

User Guide

uMod 2.0 Exporter

Modding support made easy

Trivial Interactive

Version 2.3.x

If you are a game developer then you will know how important it is to ensure that the lifetime of your game is as long as possible. One of the proven methods of extending a games lifespan is to add the ability to mod the game, which allows the community to extend and customize the games content. uMod 2.0 is a system that allows you to do just that.

uMod 2.0 is a complete modding solution for the Unity game engine and makes it quick and painless to add mod support to your game. Modders are able to extend to and modify gameplay by creating mods with assets, scripts and even entire scenes, all within the Unity editor. The uMod 2.0 Exporter means that modders are able to use the intuitive user interface of the Unity Editor to create their content and then export to mod format in a single click.

Features

- Basic mod support out of the box
- Support for PC, Mac and Linux platforms
- Supports all assets that Unity can handle, Yes! even scenes and scripts can be included.
- Supports loading from the local file system or a remote server
- Supports command line launching of mods
- Supports multi-mod loading
- Modded content can be created in the Unity editor and exported using our customizable build pipeline
- C# scripts or assemblies can be included in mods
- Script execution security allows developer to restrict modded code
- Customizable build pipeline for exporting mods
- Mod tools builder for generating game specific modding tools
- And many more features...

Contents

<u>INSTALLING.....</u>	<u>3</u>
INSTALL.....	3
UNINSTALL	3
UPDATING	3
<u>MOD ESSENTIALS.....</u>	<u>4</u>
MOD STRUCTURE.....	6
<u>EXPORT REQUIREMENTS.....</u>	<u>7</u>
EXPORTABLE CONTENT	7
REQUIRED CONTENT	7
IMPLEMENTING THE INTERFACE	8
<u>EXPORTER.....</u>	<u>9</u>
CREATING MODS	9
EXPORT SETTINGS	10
MOD.....	11
BUILD	12
BUILD AND RUN	13
EXPORTING MODS	15

Installing

Install

When installing uMod Exporter 2.0 into an existing project you should first create a backup of your Unity project as a precaution in case anything goes wrong. It is better to be safe than sorry. Once you have created a backup of the project, you can import the .unitypackage into the project as you would normally.

Once the package has imported you should see a folder named 'UModExporter' which has been added. This is the root folder for uMod Exporter and contains all content associated with the exporter tool.

Often developers will prefer to group all of their purchased or downloaded plugins into a sub folder in order to keep their project organised. You are able to move the root UModExporter folder to any location you like however there are a couple of things to note:

1. You should never rename the root 'UModExporter' folder in any case otherwise certain aspects of the plugin may fail or cause undesirable behaviour.
2. You should never "reorganize" the contents of the 'UModExporter' folder. In order for all functionality of uMod 2.0 to work as expected, the sub folder structure needs to remain the same.

Uninstall

There is no dedicated uninstaller for uMod Exporter 2.0 so if you need to uninstall it then you should do so manually. You can do this by simply deleting the root 'UModExporter' folder from its install location. By default this will be "Assets/UModExporter".

Note: *User preferences may remain even though the package has been deleted but this will not affect your project in any way.*

Updating

When updating uMod Exporter 2.0 it is recommended that you first ensure that any previous versions are removed. To do this take a look at the previous topic 'Uninstall'. Once you have removed the older version of uMod Exporter you can then import the updated version. Take a look at the previous 'Install' topic for more detail.

Hopefully you should have no issues on importing the updated package but if there are then It is recommended that you first take a look at the changelog to see if the problem is simply down to a feature change. If you are still having trouble after updating then you can contact support and we will help you the issue sorted.

Limitations

uMod 2.0 attempts to be as close as possible to Unity in terms of behaviour and usage but there are a few limitations where uMod cannot work in the same way as Unity.

Scriptable Objects

Scriptable objects are partially supported by uMod 2.0 but the scriptable object type will need to be known at compile time by both the uMod runtime and the exporter. This means that any scriptable objects whose type is defined in mod code that is compiled as part of the build process will not work correctly and any references to these objects will resolve to null. The easiest way to support scriptable objects is to create an interface assembly (see [Mod Scripting](#) for more information) where you will define the scriptable object type and then ensure that this assembly is inside the game project as well as inside the exporter project. You will then be able to create scriptable object assets from this type as part of the mod which will be loadable at runtime.

