Table of Contents

- Articles
  - Introduction
  - Considerations
  - Actions
  - Utility AI component
  - UAI Behavior assets
  - Wise Feline editor windows
  - Networking and Multiplayer
  - How to debug
  - The Sample
  - Design resources
- Api Documentation
  - NoOpArmy.WiseFeline
    - ActionBase
    - ActionData
    - ActionSet
    - AgentBehavior
    - AIObject
    - Brain
    - ConsiderationBase
    - ExtensionMethods
    - ReadOnlyAttribute
    - ReflectionUtilities
    - Search

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.

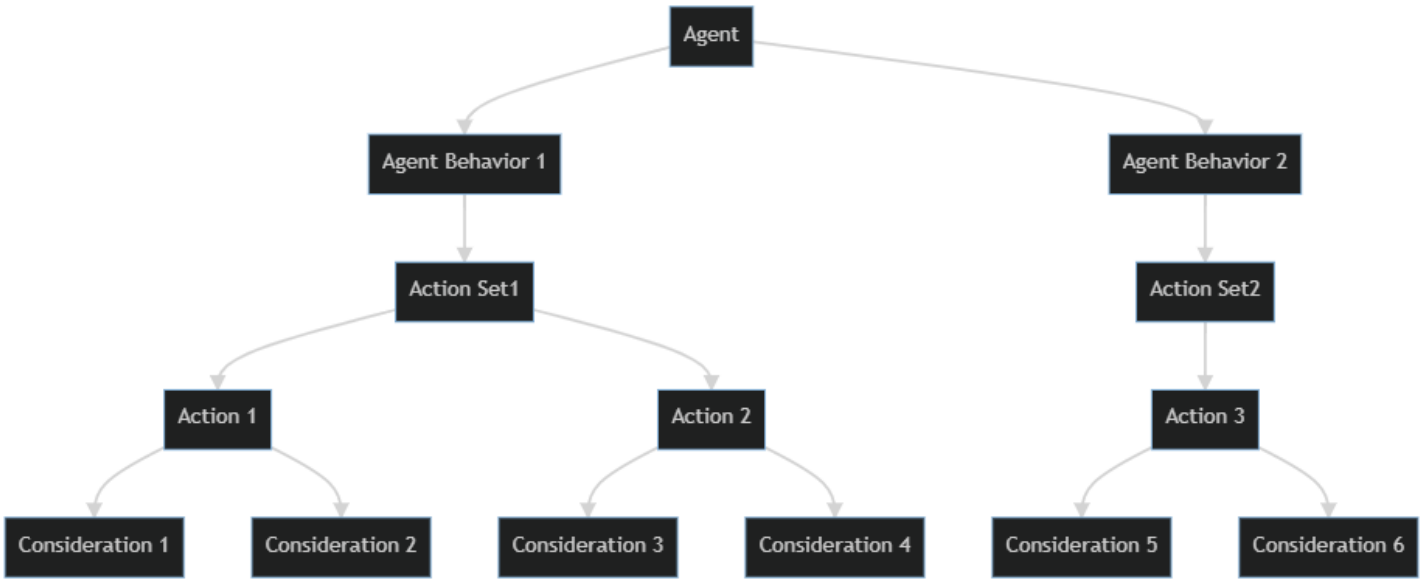
Buying food on the other hand has considerations for not being tired and not having food and being hungry and these should return their values in a way that if you are very hungry, the score becomes high even if you are tired. Resting action probably should only give a high score if you are tired but 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 UAIUtilityAIComponent component attached. 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 `UUtilityAIComponent` 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 UAIBehavior which is what you attach to a `UUtilityAIComponent` to use.

So the root of the tree is the `UUtilityAIComponent` 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 `UAIAction` and `Consideration`.

To create each agent's AI you:

- Create an UAIBehavior by right clicking in the project and choosing Create>NoOpArmy>Wise Feline>Behavior.
- Name it whatever makes sense.
- Select the UAIBehavior asset you just created.
- Open the Wise Feline window by going to Window>NoOpArmy>Wise Feline window.
- 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 demo 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 `UAICoonsideration` 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. For example create a HungerConsideration.cs file and implement `HungerConsideration` inside it like this. The example code is posted below.
- Select the desired agent behavior asset and open the Wise Feline window at *Window>NoOpArmy>Wise Feline*.
- Select the action which you want to add the consideration to by clicking on it. You might need to select the appropriate action set first.
- Now in the considerations list for the action click Add Consideration and from the context menu choose `HungerConsideration`
- Set the curve and the parameters of the consideration as you desire.

