Ultimate Starter Kit

Documentation

Henry Jooste

None

Table of contents

1. Getting Started

1.1 Requirements

The Ultimate Starter Kit plugin is only available for Unreal Engine 5.0.3 and newer. The plugin also depends on the following plugins:

1. Niagara
2. Enchanced Input

1.2 Installation

1. Download the latest release from GitHub
2. Navigate to `C:\Program Files\Epic Games\UE_{VERSION}\Engine\Plugins`
3. Create a `Marketplace` folder if needed
4. Extract the release and copy to the `Marketplace` folder
5. Open Unreal Engine
6. Click on `Edit > Plugins`
7. Enable the plugin under the `Built-in > Other` category
8. Restart Unreal Engine

1.3 Plugin Content

The Ultimate Starter Kit plugin includes content that can be used in your Blueprints. You might need to enable this first:

1. Open the `Content Browser`
2. Click on the settings button
3. Enable the `Show Plugin Content` setting

1.4 Features

The Ultimate Starter Kit plugin comes with the following features:

- Logger: A system used to easily log info to file and via on-screen messages
- Save data management: A system used to easily save/load game data with support for multiple save slots
- Currency: A system that is used to easily manage different types of currency
- Audio: A system used to manage the basic properties of audio files
- Stats: A system used to easily manage character stats with an optional regenerate ability
- 3D platformer character: Basic 3D platformer character and animation template

2. Game Instance

2.1 Introduction

A base game instance with support for saving and loading game data using multiple save slots

2.2 Dependencies

The game instance relies on other components of this plugin to work:

- Logger: Used to log useful information to help you debug any issues you might experience

2.3 Using the Game Instance

You need to create a blueprint using the `USKGameInstance` as a parent before using the game instance. After creating your own game instance blueprint, set this as the default game instance:

1. Open the Project Settings
2. Go to Project > Maps & Modes
3. Change the `Game Instance Class` value to your own blueprint

2.4 Save Data

You need to create a `USK Save Game` object before you can save/load data. This object contains all the data that you want to save. Just add the data you want to save as variables to the blueprint. The `Game Instance` will handle the rest. You also need to set the following properties before you can save/load data:

- **Save Game Class:** A reference to the `USK Save Game` class that contains the data you want to save

NB: *You are required to set the save slot before you can save/load data. If not, you will get a* `nullptr` *and might cause your game to crash*

2.5 API Reference

2.5.1 Properties

Property	Description	Type	Default Value
SaveGameClass	The class that holds the data that should be saved/loaded	TSubclassOf<USaveGame>	`nullptr`

2.5.2 Events

Name	Description	Params
OnDataLoadedEvent	Event used to notify other classes when the save data is loaded	

2.5.3 Functions

Name	Description	Params	Return
GetSaveData	Return the save data from the current save slot. You will need to cast this to your specific save game class before you can access the data		**USaveGame*** A reference to the current save data
SaveData	Save the data to the current save slot		
SetCurrentSaveSlot	Set the current save slot used to save/load data. You are required to set the save slot before using the save manager. If not, you will get a `nullptr` and might cause your game to crash	**Index (int)** The index of the save slot to use	
IsSaveSlotUsed	Returns a boolean value indicating if the specified save slot is in use (contains data)	**Index (int)** The index of the save slot to check	**bool** A boolean value indicating if the save slot is used

2.6 Blueprint Usage

You can save/load data using Blueprints by adding one of the following nodes (requires a reference to the `USK Game Instance`):

- Ultimate Starter Kit > Save Data > Set Current Save Slot
- Ultimate Starter Kit > Save Data > Get Save Data
- Ultimate Starter Kit > Save Data > Save Data
- Ultimate Starter Kit > Save Data > Is Save Slot Used

2.7 C++ Usage

Before you can use the plugin, you first need to enable the plugin in your `Build.cs` file:

```
PublicDependencyModuleNames.Add("USK");
```

The game instance can now be used in any of your C++ files:

```
#include "USK/Core/USKGameInstance.h"

void ATestActor::Test()
{
    UGameInstance* CurrentGameInstance = UGameplayStatics::GetGameInstance(GetWorld());
    UUSKGameInstance* GameInstance = dynamic_cast&lt;UUSKGameInstance*&gt;(CurrentGameInstance);

    bool IsSlotInUse = GameInstance->IsSaveSlotUsed(0);
    if (IsSlotInUse)
    {
        GameInstance->SetCurrentSaveSlot(0);
        USaveGame* SaveGame = GameInstance->GetSaveData();
        UMySaveGame* MySaveGame = dynamic_cast&lt;UMySaveGame*&gt;(SaveGame);
        MySaveGame->CurrentLevel++;
        GameInstance->SaveData();
    }
}
```

