Utility AI Combat System Plugin

By Zachary Kolansky

Features

1. Weapon System, Replicated
2. Firing Component, Replicated
3. Montage Component, Replicated
4. Utility AI System
5. Mannequin with Range and Melee animations with an Anim BP. (Retargeted Paragon Greystone, and Paragon Lt. Belecia animations to the Mannequin)
6. Interfaces and Anim notify states for Combat.
7. Stat System, Replicated. Also has a status effect system.
8. Rudimentary Saving and Loading support for Weapons,Stats, and Status Effects.
9. Hit React System.

Purpose

The objective of this project was to provide a simple UtilityAI framework, because behavior trees scale poorly as AI behaviors become increasingly complex. Even though behavior trees support pallarism, it is limited because the tasks are not fully independent. Instead of sequence and selectors, we have task layers. Instead of stacking decorators, each task has a scoring function.
Here is the framework:

1. Each task has a Scoring function, flow options, Layer, Entry, and Exit and customizable in Blueprint
    a. Task layers determine parallelism, and which tasks can interrupt each other.
    b. Tasks on different layers operate independently, and there can be only one active task per layer.
    c. Flow options: Cooldown, Randomized Cooldown, do once, Locked
2. Everytime we want to Determine the best task, we do the following
    a. Calculate anything necessary to perform scoring functions. (Ex. Distance in unreal units to the focus)
    b. Score every task on a scale of 0-1 based on its scoring function.
    c. Change to the best tasks, comparing the BestTask TMap to the CurrentTasks TMap.
3. Changing tasks
    a. Determine if the current task either: ended, can be interrupted, or can be forced to end. Exit that task, and note the free layer.
    b. Tasks that cannot be ended are still the current task for a particular layer, and stored into NewTasks Map.
    c. For each free task layer, begin the best task and add the task layer association to the NewTasks Map.
    d. CurrentTasks Map = NewTasks Map

Many games have two variables to manage a stat: a current value and a max value. While this is good for smaller projects, this framework becomes unwieldy the more complicated the project becomes. Additionally, this per-variable framework does not work with data tables, which are an invaluable tool to balance players and weapons. An abstract stat system allows for flexible data tables, calculating any normalized value. (I.e current/Max), one stat changing the value of another stat, and equipment. This also been handled.

Moreover, many games require a weapon system, and we all have been reinventing the wheel. While Epic has provided a Shooter Game template, its weapon inventory system is integrated with its character, its firing system is tied to its gun, and it is not available as a plugin. This makes it difficult to use with existing projects, and abstract it into different genres. This plugin aims to fix that, and give everyone a place to start. Hence, the weapon system must

1. Be replicated
2. Not be tied to a specific C++ character class
3. Handle both melee and ranged
4. Can fire line traces, or projectiles.

All of this is done.

Installation and Plugin Versions

1. Marketplace version
2. Github link
    a. Meant to be a 100% reflection of the marketplace version. Go here if there is an Engine update that you need ASAP.
3. ALSCombine (Github link)
    a. I Combined this plugin with Community ALS so they are together.
    b. Community ALS
4. See Github ReadMe for installation from github guide.

Getting Started

1. Make Engine content visible
2. Open the Overview map in the Maps folder of this plugin
3. Open the *MannequinCharacter_BP*
4. Read Overview of the classes.

I’m gonna walk you through where to look to get a feel for my implementations. *MannequinCharacter_BP* is a class that integrates the main functions of the Plugin. It is the best place to start noting my implementation, organized by EventGraph. Read my notes below and go through each section to get a feel about how everything is tied together.

Starting at EventGraph

- Look at Begin Play.
- AssignStatInformationWidgetInterface is implemented on the StatBar widget, and handles event binding to the *StatManger*, ensuring the widget is updated appropriately.
- Weapons are replicated actors, and spawned on the server. They are managed by *WeaponManagerBP* when AddWeaponToArray()
- PointDamageEvent demonstrates how to use the *StatManager,* changing stat values and status effects.
- On Effect Initialize, On Effect Clear, On Effect Pause, On Effect Resume demonstrates how to integrate the MaterialsManger with the StatManger, so that the skeletal mesh changes color with given status effects. Note that

the skeletal mesh material must have the *StatusEffectVisual* material function plugged into the emission material node.

DefaultInputs (EventGraph)

- Identical to the BP inputs on the BP version of the ThirdPerson Template.

WeaponInputs(EventGraph)

- Left Mouse, Q, E, Right Mouse button. Didn't use specific action events in the project settings, I leave this up to you.
- Project settings events are the same as the ThirdPerson template. (This is different in the ALS Combine version of the plugin.)

WeaponNotifyStateInterface (Event Graph)

- The interface WeaponNotifySateInterface_BP integrates the many out of the box notify states. See Blueprints/NotifyStates and Interfaces folder for an appreciation of the depth.
- The Notifies themselves check to see if the owner has Authority before setting variables or capsules traces.
- *MannequinCharacter_BP* has the variables bIsAttacking and bIsBlocking. (Variable category: Melee.) These variables are set by the notifies, from the montages. The C++ UInterface *IUtilityAIManagerToPawnInterface* is implemented by *MannequinCharacter_BP*. *IUtilityAIManagerToPawnInterface* reads values from this AActor in two general cases: when this AActor is the focus of an AI with AUtilityAIComponent, and when it is the pawn of an AI with AUtilityAIComponent. This interface is essential for reading the state of the focus and controlled pawn.

