Preface

This mod is a tool made to allow easy creation of custom cutscenes within Telltale's game engine. This tool only works with The Walking Dead: The Telltale Definitive Series. It does **not** work in other games. You need to own a valid copy of The Walking Dead: The Telltale Definitive Series (TWDTTDS) in order to use this tool.

This tool uses custom **.ini** formats, as well as **.lua** scripts, to save and load your work. Therefore, this tool cannot be used to modify any existing cutscenes made by Telltale, which use their own internal formats. You can only create new cutscenes using Telltale's resources, not edit any existing Telltale cutscenes.

This tool is meant to be used by advanced users - users who know their way around computer, know the difference between files and folders and can understand basic computer-related terminology (and can look terms up if they are unfamiliar with them). Creation of simple cutscenes does not require coding knowledge beyond following the steps of scene setup. More complex features will require writing some new, more complex code, though nothing too difficult. In general, knowledge of the basics of coding will make understanding of this tool much easier, though it is not required. Telltale scripts are written in **lua**, so if you have knowledge of that language specifically, it's also a plus. I hope to make this tool even less code-knowledge dependent in future updates.

Please note that this is an experimental tool, meaning there is a high probability of running into glitches and problems of different kinds. There are also many strange design choices, many of which come from limitations of Telltale's scripting and from incomplete knowledge. Many features of this tool may appear unfinished, half-baked or difficult to work with. I plan to improve and simplify this tool further in future updates, though my free time is limited and I do not know when the updates will be released.

Please note that the inner workings of Telltale's engine are not fully known to us, which means this tutorial may contain incorrect, misleading or unclear information. I will try to describe my understanding of how to work with the engine to the best of my knowledge and ability. But I am well aware that my knowledge is incomplete.

I advise you to make frequent backups or use version control when working with this tool due to a high potential of permanent data loss.

Support for this tool is done through Discord. Please join the Telltale Modding Server (https://discord.gg/HqpnTenqwp), go through the verification process, then open the **Cutscene Editor** thread in the **Mods** section, write down and post a detailed report about your problem or question, and ping **Mawrak** alongside it. I will try to respond

as soon as I am able. Please make the bare minimum of effort to try to make the mod work yourself before contacting me. I'm happy to help you with understanding of this mod, but I expect you to at least be familiar with this tutorial and have a concrete question in mind.

This is a PC-only and Windows-only tool. This will not work with console or mobile versions of the game. Building from Source is not supported in non-Windows operating systems. This tool should be fully compatible with Windows 7 and above.

For the purpose of working with this tool, it is highly recommended that you disable the "Hide extensions for known file types" in your system. In Windows 7, this can be done in the Folder Options settings of your system. In Windows 8/10, this can be done through the explorer itself. This will enable you to see and edit file formats known to system.

This mod is meant to be used when the game is running at **60 FPS**. If the game is locked at a number below 60 FPS, you may run into issues such as animation stutters, incorrect placement of objects in character hands. Please make sure your game is not locked to a number below 60 FPS when using this mod.

This tutorial is built around describing the inner workings of an example **Demo Scene** made with this tool. It will explain how this scene was made and use it as reference to show what can be made. It will also show you how to create the building blocks of your cutscene in similar fashion. Please note that the Demo Scene is only meant to be used as an example, and through understanding its code and structure you are meant to learn how to make your own, entirely new scenes.

The **Demo Scene** is built upon **Season 3** of the Walking Dead. Season 3 has been tested the most with this tool. I have tested that **Season 2**, **Season 4** and **Michonne** launch as well, and added basic scenes for each of them. However, it is possible there are some unresolved issues involved with working in those seasons. If you run into them, don't hesitate to contact me. And unfortunately **Season 1** has too much of a difference in scripting API compared to other seasons, and is not supported at this time (though you should be able to load its assets in Season 2 environment instead).

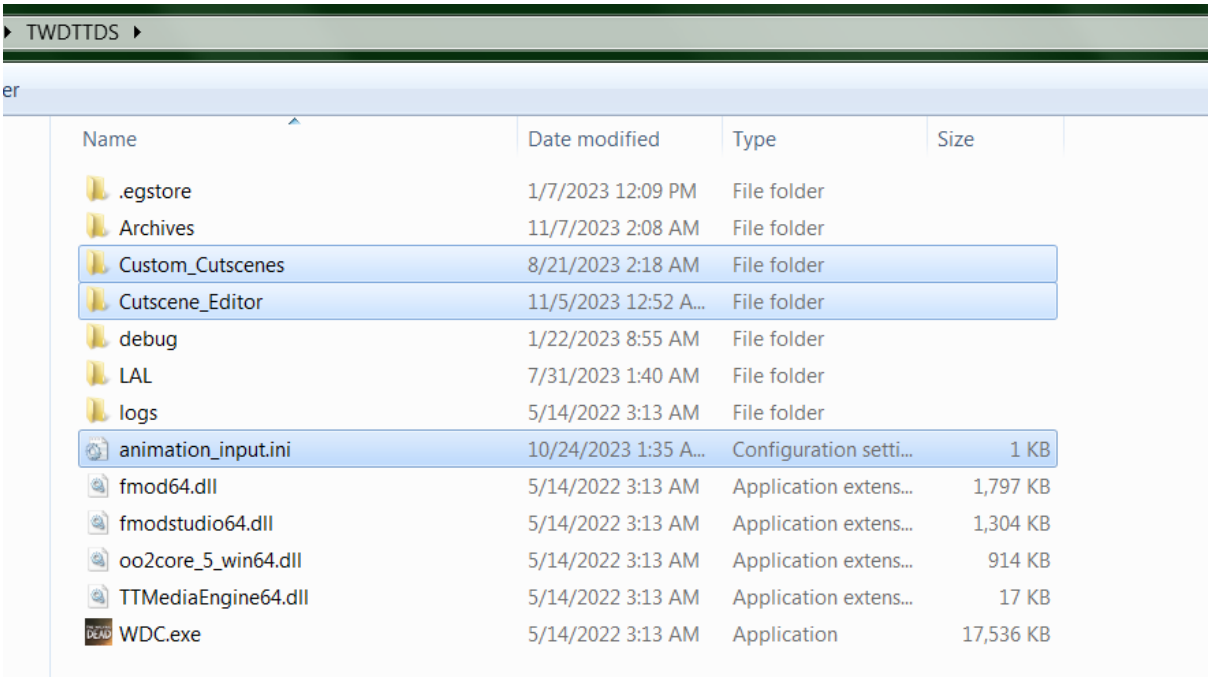
Last but not least, I am not 100% sure how loading different scenes with custom cutscenes may affect save files. Using a clean/unimportant save file is recommended when working with this tool, or when playing mods made with this tool.

Table of contents

Installation and Usage

When you download this tool, you will see that it contains two main components: **Build** and **Source**. Build contains an example **Demo Scene** ready to be installed into the game. Source contains the source code for the **coding side** demo scene, as well as everything else you would need to make your own cutscene.

If you just want to run the Demo Scene and see it in action, you only need to install the **Build** component. To do that simply put everything from the Build folder inside the game's **Root** folder, so it looks something like this:



Then you can simply run the game and launch the scene from the main menu.

If you wish to modify the Demo Scene or create your own cutscene, you will need both the Build and the Source components. Install Build the same way as outlined above. Then put **cutscene_editor** folder (the folder inside the **Source** folder) somewhere else safe (**do not put it into any of the game's directories or subdirectories!**). Then carefully read the rest of this tutorial to learn how this tool functions and how to use it to the full extent.

I recommend running the current **Build** in game a few times to see how it works and what elements does it have, and then moving on to working with **Source**.

Required and Recommended Software

Besides the game itself, working with Cutscene Editor requires you to install several other programs.

Required programs

1) **Telltale Script Editor + Tweaks v3b.1.0 or higher** - this is the main program that allows building .lua scripts into the game.
2) **Notepad++** - this is an excellent tool to edit text files, it is required for editing the .ini files used in this mod. **Do not** use regular Notepad, it can result in problems.

Recommended programs

1) **Telltale Explorer v1.3.3** - can be used to open Telltale's archives, preview and extract assets. **Note:** Explorer's .png texture extraction is broken, use .dds or raw extraction instead.
2) **ttarchext v0.3.2** - another useful assets extractor, allows easy bulk-export of files with desired formats through .bat files. The ttarchext.exe file is included with the Script Editor, and examples of usage are given in Finding animation names section.
3) **DDS-D3DTX-Converter vCLI Beta 2.0.1 or higher** - allows texture editing. **Note**: disregard tutorials on this program's github page, they are extremely outdated and will mislead you.
4) **Telltale Music Extractor v1.5.7** - program that allows you to extract music files and other sound effects from the game. Can be useful in getting resources to create custom sounds. Many music and sound effect audio files are stored as .bank soundbanks, this tool can be used to extract them.
5) **Telltale Speech Extractor v1.4.3** - program that allows you to preview and extract dialogue audio, preview subtitles for each line. Can be useful in figuring out the names of dialogue audio (and animation) files if you plan to reuse them in your cutscenes.
6) **Audacity** - best audio editor in the world, can be used to create custom sounds for your cutscene.
7) **The Telltale Inspector** - an amazing new tool developed by **Lucas Saragosa** which allows you to edit properties of .prop, .scene, .d3dmesh, .d3dtx files, extracting .bank soundbanks and more.

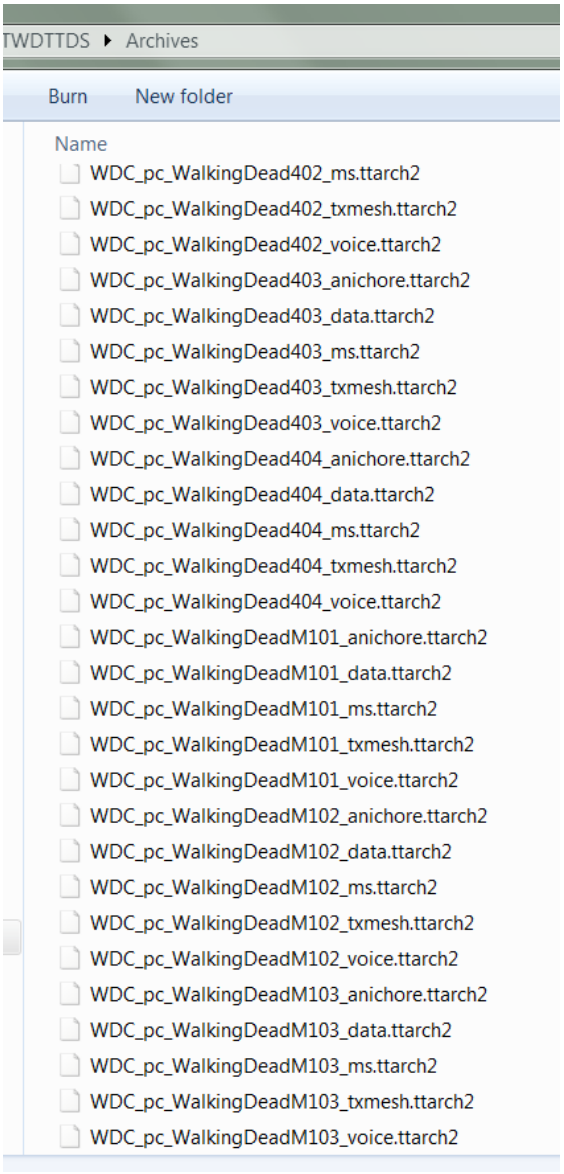
Please **do not** use versions of these programs below specified. Using outdated versions is not supported and I can confirm you will run into problems.

DO NOT launch the game while Telltale Explorer, Telltale Music Extractor or Telltale Speech Extractor are running!!! This will cause issues such as infinite loading screens, missing assets, graphical glitches, etc. This happens because these tools prevent the game from loading necessary archives. **Please make sure to close these programs before running the game and testing your scenes.**

Telltale File Structure

Modding basics

Telltale assets are normally stored inside the **.ttarch2** archives within the **Archives** folder.

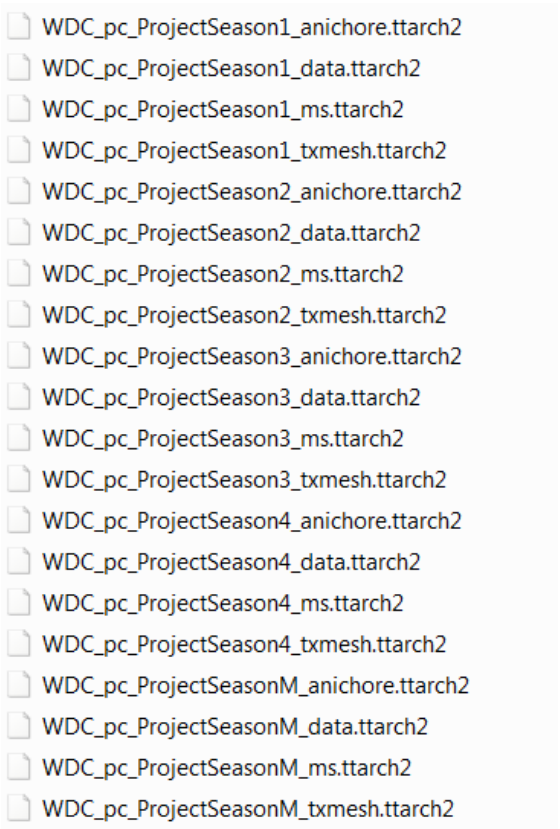


The archives that are most interesting to us are either **Episodic archives** (contain data) or **Season archives** (contain data available for an entire Season).

Episodic archives usually start with **WDC_pc_WalkingDead** in their name, followed by an **Episode number** and a **tag**. Episode number indicated resources for which episode are held in this archive. For example, 404 for S4E4, 201 is for S2E1, M103 for Michonne E3, etc. The tag shows what kind of resources can be found in this archive:

- **_data** archives contain **.prop** and **.scene** files, as well as **.skl** files and many other important resources.
- **_ms** archives contain music and other audio (use Music Extractor for extraction).
- **_txmesh** archives contain meshes and textures (**.d3dmesh** and **.d3dtx**).
- **_voice** archives contain voice audio used in dialogue.
- **_anichore** archives contain animations (.anm) and .chore files.

The **Season archives** are similar, except that they start with **WDC_pc_ProjectSeason**, followed by a **Season number** and a **tag**. They contain similar types of data as the **Episodic archives**. There are many other archives there such as **UI** ones (start with **WDC_pc_UISeason**) which may also be of use.



Usually these archives are loaded in only when the game called for it. For example, data inside **WDC_pc_ProjectSeason2_txmesh.ttarch2** file will only be loaded when you are playing Season 2. Data inside **WDC_pc_WalkingDead302_data.ttarch2** file will only be loaded when you are playing Season 3 Episode 2. Etc. Through scripting it is possible to load the desired **Episodic archives** into your scene, giving you access to the assets inside. It is also possible, and often desired, to extract assets one by one and then load them into the game separately.

Most resources can be extracted with Telltale Explorer, Music Extractor, Speech Extractor or ttarchext. Raw extracted resources, as well as any custom assets you create, can be placed inside the **Archives** folder. Any raw asset placed inside that folder will **always** be loaded by the game no matter what episode or scene you are on. In addition, they will load on top of any files with the same name and format located inside any of the default Telltale **.ttarch2** archives, overwriting their appearance in the game with their own.

It is also worth pointing out that the game is capable of loading such assets even from its **Root** folder, or other subfolders. Sometimes you may be able to avoid using the **Archives** folder, though there are some exceptions: in my experience, for example, **.wav** files related to voice audio and their associated **.anm** files associated with lip sync animations do not load from folders other than **Archives**. So I would still recommend sticking with the **Archives** folder if you are going to use this method of file loading. It is still worth keeping in mind as a possibility - you should not keep unused assets inside the game's main folder or its subfolder to prevent unwanted asset interference!

The method described above is most suitable for model or texture (or any other asset) swapping, as the extracted **raw files** will always be prioritized by the game over packed in files from **.ttarch2** folders. However, for the purpose of creating custom cutscenes I recommend going a different route and packing your assets inside your own **.ttarch2** archives alongside your scripts.

To do that you can put the assets into one of the **Source** folders inside your project (**Demo_Assets** in case of the **Demo Scene**), and then Build it from Source. For the sake of this tutorial, we will only be using this method going forward, as it creates a much more compacks and easy to install **Build**: custom **.ttarch2** archives created with the **Script Editor** do not need to be put into Archives folder, they and their assets will be loaded freely from any subfolder inside the **Root** folder. You can read more about what files are stored in **Demo_Assets** folder in the Demo Scene File Structure section.

Common Telltale formats

The common asset formats that can be used in cutscene creation are listed below.

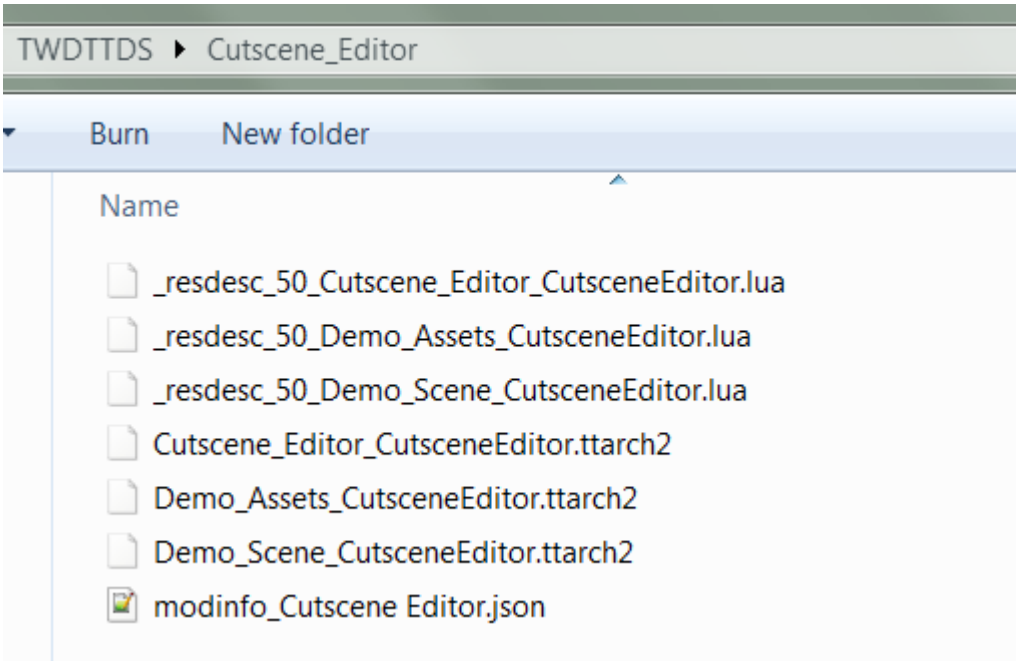
- **.anm** - animations. Animation files can often be imported from other Telltale games and will function normally in TWDTTDS. *As of writing this tutorial, these files cannot be edited.*
- **.wav** - audio. Raw extracted audio files are not playable in normal players, however they can be exported as playable files with **Speech** and **Music Extractors**. Regular .wav files can also be played back in the game, meaning you can easily create your own audio files with **Audacity**.
- **.d3dmesh** - mesh files. These are the building pieces of all 3D models you see in the game. They are also used as building pieces of the Ui. They also include references to necessary textures and materials. *As of writing this tutorial, editing of these files is limited.*
- **.skl** - skeleton files. These contain information about character bones used for playing animations. Every character has an .skl file associated with them. *As of writing this tutorial, these files cannot be edited.*
- **.d3dtx** - textures. These are image files that contain images seen on any 3D object or UI element. Can be converted into **.dds** and edited, and then converted back with **DDS-D3DTX-Converter vCLI Beta 2.0.1**. Creating entirely new textures from scratch isn't supported since every texture has an embed profile which is extracted as a corresponding **.json** file with the same name. To create an entirely new texture you should edit an existing one, then convert it back and then rename the resulting .d3dtx file into something unique. Converting edited .d3dtx back into .dds is not supported, so you should always keep your edits in .dds format as backup. Please pay attention to correct .dds formats during saving and converting.
- **.prop** - a file that contains references and other information used internally by Telltale's engine to create objects. May contain all sorts of different objects, ranging from characters and weapons to UI elements to lighting. In cutscene editor, these files are used to spawn in new objects and characters into the scene.
- **.scene** - a special type of .prop file containing all objects in a given scene. These files are necessary for loading scenes in-game.
- **.chore** - files that include various action sequences used in Telltale's native cutscenes. Usage of these files is not covered by the **Demo Scene**, but you can see Advanced Features for more info about them.

