# SCHOOL OF COMPUTING

# UNIVERSITY OF TEESSIDE

# MIDDLESBROUGH

## TS1 3BA


# AI Civilization Simulator

## Computer Games Programming BSc (Hons)

**********

## 8th May 2017


**Supervisor:** **********

**Second Reader:** **********

# 1 - Abstract:

Detailed within this report are the finding and the development procedure taken to create the project “AI Civilization Simulator”. Within you will find the processes taken into the planning, research, development and testing of the resulting project.
This project has been developed using the Unreal Engine 4 (Specifically version 4.12.5) with C++ written in Visual Studio 2015. Unreal Engine 4 is a video game engine developed by Epic Games and is the source of the functionality found in this report unless stated otherwise. The Unreal Engine 4 is well known for its Blueprint Visual Scripting System however this project was aimed at avoiding using it for the coding language C++ unless it was deemed necessary, an example of this is the modeling.
The primary aim of this project was to demonstrate that due to the technological advancements of today compared to 15-20 years ago, it is now possible to create more complex AI from within a game engine environment. The AI techniques that have been used to demonstrate this are Finite State Machines and Behaviour Trees which are both running simultaneously within the gameworld. These are demonstrated in the form of a small civilization with the citizens taking advantage of the Finite State Machine and the Town taking advantage of the Behaviour Tree making decisions.

# 2 - Acknowledgements:

I would like to give thanks quickly to a few things that gave me the inspiration and guidance for this project.
Firstly is my Tutor Dr. Eric Silverman who guided me in the right direction with this project and provided the inspiration needed to work through it.
Secondly the setting and style of how the project was laid out was inspired by the god games of the early 2000’s.
Lastly is my family who helped provide the motivation to keep working at the project despite what problems I was facing with it.

# 3 - Introduction:

The overall concept of this project is to create a pair of AI, a behaviour tree and a finite state machine which will be used to manage the production and roles of a civilization and its people to achieve the best possible outcome overall, a measure intended to be called Wellbeing originally but due to time constraints (described later in this report) it was replaced with trying to demonstrate that AI's of certain complexities no longer create such as a demand on an environment as they used to while still providing the same functionality and capabilities.
The civilisation in question will maintain simplicity of particular strategy games focusing around the concept of gathering resources such as wood and stone to progress the civilisation by constructing buildings however also including particular survival game aspects such as the citizens require food and need to rest after working otherwise it will be detrimental to their health. To make this work a single behaviour tree will be used per civilization, having access to all the information available from the world and its available resources it will be deciding where people need to work and what will be build, on the flip side a finite state machine will be created for each individual citizen, this will make sure that they are doing what they are required to do such as eating, sleeping and working.

# 4 - Methodology:

The Approach that was taken to developing this project was style of prototyping method, developing and testing each developed stage of the project. This will be done like this due to having 2 primary components to it the finite state machine and the behaviour tree, they be able to be developed independently due to the roles that they will be playing in the simulation. The finite state machine will be the citizens so the functionality can be worked on, built and tested without interference from anything due to it working by itself and the behaviour tree will be focusing on building and role sorting so this can be focused on once the other piece has been completed.

# 5 - Research:

## A - Finite State Machine:

Finite State Machines or just simply State Machines are automation that include a finite number of states that it can switch between be can be done autonomously without the involvement of an external force. They are able to make decisions based on the state their in and the variables provided to them.The example

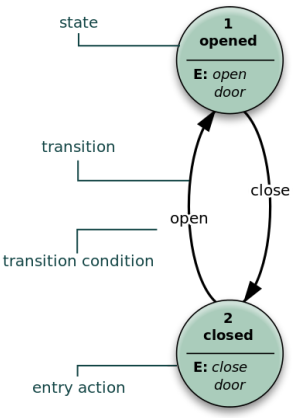



Figure 1. Finite State Machine Example: Door (Wikipedia, 2007)

provided here in Figure 1 (Wikipedia, 2007) shows a very basic Finite State Machine of a door. It contains 2 states, open and closed and it shows the transitions between them with the conditions required for it to occur.

## B - Behaviour Tree:

Behaviour Trees are a collection of tasks and conditional variables structured in a way as to generate simulated behaviour in an AI. Behaviour trees manage how it needs to operate with a series of nodes of which functionality can vary between tree's based on their required design. Commonly found designs for these nodes are one's called Selector nodes and Sequence nodes.

Selector nodes work on the functionality that it will execute its tasks until one of them returns successful. As shown in Figure 2 is an example of a Selector Node succeeding, as shown it does not require all of the task to succeed or be called at all, once one of the tasks succeeds the node will return a success.

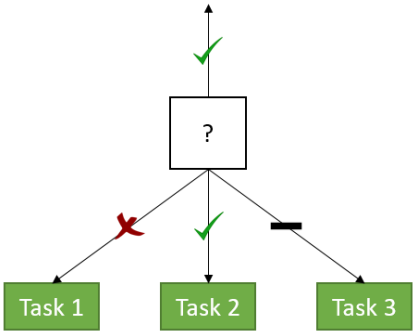



Figure 2. Selector Example

Sequence nodes work on a similar functionality to Selector nodes however their primary difference is that they require all of the tasks to succeed to return a success, as shown in Figure 2 Task 1 returned a success however Task 2 returned that it failed so the Sequence node ended and returned that it had failed. An example of a complete tree is shown here in Figure 3. As you can see from the example which is of a dual armed robot grasping objects it contains a root noot which indicates that start of the tree at the top, a sequence node and a series of selector and task nodes following them.

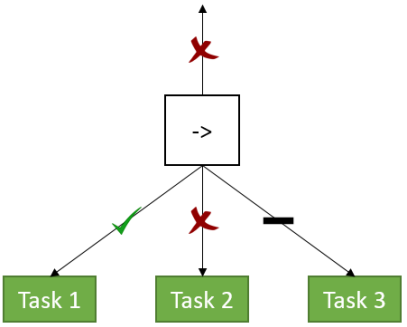



Figure 3. Sequence Example

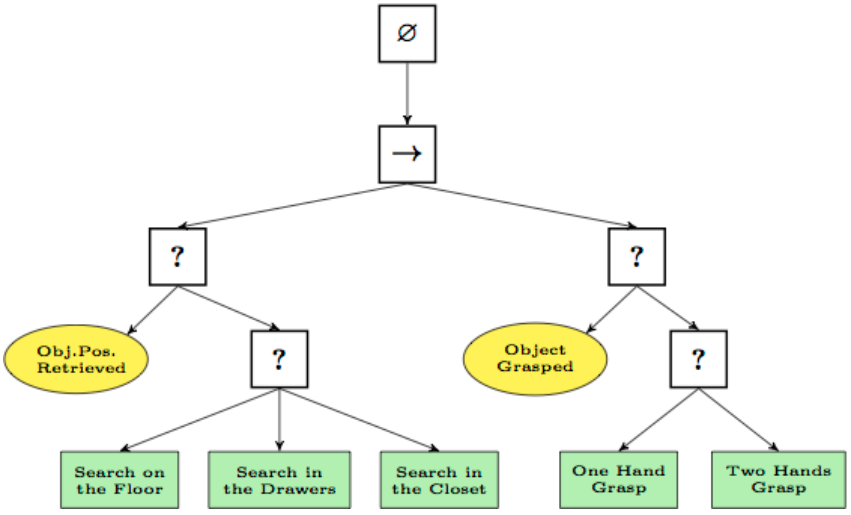



Figure 3. BT search and grasp (Aliekor, 2007)

