Table of Contents

Articles
- Introduction
- Considerations
- Actions
- Utility AI component
- WF Behavior assets
- Networking and Multiplayer
- Integrating with UE features
- How to debug
- The Sample
- Design resources

Welcome to Wise Feline

Wise Feline is a utility AI system. Wise Feline allows you to make an AI which feels unscripted and immersive and allows you to do this in a fraction of the time it takes to do AI in a scripted and sequencial manner. Wise Feline uses Utility theory and by extension the Utility AI approach championed by Dave Mark in the games industry which is used in games as big as the sims and guild wars 2.

Philosophy

An AI system usually needs to perform two functions, listing the actions which can be taken and choosing an action from the list to execute. Utility AI does this by giving all possible actions a score between 0 and 1 and then chooses the action with the highest score.

Each action has a set of considerations which estimate the action's utility at the moment. Each consideration has a score between 0 and 1 based on the current parameters supplied to it. Each action's score is calculated by calculating the score of all of its considerations and then multiplying them by each other.

This is how humans take decisions. For example if you have actions below:

- Buying food
- Eating food
- having a rest

Then the eating food action probably would have two considerations for calculating if the agent is hungry or not and also calculating if the agent has food or not. The consideration for being hungry is a continuous value between 0 and 1 but the having food available consideration is a binary which is either 0 or 1. The score of the action is calculated by multiplying these two so when the agent doesn't have food, the score for eating food will always be 0 since you don't have any food to eat.

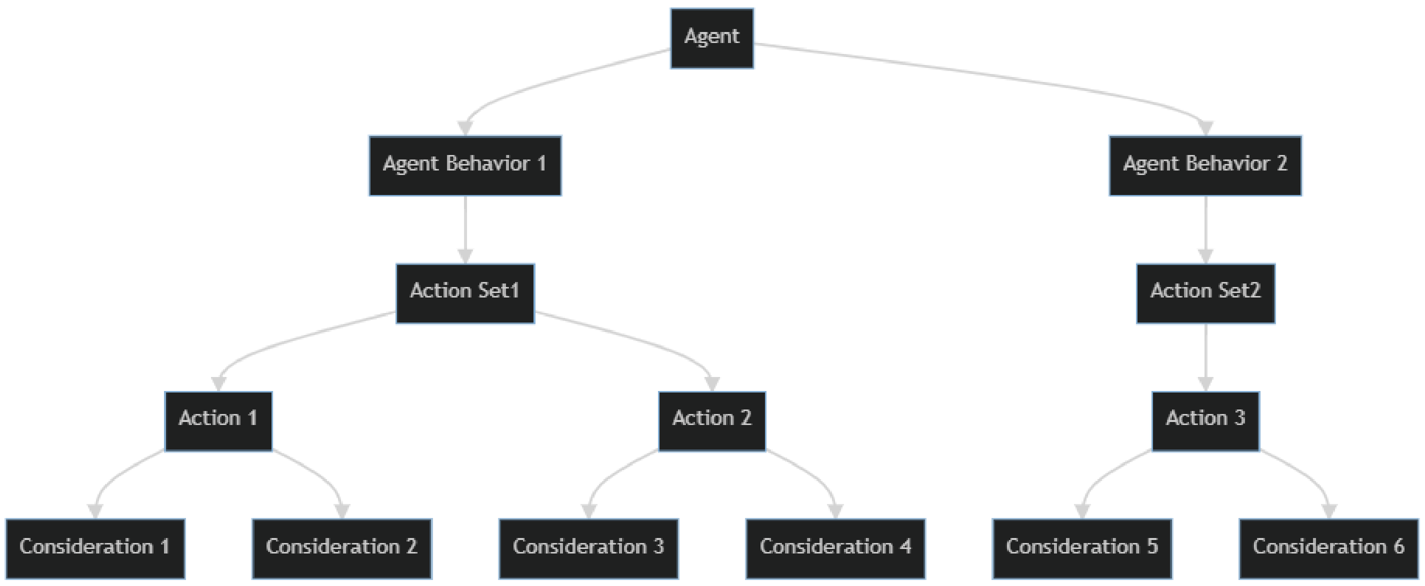
Buying food on the other hand has considerations for not being tired and not having food and being hungry and these should return their values in a way that if you are very hungry, the score becomes high even if you are tired. Resting action probably should only give a high score if you are tired but not that hungry.