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

```
// 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 agent 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.

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

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 `UAIAction`:

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 cahced 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.

WFUtilityAIComponent component

`WFUtilityAIComponent` 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.

For more information, see the samples.

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

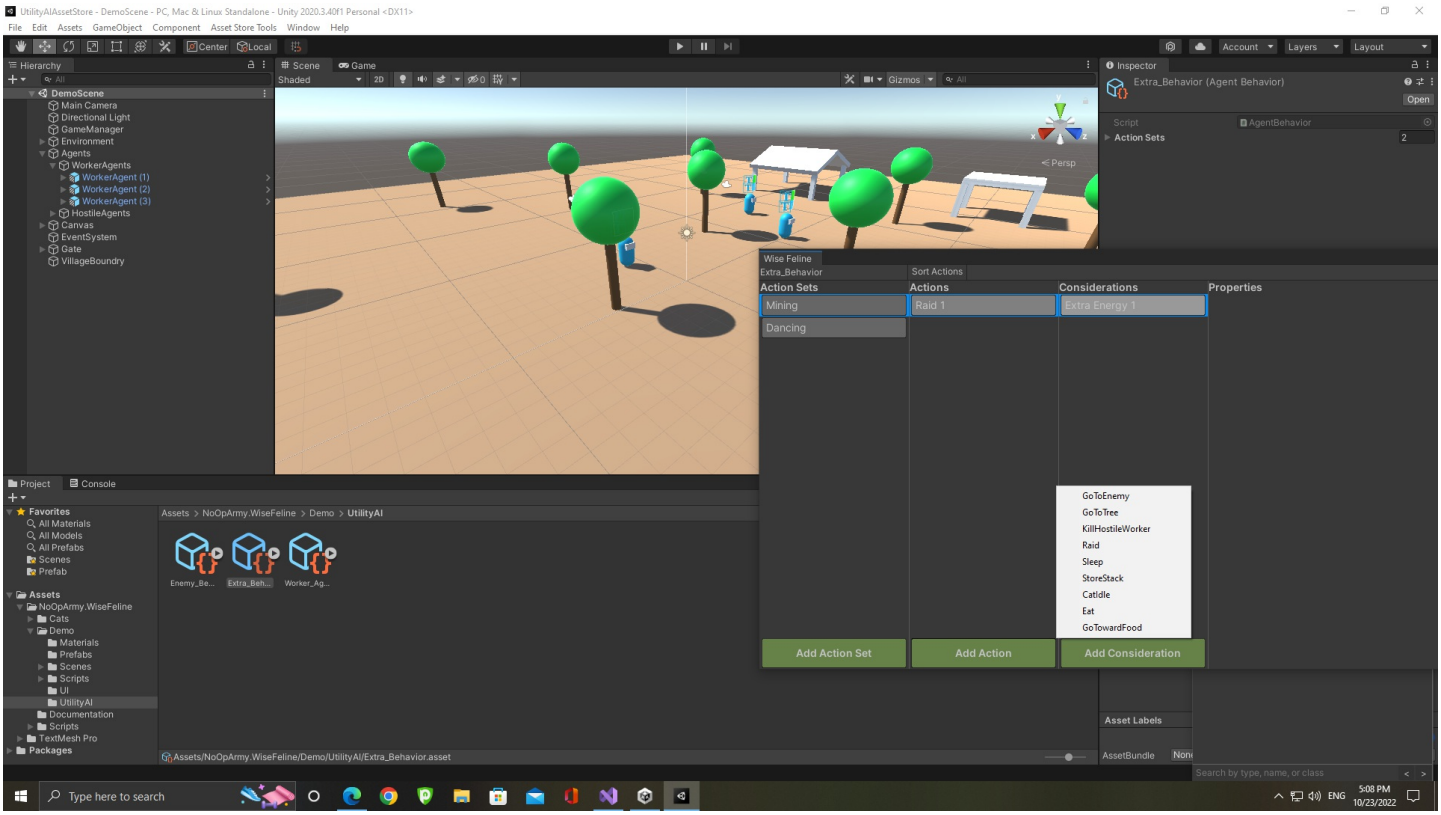
You can attach behavior assets to `WFUtilityAIComponent` components at edit time or add/remove them dynamically at runtime. Basically this is the data which is used at runtime to see what actions are available to an agent and how each action's score should be calculated.

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

Wise Feline Editor Window

The editor window can be opened at Window>NoOpArmy>Wise Feline



When the window is open you can select a Behavior Agent asset and then edit the actions, considerations and action sets available in the asset.

At runtime the window can be used to debug the scores of actions to see if curves and waits are set correctly to get the results you want. You just need to select game objects which have the brain attached in the hierarchy/scene and then open this window to see how its scores are being calculated.

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.

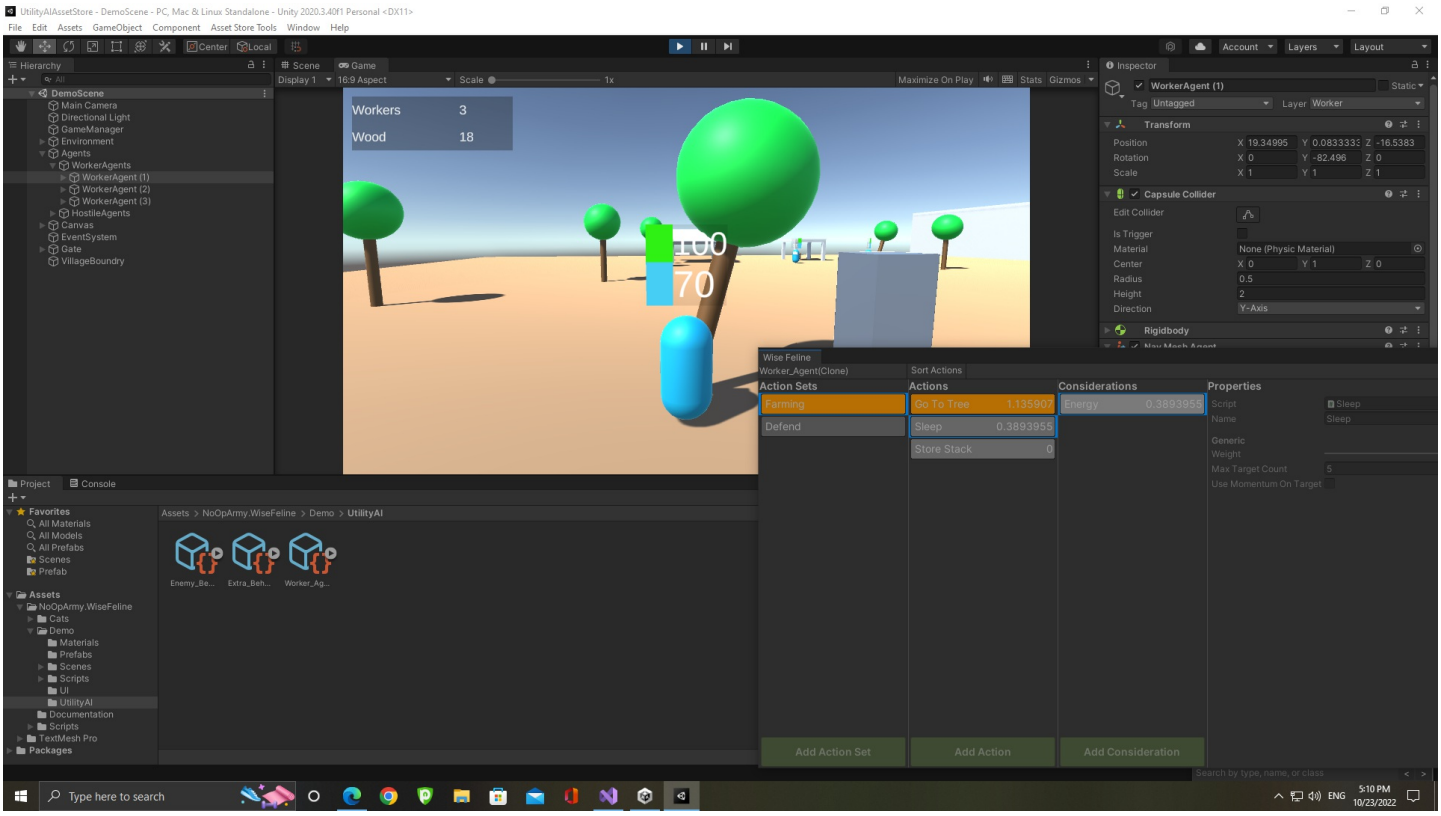
Since actions are `UObject`s and not components or actors, they cannot call RPCs and replicating their properties would become very inefficient. Usually your actions call code in your main actor object or the components attached to it and those components/the actor can easily call RPCs and replicate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

How to Debug

You can debug your AI scoring by selecting a game object which has a `Brain` component attached at runtime while the Wise Feline window is open. The window then shows the list of all action sets and 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



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

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 here

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.

Namespace NoOpArmy.WiseFeline

Classes

ActionBase