3. Logger

3.1 Introduction

A system used to easily log info to file and via on-screen messages

3.2 Log Levels

This plugin supports the following log types:

1. **Trace:** Logs that contain the most detailed messages. These messages may contain sensitive application data. These messages are disabled by default and should never be enabled in a production environment
2. **Debug:** Logs that are used for interactive investigation during development. These logs should primarily contain information useful for debugging and have no long-term value
3. **Information:** Logs that track the general flow of the application. These logs should have long-term value
4. **Warning:** Logs that highlight an abnormal or unexpected event in the application flow, but do not otherwise cause the application execution to stop
5. **Error:** Logs that highlight when the current flow of execution is stopped due to a failure. These should indicate a failure in the current activity, not an application-wide failure

The log levels corresponds to the following verbosity level in Unreal Engine:

Log Level	Log Verbosity
Trace	VeryVerbose
Debug	Verbose
Information	Display
Warning	Warning
Error	Error

The plugin will automatically ignore certain log levels based on the type of build:

Log Level	Development	Shipping
Trace	Enabled	Disabled
Debug	Enabled	Disabled
Information	Enabled	Enabled
Warning	Enabled	Enabled
Error	Enabled	Enabled

3.3 Logging Methods

There are 2 different logging methods. Both of these are used each time you log something:

- **On-screen messages:** These messages will appear on-screen for 5 seconds (only used when running the game through the editor)
- **Log File:** Everything you log is also written to a file using the Unreal Engine logging feature

3.4 API Reference

3.4.1 Macros

Name	Description	Params	Return
USK_LOG_TRACE	Log trace information using the current function name as the tag	**Text (FString)** The text that should be logged out	
USK_LOG_DEBUG	Log debug information using the current function name as the tag	**Text (FString)** The text that should be logged out	
USK_LOG_INFO	Log information using the current function name as the tag	**Text (FString)** The text that should be logged out	
USK_LOG_WARNING	Log a warning using the current function name as the tag	**Text (FString)** The text that should be logged out	
USK_LOG_ERROR	Log an error using the current function name as the tag	**Text (FString)** The text that should be logged out	

3.4.2 Functions

Name	Description	Params	Return
Trace	Log trace information	**Tag (FString)** The category of the log entry. This is usually the function or class name. It allows you to find out exactly where this is being logged **Text (FString)** The text that should be logged out	
Debug	Log debug information	**Tag (FString)** The category of the log entry. This is usually the function or class name. It allows you to find out exactly where this is being logged **Text (FString)** The text that should be logged out	
Info	Log information	**Tag (FString)** The category of the log entry. This is usually the function or class name. It allows you to find out exactly where this is being logged **Text (FString)** The text that should be logged out	
Warning	Log a warning	**Tag (FString)** The category of the log entry. This is usually the function or class name. It allows you to find out exactly where this is being logged **Text (FString)** The text that should be logged out	
Error	Log an error	**Tag (FString)** The category of the log entry. This is usually the function or class name. It allows you to find out exactly where this is being logged **Text (FString)** The text that should be logged out	

3.5 Blueprint Usage

You can easily log info using Blueprints by adding one of the following nodes:

- Ultimate Starter Kit > Logger > Log Trace
- Ultimate Starter Kit > Logger > Log Debug
- Ultimate Starter Kit > Logger > Log Info
- Ultimate Starter Kit > Logger > Log Warning
- Ultimate Starter Kit > Logger > Log Error

3.6 C++ Usage

The logging is handled through a static class/functions. You first need to enable the plugin in your `Build.cs` file:

```
PublicDependencyModuleNames.Add("USK");
```

The logger can now be used in any of your C++ files:

```
#include "USK/Logger/Log.h"

void ATestActor::Test()
{
    USK_LOG_TRACE("Testing trace logging");
    USK_LOG_DEBUG("Testing debug logging");
    USK_LOG_INFO("Testing info logging");
    USK_LOG_WARNING("Testing warning logging");
    USK_LOG_ERROR("Testing error logging");

    ULog::Trace("Custom Tag", "Testing trace logging");
    ULog::Debug("Custom Tag", "Testing debug logging");
    ULog::Info("Custom Tag", "Testing info logging");
    ULog::Warning("Custom Tag", "Testing warning logging");
    ULog::Error("Custom Tag", "Testing error logging");
}
```