Demo Scene File Structure

Build

The **Build** contains three folders and an .ini file. The **animation_input.ini** file is used during editing process inside the game itself, see Editor Controls for more info.

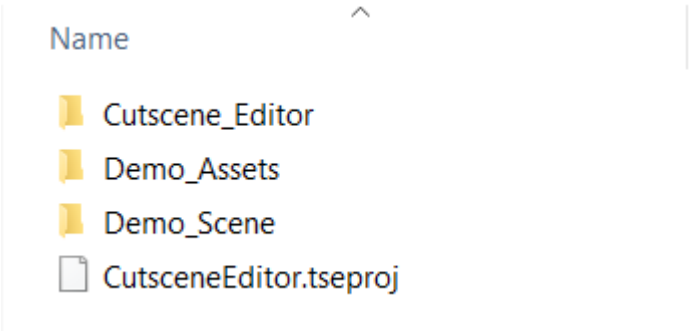
The **Cutscene_Editor** folder contains compiled archives with scripts and other assets you put inside of them, created during the building process by the **Script Editor**.



Custom_Cutscenes folder contains user-created folders containing **ini**-realted cutscene data (this is the data that is created by using the in-game Cutscene Editor, see more in Editor Controls and Features section). The **demo_cutscene** holds the **Demo Scene** data and clips. Other folders here have very basic examples for scenes from other supported seasons, they will not be covered extensively in this tutorial, but it should be pretty clear how they work if you understand the Demo Scene. They are meant to be used as a possible base for your projects. You can create your own folders here to organize your own cutscenes (see Demo scene code section).

Source

If the **cutscene_editor** folder inside **Source** is the project folder meant to be loaded with the **Script Editor**. If you look inside, you'll see that it contains three folders and a **.tseproj** file.



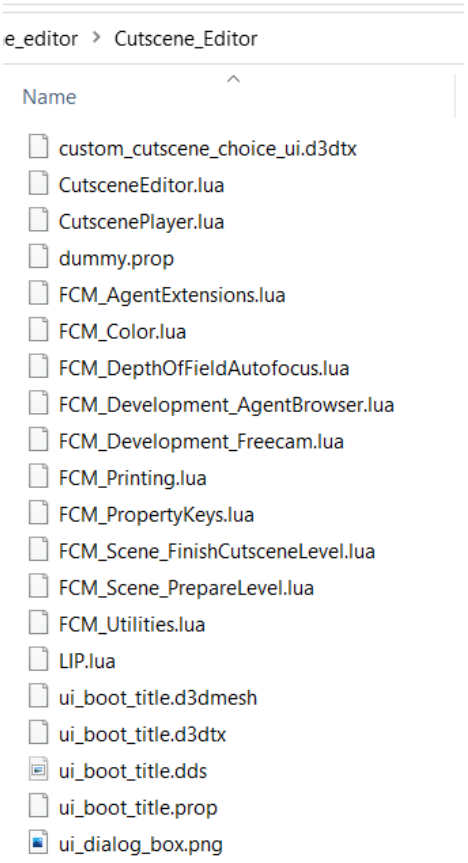
The **CutsceneEditor.tseproj** file contains some basic information about the mod. Its main purpose is to let the script editor know that this is a **project** folder. The information inside of the file is purely cosmetic and doesn't really affect anything (*Priority setting is unused in the current (as of writing this tutorial) version of the Script Editor, the Script Editor will always set archive priority to 30 no matter what to prevent conflicts with existing Telltale menus*).

If you paid attention in the previous section, you would notice that the folder names correspond to the names of the files inside **Cutscene_Editor** folder of **Build**. Indeed, the **.ttarch2** archives inside that folder contain the assets from the corresponding **Source** folders inside of them. When you are building your mod from Source, these archives will be replaced by new ones (provided you selected **Cutscene_Editor** as your mod path folder during Script Editor Setup, otherwise the files will be created in whatever other folder you select).

Separating **.lua** files and other **assets** between different folders/archives isn't really necessary, the game will still read them all together at the same time (meaning you **should not** make files with identical names in two different folders). But to avoid making a mess and to allow easier usage of the tool, I separated them into three different folders (which results in three different archives in **Build**) based on their purpose.

Cutscene_Editor folder (the one inside **Source!**) contains data related to the inner workings of the **Cutscene Editor** and **Cutscene Player** components of the tool. Files inside this folder include **CutsceneEditor.lua** and **CutscenePlayer.lua** scripts which are responsible for their respective components, **FCM scripts** which contain helpful pre-made functions actively used in this tool (these files are modified versions of similar scripts made for First Cutscene Mod), **LIP.lua** which is an ini-parser and various other assets used internally by the tool. You can think of this folder as a **Library folder**, its contents are not meant to be accessed by regular users of this tool, the inner workings of these files are quite complex and are NOT covered by this tutorial. You are free, however, to modify them as you please at your own discretion, and the functions inside may be used as references for your own scripts. I am also

ready to explain how these files work on Discord to help you understand them, if you are interested in learning about how this mod actually works.



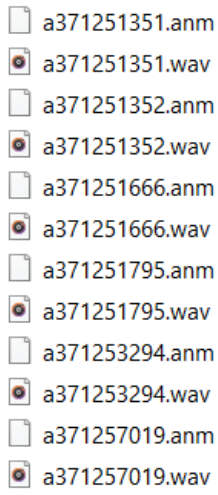
The **Demo_Assets** folder contains various assets used specifically by the **Demo Scene**. If you are making your own cutscene with this tool, you should probably clear this folder out and put your own assets inside of it, if you need. Like I mentioned in Modding basics, the game loads assets from **Episodic archives** when they are called by a specific episode. Through scripting, we are able to load any **Episodic archives** we wish into our scene. In this **Demo Scene** example, all available Season 3 assets will be loaded (Episodes 1-5), meaning that we do not need to extract them to actually access them in-game. However, this scene also uses a lot of animations from other Seasons (mostly Season 4), as well as a custom texture for the UI and many custom sounds and music that cannot be accessed through Episodic archives. For the sake of this example, these assets were put into the **Demo_Assets**, meaning they will be packed in with the mod during building, and will also be available to reference during cutscene creation.

Certain audio assets are extracted or edited files from the original game. I added a prefix to their names in order to indicate which seasons they come from and to prevent file conflicts. Some speech audio was made using audio editing and splicing existing lines to create new ones. These files have a "c" at the end (most fitting lip

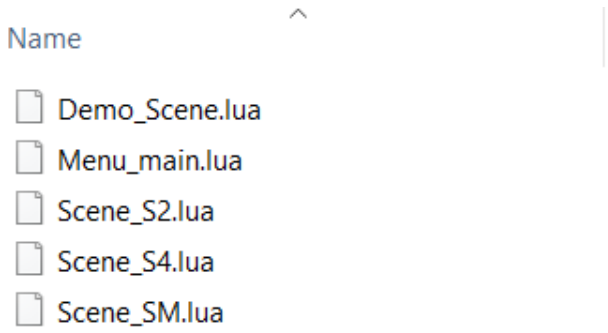
sync animations were also given the same names in order to play them together in the cutscene).

There is also one type of files that cannot be properly loaded in from **Episodic archives** through this tool due to some kind of internal Telltale scripting glitches. These are the audio files related to speech in dialogue (voice acting), and the corresponding lip sync animation files. These files have a unique naming convention of being a number. The animation files related to said dialogue line have the same name as their audio counterparts, so they are pretty easy to find. The problem is that for whatever reason the internal Telltale commands to play audio and animation do not properly recognize number-only file names if they are fed into them through a variable (and they cannot be fed differently in this case because this tool is meant to be universal and work with any file the user wants to use in their cutscene). I suspect this is because they read the variable as a float instead of a string, even if you specify that it's a string. This results in some files not playing or playing incorrect animation/audio combinations.

The only solution I found is to extract these audio and animation files and then rename them in a way that would leave no doubt about their names being a string of text (by adding letter '**a**' at the start). Therefore, all voice audio used in the demo and all lip sync animations are extracted, renamed and stored in the **Demo_Assets** folder. This is something you will likely have to do yourself for your own work as well.



Finally, the **Demo_Scene** folder contains script files responsible for loading scenes to run the custom cutscenes themselves. There are five files in this folder.



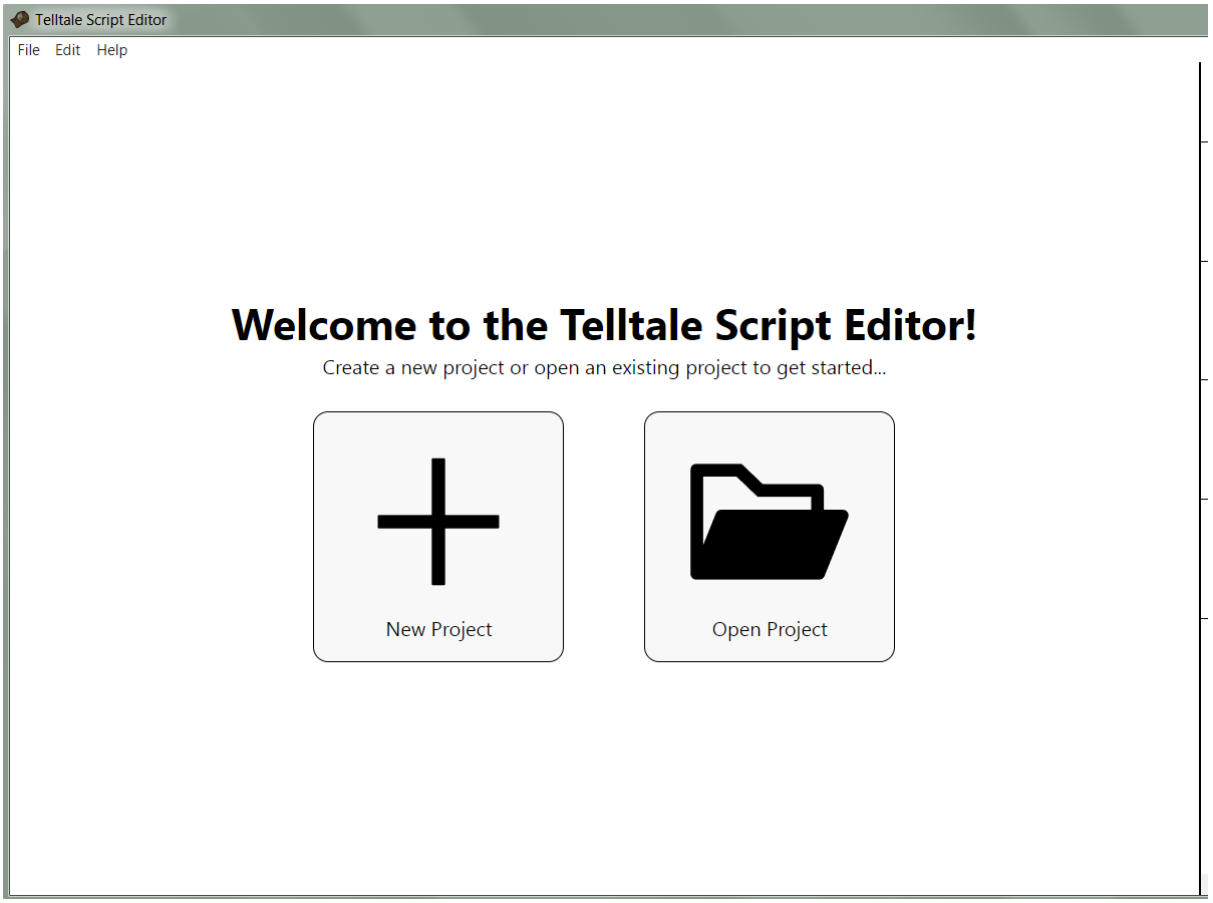
Demo_Scene.lua is used to run the **Demo Scene**, which runs in Season 3 environment.

Menu_main.lua is a main menu replacement used to add a button which launches your scene from the main menu of the game. This file will overwrite the default main menu file due to sharing the same name. This file also includes built-in compatibility support for **Load Any Level**, or **LAL** for short. It's a mod that lets you load any vanilla scene/cutscene from the main menu, it's often used for debugging by modders, though I would recommend using a clean save file if you plan to mess around with it due to the potential of save data loss. It doesn't have a lot of uses in regards to **Cutscene Editor** specifically, but this **.lua** file allows you to install **LAL** without running into conflicts.

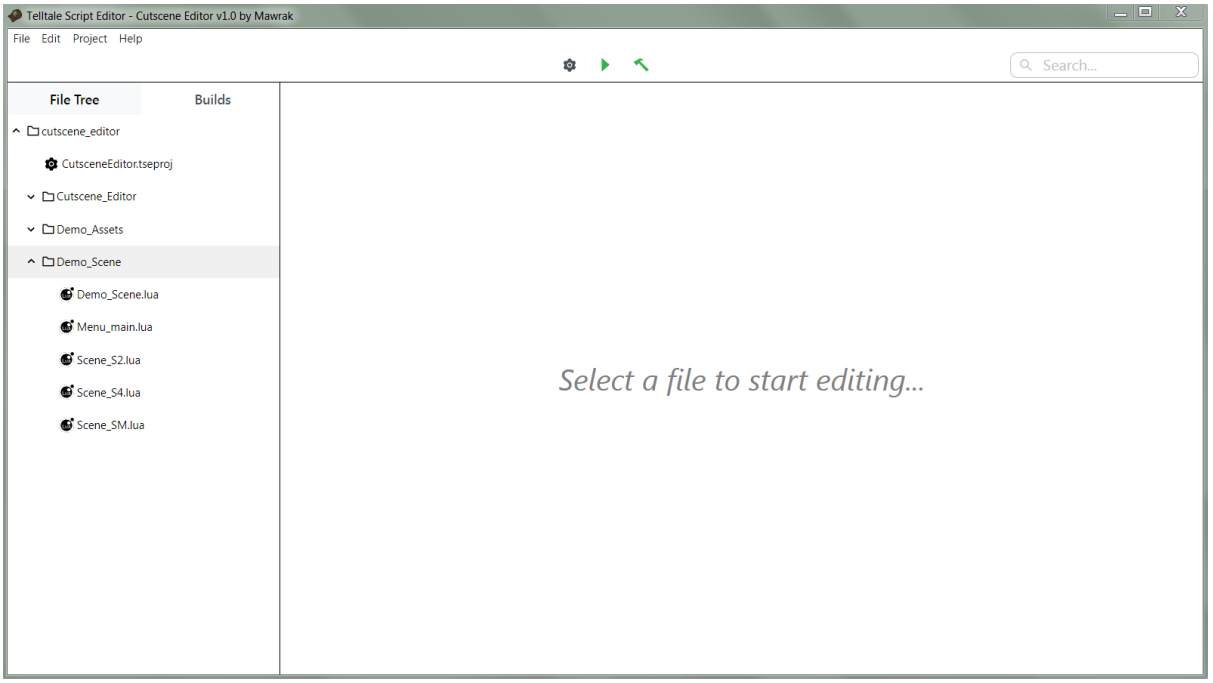
Other **scene files** are sample scenes that demonstrate the possibility of playing cutscenes in other seasons as well. They do not have any demos attached to them and are currently inaccessible in **Build** (though you can change them by adding the corresponding buttons to **Menu_main.lua**) but they can be used as a starting point to make your own scene.

Script Editor Setup

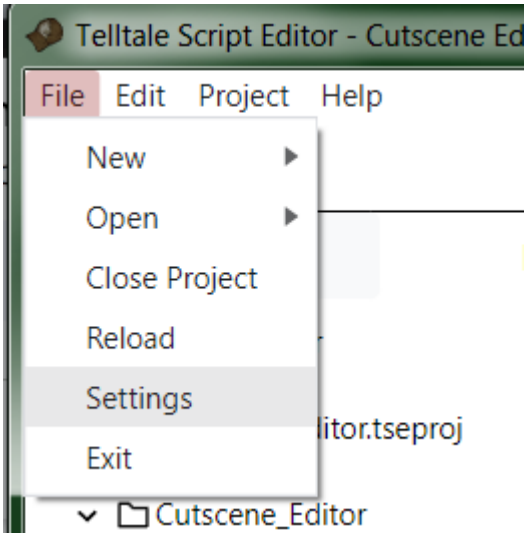
Before you start working with the mod files, you need to set up certain options inside the **Script Editor**. Extract the script editor and run **telltale-script-editor.exe**.



Click Open Project and navigate to the **cutscene_editor** folder (the one that comes from **Source**). Select the **CutsceneEditor.tseproj** file. Now you'll be able to see the project's folder structure and files.

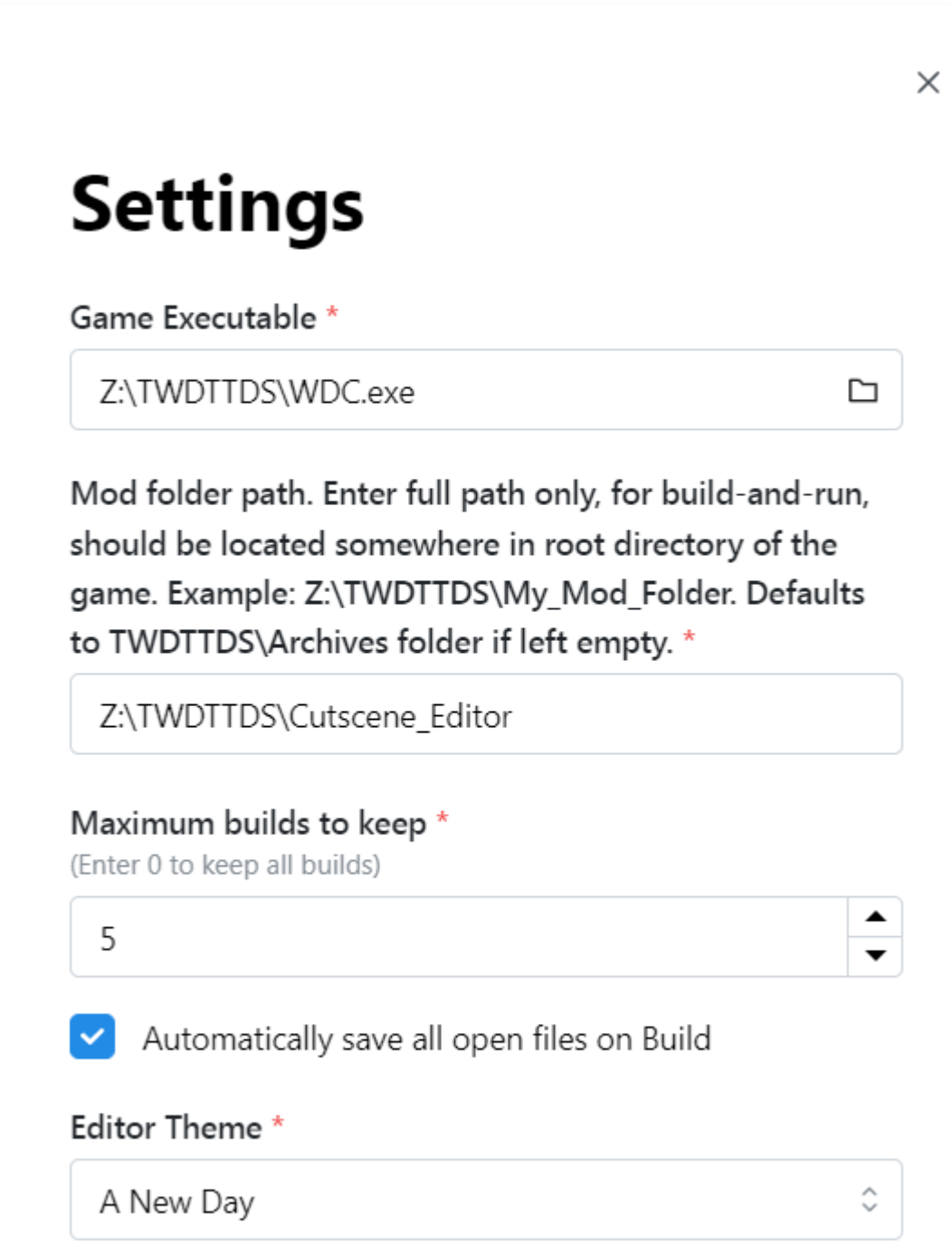


Open **File -> Settings** menu.