Each consideration is a function of a curve which its x axis is the value that we read from the agent/environment and its y axis is the output score of the consideration. So for hunger we read the hunger value of the agent and return it and then its value is given to a curve as x axis and the resulting y axis is the actual score of the consideration. In this way we can make a consideration more sensitive to certan values so for example having a hunger of 0.5 can return 0.5 in the case of a linear curve and 0.9 or 0.1 in the case of custom curves.

These curves for considerations are the main way that designers make the agent react the way they want. For example if you want the agent to go find some food and eat no matter what if the hunger goes above 0.9, you add a consideration to buying food and having food which gives the score of 1 back if hunger is above 0.9 and you might even give a revurse consideration to actions which are not important when you are hungry so as hunger goes up it returns a lower score. The same hunger consideration but with different curves can be used for the purpose in different actions.

How the system works

This is the architecture diagram of the system



Any actor which want to be controlled by utility AI should have a WFUtilityAIComponent component attached to itself or the AI Controller possessing it. This component executes the main utility AI logic by executing all action considerations once in a while and choose a high scoring action to do. Then it calls callbacks of the chosen action which can drive the agent in the world. The action and the considerations have access to the `WFUtilityAIComponent` and the actor which owns it.

Actions which an agent has, have their own considerations listed under them with custom curves per action. A group of actions themselves are stored in an asset called WFBehavior which is what you attach to a `UUtilityAIComponent` to use.

So the root of the tree is the `WFUtilityAIComponent` component which has a set of actions attached and each action contains a set of their own considerations.

You define actions and considerations by deriving classes from `WFAction` and `WFConsideration`.

To create each agent's AI you:

- Create an WFBehavior by right clicking in the content browser and choosing Wise Feline > Behavior.
- Name it whatever makes sense.
- Double click on the WFBehavior asset you just created. The editor window opens.
- Now add actions and considerations and action sets to the behavior.

Different pages of this manual describe each of these items in detail. Read them and come back to this for the bigger picture again to understand it well. There is also a sample which helps you by showing how to implement a semi-realistic sample.

Considerations

Each action's score is calculated by calculating scores of all of its considerations and then multiplying them with each other. Considerations are defined in code or Blueprint by deriving from `UWFCoonsideration` class.

After defining a consideration, you can add it to actions of a Behavior asset. To add considerations to actions

- Implement a consideration in a class.
- Select the desired behavior asset and double click on it to open its editor.
- Select the action which you want to add the consideration to by clicking on it.
- Now in the considerations list for the action click Add Consideration and from the context menu choose the implemented consideration.
- Set the curve and the parameters of the consideration as you desire.

The consideration can be defined like this in C++:

```cpp
// Fill out your copyright notice in the Description page of Project Settings.

#include "Considerations/UAIConsideration_MyAmmo.h"
#include "UAICharacter.h"
#include "UAISampleTypes.h"
#include "UAIWeapon.h"

//------------------------------------------------------------------------------------
// CTOR/DTOR & VIRTUAL FUNCTIONS
//------------------------------------------------------------------------------------

void UUAIConsideration_MyAmmo::OnBeginPlay_Implementation()
{
    Super::OnBeginPlay_Implementation();
    Character = Cast<AUAICharacter>(GetControlledActor());
}

//------------------------------------------------------------------------------------

float UUAIConsideration_MyAmmo::GetValue_Implementation(const AActor* InTargetActor)
{
    if (IsValid(Character.Get()) && IsValid(Character->Weapon.Get()))
    {
        const float TotalBullets = Character->Weapon->TotalBullets;
        const float MaxBullets = Character->Weapon->MaxBullets;
        const float Value = TotalBullets / MaxBullets;
        return Value;
    }

    return 0.0f;
}
```

Or like this in Blueprint

`OnBeginPlay()` is called when the consideration is initialized after the `UUtilityAIComponent` component is created or the behavior asset is added to an already existing component at runtime.

`GetValue()` is called whenever the action wants to calculate its score.

