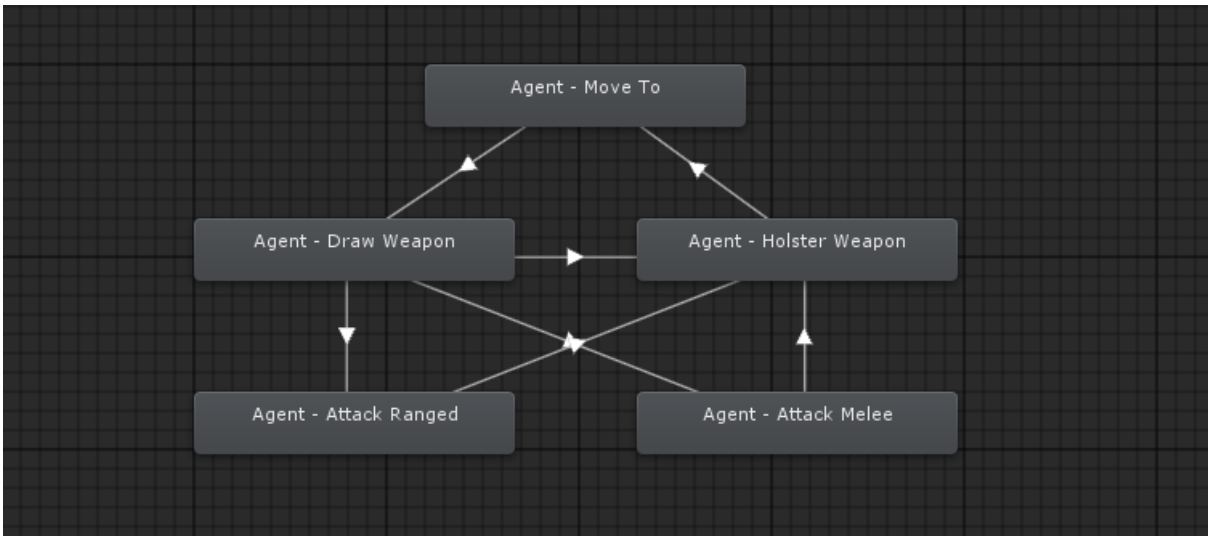
Goal Oriented Action Planning

Dynamic | Artificial Intelligence | Reusable

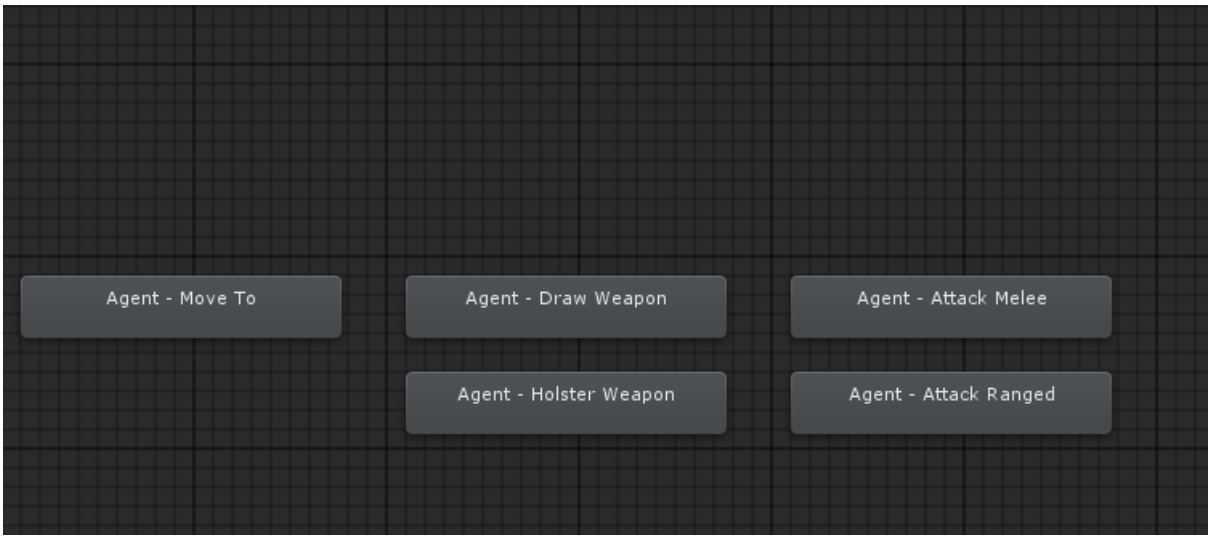
[How does Goal Oriented Action Planning work?](#)
[How to implement IGoap](#)
[How to create your own actions & goals](#)
[How does the planner work in practice](#)