Here you should do two things. First, set the path to your game's executable file. Then set the **Mod folder** path to the **Cutscene_Editor** folder inside your game's root folder (this folder should be there if you installed the **Build** correctly).

Here is an example of how your settings may look like (this is just an example, your exact folder paths will likely be different):



Technically it is possible to set the mod folder to any folder you want, then during the build process the mod files will be put there instead. But since the **Build** of the **Demo Scene** uses that folder to keep the mod archives, for the sake of this tutorial we will be using that folder to keep our mod files.

Demo Scene Setup

Basics of Telltale scripting

Now that we have our **Script Editor** all set up, we can move on to actually making scripts. But first, some important concepts must be understood.

The way Telltale's scripts work is that the game will always execute a single stream of code. You can write regular **.lua** functions and set up variables just fine. However, if you want to apply a certain setting or event to an object, you will need to find and reference that object in your code. Code cannot be executed based on signals or object interactions (unless you write your own system for such executions from scratch).

Another limitation is inability to properly pause or time execution of your code. If you try to use standard methods, you will likely run into trouble (though any experimentation is welcome, perhaps you can find a method that works which we didn't come up with). However, it is possible to add a function to **OnPostUpdate**, making it continuously execute every single frame (if you are familiar with gamedev, you can think of this as an equivalent of ***Update*** or ***Process*** function). This allows you to create timers and execute actions when you need them executed. You can read more about how to do that in Advanced Features section.

Concept of agents

Objects in the scene are called **Agents**. Naturally, creating a cutscene involved making changes to the parameters of these agents or even creating new ones. There are some important concepts to understand about Agents before we can proceed.

First of all, every agent in a scene has a unique **Agent name**. This name comes in a form of a string, and the same name must not be assigned to two different agents. Using this string it is possible to make many different interactions with agents.

Secondly, when creating new agents, you can also set up a **Variable name**. It is a variable assigned to the agent you just spawned. For many agent-related functions and interactions you can reference this **Variable name** in place of the **Agent name** if you wish, though some will require you to use the **Agent name** specifically.

You can find more information about how to find, set or use Agent names in Demo scene code and Advanced Features sections.

Demo scene code

This section is going to cover the basic scene set up done for the **Demo Scene** in **Demo_Scene.lua**. This set up involved both essential and scene-specific items of the code. The scene-specific parts of code are related to either custom functions or other scene-specific stuff like character set up, they can and should be removed if you plan to make your own scene. If you want to see how the code looks with ONLY the essential parts (the bare minimum you need to get a scene running), you can check the other **Scene** lua files for comparison.

Before we start looking at the code itself, I will describe some key info about Script Editor's controls and about how the game reads the code.

You can use ctrl+F to search for key words through the code if you need to. You can use "--" to make comments or comment out parts of code, making the script ignore them. You can save your data with ctrl+S, or by clicking the save button located on the file tab.

When the game is running, it will execute events in the code one after another. If a line of code throws an error, it may break the entire execution of your code, preventing all the future lines from being executed. Debugging your code can be quite frustrating, but you can use pcall command for debugging purposes, or to prevent code from breaking if an error appears.

Dependencies

This is an essential part of scene set up.

Any scene script will start with setting up script dependencies.

```
require("FCM_Utilities.lua");
require("FCM_AgentExtensions.lua");
require("FCM_Color.lua");
require("FCM_Printing.lua");
require("FCM_PropertyKeys.lua");
require("FCM_Development_Freecam.lua");
require("FCM_Development_AgentBrowser.lua");
require("FCM_DepthOfFieldAutofocus.lua");
require("FCM_Scene_PrepareLevel.lua");
require("LIP.lua")
local LIP = require("LIP.lua")
require("CutsceneEditor.lua");
require("CutscenePlayer.lua");
```

Normally you would want to leave this section as is, it already includes everything you need.

Script variables

This is an essential part of scene set up.

Next part includes setting up important script variables.

```
--main level variables
local kScript = "DemoScene"; --(the name of the level function which will be called as soon as this scene opens)
local kScene = "adv_richmondStreet"; --(the name of the scene asset file)
local agent_name_scene = "adv_richmondStreet.scene"; --(the name of the scene agent object)

--cutscene development variables variables (these are variables required by the development scripts)
Custom_CutsceneDev_SceneObject = kScene; --dont touch (the development scripts need to reference the main level)
Custom_CutsceneDev_SceneObjectAgentName = agent_name_scene; --dont touch (the development scripts also need to reference the name of the scene agent)
Custom_CutsceneDev_UseSeasonOneAPI = false; --dont touch (this is leftover but if the development tools were implemented inside season 1 we need to use the S1 functions because the api changes)
Custom_CutsceneDev_FreecamUseFOVScale = false; --dont touch (changes the camera zooming from modifing the FOV directly, to modifying just the FOV scalar (only useful if for some reason the main field of view property is chorelocked or something like that))

--cutscene variables
local MODE_FREECAM = false; --enable freecam rather than the cutscene camera (better leave this as false and change in the main function itself)
agent_name_cutsceneCamera = "myCutsceneCamera"; --cutscene camera agent name
agent_name_cutsceneCameraParent = "myCutsceneCameraParent";
--cutscene camera parent agent name

--hides the cursor in game
```

```
HideCusorInGame = function()
    CursorHide(true); --hide the cursor
    CursorEnable(true); --enable cusor functionality
end
```

When making your own scene, you will need to fill out these variables according to your needs.

kScript contains the name of the function that you wish to execute at the start of your cutscene (we'll refer to it as **Scene Function** from now on). In this case it's called **DemoScene**, and we will cover how this function works below. If you change that function's name, you should also change this variable.

kScene contains the name of the **.scene** file you wish to run. The **.scene** files can usually be found within the **_data** archives of the game (see Telltale File Structure). **kScene** is very often used in Telltale's scripting commands so it is easier to set it up as a variable at the start. In this case, the "**adv_richmondStreet**" scene (Richmond streets from S3E2) is loaded. If you want to use a different scene as a base for your cutscene, change this variable.

agent_name_scene should contain the same name as **kScene**, but with a .scene format at the end.

The rest of these variables should normally be left untouched. It is important to note that **Custom_CutsceneDev_SceneObject** variable equals **kScene** and can be used as an equivalent in the same functions (this is a leftover from **FCM scripts**). The **MODE_FREECAM** variable controls whether you want to use **Free camera** or **Cutscene camera** in this build. I recommend leaving it as **false** here and change it during the **Scene Function** itself.

Camera set up

This is an essential part of scene set up.

Next we have a function which is responsible for **Cutscene camera** set up.

```
Cutscene_CreateCutsceneCamera = function()
--generic camera prop (prefab) asset
local cam_prop = "module_camera.prop";

--set a default position/rotation for the camera. (in theory this
doesn't matter, but if the script somehow breaks during update the
camera will stay in this position).
```

```
local newPosition = Vector(0,0,0);
local newRotation = Vector(0,0,0);

--instaniate our cutscene camera object
cameraAgent = AgentCreate(agent_name_cutsceneCamera, cam_prop,
newPosition, newRotation, kScene, false, false);
local cameraParentAgent =
AgentCreate(agent_name_cutsceneCameraParent, "group.prop",
newPosition, newRotation, kScene, false, false);

AgentAttach(cameraAgent, cameraParentAgent);

--set the clipping planes of the camera (how close the camera can
see objects, and how far the camera can see)
--if the near is set too high we start loosing objects in the
foreground.
--if the far is set to low we will only see part or no skybox at
all
Custom_AgentSetProperty(agent_name_cutsceneCamera, "Clip Plane -
Far", 2500, kScene);
Custom_AgentSetProperty(agent_name_cutsceneCamera, "Clip Plane -
Near", 0.05, kScene);


--Custom_AgentSetProperty(agent_name_cutsceneCamera, "Field of
View", 50, Custom_CutsceneDev_SceneObject); --FOV correction for
S2

--bulk remove the original cameras that were in the scene
Custom_RemovingAgentsWithPrefix(kScene, "cam_");

--push our new current camera to the scene camera layer stack
(since we basically removed all of the original cameras just the
line before this)
CameraPush(agent_name_cutsceneCamera);
end
```

Generally you shouldn't touch this, but make sure to remove the comment lines on this code if you work with a Season 2 scene to prevent UI issues.

```
Custom_AgentSetProperty(agent_name_cutsceneCamera, "Field of
View", 50, Custom_CutsceneDev_SceneObject); --FOV correction for
```

```
S2
```

Scene function

Now we get to the **Scene function** (called **DemoScene** in the **Demo_Scene.lua** file). This is the code that will be executed at the start of the cutscene. Its code can be loosely separated into the following blocks.

Archives loading

This is an essential part of scene set up.

At the start of the scene we need to enable archives we wish to use. In case of our Demo Scene, we activate all Season 3-related archives, meaning that we get full access to **Episodic** Season 3 resources.

```
ResourceSetEnable("ProjectSeason3");
ResourceSetEnable("WalkingDead301");
ResourceSetEnable("WalkingDead302");
ResourceSetEnable("WalkingDead303");
ResourceSetEnable("WalkingDead304");
ResourceSetEnable("WalkingDead305");
```

Now, there are several important things to keep in mind when working with different seasons. First of all, loading the **Project resource** differs in other seasons. If you work with Season 3, then you should **always** enable **"ProjectSeason3"**. However, if you work with any other season, you should **always** enable **"ProjectSeason4"** instead. Yes, you need to use **"ProjectSeason4"** even if you are not working with Season 4, this is a limitation of this tool that may be fixed in the future. As an example, we can look at the code from **Scene_S2.lua**, a script that's meant to load a Season 2 scene:

```
ResourceSetEnable("ProjectSeason4");
ResourceSetEnable("WalkingDead201");
ResourceSetEnable("WalkingDead202");
ResourceSetEnable("WalkingDead203");
ResourceSetEnable("WalkingDead204");
ResourceSetEnable("WalkingDead205");
```

Loading an incorrect Project resource may cause very noticeable UI or other issues, so I do not recommend doing it.

I also do not recommend loading **Episodic archives** from seasons other than the scene you are working in to prevent unwanted files conflicts and overwrites. Season 3 meshes in particular have a lot of conflicts with all other seasons, and should not be used together. If you wish to use compatible resources from other seasons (animations, for example), I suggest extracting them and putting them into your Assets folder (**Demo_Assets** in this case) instead. You are free, however, to experiment with resource loading as you see fit.

Agents set up

This is a scene-specific part of scene set up and should be used as an example. It will differ from scene to scene and can be changed or removed if you work on your own cutscene.

Feel free to skip this section for now and return to it later if you don't have the skills to understand it yet.

Next up we have some agents set up. As I said before, agents are objects that can be used for a variety of purposes in your scenes. Scenes usually have many pre-made objects already, but in this scene we also spawn a bunch of objects ourselves. New objects are created using the **AgentCreate** function, which requires 7 **parameters**, or **arguments** (parameters are values you add to the function to make it do what you want). The first argument is a string containing the **Agent name**, the second argument is a string containing the **.prop** file you wish to spawn (names of these files can be found inside **_data** archives). Third parameter contains spawn position coordinates, fourth contains spawn rotation (in angles). The last three parameters should always be "kScene, false, false" no matter what.

In these lines we spawn in several characters and zombies through and give them **Agent names** (as well as agent **Variable names**). For example, in the first line here, using the "sk62_mariana.prop" file (available in S3E1 archives we enabled earlier), we can spawn in a Mariana character, we give her the **Agent name** "Mariana" (and a **Variable name** 'agent_mari'), and put her the coordinates (0,0,0) with rotation of (0,0,0). Then we do the same with other .props to spawn other characters.

```
agent_mari = AgentCreate("Mariana", "sk62_mariana.prop", Vector(0, 0, 0), Vector(0,0,0), kScene, false, false)

agent_anf_1 = AgentCreate("Rufus", "sk61_rufus.prop", Vector(0, 0, 0), Vector(0,0,0), kScene, false, false)

agent_anf_2 = AgentCreate("Eli", "sk61_eli.prop", Vector(0, 0, 0), Vector(0,0,0), kScene, false, false)
```

```
agent_anf_3 = AgentCreate("Roxanne", "sk62_roxanne.prop",
Vector(0, 0, 0), Vector(0,0,0), kScene, false, false)



agent_zombie_1 = AgentCreate("Zombie_1", "sk61_zombie.prop",
Vector(0, 0, 0), Vector(0,0,0), kScene, false, false)

agent_zombie_2 = AgentCreate("Zombie_2",
"sk61_zombieBulldozered.prop", Vector(0, 0, 0), Vector(0,0,0),
kScene, false, false)

agent_zombie_3 = AgentCreate("Zombie_3",
"sk61_zombieCracked.prop", Vector(0, 0, 0), Vector(0,0,0), kScene,
false, false)

agent_zombie_4 = AgentCreate("Zombie_4",
"sk61_zombieFaceless.prop", Vector(0, 0, 0), Vector(0,0,0),
kScene, false, false)

agent_zombie_5 = AgentCreate("Zombie_5", "sk61_zombieGuts.prop",
Vector(0, 0, 0), Vector(0,0,0), kScene, false, false)

agent_zombie_6 = AgentCreate("Zombie_6", "sk61_zombie.prop",
Vector(0, 0, 0), Vector(0,0,0), kScene, false, false)

agent_zombie_7 = AgentCreate("Zombie_7",
"sk61_zombieFaceless.prop", Vector(0, 0, 0), Vector(0,0,0),
kScene, false, false)
```

Notice how we didn't spawn in agents for Javier, Clementine and Gabe, even though they are present in the **Demo Scene**? Well, that's because these agents already exist on the "adv_richmondStreet" scene from the beginning. Their **Agent names** are "Javier", "Clementine" and "Gabe" respectively. Telltale usually names their characters the same as their character names, so it is often easy to guess who is called what.

In fact, for human characters it appears to be **necessary** to give them the same **Agent name** as Telltale would in their scenes, otherwise these types of agents do not spawn correctly. For example, if we were to spawn Mariana, but give her a different **Agent name** instead of "Mariana" (which is what she is called in Telltale's

scene containing Mariana), her facial animations and lip sync will be broken and won't work. This may not be 100% consistent across seasons, and some agents may work fine regardless, but I've run into this issue often enough that I say **you should always call human characters by their names**. Sadly, in some cases it may be difficult to determine which Agent name Telltale expects you to use, especially if it's some random generic villager or raider. However, it is possible to print out all of the agent names on a given scene into a **.txt** file, which can help identify potential names or agents to use (see Print agent list section for more info).

And, of course, you should not have repeating **Agent names** on two different agents, they should always be **unique**.

The next few lines of code move already existing agents that we do not want to use far away from the scene (set their position to (0, -1000, 0), which is basically *a thousands units down from (0,0,0)*), so that they don't interfere in our cutscene *(and we also move "Javier" to position of (0,0,0) for his weapon attachment*).

```
Custom_SetAgentWorldPosition("Tripp", Vector(0, -1000, 0),
kScene);

Custom_SetAgentWorldPosition("Kate", Vector(0, -1000, 0), kScene);

Custom_SetAgentWorldPosition("Conrad", Vector(0, -1000, 0),
kScene);

Custom_SetAgentWorldPosition("Eleanor", Vector(0, -1000, 0),
kScene);

Custom_SetAgentWorldPosition("Jesus", Vector(0, -1000, 0),
kScene);

Custom_SetAgentWorldPosition("Javier", Vector(0, 0, 0), kScene);
```

This is one possible way to hide unwanted objects away, though not the only one. In general, you can use **Custom_SetAgentWorldPosition** and **Custom_SetAgentWorldRotation** functions to change positions or rotations of your objects respectively. The first parameter you need to input is the **Agent name** of the agent you wish to affect, the second one is a Vector of (x,y,z) for new position coordinates or for new rotation respectively, and the last one is always kScene.

Lights creation

This is a scene-specific part of scene set up and should be used as an example. It will differ from scene to scene and can be changed or removed if you work on your own cutscene.

Feel free to skip this section for now and return to it later if you don't have the skills to understand it yet.

The next block is related to the creation of light sources in the scene. Here they are used for two things - to light up some of the spawned-in agent characters, and to create effect for the guns firing.

Why do we need to attach lights to characters? Well, that's because the characters spawned in with **.prop** files tend to have different, much darker, ambient light level, than characters who already exist in the scene. This ends up being very noticeable under certain angels, that's why I decided to attach these small spot lights to them to light up their front (where the darkness will be the most noticeable and distracting). You can use this code as an example to create your own lights for the characters.

Let's analyze the first block of code, which creates light for the "Rufus" character. At first, we set up some basic color-related parameters:

```
local flashlightColor = RGBColor(255, 255, 255, 50)
local envlight_groupEnabled =
AgentGetProperty("light_Directional", "EnvLight - Enabled Group",
kScene)
local envlight_groups = AgentGetProperty("light_Directional",
"EnvLight - Groups", kScene)
```

Then we spawn in an agent called "flashlightTestLight_spot" and set up its properties with **Custom_AgentSetProperty** function. How this function works is that it changes a value of a specified property (first parameter is an **Agent name**, second is **Property name**, third is the new **value** for the property, and fourth is **kScene**). Every agent has different properties that can be modified with this command, but you need to know the name of the property first. There isn't really any easy way to learn these, but you can use the code of the **Demo Scene**, as well as other scripting projects as reference.

