WitcherScript Documentation

Release latest

Oct 30, 2020

Contents

Caution: This documentation is work in progress!

This is an unofficial documentation for the WitcherScript scripting language of The Witcher 3: The Wild Hunt.

Witcher Script (.ws) is the primary scripting language for The Witcher 3: The Wild Hunt. A very large chunk of the game's logic is written in Witcher Script. And thus can drastically change game behavior. Witcher Script is a proprietary language, very similar to https://udn.epicgames.com/Three/UnrealScriptHome.html.

CHAPTER 1

Contact

This documentation is a community effort. Create an issue if you have any questions.

CHAPTER 2

Community

If you need to discuss witcher 3 modding or consult with the community, you should visit one of following places: Official Witcher 3 forums modding section

CHAPTER 3

Contents

3.1 Basic Witcher Script information sheet

If you have any previous experiences with any type of object oriented programming, you will have no issues with using wscript. Syntax and usage is very similar to (now obsolete) UnrealScript.

3.1.1 Basic Types

- int - Standard 32 bit integer (value range from -2,147,483,648 to 2,147,483,647)
- String - Standard string, use quotation marks to pass values, eg. “This is a string”
- name - Name type variable, essentially a string used for item names, tags, etc. Use apostrophes to pass values, eg. ‘this is an item name’
- float- Standard float type, eg. 1.0f
- Vector - 4 variable vector (Quaternion) ordered in XYZW (for the most part you don’t need to concern yourself with W value) order, eg. Vector(100, 100, 100, 1)
- EulerAngles - 3 variable rotation ordered in Pitch, Yaw, Roll order, eg. EulerAngles(0, 180, 0)
- bool - standard true/false boolean, also accepts 0 and 1 as inputs, however unlike Unreal Script does not accept Yes/No values
- Matrix - Indicates matrix variable type.
- array< X > - Indicates an array with X being the type of array, eg. array< int > being an array of integers

3.1.2 Class Types

Besides standard variable types, wscript also supports declaring class type variables, such as:

- CEntity - Standard entity
- CEntityTemplate - Entity template (resource type)

And much more, every scriptable class can be declared as a variable in order to be stored, get or set.

3.2 WS quick class reference cheat sheet

theGame=CR4Game
theServer=CServerInterface
thePlayer=CR4Player
theCamera=CCamera
theUI=CGuiWitcher
theSound=CScriptSoundSystem
theDebug=CDebugAttributesManager
theTimer=CTimerScriptKeyword
theInput=CInputManager

It is possible to modify those via redscripts.ini located in .bin folder, although remember that doing this will make custom class references not compile for users who not have them set up in their redscripts.ini

3.3 Exec functions

Exec functions are the simplest way to introduce custom code into Witcher 3. As handy as they are, it is important to remember their limitations:

An exec function cannot:

- Be called from code
- Be latent (have any execution time or delays)
- Loop (as they cannot be called from code)

3.3.1 Creating custom exec function

Go to your Witcher 3 installation folder, inside "Mods" folder, create new folder called "modExecFunctionTest", inside that folder create another folder called "content", go into it and create another folder called "scripts" and inside that folder create a folder called "local". Inside that folder create file called "latentfunction.ws" (make sure it's .ws not .ws.txt).

In short, your folder hierarchy should look as following:

```
The Witcher 3 Wild Hunt␣
↪GOTY\Mods\modExecFunctionTest\content\scripts\local\latentfunction.ws
```

Open the file with text editor of your preference (Notepad++ is strongly recommended) and add following code inside:

```
exec function printmystring(msg : String)
{
 GetWitcherPlayer().DisplayHudMessage(msg);
}
```

save the file and run the game, once you open console using "~"(tilde) key, you can write:

```
printmystring("Hello from witcher script!")
```

To see your string printed using game's default HUD message module.

Of course this is one of most simple examples imaginable. Exec functions are good for any kind of instantaneous work that needs to be done manually and single time (or not that often) which makes it naturally good for things like:

- Spawning entities
- Removing entities
- Getting some data from the game
- Setting some variables within other classes

3.4 Latent functions

This page will explain what is a latent function, why would you use a latent function and how to use them.

3.4.1 What is a latent function

Latent function are functions whose execution time can occur over many frames. To quote the UnrealEngine documentation of latent functions: *Latent functions let you manage complex chains of events that include the passage of time*. Because with latent function you cait wait for any amount of time before doing what you want, and you don't have to worry too much about your function being too slow because it won't freeze the game and instead execute over many frames/seconds.

The only downside of such functions is that only an entry function or a latent function can call another latent function. But we'll see later how to work around this and adapt the way we write our code to use the full power of latent functions.

Technical explanation: You can skip this part if you do not want to understand exactly how the latent functions work.

Most game engines use two or more threads for their games to keep everything smooth. The first thread, often called the *rendering thread* has one job: send the frames data to the gpu as frequently as possible. While the other threads to the heavy lifting, they do all the calculation needed for the game and update the data used by the rendering thread.

