# FIT3094 - Assignment 2 - Documentation

Pavle Spasic 29691931

## Outline of the Algorithms

The basis of the assignment code is based on the Goal Oriented Action Planning system from the Week06 Lab. Modifications to the GOAPActor class have been made to include variables such as health and a static mesh component.
The Week06 Lab content has actors use a state machine to separate what task to do when, and a GOAPPlanner to grab the list of actions available to reach a goal state.

Woodcutter, Stonemason and Builder have very similar functionality, all three gather resources in some way, deposit them in some place, and search for food in some way.

All three of these actors use the same action EatFoodAction to find food and heal them. The Precondition check for this is similar to what is seen in the previous labs, gets FoodActor within a certain range, and saves the closest one. When performing this action, it gets the target's health and sets it to its maximum health.
The actors are losing health through a timer manager within the parent GOAPActor class.

```
GetWorldTimerManager().SetTimer(TimerHandle, this, &AGOAPActor::DecreaseHealth, 2.0f, true, 2.0f);
```

Each class also checks each tick if they are below their health range to look for food. It calls a super function which clears the current actions and enters the idle state which will find a new set of actions to complete, having find food first.

Similar to EatFoodAction, the Woodcutter's GatherWoodAction and Stonemason's GatherStoneAction, both have the same functionality to find one in the world. The resource is gathered until the agent can no longer carry no more.

Woodcutter and Stonemason both use DepositResourceAction to drop off resources. The action uses a similar OverlapActor but only saves one Village centre, it does cast to the different agents to determine which resource to be deposited.

The builder uses separate functions for getting resources and depositing resources and the locations of which are different to the other agents, and their amounts and rate of transfer is different. Finding the buildings is the same as other actions, but the WithdrawResourceAction will make sure there are resources in the Village Centre. It also grabs the minimum between how much the builder can carry and what's in stock.

Collision avoidance is completed by using a USphereComponent which is set up to be in front of the the GOAPActor, when another GOAPActor begins overlap with the sphere, the actor will go into its State_Stall, where it will wait until that actor moves out of the area. Each tick it will also

check if any other actors that are within its sphere are also stalling. If they are, the one with the longer distance will get right of way and continue with its journey.

## Actions (Preconditions and Effects)

```
EatFoodAction* EatAction = new EatFoodAction();
EatAction->AddPrecondition("NeedFood", true);
EatAction->AddEffect("NeedFood", false);

GatherStoneAction* GatherAction = new GatherStoneAction();
GatherAction->AddPrecondition("HasRoom", true);
GatherAction->AddEffect("HasRoom", false);
GatherAction->AddEffect("HasStone", true);

DepositResourceAction* DepositAction = new DepositResourceAction();
DepositAction->AddPrecondition("DepositeStone", false);
DepositAction->AddPrecondition("HasStone", true);
DepositAction->AddEffect("DepositeStone", true);
```

## States

```
TMap<FString, bool> AStonemason::GetWorldState()
{
    TMap<FString, bool> WorldState = Super::GetWorldState();

    WorldState.Add("NeedFood", Health < 20); // false
    WorldState.Add("HasRoom", NumStone < maxStone); // true
    WorldState.Add("HasWood", NumStone > 0); //false
    WorldState.Add("DepositeStone", false); // false

    return WorldState;
}

TMap<FString, bool> AStonemason::CreateGoalState()
{
    TMap<FString, bool> GoalState;

    GoalState.Add("NeedFood", false);
    GoalState.Add("DepositeStone", true);

    return GoalState;
}
```

# Reasoning

Due to time constraints, using the existing code from the week 06 labs was the most quick and efficient start, It allowed me to focus on creating custom classes for assignment and not focus on the core functionality.

Having one function for all agents/actors use the same EatFoodAction  keeps the number of classes down and the whole system cleaner.
A Timer manager is used to keep the health management even more simpler, using this calls a function every so often instead of filling the Tick function with more code to compute.
The Gather actions were kept separate, although being similar in functionality, the different classes would make a messy Action class that would have required different inputs.

Having the agents enter the idle state when they drop below their cutoff health stops them from dying during an action or set of actions. Entering the IDLE_STATE will mean the actor will act like normal and look for new actions and it first will find food. Switching states allows me to use existing code to the best advantage.

The reason for a singular resource deposit action, is that for future agents, there may be a potential to be able to deposit Wood, Stone and/or Metal. So having one function means it can all be done at once. The current scope does not require this, but this can keep the solution simpler.
The action also only saves one VillageCentre as it is assumed only 1 will be on the map, so a TArray was not needed.

Woodcutter, Stonemason and Builder all have the same action preconditions, effects and states. Although renamed to fit class. This is because all 3 have the same generic loop they follow. Get Resource → Deposit Resource (and heal if needed). So the same states and preconditions/effects keeps it simple.

WithdrawResourceAction ensures the village centre has resources before being doable to ensure that the agent isn't walking to the VillageCentre to pick up nothing. The minimum is chosen so that the num resources do not go into the negative, aka the builder grabbing non existent resources.

# Problems Encountered

The first initial problems during development were the agent not going for food when below the certain health pool, and not moving at all. These were caused by personal typos from the code from lab tasks and a missing SetActorLocation. More specifically the GoalPlanner::CheckConditionsInState function was mistyped and only returned false so no action could pass it.

I encountered a problem with EatFoodAction, with the way I had it initially, the agent would go to the same spot in the world even though the food had been destroyed. This was fixed by ensuring the food was removed from the food list before it was destroyed.

When Adding multiple actions with connecting preconditions and effects, the initial implementation failed to find a plan, it couldn't meet the goal state. The problem was caused by there being a mismatch in the strings of the MapState, GoalState, preconditions and effects. A simpler approach with fewer goal states also contributed to the fix, the original method had 4 goal states that wanted to be a certain value, but having *only 1* made it simpler to create a route for the agents to find an action path.

An older version of the DepositBuilderAction has used the Kismet library function GetActorOfAllClass and GetAllActorsOfClass, this made initial errors where a target would not be found, so the approach done within other actions was taken.

One problem that occurred was that actions that were referencing a VillageCentre were behaving weirdly, and a variable was never being set, but passed checks to see if it wasn't a null pointer. This caused unhandled null pointer exception crashe's down the line. This had to be fixed by implementing a list to store VillageCentres instead of just one reference.

# Existing Bugs

GOAPActors May get stuck on each other.