I won't go into too much detail about the properties, though most of them should be pretty self-explanatory. I will only note that setting the "EnvLight - Shadow Type" to 0 will disable shadows from this light source, which is what we want in this case.

```
local flashlightTestLight_spot =
AgentCreate("flashlightTestLight_spot", "module_env_light.prop",
Vector(0,1.8,2), Vector(160, 0, 0), kScene, false, false)

Custom_AgentSetProperty("flashlightTestLight_spot", "EnvLight -
Type", 1, kScene) --0 point light, 1 spot light
Custom_AgentSetProperty("flashlightTestLight_spot", "EnvLight -
Intensity", 1, kScene)
Custom_AgentSetProperty("flashlightTestLight_spot", "EnvLight -
Enlighten Intensity", 0, kScene) --season 1 and season 2 only
Custom_AgentSetProperty("flashlightTestLight_spot", "EnvLight -
Radius", 80, kScene)
Custom_AgentSetProperty("flashlightTestLight_spot", "EnvLight -
Distance Falloff", 1, kScene)
Custom_AgentSetProperty("flashlightTestLight_spot", "EnvLight -
Spot Angle Inner", 10, kScene)
Custom_AgentSetProperty("flashlightTestLight_spot", "EnvLight -
Spot Angle Outer", 90, kScene)
Custom_AgentSetProperty("flashlightTestLight_spot", "EnvLight -
Color", flashlightColor, kScene)
Custom_AgentSetProperty("flashlightTestLight_spot", "EnvLight -
Enabled Group", envlight_groupEnabled, kScene)
Custom_AgentSetProperty("flashlightTestLight_spot", "EnvLight -
Groups", envlight_groups, kScene)
Custom_AgentSetProperty("flashlightTestLight_spot", "EnvLight -
Shadow Type", 0, kScene) -- 0 to disable shadows
Custom_AgentSetProperty("flashlightTestLight_spot", "EnvLight -
Wrap", 0.0, kScene)
Custom_AgentSetProperty("flashlightTestLight_spot", "EnvLight -
Shadow Quality", 3, kScene)
Custom_AgentSetProperty("flashlightTestLight_spot", "EnvLight -
HBAO Participation Type", 2, kScene)
Custom_AgentSetProperty("flashlightTestLight_spot", "EnvLight -
Shadow Near Clip", 0.0, kScene)
Custom_AgentSetProperty("flashlightTestLight_spot", "EnvLight -
Shadow Depth Bias", -0.5, kScene)
Custom_AgentSetProperty("flashlightTestLight_spot", "EnvLight -
Mobility", 2, kScene) --lets the light to move around
```

Finally, we attach our light agent to the character (Rufus, in this case). Attaching an agent means its position and rotation from now on will be tied to the agent we

attached it to. Basically, when we move Rufus around in the cutscene, the light will move with him as if it was part of him now.

```
AgentAttach("flashlightTestLight_spot", "Rufus");
```

The character is located at (0,0,0) and the light spawn position and rotation were made to match that so that in the game the light is pointed at the character's face.

In the next block another light is created for Mariana (with slightly different parameters).Then, in the following block we create seven lights for the seven zombies using a For statement. Basically, the code within the "for i = 7 do" structure is repeated seven times, but each time the agent names have its ending number go up by one (we use *tostring(i)* command to achieve that). You do not need to worry about such constructions if you lack the knowledge to understand them, you can simply create lights one by one like we did with Rufus and Mariana, or disregard these structures for now.

Lastly, two lights are created (once again, using the For statement) for the gun effects. Instead of spot lights, there are point lights, meaning they light up everything around them instead of going into one direction (we use different "EnvLight - Type" value to achieve that).

```
local light_name = "flashlightTestLight_point_gun_" ..
tostring(i);
local flashlightColor = RGBColor(255, 205, 0, 255)
local flashlightTestLight_point = AgentCreate(light_name,
"module_env_light.prop", Vector(0,-1000,0), Vector(90, 0, 0),
kScene, false, false)

Custom_AgentSetProperty(light_name, "EnvLight - Type", 0, kScene)
--0 point light, 1 spot light
Custom_AgentSetProperty(light_name, "EnvLight - Intensity", 2.0,
kScene)
Custom_AgentSetProperty(light_name, "EnvLight - Enlighten
Intensity", 0, kScene)
Custom_AgentSetProperty(light_name, "EnvLight - Radius", 10,
kScene)
Custom_AgentSetProperty(light_name, "EnvLight - Distance Falloff",
0, kScene)
Custom_AgentSetProperty(light_name, "EnvLight - Spot Angle Inner",
10, kScene)
Custom_AgentSetProperty(light_name, "EnvLight - Spot Angle Outer",
```

```
40, kScene)
Custom_AgentSetProperty(light_name, "EnvLight - Color",
flashlightColor, kScene)
Custom_AgentSetProperty(light_name, "EnvLight - Enabled Group",
envlight_groupEnabled, kScene)
Custom_AgentSetProperty(light_name, "EnvLight - Groups",
envlight_groups, kScene)
Custom_AgentSetProperty(light_name, "EnvLight - Shadow Type", 1,
kScene)
Custom_AgentSetProperty(light_name, "EnvLight - Wrap", 0.0,
kScene)
Custom_AgentSetProperty(light_name, "EnvLight - Shadow Quality",
3, kScene)
Custom_AgentSetProperty(light_name, "EnvLight - HBAO Participation
Type", 1, kScene)
Custom_AgentSetProperty(light_name, "EnvLight - Shadow Near Clip",
0.0, kScene)
Custom_AgentSetProperty(light_name, "EnvLight - Shadow Depth
Bias", 0.0, kScene)
Custom_AgentSetProperty(light_name, "EnvLight - Mobility", 2,
kScene) --lets the light to move around
```

These two lights will be used for Clementine's and Javier gun lighting during the shootout (see how the gun lights are programmed in Weapon lights section of Advanced Features).

Item attachment

This is a scene-specific part of scene set up and should be used as an example. It will differ from scene to scene and can be changed or removed if you work on your own cutscene.

Feel free to skip this section for now and return to it later if you don't have the skills to understand it yet.

Now, we move on to item attachment. It is possible to attach items to certain parts of character skeleton (to certain character bones) in order to have the characters use them in animations in realistic ways. Unfortunately, at this stage this is a very difficult thing to do properly, I will try to explain it the best I can but if you try to do a similar thing yourself, you will probably still need to spend quite a lot of time on it.

At first we simply spawn our agents for items (like weapons, guns and a bat in this example) at the *correct coordinates*.

```
agent_gun_1 = AgentCreate("Rifle_1", "obj_gunAK47.prop",
Vector(-0.56, 0.90, 0.11), Vector(90,0,-24), kScene, false, false)

agent_gun_2 = AgentCreate("Rifle_2", "obj_gunAK47.prop",
Vector(-0.56, 0.90, 0.11), Vector(90,0,-24), kScene, false, false)

agent_bat = AgentCreate("Bat", "obj_batAluminum.prop",
Vector(-0.541, 0.93, -0.03), Vector(10,-15,50), kScene, false,
false)
```

Then, we use the following code to attach them to the right wrist ("wrist_R" bone) of our characters with **AgentAttachToNode** function. Notice how this function can work with agent **Variable names** as well as **Agent names**. We also make a check to see if this bone even exists to prevent code from breaking in case it doesn't with an conditional **if** statement.

```
local nodeName = "wrist_R";
if AgentHasNode(agent_anf_1, nodeName) then
    AgentAttachToNode(agent_gun_1, agent_anf_1, nodeName);
end
if AgentHasNode(agent_anf_2, nodeName) then
    AgentAttachToNode(agent_gun_2, agent_anf_2, nodeName);
end
if AgentHasNode("Javier", nodeName) then
    AgentAttachToNode(agent_bat, "Javier", nodeName);
end
```

agent_gun_1 and **agent_gun_2** attached to ANF characters we made earlier. The bat is attached to Javier. Seems simple, right? Well, remember how I said you need to spawn these guns at the *correct coordinates*? The main issue is getting these coordinates. From what I understand, these are the position and rotation of a weapon that would fit into the character's hand (with respect to their own position and rotation) if the character were to be in an A-pose. The problem is that as far as I know, you can't simply put the character into an A-pose, and a lot of them have dynamic idle animations from the start. So getting these coordinates can be quite hard, especially considering that it is very difficult to analyze item's position post-attachment. Getting these numbers involves a lot of guesswork and trial-and-error, and I have no advice on how to make it easier.

As you may have noticed, these aren't all of the weapons you can see in the **Demo Scene**. Indeed, certain weapons are created at runtime instead. You can see how it's done in Attaching items section of Advanced Features.

There is one more interesting line of code that should be mentioned:

```
Custom_AgentSetProperty("Bat",  "Runtime: Visible", false, kScene)
```

This is a second way to hide an agent. Setting the "**Runtime: Visible**" property to **false** using the **Custom_AgentSetProperty** function will turn any agent invisible. Meaning that it will still function normally as an agent, but it will not be shown in the scene until you set this property to **true**. In this case we want to hide the bat from being shown until Javier needs it during the fight (see Advanced Features for more info).

Lastly, there is this line of code.

```
Callback_OnPostUpdate:Add(gun_light_update);
```

It adds an update function related to spawning gun lights (see in Weapon lights section of Advanced Features to learn how it functions). If you are making your own scene and don't want to use this functionality, you wouldn't need this line and should remove it.

Camera spawn

This is an essential part of scene set up.

The following block of code is responsible for spawning the **cutscene camera** or the **developer free camera**.

```
--MODE_FREECAM = true;
--if we are not in freecam mode, go ahead and create the cutscene
camera
if (MODE_FREECAM == false) then
    Cutscene_CreateCutsceneCamera(); --create our cutscene camera
in the scene
--create our free camera and our cutscene dev tools
else
    Custom_CutsceneDev_CreateFreeCamera();
    Custom_CutsceneDev_InitalizeCutsceneTools();
--add these development update functions, and have them run every
frame
   Callback_OnPostUpdate:Add(Custom_CutsceneDev_UpdateFreeCamera);

Callback_OnPostUpdate:Add(Custom_CutsceneDev_UpdateCutsceneTools_I
nput);
```

```
Callback_OnPostUpdate:Add(Custom_CutsceneDev_UpdateCutsceneTools_Main);
end
```

Which camera spawned in is controlled by the **MODE_FREECAM** variable. By default it equals to **false**, meaning that cutscene camera will be spawned. Notice the very first commented out line. If you remove the comment, it will set this variable to **true** right before execution of this code. I recommend using this line to control this variable to switch between your cameras in build (comment it out to set **MODE_FREECAM** to **false**, uncomment to set it to **true**).

```
MODE_FREECAM = true;
```

Print agent list as .txt

The next line is commented out, but if you uncomment it, it will look like this:

```
PrintSceneListToTXT(kScene, "ObjectList.txt");
```

This command will create a file called "**ObjectList.txt**" in the game's **Root** folder with the list of all the **Agent names** present in the scene. This can be helpful if you are looking for a name of a particular agent. Note that you should delete this file every time you run the scene, as it will not clear out previous data, but keep adding info to the end of the list every time this line is executed.

This is a debug/development-related line and you would normally comment it out or remove it before making a release build.

Player and Editor functions

This is an essential part of scene set up.

At the end of the function, just before the "end" command, you will see three lines, two of which are commented out.

```
--CutsceneEditor("demo_cutscene","sk61_tripp.prop");
CutscenePlayer("demo_cutscene", 0, 1, 1);
 --CutscenePlayer("demo_cutscene", 3, 0, 1);
```

These lines are used to launch the scene in either **Editor mode** or **Player mode**. **Editor mode** is used during cutscene making to set up your clips and save them as .ini files. **Player mode** is used to actually play the clips in your cutscene. By default

the **Demo Scene** runs in **Player mode**, actually playing back the clips as intended. If you wish to run it in **Editor mode** instead, you should comment out the line with **CutscenePlayer** function and uncomment the **CutsceneEditor** line:

```
CutsceneEditor("demo_cutscene","sk61_tripp.prop");
--CutscenePlayer("demo_cutscene", 0, 1, 1);
--CutscenePlayer("demo_cutscene", 3, 0, 1);
```

This will allow you to enter **Editor mode** after you build your mod and modify your cutscene.

As you probably noticed, both functions require you to add several parameters. The first parameter for both functions is a string of text containing the name of your cutscene's **clip folder**. This is the folder located inside the **Custom_Cutscenes** folder of your Build.

Cutscene Editor component only requires you to add one more parameter, and that would be a **.prop** file used to create a dummy agent in continuous motion set up (see In-game editor for more info). The only purpose of this agent is to visually indicate the final position to which your selected agent will move to. It doesn't really matter what this .prop file is, but it should probably be a humanoid character, and it should be part of the archives that you loaded in (or otherwise be available for the game to load in). In this case we use the Tripp **.prop** file to spawn the agent..

Cutscene Player component needs a bit more parameters to function. The Second parameter for Player is a number which indicates the name of a starting clip for your cutscene. Normally this would be 0, though you are free to load whichever clip you want as a starting one (it can be very useful to set your starting clip in the middle of the cutscene during testing). The **doc**s folder included with this tool contains "**dialogue-script.docx**" file, this file contains information about what dialogue or acting belongs in each clip inside the **Demo Scene**. This can help you understand the purpose of each of the clips in the cutscene. Creating and modification of clip files is described in detail in Editor Controls and Features section.

The third parameter indicates which **Dialogue mode** you wish to use in your cutscene. There are two modes available as of right now. If you set this parameter to **1**, you will use the **Main** dialogue mode with buttons that need to be clicked with a mouse (similar to how S2-S4 dialogues work in the original Telltale games).

If you set this parameter to **0**, you will be using the **Basic** dialogue mode, which looks closer to S1 style of dialogue. In that mode the choices are cycled through with W and S keys, or the Up and Down arrow keys, and then selected with a mouse click. *Note that **Main** mode only works in Fullscreen 1920x1080 Screen Resolution.* If resolution is different, it will automatically change into **Basic** mode (if the resolution changes during playback, it will dynamically adapt the UI to whichever mode is appropriate).

Lastly, the fourth parameter is used to tell the **Player** component if you are running a Season 3-based scene or a scene from some other season. Set the value to **0** if you are running another season, and to **1** if you are running the scene in Season 3 environment. This information is required for correct UI set up.

The third (commented out) line gives you an example of alternative parameter input for the **Player mode**. This line, if uncommented, will start the cutscene from **clip 3** and will use **Basic** dialogue mode.

```
CutscenePlayer("demo_cutscene", 3, 0, 1);
```

Feel free to play around with these parameters to see what you can do. Note that only one instance of **Editor** or **Player** should ever be initiated. **Do not** execute both **Player** and **Editor** lines at the same time, and do not execute multiple **Player** or **Editor** lines, otherwise you will simply break your code. Comment out or remove unused lines before building.

Another important thing to remember is to enable **Developer Free Camera** (as described in Camera spawn section) when running **Editor mode**. Editor mode will not function if **MODE_FREECAM** was set to **false** before camera set up code executes. You can also use **Free Camera** in **Player mode** for testing purposes if you wish, but it is **required** for **Editor mode**.

After the **Scene function** ends comes the section with the **custom functions**. This section will be covered in detail in Advanced Features. For now you can disregard that section and move to the very end of the file. There, you will see the following line.

```
--open the scene with this script
SceneOpen(kScene, kScript);
```

This line **should always be present** at the end of your script, as it loads the correct **.scene** file, and initiates execution of the **Scene function**.

Main menu code

Now it's time to open the main menu files. Open Menu_main.lua in your script editor. Like I said before, this script will replace the vanilla main menu, allowing you to add your own elements to it. For the most part, you should probably leave this file untouched, except for the part where you add the button to run your scene. In this case, you should search for the following line:

```
Menu_Add(ListButtonLite, "firstcutscenelevel_3", "Launch Demo
Scene", "Menu_LoadLevel_Copy(\"" .. "WalkingDead302" .. "\", \""
.. "Demo_Scene" .. "\")")
```

This line is what adds the button to load your scene to the main menu. You can easily modify it to fit your own needs. The first parameter should always be **ListButtonLite**, the second parameter should be a unique string of text, the third parameter is the text that will actually show up in the game.

The fourth parameter is a complex string that you can change to fit your own code: replace the season and episode numbers in **WalkingDead302** with whatever episode your scene comes from, and replace **Demo_Scene** with the name of the **.lua** script you wish to run.

There are several commented out lines here too, if you uncomment them, more buttons to run other sample **Scene_ .lua** files will be added to the main menu. You can use these lines for references as well.

```
Menu_Add(ListButtonLite, "firstcutscenelevel_2", "Launch S2
Scene", "Menu_LoadLevel_Copy(\"" .. "WalkingDead202" .. "\", \""
.. "Scene_S2" .. "\")")

Menu_Add(ListButtonLite, "firstcutscenelevel_4", "Launch S4
Scene", "Menu_LoadLevel_Copy(\"" .. "WalkingDead402" .. "\", \""
.. "Scene_S4" .. "\")")

Menu_Add(ListButtonLite, "firstcutscenelevel_m", "Launch M Scene",
"Menu_LoadLevel_Copy(\"" .. "WalkingDeadM102" .. "\", \"" ..
"Scene_SM" .. "\")")
```

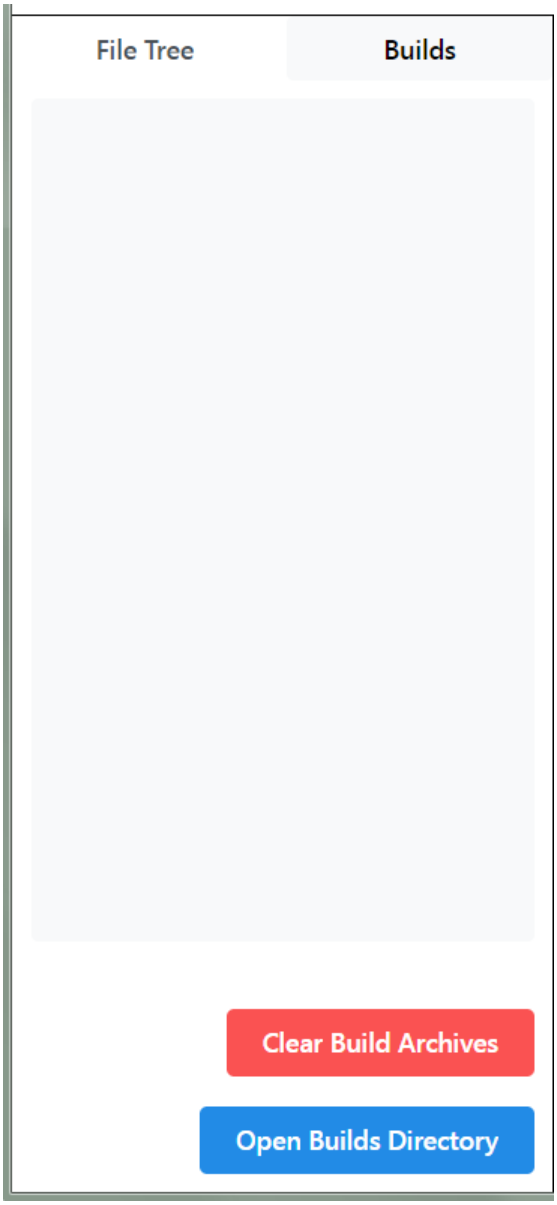
Building from Source

In order to apply the changes you make to the code, the mod files need to be rebuilt. In order to rebuild the mod archives, you need to click on the "**Play**" button on top on of the **Script Editor** window:

This will execute the building process and then start up the game so you can check your changes right away. If the game doesn't start, please check console in the "**Builds**" tab on the left - it is possible the code has a syntax error which will be highlighted there.

It is important to close the game before building your files and wait for it to fully stop functioning. You can use **Alt+F4** key combination to quit the game at any time while it is an active window. **Do not** execute the building process until "*Game closed!*" text appears in the console after you close the game. If you do, the Script Editor will get stuck and will throw errors during building even after the game is closed. If this happens, just try clicking the "**Play**" button some more (do not spam it, wait for it to either throw an error or get through the process). If this doesn't help, try restarting the **Script Editor**.

The **Script Editor** uses cache for faster building. Unfortunately, sometimes it doesn't clear or update information properly, in which case it becomes a detriment. In the "Builds" tab there is a button called "Clear Build Archies" which will clear out all of the cache (it does not delete the mod files already installed into the game).



It is recommended to run this process from time to time to prevent accumulation of errors. It is also **necessary** to perform if you make any modifications to the folder structure of your mod, rename or delete any **.lua** files or move them to a different folder, create new folders or **.lua** files, or edit the project's **.tseproj** file. You should also manually clear the contents of **Cutscene_Editor** folder of your **Build** (the folder inside game's **Root**) if you do any of the above to prevent outdated leftovers from staying there (proper files should be generated anew during build process anyway). Just adding new assets into folders does not seem to require clearing of cache.

Clearing cache can also sometimes help you if your **Script Editor** gets stuck but you don't want to restart it.