# C - Unreal Engine Terminology:

As the project is to be created within the Unreal Engine research was required on whether if project is possible to complete within its environment. The first thing to analyse was the engines Tick system, the Tick system is a function attached to all objects within the gameworld and is called as fast as the engine can handle, subsequently meaning that if the engine would start to lag so would all objects currently running it. Secondly I found that the Engine contains its own integrated system for creating and running behaviour trees, at a first glance it appeared to originally contain functionality to create tasks from its blueprint system however after conducting some research it is distinctly possible to create any of the required nodes in C++ although their is only a handful of example code available to create these nodes from and even then these span over several versions of the engine. Despite this I think it would be a good learning curve to try and create these nodes for their trees as not only is it the first step into creating your own behaviour trees outside of the engine it also means that I would be a step closer to taking full advantage of all of the engines features available to C++ developers. An example of a behaviour tree in UE4 is shown in Figure 4.

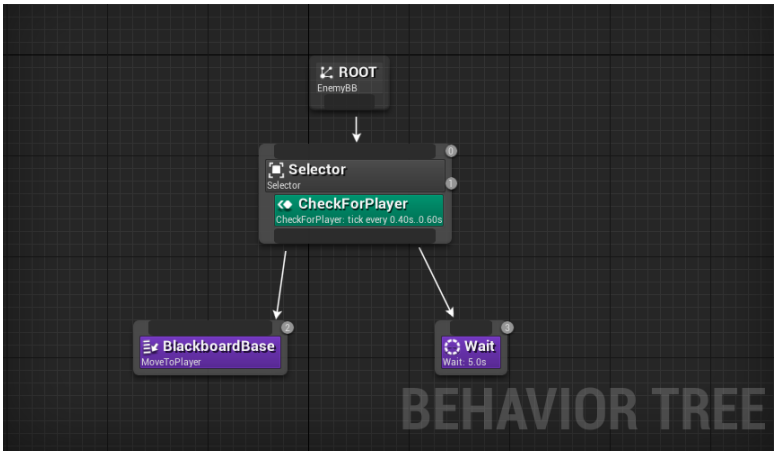



Figure 4. Unreal Engine 4 Behaviour Tree

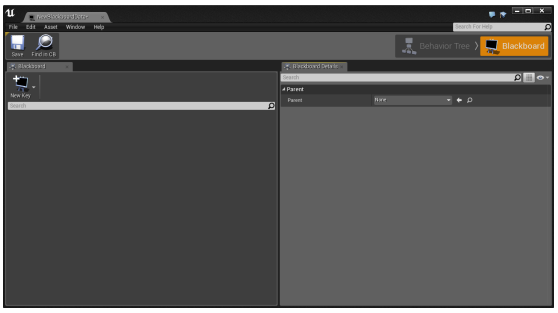


Figure 5. Blackboard example


Figure 6. Service example

The Unreal Engine's behaviour trees also have extended functionality integrated as well allowing users to efficiently make use of the trees. These functionalities include an extension to the behaviour tree called a blackboard which allows quick and seamless input of objects and variables that both the users and the behaviour tree itself can edit during game time as shown in Figure 5.To help alter any data on the blackboard without the need to add the functionality to task nodes the Unreal Engine includes an extension to the nodes in the tree that allow the data in the blackboard to be altered when the tree reaches a branch while its working. These are called services and don't affect how the tree makes decisions unless the tree depends on certain blackboard variables. How it is structured in the tree is shown in figure 6 as a green box.

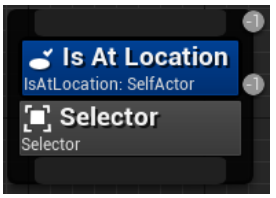



Figure 7. Decorator example

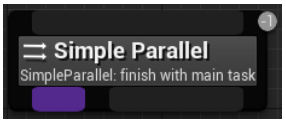



Figure 8. Simple Parallel node example

An additional extension to the nodes is also included called decorator nodes, these are completely different to services which don't directly affect the behaviour tree whereas decorators do. Decorators shown in blue act as conditional statements that can stop the

progress of a tree before the node is even called. Some presets included are checking if their at the required location as shown in figure 7 or other types such a loop.
Also added is an additional node like the sequence and selection nodes mentioned previously, it is called a simple parallel node. What this does is that it allows a task node to be called and executed at the same time as another node, it returns whether it succeeded or not the same as the other nodes and allows the option to return once section is complete or after both are complete . Figure 8 shows a simple parallel node.

# 6 - Design:

## A - Finite State Machine, The Citizens:

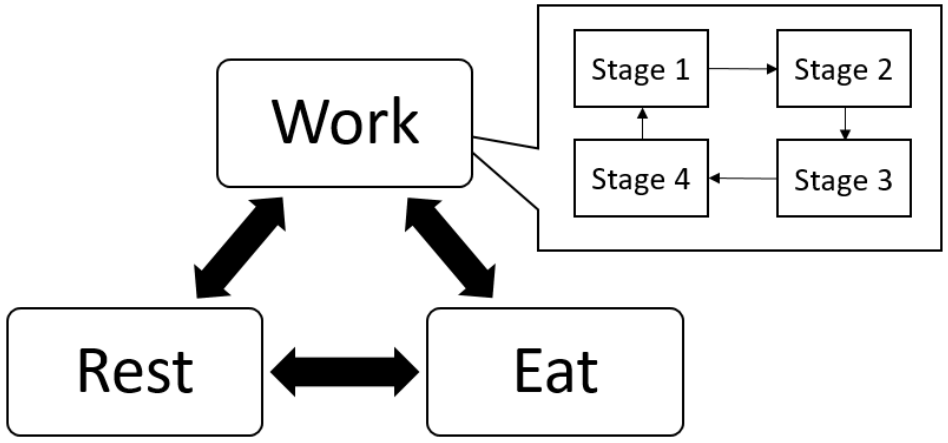



Figure 9. Finite State Machine Plan

For the citizen to work as intended they need to be able to look after themselves and be able to carry on with their allocated task, to allow this 3 primary states will be used to distinguish what the citizen needs to be doing. Each of the 3 states will be able to switch into either of the other 2 when required, as shown in Figure 9 the 3 states will be for when the citizen is either working, eating or resting. Eating and resting will take precedence over the working state as they are states making sure that the citizen does not die due to ether stress or hunger.
Both the eat and rest nodes will follow similar principles, once the citizen has reached either the hunger or stress (respectively) then the node will be triggered and perform its actions.
For the eat node the citizen will stop working and then make their way home to consume the food stores their, if their is no more food at home then they will make their way to a storehouse which has available food, preferably one which is close to the citizen. Once there the citizen will consume food until their hunger reaches 0 or a predefined value, then they will restock their house with food and continue back to work.
For the rest node the citizen will stop working and then proceed to their allocated home, from their they will enter their home and proceed to rest (reduce stress) until it reaches zero or another predefined value of which they will leave their home and move on to their next task.

