Axela Brockett UE5 Plugin: Utility AI

v1.0.2 Documentation

GitHub download link: https://github.com/AxelaBrockett/Utility-AI-Plugin-UE5.6

Contents:

Basic Overview of the Plugin

Utility AI is a decision making method for AI that gives the AI a list of Actions to choose from, gives each of those Actions a score, and then the AI picks the most appropriate Action, most commonly the highest scoring one. This plugin uses the following terms throughout:

- Action
    - A task the AI can do, similar to a Behavior Tree task or State Tree task
- Requirement
    - A prerequisite that must be passed for the action to be available in the moment
- Factor
    - A Factor is considered when calculating the score for an Action

Other terms will also be explained as they appear within the documentation.
The plugin contains a Utility Component, base classes for an Action, a Requirement and a Factor, a Data Asset class, helper files, debugging and validation code. This documentation will go over each of the available classes and how to use each of them to create a usable AI.

FAxelaBrockettUtilityAIBaseModule

This is the base module class that is used for the runtime part of this plugin.

Parent class: *IModuleInterface*

Functions:

Function	**Params**	**Logic**
IModuleInterface implementation		
virtual void StartupModule() override		Registers the Gameplay Debugger Category for Utility AI
virtual void ShutdownModule() override		If the Gameplay Debugger is still available at the time this is called, unregisters the Gameplay Debugger Category for Utility AI

FAxelaBrockettUtilityAIEditorBaseModule

This is the base module class that is used for the editor part of this plugin.

Parent class: *IModuleInterface*

Functions:

Function	**Params**	**Logic**
IModuleInterface implementation		
virtual void StartupModule() override		Empty
virtual void ShutdownModule() override		Empty

UABUtilityComponent

This is the Brain Component that is run by the AI Controller. It contains the Utility Data Asset that has the available Actions, and has both synchronous and asynchronous logic available to the user. The component is completely event driven, so ticking is actively turned off in the constructor, and the *SetComponentTickEnabled(bool bEnabled)* function has been overridden to ensure tick does not get activated. It is not recommended to allow this component to tick.

Parent class: *UBrainComponent*

Functions:

Function	**Params**	**Logic**
UABUtilityComponent()		Used to turn off the component tick, this is an event driven component
UBrainComponent overrides		
void SetComponentTickEnabled(bool bEnabled) override	*bEnabled* - Usually allows users to activate or deactivate the tick functionality	Overridden to ensure tick stays disabled as this component is completely event driven
void StartLogic() override		Caches the Actions from the Utility Data Asset, binds to relevant delegates, and tries to start the first Action
void PauseLogic(const FString& Reason) override	*Reason* - Allows users to specify why the component is pausing	Logs that the component is pausing
EAILogicResuming::Type ResumeLogic(const FString& Reason) override	*Reason* - Allows users to specify why the component is now resuming	If *ShouldRestartLogicWhenResumingComponentLogic()* returns *true*, the component will restart. Otherwise, if there is a valid current Action, it will attempt to keep running

		that. If there is no valid current Action or if the valid action fails to resume, a new Action will be chosen. Returns the logic resuming type of either *Continue* or *RestartedInstead*
void RestartLogic() override		Calls *StopLogic* and then *StartLogic*
void StopLogic(const FString& Reason) override	*Reason* - Allows users to specify why the component is stopping	Aborts the current Action, unbinds from all Action delegates, and empties the Action cache
void Cleanup() override		Calls *StopLogic*
Getters		
const UABUtilityActionBase GetCurrentAction() const*		Returns a *const* pointer to the current Action
const UABUtilityActionBase GetLastSuccessfulAction() const*		Returns a *const* pointer to the last successful Action
const EABActionSelectionMode & GetActionSelectionMode() const		Returns a *const* reference to the Action selection mode
const FTimerHandle& GetSuccessfulActionCooldownHandle() const		Returns a *const* reference to the *SuccessfulActionCooldownHandle*
const FAJBUtilityComponentAsyncData& GetAsyncData() const		Returns a *const* reference to the components async data
FAJBUtilityComponentAsyncData& GetMutableAsyncData()		Returns a mutable reference to the components async data
const		Returns a *const* reference

TArray<TObjectPtr<UABUtilityActionBase>>& GetUtilityActions() const		to the components available Actions
Pause and Abort functions		
virtual void AbortCurrentAction(const EABUtilityAbortActionLogic& AbortLogic)	*AbortLogic* - Allow the user to specify what happens to the component when we abort the current Action - stopping the component, pausing the component, or choosing a new Action	Aborts the current Action and then reacts to the input parameter by either calling *StopLogic*, calling *PauseLogicWithActionDecision*, or by choosing a new Action
virtual void PauseLogicWithActionDecision(const FString& Reason, const EABUtilityComponentPauseActionLogic& ActionDecision)	*Reason* - Allow the user to specify why we are pausing the logic *ActionDecision* - Allow the user to specify what happens to the current Action upon pausing the component	Either pauses or aborts the current Action if there is one, then passes the *Reason* into a *PauseLogic* call
Inlines		
const bool ShouldRestartLogicWhenResumingComponentLogic() const		Returns the current status of *bRestartLogicWhenResumingComponentLogic*
const bool IsComponentRunningAsynchronously() const		Returns the current status of *bRunComponentAsynchronously*
const bool IsUsingParallelRequirementChecking() const		Returns the current status of *bUseParallelRequirementChecking*
const bool ShouldRetryOnFailure() const		Returns the current status of *bRetryOnFailure*
Replication functions		
void GetLifetimeReplicatedProps(TArray<FLifetimeProp	*OutLifetimeProps* - Used to track a property that is marked to be replicated	Used to specify which members are to be replicated

erty>& OutLifetimeProps) const override	for the lifetime of the actor channel. This doesn't mean the property will necessarily always be replicated, it just means: "check this property for replication for the life of the actor, and I don't want to think about it anymore". A secondary condition can also be used to skip replication based on the condition results	
UFUNCTION(BlueprintNativeEvent) void OnRep_CurrentAction()		Blueprint function that runs logic for when *CurrentAction* gets replicated
UFUNCTION(BlueprintNativeEvent) void OnRep_LastSuccessfulAction()		Blueprint function that runs logic for when *LastSuccessfulAction* gets replicated
virtual void OnRep_CurrentAction_Implementation()		Code parent of *OnRep_CurrentAction* that handles logic for when *CurrentAction* gets replicated
virtual void OnRep_LastSuccessfulAction_Implementation()		Code parent of *OnRep_LastSuccessfulAction* that handles logic for when *LastSuccessfulAction* gets replicated
Debugging		
virtual void DescribeSelfToGameplayDebugger(FGameplayDebuggerCategory DebuggerCategory) const*	*DebuggerCategory* - The instance of the *FGameplayDebuggerCategory_Utility* currently being used in the Gameplay Debugger Menu	Adds relevant debug text and debug shapes to the Gameplay Debugger for it to visualise in the Gameplay Debugger Menu
Delegate binding functions		
virtual void	*Action* - The Action that	Tracks the Actions