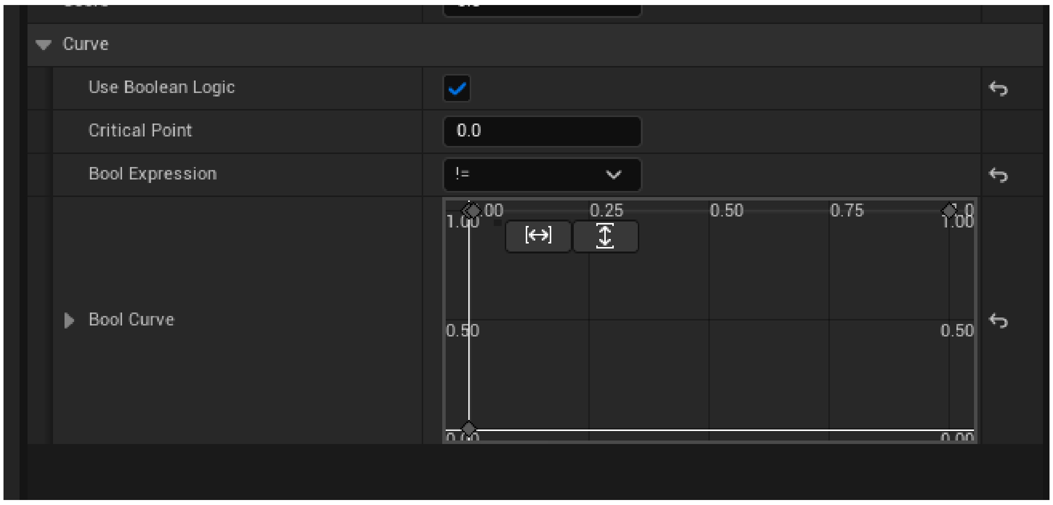
What you return in GetValue should be a number between 0 and 1 (both inclusive) which will be fed to the curve of the consideration as the value for the x axis and the y axis value at that point on the curve is the score of the consideration used in score calculations.

You can check the box for having targets in the behavior editor UI for the consideration so the action's current target is passed to the consideration if needed.

As you can see this consideration assumes that the consideration is used in an actor which is owned by a controller (usually an AIController).

Setting curves for considerations

When you use a consideration with an action, you set its curve depending on what the action needs. For example if your action needs to consider the distance of the agent with a target, it might want to have a higher score if the target is closer or a lower score if it is closer. It also might want a linear relationship or an exponential one. Usually what you want is a curve which goes from 0 to 1 and returns a value between 0 and 1 depending on the shape of the curve, but sometimes you want a boolean curve which returns 0 when the x axis is less than a value or less than or equal a value and 1 otherwise.



Curve

Use Boolean Logic: ✓

Critical Point: 0.0

Bool Expression: !=

Bool Curve

As you can see you can modify the number which is the border between 0 and 1 and also the operator used to decide what happens to the number around the border.

For more information, check the sample and the source code out.

Actions

Actions are the most important part of a utility AI system alongside considerations. An action is a concrete and most of the time atomic unit of work which an agent can do. Shoot, Find health pack, Find cover, Take cover in the found cover, eat food and Choose the best camera to shot the scene with are all examples of actions. However an action doesn't have to be low level and can be pretty high level, you can have a hierarchy of them in your game and for example have a set of actions for groups of NPCs which can be attack the target, spread members in the room, defend sensitive points, find weak spots by bombing every place and then some NPC actions which take command from the group so when the group chooses to attack the target, a member NPC might execute its shoot action or move toward target action or cover the attackers.

To create an action you should inherit a class in C++ or blueprint from `UWFAction`:

Here is an action which moves an actor toward a target.

```
#include "Actions/UAIAction_ActorMoveToTarget.h"

#include "UAICharacter.h"
#include "Kismet/GameplayStatics.h"

// CTOR/DTOR & VIRTUAL FUNCTIONS

UUAIAction_ActorMoveToTarget::UUAIAction_ActorMoveToTarget()
{
}

//gets called every tick when this action is chosen to execute for an AI

void UUAIAction_ActorMoveToTarget::TickAction_Implementation(float DeltaTime)
{
    Super::TickAction_Implementation(DeltaTime);

    if (IsValid(GetControlledActor()) && IsValid(TargetActor))
    {
        const FVector NewLocation = FMath::VInterpConstantTo(GetControlledActor()->GetActorLocation(),
TargetActor->GetActorLocation(), DeltaTime, 250.0f);
        GetControlledActor()->SetActorLocation(NewLocation);
    }
}

//updates the list of potential targets which the action should pick from

void UUAIAction_ActorMoveToTarget::OnUpdateTargets_Implementation()
{
    Super::OnUpdateTargets_Implementation();

    if (IsValid(GetControlledActor()))
    {
        ClearTargetActors();
        TArray<AActor*> OutActors;
        UKismetSystemLibrary::SphereOverlapActors(GetWorld(),
        GetControlledActor()->GetActorLocation(),
        2000.0f,
        TArray<TEnumAsByte<EObjectTypeQuery>>{ObjectTypeQuery3},
        AUAICharacter::StaticClass(),
        TArray<AActor*>{},
        OutActors);
        TargetActors.Append(OutActors);
    }
}
```

There are several things to note:

First of all there are two types of actions, targeted and none-targeted. Targeted ones are those which work on a target like shoot and eat pack. This action is targeted. None-targeted ones are those which don't have a target like idle or useHealthPackFromInventory.