UtilityCPCInterface (Event Graph)

- UtilityControlledPawnCommands_BP is a Blueprint Interface implemented here.
- This interface is what the TaskComponents in Blueprints/ActorComponents/TaskComponents call to direct the AI to do specific tasks. This is what you want to extend when adding your own functionality.

- For example, the MeleeWeaponTask_BP tells the controlled pawn via the CPC interface to swing the weapon when the task is started. *MannequinCharacter_BP* handles deciding how to actually do the task. Here, we query the *WeaponManager_BP* for the active weapon, determine if it has a MeleeMontageComponent, and *PlayNextMontage()*. See Weapons/Sword_BP for an example. *MeleeMontageComponent_BP* is a *UMeleeMontageComponent* that has variables and methods for specifically controlling when a given montage can play, sequencing montages, and interrupting other montages. Note that the component uses soft references, so you are responsible for loading the montages in. (See Soft references section.)

WeaponDeletages (Event Graph)

- Handles changing the animation state of the character skeletal mesh. *WeaponManager_BP* automatically calls delegates when the character changes weapon. Replicated.

Other Character Notes

1. UE4_Mannequin_Skeleton has been modified. Important sockets are Sword_Holster, Gun_Holster, Gun_Equip, Sword_Equip. *WeaponManager_BP* reads the data table *EquipData* to know what montages and sockets to use for a specific weapon type. There is no limit to the number of weapon categories, but each needs a unique FName that must match that look up row in the *EquipData* UDataTable*
2. Weapons have their own *UStatManager*, and their own stat data table describing their base stats. When a base stat is queried by the character, the character *UStatManager* will sum its own stats with the stats of equipped weapons. *AWeaponActor* handles interacting with the *UStatManager* of the actor who wields it. *UWeaponManager* calls *Equip(), Holster(), Remove()* on the *AWeaponActor*, to handle these operations. Comically, weapons can have their own status effects, and those calculations will be handled.
3. Stat Nuances
    a. GetStatValue() Sum of a stat using: Dictionary, Effectors (status Effects). Doesn't include other StatManagers.
    b. GetStatTotal() Sum of a stat using: Dictionary, Effectors, Bindings, Other Stat Managers

c. GetRawStat(): Only return the value of stat as defined in the *StatDictionary*. Doesn't care about Bindings, Other stat managers, and Effectors. Keeps things simple.

Intended Audience

This product is for users with Unreal experience. I make use of soft pointers and expect you to either read up on my notes, or already understand how to use them. While I give an overview about the function of each part, important methods, important variables, and integration, users familiar with C++ will get the most out of this plugin. I make use of both C++ and Blueprint, and nearly everything is exposed to Blueprint.

I organized the header files to contain the variables and methods in alphabetical order, with _Server and _Multicast variants near the corresponding method.

You don't need to understand replication to use this system, and I actively hide the need to think about replication from you. I do this with a very simple set up that I use throughout the plugin. Replication is often handled at the component level, and uses RPC (Remote procedure calls)

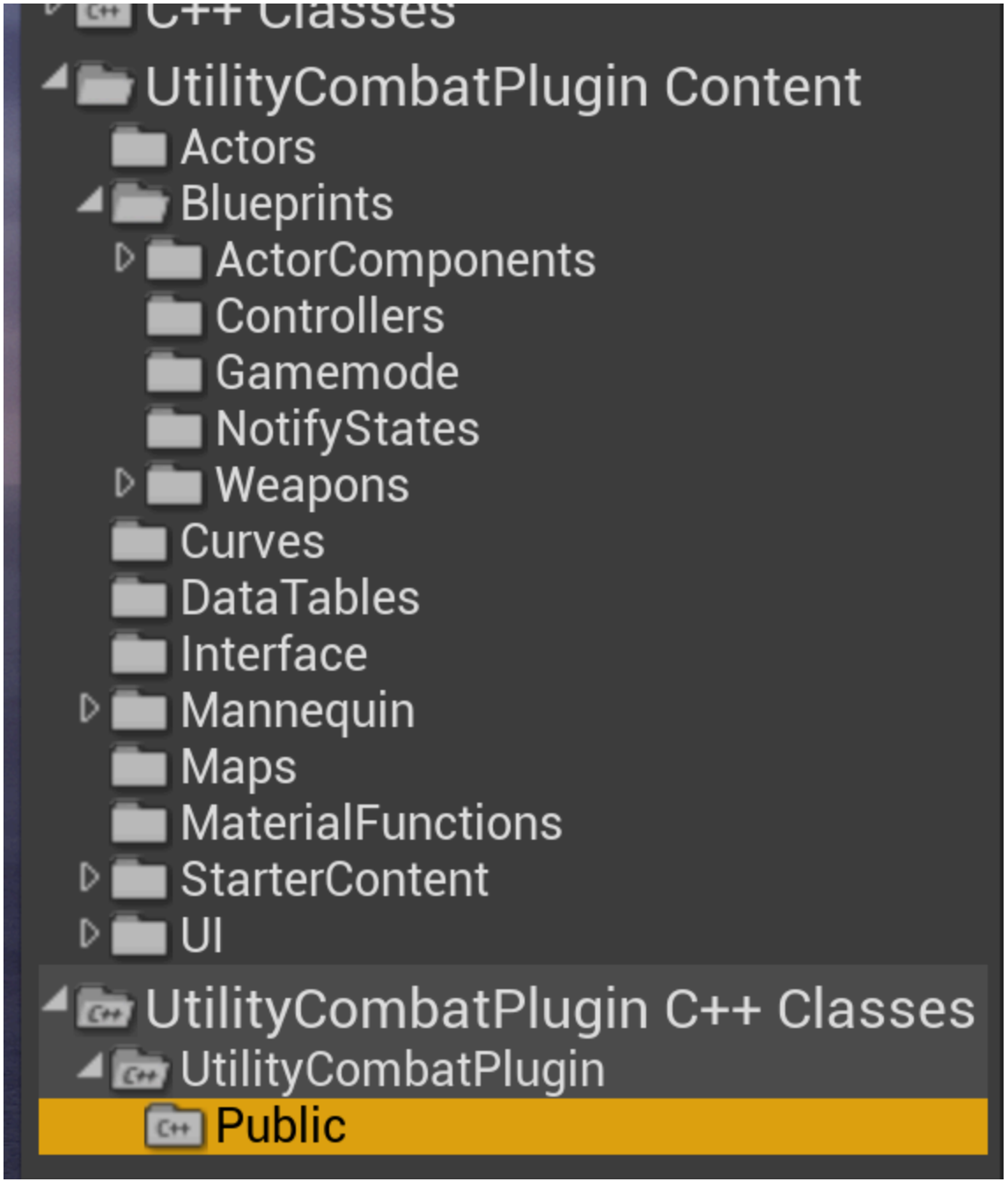
Example:

```
UFUNCTION(BlueprintCallable)
Void DoSomething()

UFUNCTION(BlueprintCallable, Server, Reliable)
Void DoSomethingServer()

UFUNCTION(BlueprintCallable, NetMulticast, Reliable)
Void DoSomethingMulticast()

If the OwnerActor does not have authority, and is an autonomous proxy,
        DoSomethingSever()
Else if OwnerActor HasAuthority()
        DoSomethingMulticast()
```