Last to cover is the work node, as shown on Figure 9 it has more complexity to it than the other two nodes. On the right hand side of the node is a look at what is to be completed to repeat the node, almost like a finite state machine within a finite state machine. Each stage represents a part of the where the citizen is in the work cycle and this will be repeated until the main state is changed away from the work node. Each stage effectively works as what the current task is for the citizen to do now and can't progress to the next stage until it has been completed. Stage 1 is for the citizen to move towards their current roles job location as if you're a farmer you can't harvest the food if you're not their. Stage 2 involves actually working their role until condition is met such as if the citizen currently can't carry anymore resources. Stage 3 involves taking the resources their carrying to a storehouse that has the space for it and lastly is stage 4 of which is the action of depositing the resources in the storehouse. After stage 4 is complete a reset function will be called and everything repeats itself.
The reason why to it being structured like this is taking into consideration Unreal Engines tick function as described earlier in this report. To make the finite state machine work within the Unreal Engine you can't make it run separately to everything else it will be put within the citizens AIController which is the brain of the character. It needs to be in here as it needs so to be able to have access to the characters variables and also use the AIController movement functionality. Because of this to make sure the finite state machine is running it will be called within the tick function and re-enter each state with each tick, for the eat and rest states they are simple enough to just move to an area, they are their then they can either eat or rest and there are no problems but due to the work function having more parts to it, it required more planning on whether it can switch between each stage of the state.

# B - Behaviour Tree, The Town:

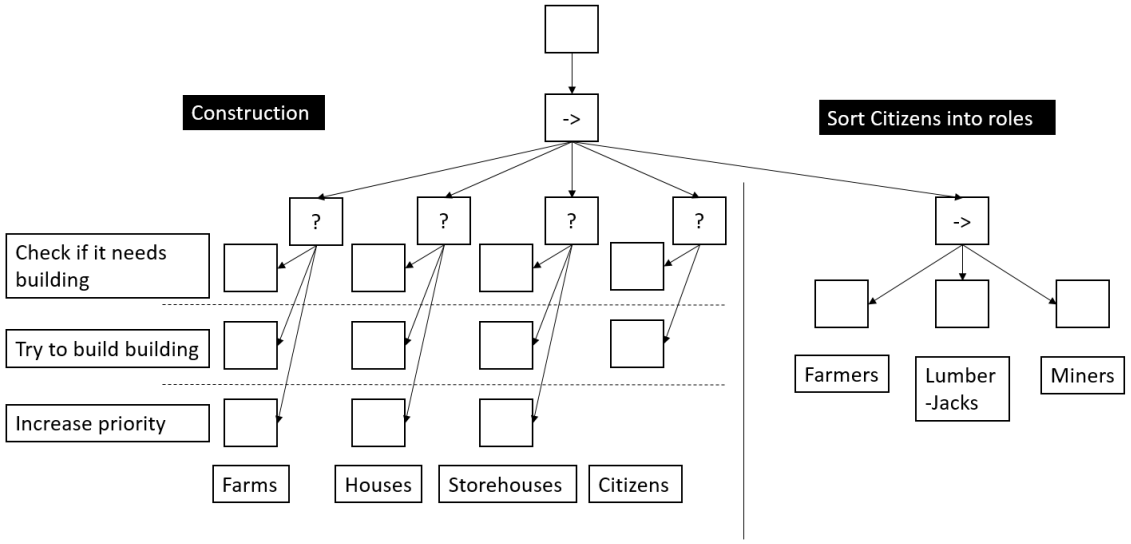



Figure 10. Behaviour Tree Plan

After the citizen is complete it will be time to move onto making the behaviour tree. The functionality of the tree will be attached to what will be the Town hall, which will be acting as the center of the the civilization. The Town hall will be the host of all the information about

the civilization, from their it will have pointers to all citizens and buildings while also keeping track of important information such as the total resource count, how much food is coming in and out and the locations of all surrounding resources.
From what has been found about initialising a behaviour tree, it must be called from within a AIController so the Town hall will possess its own AIController. From their the behaviour tree should have access to all the information the Town hall hold however due to a lack of information about C++ and behaviour trees this isn't guaranteed but because of the blackboard feature this can be worked around.
Looking to the structure of the behaviour tree, figure 10 shows the a plan for the behaviour tree. Overall it's main job is to build building that are needed and to sort citizens into the roles required for the situations. These 2 sections will be shortened to construction and sorting. In the construction section there will be 3 tasks for each option except for when it builds citizens, of these 3 tasks the first to be called will be a check to see if it needs to build that particular building. These checks vary by the type of building it will be checking against, for farms it will check it compared to if the civilisation is in a food deficit, homes will be checked against if there's enough homes to house the total citizens, the storehouses will be checked against how close the resources are to hitting their maximum and lastly the citizens will be compared to the food rate and total food. Next will be the task to try and build the building/citizen, this will have attached to it a decorator which will check that there is enough resources to execute the task to build the building and lastly will be a task node that increase priority of gathering resources needed to build the selected building. Building citizen won't have this check as to not grow out of control.
Next is the sorting section, once construction is finished the behaviour tree will move onto sorting the citizens into the needed roles, for farmers the check will be if the town is in a food deficit and if there is not enough farmers/farms then the behaviour tree will resort the roles to fill the farmers. For lumberjacks this check will be for the amount of priority that's been set, allowing for more people that can be set to lumberjack so that building can get built. And lastly for miners as it's something for later development it will only be set in ratio to the maximum capacity for stone, miners will be given the lowest priority of all 3 types of roles while allows them to move into the other 2 roles when needed.