Script Serialization

uMod 2.0 will attempt to serialize all scripts attached to scene objects or prefabs as closely as possible to the way Unity would but there are some limitations to this system. The main limitation is that MonoBehaviour component references are not supported by uMod 2.0 meaning that the following code will cause the public variable 'exampleReference' to resolve to null when loaded regardless of what was assigned in the editor prior to export:

C# Code

```
1 using UnityEngine;
2
3 public class ExampleComponent : MonoBehaviour { }
4
5 public class Example : MonoBehaviour
6 {
7     // Unity would serialize this correctly but uMod will not
8     public ExampleComponent exampleReference;
9 }
```

Instead you should reference the game object that has the component you want to reference and then use 'GetComponent' in order to get a reference. Here is the revised example:

C# Code

```
1 using UnityEngine;
2
3 public class ExampleComponent : MonoBehaviour { }
4
5 public class Example : MonoBehaviour
6 {
7     private ExampleComponent exampleReference;
8
9     // uMod will serialize an object reference correctly
10    public GameObject exampleObject;
11
12    public void Awake()
13    {
14        // Store the reference in awake
15        exampleReference =
16        exampleObject.GetComponent<ExampleComponent>();
17    }
18 }
```

Note: uMod 2.0 will serialize built in Unity component references correctly. This limitation only applies to custom scripts that inherit from MonoBehaviour as uMod treats them specially when serializing the data.

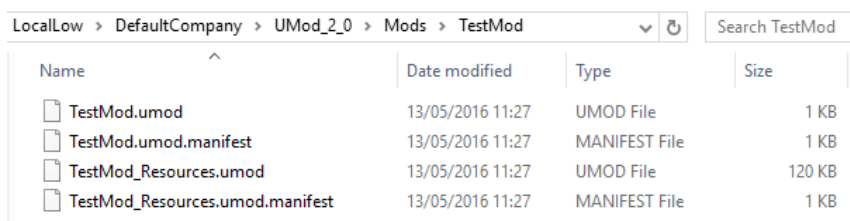
Mod Essentials

The following section will cover the essential concepts used by uMod 2.0. It is highly recommended that you read this section fully to ensure you understand the way uMod works.

Mod Structure

When you export a mod using the uMod 2.0 Exporter you will see a directory which has the name of the mod that was specified during export. This directory is known as the 'Mod Folder' and is where all the essential mod files are located. All sub-files and folders are associated with the mod and may or may not include the mod name in their file name.

The image below shows a typical mod folder after a successful export:



The screenshot shows a Windows File Explorer window with the address bar set to 'LocalLow > DefaultCompany > UMod_2_0 > Mods > TestMod'. The search bar contains 'Search TestMod'. The main area displays a table of files and folders:

Name	Date modified	Type	Size
TestMod.umod	13/05/2016 11:27	UMOD File	1 KB
TestMod.umod.manifest	13/05/2016 11:27	MANIFEST File	1 KB
TestMod_Resources.umod	13/05/2016 11:27	UMOD File	120 KB
TestMod_Resources.umod.manifest	13/05/2016 11:27	MANIFEST File	1 KB

Figure 1

The mod directory will contain a number of files depending upon the settings used during export, however there are 2 main files that are particularly important and must be present in order for the mod to load successfully. These files are:

- [ModName].umod: This is a lightweight file that contains meta data about the mod and the content that it includes. The file is typically used to quickly retrieve information about the mod without actually loading the mod.
- [ModName]_Resources.umod: This is the main data file for the mod and contains all the exported content.

Note: [ModName] represents the name of the mod and should be replaced with the actual name of the mod. For example, a mod called 'TestMod' will have 'TestMod.umod' and 'TestMod_Resources.umod' files located in its mod folder.

Any other files that are located in the mod folder will generally contain additional nonessential information and the files may be disregarded if required. These files will be used by the build engine during the export process to determine what type of content is included in the mod, however if they are not present then they are simply ignored.