OnActionCompleted(UABUtilityActionBase Action)*	triggered this instance of the callback	completion status through either logging or updating *LastSuccessfulAction*, resets the Action and resets *CurrentAction*, then if the Action was successful it runs a timer for the length of the Actions cooldown value, otherwise it will set a timer for the next tick. The timer triggers a new Action once complete
void OnActionRequirementsChecked(UABUtilityActionBase Action, const bool bRequirementsPassed)*	*Action* - The Action that triggered this instance of the callback *bRequirementsPassed* - Whether or not all of the Actions Requirements passed and therefore if the Action is possible right now	If the Action passed the Requirement checks, the score is calculated. Once all scores have been calculated, if there are possible Actions, they are sorted by score and then iterated through in a similar manner to *RequestUtilityAction*
Action Logic Functions		
bool PerformUtilityAction()		Attempts to run an Action that is found either synchronously or asynchronously depending on the value returned by *IsComponentRunningAsynchronously*. Returns the success status of the attempt
bool RequestUtilityAction()		Synchronous version. Requests the possible Actions, sorts them by score, then tries to run one depending on the Action Selection Mode. If it is set to Highest Score, the component will iterate through the Actions in order until one is successful. If it is set to Random From Top 10

		Percent, the top 10% of Actions are grabbed and randomly shuffled, then iterated through until one is successful. If it is set to Random From Top 25 Percent, it is the same as before except the component grabs the top 25% of Actions. Returns the success status of the attempt
void AsyncRequestUtilityAction()		Asynchronously requests the possible Actions
void GetPossibleActions(TMap<UABUtilityActionBase, const float>& PossibleActionsToScores)*	*PossibleActionsToScores* - A map with the key as the Action and the value as the score for that Action	Gets the possible Actions first by checking the Actions Requirements and filtering out any that do not pass *CheckAllRequirements*, then by getting the score for the Action and filtering out any that have a score less than or equal to 0
void ParallelGetPossibleActions(TMap<UABUtilityActionBase, const float>& PossibleActionsToScores)*	*PossibleActionsToScores* - A map with they key as the Action and the value as the score for that Action	Uses the engines *ParallelFor* helper to help spread the Action Requirement and score checks across threads
void AsyncGetPossibleActions()		Sets up asynchronous tasks for the Actions to have their Requirements checked with *AsyncCheckAllRequirements*
void ParallelAsyncGetPossibleActions()		Uses the engines *ParallelFor* helper to spread the *AsyncTask* helper calls across threads
Retry logic		
bool AttemptRetry()		Attempts to use the timer

		manager to trigger *PerformUtilityAction()* on the next tick

Members:

Member	**Use**
Data assets	
UPROPERTY(EditAnywhere, Category = "Utility") *TObjectPtr<UABUtilityDataAsset> UtilityDataAsset*	Allows the user to specify which Utility Data Asset will be used by this Utility Component
Replicated properties	
UPROPERTY(ReplicatedUsing = OnRep_CurrentAction) *TObjectPtr<UABUtilityActionBase> CurrentAction*	Stores the current Action being run by the AI. It is a replicated member, with replication logic running in *OnRep_CurrentAction*
UPROPERTY(ReplicatedUsing = OnRep_LastSuccessfulAction) *TObjectPtr<UABUtilityActionBase> LastSuccessfulAction*	Stores the most recent successful Action run by the AI. It is a replicated member, with replication logic running in *OnRep_LastSuccessfulAction*
Actions	
TArray<TObjectPtr<UABUtilityActionBase>> UtilityActions	Local cache for the available Actions
EABActionSelectionMode ActionSelectionMode	Cached Action selection mode grabbed from the Utility Data Asset, determines whether the component iterates through all Actions from the highest score down, randomly iterates through the top 10% of Actions, or randomly iterates through the top 25% of actions
Inlines	
UPROPERTY(EditAnywhere, Category = "Pausing") *uint32 bRestartLogicWhenResumingComponentLogic : 1*	Allows the user to define whether or not the component will restart logic when *ResumeLogic* is called

UPROPERTY(EditAnywhere, Category = "Async") *uint32 bRunComponentAsynchronously : 1*	Allows the user to define whether or not the component runs its async logic (*AsyncRequestUtilityAction* instead of *RequestUtilityAction*, and either *AsyncGetPossibleActions* or *ParallelAsyncGetPossibleActions* instead of *GetPossibleActions* or *ParallelGetPossibleActions*)
UPROPERTY(EditAnywhere, Category = "Async") *uint32 bUseParallelRequirementChecking : 1*	Allows the user to define whether or not the component uses standard *for* iterations or the *ParallelFor* helper (*ParallelGetPossibleActions* instead of *GetPossibleActions*, and *ParallelAsyncGetPossibleActions* instead of *AsyncGetPossibleActions*)
UPROPERTY(EditAnywhere) *uint32 bRetryOnFailure : 1*	Allows the user to define whether or not the component attempts to retry running an Action if it fails
Async functionality	
FAJBUtilityComponentAsyncData AsyncData	The data stored for async operations
Timers	
FTimerHandle SuccessfulActionCooldownHandle	The handle used by the Timer Manager when running the cooldown after a successful Action

Other:

Types	
struct FAJBUtilityComponentAsyncData	The protected struct used to hold the data required for async operations, including the number of Actions to be checked (*int32*), the number of Actions that have been checked (*int32*), a map with the keys as Actions and the values as the scores for those Actions (*TMap<UABUtilityActionBase*, const float>*), and a *Reset()* function that sets the Action numbers back to 0 and empties the map
enum class	Defines the different methods available

EABUtilityComponentPauseActionLogic	for an Action when the component is paused. Contains *PauseAction* and *AbortAction*

UABUtilityActionBase

This is the base class for all Actions made both in this plugin and by others who may use this plugin. It contains Requirements for checking if it can run at all, Factors for calculating the score for the Action, and Tick functionality for any Actions that require running over multiple frames. Once the Action completes, it will trigger a delegate broadcast that the Utility Component should be tied to that will allow the component to find and run a new Action.