# 7 - Implementation:

## A - Initial buildings

Before the construction on the finite state machine can commence the objects in the game world need to be built first, the objects include the main buildings in the design so this includes the house, storehouse and the farm, and also the resources which includes the trees and the rocks.
To simplify the mechanics the farm will be included in with the resources with its inheritance properties.
First stage of development began with the base classes of the buildings and the resources, starting with the the buildings as it will only be covering two buildings they will share the same variables as each other but set to different values on initialisation in the child classes. The base building class contains variables on how much resources it's currently holding, as

food is needed in both the storehouse and the houses their values are set appropriately in the child classes otherwise they are initialised to zero. Other variables set up also include arrays to store pointers for the citizens currently associated with the building, as it's stored as a TArray the pointers don't needed to be initialised which saves time performing checks for null pointers.
Below in figure 11 shows my header file with all my values assigned for the buildings. The values for Health and Tier were never used and where though of as early for features I never got around to.

```
int Health;
int Tier = 0;

TArray<class ACitizenBody*> HousedCitizens;
int32 CountedHousedCitizens; // Counting current housed citizens

TArray<class ACitizenBody*> AssignedCitizens;
int32 CountedAssignedCitizens = 0;

int32 MaxHousedCitizens; // used to compare assigned and current housed
bool CanHouseCitizens;

int HousedFood = 0;
int MaxFood;
int HousedWood = 0;
int MaxWood;
int HousedStone = 0;
int MaxStone;
```

Figure 11. Building.h

Next is the resource class, this class only contains the values for how much resources it stores, the type of resource and whether the resource has died or not. For trees and rocks they are both the same in functionality mainly differing in the amount and type of the resource, for the farm however it requires the ability to generate food so some functionality was added to it to make it create food, once the farm has the max food it stops growing it stops making food and signals that it can be harvested. Farms also required a pointer to the citizen that is currently assigned to working it, this is so citizens are able to check they are the ones assigned to the farm and if not they will find one that has not got anyone assigned to it. Below in Figures 12 and 13 show the variables in the headers for the resource class

```
int Health;
bool Dead = false;

int TotalResource = 1000;
int ResourceType; // 0 = Food, 1 = Wood, 2 = Stone
```

Figure 12. Resource.h

and the farm class.

```
int ResourceType = 1;
int MaxResources = 1000;

int RegenTimer = 10;
int counter = 0;

bool Worker;
bool HarvestReady = false;

int OwnedTown;

ACitizenBody* WorkingCitizen;
```

Figure 13. Farm.h

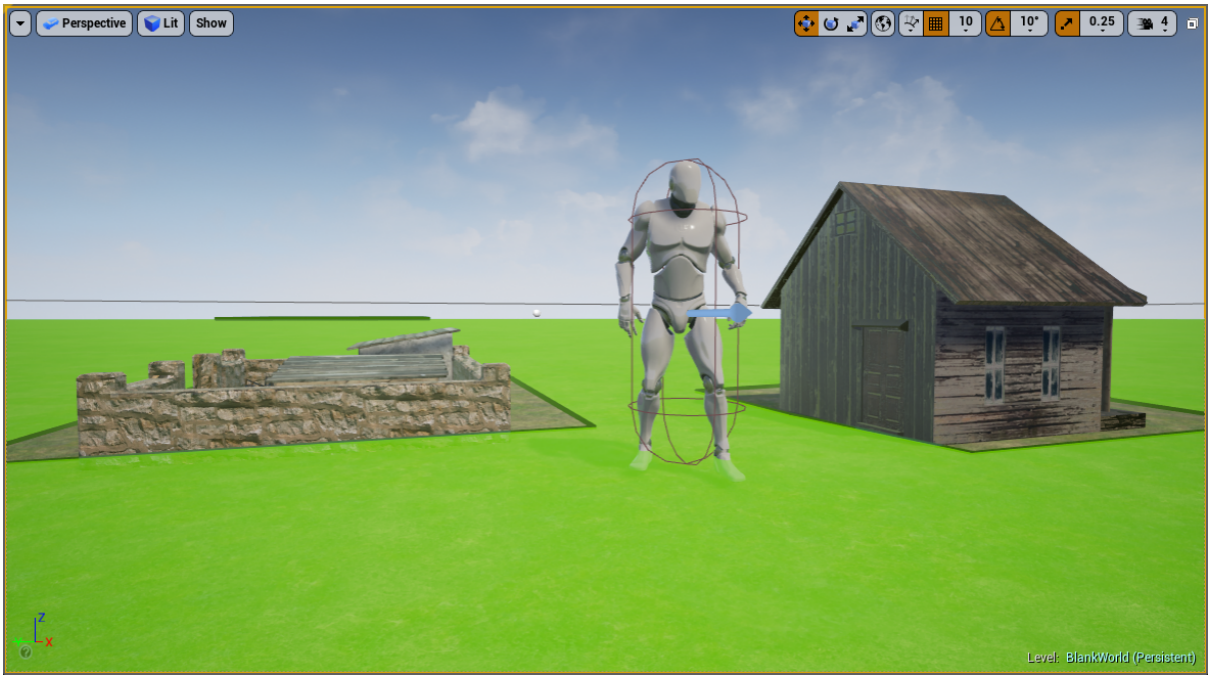
To provide the visual representation of the objects in the game world, blueprint children were created in the editor. Blueprint children of C++ classes maintain the same functionality as if it was entirely done in C++ except for the added functionality needed to be added to the files such as asset address locations, positioning and texturing. The blueprints allow already imported files to be dragged and dropped into the required buildings quickly creating a visual representation. This was greatly useful with the citizen as with the default mannequin assets, it includes an animated skeleton mesh showing a better representation of a person. Figure 14 show the character shown in the editor.


Figure 14. Citizen Mesh within Unreal Engine

# B - Finite State Machine:

## Citizen base:

The first part of the citizen to set up was the movement capabilities. Making a character move within unreal is an easy task as if it has an AIController attached to it with a movement component setup on the mesh and the game world has a NavMesh active, a function can be called named MoveToLocation and the AIController will take over the movement, find a path on the NavMesh and automatically move the character to the point provided. The difference in the situation however is that buildings will be spawned in the world and usually the NavMesh is created static so it won't update during runtime meaning the and buildings placed during run time meant the character would not run smoothly to any locations provided or get stuck entirely. To get around this problem the engine allows to operate using a dynamic NavMesh however with a large area and many obstacles it can slow down the game, also to make the movement work all obstacles required to be set as an dynamic obstacle except for the character, for the character its collision with the NavMesh had to be removed as since it was identifying itself as an obstacle and not drawing the NavMesh area around them preventing the AIController from finding a point to any location. Figure 15 shows the NavMesh generation, on the left is a storehouse model, center is a citizen and the right shows a house model. The green area painted around the buildings is the NavMesh and the area where the AIController can move to, notice the citizen does not block the




NavMesh.