Export Requirements

The following section will cover the requirements of the exporter tool in order for it to function correctly.

Exportable Content

The uMod Exporter tool is able to compile almost any type of unity asset into mod format so that it can be loaded dynamically at runtime. There are however a few things to consider before you think about exporting.

uMod 2.0 defines three different types of mod content which may be included in any combination. These types are:

- **Prefabs:** If you are familiar with Unity then you will know that prefabs are a sort of asset template from which game objects can be created. They are a vital asset in Unity and are supported by the exporter, however the way in which you reference them from code differs slightly. This is discussed later in the documentation.
- **Scenes:** Unity scenes may also be exported directly and allow entire map mods to be created where entire game levels are modded into the game to provide alternate scenarios.
- **Scripts:** Scripts can be included in the mod but they must be precompiled into a managed assembly in order to pass validation. This process is handled automatically by the exporter when compiling a mod and any script references and inspector values are serialized into a custom format. This means that once a mod object is loaded into the game, any referenced scripts will automatically be attached just as if they were never removed, retaining all their inspected values in the process. This process is known as dynamic re-linking and is the entire reason scripting is a viable option for modding. Without it scripts would simply be a collection of methods with all data set to default values.

Required Content

In order for a mod to export successfully it may or may not have to contain certain files depending upon the type of content that is included. If the mod relies in any way upon behaviour scripts then you will need to ensure that you implement the 'IMod' interface which provides entry and exit callbacks to the mod. The IMod interface defines three methods that must be implemented which are shown below:

C#	Code
1	<code>public interface IMod</code>
2	<code>{</code>
3	<code> void OnModLoaded();</code>
4	<code> void OnModUpdate();</code>
5	<code> void OnModUnload();</code>
6	<code>}</code>

The purpose of each method is outlined below:

- **OnModLoaded:** The main entry point to the mod which is called just after the mod has been successfully loaded and initialized.

- **OnModUpdate:** A method that is called every frame (Where possible) similar to Unity's 'Update' method, however the script does not necessarily need to be attached to a game object in order to receive this event. This is useful for pure scripting mods that need to check conditions periodically.
- **OnModUnload:** Called just before the mod will be unloaded from memory. It provides a last chance to perform clean-up tasks or to serialize and mod data before execution stops.

Note: *There is no limitation to the number of scripts that implement the IMod interface. All scripts that implement the interface will in turn receive all three events. For more information about the IMod interface take a look at the [uMod 2.0 Scripting Reference](#)*

Implementing the Interface

The IMod interface must be implemented for modded scripts to load correctly but there are a number of different ways which you can implement this interface. uMod provides a number of classes that themselves implement this interface meaning that inheriting from these classes also means implementing the interface. This method also has the added bonus of allowing you to override the method required since all derived class leave these events exposed as virtual methods. These scripts are:

- **ModScript:** A base class designed to be inherited by modded scripts when a game object is not required. The script will run without the need for a parent object but will not receive the typical MonoBehaviour events such as Start or Update. The script also exposes a number of properties that can be used to communicate with the game and includes functionality to load assets from the mod and send messages to game scripts.
- **ModScriptBehaviour:** A base class to be inherited by modded scripts that provides the same members as the ModScript class. The difference is that the ModScriptBehaviour inherits from the MonoBehaviour class and as such you will receive all the events you would expect. This script should be used when you need a script component that receives mod events.
- **ModInherit:** A base class to be inherited by modded scripts which inherits directly from ModScriptBehaviour. This script is used along with an attribute and is intended to replicate inheritance of a game script by a modded script. The mod script is essentially able to extend the game script.

If you inherit from any one of these base classes then you are automatically implementing the interface. The method you choose is best decided by working backwards through that list asking the following questions.

1. Do I need inheritance - If so, use ModInherit
2. Do I need to act upon a game object - If so, use ModScriptBehaviour
3. Do I need to be able to load mod prefabs or communicate with game scripts in any way - If so, use ModScript
4. If you don't need any of the above then simply implement the IMod interface.