Editor Controls and Features

Now that we have our scene set up, the actual cutscene-making will mostly happen within the game itself, without the need to write more code (unless you want to implement something from Advance Features). Please set your script to **Editor mode** and perform building process before continuing in this section.

.ini files

A lot of scene specific data is stored inside the special **.ini** files located in your cutscene's folder within **Custom_Cutscenes** folder of the **Build** (in case of the **Demo Scene** that would be **demo_cutscene** folder). These .ini files store humanly-readable and editable data inside (use **Notepad++** to work with these files). Every cutscene folder must include at least **agents.ini** and **player.ini** files, and at least one **clip** file before **Player mode** can be initiated. Clip files are created through the In-game editor and their file names include only numbers, you can think of them as building pieces of your cutscene.

agents.ini holds information about which agents can be controlled by our **Player component** and should be filler out manually before using the **Editor**. Only these agents will be selectable during **Edit mode**. You can think of this as a list of selectable agents who will be used as actors in your cutscene.

```
[agents_names]
1=Mariana
2=Gabe
3=Javier
4=Clementine
5=Rufus
6=Eli
7=Roxanne
8=Zombie_1
9=Zombie_2
10=Zombie_3
11=Zombie_4
12=Zombie_5
13=Zombie_6
14=Zombie_7
15=empty_15
16=empty_16
...
```

This file should always contain **[agents_names]** section, and then number variables in order starting from **1** (**Agent numbers**), each holding a value equal to an **Agent name**. The actual code of this tool uses these numbers to find data related to the agents in individual clips. This means that you can modify this file's **Agent names** and this will make the Player use different characters in place of the previous ones. For example, if you switch Mariana and Gabe's Agent names here like this:

```
[agents_names]
1=Gabe
2=Mariana
3=Javier
...
```

then Mariana will take Gabe's place in the cutscene, and Gabe will take Mariana's. **Editor** also finds data corresponding to each agent based on the given number during clip loading.

You may notice that agents between 15 and 29 **Agent numbers** are not actually present in the scene and have fake "empty" names instead. These **Agent numbers** were filled in due to **a very crucial limitation** of this tool: **you cannot increase the amount of agents after you start working on a cutscene**. Well, technically you can, but you would either need to load and re-save every clip made before adding new agents though the **Editor**, or manually add the missing data to the **clip .ini** files. Having missing data inside clip files will cause the tool to work incorrectly. Since I didn't know the exact number of agents I would need in my cutscene beforehand, I went ahead and filled out all the necessary fields fo 29 agents in this **.ini** file. Then, I was adding new agents into the cutscene by replacing the empty names with legit ones when I needed to.

It is recommended that you do the same for your scenes, as adding new agents to an already existing scene can be a huge pain and waste of time. Empty agents do not appear to cause any issues with the tool, and you can remove them from the file without issues after you finish your cutscene if you do not want them to clutter the file (their old data will still be saved within the **clip .ini** files unless you re-save them with a new amount of **Agent numbers**).

player.ini file tells the **Player component** which order to play the clips in, which clip comes after which, as well as the length of each clip. It also contains data related to dialogue choices, and to **custom scripts**. Every clip present in the scene must have exactly one corresponding section inside this file. Clips without this section cannot be loaded by the **Player** (though they can be loaded in the **Editor**). You can think of this file as a kind of *timeline* for the cutscene. The contents of this file are also filled out manually and look something like this:

```
[4]
duration=5
choices=0
choice1_text=t1
choice2_text=t2
choice3_text=t3
choice4_text=t4
next_clip=5
next_choice1=0
next_choice2=0
next_choice3=0
next_choice4=0
```

The action name matches the name of the clip we're filling the data for. The **duration** variable is the amount of seconds the clip will last for. The **choices** variable is the amount of choices this clip will present. Value of **0** means there will be no choice present, and after the duration of the clip ends, the Player will switch the playing the next clip, which is indicated by the **next_clip** variable.

If the **choices** variable contains a value of **1** to **4**, then that amount of choices will be given to the player. These choices will contain text which you can fill in as a string in **choice1_text** to **choice4_text** variables. Selecting any of these choices before the clip ends will instead switch the **Player** to run a different clip. The **next_choice1** to **next_choice4** variables contain numbers indicating which clips should play during selection of each of the four choices. In this example the clip has zero choices, so these variables' contents don't really matter, but if if you were to look at a clip with choices, it would look something like this:

```
[9]
duration=10
choices=4
choice1_text=You let them brand you?!
choice2_text=You were with those monsters?!
choice3_text=I can't believe you lied to me...
choice4_text=...
next_clip=15
next_choice1=10
next_choice2=12
next_choice3=13
next_choice4=15
```

You will also notice that some clips contain another variable - **custom_script**.

```
[45]
duration=4.8
choices=0
choice1_text=text
choice2_text=text
choice3_text=text
choice4_text=text
next_clip=46
next_choice1=0
next_choice2=0
next_choice3=0
next_choice4=0
custom_script=music_clip_45
```

This variable contains the name of a **function** that will be executed at the start of this clip. It should only be present in a clip if a function of this name actually exists in the code, otherwise the **Player** will work incorrectly. If you wish to prevent a custom function from playing, remove this variable from the clip section or comment it out with a ";" symbol:

```
[1]
duration=3
choices=0
choice1_text=t1
choice2_text=t2
choice3_text=t3
choice4_text=t4
next_clip=2
next_choice1=0
next_choice2=0
next_choice3=0
next_choice4=0
;custom_script=custom_function_clip_2
```

Usage of these functions is considered an Advanced feature and is covered in the corresponding section.

You may also notice another file in the **Demo Scene**'s folder - **persistent.ini**. This file is used to contain **Persistent data** for the cutscene, and its usage is also described in Advanced Features. Note that this is a scene-specific file, it does not necessarily have to be present in a cutscene unless you are using a similar code to store your **Persistent data**.

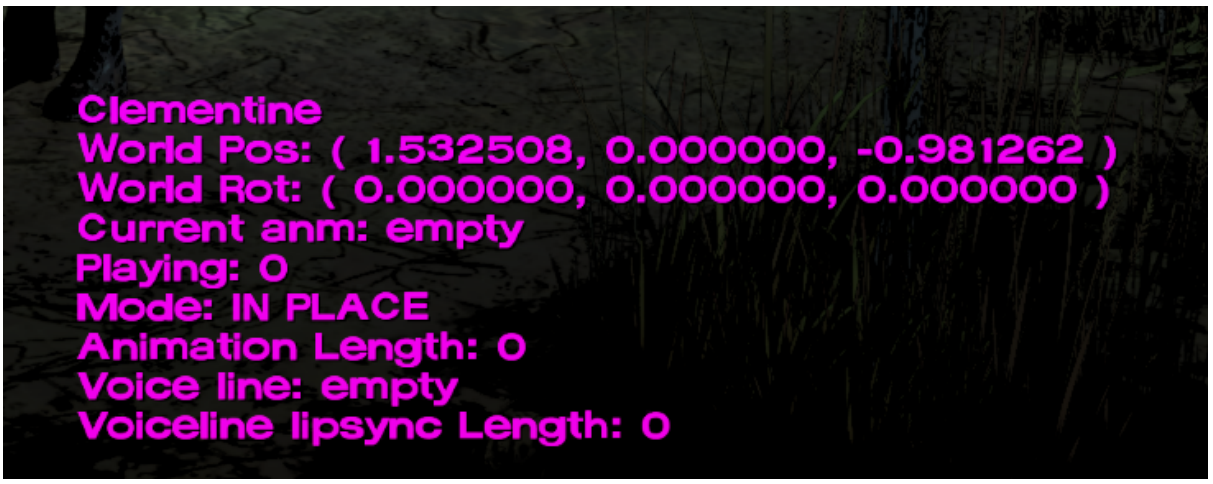
Finally, the **clip** files contain data about each agent used in the cutscene (and the camera as well), about their position, rotation and actions they should perform during this clip. Since these files are usually created through the **Editor** itself and closely correspond with its own interface, we will look at these files more extensively after the In-game editor section, in .ini files Part 2.

In-game editor

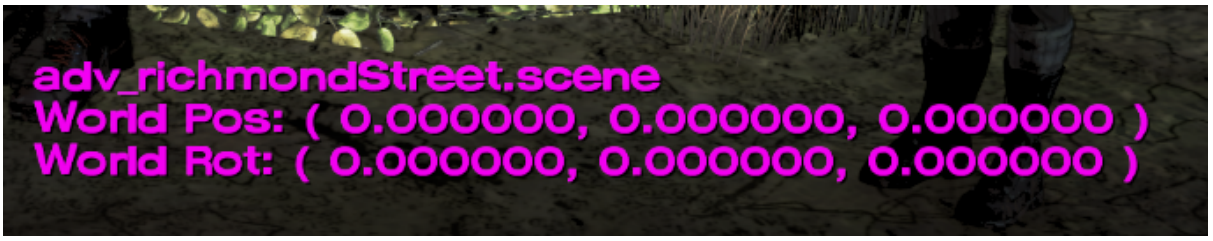
Before running the scene in **Editor mode**, you should also open **animation_input.ini** (located in the **Root** directory of the game) in **Notepad++**. This file is vital to feed the **Editor** with information about which resources (such as voice lines and animations) or other data you want to add to your agent or the clip's camera, and it is also used to select different agents to work with.

Running the scene in **Editor mode** with enabled **Developer Free Camera** should look something like this:

The text in the top left corner contains some basic information about the runtime, as well as current camera coordinates. The **Pink text** should be located near the currently selected agent and will contain clip data related to it.



If you do not see the **Pink text**, and it shows the name of the scene instead, it probably means you do not have a valid object selected:



Switch to **animation_input.ini** and change the **agent_name** variable to any **Agent name** that is present in your scene's **agents.ini** file.

The **Editor** has pretty simple controls. You can look around using your **mouse**, you can move the camera around with **WASD**, you can use **E** and **Q** to move up and down.

From now on it's up to you to set up your clip the way you want. You can move around your selected agent with **arrow keys** (**X** and **Z axes**), with "**+**" and "**-**" keys (up and down, aka **Y axis**) and you can hold **shift** to move the agent faster. If you hold **alt** and press the same keys, you will **rotate** the agent instead. Usually you would need to only use **LEFT** and **RIGHT** arrow keys to rotate the agent around the **Y axis**, but you also use **UP** and **DOWN** keys (**X axis**) or "**+**" and "**-**" keys (**Z axis**). Current position and rotation of the agent are shown in pink text.

Now let's look closer at the **animation_input.ini** file.

```
[info]
animationinput=empty
agent_name=Clementine
voice_line=empty
```

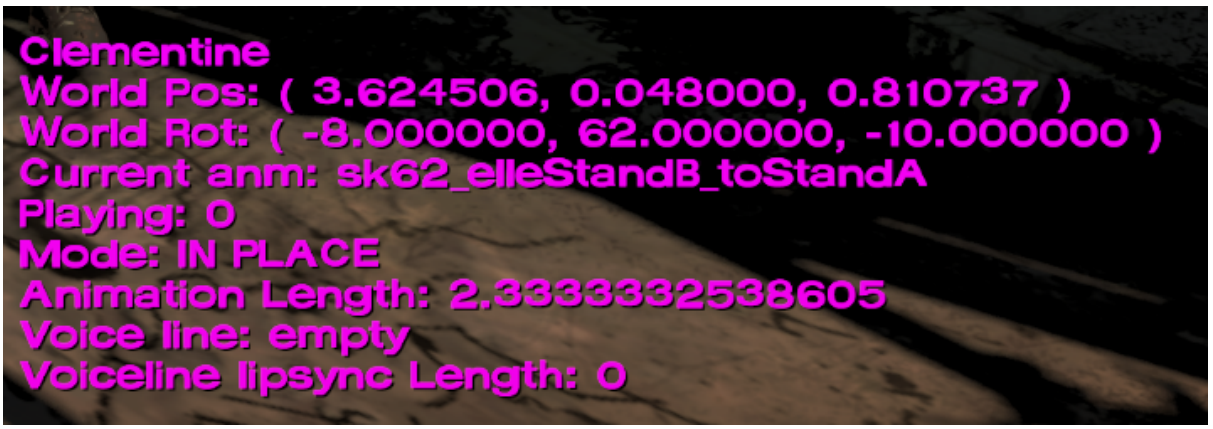
```
speed=0
r_speed=0
cam_speed=0
cam_r_speed=0
```

This file contains a bunch of parameters you can set up for the agent and the scene. **animationinput** variable requires a name of an animation file you wish to play on the agent. You can set this to **empty** if you do not wish to play any animation. Or you can set it to any animation name you wish to play. For example, for Clementine we can try to play animation **sk62_elleStandB_toStandA.anm**.

Switch to the open **animation_input.ini** window in **Notepad++** without closing the game (you can use **Alt+Tab** command to do this) and change the **animationinput** variable:

```
[info]
animationinput=sk62_elleStandB_toStandA
agent_name=Clementine
voice_line=empty
speed=0
r_speed=0
cam_speed=0
cam_r_speed=0
```

Now if you switch back to the game window, you will see that the agent's **Pink text** has changed, reflecting our newly selected animation.

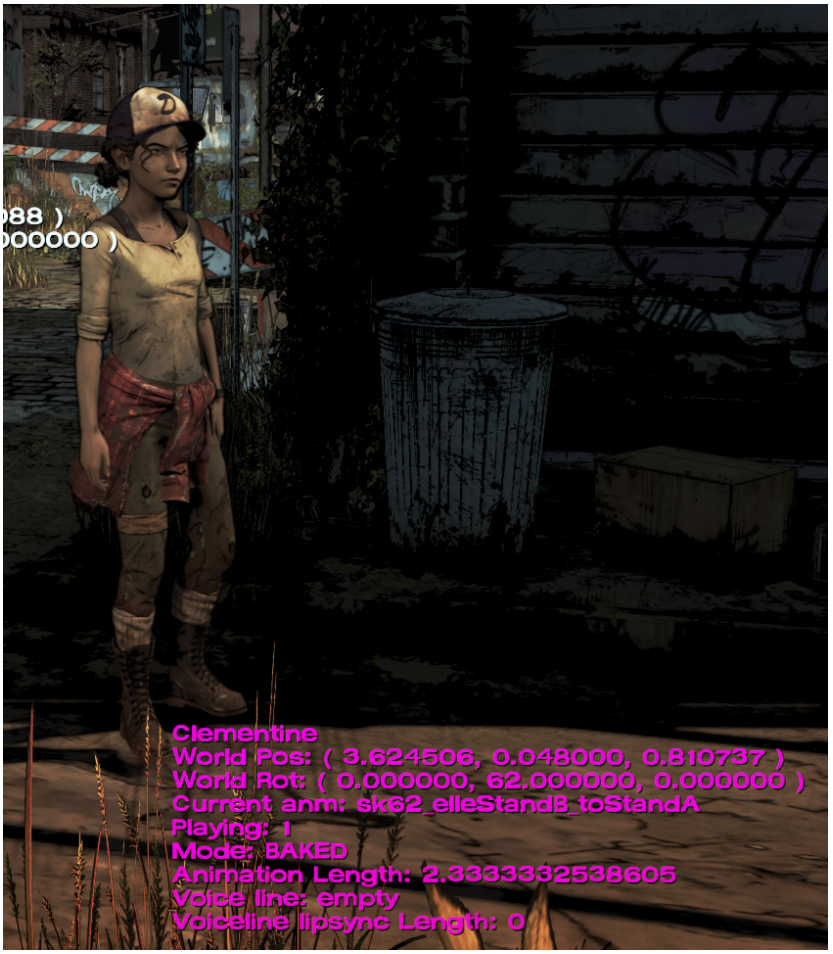


The tool also supports three different **animation modes** (as reflected in the **Pink text**) which you can set up for each agent in your clip. You can switch between modes in-game by pressing the **T** key.

The following modes are available to you:

- **IN PLACE** - agent will stay in place while the animation plays out in a loop.
- **BAKED** - agent will move with the animation, and will reset to its default position after the animation ends and loop. Useful for various animations which involve characters moving.
- **ROOT MOTION** - agent will continuously move with the animation, and its position will not reset position after the animation. The tool will use agent's last position as the starting position for the next animation loop. Useful to make characters run continuously in a straight line, not as useful for other types of animations.

You can preview how the animation will play in each mode by pressing the **SPACEBAR** key. If you wish to stop the preview, you can press **ENTER** key.



IN PLACE also allows the user to set up coordinates and have the agent continuously move there. Press **R** key to spawn in a temporary character object (the one you set up when calling **CutsceneEditor** function, Tripp in the case of **Demo Scene**). You can now move this object around, and set up a new position and rotation for it:

If you wish to cancel out of selecting this **move-to position and rotation**, press **R** again. If you wish to confirm and record the coordinates of the temporary character, press **Left Mouse Button**.

Now you have the **move-to coordinates** recorded, but in order to use them we need to set character's moving speed. Switch to **animation_input.ini** window and change the **speed** and **r_speed** values. These values indicate the amount of seconds it will take for the agent to move from starting coordinates to **move-to** coordinates (the higher these numbers are, the slower will the agent move; values of 0 mean no movement). Variable of **speed** is used for position change and variable **r_speed** is used for rotation. You can then preview how you agent will move now by switching to game window and pressing the **SPACEBAR** key (make sure you have **IN PLACE** animation mode selected!). The preview will show continuous motion of your agent. In the preview the agent's movement will keep looping, however during actual cutscene it will not loop, the agent will reach the **move-to position** and **rotation** at given speeds and stay in that location until the end of the clip.

One more thing to note about animations - character animations, as well as character prop files, all have an **sk prefix** with a number which indicates character's skeleton type (for example, the prefix for **sk62_clementineTense_palmsDownDismiss_add.anm** is **sk62**, meaning this animation is meant to be used with character of **sk62-type** skeleton). Generally, **you**

should always try to make sure that the animations you use match the skeleton type of the character, otherwise you may run into strange deformation issues, some of which may remain even after the animation stops playing. There are exceptions - some animations work fine on mismatching skeleton types, and some character specific-animations may not look correctly on other characters with the same skeleton type. You are free to experiment with it on a case-by-case basis. But for most animations, it's better to stick to the same skeleton types.

Sometimes preview may not correctly showcase how the animation will play out in the **Player**, but it should suffice for the vast majority of cases.

The **Pink text** will also show you the current animation length (the amount of seconds animation will last, after this time passes animation will start playing from the beginning) for easier timing of your clips *(note - you can use **AnimationGetLength(string)** command in your code to return this number for any animation if you need to)*.