Second there are a set of callbacks in an action which you can override. `OnBeginPlay()` is called as soon as the action is added to the `WfUtilityAIComponent` or in the first frame if the behavior is in the `WFUtilityAIComponent` when the actor is created. `OnTick()` is called every frame when the action is the chosen action to be executed for the actor because it has the highest score. `OnActivate` is called when the action is chosen to become the action for the actor before the first tick. `OnDeactivate` is called when the action is no longer the chosen one to be executed and will be replaced by another action this frame.

If the action is targeted then the `Target` field has the best target in it which you should shoot at, eat or move toward or do whatever which is right for your action. An action becomes a targeted one if you choose the box in the behavior editor window for it and then `OnUpdateTarget()` is called for you to fill the list of targets in. A targeted action means all considerations are calculated for all targets which you gave to the action in the `OnUpdateTargets()` callback and the one with the highest score is set in the `Target` field. `OnUpdateTargets()` is called once per `UpdateTargetsInterval` seconds, which its frequency is chosen in the action itself unlike the `EvalulationIntervalTime` which is the time between action score calculations and is set in the `WFUtilityAIComponent` for all actions.

The scores of all actions are calculated once per `WFUtilityAIComponent::EvaluationIntervalTime` and then the highest score action will be selected to run. Here you just define how the action will execute and that's it.

When filling the list of targets, you can fill it with actors with whatever method you want. You can use EQS, your own special systems for the game or simple raycast or cached lists in a manager script. The only important thing is for the list to be accurate and not take much time to generate.

You don't have to clear and rebuild the list of targets every time either but in that case you should be careful to not add something multiple times and also to remove objects which are not good target candidates as well. Also if your action does not need any targets, You don't need to add any targets in `OnUpdateTargets()`.

For more information, browse the sample and the source code.

UWFBrainComponent component

`UWFBrainComponent` is the main component in the Wise Feline system which allows you to direct a actor using your Behavior assets. You should attach this component to any actor which you want to control using utility AI or the AI Controller which the actor is attached to. The main parameter that the component takes is the Behavior asset to use but there are additional parameters like `EvaluationIntervalTime` which is the time in seconds before action score recalculations.

Keep in mind that this component allows you to add and remove behaviors at runtime so you can add region specific behaviors or situation specific behaviors when an agent needs to consider them during gameplay. This makes your job much easier when designing action scores with considerations. If this feature did not exist you had to have lots of flag considerations with 0 and 1 responses for the times that action was possible or not but thanks to this feature you can simply add jungle specific behaviors to agents only when they enter the jungle zone and remove it when they exit.

The component allows you to choose how often decision making (i.e. action scoring) runs and actions choose how often the action updates its list of targets to consider. You see this by how often your actions and considerations are called to calculate their scores. However the `OnTick()` callback of your current action with its current target (if applicable) will be called every tick no matter how low or high you set the frequency for thinking or updating targets.

How to dynamically disable AI for NPCs which are far away?

You can reduce their evaluation interval time or call `SetComponentTickEnabled(false)` on them when they are far away and then again call `SetComponentTickEnabled(true)` on them when they are closer.

For more information, see the sample projects.

Wise Feline Behavior asset

An Agent Behavior asset defines a list of actions which are scored and executed when the behavior is attached to a `WFUtilityAIComponent`. To modify the WF behavior asset you use the behavior editor window. You can bring the window up by double clicking on a wise feline behavior asset.

To create a new behavior:

- right click in the content browser and go to the Wise Feline menu, then choose *Behavior*.
- Name the behavior whatever you want. Usually this is the name of the agent type or the name of set of behaviors. For example this can be enemy behaviors, forest behaviors or more specifically enemy at forest behaviors.
- Then double click on the the created asset to open the editor window.

Here you can add actions and for each action you can define a set of considerations which effects how the action is scored at each think operation. Closing the window automatically serializes all changes and saves them to the disk.

More details

You can attach behavior assets to `WFUtilityAIComponent` components at edit time or add/remove them dynamically at runtime. Basically this is the data which is used at runtime to see what actions are available to an agent and how each action's score should be calculated.

The `WFUtilityAIComponent` has properties about what behavior assets to use for the actor and how often it should execute the AI and the behavior asset contains what actions are available to execute and how their scores should be calculated.