Next to develop is the primary functionality required for the Finite State Machine to work with. These are variables and functions that couldn't have been debugged until after the Finite State Machine was working as that is the time when they were used. Variables included are pointers to required targets such as the resources or building destinations and functions to enter buildings or to perform tasks. Figures 16 and 17 show the variables and

functions used in the citizen. UPROPERTY() and UFUNCTION() are features added to the Unreal Engine which both adds a protective layer to the engine, so if variables such as pointers were left unused or were used then deleted, originally left unchecked would crash the engine but if assigned UPROPERTY() it acts as a safeguard, it doesn't stop crashes entirely but they do help prevent them. Another purpose of them is to expose variables to Blueprints which helps speed up spawning procedures. An example of this, at the bottom of Figure 17 is a linker to a blueprint. Within blueprints the appropriate blueprint class can be selected and in C++ that item can be spawned into the world quickly and efficiently using the world function “SpawnActor<>()”.

```
int Health = 100;
int Stress = 0;
int StressTimer = 0; // Timer for intervals taking on stress
int Hunger = 0;
int HungerTimer = 0; // Timer for intervals for becoming hungry

UPROPERTY(EditAnywhere)
    int StressThreshold = 70;

UPROPERTY(EditAnywhere)
    int StressResetThreshold = 0;

UPROPERTY(EditAnywhere)
    int HungerThreshold = 70;

UPROPERTY(EditAnywhere)
    int HungerResetThreshold = 0;

// future work
int Age;
bool Gender;

int TownNum; // used to check associated town, if many are present.

int NumResources = 0;
int CarryWeight = 100;
int ResourceType = 0; // 0 = none, 1 = food, 2 = wood, 3 = stone

UPROPERTY()
    ACitizenBody* Partner = nullptr;

UPROPERTY()
    AHouse* Home = nullptr;

UPROPERTY()
    ABuilding* TargetBuilding = nullptr;

UPROPERTY()
    AResource* TargetResource = nullptr; // Dropped Resource

UPROPERTY()
    AFarm* AssignedFarm = nullptr;

UPROPERTY()
    ATree* TargetTree = nullptr;

UPROPERTY()
    AStone* TargetStone = nullptr;

UPROPERTY()
    ATownHall* MyTownHall = nullptr;

UPROPERTY()
    AStoreHouse* TargetStoreHouse = nullptr;

int Wellbeing;

int Role; // Dont directly change, only change with RoleChange as it is used to override
int RoleChange; // if changed it will reset current work task then change role to this number
```

Figure 16. Citizen.h part 1

```
//////////////////////////////////////////////////////////////
bool AtHome = false;
bool AtTaskLocation = false;
bool Resting = false;
bool RestingPenalty = false;
bool Working = false;
//////////////////////////////////////////////////////////////

UFUNCTION()
    void DoTask(int ToDo, bool AtStoreHouse = false);

UFUNCTION()
    void Eating();

UFUNCTION()
    void StressDamage();

UFUNCTION()
    void HungerDamage();

UFUNCTION()
    void CalculateWellbeing();

UFUNCTION()
    void EnterHome();

UFUNCTION()
    void LeaveHome();

UFUNCTION()
    void DropResources();

UFUNCTION()
    void Sleeping();
int restTick = 0;

UFUNCTION()
    void EmergencyStop(); // if triggered rest until Stress and are Hunger < 10
    bool EmergencyStopActive = false;

UFUNCTION()
    void CheckEStopStatus();

UFUNCTION()
    void ConfirmPartner();

//////////////////////////////////////////////////////////////////

UPROPERTY(EditAnywhere, Category = "Resource")
    TSubclassOf<AResource> DropStuff;
```

Figure 17. Citizen.h part 2

## AI (Finite State Machine Part):

After the functionality for the citizen was created next was to build the Finite State Machine, using Figure 9 as my guide, the first pieces to make were the eat and rest phase as they were the simplest to implement. Using pointers to test building placed in the world I set up the functions in two parts, the first for moving and the second part for eating or resting. A check has also been placed function to automatically change the state back once the citizen doesn't need to eat or rest any more. Figures 18 and 19 show the eat and rest states within the AIController.

```
void AFiniteStateMachineAI::GoEat() // Go get food
{
    if (Cit->Home)
    {
        if (Cit->Home->HousedFood > 0)
        {
            if (GoToLocation(Cit->Home->GetActorLocation()))
            {
                Cit->AtHome = true;
            }
            Cit->DoTask(5);
            // eat
        }
    }
    else
    {
        for (const auto& Itr : Cit->MyTownHall->AllStoreHouses)
        {
            if (Itr->HousedFood > 0)
            {
                Cit->TargetBuilding = Itr;
                break;
            }
        }
        if (GoToLocation(Cit->TargetBuilding->GetActorLocation()))
        {
            Cit->DoTask(5, true);
            // Eat from storehouse
        }
    }
    if (Cit->Hunger < 10)
    {
        State = 0;
    }
}
```

Figure 18. Eat State .cpp code

```
void AFiniteStateMachineAI::GoRest() // Go home to rest
{
    if (Cit->Home)
    {
        if (Cit->AtHome)
        {
            Cit->Resting = true;
            Cit->EnterHome();
        }
        else
        {
            if (GoToLocation(Cit->Home->GetActorLocation()))
            {
                Cit->AtHome = true;
            }
        }
    }
    else
    {
        Cit->MyTownHall->AssignHousing(Cit);
        if (!Cit->Home)
        {
            Cit->RestingPenalty = true;
            Cit->Resting = true;
        }
    }

    if (Cit->Stress < 10)
    {
        Cit->LeaveHome();
        State = 0;
    }
}
```

Figure 19. Rest State .cpp code

Now to talk about the work function, this is shown through figures 20 to 24. First of all figure 20 shows the structure of the stages within the main work state self. Before starting the stages it will automatically identify if the citizen is performing his assigned role, if the role in the AIController doesn't match the one found on the citizen then the work state will reset. The four stages in the work state each have a different task to carry out separately to each other, stages one and three handle moving to the required locations and two and four perform the needed tasks. As stages one and three are for location confirmation they will be called during stages two and four respectively to confirm they are still at their required locations otherwise it will trip them back being false and restart the movement process for that stage.

```
void AFiniteStateMachineAI::Work()
{
    // 1 - move to work location, 2 - working, 3 - go to storehouse, 4 - put resources in storehouse, reset
    Cit->Resting = false;

    if (Role != Cit->Role) // if citizens role is changed
    {
        ResetWork();
        Role = Cit->Role;
        return;
    }

    if (!Stage1) // FIRST CHECK START
    {
        Stage1Work();
    }
    else
    {
        if (!Stage2) { // resets stage1 back to false if fails (double chack), once task is done then stop
            Stage1Work();
        }

        if (Stage1) // END FIRST CHECK
        {
            if (!Stage2) // START SECOND CHECK
            {
                Stage2Work();
            }
            else // END SECOND CHECK
            {
                if (!Stage3) // START THIRD CHECK
                {
                    Stage3Work();
                }
                else
                {
                    if (!Stage4) // resets stage3 back to false if fails (double chack)
                    {
                        Stage3Work();
                    }
                    if (Stage3) // END THIRD CHECK
                    {
                        Stage4Work(); // START FORTH CHECK
                        if (Stage4) // END FORTH CHECK
                        {
                            ResetWork(); // RESET
                        }
                    }
                }
            }
        }
    }
}
```