Exporter

Creating Mods

In order to export modded assets using the uMod Exporter, you will first need to create a mod where you will place all these assets. To keep things simple, the uMod exporter allows a folder under the Unity assets folder to be selected as a mod folder. This mod folder can then be selected for export.

There are a few requirements to creating mod folders which are:

- Mod folders cannot be a sub-folder of the 'uModExporter' folder. This will cause errors during the build as the exporter will attempt to export locked files.
- Mod folders must contain at least one or more valid assets types. If no supported assets are found then the build will fail.
- Mod folders must be named uniquely. Unity will not allow folders with the same names so this is not really an issue.

To keep the export process as simple as possible, uMod includes a basic wizard that allows you to create new mods in a few simple steps. By creating a mod using the wizard you will generate a dedicated mod folder located under the assets folder or the folder specified. The wizard will also update the export profile configuration so that the newly created mod is set as the active export target. To launch the wizard simply go to the menu: 'Tools/uMod Exporter/Create/New Mod':

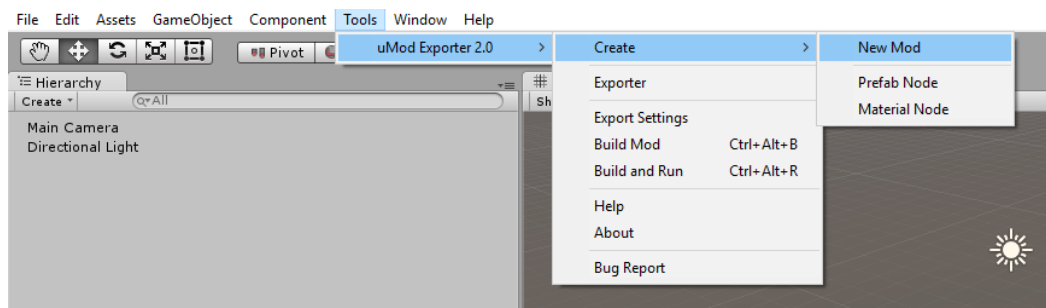


Figure 2

When the wizard window opens you will be required to enter a mod name that is unique and optionally you can specify the location where the mod folder will be created. If the specified name is already in use by another mod folder then you will not be able to continue and will receive a warning message towards the bottom of the window. The window also has the option 'Set as active build target'. By default this is enabled and will modify the export configuration so that the resulting mod will be the target for exporting.

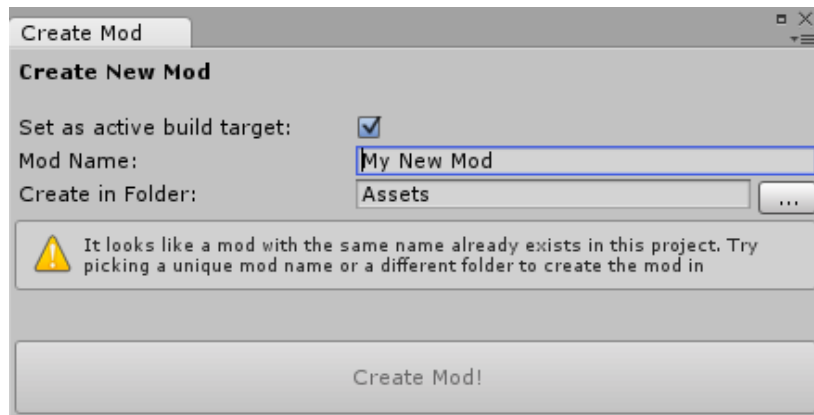


Figure 3

Once you have entered a unique name you can create the mod which will result in a mod folder being created. The following image shows a typical mod folder created by the wizard.