4. Trackable Data

4.1 Overview

4.1.1 Introduction

A system that is used to easily manage different types of actor data

4.1.2 Dependencies

The currency system relies on other components of this plugin to work:

- Game Instance: Used to automatically save/load the data managed by the component(s)
- Logger: Used to log useful information to help you debug any issues you might experience
- Save Data: Used to automatically save/load currency values (optional)

4.1.3 Trackable Data Component

Before you can manage the data, you need to create a Trackable Data Component and add it to the actor/character containing the data

4.1.4 Built-in data

The following data can automatically be managed without creating custom components:

1. Currency (using the `Currency Component`)
2. Stats (using the `Stats Component`)

4.2 Data

4.2.1 Introduction

All trackable data use the `FTrackableData` struct to specify the default values and behaviours

4.2.2 Properties

Property	Description	Type	Default Value
Initial Value	The initial value of the data	float	0.0f
Enforce Max Value	Should we enforce a maximum value?	bool	false
Max Value	The maximum value of the data	float	100.0f
Auto Save	Should all value updates automatically be saved using the game instance?	bool	false
Auto Generate	Should we automatically generate value every second?	bool	false
Generate Amount	The amount of value to generate every second	float	0.0f
Generate Delay	The delay before the value starts generating after losing value	float	0.0f

4.3 Component

4.3.1 Introduction

A component that is used to easily manage different types of actor data. The data to track is configured by adding items to the `Data` map. The component should be added to the actor/character containing the data

4.3.2 API Reference

Properties

Property	Description	Type	Default Value
Data	The map of data to track	TMap<FName, FTrackableData>	

Events

Name	Description	Params
OnValueZero	Event that is broadcasted every time the data value reaches 0	**Name (FName)** The name of the data item
OnValueUpdated	Event that is broadcasted every time the data value reaches 0	**Name (FName)** The name of the data item **Value (float)** The current value of the data item **ValuePercentage (float)** The percentage of the current value compared to the max value of the data item

Functions

Name	Description	Params	Return
GetValue	Get the amount of the data	**Name (FName)** The name of the data item	**float** The current amount of the data item
GetValuePercentage	Get the value of the data as a percentage of to the max value	**Name (FName)** The name of the data item	**float** The value of the data as a percentage of to the max value
Add	Add an amount to the data	**Name (FName)** The name of the data item **Amount (float)** The amount to add	**float** The new amount of the data item
Remove	Remove an amount from the data	**Name (FName)** The name of the data item **Amount (float)** The amount to remove	**float** The new amount of the data item

4.3.3 Blueprint Usage

You can easily manage the data using Blueprints by adding one of the following nodes:

- Ultimate Starter Kit > Trackable Data > Get Value
- Ultimate Starter Kit > Trackable Data > Get Value Percentage
- Ultimate Starter Kit > Trackable Data > Add
- Ultimate Starter Kit > Trackable Data > Remove

4.3.4 C++ Usage

Before you can use the plugin, you first need to enable the plugin in your `Build.cs` file:

```
PublicDependencyModuleNames.Add("USK");
```

The `Trackable Data Component` can now be used in any of your C++ files:

```
#include "USK/Logger/Log.h"

void ATestActor::Test()
{
    // StatsTracker is a pointer to the UStatsComponent

    float Health = StatsTracker->GetValue("HEALTH");
    Health = StatsTracker->Add("HEALTH", 50.0f);
    Health = StatsTracker->Remove("HEALTH", 75.0f);
    float HealthPercentage = StatsTracker->GetValuePercentage("HEALTH");
}
```

5. Audio

5.1 Audio Overview

5.1.1 Introduction

A system used to manage the basic properties of audio files. It includes different sound classes, a sound mix and sound attenuation settings

5.1.2 Dependencies

The audio system relies on other components of this plugin to work:

- Logger: Used to log useful information to help you debug any issues you might experience

5.1.3 Sound Classes

The audio system includes a few basic preconfigured sound classes:

Class name	Group	Volume
USK_EffectsSoundClass	Effects	1.0
USK_MusicSoundClass	Music	0.5
USK_UISoundClass	UI	1.0
USK_VoiceSoundClass	Voice	3.0