How does Goal Oriented Action Planning work?
Goal Oriented Action Planning is a different way to handle Artificial Intelligence within video games. Although used by games like Fallout 3 and F.E.A.R., it's still relatively unknown.

Goal Oriented Action Planning, from here on referenced as Goap, gets rid of the 'classic' statemachine used by developers. Instead we supply Goap with separate states, referenced to as 'Actions', and guidelines, referenced to as 'Goals'. Goap in turn looks at the most important goal, and will string actions together in order to achieve that goal. Everything is done automatically, so there's no need to update a statemachine or code transitions to fit the actions together.



A traditional statemachine. All transitions have to be done by hand.



GOAP instead separates the states

So how does Goap string actions together to achieve a goal? Instead of having to tell which actions go together yourself, actions and goals use conditions. A conditions consists of 2 parts; the identifier, a string variable, and a value, which is a unity object.

Example: The condition [“applesInInventory”, 10] holds data, while the condition [“isHungry”, true] does the same, but in a slightly different way.

```
Condition applesInInventory = new Condition("applesInInventory", 10);
Condition hungry = new Condition("isHungry", false);
```

Actions have 2 lists of conditions, referenced to as preconditions and effects. The preconditions have to be met before the action is considered, while the effects are what the action does after it is finished.

```
public List<Condition> preconditions = new List<Condition>();
public List<Condition> effects = new List<Condition>();
```

Goals have 1 list of conditions, referenced to as succes. If every condition in the succes list is met, the goal is viable and a string of actions is found.

```
public List<Condition> succes = new List<Condition>();
```

Example: We have the goal ‘SatisfyHunger’. SatisfyHunger’s succes factor is the condition [“isHungry”, false]. Thus we know the goal is successful if the condition [“isHungry”, false] is met. We add this condition to a condition list referenced as ‘currentState’. This is so we can easily keep track of what the current condition state is.

Goals	**SatisfyHunger**
Succes	[“isHungry”, false]

Actions	**GatherFood**	**EatFood**
Preconditions	-	[“hasFood”, true]
Effects	[“hasFood”, true]	[“isHungry”, false]

We have 2 Actions to try and empty the currentState with. The effect of the Action ‘EatFood’ is the condition [“isHungry”, false]. Since the effect matches one or more conditions in the Current State, we know its a action that will empty the list. Thus the condition [“isHungry”, false] is removed from the current state. Since the action ‘EatFood’ does have a precondition, the condition [“hasFood”, true] is added to the current state.

With the updated Current State, we can see that ‘GatherFood’ has become a purposeful action, since its effects matches one or more conditions in the Current State. The condition [“hasFood”, true] is removed from the current state. Since ‘GatherFood’ does not have any preconditions, the Current State is left empty. This means a successful plan for the goal has been found, namely ‘GatherFood’ followed by ‘EatFood’.

CurrentState	**Conditions**
	[“isHungry”, false]

The currentState conditions are filled with the goal’s succes list.

CurrentState	**Conditions**
	[“isHungry”, false]
	[“hasFood”, true]

We remove the effects of the action ‘EatFood’, while adding the preconditions of ‘EatFood’.

CurrentState	**Conditions**
	[“hasFood”, true]

We remove the effects of the Action ‘GatherFood’. Since no new preconditions are added, the currentState is empty and a plan has been found.

Reusability is a big part of Goap. In this example we have a Orc and a Knight, both with the action ‘AttackMelee’ and the goal ‘EliminateTarget’. Since the actual target is determined in the Orc/Knight class, both these classes can use the same actions and goals with different targets/results.