For more information, checkout the samples

Networking

The networking features of Unreal work well with our Utility AI system. No AI runs on any actor which has a role other than `ROLE_AUTHORITY`. It means client only and server actors in multiplayer games and all actors in standalone games run their UtilityAI logic but simulated and autonomous proxies don't run utility AI and instead receive replicated properties and RPCs from the server version of the actor.

Since actions are `UObject`s and not components or actors, they cannot call RPCs and replicating their properties would become very inefficient. Usually your actions call code in your main actor object or the components attached to it and those components/the actor can easily call RPCs and replicate properties from the server to the client.

If you have an actor which is only available in a client and not spawned on the server then this actor can have its own AI and its role is `ROLE_AUTHORITY` on the client which spawned it. This actor doesn't get spawned on other players machines or the server and cannot call RPCs/replicate properties but can do anything you need client side. For example if you have a utility AI system for the lights in your level which doesn't need to be replicated from the server then you can use client only actors for it.

Integrating with UE features

You can easily integrate EQS, Blackboards and other UE features to your Utility AI actions and considerations. The following is a part of move to cover action of the sample.

```
FEnvQueryRequest HidingSpotQueryRequest = FEnvQueryRequest(EnvQuery, GetControlledActor());
    HidingSpotQueryRequest.Execute(EEnvQueryRunMode::AllMatching, this,
&UWFAction_MoveToCover::MoveToQueryResult);
```

We did not integrate other features like preception and blackboards into the sample but nothing in Wise Feline takes the ability from you and you don't have to consider anything special or follow any specific rules to do so.

How to Debug

There are multiple ways to debug the AI systems of Wise Feline.

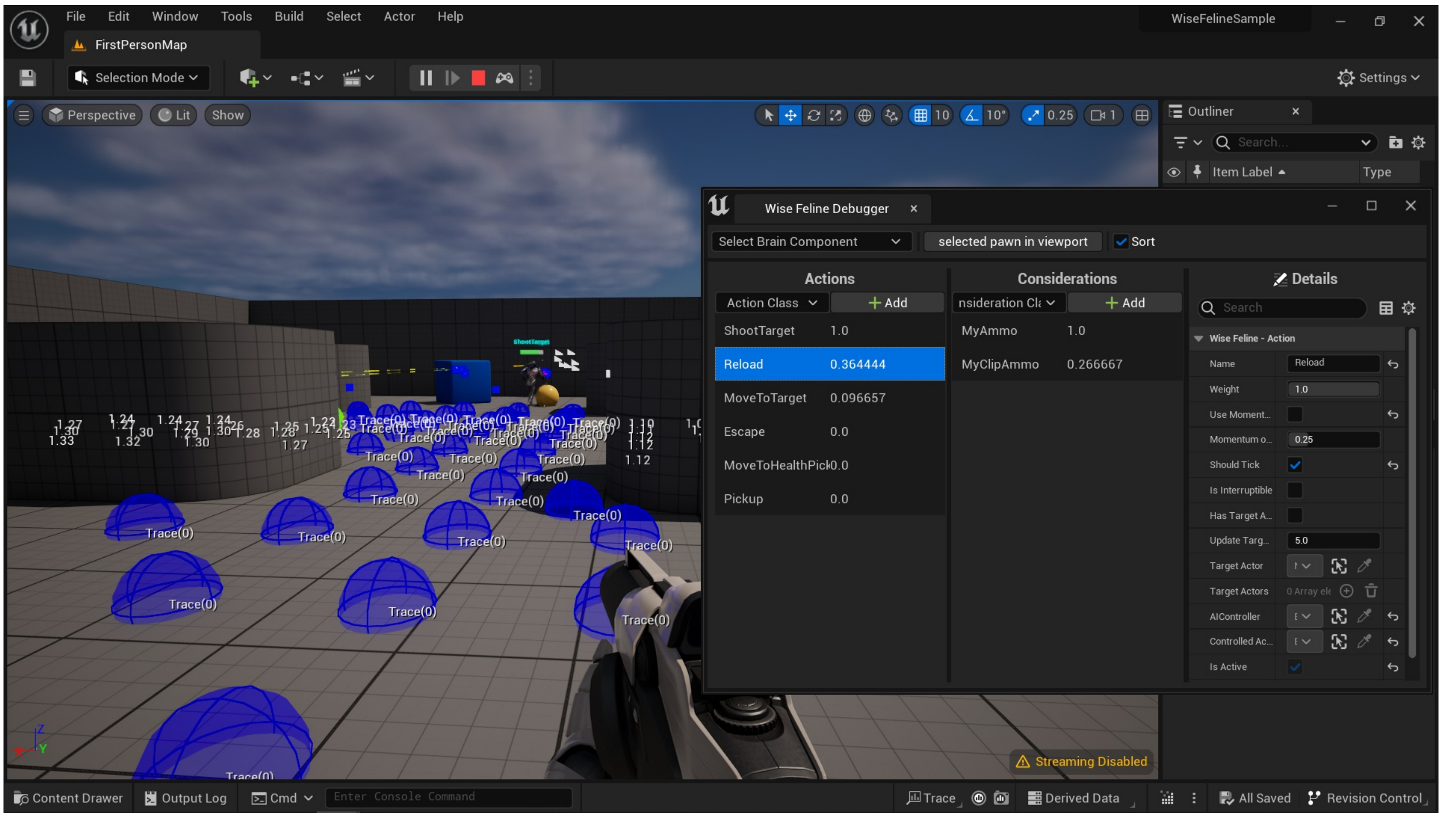
- Wise Feline Debugger
- Visual Logger
- GamePlay Debugger