Figure 20. Work state .cpp code

For Stage 1 as shown in figure 21, determined from the current role of the citizen will at first get an available nearby resource location from the town and will then proceed to move to its location and until the citizen is with a certain radius the function will set Stage 1 to false. At the bottom I had intended to create a role for gathering dropped resources when there's nowhere for it to go however due to time constraints there was no time to implement it.

```cpp
void AFiniteStateMachineAI::Stage1Work() // Checks citizen is at the work location
{
    if (Role == 1) // Farmers and gatheres
    {
        if (!Cit->AssignedFarm) // if no farm
        {
            Cit->AssignedFarm = Cit->MyTownHall->GetUnassignedFarm();
            Cit->AssignedFarm->WorkingCitizen = Cit;
        }
        else if (Cit != Cit->AssignedFarm->WorkingCitizen) // if farm isnt theirs
        {
            Cit->AssignedFarm = Cit->MyTownHall->GetUnassignedFarm();
            Cit->AssignedFarm->WorkingCitizen = Cit;
        }
        else
        {
            Stage1 = GoToLocation(Cit->AssignedFarm->GetActorLocation());
        }
    }
    else if (Role == 2)
    {
        if (!Cit->TargetTree) // if no assigned tree
        {
            Cit->TargetTree = Cit->MyTownHall->GetClosestTree();
        }
        else if (Cit->TargetTree->Dead) // if tree has died
        {
            Cit->TargetTree = Cit->MyTownHall->GetClosestTree();
        }
        else
        {
            Stage1 = GoToLocation(Cit->TargetTree->GetActorLocation());
        }
    }
    else if (Role == 3)
    {
        if (!Cit->TargetStone) // if no assigned stone
        {
            Cit->TargetStone = Cit->MyTownHall->GetClosestStone();
        }
        else if (Cit->TargetStone->Dead) // if stone has died
        {
            Cit->TargetStone = Cit->MyTownHall->GetClosestStone();
        }
        else
        {
            Stage1 = GoToLocation(Cit->TargetStone->GetActorLocation());
        }
    }
    else
    {
    // #Future work# get dropped resources
    }
}
```

Figure 21. Stage 1 of work state .cpp code

Stage 2 as shown in figure 22, proceeds to start gathering from the resource with checks making sure that it is carrying the correct resource type and whether they can carry any more of it. Once the citizen can't carry any more then will then proceed to move onto stage 3.

```
void AFiniteStateMachineAI::Stage2Work() // Perform the task at location
{

    switch (Role)
    {
    case 1: // Farm ////////////////////////////////////////
        if (Cit->ResourceType != 1)
        {
            if (Cit->NumResources > 0)
            {
                Cit->DropResources();
            }
            Cit->ResourceType = 1;
        }

        Cit->AssignedFarm->Worker = true;

        if (Cit->NumResources >= Cit->CarryWeight)
        {
            Stage2 = true;
            Cit->AssignedFarm->Worker = false;
        }
        else
        {
            if (Stage1)
            {
                // work farm
                Cit->DoTask(1);
            }
        }

        break;
    case 2: // Lumberjack ////////////////////////////////
        if (Cit->ResourceType != 2)
        {
            if (Cit->NumResources > 0)
            {
                Cit->DropResources();
            }
            Cit->ResourceType = 2;
        }
        if (Cit->NumResources >= Cit->CarryWeight)
        {
            Stage2 = true;
        }
        else
        {
            if (Stage1)
            {
                // take wood
                Cit->DoTask(2);
            }
        }
        break;
    case 3: // Miner ////////////////////////////////////////
        if (Cit->ResourceType != 3)
        {
            if (Cit->NumResources > 0)
            {
                Cit->DropResources();
            }
            Cit->ResourceType = 3;
        }
        if (Cit->NumResources >= Cit->CarryWeight)
        {
            Stage2 = true;
        }
        else
        {
            if (Stage1)
            {
                // take stone
                Cit->DoTask(3);
            }
        }
```

Figure 22. Stage 2 of Work state .cpp code

For Stage 3 shown in Figure 23, as it is the same goal for each of the roles it grabs an available storehouse that has space for the resources and sets the stage to false until the citizen is at the storehouse from where it moves onto stage 4.

```
void AFiniteStateMachineAI::Stage3Work() // go to storehouse
{
    Cit->TargetStoreHouse = Cit->MyTownHall->GetAvailableStoreHouse(Role);
    Stage3 = GoToLocation(Cit->TargetStoreHouse->GetActorLocation());
}
```

Figure 23. Stage 3 of Work state .cpp code

Lastly for stage 4 shown in figure 24 is to put the resource into the storehouse, once the citizens resources hits 0 then it will reset the work state.

```
void AFiniteStateMachineAI::Stage4Work() // drop off resources
{
    if (Stage3)
    {
        Cit->DoTask(4);
    }
    if (Cit->NumResources <= 0)
    {
        Cit->NumResources = 0;
        Stage4 = true;
    }
}
```

Figure 24. Stage 4 of Work state .cpp code

# C - Behaviour Tree:

## Town base:

The Town Hall is to act as the main part of the civilization, it will be storing all relative information about the world such as where all of the resources are and will be spawning all of the buildings and citizens in both while running and for initial spawning. The town hall has functions as well to sort and distribute the information it has stored such as able to find the closest resources (Figure 25), which storehouse has available space and assigning people to spare houses and farms. The Town Hall also has the the function to build any of the objects in the civilization (except trees and stones) (Figure 26).

```
AStone* ATownHall::GetClosestStone()
{
    AStone* ClosestStone = nullptr;
    float DistanceToStone = 0;
    if (FoundStone.Num() > 0)
    {
        for (const auto& Stones : FoundStone)
        {
            if (!Stones->Dead)
            {
                float Distance = (this->GetActorLocation() - Stones->GetActorLocation()).Size();
                if (DistanceToStone == 0 || DistanceToStone > Distance)
                {
                    DistanceToStone = Distance;
                    ClosestStone = Stones;
                }
            }
            else
            {
                FoundStone.Remove(Stones);
            }
        }
    }
    else
    {
        ScanForResources();
    }

    return ClosestStone;
}
```