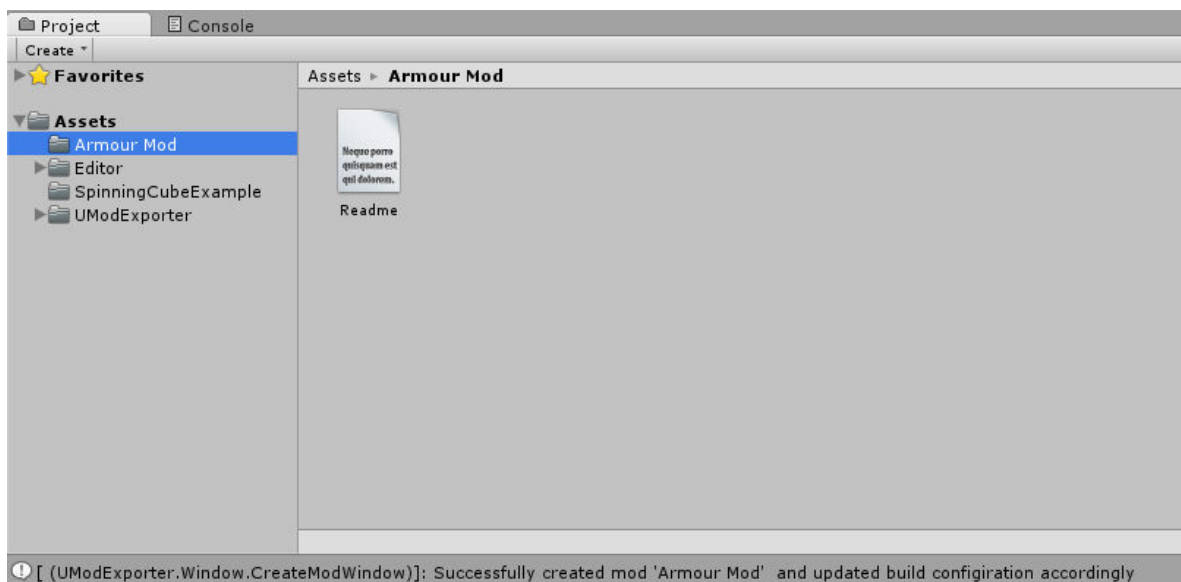


Figure 4

Export Settings

The uMod Exporter includes a dedicated settings window where you are able to specify which mod should be exported along with custom build settings such as the log level used. The settings window can be accessed by selecting 'Tools/UMod Exporter/Export Settings' which will open up an editor window as shown below:

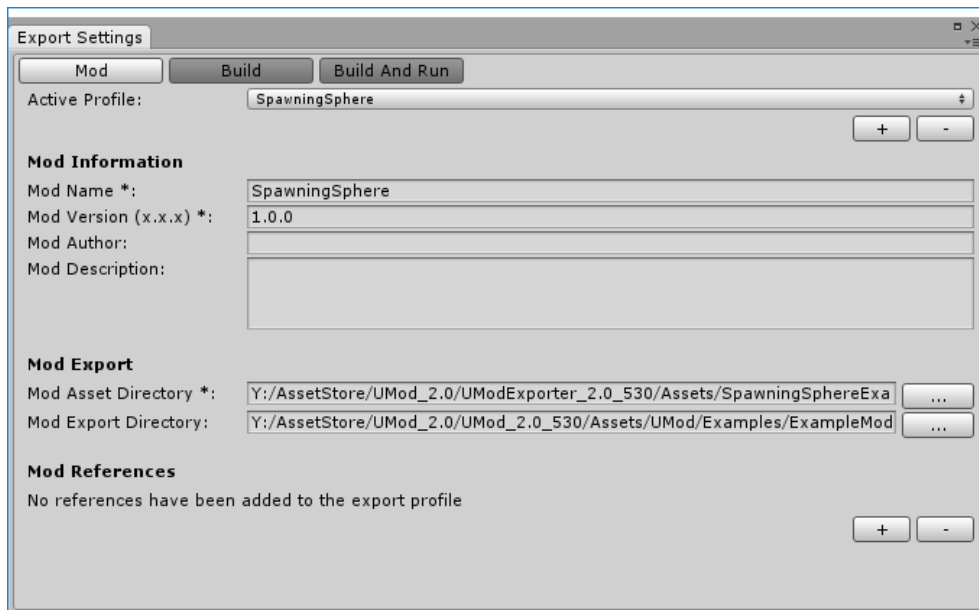


Figure 5

There are a number of tabs in the settings window that segment the various settings into appropriate categories. Each category is outlined below:

- **Mod:** This tab is responsible for managing the current export settings used by the build engine. You are able to switch the active mod folder as well as rename mods.
- **Build:** This tab contains more advanced settings that are used during the export to change the export behaviour. For most users, the settings in this tab should remain at their default settings.
- **Build And Run:** This tab allows modders to setup quick test functionality by specifying where the game executable is located. When a successful export has finished, the build engine will attempt to launch the game with the exported mod loaded.