	Orc	**Knight**
Actions	‘MoveTowards’ ‘AttackMelee’	‘MoveTowards’ ‘AttackMelee’
Goals	‘EliminateTarget’	‘EliminateTarget’

Goap also offers dynamic behaviour by keeping the same goals, but giving entities different actions. In this example both the Orc and Knight have the goal ‘EnterRoom’. The Orc uses the action ‘RunTowards’ to get in range, and then uses the action ‘SmashDoor’ to get inside the room. The Knight however uses the action “MoveTowards” to get in range, and then uses the action ‘OpenDoor’ to get inside.

	Orc	**Knight**
Actions	‘RunTowards’ ‘SmashDoor’	‘MoveTowards’ ‘OpenDoor’
Goals	‘EnterRoom’	‘EnterRoom’

Dynamic behaviour is also accomplished by the fact that actions have a cost, while goals have a priority. Goals with a higher priority are preferred, but if the goal isn’t viable a lower priority goal will be selected. Actions use a cost. The lower the cost, the better. The cost and priority can be changed on runtime however, allowing dynamic behaviour best suited for the situation.

Example: We have a warrior who has the action ‘MoveTowards’, ‘AttackMelee’, and ‘AttackRanged’. The ‘AttackMelee’ forces the Warrior to use the MoveTowards action, creating a higher total cost. The MoveTowards cost however, is dependant on the range between the Warrior and its target. The further away, the higher the cost of MoveTowards will be. Thus if the Warrior has to walk too far, it will prefer the ‘AttackRanged’ action. If the target is close however, the Warrior will combine ‘MoveTowards’ with ‘AttackMelee’.

Actions	**Preconditions**	**Effects**	**Cost**
MoveTowards	-	[“inRange”, true]	2 - 10
AttackMelee	[“inRange”, true]	[“eliminateTarget”, true]	5
AttackRanged	-	[“eliminateTarget”, true]	10

How to implement IGoap

The first step to implementing the Goap Framework is by creating a new class and let it use the namespace goap. Implement the goap interface ‘IGoap’. Implement the 3 properties IGoap requires, namely:

- State. A list of conditions this class has. These conditions are used by the planner
- availableGoals. A list of Goals this class will try to satisfy
- availableActions. A list of actions this class has, to try and satisfy the goals with

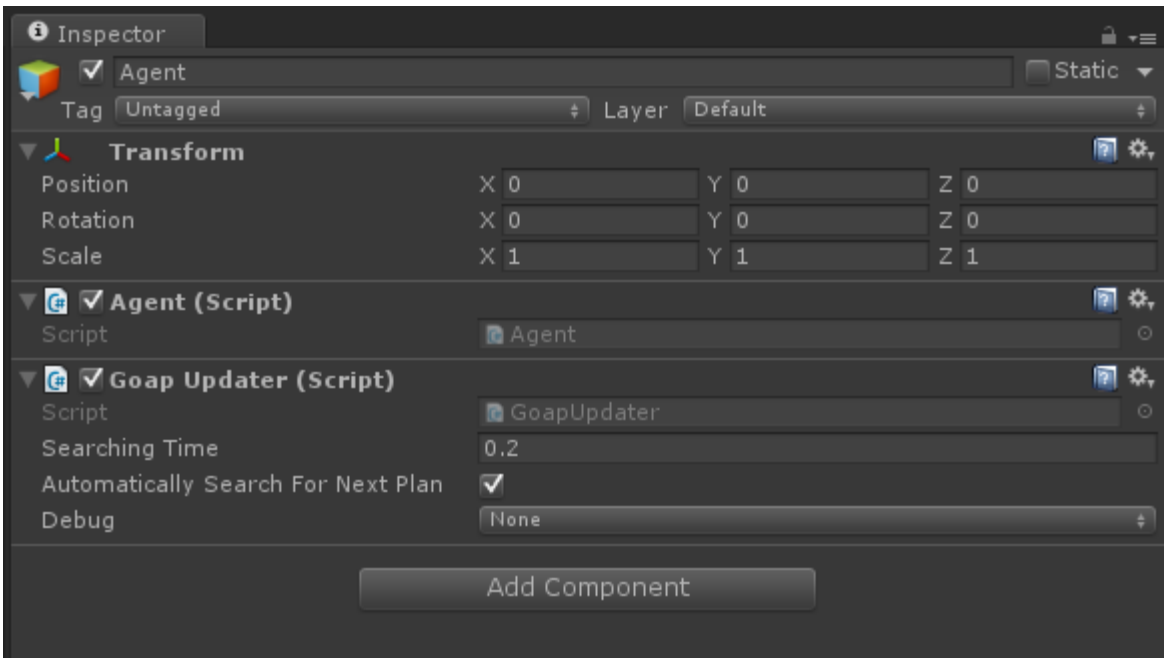
```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using GOAP;

public class Entity : MonoBehaviour, IGoap {

    // These variables are essential for the GoapUpdater
    public List<Condition> state { get; set; }
    public List<Goal> availableGoals { get; set; }
    public List<Action> availableActions { get; set; }

}
```