Parent class: *UObject*

Functions:

Function	**Params**	**Logic**
UABUtilityActionBase()		Used to turn Tick off for now, but set up the Action so it can Tick if required
Delegate binding and unbinding functions		
void BindToRequirementDelegates() const		Binds the Action to all of its Requirements *OnAsyncRequirementCheckedDelegate* for async functionality
void UnbindFromRequirementDelegates() const		Unbinds the Action from all of its Requirements *OnAsyncRequirementCheckedDelegate*
World operations		
void CacheWorld(UWorld InWorld)*	*InWorld* - The UWorld to be cached within this Action	Caches the world for future use
UWorld GetCachedWorld()*		Returns a pointer to the cached world
Action Requirement Checking		
const bool CheckAllRequirements(co	*OwnerComp* - The owning Utility Component	Runs one of two internal Requirement checking

nst UABUtilityComponent& OwnerComp) const		functions depending on the return value of *IsUsingParallelRequirementChecking()*. Returns the return value of the chosen function
void AsyncCheckAllRequirements(const UABUtilityComponent& OwnerComp)	*OwnerComp* - The owning Utility Component	Runs one of two internal async Requirement checking functions depending on the return value of *IsUsingParallelRequirementChecking()*
Getters		
const TArray<UABUtilityRequirementBase>& GetRequirements() const*		Returns a reference to the list of Requirement pointers held by this Action
const TArray<UABUtilityFactorBase>& GetFactors() const*		Returns a reference to the list of Factor pointers held by this Action
const float GetOnSuccessCooldownValue(UABUtilityComponent& OwnerComp) const	*OwnerComp* - The owning Utility Component	Returns the value in seconds for the success cooldown of this Action
const float GetUtilityWeightValue(const UABUtilityComponent& OwnerComp) const	*OwnerComp* - The owning Utility Component	Returns the value for the Utility Weight of this Action
Action scoring		
const float GetActionScore(const UABUtilityComponent& OwnerComp, const bool bUseConsiderationFactor) const	*OwnerComp* - The owning Utility Component *bUseConsiderationFactor* - Whether or not the scoring will be modified by the Consideration Factor	Returns the score for the Action
Action logic		
const	*OwnerComp* - The	Runs the Action, then

EABUtilityActionRunStatus& PerformAction(UABUtilityComponent& OwnerComp)	owning Utility Component	either completes the Action or, if the status is *Running*, enables the Tick for the Action. Returns the status of the Action after the initial *DoAction()* call
Run status		
const EABUtilityActionRunStatus& GetCurrentRunStatus() const		Returns the current status of the Action
void ResetCurrentRunStatus()		Resets the Actions status back to *NotRunning*
Pause, Resume and Abort logic		
virtual void PauseAction(UABUtilityComponent OwnerComp, const FString& Reason)*	*OwnerComp* - The owning Utility Component *Reason* - The reason the Action is being paused	Logs the *Reason* and disables the Tick
virtual bool ResumeAction(UABUtilityComponent OwnerComp)*	*OwnerComp* - The owning Utility Component	Logs the Action resuming and enables the Tick. Returns the success status of resuming the Action
virtual void AbortAction(UABUtilityComponent OwnerComp, const FString& Reason)*	*OwnerComp* - The owning Utility Component *Reason* - The reason the Action is being aborted	Logs the *Reason*, resets the Actions status then completes the Action
Tick logic		
virtual void TickAction(float DeltaTime, enum ELevelTick TickType, FABUtilityActionTickFunction& ThisTickFunction)	*DeltaTime* - The Tick time for this frame *TickType* - The levels Tick type *ThisTickFunction* - The Tick class for Actions	Ticks the Action then completes it if the status is no longer *Running*
Inlines		

const bool IsUsingParallelRequirementChecking() const		Returns the current status of *bUseParallelRequirementChecking*
const bool IsConsideringFactorScoreZero() const		Returns the current status of *bConsiderFactorScoreZero*
Validation		
virtual const bool ValidateAction(FString& InvalidationReason) const	*InvalidationReason* - The string to be printed if anything about the Action is invalid	An editor-time validation function that allows the user to ensure everything about the Action is valid when the user saves the Utility Data Asset
Debugging		
virtual void DescribeSelfToGameplayDebugger(const UABUtilityComponent& OwnerComp, const bool bUseConsiderationFactor, FGameplayDebuggerCategory DebuggerCategory) const*	*OwnerComp* - The owning Utility Component *bUseConsiderationFactor* - Whether or not the score will be modified by the consideration factor *DebuggerCategory* - The instance of the *FGameplayDebuggerCategory_Utility* currently being used in the Gameplay Debugger Menu	Adds relevant debug text and debug shapes to the Gameplay Debugger for it to visualise in the Gameplay Debugger Menu
Internal Action logic		
virtual EABUtilityActionRunStatus DoAction(UABUtilityComponent& OwnerComp)	*OwnerComp* - The owning Utility Component	Performs the Actions main logic. Returns the resulting status for the Action
virtual void OnActionCompleted()		Disables the Tick functionality and broadcasts a completion notification
Internal Tick logic		

virtual void Tick(UABUtilityComponent& OwnerComp, float DeltaSeconds)	*OwnerComp* - The owning Utility Component *DeltaSeconds* - The Tick time for this frame	Performs the Actions Tick logic
Internal Action Requirement checking		
const bool InternalCheckAllRequirements(const UABUtilityComponent& OwnerComp) const	*OwnerComp* - The owning Utility Component	Iterates through all of the Actions Requirements until all have passed or until one fails. Returns the success status of the Requirements. Returns the success status of all Requirements
const bool InternalParallelCheckAllRequirements(const UABUtilityComponent& OwnerComp) const	*OwnerComp* - The owning Utility Component	Uses the engines *ParallelFor* helper to spread the Requirement checks across threads. Returns the success status of all Requirements
void InternalAsyncCheckAllRequirements(const UABUtilityComponent& OwnerComp)	*OwnerComp* - The owning Utility Component	Sets up asynchronous tasks for the Requirements to be checked using *AsyncCheckRequirement*
void InternalParallelAsyncCheckAllRequirements(const UABUtilityComponent& OwnerComp)	*OwnerComp* - The owning Utility Component	Uses the engines *ParallelFor* helper to spread the *AsyncTask* calls across threads
virtual void OnAsyncRequirementChecked(const UABUtilityRequirementBase const Requirement, const bool bRequirementPassed)*	*Requirement* - The Requirement that triggered this instance of the callback *bRequirementPassed* - Whether or not the Requirement passed	Keeps track of the async Requirement pass status. Once all Requirements have returned their pass status, sets up an async task for the game thread to broadcast the success status of Requirements check
Internal Action scoring		
void ModifyTotalScore(const float& ModifyingScore,	*ModifyingScore* - The score to modify the total by	Modifies the current total score by the current Factors score using the

const float& NumFactors, const int32& CurrentIndex, float& OutTotalScore) const	*NumFactors* - The number of Factors held by the Action as a *float* *CurrentIndex* - The index of the current Factor being calculated *OutTotalScore* - The score after the modifier has been applied	user-set score totalling method
Other		
virtual const bool ShouldTickIfViewportsOnly() const		Returns the current status of *bShouldTickIfViewportsOnly*

