example if the player soldiers outnumber the AI attack / scout by a small margin, the AI should train troops to equal the amount of player units last seen then attack the location, if the AI manages to destroy the players units, then it lowers the aggression state and sends no more reinforcements / minimum reinforcements. If the AI is defeated, it now meets the same troop numbers plus a few extra, or fleshes out more weaker units to appear larger before attacking so that the player is forced to react with a larger force presence or lose the position.

If no player fighting units are found, the AI should destroy any buildings or harvester units then stay at the strategic location / build their own harvesting units until the player retakes it. Send scouting party to assess aggression level, then meet this level, if AI forces are defeated, raise aggression level. If AI forces are victorious, lower aggression level and do not reinforce above the minimum. The AI should scout all strategic resources and target the least defended ones first.

I believe this AI structure would work because it would become clear to the player without telling them, that the AI will continue to match / challenge the players position until the objective is complete, encouraging the player to always ensure there is a sufficient force of resistance at the strategic locations and not to leave any locations undefended for an extended period of time, it would also help players notice scouting parties and if a location is scouted first, to send a response team to that location knowing the AI is likely going to attack there soon.

I cannot see this Ai implementation being considerably expensive or time consuming to implement, it would use a player unit number check to change the state, then send the NPC units out to the location to fight, make the same check, then react.