Wise Feline Debugger

This window can be openned by going to *Tools>Wise Feline Debugger* menu in the Unreal editor.

You can debug your AI scoring by selecting a an actor which has a `WFUtilityAIComponent` component attached (or is posessed by an AI Controller which has one) at runtime while the Wise Feline Debugger window is open. The window then shows the list of actions available to the agent and shows you what score each of the actions got. You can also sort the list of actions by score.

The window looks like this



Actions		Considerations	
ShootTarget	1.0	MyAmmo	1.0
Reload	0.364444	MyClipAmmo	0.266667
MoveToTarget	0.096657		
Escape	0.0		
MoveToHealthPick	0.0		
Pickup	0.0		

Details — Wise Feline - Action

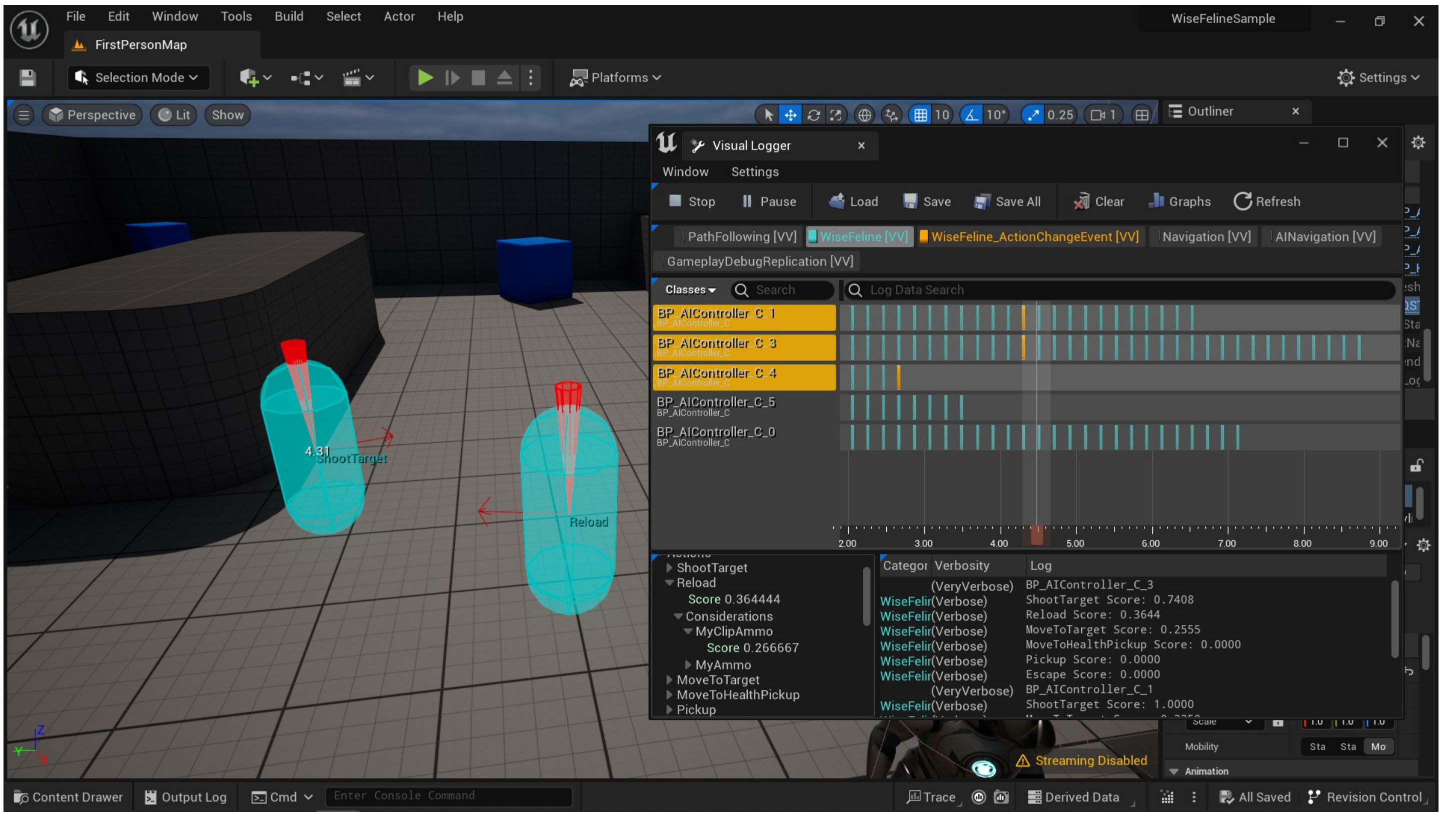
Name	Reload
Weight	1.0
Use Moment...	☐
Momentum o...	0.25
Should Tick	✓
Is Interruptible	☐
Has Target A...	☐
Update Targ...	5.0
Target Actor	
Target Actors	0 Array ele
AIController	
Controlled Ac...	
Is Active	✓