Mod

This section will cover the various settings of the mod tab and how they alter the export process:

- **Active Profile:** uMod exporter allows you to create a number of export profiles targeting different mod folders so you can conveniently maintain a number of mods within the same project. The active profile option allows you to set the mod that will be exported. You can add and remove export profiles using the '+' and '-' buttons under the drop down menu.
- **Mod Name:** This is the name that the mod will have when exported. This name can differ from the name of the mod folder in which case it will replace the mod folder name.
- **Mod Version:** This is the 3 value version of the mod, for example '1.0.0'. If a mod update is released, this value should be increased to a suitable value so that the game is able to load the newest version of the mod.
- **Mod Author:** This field should specify the name of the person or company that created the mod. The field is not required but it is recommended that you include a name.

- **Mod Description:** This field should include a brief description of the mod and what it does. This value may be used by the game to display a list of mods to the user.
- **Mod Asset Directory:** This field should point to the mod folder that you want to export. The mod folder should be a folder located under the Assets directory that includes all mod assets.
- **Mod Export Directory:** This field should point to the location where the compiled mod should be created. Note that the compiled mod will consist of a directory with the same name as specified in the Mod Name field.
- **Mod References:** uMod allows mods to reference other mods which allows access to the referenced mods assets scenes and code. You can use the references section to add or remove references to pre-exported mods using the '+' and '-' buttons respectively. Note that when adding a reference uMod must be able to find and load all of its dependencies.

The mod tab is able to identify invalid or unacceptable content and provide warnings to help ensure that valid information is provided. The following image should be the mod tab where unacceptable data has been entered. You will not be able to export the mod until the settings window is provided with appropriate data.

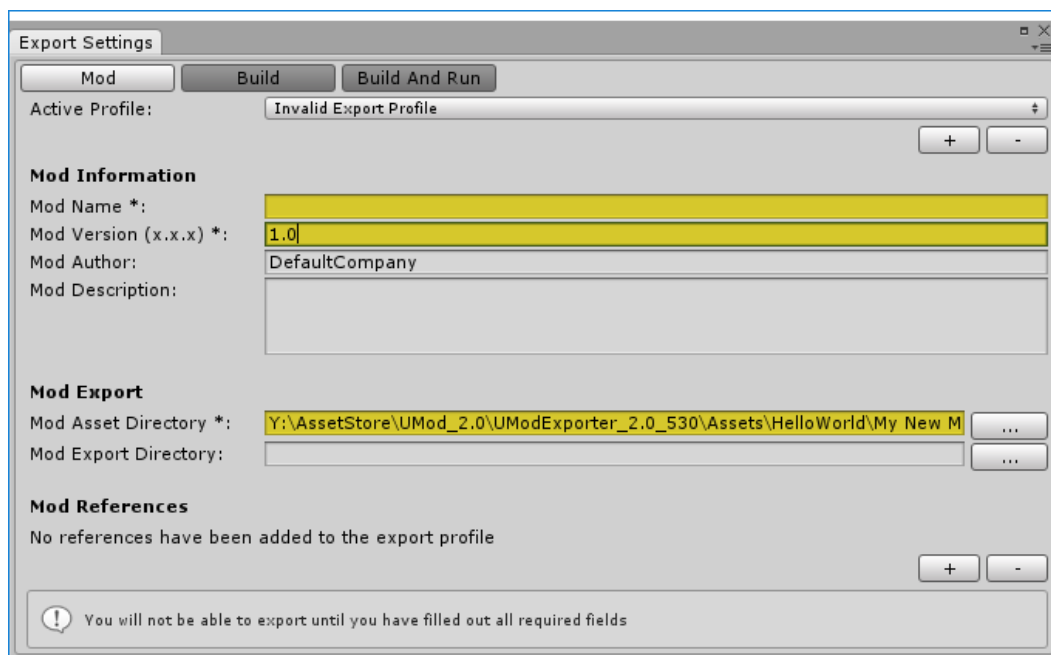


Figure 6

Build

This section will cover the settings of the build tab. These settings should only be modified by advanced users as they can change the export behaviour.