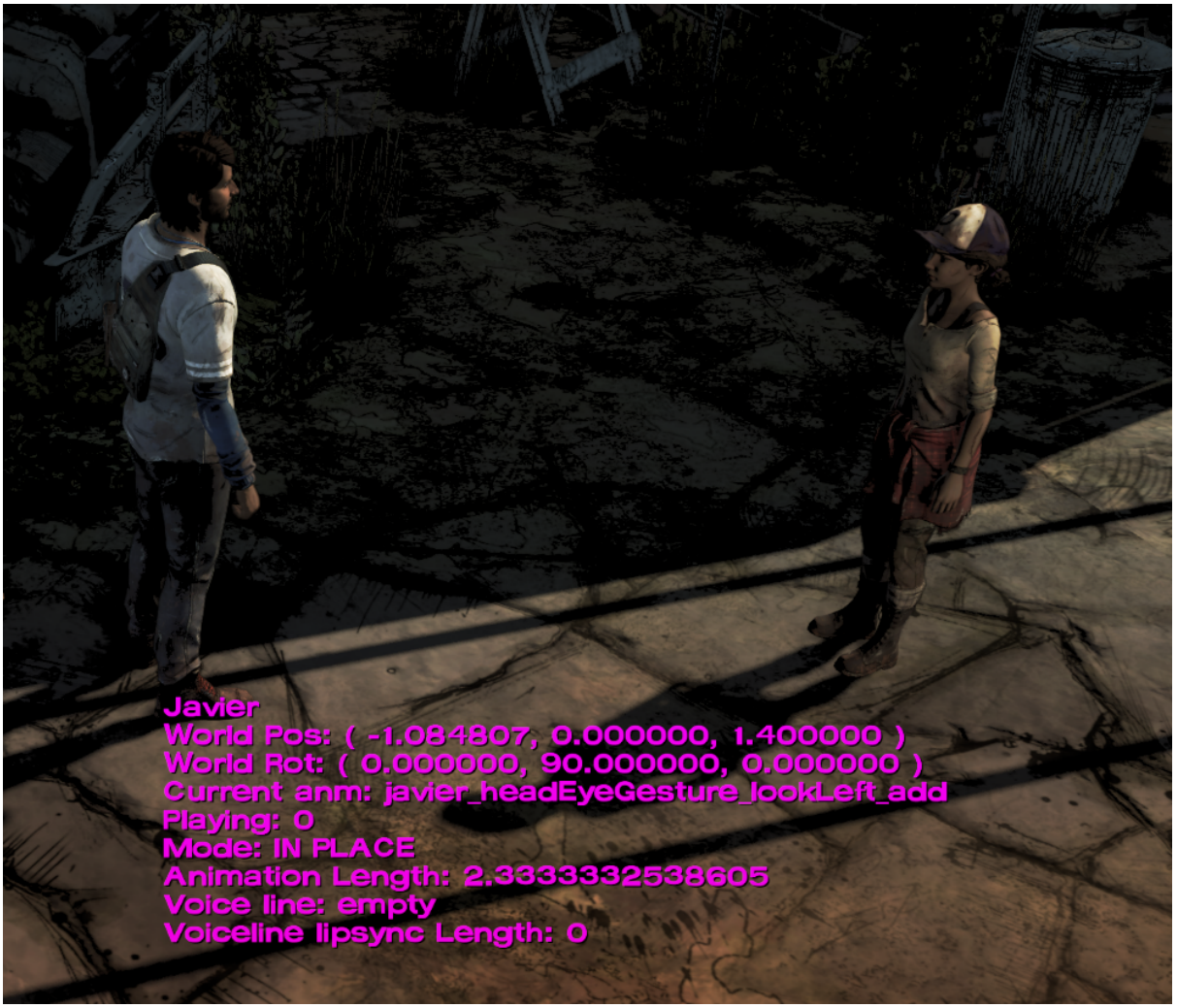
When moving around character agents in the scene, you may notice that they appear to enter walk cycle animation, and the speed of their animation is determined by movement speed. This is a feature that some, though not all, character agents may possess. It is useful because it allows you to use **IN PLACE** animation mode to move agents around and have them walk with the appropriate speed without having to search for and pick good looking walking animations (for this walking cycle to work correctly you can leave the animation blank). This is used frequently in the **Demo Scene**, for example in the very first clip - **clip 0**.

You can also add voice lines to the characters to speak. You can read more about how to correctly access and use voice data related audio and animations in Source File Structure section. If you followed the steps described in said section, you can apply your voice lines to the selected agent by adding the name of the voice line file to the **voice_line** variable in **animation_input.ini** file. You can preview the voice line being spoken in-game by pressing the **F** key. The **Pink text** also shows you the length of lip sync animation, giving an idea of how fast the character will speak the line.

The tool assumes that voice line and lip sync animation share the same name, meaning that if you use custom audio, you must also make a copy of an animation you wish to use for lip sync with the same name. If the animation file isn't present, the lip sync simply won't play, meaning you can also use the **voice line** variable to simply play sounds at the start of a clip (which is done in **clip 85**, for example).

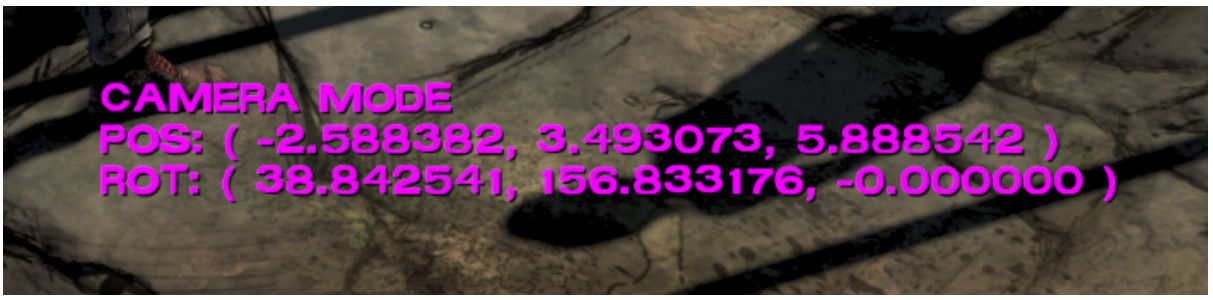
If you want to work with a different agent, you need to change the value of **agent_name** in **animation_input.ini** to a different **Agent name** (make sure the agent you are trying to select exists in **agents.ini**; if it doesn't, add it, save your progress and restart the game).

When you switch back to the game, the **Pink text** will move to the location of the new agent and this agent's variables will be loaded instead (the data inside **animation_input.ini** will be automatically updated as well, which is intended). The old agent will remain where it was and will remember its own variables until the game is closed.

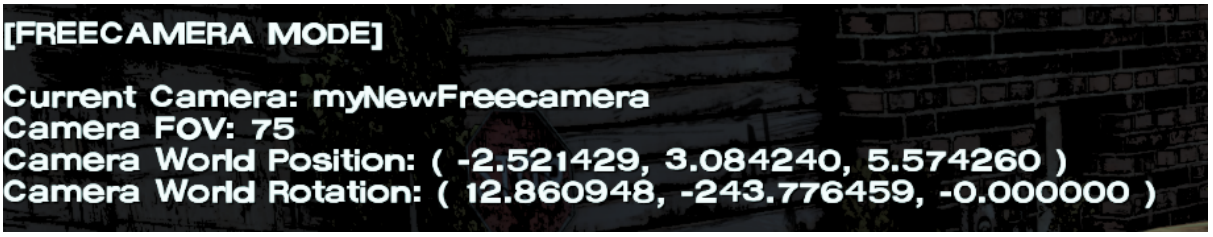


Note: After you change the **Agent name** you **must** switch to the game's window at least once to load the new agent's variables before you can edit them. Otherwise any changes you make to the variables will not be saved.

Lastly, you can set up your **camera angle** and **camera movement** for the scene. Press **Middle Mouse Button** to enter **Camera Mode**.



In **Camera Mode** your controls will be limited, and the **Pink text** will show your **free camera**'s position and rotation. Note that **the rotation will not be shown correctly in Pink text**, you should refer to the values given in white text in the top left corner instead:

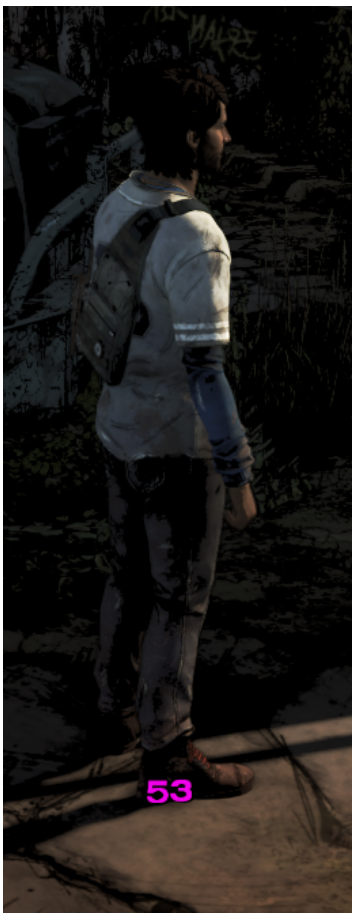


You can leave Camera Mode at any time by pressing the **Middle Mouse Button** once again. You can also set your camera's **starting coordinates** by pressing the **Left Mouse Button**, and **move-to coordinates** by pressing the **Left Mouse Button**. Camera movement works exactly like continuous agent movement in the **IN PLACE** animation mode, with **cam_speed** and **cam_r_speed** values being used to determine the speed of **position** and **rotation** movement of the camera. Note that these values are actually recorded by the game when you

There is no preview for the camera as of now, and the camera related values are not agent-specific and will be saved once for the entire scene.

Saving and loading clips

The values you set for agents will be retained by the game until you close. But in order to make use of your work, you need to record them into .ini files for future use in **Player**. If you wish to save everything you did in the **Editor**, or load an already existing clip for further editing, Press the **Z** key. You will enter the **save-load mode**, and you will be expected to enter the **clip name** using **number keys** on the keyboard (the name you enter will appear in **Pink**, while the regular **Pink text** will be hidden).



After you type in the desired name, you can click **Insert** key to save the clip as an .ini file with said name. The file will appear along all the other .ini files. If a file with the same name already exists, **it will be replaced**.

You can also press **Delete** key to load the clip of the same name instead. If you do so, all of the clip's data will be loaded into the scene, and all the agents will update their positions and variables accordingly.

Please be mindful of which button you press, **do not** accidentally overwrite your finished clips. **There will be no confirmation warnings about overwriting existing files. Make frequent backups!**

if you wish to exit **save-load mode** without saving or loading anything, press the **Z** button again.

After you set up and save the clips you want, you can then disable **Free Camera** and switch your script to **Player mode**, and see how they all play in action (don't forget to set all necessary clip data in **player.ini** and select your desired **starting clip** in the script).

.ini files Part 2

Now let's take a closer look at how the saved clip files actually work. All of them are saved in a humanly readable format and can be edited manually through **Notepad++**. Editing these files through the **Editor** and through **Notepad++** are both useful and valid methods of working with clip files.

If you open the files in Notepad++, you'll see something like this (we are using data from **14.ini** as an example here):

```
[1]
speed=0
move_to_x=15.856069564819
move_to_z=0.073998957872391
rot_z=0
r_speed=0
move_to_y=0
pos_y=0
rot_x=0
anm=empty
rot_to_x=0
mode=0
voice_line=empty
rot_y=90
pos_x=-2.1747803688049
rot_to_y=55
rot_to_z=0
pos_z=1.25

[2]
speed=0
move_to_x=16.390050888062
move_to_z=0.61399859189987
rot_z=0
r_speed=1.5
move_to_y=0.0059999991208315
pos_y=0.0059999991208315
rot_x=0
anm=empty
rot_to_x=0
mode=0
voice_line=empty
```

```
rot_y=90
pos_x=-1.6407990455627
rot_to_z=0
rot_to_y=55
pos_z=0.61399859189987

[3]
speed=0
move_to_x=-1.0848069190979
rot_y=90
rot_z=0
r_speed=1.5
move_to_y=0
pos_y=0
rot_x=0
anm=sk61_javierStandA_indifferentHeadSideToSide_add
pos_x=-1.0848069190979
mode=1
voice_line=a371405041
rot_to_z=0
rot_to_y=55
rot_to_x=0
move_to_z=1.4000000953674
pos_z=1.4000000953674

[4]
speed=0
move_to_x=-0.3007800579071
move_to_z=2.2640020847321
rot_z=0
r_speed=0
move_to_y=0
pos_y=0
rot_x=0
anm=empty
rot_to_z=0
mode=0
voice_line=empty
rot_y=-101.99997711182
pos_x=1.1672191619873
rot_to_x=0
rot_to_y=217
```

```
pos_z=1.8640022277832

...

[camera]
speed=2
move_to_x=-0.31731742620468
move_to_z=1.2456296682358
rot_to_z=0
pos_z=1.0144535303116
rot_to_y=-76.786193847656
r_speed=2
move_to_y=1.6137666225433
rot_z=0
pos_x=-0.38959819078445
pos_y=1.6004682302475
rot_y=-55.702968597412
rot_to_x=1.1482236385345
rot_x=-0.76847833395004
```

The section names correspond to the number of each agent inside **agents.ini**. The data in each section holds information directly related to what you did to each particular agent in the **Editor mode**.

- **pos_x, pos_y, pos_z** - contain data about agent's starting position.
- **rot_x, rot_y, rot_z** - contain data about agent's starting rotation.
- **mode** - contains data about agent's selected animation mode *(0 = IN PLACE, 1 = BAKED, 2 = ROOT MOTION)*.
- **anm** - contains the name of the selected animation for agent to play, if any.
- **voice_line** - contains the name of the selected voice line for agent to speak, if any.
- **move_to_x, move_to_y, move_to_z** - contain data about agent's move-to position.
- **rot_to_x, rot_to_y, rot_to_z** - contain data about agent's move-to rotation.
- **speed** - the amount of seconds it will take for agent to move from the starting position to move-to position.
- **r_speed** - the amount of seconds it will take for agent to rotate from the starting rotation to move-to rotation.

At the bottom of the file there is also a **[camera]** section, which holds camera-related data. Its purpose is the same as for other agents, but the amount of variables is smaller, since the camera does not have animations or voice lines.

Note that if you plan to edit **clip files** through **Notepad++**, please make sure that there is no missing data for any of these variables, and that every variable is present in each section it should be present in. Having missing data in your clip files will cause the **Player** to work incorrectly.

Finding animation names

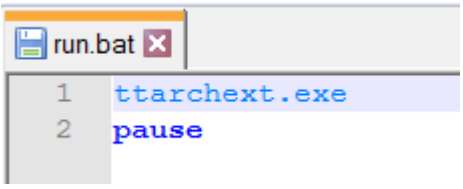
When using assets from Telltale's archives, you need to find their names. Usually you can simply load the necessary archive into **Telltale Explorer** and find the name that way. However, things like animations are used extensively and it can be a pain in the ass to constantly have to search for them through **Telltale Explorer** - it's not the fastest program out there, and its extraction capabilities are limited and also take a long time. This is where **ttarchext v0.3.2** comes in. It's an old tool used to bulk-extract Telltale's archives. Through creating special **.bat** files we can extract assets that we need from archives we need.

If you open your **Script Editor**'s resources folder (**telltale-script-editor-win32-x64\resources**), you will see the **ttarchext.exe** file. Just clicking the file won't give you any results, you need to create special files with parameters and run those files instead.

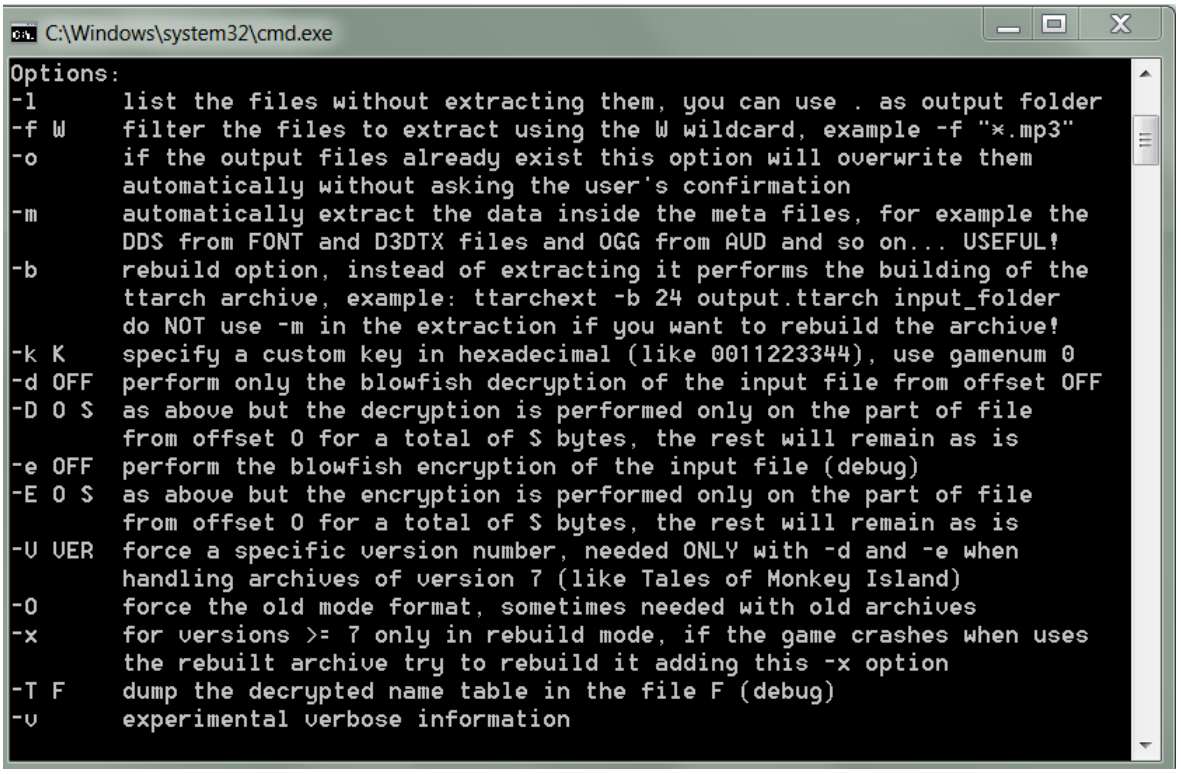
In the same folder as ttarchext.exe create a file with **.bat** format. You can do this by creating a text .txt file and then changing the format through renaming (the name itself can be anything). If you are unable to change the format of a text file, you need to enable editing of known file formats as described in Preface.

After creating the **.bat** file, open it in **Notepad++**. At first, you can simply write these lines and then run the save like you normally would run a program:

```
ttarchext.exe
pause
```



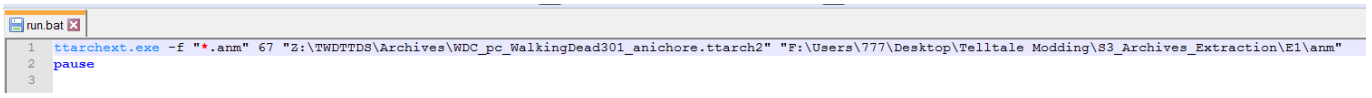
This will run the **ttarchext.exe** but prevent its window from closing, letting you read about the program and all its available options. You can take advantage of them as you see fit.



Now, using these descriptions, we can write a .bat file which will let us extract the files we need.

```
ttarchext.exe -f "*.anm" 67
"Z:\TWDTTDS\Archives\WDC_pc_WalkingDead301_anichore.ttarch2"
"F:\Users\777\Desktop\Telltale
Modding\S3_Archives_Extraction\E1\anm"

pause
```



The **-f "*.anm"** parameter lets us select the names and formats of files we want to extract, animations in this case (the * symbol before the format means we accept files with any names).

The number **67** is the game's index number, for TWDTTDS it should always be **67**.

The next parameter is the path to the game's archive, in this case its **"Z:\TWDTTDS\Archives\WDC_pc_WalkingDead301_anichore.ttarch2"** (S3E1 animations archive).

Next is the output folder for our files, in this case it's **"F:\Users\777\Desktop\Telltale Modding\S3_Archives_Extraction\E1\anm"**. Note that you need to actually create this folder first or you will receive an error. Also make sure you actually have the space on the disk to contain the extracted files. **Do not** create this folder in your **Script Editor**'s project folders, keep it in a separate location (we do this so that we can find the file names more easily, not to actually use these files).

The next line says **pause** which simply prevents the **ttarchext.exe** window from closing, and lets you read if the extraction was successful or not.

Should you do everything right, running a **.bat** file with this input will initiate the extraction process, it may take a while but it will be faster than Explorer's and you will end up with a folder with all of the animation files from S3E1. Now you can simply look through this folder and see all of the existing files and their names any time you want.