5.2 Audio Utils

5.2.1 Introduction

The audio utils class is used to easily play sound effects

5.2.2 API Reference

Functions

Name	Description	Params	Return
PlaySound2D	Play a 2D sound	**SoundFX (USoundBase*)** The USoundBase to play	
PlayRandomSound2D	Play a random 2D sound	**SoundFX (TArray<USoundBase*>)** The array of USoundBase to select the random sound from	
PlaySound	Play a sound at the specified actor's location	**Actor (AActor*)** The actor where the sound will be played **SoundFX (USoundBase*)** The USoundBase to play	
PlayRandomSound	Play a random sound at the specified actor's location	**Actor (AActor*)** The actor where the sound will be played **SoundFX (TArray<USoundBase*>)** The array of USoundBase to select the random sound from	

5.2.3 Blueprint Usage

You can play sounds using Blueprints by adding one of the following nodes:

- Ultimate Starter Kit > Audio > Play Sound 2D
- Ultimate Starter Kit > Audio > Play Random Sound 2D
- Ultimate Starter Kit > Audio > Play Sound
- Ultimate Starter Kit > Audio > Play Random Sound

5.2.4 C++ Usage

Before you can use the plugin, you first need to enable the plugin in your `Build.cs` file:

```
PublicDependencyModuleNames.Add("USK");
```

The currency system can now be used in any of your C++ files:

```
#include "USK/Audio/AudioUtils.h"

void ATestActor::Test(const UObject* WorldContext)
{
    // SoundFx is a pointer to a USoundBase
    UAudioUtils::PlaySound2D(WorldContext, SoundFx);

    // SoundFxArray is an array of pointers to a USoundBase
    UAudioUtils::PlayRandomSound2D(WorldContext, SoundFxArray);

    // Player is a pointer to an Actor
    // SoundFx is a pointer to a USoundBase
    UAudioUtils::PlaySound(Player, SoundFx);
```

```
    // Player is a pointer to an Actor
    // SoundFxArray is an array of pointers to a USoundBase
    UAudioUtils::PlayRandomSound(Player, SoundFx);
}
```

5.3 Music Player

5.3.1 Introduction

The music player can be used to play, pause and stop music. It also allows you to easily adjust the volume of the music

5.3.2 API Reference

Components

Name	Type	Description
AudioPlayer	UAudioComponent*	The audio player component is responsible for the music playback

Properties

Property	Description	Type	Default Value
PlayOnStart	Should the music automatically play when the actor is spawned?	bool	true

Functions

Name	Description	Params	Return
SetVolume	Adjust the playback volume of the music	**Volume (float)** The new volume of the music	
Play	Play the music		
Pause	Pause the music		
Stop	Stop the music		

5.3.3 Blueprint Usage

You can play sounds using Blueprints by adding one of the following nodes (requires a reference to a `Music Player` actor):

- Ultimate Starter Kit > Audio > Set Volume
- Ultimate Starter Kit > Audio > Play
- Ultimate Starter Kit > Audio > Pause
- Ultimate Starter Kit > Audio > Stop

5.3.4 C++ Usage

Before you can use the plugin, you first need to enable the plugin in your `Build.cs` file:

```
PublicDependencyModuleNames.Add("USK");
```

The currency system can now be used in any of your C++ files:

```
#include "USK/Audio/MusicPlayer.h"

void ATestActor::Test()
{
    // MusicPlayer is a pointer to a Music Player actor
    MusicPlayer->SetVolume(2.0f);
    MusicPlayer->Pause();
    MusicPlayer->Play();
    MusicPlayer->Stop();
}
```

6. 3D Platformer

6.1 Overview

6.1.1 Introduction

The plugin includes a basic 3D platformer character and animation template. This can easily be extended to add unique functionality

6.1.2 Dependencies

The 3D platformer relies on other components of this plugin to work:

- Logger: Used to log useful information to help you debug any issues you might experience

6.2 Character

6.2.1 Introduction

The 3D platformer character extends the base character class and adds some features used by popular 3D platformers

6.2.2 Features

The following features are included in the 3D platformer character class:

1. **Double Jumping**: Additional jump with a different jump animation
2. **Variable Jump Height**: Adjust jump height based on how long the jump button is pressed
3. **Coyote Time**: Allow the character to jump for a short time after falling off the platform
4. **Shadow Decal**: A decal used as a shadow to indicate where the character will land
5. **Adjustable Camera Distance**: The camera automatically zooms in on the character while idle and zooms out as soon as the character starts moving
6. **Jump & Land effects**: Sound effects and particles when jumping and landing

All these features can be configured to meet your needs and can also be disabled

6.2.3 Components

The 3D platformer character adds the following components compared to the base character class:

- SpringArmComponent: The spring arm component responsible for controlling the distance of the camera
- CameraComponent: The camera used by the character