This technique was created to keep a high framerate even when there are lots of moving parts in the game. Even if a function is slow it won't delay the next frame or even worse freeze the game, instead the rendering thread will continue updating the images while the other threads take their time to update the data. This way the users keep their 60 frames per seconds even in complicated scenes.

So what is a latent function? It is a function that comes with his own thread. Because when you call a simple function that is not latent, high are the chances it will be executed in the rendering thread. And doing things in the rendering thread is not a good idea, unless you want to impact your users' framerates. Of course not each latent function comes with one thread, whenever you call a latent a function from another latent function both will use the same thread.

Obviously we can't know exactly how the red engine works, but after seeing functions like

```
// Kill internal state thread ( LATENT and ENTRY only )
import latent function KillThread();
```

we can deduce it works like any other game engine in the world.

3.4.2 Why use a latent function

When inside latent functions you can call other latent functions and the engine offers two interesting functions you can use when developping your mods:

- `Sleep(seconds:  float)`
- `SleepOneFrame()`

These two functions allow you to control the game over the time. You can tell to do one action, wait 10 seconds and do the same action again. You should also prefer doing the heavy work in latent function to avoid freezing the game.

So my advice would be to use latent functions whenever possible. But the amount of code you have to write to be able to use a latent function may sometimes be not worth the effort, if you're writing a small and supposedly fast function you may not need to write twice the amount of code for it. It's up to you at this point.

3.4.3 How to use a latent function

In order to be able to use call a latent function you must either be in an entry function or another latent function. So in short you must know how to create entry functions to be able to use latent functions.

So what's an entry function. An entry function can only be inside a state of a statemachine class. Imagine the following statemachine class:

```
// statemachines should extend CEntity to gain access to useful methods
statemachine class StateMachineExample extends CEntity {
 public var delay: int;
 default delay = 10;

 public function start() {
  this.GotoState("ExampleState");
 }
}
```

Once you're in a statemachine you can tell your class to enter in a `state`, in the example we can see the class enters the `ExampleState` state. Once you've done that you can write your state like so:

```
state ExampleState in StateMachineExample {
 event OnEnterState(previous_state_name: name) {
  this.entryFunctionExample();
 }

 entry function entryFunctionExample() {
  this.latentFunctionExample();
 }

 latent function latentFunctionExample() {
  Sleep(parent.delay); // access the parent's variable
 }
}
```

A state is declared like a class, but with the `state` keyword instead. It acts almost like a class, meaning you can add attributes to the state and you can even extends another state.

In a state everything starts from the event `OnEnterState` that is called everytime the statemachine enters this specific state. And from this event you can call the entry functions you declared, and then the latent functions.

As you can see it is not **that** complicated to use latent functions. But the overhead of writing a statemachine and using states simply to use a latent function may not be worth the hassle. Note that many latent functions available in the scripts come in the synchronous version too, but these function will cause the game to freeze while they execute. A good example are the two functions `function LoadResource` and `latent function LoadResourceAsync`, both do the same thing but one is synchronous and freezes the game while the game loads the resource from the disk (a good second depending your disk speed) while the other is latent and doesn't cause any hiccup.

3.5 Importing classes

Importing classes is a very important but very tricky feature in WitcherScript. The problem is that you will have conflicts if you import a class multiple times so its recommended to use the SharedImports: https://www.nexusmods.com/witcher3/mods/2110? mod to resolve these.

3.5.1 Importing a class and using it in a Script

- Run the game with -dumprtti and it will generate an xml file called rttidump.xml
- Create a ws file and inside that follow the guidance of the xml.

Example:

```
import class CParticleSystem extends CResource
{
    import var previewBackgroundColor : Color;
    import var previewShowGrid : Bool;
    import var visibleThroughWalls : Bool;
    import var prewarmingTime : Float;
    import var autoHideDistance : Float;
    import var autoHideRange : Float;
    import var renderingPlane : ERenderingPlane;
}
```

After that you can use it like so:

```
exec function w2p()
{
    var particle : CParticleSystem;
    particle = ( CParticleSystem )LoadResource(
↪"characters\models\common\special\demon_horse\flies.w2p", true );
    theGame.GetGuiManager().ShowNotification("Loaded:␣
↪characters\models\common\special\demon_horse\flies.w2p" + "<br>" +
     "renderingPlane: " + particle.renderingPlane + "<br>" +
     "previewBackgroundColor: " + "<br>" +
     "   Red:" + particle.previewBackgroundColor.Red + "<br>" +
     "   Green:" + particle.previewBackgroundColor.Green + "<br>" +
     "   Blue:" + particle.previewBackgroundColor.Blue + "<br>" +
     "   Alpha:" + particle.previewBackgroundColor.Alpha + "<br>" +
     "previewShowGrid: " + particle.previewShowGrid  + "<br>" +
     "visibleThroughWalls: " + particle.visibleThroughWalls  + "<br>" +
     "prewarmingTime: " + particle.prewarmingTime  + "<br>" +
     "autoHideDistance: " + particle.autoHideDistance  + "<br>" +
     "autoHideRange: " + particle.autoHideRange  + "<br>" +
     "renderingPlane: " + particle.renderingPlane);
}
```