Members:

Member	**Use**
Delegate broadcasts	
FOnActionCompletedDelegate OnActionCompletedDelegate	The delegate used to broadcast the Action completed notification
FOnAsyncRequirementsChecked OnAsyncRequirementsCheckedDelegate	The delegate used to broadcast the async Requirements checked notification
Tick function	
UPROPERTY(EditDefaultsOnly, Category = Tick) *struct FABUtilityActionTickFunction PrimaryUtilityActionTick*	The Tick function used by Actions
Cached members	
EABUtilityActionRunStatus CurrentRunStatus	The current status of the Action
TWeakObjectPtr<UABUtilityComponent > OwnerComponent	The cached instance of the owning Utility Component, used in the Tick as the Component doesn't trigger Tick functionality in any Action

TObjectPtr<UWorld> CachedWorld	The cached instance of the world
Async functionality	
FAJBUtilityActionAsyncData AsyncData	The required data for the Actions async functionality
Data providers	
UPROPERTY(EditAnywhere, meta = (ClampMin = 0.0, UIMin = 0.0)) FAIDataProviderFloatValue UtilityWeight	Float Data Provider that defines the weight for this Action
UPROPERTY(EditAnywhere, meta = (ClampMin = 0.0, UIMin = 0.0)) FAIDataProviderFloatValue OnSuccessCooldown	Float Data Provider that defines the cooldown time for this Action should it be successful
Scoring	
UPROPERTY(EditAnywhere) EABFactorScoreTotallingMethod ScoreAggregationMethod	The method used to total the Actions Factors scores. *Sum* adds the scores together. *Multiply* multiplies the scores together. *Average* calculates the mean value of the scores once they have all been calculated. *Min* uses the Actions lowest scoring Factor. *Max* uses the Actions highest scoring Factor.
Requirements and Factors	
UPROPERTY(EditAnywhere, Instanced) TArray<TObjectPtr<UABUtilityRequirementBase>> Requirements	The list of Requirements set by the user that must pass for the Action to be able to run
UPROPERTY(EditAnywhere, Instanced) TArray<TObjectPtr<UABUtilityFactorBase>> Factors	The list of Factors set by the user that are used to calculate the score for the Action
Inlines	
UPROPERTY(EditAnywhere) uint32 bUseParallelRequirementChecking : 1	Defines whether or not the Action uses the engines *ParallelFor* helper when checking Requirements (*InternalParallelCheckAllRequirements* instead of *InternalCheckAllRequirements*, and *InternalParallelAsyncCheckAllRequirem*

	ents instead of *InternalAsyncCheckAllRequirements*)
UPROPERTY(EditAnywhere) uint32 bConsiderFactorScoreZero : 1	Defines whether or not the Action will still consider a Factor if the score comes back as 0
Other	
UPROPERTY(EditAnywhere) bool bShouldTickIfViewportsOnly	Sets whether the Tick runs if the Levels Tick type is *LEVELTICK_ViewportsOnly*

Other:

Types	
struct FAJBUtilityActionAsyncData	The data required for the Actions sync functionality, including the number of Requirements to be checked (*int32*), the number of Requirements that have already been checked (*int32*), whether or not all Requirements have been passed (*bool*), and a *Reset()* function that resets both numbers back to 0 and the Boolean back to *true*.
enum class EABFactorScoreTotallingMethod	Defines the different available methods of totalling the Actions scores
struct FABUtilityActionTickFunction : public FTickFunction	The Tick function for Actions

UABUtilityRequirementBase

Requirements are used to check if an Action is able to run at that moment. When the Utility Component tries to run a new Action, it checks the Requirements of that Action first. They must all pass for the Action to be possible when checked. The only case where not every Requirement needs to pass is if the user adds a Multi-Requirement to the Action that groups with an OR operator instead of an AND operator.

Parent class: *UObject*

Functions:

Function	**Params**	**Logic**
Requirement checking		
virtual const bool CheckRequirement(const UABUtilityComponent& OwnerComp) const	*OwnerComp* - The owning Utility Component	Runs the main logic for the Requirement. Returns whether or not the Requirement passes
void AsyncCheckRequirement(const UABUtilityComponent& OwnerComp) const	*OwnerComp* - The owning Utility Component	Is run by the Actions async logic, runs *CheckRequirement* and then sets up an async task to broadcast the result on the game thread
Validation		
virtual const bool ValidateRequirement(FString& InvalidationReason) const	*InvalidationReason* - The string to be printed if anything about the Requirement is invalid	An editor-time validation function that allows the user to ensure everything about the Requirement is valid when the user saves the Utility Data Asset
Debugging		
virtual void DescribeSelfToGameplay Debugger(const UABUtilityComponent&	*OwnerComp* - The owning Utility Component *DebuggerCategory* - The	Adds relevant debug text and debug shapes to the Gameplay Debugger for it to visualise in the

OwnerComp, FGameplayDebuggerCategory DebuggerCategory) const*	instance of the *FGameplayDebuggerCategory_Utility* currently being used in the Gameplay Debugger Menu	Gameplay Debugger Menu

Members:

Member	**Use**
Delegate broadcasts	
FOnAsyncRequirementChecked OnAsyncRequirementCheckedDelegate	The delegate that broadcasts the notification when the Requirement has finished being checked asynchronously

UABUtilityFactorBase

Factors are used to calculate the score for an Action when the Utility Component has already checked its Requirements and is considering it as a possibility. Each Factor has its own score that is evaluated when the Action is having its score calculated, then all of an Actions scores are totalled using a user-set method. If a Factors score returns 0, it will usually be ignored unless the user has enabled *bConsiderFactorScoreZero* on the owning Action. The Factor has an optional extra bit of functionality called the “Consideration Factor”, which is inspired by and described well in this video: [GDC Vault - Building a Better Centaur: AI at Massive Scale]

Parent class: *UObject*

Functions:

Function	**Params**	**Logic**
	Scoring	
const float GetFactorScore(const UABUtilityComponent& OwnerComp, const float NumberOfFactors, const bool bUseConsiderationFactor) const	*OwnerComp* - The owning Utility Component *NumberOfFactors* - The number of Factors owned by the owning Action, used by the Consideration Factor *bUseConsiderationFactor* - Whether or not the score will be modified by the Consideration Factor	Retrieves the main Factor score from *CalculateFactorScore*, inverses it if *bInverseScore* is enabled, runs the score through the Consideration Factor if *bUseConsiderationFactor* is enabled, and returns the final score for this Factor
	Inlines	
const bool IsInversed() const		Returns the current status of *bIsInversed*
	Validation	

virtual const bool ValidateFactor(FString& InvalidationReasons) const	*InvalidationReason* - The string to be printed if anything about the Factor is invalid	An editor-time validation function that allows the user to ensure everything about the Factor is valid when the user saves the Utility Data Asset
Debugging		
virtual void DescribeSelfToGameplayDebugger(const UABUtilityComponent& OwnerComp, const float NumberOfFactors, const bool bUseConsiderationFactor, FGameplayDebuggerCategory DebuggerCategory) const*	*OwnerComp* - The owning Utility Component *NumberOfFactors* - The number of Factors owned by the owning Action, is used by the Consideration Factor *bUseConsiderationFactor* - Whether or not the score will be modified by the Consideration Factor *DebuggerCategory* - The instance of the *FGameplayDebuggerCategory_Utility* currently being used in the Gameplay Debugger	Adds relevant debug text and debug shapes to the Gameplay Debugger for it to visualise in the Gameplay Debugger Menu
Internal scoring		
virtual const float CalculateFactorScore(const UABUtilityComponent& OwnerComp) const	*OwnerComp* - The owning Utility Component	Calculates the main Factor score
void AddConsiderationFactor(const float NumberOfFactors, float& OutFloat) const	*NumberOfFactors* - The number of Factors owned by the owning Action *OutFloat* - The Factor score to be used in the calculation and set at the end of the logic	Runs the current Factor score through the Consideration Factor described in the GDC video

Members:

Member	Use
Inlines	
UPROPERTY(EditAnywhere, Category = "Scoring") *uint32 bInverseScore : 1*	Defines whether or not the Factor score is inversed (1 - FactorScore)

UABUtilityActionBlueprintBase

This Action is the base for any Actions that users wish to make in Blueprints instead of C++. It overrides the relevant functions from the Action base, and adds Blueprint functions that allow users to validate, debug, run and Tick an Action.

Parent class: *UABUtilityActionBase*

Functions:

Function	**Params**	**Logic**
Validation		
const bool ValidateAction(FString& InvalidationReason) const override	*InvalidationReason* - The string to be printed if anything about the Action is invalid	Runs the parents *ValidateAction()* and the Blueprint *ValidateBlueprintAction()*, and returns the result of both together with an AND operator
UFUNCTION(BlueprintNativeEvent) *const bool ValidateBlueprintAction(const FString& InInvalidationReason, FString& OutInvalidationReason) const*	*InInvalidationReason* - The string that already exists to be printed *OutInvalidationReason* - The string to be printed if anything about the Action is invalid, should contain the string passed into *InInvalidationReason*	To be overridden in the Blueprint allowing users to run editor-time validation on any Blueprint members in this Action
virtual const bool ValidateBlueprintAction_Implementation(const FString& InInvalidationReason, FString& OutInvalidationReason) const	*InInvalidationReason* - The string that already exists to be printed *OutInvalidationReason* - The string to be printed if anything about the Action is invalid, should contain the string passed into *InInvalidationReason*	The C++ parent of *ValidateBlueprintAction()* for validating any members for this Action that exist in the code
Debugging		
void	*OwnerComp* - The	Runs the parents

DescribeSelfToGameplayDebugger(const UABUtilityComponent& OwnerComp, const bool bUseConsiderationFactor, FGameplayDebuggerCategory DebuggerCategory) const override*	owning Utility Component *bUseConsiderationFactor* - Whether or not the score will be modified by the Consideration Factor *DebuggerCategory* - The instance of the *FGameplayDebuggerCategory_Utility* currently being used in the Gameplay Debugger Menu	*DescribeSelfToGameplayDebugger()*, then adds a text line to the *DebuggerCategory* containing the return value of *DescribeBlueprintSelfToGameplayDebugger()*
UFUNCTION(BlueprintNativeEvent) *const FString DescribeBlueprintSelfToGameplayDebugger(const UABUtilityComponent* const OwnerComp) const*	*OwnerComp* - The owning Utility Component	To be overridden in the Blueprint allowing users to debug any member values and function outcomes that exist in the Blueprint of this Action in the Gameplay Debugger Menu. Returns the debug string to be used
virtual const FString DescribeBlueprintSelfToGameplayDebugger_Implementation(const UABUtilityComponent const OwnerComp) const*	*OwnerComp* - The owning Utility Component	The C++ parent of *DescribeBlueprintSelfToGameplayDebugger()* for debugging any member values and function outcomes that exist in the code
Internal Action logic		
EABUtilityActionRunStatus DoAction(UABUtilityComponent& OwnerComp) override	*OwnerComp* - The owning Utility Component	Sets the current Action status to *Running*, runs the Blueprint *DoBlueprintAction()*, and returns the Action status after that call (*DoBlueprintAction()* may change the Action status)
UFUNCTION(BlueprintNativeEvent) *void DoBlueprintAction(UABUtilityComponent* OwnerComp)*	*OwnerComp* - The owning Utility Component	To be overridden in the Blueprint allowing users to define the Action logic in Blueprint instead of in code

virtual void DoBlueprintAction_Implementation(UABUtilityComponent OwnerComp)*	*OwnerComp* - The owning Utility Component	The C++ parent of *DoBlueprintAction()* for running any Action logic that should exist in code
Internal Tick logic		
void Tick(UABUtilityComponent& OwnerComp, float DeltaSeconds) override	*OwnerComp* - The owning Utility Component *DeltaSeconds* - The Tick time in seconds for this frame	Runs the Blueprint *TickBlueprintAction()*
UFUNCTION(BlueprintNativeEvent) *void TickBlueprintAction(UABUtilityComponent* OwnerComp, float DeltaSeconds)*	*OwnerComp* - The owning Utility Component *DeltaSeconds* - The Tick time in seconds for this frame	To be overridden in the Blueprint allowing users to define the Action Tick logic in Blueprint instead of code
virtual void TickBlueprintAction_Implementation(UABUtilityComponent OwnerComp, float DeltaSeconds)*	*OwnerComp* - The owning Utility Component	The C++ parent of *TickBlueprintAction()* for running any Action Tick logic that should exist in code
Completion function		
UFUNCTION(BlueprintCallable) *void FinishAction(bool bSuccess)*	*bSuccess* - Whether or not the Blueprint Action was a success or failure	Sets the Action status to either *Succeeded* or *Failed* depending on the value of *bSuccess*

UABUtilityRequirementBlueprintBase

This Requirement is the base for any Requirements that users wish to make in Blueprint instead of C++. It overrides the relevant functions from the Requirement base, and adds Blueprint functions that allow users to validate, debug and check the Requirement.