- **Log Level:** This value specifies how detailed the build information is while exporting. By default the value is set to warnings meaning that only warning information or more severe will be logged.

- **Optimize For:** This value can be used to trade-off build time for output size. The default value is set to build time meaning that the build engine will attempt to export the mod as fast as possible but may generate larger files.
- **Exporter Mode:** This value can be used to change the appearance of the exporter window. The reduced mode hides the build log window while exporting allowing the exporter window to become much smaller.
- **Compile Scripts:** This value determines whether the build engine will attempt to compile any scripts located in the mod folder. If any asset relies on one or more scripts then this option will need to be enabled otherwise you may receive linking errors or missing type errors in game. If there are no scripts in the project then script compilation can be disabled to allow for quicker build times.
- **Clear Console on Build:** When enabled, the Unity console will be cleared prior to export so that logged messages can be viewed more easily.
- **Show Output Directory:** When enabled, the build engine will show the exported mod in its folder if the build is successful.
- **Generate Manifest Files:** When enabled, the build engine will generate additional meta data files containing information about the compiled assets.

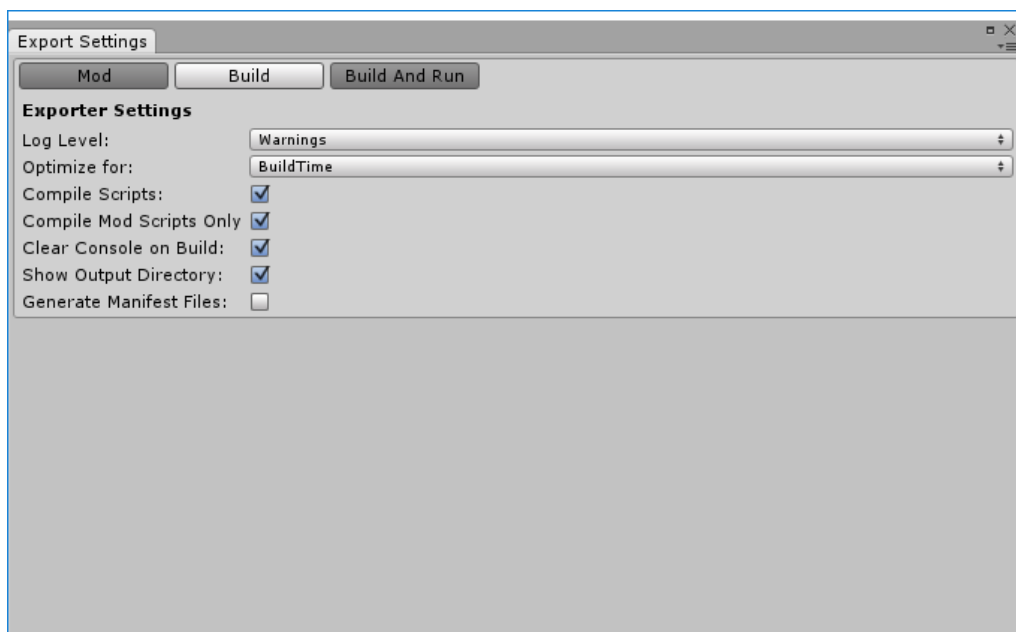


Figure 7

Build and Run

This section will cover the build and run tab which can be used for quick testing during mod development. Build and run functionality can be very powerful but it may not be supported by the game you are targeting. This feature requires command line launching of mods to be supported which must be enabled by the developers so before proceeding we recommend that you check with the games documentation as to whether command line launching is supported. You may also need

to modify the command line format used by the game which should also be documented by the developers.