AI actions should derive from this class to define a custom action which can be added to the set of actions for an agent

ActionSet

Action sets are like directories which group a set of actions together but you don't need to deal with them in code unless you are making debugging/editor tools or highly advanced systems An Agent Behavior asset is made of a set of ActionSet objects which each of them contain a set of actions.

AgentBehavior

This class is used to serialize a group of action sets as a scriptable object. You usually create one of these by going to the Assets>Create>NoOpArmy>WiseFeline>AgentBehavior menu and then select it and open the Window>NoOpArmy>WiseFeline to add action sets, actions and considerations to it and modify their parameters.

AIObject

This is the base class of Wise Feline AI assets. You never need to use this directly.

Brain

You should attach this component to any GameObject which wishes to be an AI agent controlled by a set of actions

ConsiderationBase

Considerations should inherit from this class. Each action has a list of these and multiplies the value of calling their GetValue methods to calculate its score. Each consideration is either a consideration for the target of the action or the action itself. The target based ones will be executed once per target when the action score is being calculated for that target.

ExtensionMethods

These are different methods extending types with methods we needed to make the code more readable.

ReadOnlyAttribute

This attribute has a property drawer which causes the property to be drawn without the ability to be edited.

ReflectionUtilities

Contains utility methods for working with data types defined as actions and considerations and ... You should not need to use this unless you are making debug/editor tools for Wise Feline

Search

This is a utility class which you can use to find GameObjects in the scene. However this is not the most performant way to do so and you can use any mechanism which suits your game. However these methods have the minimum like not allocating memory for colliders

Structs

ActionData

This struct represents the runtime info for a an action and its final score

Class ActionBase

AI actions should derive from this class to define a custom action which can be added to the set of actions for an agent

Inheritance

System.Object
AIObject
ActionBase

Inherited Members

AIObject.guid
AIObject.Name

Namespace: NoOpArmy.WiseFeline

Assembly: cs.temp.dll.dll

Syntax