As you can see the left most column are the actions and then considerations. The window updates whenever scores are updated.

Visual Logger

All `WFUtilityAIComponent`s write their log to the visual logger which can be openned using the vislog console command or from the Unreal menu bar. It both prints actions and their scores to the log and the snapshot and also draws shapes which help you debug and mark action change events as well.

This and in general visual logger are very helpful for reproducing bugs and situations which others ran into, specially because it allows you to move back and forth between recorded frames.



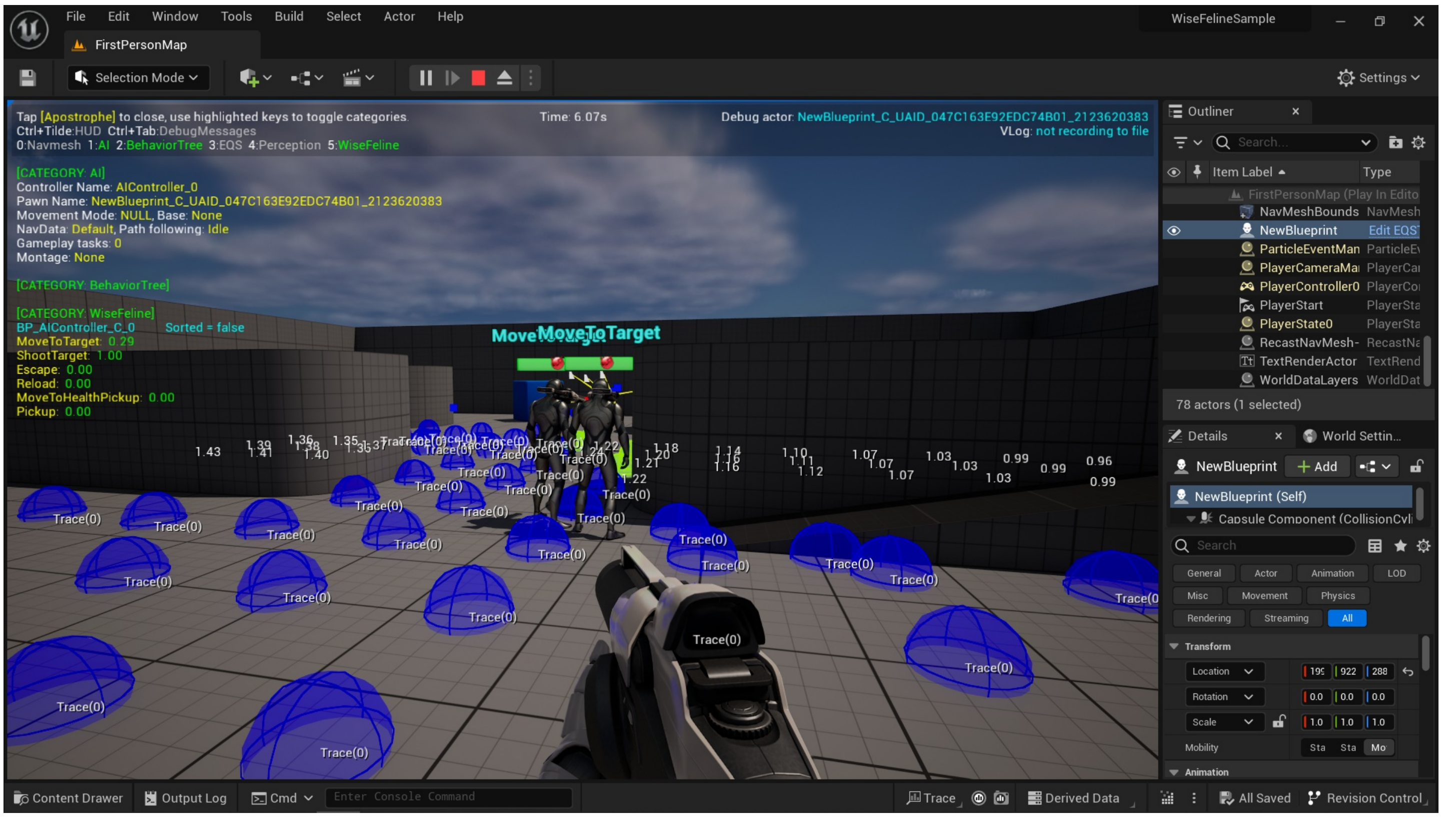
```
ShootTarget
Reload
  Score 0.364444
  Considerations
    MyClipAmmo
      Score 0.266667
    MyAmmo
MoveToTarget
MoveToHealthPickup
Pickup
```

```
Categor Verbosity      Log
        (VeryVerbose)  BP_AIController_C_3
WiseFelir(Verbose)     ShootTarget Score: 0.7408
WiseFelir(Verbose)     Reload Score: 0.3644
WiseFelir(Verbose)     MoveToTarget Score: 0.2555
WiseFelir(Verbose)     MoveToHealthPickup Score: 0.0000
WiseFelir(Verbose)     Pickup Score: 0.0000
WiseFelir(Verbose)     Escape Score: 0.0000
        (VeryVerbose)  BP_AIController_C_1
WiseFelir(Verbose)     ShootTarget Score: 1.0000
```