The function DoSomethingServer() simply calls DoSomethingMulticast(). When a multicast function runs on the server, it will execute it locally, and then tell the clients to run DoSomethingMulticast(). Essentially, the original DoSomething() function is a simple wrapper for ensuring Multicast functions are called on server, or that a player (usually an autonomous proxy) will be able to call to a server RPC, to call the multicast RPC, to ensure actions happen to multiple clients. AI will have to be run on the server, and then call multicast RPCs.

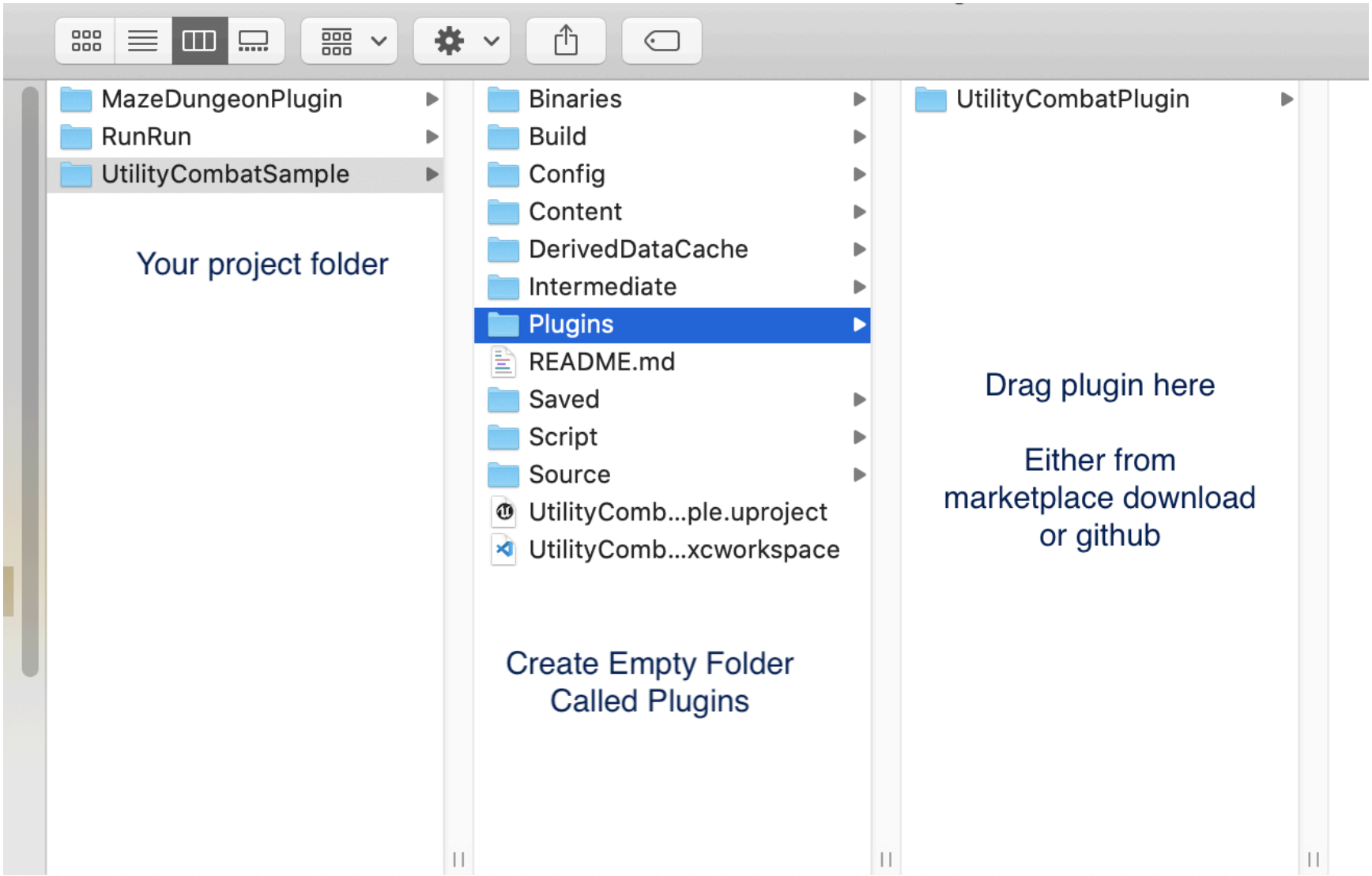
Recommendations

1. This system is large and complicated. Read all the documentation, and go through each header file. Pay close attention to C++ overview table; it will help orient you to the pieces. The time you save making use of an out of the box solution is offset by the time it will take to understand everything that is going on. However, this should still result in a net amount of time saved.
2. Create a folder called “Plugins" in your project, and drag this plugin into it. I include things like a PlayerController, AIController, WeaponManagerComponent. Each project will likely have specific needs, and you’ll want to be able to easily edit these files to tailor this plugin to your specific needs and use it as a springboard for your work. If you don’t see the content, be sure to toggle show engine content in the view options.
3. If you don’t want to drag this into the project folder, then check slow engine content in the view options. The two folders you should see are UtilityCombatPlugin Content and UtilityCombatPlugin C++ Classes
4. For the C++ classes, note I often call an Initialize() function to set default values, like a WorldTimer manager, or find specific components. Note these methods when you read the C++. If you wish to write your own subclasses, ensure Initialize() is called as appropriate.
5. In Blueprint, when you drag a pin of an object that is derived from a C++ class, type one of the categories I’ve noted in the overview table. This will give you an idea of what each part can do.
6. Overview map has a basic implementation of everything. WASD, mouse, Q/E change weapons.