```
public abstract class ActionBase : AIObject
```

Fields

_maxTargetCount

Maximum number of targets which this action should consider

Declaration

```
protected int _maxTargetCount
```

Field Value

TYPE	DESCRIPTION
System.Int32	

_useMomentumOnTarget

Should the action add a 25% score bonus to the current target to not change the target multiple times too quickly when scores are too close.

Declaration

```
protected bool _useMomentumOnTarget
```

Field Value

TYPE	DESCRIPTION
System.Boolean	

ChosenTarget

The target with the best score. The type of this component depends on the type that you yourself store in the list in the UpdateTargets callback. If the action doesn't have any targets and target based considerations then this field doesn't matter and it value should be considered undefined.

Declaration

```
protected Component ChosenTarget
```

Field Value

TYPE	DESCRIPTION
Component	

Properties

Brain

The brain component which the action is executing for.

Declaration

```
protected Brain Brain { get; }
```

Property Value

TYPE	DESCRIPTION
Brain	

Considerations

List of the considerations for the action. The score of the action is the result of multiplication of the scores of all of these considerations

Declaration

```
public List<ConsiderationBase> Considerations { get; }
```

Property Value

TYPE	DESCRIPTION
System.Collections.Generic.List<ConsiderationBase>	

IsInitialized

Is the action initialized?

Declaration

```
public bool IsInitialized { get; protected set; }
```

Property Value

TYPE	DESCRIPTION
System.Boolean	

Score

The last calculated score of the action in the last think operation.

Declaration

```
public float Score { get; }
```

Property Value

TYPE	DESCRIPTION
System.Single	

Weight

The wait of the action which is multiplied by the score to allow you to prioritize some actions

Declaration