We can do the same with any other archives we need. And we can even put all the instructions in the same **.bat** file. For example, a file such as this will extract all S3 animations and voice sounds into corresponding folders (each line is executed one after another).

snd_anm_extraction_s3.bat

```
ttarchext.exe -f "*.anm" 67 "Z:\TWDTTDS\Archives\WDC_pc_WalkingDead301_anichore.ttarch2" "F:\Users\777\Desktop\Telltale Modding\S3_Archives_Extraction\E1\anm"
ttarchext.exe -f "*.wav" 67 "Z:\TWDTTDS\Archives\WDC_pc_WalkingDead301_voice.ttarch2" "F:\Users\777\Desktop\Telltale Modding\S3_Archives_Extraction\E1\snd"

ttarchext.exe -f "*.anm" 67 "Z:\TWDTTDS\Archives\WDC_pc_WalkingDead302_anichore.ttarch2" "F:\Users\777\Desktop\Telltale Modding\S3_Archives_Extraction\E2\anm"
ttarchext.exe -f "*.wav" 67 "Z:\TWDTTDS\Archives\WDC_pc_WalkingDead302_voice.ttarch2" "F:\Users\777\Desktop\Telltale Modding\S3_Archives_Extraction\E2\snd"

ttarchext.exe -f "*.anm" 67 "Z:\TWDTTDS\Archives\WDC_pc_WalkingDead303_anichore.ttarch2" "F:\Users\777\Desktop\Telltale Modding\S3_Archives_Extraction\E3\anm"
ttarchext.exe -f "*.wav" 67 "Z:\TWDTTDS\Archives\WDC_pc_WalkingDead303_voice.ttarch2" "F:\Users\777\Desktop\Telltale Modding\S3_Archives_Extraction\E3\snd"

ttarchext.exe -f "*.anm" 67 "Z:\TWDTTDS\Archives\WDC_pc_WalkingDead304_anichore.ttarch2" "F:\Users\777\Desktop\Telltale Modding\S3_Archives_Extraction\E4\anm"
ttarchext.exe -f "*.wav" 67 "Z:\TWDTTDS\Archives\WDC_pc_WalkingDead304_voice.ttarch2" "F:\Users\777\Desktop\Telltale Modding\S3_Archives_Extraction\E4\snd"

ttarchext.exe -f "*.anm" 67 "Z:\TWDTTDS\Archives\WDC_pc_WalkingDead305_anichore.ttarch2" "F:\Users\777\Desktop\Telltale Modding\S3_Archives_Extraction\E5\anm"
ttarchext.exe -f "*.wav" 67 "Z:\TWDTTDS\Archives\WDC_pc_WalkingDead305_voice.ttarch2" "F:\Users\777\Desktop\Telltale Modding\S3_Archives_Extraction\E5\snd"

pause
```

Releasing the mod

When you are ready to release your cutscene mod, you should clear the cache and make the final build (don't forget to disable **Free Camera** and enable **Player mode**).

Don't forget to remove unused code and asset leftovers from the Demo Scene so that you don't bloat your mod with unused data.

Do a final test in-game to see that everything works as intended. Then, you are ready for release.

If you only wish to release the final build of your mod, you simply need to distribute the two **Build** folders, one containing your compiled archives (which would be the **Cutscene_Editor** folder, if you didn't change it for your project) and one containing

the folders with **.ini** files (**Custom_Cutscenes**). These should be enough for your mod to work (**animation_input.ini** is only required for **Editor mode**).

With that said, I highly recommend releasing the full source code alongside the final **Build**, since this will let other players work with your project and use it as reference, or make their own versions of it. Because the Telltale modding community is very small and our knowledge of the games' engine is extremely limited, we highly encourage and promote resource and knowledge sharing. **Tools like this one wouldn't exist if everybody was withholding information and gatekeeping the community like in the early TTG modding days.** Though ultimately it's up to you. Note that people will still be able to decompile your code from the released **Build**, so it's not like you can really prevent them from gaining access to it anyway.

If you wish to include the **Source** in your release, you should include your **Script Editor** project, as well as **animation_input.ini** for Editor mode. I also recommend clearing cache before releasing the **Source**, since it's not needed for final release. You may keep **Build** and **Source**-related files as separate uploads to make it easier to install the mod for people who just want to play it, but make **Source** available for modders and others who wish to have access to it.

I recommend packing up all the necessary files you wish to release and uploading them to the TWDTTDS Nexus page (and/or other file sharing websites of your preference, such as ModDB or Github, though Nexus currently is the most active for TWDTTDS modding) so that others can find your mod more easily.

I also ask that you leave a link to this tool's page on your own mod's page if possible (so that more people can find this tool and learn how to use it), though you don't have to.

Advanced Features

Custom functions

Advanced features section will go into details about the potential of custom functions. Custom functions are optional scripts you can set up for each clip. Custom functions execute once at the start of the clip (at least for now; I plan to expand their basic functionality in the future), however you can also use them to create timed events that happen in the future by adding functions to **OnPostUpdate**. This section assumes general knowledge of coding and understanding of lua language and will be more focused on showcasing different commands you can use in the Telltale's engine to make things work.

An example of a simple custom function would be music_clip_45.

```
music_clip_45 = function()
    controller_sting1 = SoundPlay("sting1_mus_sting_Tense_02");
    controller_music2 = SoundPlay("music2_mus_loop_Tense_30");
    ControllerSetSoundVolume(controller_music2, 0.4);
end
```

This function will play two sounds with **SoundPlay** command, as well as create **controllers** for these sounds. Properties of **controllers** can then be modified through various commands by referencing their **Variable names**. In this case **ControllerSetSoundVolume** command is used to change the volume of the second controller, which is playing the music.

This function will be executed once at the start of **clip 45**, as seen in the **player.ini**:

```
[45]
duration=4.8
choices=0
choice1_text=text
choice2_text=text
choice3_text=text
choice4_text=text
next_clip=46
next_choice1=0
next_choice2=0
next_choice3=0
next_choice4=0
custom_script=music_clip_45
```

_add animations

Another example of a simple custom function is **anm_clip_53**.

```
anm_clip_53 = function()
    anm_clip_53_controller = PlayAnimation("Eli",
"sk61_javierStandA_lookAround_add");
    ControllerSetLooping(anm_clip_53_controller , false);
end
```

Once again, this function creates a controller, but this time for an animation. It applies a "look around" animation to the **Eli** character. It also makes sure this animation only plays once by using the **ControllerSetLooping** command on the

controller and setting it to **false**. If it was set to **true**, the animation would keep replaying until the controller was disabled.

This function executes at the start of **clip 53**, and if you look at this clip in-game, you will see that **Eli** does indeed look around while walking. But why does he keep playing his walking animation (which is set in the **clip ini**) after we make a new controller with a different animation? That's because here we are applying an **_add animation**. These are special types of animations, they have an **_add** postfix in their names and they only affect certain parts of the character (in this case, only the upper body bones). These types of animations allow you to combine multiple animations and play them at the same time, which is what we do here.

Timed events

What if we need to execute certain events at a specific point in time instead of at the start of a clip? Let's take a look at **anm_clip_54** function.

```
anm_clip_54 = function()
    clip_55_done = 0;
    clip_55_time = GetTotalTime() + 3;
    anm_clip_54_controller = PlayAnimation("Rufus",
"sk61_javierTense_lookBehindRight_add");
    ControllerSetLooping(anm_clip_54_controller , false);
    Callback_OnPostUpdate:Add(anm_clip_54_update);
end
```

This function is executed at the start of **clip 54**, and, like in the previous example, it plays an **_add animation** on the **Rufus** character, making him look behind. There is, however, an issue. If we just play this animation, Rufus will indeed turn around, however his eyes will keep being directed forward, which causes a really strange and unnatural look. So, how did I fix this issue in my cutscene?

Many character agents have a special agent attached to them, which determines the position their eyes are looking in. For **Rufus** it's called **obj_lookAtRufusEyes**. Normally, this object's position is automatically reset every frame, meaning that if we want to have a character look consistently into desired direction, we'd need to set this agent's position every frame as well.

This is where we use **Callback_OnPostUpdate:Add** command. This command adds a stated function to the **PostUpdate**, meaning that from now on that function will be executed every single frame. In this case we do this to **anm_clip_54_update** function.

```
anm_clip_54_update = function()
    if clip_55_done == 0 then
        if GetTotalTime() < clip_55_time then
            Custom_SetAgentWorldPosition("obj_lookAtRufusEyes",
Vector(1000, 0, 0), kScene);
        else
            clip_55_done = 1
        end
    end
end
```

This function can now be used to create a timer for our event. In **anm_clip_54** we set up a special variable to determine the timing of execution of our desired code.

```
clip_55_time = GetTotalTime() + 3;
```

GetTotalTime() command returns the amount of time that has passed since the game's launch. By referencing this command and adding a number to it, we can now create a timer for our events, in this case the check will be for when 3 seconds pass after **clip 54** starts playing.

Now, in **anm_clip_54_update** we can create a condition:

```
if GetTotalTime() < clip_55_time then
```

This condition checks if the three seconds have passed, comparing the current **GetTotalTime()** to the time we recorded as **clip_55_time**.

In this case, we're going to be setting **obj_lookAtRufusEyes** agent's position to (1000, 0, 0) for the first three seconds, making the character look in the right direction. We also use **clip_55_done** variable to determine if we still want to execute the code inside **anm_clip_54_update** or not. Setting this variable to 1 prevents the first check of

```
if clip_55_done == 0 then
```

to trigger, meaning that after three seconds the code will not be executed again. Setting this variable up is important because it prevents unnecessary checks executions, which improves performance.

Note that if your clip plays multiple times, the **custom function** attached to that clip will be executed every time. This includes execution of

Callback_OnPostUpdate:Add command, meaning that it will add an additional instance of a function to **PostUpdate** every time it runs. This is undesired because it can mess up your code and cause multiple executions of the same lines *(played sounds, for example, will get louder with each additional function added in, since you would technically be playing the sound multiple times now)*.

There are several ways one can combat this. In the **Demo Scene**, the only time clips may get played more than once is if you fail the QTE. Failing the QTE plays **clip 82**, and then sends you back to **clip 78**. During **clip 78** we need to play several sound effects, meaning that we may want to use a **PostUpdate** function. To prevent this function from being added in the subsequent times the clips are playing, we simply add this function to **PostUpdate** during **clip 77**'s custom function instead.

```
sounds_clip_77 = function()
    local controller_sound = SoundPlay("S4_ZombieBG_Scott_1");
    Callback_OnPostUpdate:Add(sounds_clip_78_update);
end
```

Then we use clip 78's custom function only to set up the timing variables and create/update finish check variables:

```
sounds_clip_78 = function()
    clip_78_click1 = 0;
    clip_78_click2 = 0;
    clip_78_time_click2 = GetTotalTime() + 0.66;
    clip_78_start = 0;
end
```

*Please note that the actual **custom function** that's called from **clip 78** is **gun_visible_clip_78**, which switches weapon visibility, and that function is what calls **sounds_clip_78** at the end. The update-related variables were moved to a separate function for convenience's sake, but it doesn't really matter.*

And then we use the **PostUpdate** function to play sounds based on values of these variables.

```
sounds_clip_78_update = function()
    if clip_78_start == 0 then
        if clip_78_click1 == 0 then
            local controller_sound =
SoundPlay("S4_SFX_Gun_Handle_ClickSlide_04");
            clip_78_click1 = 1;
```

```
        end
        if clip_78_click2 == 0 and GetTotalTime() >
clip_78_time_click2 then
            local controller_sound =
SoundPlay("S4_SFX_Gun_Handle_ClickSlide_04");
            clip_78_click2 = 1;
        end
    end
end
```

Now technically since this **PostUpdate** script is created before **clip 78**, you may want to add additional checks for the mere existence of these variables (checking non-existing variables may break the code), or at least declare them earlier in the code. But in this case it doesn't really matter if the function breaks since it's not going to do much of anything prior to **clip 78** anyway, meaning that we do not have to do that.

Another way to prevent unnecessary **Callback_OnPostUpdate:Add** executions is to simply create a variable that checks if the clip has ever been run before. In **Demo Scene** there are two variables like this and they are first declared at the start of custom functions section:

```
clip_81_played_once = 0;
clip_82_played_once = 0;
```

They are used in clip 81 and clip 82 custom functions to prevent **Callback_OnPostUpdate:Add** commands from triggering more than once.

```
if clip_81_played_once == 0 then
    Callback_OnPostUpdate:Add(clip_81_QTE_update);
    clip_81_played_once = 1;
end
```

```
if clip_82_played_once == 0 then
    Callback_OnPostUpdate:Add(snd_clip_82_update);
    clip_82_played_once = 1;
end
```

Once their values are changed to 1, the code inside this condition won't trigger anymore, and the values of these variables are never set back to 0 again.

Agent position fix

Sometimes certain animations have a strange starting position for the character in **BAKED** or **ROOT MOTION**. In these cases the character may stay in their original position for one frame before being properly controlled by the animation. This seems to be a limitation of Telltale's scripting.

When this happens, it may not look very well in the cutscene, it will look as if the character is teleporting around. To prevent this, I prefer to teleport such characters out of the camera's view for the first frame of the clip.

AN example of this can be seen in **clip 75**.

```
mari_pos_correction_clip_75 = function()
    Custom_SetAgentWorldPosition("Mariana", Vector(0, 0, 0),
kScene); -- Mari one-frame teleport fix
    Custom_SetAgentWorldPosition("Gabe", Vector(0, 0, 0), kScene);
-- Gabe one-frame teleport fix
end
```

Characters are teleported to the position of (0, 0, 0) at the start of the clip. Since this isn't a **PostUpdate** function, it will only run once, and at the next frame the position of the agent will be controlled normally by the tool once again, bringing them back where they belong. You will still have the effect of characters not being present in the scene for 1 frame, but it will look much less jarring than if they were teleporting from one position to the next.

You may try to remove the custom function from the clip to see how it would look otherwise.

Attaching items at runtime

This example shows an alternative way to spawn items/weapons for characters to hold. I do not recommend using this method, but it is an option.

```
spawn_guns_clip_67 = function()
    mari_animation_clip_63_controller = PlayAnimation("Mariana",
"sk62_clementineStandA_toTense");
    ControllerSetLooping(mari_animation_clip_63_controller ,
false);
    ControllerSetContribution(mari_animation_clip_63_controller,
0.6);
    ControllerSetPriority(mari_animation_clip_63_controller, 300);
```

```
    agent_pistol_1 = AgentCreate("Pistol_1", "obj_gunM1911.prop",
Vector(-0.05, 0.778, 0.25), Vector(90,95,0), kScene, false,
false)--DO NOT get these numbers from starting clip
    agent_pistol_2 = AgentCreate("Pistol_2", "obj_gunP250.prop",
Vector(-0.08, 0.87, 0.25), Vector(5,0,-60), kScene, false,
false)--DO NOT get these numbers from starting clip

    Custom_SetAgentWorldPosition("Javier", Vector(0, 0, 0),
kScene);
    Custom_SetAgentWorldPosition("Clementine", Vector(0, 0, 0),
kScene);

    --wrist_L
    --wrist_R
    local nodeName = "wrist_R";
    if AgentHasNode("Javier", nodeName) then
        AgentAttachToNode(agent_pistol_1, "Javier", nodeName);
    end
    if AgentHasNode("Clementine", nodeName) then
        AgentAttachToNode(agent_pistol_2, "Clementine", nodeName);
    end

    controller_music3 = SoundPlay("music_3_mus_loop_Action_09b");
    ControllerSetSoundVolume(controller_music3, 0.4);
    ControllerSetLooping(controller_music3, true);
end
```

This function has quite a lot of stuff in it, but we are only really interested in this part now:

```
agent_pistol_1 = AgentCreate("Pistol_1", "obj_gunM1911.prop",
Vector(-0.05, 0.778, 0.25), Vector(90,95,0), kScene, false,
false)--DO NOT get these numbers from starting clip
agent_pistol_2 = AgentCreate("Pistol_2", "obj_gunP250.prop",
Vector(-0.08, 0.87, 0.25), Vector(5,0,-60), kScene, false,
false)--DO NOT get these numbers from starting clip

Custom_SetAgentWorldPosition("Javier", Vector(0, 0, 0), kScene);
Custom_SetAgentWorldPosition("Clementine", Vector(0, 0, 0),
kScene);
```

```
--wrist_L
--wrist_R
local nodeName = "wrist_R";
if AgentHasNode("Javier", nodeName) then
    AgentAttachToNode(agent_pistol_1, "Javier", nodeName);
end
if AgentHasNode("Clementine", nodeName) then
    AgentAttachToNode(agent_pistol_2, "Clementine", nodeName);
end
```

Here we declare our weapon agents and then attach these agents to character's bones, like before (**Pistol_1** is **Javier**'s gun, **Pistol_2** is **Clementine**'s). We, however, teleport the agents to the position of (0, 0, 0), but we do not touch their rotations. Both position and rotation of the characters during attachment will affect the initial coordinates of item agents required for correct attachment. Please keep that in mind. Other than that, it involved a lot of guesswork, probably even more than it would in the scene set up, because other things in the cutscene may mess something up too.

For example, when testing this function, you should never set its clip as the starting clip (use at least the first the clip before it), otherwise the position of spawned weapons will be different and you will not be testing for correct numbers. I'm not sure what's causing it, but **do not use** starting clip for testing attachment to bones.

Because of this, I think it is better to set up all item stuff in Scene function and then only switch their visibility when needed:

```
gun_visible_clip_78 = function()
    Custom_AgentSetProperty("Pistol_1",  "Runtime: Visible", true,
kScene)
    Custom_AgentSetProperty("Bat",  "Runtime: Visible", false,
kScene)
    sounds_clip_78();
end
```

Another example of spawning items at runtime can be seen in **spawn_knife_clip_70** function, though here we change booth rotation and position of **Mariana**, meaning the item position numbers will have to account for that.

Weapon lights

A custom system was written for this scene to get the firing lights to appear when Javier and Clementine fire their guns. You may use or disregard it at your own accord.

The code for it starts at the beginning of the **custom functions** section. First, we declare the necessary variables.

```
gun_1_check = 0;
gun_2_check = 0;
gun_1_timer = 0;
gun_2_timer = 0;
```

At the start of the scene they are all equal to 0, however we will be making use of them later.

Then we have the actual function, which looks like this.

```
gun_light = function(gun_number)
    if gun_number == 1 then
        local gun_pos = AgentGetWorldPos("Pistol_1");

Custom_SetAgentWorldPosition("flashlightTestLight_point_gun_1",
gun_pos, kScene);
        gun_1_timer = GetTotalTime()+0.05;
        gun_1_check = 1;
    else
        local gun_pos = AgentGetWorldPos("Pistol_2");

Custom_SetAgentWorldPosition("flashlightTestLight_point_gun_2",
gun_pos, kScene);
        gun_2_timer = GetTotalTime()+0.05;
        gun_2_check= 1;
    end
end
```

It takes one parameter, which is simply the number of the gun agent we want to fire from (1 is for **Pistol_1**, 2 is for **Pistol_2**) What this function does is get gun agent's world position through **AgentGetWorldPos** command, and then moves one of the two lights we spawned during scene set up to that position with **Custom_SetAgentWorldPosition**.

We also change the corresponding check variable to 1, to show that we initialize the firing of said gun, and change the gun's timer to **GetTotalTime()+0.05** *(current time + 0.05 seconds).*

Finally, there is the **PostUpdate** function, which we also called during scene set up.

```
gun_light_update = function()
    if gun_1_check == 1 then
        if GetTotalTime() > gun_1_timer then

Custom_SetAgentWorldPosition("flashlightTestLight_point_gun_1",
Vector(0, -1000, 0), kScene);
            gun_1_check = 0;
        end
    end

    if gun_2_check == 1 then
        if GetTotalTime() > gun_2_timer then

Custom_SetAgentWorldPosition("flashlightTestLight_point_gun_2",
Vector(0, -1000, 0), kScene);
            gun_2_check = 0;
        end
    end
end
```

This function checks if each of the guns is supposed to fire and then checks if the set time of GetTotalTime()+0.05 has passed, and if it did, it moves the related light to (0, -1000, 0), which would be far underneath the map, and sets the check variable back to 0, preventing further activation.