C++ Classes
UtilityCombatPlugin Content
Actors
Blueprints
ActorComponents
Controllers
Gamemode
NotifyStates
Weapons
Curves
DataTables
Interface
Mannequin
Maps
MaterialFunctions
StarterContent
UI
UtilityCombatPlugin C++ Classes
UtilityCombatPlugin
Public

I recommend you have this plugin in a created plugin folder, so your edits to the plugin are specific to a particular project. Many projects have their own unique needs when it comes to weapon systems, and this allows you to better customize the project to your needs.

(See image on next page, google doc not behaving)


MazeDungeonPlugin
RunRun
UtilityCombatSample
Your project folder
Binaries
Build
Config
Content
DerivedDataCache
Intermediate
Plugins
README.md
Saved
Script
Source
UtilityComb...ple.uproject
UtilityComb...xcworkspace
Create Empty Folder
Called Plugins
UtilityCombatPlugin
Drag plugin here
Either from
marketplace download
or github

Soft References

Soft References are a way to create references that don't cause dependencies (like hard references). Additionally, a SoftReference requires that you manually load the asset. This is nice because it gives the designer choices about when to load what, and it avoids frame hitches.

Instead of using a hard reference for Montage objects with a UAnimMontage* pointer, I use a TSoftObjectPtr<UAnimMontage> object. I made this choice because montages can reference a lot of different assets. (Sounds, animation notify + dependencies, Particle effects, Skeletal Mesh with its materials and textures, and associated animation data). As the montages are variables on components, it becomes incredibly easy to bloat your dependency graph with montages on the actor component.

For example a "GunWeapon" could have a Firing Component with a Actor class to reference its bullet, and a montage for reloading. When said "GunWeapon" is loaded, all its hard references would have to be loaded with it. And if skeletal mesh + other dependencies for that montage aren't in use, you just bloated your project with *one weapon*. By using Soft References for the Actor bullet and the montages, this is avoided. I take this approach here.