```
public float Weight { get; }
```

Property Value

TYPE	DESCRIPTION
System.Single	

Methods

ActionFailed()

Declaration

```
protected void ActionFailed()
```

ActionSucceed()

Declaration

```
protected void ActionSucceed()
```

AddTarget(Component)

Adds a component to the targets list of the action

Declaration

```
protected void AddTarget(Component t)
```

Parameters

TYPE	NAME	DESCRIPTION
Component	t	

AddTargets<T>(T[])

Adds an array of targets to the list of targets for the action

Declaration

```
protected void AddTargets<T>(T[] targets)
    where T : Component
```

Parameters

TYPE	NAME	DESCRIPTION
T[]	targets	

Type Parameters

NAME	DESCRIPTION
T	

AddTargets<T>(List<T>)

Adds an array of targets to the list of targets for the action

Declaration

```
protected void AddTargets<T>(List<T> targets)
    where T : Component
```

Parameters

TYPE	NAME	DESCRIPTION
System.Collections.Generic.List<T>	targets	

Type Parameters

NAME	DESCRIPTION
T	

ClearTargets()

Clears the list of targets for the action

Declaration

```
protected void ClearTargets()
```

OnFinish()

Called when the action is finished, either by failing or succeeding in achieving a desired behaviour

Declaration

```
protected virtual void OnFinish()
```

OnFixedUpdate()

Should be used like MonoBehaviour's FixedUpdate for the action

Declaration

```
protected virtual void OnFixedUpdate()
```

OnInitialized()

Called when the action is initialized

Declaration

```
protected virtual void OnInitialized()
```

OnLateUpdate()

Should be used like MonoBehaviour's LateUpdate for the action

Declaration

```
protected virtual void OnLateUpdate()
```

OnStart()

Should be used like MonoBehaviour's Start for the action

Declaration

```
protected virtual void OnStart()
```

OnUpdate()

Should be used like MonoBehaviour's Update for the action

Declaration

```
protected virtual void OnUpdate()
```

RemoveTarget(Component)

Removes a component from the list of targets for the action

Declaration

```
protected void RemoveTarget(Component t)
```

Parameters

TYPE	NAME	DESCRIPTION
Component	t	

UpdateTargets()

Fires when you need to update the list of targets

Declaration

```
protected abstract void UpdateTargets()
```

Struct ActionData

This struct represents the runtime info for a an action and its final score

Inherited Members

System.ValueType.Equals(System.Object)
System.ValueType.GetHashCode()
System.ValueType.ToString()
System.Object.Equals(System.Object, System.Object)
System.Object.ReferenceEquals(System.Object, System.Object)
System.Object.GetType()

Namespace: NoOpArmy.WiseFeline

Assembly: cs.temp.dll.dll

Syntax

```
public struct ActionData
```

Constructors

ActionData(ActionSet, ActionBase, Single)

Declaration

```
public ActionData(ActionSet set, ActionBase action, float score)
```

Parameters

TYPE	NAME	DESCRIPTION
ActionSet	set	
ActionBase	action	
System.Single	score	

Fields

Action

Which action this instance refers to.

Declaration

```
public ActionBase Action
```

Field Value

TYPE	DESCRIPTION
ActionBase	

ActionSet

The action set this instance refers to.

Declaration

```
public ActionSet ActionSet
```

Field Value

TYPE	DESCRIPTION
ActionSet	

Score

The score of the action in the last think operation.

Declaration

```
public float Score
```

Field Value

TYPE	DESCRIPTION
System.Single	

Class ActionSet

Action sets are like directories which group a set of actions together but you don't need to deal with them in code unless you are making debugging/editor tools or highly advanced systems An Agent Behavior asset is made of a set of ActionSet objects which each of them contain a set of actions.

Inheritance

System.Object
AIObject
ActionSet

Inherited Members

AIObject.guid
AIObject.Name

Namespace: NoOpArmy.WiseFeline

Assembly: cs.temp.dll.dll

Syntax

```
public class ActionSet : AIObject
```

Fields

Behavior

The agent behavior that this instance belongs too.

Declaration

```
public AgentBehavior Behavior
```

Field Value

TYPE	DESCRIPTION
AgentBehavior	

Properties

Actions

The list of actions which are contained in this action set.

Declaration