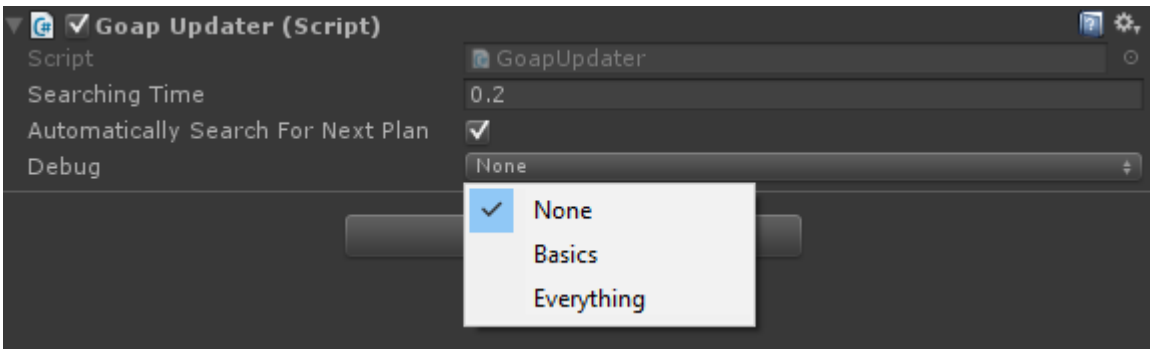
After implementation is done, add this component to a GameObject, and add the component ‘GoapUpdater’ on the same GameObject.



As soon as the game starts the GoapUpdater will look for the IGoap component. If the IGoap component is found, the GoapUpdater will then try to find a plan by using the Planner class, and sending it the IGoap state, goals and actions. If a plan is found, and thus a goal can be satisfied, the GoapUpdater will perform the first action of the plan. As soon as the action is finished performing, the GoapUpdater will remove this action and take the next action. This process is repeated until the plan is empty, or an action is aborted. If that's the case, the GoapUpdater will automatically look to satisfy another goal, or try to find a new way to satisfy the current goal.

The GoapUpdater has several variables which can be adjusted.

- searchingTime. Time in seconds it takes before the planner searches for a new plan, as soon as the old plan is aborted or finished. The higher the searchingTime, the lower the performance cost.
- automaticallySearchForNextPlan. Can be set to true or false. If false, the planner won't update itself if the plan is aborted or finished.
- Planner.DebugPlanning. Used by the planner to debug the planning process. Choose between None, Basic and Everything.
- The GoapUpdater also has 6 System.Actions, which are called when a action or goal is started, finished or aborted.



The GoapUpdater also has 2 public functions.

- PushGoal (Goal goal). Aborts the current goal and tries to find a plan for the goal received as a parameter. The given goal does not need to be in the IGoap goals list
- AbortGoal. Aborts the current goal and actions.

How to create your own actions & goals

Action

Creating a new action is really simple. Create a new C# script, use the namespace GOAP and inherit from the class action.

The action has several functions which can be used

- OnActionSetup (IGoap iGoap, List<Condition> state). Called by the planner when the action is being considered. This function is used to receive the iGoap and current state variables.
- OnActionStart. Called by the GoapUpdater, this function is called when this action is in the current plan and started.
- OnActionPerform. Called by the GoapUpdater, this function will be called until the action is either finished, or aborted
- OnActionFinished. Called by the GoapUpdater, this function will be called when this action is finished.
- OnActionAborted. Called by the GoapUpdater, this function will be called when this action is aborted.
- Ofcourse you can also use the constructor to set variables, or add/update/remove preconditions and effects. To change condition states, simply use the functions Add/Update/Remove Preconditions or Effects.

```
using UnityEngine;
using GOAP;
using System.Collections;

public class NewAction : Action {

    public NewAction() {

    }

    public override void OnActionSetup(IGoap igoap, System.Collections.Generic.List<Condition> state) {

    }

    public override void OnActionStart() {

    }

    public override void OnActionPerform() {

    }

    public override void OnActionFinished() {

    }

    public override void OnActionAborted() {

    }
}
```