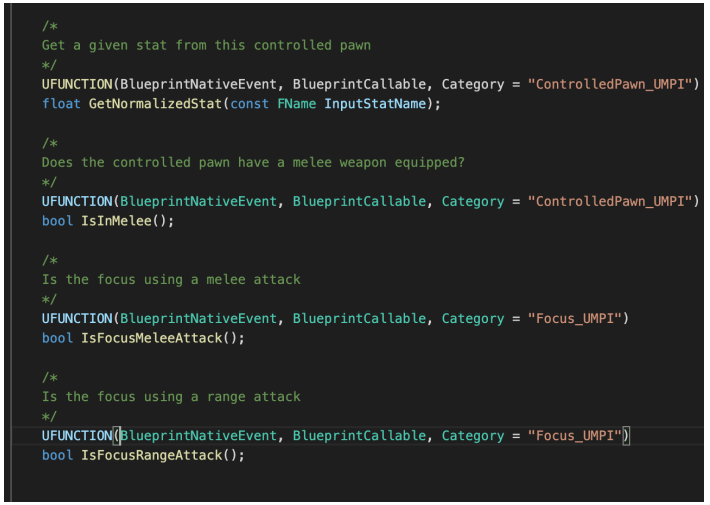
How to load Soft References

1. In Blueprint (Not my expertise, but here is a resource.) Gotta know how to use the primary asset manager
    a. https://www.youtube.com/watch?v=K0ENnLV19Cw
2. It is possible to do it in C++.
    a. Read this https://docs.unrealengine.com/en-US/Programming/Assets/ReferencingAssets/index.html
    b. And this https://docs.unrealengine.com/en-US/Programming/Assets/AsyncLoading/index.html
    c. You will notice an object called UGameGlobals is referenced, but it doesn't exist in Unreal's Source code. This class is called a singleton. To set up a singleton, read my write up on it. (It is here, see SoftReference section: https://docs.google.com/document/d/16Nb3Mn-8Z-CkRFEgq8F6EA_wW-dhYR5YOuj042mgsyg/edit?usp=sharing)
3. Drag the object that is being referred to by the Soft Reference into the scene. This force loads it, and creates a hard reference between the level and said object. This is useful for debugging and quick and dirty testing. I take this object approach in the overview map, and leave it to you to decide how you want to load your objects. If you were to remove the actor playing a montage from the scene, and then try to play that montage, you will notice that the montage will fail to play. In the same vain, if you were to remove the bullet actor from the scene, you wouldn't be able to fire bullets.

4. Trust I know what I'm doing, use my implementation that is in another plugin. I included an Asyc asset loader (On the actor component level.) with my Maze Dungeon Generator Plugin. I already needed to have a singleton to asynchronously load the UWorlds, so it made sense to add the functionality there.

Overview of C++ classes.

Class	Parent	Blueprint(s)	Description
UMaterialManagerComponent	UActorComponent	MaterialManager_BP	Sets up Dynamic materials on a given skeletal mesh, and manages the visual effects of status effects. Important Method Categories: Effects
UMeleeMontageComponent	UActorComponent	MeleeMontageComponent_BP	Uses Multicast RPC to play montages over the network. Has several ways to determine whether a montage should be played. Important Method Categories: Montage
UStatManager	UActorComponent	StatManager_BP	Uses RPC's to keep Stat dictionaries in sync. Has methods relating to creation of Status effects, and reading data tables. Also can reference other Stat components to get a total sum of a stat between them. Important Method Categories: Stat, Status, Stat Binding, Stat Query, Utility, Saving And Loading
UStatusEffectComponent	UActorComponent	None	Created and destroyed at Runtime as needed. Sends a multicast RPC via the master stat manager everytime it wishes to apply an effect.
AUtilityAIController	AAIController	UtilityAIControllerMeleeFocus_BP UtilityAIControllerRangeFocus_BP	1. Exposes aspects of the IGenericTeamAgentInterface to Blueprint, so you can set up teams. 2. Provides basic implementation of GetTeamAttitudeTowards(). 3. Overrides GetFocalPointOnActor(), and provides exposed float *FocusEyeHeight*, so your enemies won't stare at the Player's feet.

UUtilityAIManagerComponent	UActorComponent	UtilityTaskManagerComp_BP	Manages all the UtilityTasks, implemented as ActorComponent. Decides which is the best task, and stores relevant variables relating to the state of the focus. It is meant to be attached to an AAIController.
IUtilityAIManagerToPawnInterface	UInterface	None	Allows the UtilityAIManagerComponent to query information about its controlled pawn, independent of class. It allows this component to query the normalized value of a stat, whether the controlled Pawn is in Melee, and how the focus is attacking. (This interface would have to be implemented on both the enemy character, and player character.)  ``` /* Get a given stat from this controlled pawn */ UFUNCTION(BlueprintNativeEvent, BlueprintCallable, Category = "ControlledPawn_UMPI") float GetNormalizedStat(const FName InputStatName); /* Does the controlled pawn have a melee weapon equipped? */ UFUNCTION(BlueprintNativeEvent, BlueprintCallable, Category = "ControlledPawn_UMPI") bool IsInMelee(); /* Is the focus using a melee attack */ UFUNCTION(BlueprintNativeEvent, BlueprintCallable, Category = "Focus_UMPI") bool IsFocusMeleeAttack(); /* Is the focus using a range attack */ UFUNCTION(BlueprintNativeEvent, BlueprintCallable, Category = "Focus_UMPI") bool IsFocusRangeAttack(); ```
UUtilityCombatTaskComponent	UActorComponent	UtilityTaskComponentBase_BP And every task in the task component folder	Meant to be attached to an AAIController, and each component encapsulates a specific task for the AI. The function Initialize() on the UUtilityAIManagerComponent finds all of these components, assigns each task component the controller. See dedicated page for details
AUtilityPlayerController	APlayerController	UtilityPlayerController_BP	Implements and exposes aspects of the IGenericTeamAgentInterface. Also implements line trace