```
public List<ActionBase> Actions { get; }
```

Property Value

TYPE	DESCRIPTION
System.Collections.Generic.List<ActionBase>	

Class AgentBehavior

This class is used to serialize a group of action sets as a scriptable object. You usually create one of these by going to the Assets>Create>NoOpArmy>WiseFeline>AgentBehavior menu and then select it and open the Window>NoOpArmy>WiseFeline to add action sets, actions and considerations to it and modify their parameters.

Inheritance

System.Object
AgentBehavior

Namespace: NoOpArmy.WiseFeline

Assembly: cs.temp.dll.dll

Syntax

```
public class AgentBehavior : ScriptableObject
```

Fields

OnThinkDone

Fires when an action is done

Declaration

```
public Action<ActionData> OnThinkDone
```

Field Value

TYPE	DESCRIPTION
System.Action<ActionData>	

SelectedAction

The action currently selected

Declaration

```
public ActionBase SelectedAction
```

Field Value

TYPE	DESCRIPTION
ActionBase	

SelectedActionSet

The action set selected at the moment in the UI

Declaration

```
public ActionSet SelectedActionSet
```

Field Value

TYPE	DESCRIPTION
ActionSet	

Properties

ActionSets

The list of action sets in the asset

Declaration

```
public List<ActionSet> ActionSets { get; }
```

Property Value

TYPE	DESCRIPTION
System.Collections.Generic.List<ActionSet>	

Methods

Clone()

Clones the behavior for runtime execution by a Brain component.

Declaration

```
public AgentBehavior Clone()
```

Returns

TYPE	DESCRIPTION
AgentBehavior	

Class AIObject

This is the base class of Wise Feline AI assets. You never need to use this directly.

Inheritance

System.Object
AIObject
ActionBase
ActionSet
ConsiderationBase

Namespace: NoOpArmy.WiseFeline

Assembly: cs.temp.dll.dll

Syntax

```
public class AIObject : ScriptableObject
```

Fields

guid

The unique id of the asset.

Declaration

```
public string guid
```

Field Value

TYPE	DESCRIPTION
System.String	

Name

The name of the asset

Declaration

```
public string Name
```

Field Value

TYPE	DESCRIPTION
System.String	

Class Brain

You should attach this component to any GameObject which wishes to be an AI agent controlled by a set of actions

Inheritance

System.Object
Brain

Namespace: NoOpArmy.WiseFeline

Assembly: cs.temp.dll.dll

Syntax

```
public class Brain : MonoBehaviour
```

Fields

_thinkDuration

The wait period between calculations of action scores

Declaration

```
public float _thinkDuration
```

Field Value

TYPE	DESCRIPTION
System.Single	

_updateTargetsDuration

The wait period between calls to update target lists. Since updating this list is usually heavy, you should do this less often than score calculations and deal with null targets in your actions.

Declaration

```
public float _updateTargetsDuration
```

Field Value

TYPE	DESCRIPTION
System.Single	

OnBehaviorListModified

Fires when the list of behaviors available to this brain is modified by calling AddBehavior() or RemoveBehavior()

Declaration

```
public Action OnBehaviorListModified
```

Field Value

TYPE	DESCRIPTION
Action	

Properties

Behavior

Declaration

```
public AgentBehavior Behavior { get; }
```

Property Value

TYPE	DESCRIPTION
AgentBehavior	

Methods

AddActionSet(ActionSet)

Adds an action set to the set of behaviors this brain has available to score and execute

Declaration

```
public void AddActionSet(ActionSet set)
```

Parameters

TYPE	NAME	DESCRIPTION
ActionSet	set	The action set to add

Remarks

Duplicat actions which already exist in the brain will be added again too so you should design your action sets acordingly. You usually do not need multiple instances of an action in the same brain

AddBehavior(AgentBehavior)

Adds actions inside a behavior agent asset to the actions of this brain

Declaration

```
public void AddBehavior(AgentBehavior agentBehavior)
```

Parameters

TYPE	NAME	DESCRIPTION
AgentBehavior	agentBehavior	The agent behavior asset to add

Remarks

Duplicate actions which already exist in the brain will be added again too. You should design your behaviors so they don't have duplicate actions You usually do not need multiple instances of an action in the same brain

PauseExecutingActions(Boolean)

Pauses/Resumes executing the current action.

Declaration