For more information, check Unreal's Visual Logger documentation.

Gameplay Debugger

The GamePlay Debugger is also a very helpful tool which at runtime shows scores of your top actions to you so you can see why an AI agent is doing what it is doing and not something else.



```
Tap [Apostrophe] to close, use highlighted keys to toggle categories.        Time: 6.07s
Ctrl+Tilde:HUD  Ctrl+Tab:DebugMessages
0:Navmesh  1:AI  2:BehaviorTree  3:EQS  4:Perception  5:WiseFeline

Debug actor: NewBlueprint_C_UAID_047C163E92EDC74B01_2123620383
VLog: not recording to file

[CATEGORY: AI]
Controller Name: AIController_0
Pawn Name: NewBlueprint_C_UAID_047C163E92EDC74B01_2123620383
Movement Mode: NULL, Base: None
NavData: Default, Path following: Idle
Gameplay tasks: 0
Montage: None

[CATEGORY: BehaviorTree]

[CATEGORY: WiseFeline]
BP_AIController_C_0        Sorted = false
MoveToTarget:  0.29
ShootTarget:  1.00
Escape:  0.00
Reload:  0.00
MoveToHealthPickup:  0.00
Pickup:  0.00
```

If you need deeper debugging, then you can use the debugger or visual logger to dig deeper.

The sample

There is a sample in the project with two levels. One is pretty simple and without AI controllers and another is a FPS and has lots of actions and considerations with FPS enemies fully made in utility AI.

Download the sample from GitHub.

How to run the demo

In the Maps folder there are two maps, FirstPerson and Simple. Both can be simply opened and played.

Important Folders

- Sources obviously contains the source code for all actions and considerations and actors and ... and the sample is mostly C++ but we have a few Blueprints to show it is possible to do everything in Blueprints too.
- Content/AI contains the behaviors created for the samples.
- Content/Actions contains the Bluepint actions.
- Content/Considerations contains Blueprint considerations
- Content/Blueprint contains other Blueprints.
- Content/Curves contains a few curves we used in the actions.

Design Resources

There is a good amount of resources regarding utility AI from Dave Mark's book to his videos and other things which all can be found in this wiki page of a utility AI sandbox project here.

We are not associated with that project or Mr. Mark but his book and these videos were very useful to us so we thought it might help you guys too and hopefully we give back a bit to him by sending a few potential book purchases over.