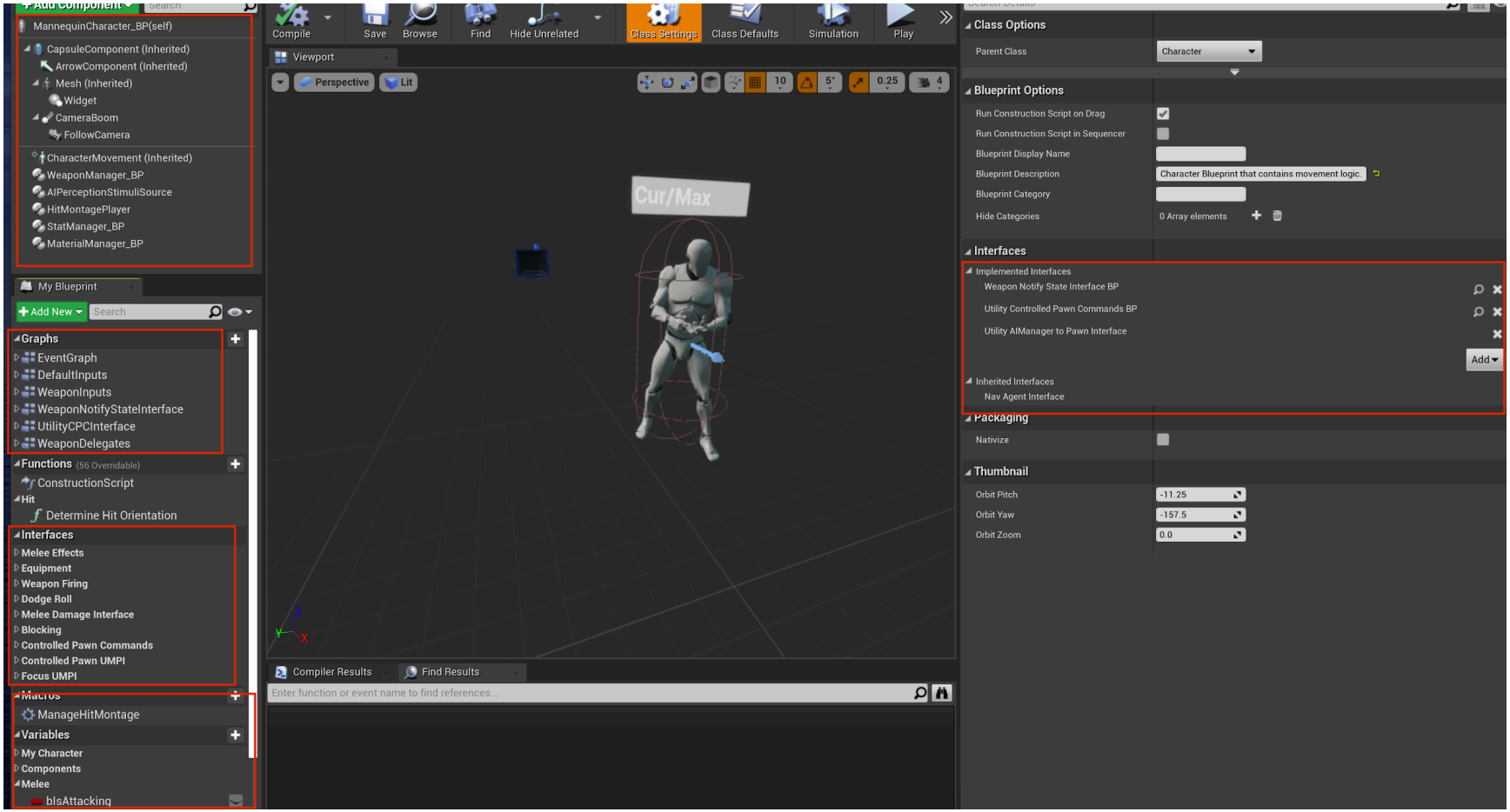
			method for crosshairs.
AWeaponActor	ASkeletalMeshActor	Sword_BP BurstFireGun_BP FullAutoGun_BP Shotgun_BP	A base class for a replicated weapon, meant to be attached to a character's skeletal mesh. Overrides GetNetConnection() to return the net connection from GetAttachParentActor(). This ensures actor components attached to this actor can fire RPC's. Has a UStatManager created in the constructor. Delegates: On Weapon Equipped On Weapon Holstered Query Methods: HasFiringComponent() HasMeleeComponent() Blueprint Native Event: (events that can be overridden in BP) ReloadWeapon() FireWeapon() FireWeaponEnd() Functions for internal use. SetStatusOfDictionary() Equip() Holster() Remove()
UWeaponFiringComponent	USceneComponent	FiringComp_BP	A replicated component that can fire line traces or any actor as projectiles. Can be attached to a ACharacter or AWeapon,with no change in functionality or behavior. It handles all of that behind the scenes. Also handles: Ammunition Single Fire vs Burst Fire Firing Cooldown, and Automatic Firing

			Manage Timers, and cleaning up timers Reloading, playing reload montage CanFire() method Firing line traces and projectiles in patterns. Important Method Categories: Firing
UWeaponManager	UActorComponent	WeaponManager_BP	A replicated component that manages the inventory of AWeaponActor's.Meant to attached to ACharacter with a skeletal mesh.Responsible for: 1. Attaching and detaching weapon to the owner's skeletal mesh at specific sockets. 2. Playing equip and holster montages, (optional to change the equipped weapon) 3. Having an active weapon, but have said equip weapon be either equipped or holstered. 4.Manage changes in the Owner's UStatManager and the Weapon's UStatManager to respond to equipment changes. Replicated. When EquipWeapon() and HolsterWeapon() are called with the parameter bImmediatelyChangeActorSocket = true, then the AWeaponActor'sEquip() and Holster() will be called respectively. This passes down Owner's UStatManager to the weapon, such that the weapon can add it's stat component to the owner's OtherStatManagers Array.The AWeaponActor also changes bStatDictionaryCanModifyOtherStatDictionary in its UStatManager to respond to whether it is equipped or holstered. Important Method Categories: Weapons