Parent class: *UABUtilityRequirementBase*

Functions:

Function	**Params**	**Logic**
Requirement checking		
const bool CheckRequirement(const UABUtilityComponent& OwnerComp) const override	*OwnerComp* - The owning Utility Component	Runs and returns the return value of *CheckBlueprintRequirement()*
Validation		
const bool ValidateRequirement(FString& InvalidationReason) const override	*InvalidationReason* - The string to be printed if anything about the Requirement is invalid	Runs the parents *ValidateRequirement()* and the Blueprint *ValidateBlueprintRequirement()*, and returns the result of both together with an AND operator
const bool ValidateBlueprintRequirement(const FString& InInvalidationReason, FString& OutInvalidationReason) const	*InInvalidationReason* - The string that already exists to be printed *OutInvalidationReason* - The string to be printed if anything about the Requirement is invalid, should contain the string passed into *InInvalidationReason*	To be overridden in the Blueprint allowing users to run editor-time validation on any Blueprint members in this Requirement
virtual const bool ValidateBlueprintRequirement_Implementation(con	*InInvalidationReason* - The string that already exists to be printed	The C++ parent of *ValidateBlueprintRequirement()* for validating any

st FString& InInvalidationReason, FString& OutInValidationReason) const	*OutInvalidationReason* - The string to be printed if anything about the Requirement is invalid, should contain the string passed into *InInvalidationReason*	members for this Requirement that exist in the code
Debugging		
void DescribeSelfToGameplay Debugger(const UABUtilityComponent& OwnerComp, FGameplayDebuggerCat egory DebuggerCategory) const override*	*OwnerComp* - The owning Utility Component *DebuggerCategory* - The instance of the *FGameplayDebuggerCat egory_Utility* currently being used in the Gameplay Debugger Menu	Runs the parents *DescribeSelfToGameplay Debugger()*, then adds a text line to the *DebuggerCategory* containing the return value of *DescribeBlueprintSelfToG ameplayDebugger()*
UFUNCTION(BlueprintNa tiveEvent) const FString DescribeBlueprintSelfToG ameplayDebugger(const UABUtilityComponent const OwnerComp) const*	*OwnerComp* - The owning Utility Component	To be overridden in the Blueprint allowing users to debug any member values and function outcomes that exist in the Blueprint of this Requirement in the Gameplay Debugger Menu. Returns the debug string to be used
virtual const FString DescribeBlueprintSelfToG ameplayDebugger_Imple mentation(const UABUtilityComponent const OwnerComp) const*	*OwnerComp* - The owning Utility Component	The C++ parent of *DescribeBlueprintSelfToG ameplayDebugger()* for debugging any member values and function outcomes that exist in the code
Internal requirement checking		
UFUNCTION(BlueprintNa tiveEvent) const bool CheckBlueprintRequirem ent(const UABUtilityComponent const OwnerComp) const*	*OwnerComp* - The owning Utility Component	To be overridden in the Blueprint allowing users to define the Requirement logic in Blueprint instead of code

virtual const bool CheckBlueprintRequirement_Implementation(const UABUtilityComponent const OwnerComp) const*	*OwnerComp* - The owning Utility Component	The C++ parent of *CheckBlueprintRequirement()* for running any Requirement checking logic that should be run in code

UABUtilityFactorBlueprintBase

This Factor is the base for any Factors that users wish to make in Blueprint instead of C++. It overrides the relevant functions from the Factor base, and adds Blueprint functions that allow users to validate, debug and calculate the Factor score.

Parent class: *UABUtilityFactorBase*

Functions:

Function	Params	Logic
Validation		
const bool ValidateFactor(FString& InvalidationReasons) const override	*InvalidationReason* - The string to be printed if anything about the Factor is invalid	Runs the parents *ValidateFactor()* and the Blueprint *ValidateBlueprintFactor()*, and returns the result of both together with an AND operator
UFUNCTION(BlueprintNativeEvent) const bool ValidateBlueprintFactor(const FString& InInvalidationReason, FString& OutInvalidationReason) const	*InInvalidationReason* - The string that already exists to be printed *OutInvalidationReason* - The string to be printed if anything about the Factor is invalid, should contain the string passed into *InInvalidationReason*	To be overridden in the Blueprint allowing users to run editor-time validation on any Blueprint members in this Factor
virtual const bool ValidateBlueprintFactor_Implementation(const FString& InInvalidationReason, FString& OutInvalidationReason) const	*InInvalidationReason* - The string that already exists to be printed *OutInvalidationReason* - The string to be printed if anything about the Factor is invalid, should contain the string passed into *InInvalidationReason*	The C++ parent of *ValidateBlueprintFactor()* for validating any members for this Factor that exist in the code
Debugging		
void	*OwnerComp* - The	Runs the parents

DescribeSelfToGameplayDebugger(const UABUtilityComponent& OwnerComp, const float NumberOfFactors, const bool bUseConsiderationFactor, FGameplayDebuggerCategory DebuggerCategory) const override*	owning Utility Component *NumberOfFactors* - The number of Factors owned by the owning Action *bUseConsiderationFactor* - Whether or not the score will be modified by the Consideration Factor *DebuggerCategory* - The instance of the *FGameplayDebuggerCategory_Utility* currently being used in the Gameplay Debugger Menu	*DescribeSelfToGameplayDebugger()*, then adds a text line to the *DebuggerCategory* containing the return value of *DescribeBlueprintSelfToGameplayDebugger()*
UFUNCTION(BlueprintNativeEvent) *const FString DescribeBlueprintSelfToGameplayDebugger(const UABUtilityComponent* const OwnerComp, const float NumberOfFactors) const*	*OwnerComp* - The owning Utility Component *NumberOfFactors* - The number of Factors owned by the owning Action	To be overridden in the Blueprint allowing users to debug any member values and function outcomes that exist in the Blueprint of this Factor in the Gameplay Debugger Menu. Returns the debug string to be used
virtual const FString DescribeBlueprintSelfToGameplayDebugger_Implementation(const UABUtilityComponent const OwnerComp, const float NumberOfFactors) const*	*OwnerComp* - The owning Utility Component *NumberOfFactors* - The number of Factors owned by the owning Action	The C++ parent of *DescribeBlueprintSelfToGameplayDebugger()* for debugging any member values and function outcomes that exist in the code
Internal scoring		
const float CalculateFactorScore(const UABUtilityComponent& OwnerComp) const override	*OwnerComp* - The owning Utility Component	Runs and returns the return value of *CalculateBlueprintFactorScore()*
UFUNCTION(BlueprintNativeEvent) *const float CalculateBlueprintFactorS*	*OwnerComp* - The owning Utility Component	To be overridden in the Blueprint allowing users to create Blueprint logic to calculate the Factor score

core(const UABUtilityComponent const OwnerComp) const*		
virtual const float CalculateBlueprintFactorScore_Implementation(const UABUtilityComponent const OwnerComp) const*	*OwnerComp* - The owning Utility Component	The C++ parent of *CalculateBlueprintFactorScore()* for running any score calculations that should be run in code