Besides these functions, the action class has several important variables. It has 2 lists filled with conditions, namely the preconditions and the effects. It also has a isViable, isFinished and isAborte' boolean. The isViable variable is used by the planner to see if the action is viable. Since this variable is inheritely false, make sure to set it to true in the constructor, of if you want to do a runtime check set it to true in the OnActionSetup.

isFinished and isAborted are used by the GoapUpdater. If these variables are set to true, the updater will act accordingly and move on to the next action or abort the action respectively

Actions also have a cost. A lower cost means the planner favors this action.
Actions also have currentPerformTime and a totalPerformTime. These are not essential, but can be used to easily add a timer to an action. Just remember to add to the currentPerformTime in the OnActionPerform, and check to see if progress is set to 1.

```
public override void OnActionPerform() {
    currentPerformTime += Time.deltaTime;
    if (progress == 1f) isFinished = true;
}
```

Note: All action functions are override, so you can decide which functions to implement.

Goal
Creating a new Goal is almost like creating a new action. Create a new C# script, use the namespace Goap and let it inherit from the goal class.

The action has several functions which can be called

- OnGoalInitialize. Should be called from the iGoap class. This initializes the goal class. The reason a goal has a initialize function, and the actions don't is because action are copied by the planner, while a goal remains the same throughout.
- OnGoalSetup. Called from the planner. Useful to change values on runtime
- OnGoalFinished. Called by the goapUpdater, this will be called when the plan was succesful and the goal is finished.
- OnGoalAborted. Called from the goapUpdater, this will be called when one of the actions, and thus the plan, was aborted.
- IsGoalRelevant. Called from the planner, the returned value is used to see if the goal can be used by the planner. This is called after OnGoalSetup. This function is marked abstract, and thus is required to always be implemented.

Besides these functions, the goal class has several important variables. Namely a list of conditions for the succes factor and a priority variable used by the planner. The higher the priority, the more the planner will favor this goal.

How does the planner work in practice

The planner is called from the GoapUpdater component, and requires 3 important parameters.

- The iGoap variable. This will be used to send to actions and goals
- A list of actions. These actions will be used to try and satisfy a goal
- A list of goals. A relevant goal with the highest priority will be selected automatically.

The planner starts off by sorting the goals based on the highest priority. The goal with the highest priority will be used, and OnGoalSetup will be called. Next up the planner will call IsGoalRelevant to see if the goal can be used. If the goal isn't relevant, the planner will discard this goal and use the next one.

The planner will create a new list of conditions, called 'current state', and will fill it with all the Goal succes conditions.

Note: For performance reasons the planner works backwards, starting at the goal state and working towards an empty state. As soon as a plan is found the actions are inversed to make sure the GoapUpdater starts with the correct action.

Next up, the planner will loop through all available actions and make a copy of that action. Foreach action, the planner will;

- Call OnActionSetup, sending it the current iGoap and current state
- Check to see if the action's effects conditions match one or more of the current state conditions.
- If there's a match, the planner will check to see if the action is viable, by checking the action's isViable variable.
- If the action is viable, the planner will take the current state and add all the action's preconditions.
- Next the planner will take the action's effects, and remove all matching conditions from the current state.
- Finally the planner will take the iGoap state, and remove all matching conditions from the current state.

After this step, the planner does a check to make sure the current state has changed, so no useless actions are added.

If the action is viable and the current state has changed, the planner will create a new node, adding the correct parent node, action and current state.

The next node will be picked from the open list, based on the node with the lowest cost.

This progress is repeated until the open list of nodes is empty, or the current state is empty. If the current state is empty, it means a plan has been found and it will be returned. If the open list is empty, it means no plan could be found. In this case the current goal will be discarded and the goal with the next highest priority will be picked.

In many ways the planner functions similarly to an A* pathfinding planner.