6.2.4 API Reference

Properties

Property	Description	Type	Default Value
InputMappingContext	The input mapping context used by the player	UInputMappingContext*	`nullptr`
MoveAction	The move input action	UInputAction*	`nullptr`
LookAroundAction	The camera rotation input action	UInputAction*	`nullptr`
JumpAction	The jump input action	UInputAction*	`nullptr`
ShadowDecalClass	The shadow decal class used to draw a shadow below the character while in the air	TSubclassOf<AShadowDecal>	
JumpSoundEffects	An array of sound effects played when jumping	TArray<USoundBase>	
JumpParticleFx	The particle effects spawned when jumping	UNiagaraSystem*	`nullptr`
JumpParticleFxSpawnOffset	The offset applied to the location of the jump particles when spawning	FVector	(0.0f, 0.0f, 0.0f)
LandedSoundEffects	An array of sound effects played when landing	TArray<USoundBase>	
LandParticleFx	The particle effects spawned when landing	UNiagaraSystem*	`nullptr`
LandParticleFxSpawnOffset	The offset applied to the location of the land particles when spawning	FVector	(0.0f, 0.0f, 0.0f)
VariableJumpHeight	Does the character support variable jump height?	bool	true
VariableJumpHeightMaxHoldTime	The amount of time to hold the jump button to reach the max height	float	0.3f
JumpVelocity	The velocity applied to the character when jumping	float	500.0f
AirControl	The amount of lateral movement control available to the character while in the air	float	1000.0f
FallingFriction	The amount of friction to apply to lateral air movement when falling	float	3.5f
Gravity	The amount of gravity applied to the character	float	2.0f
CanDoubleJump	Can the character perform a double jump?	bool	true
IsDoubleJumping	Is the character double jumping?	bool	false
CanCoyoteJump	Does the character support coyote time when trying to jump?	bool	true
CoyoteJumpTime	The amount of coyote time for the character	float	0.375f
CoyoteJumpVelocity	The velocity applied to the character when performing a coyote jump	float	700.0f
BrakingFriction	Friction coefficient applied when braking	float	10.0f
MaxAcceleration	The rate of change of velocity	float	2500.0f
TargetArmLength	Length of the spring arm component	float	350.0f
ArmLengthMultiplier	The multiplier applied to the spring arm component when the character is moving	float	0.4f
CameraAdjustmentSpeed	The speed used when adjusting the camera distance	float	3.0f

6.2.5 Blueprint Usage

There is no additional functions exposed to Blueprints. Just create the character and add it to your level

6.3 Animation Instance

6.3.1 Introduction

The plugin includes an animation template that can be used on your 3D platformer characters

6.3.2 API Reference

Properties

Property	Description	Type	Default Value
MovementSpeed	The movement speed fo the character	float	0.0f
IsInAir	Is the character currently in the air?	bool	false
IsDoubleJumping	Is the character double jumping?	bool	false
IdleAnimation	The animation used when the character is in the idle state	UAnimSequence*	`nullptr`
WalkAnimation	The animation used when the character is walking	UAnimSequence*	`nullptr`
RunAnimation	The animation used when the character is running	UAnimSequence*	`nullptr`
JumpAnimation	The animation used when the character is jumping	UAnimSequence*	`nullptr`
DoubleJumpAnimation	The animation used when the character is double jumping	UAnimSequence*	`nullptr`
FallAnimation	The animation used when the character is falling	UAnimSequence*	`nullptr`
LandAnimation	The animation used when the character is landing	UAnimSequence*	`nullptr`

6.3.3 Blueprint Usage

You can use this template by creating your own animation blueprint and selecting `UPlatformerAnimationInstance` as the parent class. Set your animations and use this for your 3D platformer character

6.4 Shadow Decal

6.4.1 Introduction

A decal used as a shadow to indicate where the character will land

6.4.2 API Usage

Functions

Name	Description	Params	Return
Initialize	Initialize the shadow decal	**OwnerCharacter (ACharacter*)** The character owning this shadow decal	

6.4.3 Blueprint Usage

You can initialize a shadow decal using Blueprints by adding the following node (requires a reference to the `Shadow Decal Instance`):

- Ultimate Starter Kit > Shadow Decal > Initialize

6.4.4 C++ Usage

Before you can use the plugin, you first need to enable the plugin in your `Build.cs` file:

```
PublicDependencyModuleNames.Add("USK");
```

The shadow decal can now be initialized in any of your C++ files:

```
#include "USK/Character/ShadowDecal.h"

void ATestCharacter::Test()
{
    // ShadowDecal is a pointer to the AShadowDecal actor
    ShadowDecal->Initialize(this);
}
```