UABUtilityDataAsset

This is the data asset used to define the Actions available to an AI. It allows users to define a list of Actions and fill out the data for those Actions, as well as define lists of Requirements and Factors for those Actions and fill out the data for those as well.

Parent class: *UDataAsset*

Functions:

Function	Params	Logic
Actions		
const TArray<TObjectPtr<UABUtilityActionBase>>& GetActions() const		Returns the *const* list of Action pointers defined in the asset by the user
const EABActionSelectionMode & GetActionSelectionMode() const		Returns a *const* reference to the Action selection mode chosen by the user
Scoring		
const bool IsUsingConsiderationFactor() const		Returns the current status of *bUseConsiderationFactor*

Members:

Member	Use
Actions	
UPROPERTY(EditAnywhere) EABActionSelectionMode ActionSelectionMode	The method used by the component to iterate through the available Actions
UPROPERTY(EditAnywhere, Instanced) TArray<TObjectPtr<UABUtilityActionBase>> UtilityActions	The list of instanced Actions as pointers defined by the user in the asset

Scoring	
UPROPERTY(EditAnywhere, Category = "Scoring") *bool bUseConsiderationFactor*	Whether or not the score of Factors within this asset will be modified by the Consideration Factor

Other:

Types	
enum class EABActionSelectionMode	Defines the possible methods for selecting Actions

FGameplayDebuggerCategory_Utility

The custom Gameplay Debugger Category used for debugging the Utility Component, Actions, Requirements and Factors.

Parent class: *FGameplayDebuggerCategory*

Functions:

Function	**Params**	**Logic**
Debugging		
FGameplayDebuggerCategory_Utility()		Empty
void CollectData(APlayerController OwnerPC, AActor* DebugActor) override*	*OwnerPC* - The player controller for the user that enabled the Gameplay Debugger Menu *DebugActor* - The actor in the world that is being debugged and having its information displayed in the Gameplay Debugger Menu	Casts the *DebugActor* to a Pawn, gets its AI Controller, gets that controllers Brain Component, casts that Brain Component to a Utility Component, then calls that components *DescribeSelfToGameplayDebugger()*
static TSharedRef<FGameplayDebuggerCategory> MakeInstance()		Creates a new shareable reference to an instance of *FGameplayDebuggerCategory_Utility*

UABUtilityDataAssetValidator

The editor validation class that validates the Utility Data Asset that has just been saved.

Parent class: *UEditorValidatorBase*

Functions:

Function	Params	Logic
Validation		
bool CanValidateAsset_Implementation(const FAssetData& InAssetData, UObject InObject, FDataValidationContext& InContext) const override*	*InAssetData* - Holds important information about the asset found by the Asset Registry *InObject* - The asset that is being validated *Context* - The interface between *UObject::IsDataValid* and the data validation system	Returns whether or not the *InObject* is of the class or is a child of the class *UABUtilityDataAsset*. This determines whether or not *ValidateLoadedAsset()* will run
EDataValidationResult ValidateLoadedAsset_Implementation(const FAssetData& InAssetData, UObject InAsset, FDataValidationContext& Context) override*	*InAssetData* - Holds important information about the asset found by the Asset Registry *InAsset* - The asset that is being validated *Context* - The interface between *UObject::IsDataValid* and the data validation system	Ensures *CanValidateAsset()* passed correctly, builds up string arrays containing the invalidation reasons for each Action, Requirement and Factor, and if anything is invalid, concatenates these into a single error message that is passed into *AssetFails()*

ABUtilityTypes.h

This is a helper file containing types that exist across multiple classes.

Type	**Use**
UENUM(BlueprintType) *enum class EABUtilityActionRunStatus*	Defines the states an Action can be in at any time, including *NotRunning*, *Running*, *Succeeded* and *Failed*
enum class EABUtilityAbortActionLogic	Defines the types of abort that can be used in the Utility Components *AbortCurrentAction*, including *StopComponent*, *PauseLogicPauseAction*, *PauseLogicAbortAction* and *ChooseNewAction*

ABUtilityHelpers.h

This is a helper file containing namespaces and classes that are used by multiple classes and/or structs.

Namespaces:

ABUtility::Helpers::Requirements		
Function	**Params**	**Logic**
bool CompareValues(const float Left, const float Right, const EGenericAICheck& ComparisonType)	*Left* - The first value to be compared *Right* - The second value to be compared *ComparisonType* - How the values should be compared	Compares the two values in different ways depending on *ComparisonType*

Examples

Component Setup

The first thing to set up is an AI Controller. Create a custom class that inherits from *AAIController*, expose a pointer to a *UABUtilityComponent* and in the constructor, create the component and set it as the *BrainComponent*.

.h file

```
UCLASS()
class AJIMBOBUTILITYAI_API ATestUtilAIController : public AAIController
{
    GENERATED_BODY()

public:
    ATestUtilAIController(const FObjectInitializer& ObjectInitializer);

private:
    UPROPERTY(EditAnywhere)
    TObjectPtr<UABUtilityComponent> UtilityComponent = nullptr;
};
```

.cpp file

```
ATestUtilAIController::ATestUtilAIController(const FObjectInitializer&
ObjectInitializer)
{
    bStartAILogicOnPossess = true;
    bStopAILogicOnUnposses = true;

    UtilityComponent =
CreateDefaultSubobject<UABUtilityComponent>(TEXT("UtilityComponent"));
    BrainComponent = UtilityComponent;
}
```

Create a Blueprint of this custom AI Controller and, in the *UtilityComponent*, find the *UtilityDataAsset*. This is where the data asset will go once created.

Create a new data asset of type *UABUtilityDataAsset*. Open and save it, and set up the component so that *UtilityDataAsset* is set to this new asset. Now the asset can be built up.