Figure 25. Get Stone Function

Due to the sorting functions for the resources and the buildings all being of a similar structure I will use Figure 25 as an example of the functionality of the sorting functions. In the Unreal Engine conventional arrays don't work and will crash the engine when used, so implemented into the engine was "TArray" which allow efficient one dimensional sorting but the values inside it can't be acquired easily as it requires a for loop shown in Figure 25 going through each individual value.

```cpp
void ATownHall::BuildBuilding(int Choice, int Radius, bool InitialSpawn)
{
    FActorSpawnParameters SpawnParameters;
    SpawnParameters.bNoCollisionFail = false;
    FTransform SpawnTransform = GetTransform();
    if (Radius < 2500) // Testing mainly, currently could be problems at less than this value
    {
        Radius = 2500;
    }
    SpawnTransform.SetLocation(GetRandomMeshPoint(Radius)); // Gets a point nearby that can be navigated to on the navmesh

    if (GetWorld())
    {
        if (Choice == 0) // StoreHouse
        {
            AStoreHouse* SpawnStoreHouse = GetWorld()->SpawnActor<AStoreHouse>(StoreHouse, SpawnTransform, SpawnParameters);
            if (InitialSpawn) // Outside of first spawning these will be empty
            {
                SpawnStoreHouse->HousedFood = 1000;
                SpawnStoreHouse->HousedWood = 1000;
            }
            AllStoreHouses.Add(SpawnStoreHouse);
        }
        else if (Choice == 1) // House
        {
            AHouse* SpawnHouse = GetWorld()->SpawnActor<AHouse>(House, SpawnTransform, SpawnParameters);
            AllHouses.Add(SpawnHouse);
            if (InitialSpawn) // Outside of first spawning these will be empty
            {
                SpawnHouse->HousedFood = 1000;
            }
        }
        else if (Choice == 2) // Farm
        {
            AFarm* SpawnFarm = GetWorld()->SpawnActor<AFarm>(Farm, SpawnTransform, SpawnParameters);
            OwnedFarms.Add(SpawnFarm);
        }
        else if (Choice == 3) // Citizen
        {
            ACitizenBody* SpawnCitizen = GetWorld()->SpawnActor<ACitizenBody>(Citizen, SpawnTransform, SpawnParameters);
            SpawnCitizen->MyTownHall = this;
            SpawnCitizen->TownNum = TownNum;
            if (InitialSpawn) // If sorting works then their will be no need for this check
            {
                SpawnCitizen->Role = FMath::FRandRange(1, 3); // Temp for testing
            }
            AllCitizens.Add(SpawnCitizen);
            TotalCitizens++;
        }
    }
}
```

Figure 26. Build Building Function

Referring to the example described previously in Figure 17 of this report, it allows you to spawn objects into the world relatively easily as shown now in Figure 26. After assigning each blueprint to the corresponding variable in the blueprint child of this class it allows you to spawn the object in while retaining a pointer to that object of which is then stored in an allocated array for later use. The blueprints linking them together are shown in Figure 27.