- **Game Executable:** The path to the game executable that you are modding. Currently only windows platforms are supported for build and run so the executable type will be .exe files.
- **Command Line Symbol:** This value is a formatted string that specifies how the target game expects the command line arguments to be passed. You are able to use the macro '%PATH%' (case sensitive) to specify in the format string where the mod path will be inserted. For example, if we want to launch a mod at 'C:/Mods/MyMod' using the default command line format of '+mod=%PATH%', the build engine will translate that string into '+mod=C:/Mods/MyMod' meaning that the game is able to access the full path to the mod.

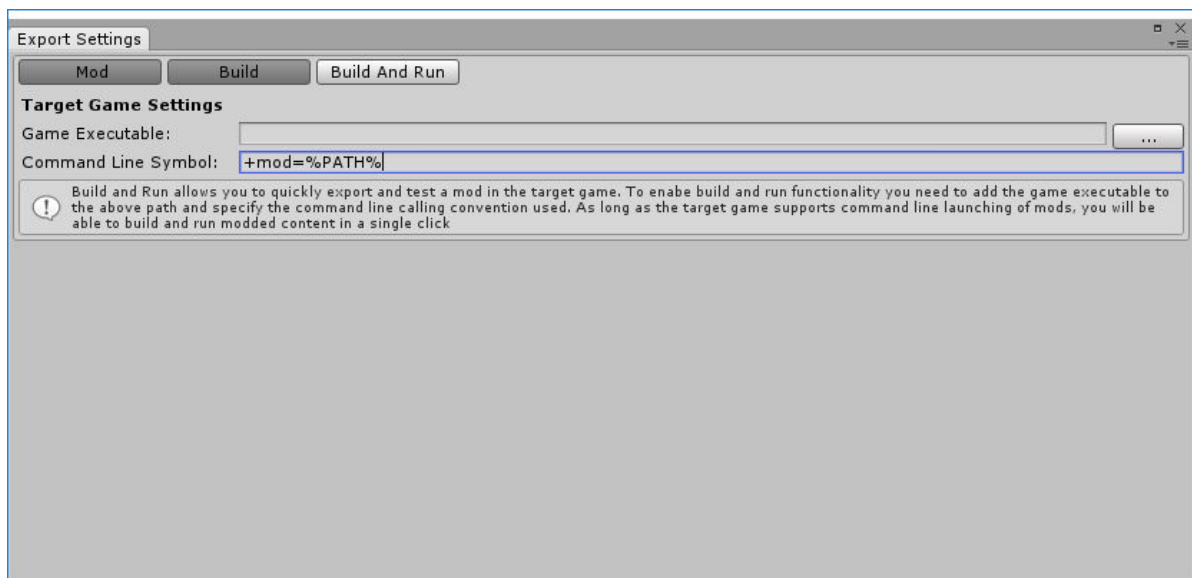


Figure 8

Exporting Mods

Once you have configured the export settings you are able to export mods at the click of a button. Simply open up the exporter window by selecting 'Tools/UMod Exporter/Exporter' and then click 'Build Mod'. This will begin the process of compiling all assets in the active mod folder and exporting them into mod format. You will only be able to export the mod if the export settings are valid. If the settings are not valid then you will be prompted to correct the settings before continuing the build.

Note that the exporter window also allows you to change the active export target via the drop down menu before exporting.

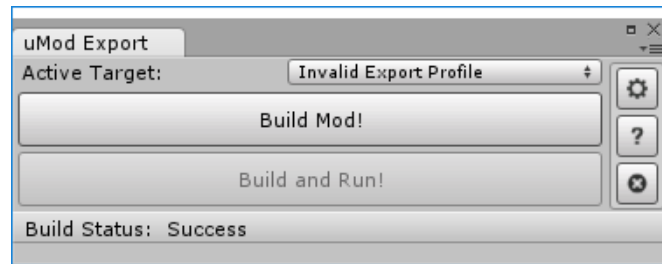


Figure 9

You are able to begin building a mod by using keyboard shortcuts. If you are building and testing mods very frequently then it can improve the workflow. The shortcuts are listed below:

- **Build Mod:** (CTRL + ALT + B) This will open the exporter window and launch a build using the current settings.
- **Build and Run:** (CTRL + ALT + R) This will open the exporter window and launch a build using the current settings. If the build is successfully then the target game will be launched with the build mod loaded.