In the data asset, find the *ActionSelectionMode* dropdown and set this to whichever mode most suits the AI that will use this asset. Now it can be filled with Actions.

Code Action

To create an Action in code, go to the code files and create a new class that inherits from *UABUtilityActionBase*. Name it something that reflects what the Action should make the AI do.

Override *DoAction()*, this is the function where the main logic for the Action will exist. If the Action requires Ticking, also override *Tick()* as the ticked logic will exist here.

In the *.cpp* file for this class, build up these two functions so that the Action does what is desired. If the Action should have other custom logic, functions such as *PauseAction()*, *ResumeAction()*, and *AbortAction()* can also be overridden to add custom logic to these processes.

If exposed members are being defined in the class, make sure to override *ValidateAction()* to ensure that the member is always valid (this will do editor-time validation when the asset is saved). Also override *DescribeSelfToGameplayDebugger()* to ensure the value of the member can be shown in the Gameplay Debugger Menu for easier debugging.

Internal members and functions can be added freely to keep the Action code tidy and customisable to users.

Blueprint Action

Another option for creating an Action is to create one in Blueprint. To do this, create a new Blueprint class of type *UABUtilityActionBlueprintBase* and name it something that reflects the purpose of the Action.

In this Action, override the function *DoBlueprintAction()* to define main Action logic, *TickBlueprintAction()* to define any logic that needs to be ticked, and if validation or debugging is desired then the functions *ValidateBlueprintAction()* and *DescribeBlueprintSelfToGameplayDebugger()* can be overridden in the Blueprint to supply these.

In the Action logic, either as part of *DoBlueprintAction()* or *TickBlueprintAction()* if that has been overridden, ensure to add a call to *FinishAction()* and pass in whether the Action has succeeded or failed. **Without this call, the Action will never complete as the *CurrentRunStatus* in code will stay as *Running*.**

Action Setup in Data Asset

In the data asset, add the new Action into the list of Actions. Any custom members will appear under the name of the Action. Tick settings can be found under the *Tick* section. Base Action settings can be found under the *ABUtility Action Base* section. The most important of these is the *Utility Weight*, as if this is 0 the Action will never run. Set this to a value suitable compared to all Utility Weights for all Actions in this data asset.

The next thing to set up is the Requirements and Factors for the Action.

Code Requirement

To create a Requirement in code, go to the code files and create a new class that inherits from *UABUtilityRequirementBase*. Name it something that reflects what the Requirement is checking.

Override the *CheckRequirement()* function and set it up so that it checks the correct thing. Custom functions and members can be added to ensure code cleanliness and customisability, and members can be validated and debugged in the *ValidateRequirement()* and *DescribeSelfToGameplayDebugger()* function overrides respectively.

Blueprint Requirement

Requirements can also be made in Blueprints. Create a new Blueprint class of type *UABUtilityRequirementBlueprintBase* and, in the new Blueprint, override *CheckBlueprintRequirement()* and set up the logic to check for what is required. Custom members can be validated and debugged in the overridable functions *ValidateBlueprintRequirement()* and *DescribeBlueprintSelfToGameplayDebugger()*.

Multi-Requirement

There is an example Requirement that comes with the plugin that is a “Multi-Requirement”. This groups several Requirements together and supplies an operator, either AND or OR. This is a good way to group Requirements and allows for the OR operator if only one of a certain group of Requirements needs to be true for the Action to be possible.

Code Factor

To create a Factor in code, go to the code files and create a new class that inherits from *UABUtilityFactorBase*. Name it something that reflects what the Factor will score against.

Override the *CalculateFactorScore()* function and set it up so it calculates a normalised score. Custom functions and members can be added to ensure code cleanliness and customisability, and members can be validated and debugged in the *ValidateFactor()* and *DescribeSelfToGameplayDebugger()* function overrides respectively.

Blueprint Factor

Factors can also be made in Blueprints. Create a new Blueprint class of type *UABUtilityFactorBlueprintBase* and, in the new Blueprint, override *CalculateBlueprintFactorScore()* and set up the logic that will calculate and return the Factor score. Custom members can be validated and debugged in the overridable functions *ValidateBlueprintFactor()* and *DescribeBlueprintSelfToGameplayDebugger()*.

Requirement and Factor Setup in Data Asset

Back in the data asset, the Action added previously has two lists, one named Requirements and one named Factors. Add the newly created Requirement and Factor into these lists and set up the correct data for any exposed members.

Asynchronous Logic and Parallel For

The plugin comes with built-in asynchronous logic and usage of the engines *AsyncTask* and *ParallelFor* helpers. These are not used automatically, as they may not always help optimise the game due to the added overhead.

To enable the asynchronous logic, go to the component set up in the AI Controller Blueprint and enable *RunComponentAsynchronously*. This will use *AsyncTask* to check the Requirements for each Action before choosing an Action once complete.

To use the *ParallelFor* helper in the component, enable *UseParallelRequirementChecking*. This will swap the generic *for* loop for the *ParallelFor* helper and spread out the iterations across threads.

Each Action also has this option, so it can be enabled for individual Actions if that Action should benefit from it.

On the component, both *RunComponentAsynchronously* and *UseParallelRequirementChecking* can be enabled at the same time, but it is not required.

Ensure to measure performance both before and after enabling or disabling each of these, including after enabling or disabling *UseParallelRequirementChecking* on each Action individually to make sure the game is running as best as it can.

Validation

When saving the asset, all of the Actions and their Requirements and Factors will automatically be validated. The *UABUtilityDataAssetValidator* class makes sure it's validating a *UABUtilityDataAsset*, and then runs *ValidateAction()* on each of the data assets Actions, *ValidateRequirement()* on each of each Actions Requirements, and

ValidateFactor() on each of each Actions Factors. It is wise to override the relevant validation function on each Action, Requirement and Factor to ensure that each member of each class is valid at editor time.

Debugging

The Gameplay Debugger Menu can be enabled at runtime by pressing the apostrophe (‘) key. One of the categories that should come up will be labelled “UtilityAI”. Enabling this category will run *DescribeSelfToGameplayDebugger()* on the Utility Component of the debugged actor and all Actions, Requirements and Factors found within that component, displaying text describing the current values of class members and drawing any debug shapes relating to the current Action being performed. It is wise to override the debugging function on each Action, Requirement and Factor to ensure each member of each class can be debugged at runtime.

Comments

The plugin is built and maintained by Axela Brockett. Any comments regarding improvements, bugs found or anything else can be directed to her either in the GitHub repository comments or on LinkedIn at https://www.linkedin.com/in/axelajimbobbrockett/.