Overview of Blueprint Only Objects

Blueprint	Parent	Description
AnimationCommands_BP	Interface	This interface is meant to be implemented by an Anim Instance (Animation Blueprint). It provides useful methods for communicating to skeletal meshes anim instance without casting. For example, setting death state, Equip State (From BP enum), or playing hit montages. While there are methods for attack and block montages, they are **NOT** used here. Attacking and blocking montages are handled on the component level. They are defined here in case you want to define attack animations in the animation blueprint itself. For example, enemies that have a weapon attached to the mesh that don't need a complicated weapon system, and can just be commanded to attack from the animation BP level.
WeaponNotifyStateInterface_BP	Interface	This interface is meant to be implemented by a character blueprint. It allows various notify states to perform their function. They communicate to the their owner character via this interface, without caring about the class of the character It contains the categories: 1. Melee Damage Interface (For Melee combat. Getting the sockets for capsule trace) 2. Weapon Firing. (When it is necessary to fire from animation notify state) 3. Blocking 4. Melee Effects. (When we have a successful hit with a melee

		weapon) 5. Dodge Roll (For setting the state of a bool to keep track of when a dodge roll should be active or not) 6. Equipment (For changing the weapon socket of an equipped AWeaponActor at a specific point on a montage)
UtilityControlledPawnCommands_BP	Interface	This interface is meant to be implemented by a character. It is used by the blueprint tasks defined in Blueprints/ActorComponents/TaskComponents. Implement this on characters you want my AI system to control.
WidgetInterface_BP	Interface	This interface is implemented by the StatBar blueprint. In UI/Widgets. It has one method, AssignStatInformation(). It is used to link the widget to a UStatManager, and give the widget a color and stat name to manage.
RangeAndMeleeAnimBlueprint	Anim Instance	An animation BP that uses animations from the Paragon Characters GreyStone and Belecia. Its formatting also takes after those anim BP. I have already set it up to change between melee and range animations from a given EquipType.
PlayerHud	HUD	Quick and dirty hub that relies on casting to MannequinCharacter_BP to get its stat component. Then it adds a PlayerHudWidget to the viewport
PlayerHudWidget	User Widget	Quick and dirty HUD that displays the health stat.
MannequinCharacter_BP	ACharacter	Contains blueprint inputs identical to the Third Person Template blueprint starter project. I implemented many of my interfaces already, and set up event graphs. A picture is worth a 1000 words, and you can see what components are already on the blueprint. It's designed to work out of the box, and have events be broken down into logical categories.


Add Component
MannequinCharacter_BP(self)
CapsuleComponent (Inherited)
ArrowComponent (Inherited)
Mesh (Inherited)
Widget
CameraBoom
FollowCamera
CharacterMovement (Inherited)
WeaponManager_BP
AIPerceptionStimuliSource
HitMontagePlayer
StatManager_BP
MaterialManager_BP
My Blueprint
Add New
Search
Graphs
EventGraph
DefaultInputs
WeaponInputs
WeaponNotifyStateInterface
UtilityCPCInterface
WeaponDelegates
Functions (56 Overridable)
ConstructionScript
Hit
Determine Hit Orientation
Interfaces
Melee Effects
Equipment
Weapon Firing
Dodge Roll
Melee Damage Interface
Blocking
Controlled Pawn Commands
Controlled Pawn UMPI
Focus UMPI
Macros
ManageHitMontage
Variables
My Character
Components
Melee
bIsAttacking
Compile
Save
Browse
Find
Hide Unrelated
Class Settings
Class Defaults
Simulation
Play
Viewport
Perspective
Lit
Cur/Max
Compiler Results
Find Results
Enter function or event name to find references...
Class Options
Parent Class
Character
Blueprint Options
Run Construction Script on Drag
Run Construction Script in Sequencer
Blueprint Display Name
Blueprint Description
Character Blueprint that contains movement logic.
Blueprint Category
Hide Categories
0 Array elements
Interfaces
Implemented Interfaces
Weapon Notify State Interface BP
Utility Controlled Pawn Commands BP
Utility AIManager to Pawn Interface
Add
Inherited Interfaces
Nav Agent Interface
Packaging
Nativize
Thumbnail
Orbit Pitch
-11.25
Orbit Yaw
-157.5
Orbit Zoom
0.0

UtilityCombat Task Component Notes and Scoring

Many specific tasks have already been implemented onto an AI Controller. See the Blueprint UtilityAIControllerRangeFocus_BP to see the integrated system, how I initialized it, how I managed AI perception, and how I added all the tasks. The function CalculateTaskScore() returns a float from 0.0f to 1.0f that determines how optimal the task is at a given time. (1.0f = Best, 0.0f is worst). This function is a *BlueprintNativeEvent*, so you can account for whatever blueprint related variables. (Do call the parent C++ function though.) Importantly, CalculateTaskScore() first calls IsTaskReady(), to determine if we can even attempt the task. If it fails this check, CalculateTaskScore() returns 0.0f.

In order to explain how each task component is scored, I first need to explain the FUtilityCurveCollection data structure. This data structure contains an *ECurveInputQuery CurveInputQuery*, a *StatName* (FName), a *CurveFloat* (the actual curve), *CurveDampen* (float), *CurveOutput* (float), and *bMultiplyThisCurveOutputToRunningTotal*.