```
public void PauseExecutingActions(bool pause)
```

Parameters

TYPE	NAME	DESCRIPTION
System.Boolean	pause	pauses ecution if true and resumes it if false

PauseThinking(Boolean)

Puases/Resumes thinking which changes action scores

Declaration

```
public void PauseThinking(bool pause)
```

Parameters

TYPE	NAME	DESCRIPTION
System.Boolean	pause	Pauses the thinking if true and resumes it if false.

RemoveActionSet(ActionSet)

Removes a specific action set from the list of behaviors of this brain

Declaration

```
public void RemoveActionSet(ActionSet set)
```

Parameters

TYPE	NAME	DESCRIPTION
ActionSet	set	

RemoveBehavior(AgentBehavior)

Removes actions of a behavior from this brain

Declaration

```
public void RemoveBehavior(AgentBehavior agentBehavior)
```

Parameters

TYPE	NAME	DESCRIPTION
AgentBehavior	agentBehavior	

Class ConsiderationBase

Considerations should inherit from this class. Each action has a list of these and multiplies the value of calling their GetValue methods to calculate its score. Each consideration is either a consideration for the target of the action or the action itself. The target based ones will be executed once per target when the action score is being calculated for that target.

Inheritance

System.Object
AIObject
ConsiderationBase

Inherited Members

AIObject.guid
AIObject.Name

Namespace: NoOpArmy.WiseFeline

Assembly: cs.temp.dll.dll

Syntax

```
public abstract class ConsiderationBase : AIObject
```

Fields

instantiatePerAgent

If true, a new instance of the consideration will be created per action per instance of the brain component which needs this. Otherwise a single instance will be used for all of them. If you don't use fields in the class and the GetValue method is stateless, then turn this off to allocate less memory.

Declaration

```
protected bool instantiatePerAgent
```

Field Value

TYPE	DESCRIPTION
System.Boolean	

maxRange

Maximum value that the consideration can return from GetValue

Declaration

```
protected float maxRange
```

Field Value

TYPE	DESCRIPTION
System.Single	

minRange

Minimum value that the consideration can return from GetValue

Declaration

```
protected float minRange
```

Field Value

TYPE	DESCRIPTION
System.Single	

Properties

Brain

The Brain component that this consideration is being attached to one of its actions

Declaration

```
protected Brain Brain { get; }
```

Property Value

TYPE	DESCRIPTION
Brain	

Score

The score of this consideration returned from the last GetValue call.

Declaration

```
public float Score { get; }
```

Property Value

TYPE	DESCRIPTION
System.Single	

Methods

GetValue(Component)

This method should return the current value of the consideration which is used for multiplication with other considerations to calculate the score of the action

Declaration

```
protected abstract float GetValue(Component target)
```

Parameters

TYPE	NAME	DESCRIPTION
Component	target	If the consideration has a target then this value is the target component which the consideration should be calculated for. The actual type of this depends on the type of the component that the action stores in its target list in the UpdateTargets call. If the consideration doesn't have a target, then the Brain component of the executing agent itself is passed to the method.

Returns

TYPE	DESCRIPTION
System.Single	A value between minRange and maxRange which indicates the current value/score of the consideration

OnInitialized()

Fired when the consideration is initialized

Declaration

```
protected virtual void OnInitialized()
```

UpdateRange(Single, Single)

Updates the range of acceptable values from GetValue

Declaration

```
public void UpdateRange(float min, float max)
```

Parameters

TYPE	NAME	DESCRIPTION
System.Single	min	
System.Single	max	

Class ExtensionMethods

These are different methods extending types with methods we needed to make the code more readable.

Inheritance

System.Object
ExtensionMethods

Inherited Members

System.Object.ToString()
System.Object.Equals(System.Object)
System.Object.Equals(System.Object, System.Object)
System.Object.ReferenceEquals(System.Object, System.Object)
System.Object.GetHashCode()
System.Object.GetType()
System.Object.MemberwiseClone()

Namespace: NoOpArmy.WiseFeline

Assembly: cs.temp.dll.dll

Syntax

```
public static class ExtensionMethods
```

Methods

MaxIndex(Single[])

Gets the index of the item with the highest value in the array.

Declaration

```
public static int MaxIndex(this float[] array)
```

Parameters

TYPE	NAME	DESCRIPTION
System.Single[]	array	

Returns

TYPE	DESCRIPTION
System.Int32	

MaxIndices(Single[])

Returns the indices of all items with the maximum value in the array.

Declaration