Figure 27. Blueprint Spawning Link

## AI (Behaviour Tree Part):

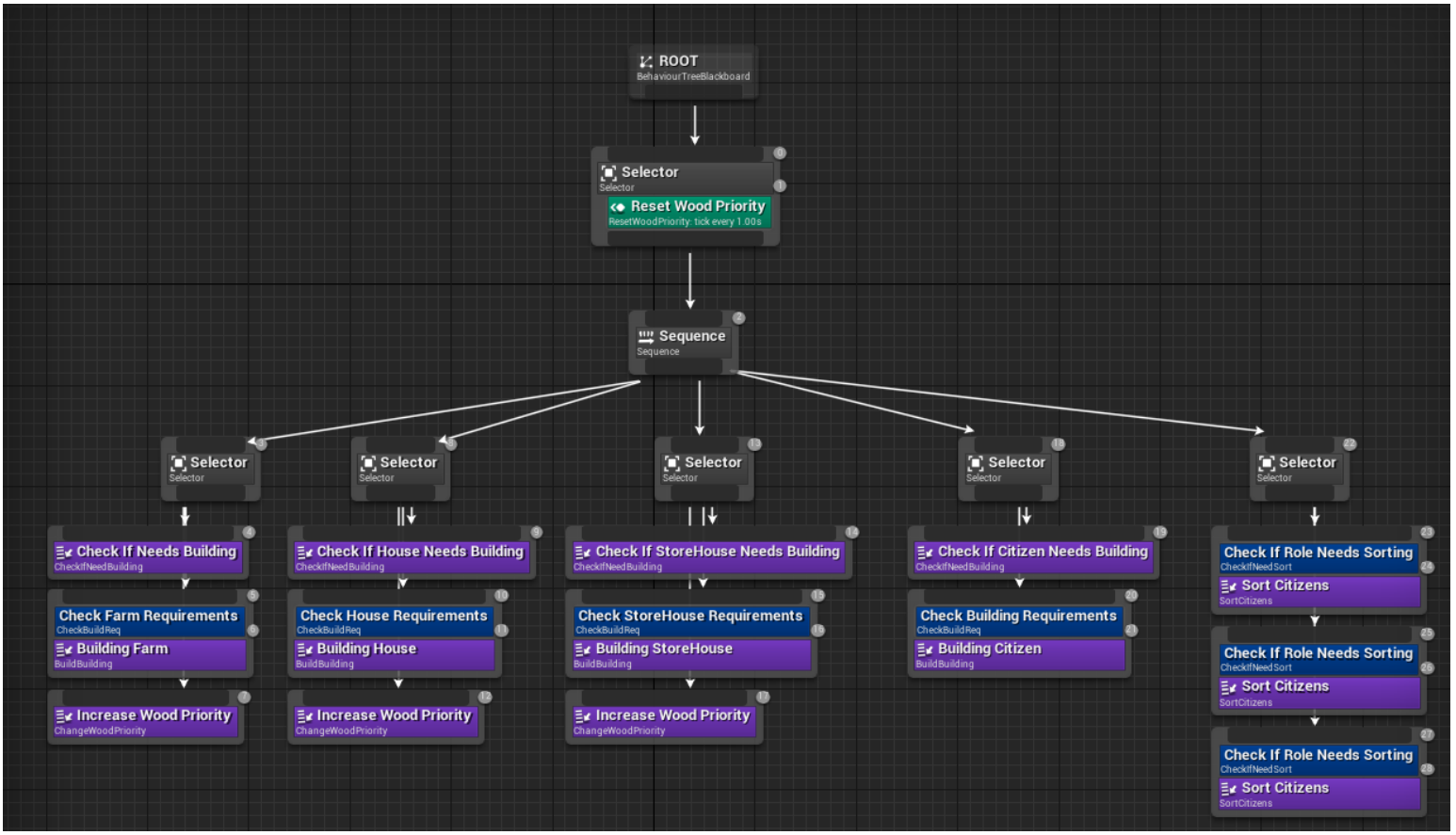



Figure 28. Unreal Engine Behaviour Tree

Due to time constraints the nodes in the behaviour tree are mostly untested and incomplete but the overall structure of the tree was completed and is functional as seen in Figure 28. The tree successfully builds the necessary building that is required however the sorting mechanisms and the wood priority value is mostly unfinished.
All the nodes in Figure 28 except for the selector, sequence and root nodes are created in C++. To create each of the different nodes each have to be derived from a different source, the task nodes (purple) are derived from a class called BTTaskNode and must have BTTask in front of it to be identified as a task node by the behaviour tree blueprints. This is similar for the service nodes which are derived from BTService with BTService at the start of its name and decorators which are derived from BTDecorator with BTDecorator at the start of its name. Deriving from any other class has a higher chance of causing errors within the behaviour tree and not even running.
Each of the nodes must return different parameters as well whereas services are void functions so no parameter is returned, decorators are bool functions which require a true or false answer returned with true allowing the tree to proceed and false making the node automatically fail, and tasks which are EBTNodeResult functions that need a value of this type returning, such values are Succeeded, Failed, Aborted (which is the same as failed) and InProgress which allows the node to be returned to.

As their was no requirement to create new selector nodes there was no attempt to create new nodes similar to the simple parallel node but on inspection it could be possible to create as all 3 nodes derive from the same class so with more work and research the behaviour tree could have distinctly more possibilities than what's already available to blueprints.

# 8 - Testing:

## A - Overall results:

Due to the method of constructing this project the citizen and the finite state machine was constructed fully and operates mostly without error, there are still a few bugs here and there but they are difficult to find due to some complications detailed in the next section. The town hall and its functions to which I completed also work as well mostly without error, like before possibly an error here and there but their but once again it's not that easy to find.
For functionality the engine is currently able run the finite state machines up to a high amount, it is able to support a wide amount of citizens at the same time but however exact numbers are not available.
Currently due to the resource management I want able to implement the resources being taken away so once the behaviour tree reaches a requirement on building a building it will constantly build it endlessly and as citizens are included in that spawning procedure it is evident that the sorting section of the tree currently doesn't work which is disappointing.
Lastly due to the way the behaviour trees are set up currently only one AI can make use of it leaving other towns to simple do nothing when spawned.

## B - Difficulties and Complications:

Throughout this project I have experienced many problems with its development. At the start of this project I was taken ill for approximately 1 week which was vital time lost due to my recovery. Shortly after my illness I found myself with technical difficulties found in the university labs as i found in the room was working in did not have admin privileges on the computers nor were they kept updated and due to this those computers possessed a corrupted versions of the Unreal Engine and the project would crash and corrupt constantly until a new room was found with the appropriate computers to run the project on. The total time lost to those corrupted computers was also approximately a week long meaning out of the whole schedule a total of 2 weeks was lost including the illness.
Looking forward to debugging on these new computer is something I never noticed until a couple days ago which was that the Unreal Editor crash report system had been disabled for some unknown reason, I had originally thought that due to the nature of the crash there was nothing for the crash report to show but evidently testing it on my laptop (which was not suitable to build the project on due to it not being very fast) an error report appeared showing where the error originated which allowed me to fix them instantly, however due to the small timespan there was to finding them all due to realising this last week many still remain.
Lastly the Unreal Engine's documentation is significantly lacking in details especially when it comes to the more advanced works such as creating your own behaviour tree nodes of which I struggled to find any at all, my last resort was to find the Unreal Engine Project found

on Github (Epic Games 2017) and look at the source to see how the current preset ones were created.

# 9 - Evaluation:

Due to the time constraints I wasn't successful in completing the project. Although the finite state machine is mostly complete awaiting features adding to the town hall, the behaviour tree is far from complete. The original structure of the tree is their however due to it being incomplete and untested it will just spawn thing quite randomly and not in anyway an organised fashion.
Many of the finite state machines can be spawned and operate at the same time however currently only one behaviour tree can be running at a time which means one civilisation an be operating at a time.

# 10 - Conclusion:

Overall it is definitely possible to complete this project to the standards I originally set out to achieve but more time must be allocated to working on this project to insure it does get completed to a good standard, in terms of time even if I had not encountered the errors and problems I had faced I would have been stretched for time anyway.
For the Unreal Engine it is a significant learning curve to master due to the lack of documentation but overall now I know how to create a lot more within the engine in C++ now.
Due to its layout I understand now that although it is trying to sway towards allowing people to create games only in blueprint, using blueprints is the way you get around using the engine. As a C++ developer knowing how you can integrate your code with the blueprint system not only allows you to work better with teams using unreal but simplies the process of building your games, especially after i decided to leave all the modeling to the blueprint system and attach my objects together using it, I know now that it makes you life far easier working with it instead of constantly linking everything together in the instantiation function of your class.
Doing this project has now gave me a greater understanding on how to build and maintain both finite state machines and behaviour trees which was one of my personal goals as i've never performed well at developing AI and seeing that i've successfully created two AI (well at least 1) is a great inspiration for me to create more of them in my own time and in time I may come back and complete this project as a good example to myself on how it's done.

# 11 - References:

Finite State Machine Example <https://commons.wikimedia.org/wiki/File:Finite_state_machine_example_with_comments.svg>

BT search and grasp (Aliekor, 2007)
<https://commons.wikimedia.org/wiki/File:BT_search_and_grasp.png>
Epic Games, 2017 <https://github.com/EpicGames>

# 12 - Bibliography:


Waltham, M, 2016, An Analysis of Artificial Intelligence Techniques in Multiplayer Online Battle Arena Game Environments <http://dl.acm.org/citation.cfm?id=2987513>

Othman, M, 2014, Implementing game artificial intelligence to decision making of agents in emergency egress <http://ieeexplore.ieee.org/document/6986036/>

Rabelo, R, 2013, Automatic code generation of SIMUROSOT game strategies: an approach based on finite state machines <http://dl.acm.org/citation.cfm?id=2439983>

Elogeel, A, 2015, Selecting Robust Strategies in RTS Games via Concurrent Plan Augmentation <http://dl.acm.org/citation.cfm?id=2772902>

Epic Games, 2017 <https://forums.unrealengine.com/>

Wikipeadia, Decision tree learning <https://en.wikipedia.org/wiki/Decision_tree_learning>

Wikipeadia, Behavior tree
<https://en.wikipedia.org/wiki/Behavior_tree_(artificial_intelligence,_robotics_and_control)>