1. The *CurveFloat* is a variable of type UCurveFloat* . It expects to reference a curve that can take some normalized input (from zero to one), and output a normalized value. I don't strictly enforce this, and expect you to be cognizant of how you set up your tasks.
2. The enum *CurveInputQuery* defines what values will be *input* into the given *CurveFloat*. There are various comparisons that query variables in the UtilityAIManagerComponent. These are explained in *UtilityCombatDataStructures.h* and defined in the function UUtilityAIManagerComponent::GetNormalizedStat(const ECurveInputQuery) const. Many of these functions return 0.0f if a specific condition is false, and 1.0f if a condition is true. For example, passing in ECurveInputQuery::HasFocus will return 1.0f if there is a focus, 0.0f if there is not a focus. In this way, I can map relevant conditions to a float and design curves accordingly.
3. StatName is an FName, and will only matter if CurveInputQuery = ECurveInputQuery::STAT_BY_FNAME. Even though ECurveInputQuery covers many relevant conditions, it isn't comprehensive and some sort of general system is necessary. If you decide to designate input by StatName, the function UUtilityAIManagerComponent::GetNormalizedStat(const FName& InputStat) const will be called. This function queries the controlled pawn, to determine if it implements the UUtilityAIManagerToPawnInterface; If it doesn't, then the curve input will be 0.0f. If it does, then interface function GetNormalizedStat() will be executed on the AI's controlled pawn. This interface function is exposed to blueprint, and is already implemented for you in MannequinCharacter_BP. (Category: Controlled Pawn UMPI). Hence, any FName can be used to query any specific value on the controlled pawn. Just remember to normalize the value by dividing the current value over the max value.

4. Once we have a suitable normalized input, we evaluate it at the curve. [Calling CurveFloat->GetFloatValue(NormalizeInput).] This value is the *NormalizedCurveOutput*. Then we set *CurveOutput* = *NormalizedCurveOutput*CurveDampen*. This allows you to debug your AI by viewing the output of each curve.
5. Note that the *CurveDampen* is a scaling factor for a curve. It's default value is 1.0f.

Finally, the total score of a task is calculated as a running total. Starting at 1.0f, we evaluate the first curve. If FUtilityCurveCollection has *bMultiplyThisCurveOutputToRunningTotal* set to true, then the running total will be multiplied by *NormalizedCurveOutput*CurveDampen*. If *bMultiplyThisCurveOutputToRunningTotal* is set to false, then *NormalizedCurveOutput*CurveDampen* will be added to *RunningNormalizedUtilityValue*.

NOTES:

- Having a series of curves multiplied together serves as **AND** operator between them.
- Having a series of curves added together is similar to the **OR** operator, except the **OR** can be weighted based on many different conditions. Ensure you scale *CurveDampen* appropriately, or tasks will be given a UtilityScore greater than 1.0f. This means the task will be picked over tasks that cannot become greater than 1.0f.
- Curves are evaluated in the order as defined in the component variable. TArray<FUtilityCurveCollection>*CurveCollection*.
- EnterTask() and ExitTask() are BlueprintNativeEvents. Ensure you call their parent functions, and override these to have your AI perform the tasks themselves.

Major variables

Type	Name	Description
bool	bIsTaskActive	True if the task is occurring, false if the task is done. Controlled internally by EnterTask() and ExitTask()
TArray<FUtilityCurveCollection>	CurveCollection	The list of curves you want to evaluate. Contains information about how you want to query them.
bool	bOnlyDoTaskOnce, bPerformedTask, bTaskLocked	Several variables under the flow category. They influence whether a task is ready. If bTaskLocked is true, then IsTaskReady() will return false.

float	CurrentCooldown, MinCooldown, MaxCooldown	Category: Cooldown. We keep track of the time when tasks fire, and we can force tasks to wait for a period of time before firing again. If true, *bSetCurrentCooldownBetweenMinMax* will randomize the cooldown when a task is Entered.
int32	TaskLayer	Tasks on different layers can operate concurrently. Each layer can only have one active task at a time
FName	TaskName	VERY IMPORTANT. I use the TaskName to compare task components. (I.e Determine whether a CurrentTask is different from the BestTask.) This information is also broadcasted to the delegates *OnAnyTaskEnter* and *OnAnyTaskExit*
EUtilityInterruptionType	InterruptionType	Determines if another task can stop this task if its UtilityScore becomes higher than this task. EUtilityInterruptionType::ALWAYS EUtilityInterruptionType::NEVER EUtilityInterruptionType::PRIORITY EUtilityInterruptionType::PRIORITY Means another task can only interrupt this task if its *InterruptionPriorityNumber* is strictly greater than this task's *InterruptionPriorityNumber*

Known Issues

Crash upon using StatManager_BP.
Cause: StatusEffect data table in stat data table variable. (I swore I fixed this earlier.)
Fix: Clear the variables and assign the correct data tables.
This should be fixed in 4.27

Changelog

4.27 Minimal API changes to StatManager.h

So I now get a build Error for trying to use a TMap<> in an RPC. So I modified the Server and Multicast RPC's for SetStatBindings, and SetStatDictionary. The blackbox function takes a dictionary, then breaks it into two TArrays, and then passes them into the correct RPC. The multicast version rebuilds the dictionary based on the given TArrays.

Note: I didn't flag the pass by reference with UPARAM(ref) macro, but those RPC's aren't exposed to BP. Let me know if you want me to expose them to BP.

5.0-5.1 fixed bug related to reloading on the Scene component. Github version has the fix. I was going to push this fix out to all versions but now I have a build problems on xcode 14.

Status of 5.1

5.1 is delayed until I either can get my windows computer up in running, or Epic fixes Mac building in Unreal with Xcode 14.
I do my development on a Mac. This normally isn't an issue. It became an issue. Here are the details.

Unreal Engine 5.1 requires Xcode 14. I was on Xcode 12. To get Xcode 14, you have to upgrade your OS. I did that.
With 5.1 I try to build the plugin, I get the error "Platform Mac is no a valid platform to build".

I do a bit of research, find that this error has already been reported and Epic published a quick fix. QFE patch. I download the patch. I cannot install the patch because I get the error that the QFE is damaged. I redownload it. Same error.

I'm tired.