This is a **PostUpdate**, meaning it will scan for changes in the check variables every frame. This means that in order to have the firing light effect trigger, we just need to execute the **gun_light** function (with 1 or 2 as a parameter to select the gun) at any desired point in the scene. There are plenty of examples of this function being used, most complex probably being **sounds_clip_68_update**.

Music

Playing music is as simple as setting up a sound controller and then setting its volume and other desired parameters. For example, an ambient and a music controller is initiated in the **moods** function.

```
controller_amb = SoundPlay("S3_AMB_Richmond_Alley_Dusk");
ControllerSetLooping(controller_amb, true);
ControllerSetSoundVolume(controller_amb, 0.3);
ControllerFadeIn(controller_amb, 2.0);

controller_music1 = SoundPlay("music1_mus_loop_Neutral_21");
ControllerSetSoundVolume(controller_music1, 0.3);
```

Setting **ControllerSetLooping** to true makes your controller play in a loop. **ControllerSetSoundVolume** lets you control the volume level. And **ControllerFadeIn** lets you set up a fade in effect with desired length.

Another very useful function **ControllerKill**, which lets you destroy your controller and stop music from playing further (works with **animation controllers** too). *It may not work if you set your controller variable as a local variable.*

For example, in shot_clip_85_update we use it to kill the playback of the controller_music3 controller during a timed event.

```
ControllerKill(controller_music3);
```

QTE

The QTE in the Demo Scene happens during **clip 81**, and consists of two parts - the visual part and the actual code that controls it. The visual part is a special agent which is created during the moods function (which is the **custom function** of **clip 0** aka the starting clip).

First we spawn the QTE agent from **ui_nextTimeOn_titleLogo.prop**. It's a UI prop but it will spawn as a mesh in 3D space. This is the prop used for S3, if you make cutscenes in other seasons, use **ui_boot_title.prop** instead.

```
--QTE set up
agent_demo_scene_qte = AgentCreate("agent_demo_scene_qte",
"ui_nextTimeOn_titleLogo.prop", Vector(0, 0, 0), Vector(0,0,0),
Custom_CutsceneDev_SceneObject, false, false)
```

Next thing to do is to change its texture to our custom one (**demo_scene_qte_e.d3dtx**) using **ShaderSwapTexture** command. Note that if you are using **ui_boot_title.prop**, the reference to the original texture should be changed from **ui_nextTimeOn_titleLogo.d3dtx** to **ui_boot_title.d3dtx** as well. And you may use the **ui_boot_title.d3dtx** file located in

cutscene_editor\Cutscene_Editor folder as a base for any of your custom textures for both cases if you wish.

```
ShaderSwapTexture(agent_demo_scene_qte,
"ui_nextTimeOn_titleLogo.d3dtx", "demo_scene_qte_e.d3dtx");
```

Next we simply set up a bunch of properties which will make our agent always render on top of everything else and generally function more like a UI element than a 3D object.

```
--set properties, transparency, etc
Custom_AgentSetProperty("agent_demo_scene_qte", "Render Depth
Test", false, Custom_CutsceneDev_SceneObject)

Custom_AgentSetProperty("agent_demo_scene_qte", "Render Depth
Write", false, Custom_CutsceneDev_SceneObject)

Custom_AgentSetProperty("agent_demo_scene_qte", "Render Depth
Write Alpha", false, Custom_CutsceneDev_SceneObject)

Custom_AgentSetProperty("agent_demo_scene_qte", "Render Layer",
95, Custom_CutsceneDev_SceneObject)
```

Finally, we attach this object to our cutscene camera:

```
AgentAttach("agent_demo_scene_qte", agent_name_cutsceneCamera);
```

It will now follow the camera along.

Then we modify the scale of our agent for each axis by affecting this property:

```
Custom_AgentSetProperty("agent_demo_scene_qte", "Render Axis
Scale", Vector(1.8,1,1), Custom_CutsceneDev_SceneObject);
```

We really just want to stretch the agent along the **X axis** for **1.8** of the base size to make it look better.

Then we place the agent at the correct position in front of the camera, so that it will be seen in the cutscene:

```
Custom_SetAgentPosition("agent_demo_scene_qte", Vector(0,-4.5,22),
Custom_CutsceneDev_SceneObject);
```

```
Custom_SetAgentRotation("agent_demo_scene_qte", Vector(0,180,0),
Custom_CutsceneDev_SceneObject);
```

Note that we are using **Custom_SetAgentPosition** and **Custom_SetAgentRotation** instead of **Custom_SetAgentWorldPosition** and **Custom_SetAgentWorldRotation**. These special functions allow us to move the agent in relation to its **parent agent** (agent it was attached to, in this case the camera).

Finally, we make the agent invisible for now.

```
Custom_AgentSetProperty("agent_demo_scene_qte",  "Runtime:
Visible", false, kScene)
```

All we have to do now to make the QTE appear on the screen is to set this property to true, which is done in **spawn_bat_clip_81** and changed back to false in subsequent scripts when the player succeeds or fails the QTE.

All the actual code of the QTE is done through the **clip_81_QTE_update**. It is a pretty standard timed event function, but I would like to bring attention to how it registers **E** button click:

```
elseif Custom_InputKeyPress(69) then -- Check if E is pressed
    QTE_done = 1;
    Custom_AgentSetProperty("agent_demo_scene_qte",  "Runtime:
Visible", false, kScene)
end
```

Custom_InputKeyPress command lets you check if a button is pressed. The number you put in as a parameter corresponds to different keys on the keyboard.

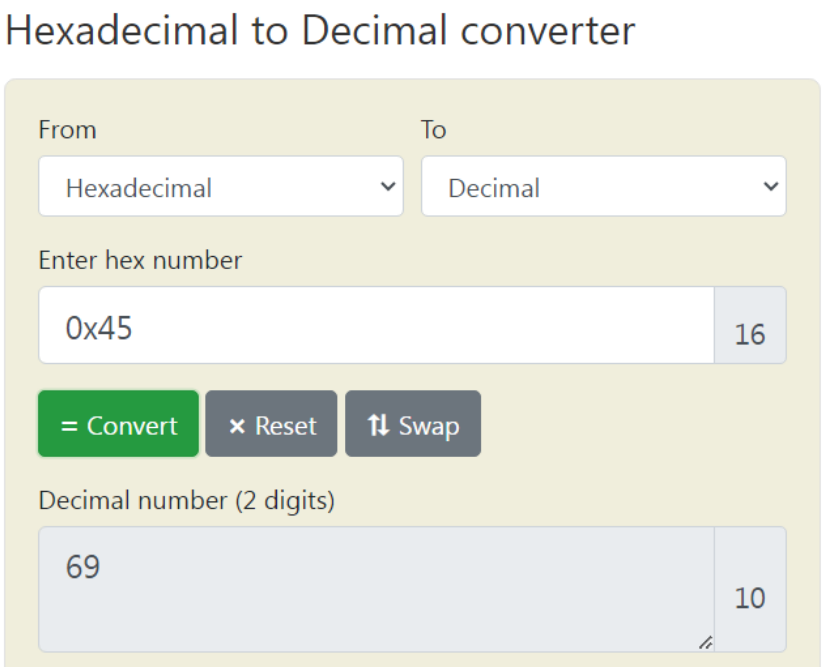
To see the full list of available keys check here:
https://learn.microsoft.com/en-us/windows/win32/inputdev/virtual-key-codes

To actually convert their code to a number, use this calculator:
https://www.rapidtables.com/convert/number/hex-to-decimal.html

For example, the **E** key has a code of 0x45.

0x44	D key
0x45	E key
0x46	F key
0x47	G key
0x48	H key

And the actual number of **E** key is **69**:



Let's also look at how the actual clip change occurs:

```
if GetTotalTime() > clip_81_time and QTE_done == 1 then
    chosen_text = 1;
    clip_81_done = 1;
elseif Custom_InputKeyPress(69) then -- Check if E is pressed
    QTE_done = 1;
    Custom_AgentSetProperty("agent_demo_scene_qte",  "Runtime:
Visible", false, kScene)
end
```

If the QTE detects the button press, and the timing conditions are met, the code changes the **chosen_text** variable to **1**. This causes the clip to change to **clip 83**.

Otherwise, if the QTE requirements are not met, at the end of the clip we switch to **clip 82**.

How does that work? Well, the **chosen_text** variable is an internal variable normally used by the **Player** during choice selection. The variable equals **0** by default, but if a choice selection occurs, it will change to a number from **1** to **4** (based on which choice was picked), making the clip switch one of the clips listed in **player.ini** for this choice. The **chosen_text** variable, of course, resets back to **0** at the start of any new clip.

For example, in this clip picking the first choice will switch **chosen_text** to **1**, and this will tell the **Player** to switch to **clip 48**. Picking the second choice will switch **chosen_text** to **2**, and this will tell the **Player** to switch to **clip 49**. Picking the third choice will switch **chosen_text** to **3**, and this will tell the **Player** to switch to **clip 50**. And picking the fourth choice is not possible because this clip only has 3 choices available, but if it had four, you would be able to change **chosen_text** to **4** by picking the last choice. And if no choices are made, **chosen_text** remains at **0** and **Player** proceeds normally to **clip 50** after this clip ends.

```
[47]
duration=6
choices=3
choice1_text=Follow me!
choice2_text=Come here.
choice3_text=...
choice4_text=text
next_clip=50
next_choice1=48
next_choice2=49
next_choice3=50
next_choice4=0
```

While this system was made for dialogue choices, we can manually change **chosen_text** during any clip to make it immediately switch to a clip corresponding to **chosen_text**'s value. If we look at the section of **clip 81**, code will become much more clear:

```
[81]
duration=3.7
choices=0
choice1_text=text
choice2_text=text
```

```
choice3_text=text
choice4_text=text
next_clip=82
next_choice1=83
next_choice2=0
next_choice3=0
next_choice4=0
custom_script=spawn_bat_clip_81
```

Here we can see that despite there being no choices in this clip, **next_choice1** variable is set to **83**. This means that if at any point during this clip **chosen_text** variable changes to **1**, the **Player** will switch to **clip 83** immediately after it happens. Which is exactly what we do through our **PostUpdate** QTE function. You can use this knowledge of the tool to create your own QTE events.

Mood manager

I have made a simple **mood manager** for the Demo Scene to showcase a possible way to apply mood-related animations.

First in the **moods** function we set up our variable and call the **PostUpdate** function.

```
mood_clip = 0;
Callback_OnPostUpdate:Add(moods_update);
```

This variable will be used to detect active clip change.

In **moods_update** we see the following system:

```
moods_update = function()
    if player_clip ~= mood_clip then
        mood_clip = player_clip;
        pcall(ControllerKill, mood_clem)
        pcall(ControllerKill, mood_javi)
        pcall(ControllerKill, mood_mari)
        pcall(ControllerKill, mood_gabe)

        if player_clip == 7 then
            mood_gabe = PlayAnimation("Gabe",
"gabe_face_moodShockA");
            ControllerSetLooping(mood_gabe , true);
            ControllerSetContribution(mood_gabe, 0.6);
```

```
        elseif player_clip == 26 then
            mood_gabe = PlayAnimation("Gabe",
"gabe_face_moodAngryA");
            ControllerSetLooping(mood_gabe , true);
            ControllerSetContribution(mood_gabe, 0.6);
        elseif player_clip == 27 then
            mood_mari = PlayAnimation("Mariana",
"clementine_face_moodWorryA");
            ControllerSetLooping(mood_mari , true);
            ControllerSetContribution(mood_mari, 0.6);
        elseif player_clip == 28 then
            mood_gabe = PlayAnimation("Gabe",
"gabe_face_moodAngryB");
            ControllerSetLooping(mood_gabe , true);
            ControllerSetContribution(mood_gabe, 0.6);

...

        elseif player_clip == 79 then
            mood_javi = PlayAnimation("Javier",
"javier_face_moodPainC");
            ControllerSetLooping(mood_javi , true);
            ControllerSetContribution(mood_javi, 0.6);
        end
    end
end
```

The **player_clip** variable is another internal **Player** variable that holds the number of the current function. By comparing our variable to this one we can see if there were any clip changes.

The general idea for this mood manager is that every time a clip changes we first run a bunch of pcall commands to Kill all of the possible the **mood animation controllers** (we use pcall because we can never be sure if these controllers are running in the given clip or not), and then check for which clip is currently playing and apply the desired animations to the desired characters though creating **mood animation controllers**. *I understand that this code is crude and unoptimized, it's just a quick example I threw together as a showcase, and it can be improved upon.*

In-game you may notice that sometimes the mood animations are a bit jittery. I'm not 100% sure why is that, but I suspect they have a bit of a conflict with lip sync animations, which also sometimes control facial animation bones. I also use the **ControllerSetContribution** here to smooth out the effects of this issue - this

command determines how prevalent the animation of the controller is. Another possible way to fix this would be to apply .chore mood-related files instead.

Another command that may come in use when playing animations is ControllerSetPriority, which changes the priority of a controller compared to other controllers (you can see an example of it in **spawn_guns_clip_67**).

Persistent data

Usually, you can use **player.ini** to create different reactions to conversation choices, or even different branches of your cutscene. However, in some cases it may be desirable to externally store some of this choice data in order to reference it later. This can be done through creating your own **.ini files**.

In case of the **Demo Scene**, we use **persistent.ini**. You can, of course, use different files if you desire. We will be using the **LIP** component functions to save our data into the file.

The contents of persistent.ini are pretty simple:

```
[persistent]
path=1
```

There is only one section and one variable that may be equal to either 1 or 2. You may have more variables depending on your needs. But you must make sure that the .ini files you are using actually exist (create them) and have all of the variables set up with default values. Otherwise LIP will not be able to record their data.

Now let's look at how this is realized in code. First, two empty arrays are created:

```
persistent_data = {};
persistent_data_check = {};
```

First one will be used to save our data to the file, and the second one - to retrieve it later in the scene.

The clip where the actual choice of path happens is **clip 9**.

```
[9]
duration=10
choices=4
choice1_text=You let them brand you?!
choice2_text=You were with those monsters?!
```

```
choice3_text=I can't believe you lied to me...
choice4_text=...
next_clip=15
next_choice1=10
next_choice2=12
next_choice3=13
next_choice4=15
```

The clips this clip can lead to are **10**, **12**, **13** and **15**. If you look at their sections, you will see that each of them has a custom function of either **path_1** or **path_2**. These are the functions that record the choice we made in **clip 9**. Lets take a closer look at **path_1** function:

```
path_1 = function()
    local persistent_data_save = {path = 1};
    persistent_data.persistent = persistent_data_save;
    local persistent_data_path =
"Custom_Cutscenes/demo_cutscene/persistent.ini";
    LIP.save(persistent_data_path, persistent_data);
end
```

The algorithm to save this data to the file is this:

1) We create a local array with a variable (or multiple ones) and its value.
2) We then put that array as a third layer subarray of the **persistent_data** variable. The second level of subarray is a section name for our file, in this case **[persistent]**.
3) We then use the LIP.save command to actually record the data into the file we specify.

The script for path_2 works similarly, just sets up a different value for the variable.

```
path_2 = function()
    local persistent_data_save = {path = 2};
    persistent_data.persistent = persistent_data_save;
    local persistent_data_path = "Custom_Cutscenes/" ..
player_name_of_the_cutscene .."/persistent.ini";
    LIP.save(persistent_data_path, persistent_data);
end
```

In this function we are actually referencing the **player_name_of_the_cutscene**, an internal **Player** variable which holds the name of the cutscene folder, instead of writing the entire path directly.

The check for which path is taken happens in **clip 21** through **check_path** function.

```
check_path = function()
    check_done = 0;
    check_path_time = GetTotalTime() + 3;
    Callback_OnPostUpdate:Add(check_path_update);
end
```

This function is used to set up the **PostUpdate** function called **check_path_update**.

```
check_path_update = function()
    if check_done == 0 then
        if GetTotalTime() > check_path_time then
            local persistent_data_path = "Custom_Cutscenes/" ..
player_name_of_the_cutscene .."/persistent.ini";
            persistent_data_check =
LIP.load(persistent_data_path);

            if persistent_data_check.persistent.path == 1 then
                chosen_text = 1;
            else
                chosen_text = 2;
            end
            check_done = 1;
        end
    end
end
```

What this function does is wait for a certain amount of time to pass, and the load the recorded data as a **persistent_data_check** array through **LIP.load** command.

It then makes a check for what the path variable equals to, and switches the clip similarly to how we did it in the QTE section (by changing the **chosen_text** variable).

Finally, if we take a look at the section of **clip 21**, we'll see the follow up clips of **22** and **24** set up as next_choice values.

```
[21]
duration=10
```

```
choices=0
choice1_text=text
choice2_text=text
choice3_text=text
choice4_text=text
next_clip=24
next_choice1=22
next_choice2=24
next_choice3=0
next_choice4=0
custom_script=check_path
```

Also worth pointing out that the time for this clip is set to be significantly longer than it actually is intended to play, since we will be controlling the time of switching to one of the next clips through **check_path_update**.

Return to main menu

Clip 88 is the last clip of the Demo Scene, and its function (**cutscene_end_clip_88**) will return the player to the main menu. It is as simple as writing one line:

```
cutscene_end_clip_88 = function()
    SubProject_Switch("Menu", "Menu_main.lua")
end
```

The first parameter here is what resources should be used as a base for the environment, and the second one is the name of the script file to run.

Similar commands can be used to switch between different scenes (for the first parameter set the episodic archive you wish to load the scene from, for example you can set it to "WalkingDead202" for S2E2-based scenes).

.chore files

These files are action files that may be applied in a scene. I avoided using them because we can't really edit them in any way as of writing this tutorial (though it is expected that **The Telltale Inspector** will support them at some point), or control them properly. Once they are applied, they are permanent and their effects will remain (unless you use a .chore to cancel them out, if such exists, or if it is not made to be permanent). However, they can be used to achieve certain effects otherwise impossible or hard to do.

The FCM scripts include a custom function that you can use to play a chore specifically on an agent:

```
Custom_ChorePlayOnAgent(chore, agentName, priority, bWait)
```

You may also take advantage of the native Telltale commands such as **ChorePlay**, **ChorePlayAndWait** and **ChorePlayAndSync**.

Examples of usage:

```
ChorePlay("ui_alphaGradient_show")
ChorePlay(chore, priority, "default", agentName);
ChorePlayAndWait("ui_menuMain_hide")
ChorePlayAndWait(chore, priority, "default", agentName);
ChorePlayAndSync("ui_menu_ambientFadeIn", mControllerAmbient)
```

Other projects for references

Despite managing to write this tool, my Telltale scripting knowledge is limited. Other people have figured out many different secrets in their scripting projects, and released their code for everyone to see. I highly recommend studying their code for references and commands you can use in your own projects:

- **First Cutscene Mod** by *David Matos* - the first ever custom cutscene mod. This tool uses a lot of code from this mod, though there are still techniques of interest present there.
- **Load Any Level** by *droyti* - mod which allows you to load and play any vanilla scene in the game.
- **Relight** by *David Matos* - graphical/lighting improvements done through scripting.
- **No Outlines** by *David Matos* - disables character outlines.
- **Menu Rain** by *David Matos* - modification of the main menu.
- **Kenny The Boat Master** by *David Matos* - a complex minigame where you drive the boat as Kenny, done entirely in-engine.

Special Thanks

Kas (PrincessPeachFan2014) - for coming up with, writing down and keeping track of the entire sequence of events and dialogues in the Demo Scene, for making incredibly helpful suggestions and critiques of my work during development, and for showing interest and support for the development of this tool.

David Matos (frostbone25) - for pioneering Telltale scripting and helping me understand it, along with many other important things about Telltale modding.

Violet (droyti) - for creating the original Script Editor.

Lucas Saragosa - for helping me and others understand and edit Telltale's formats, and for amazing and continued contributions to the modding community.