```
public static int[] MaxIndices(this float[] array)
```

Parameters

TYPE	NAME	DESCRIPTION
System.Single[]	array	

Returns

TYPE	DESCRIPTION
System.Int32[]	

Class ReadOnlyAttribute

This attribute has a property drawer which causes the property to be drawn without the ability to be edited.

Inheritance

System.Object

ReadOnlyAttribute

Namespace: NoOpArmy.WiseFeline

Assembly: cs.temp.dll.dll

Syntax

```
public class ReadOnlyAttribute : PropertyAttribute
```

Class ReflectionUtilities

Contains utility methods for working with data types defined as actions and considerations and ... You should not need to use this unless you are making debug/editor tools for Wise Feline

Inheritance

System.Object
ReflectionUtilities

Inherited Members

System.Object.ToString()
System.Object.Equals(System.Object)
System.Object.Equals(System.Object, System.Object)
System.Object.ReferenceEquals(System.Object, System.Object)
System.Object.GetHashCode()
System.Object.GetType()
System.Object.MemberwiseClone()

Namespace: NoOpArmy.WiseFeline

Assembly: cs.temp.dll.dll

Syntax

```
public static class ReflectionUtilities
```

Methods

GetAllDerivedTypes(Type)

Get all types derived from a specific type defined in the project.

Declaration

```
public static Type[] GetAllDerivedTypes(Type baseType)
```

Parameters

TYPE	NAME	DESCRIPTION
Type	baseType	

Returns

TYPE	DESCRIPTION
Type[]	

Class Search

This is a utility class which you can use to find GameObjects in the scene. However this is not the most performant way to do so and you can use any mechanism which suits your game. However these methods have the minimum like not allocating memory for colliders

Inheritance

System.Object
Search

Inherited Members

System.Object.ToString()
System.Object.Equals(System.Object)
System.Object.Equals(System.Object, System.Object)
System.Object.ReferenceEquals(System.Object, System.Object)
System.Object.GetHashCode()
System.Object.GetType()
System.Object.MemberwiseClone()

Namespace: NoOpArmy.WiseFeline

Assembly: cs.temp.dll.dll

Syntax

```
public static class Search
```

Remarks

We intend to release additional modules which makes it easier to work with different geo-spacial structures and gameplay types and search for objects

Methods

FindClosestCollider(Vector3, Single, LayerMask, ref Collider[])

Find the closes collider in a sphere without allocating a new array for the raycast results.

Declaration

```
public static Collider FindClosestCollider(Vector3 center, float radius, LayerMask layerMask, ref Collider[]
colliders)
```

Parameters

TYPE	NAME	DESCRIPTION
Vector3	center	The center of the sphere
System.Single	radius	The radius of the sphere
LayerMask	layerMask	The layermask which we should do the cast against
Collider[]	colliders	The array of colliders to be used for the cast operation

Returns

TYPE	DESCRIPTION
Collider	The closes collider or null if no colliders can be found

GetOverlappingColliders(Vector3, Single, LayerMask, ref Collider[])

Gets the array of overlapping colliders with a sphere

Declaration

```
public static void GetOverlappingColliders(Vector3 center, float radius, LayerMask layerMask, ref Collider[]
colliders)
```

Parameters

TYPE	NAME	DESCRIPTION
Vector3	center	The center of the sphere
System.Single	radius	The radius of the sphere
LayerMask	layerMask	The layer mask to do the cast against
Collider[]	colliders	The array of colliders to be filled by the cast operation

GetSortedTransforms(Vector3, Single, LayerMask, ref Collider[], ref List<Transform>, Int32)

Gets the list of colliders overlapping by a sphere sorted with their distance from the center. This is an expensive method only good for samples and quick prototyping because it uses a relatively expensive sort operation

Declaration

```
public static void GetSortedTransforms(Vector3 center, float radius, LayerMask layerMask, ref Collider[]
colliders, ref List<Transform> transforms, int size = 0)
```

Parameters

TYPE	NAME	DESCRIPTION
Vector3	center	The center of the sphere
System.Single	radius	The radius of the sphere
LayerMask	layerMask	The layer mask to cast against
Collider[]	colliders	The array of colliders to use for the casting

TYPE	NAME	DESCRIPTION
System.Collections.Generic.List<Transform>	transforms	The transforms list to fill
System.Int32	size	The maximum size of the list