User Guide

uMod 2.0

Modding support made easy

Trivial Interactive

Version 2.8.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
- Moddable content system for applying custom mod materials, textures and meshes to game assets
- And many more features...

Contents

Installing

Install

When installing uMod 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 'UMod' which has been added. This is the root folder for uMod and contains all content associated with uMod.

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 UMod folder to any location you like however there are a couple of things to note:

1. You should never rename the root 'UMod' 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 'UMod' folder. In order for all functionality of uMod 2.0 to work as expected, the sub folder structure need to remain the same.

Updating

Before updating uMod 2.0 it is recommended that you take a look at the Upgrade Guide section to check for any breaking changes that may affect you. We would always recommend using the latest uMod version however we understand that this may not be practical for some projects where upgrading may break months of work.

When updating uMod 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 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.

Uninstall

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

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

Upgrade Guide

The following section lists any major breaking changes between uMod versions.

uMod 2.6.4

This version adds major changes to the build tools system and the mod tools builder wizard content page has been modified. The content page must be setup with a modding folder path before any content can be added to the package build. See the ‘Mod Tools Builder -> Content’ section for more information.

uMod 2.6.x

This version does not contain any breaking changes from the previous version and introduces the new moddable content system.

uMod 2.5.x

This version does not contain any breaking changes from the previous version.

uMod 2.4.x

This version includes many major breaking changes that will cause existing code that uses the uMod API to generate compiler errors. Mods are now also created as a single file instead of a folder meaning that mods created with previous versions of the exporter are not loadable by this version. All the breaking changes are listed below:

General Changes

- uMod exporter is no longer compatible with this version. Instead you will now build a custom mod tools package that you will distribute to your modders to create mods for your game. This allows for much more configuration by the developer.
- Major changes to uMod file format. uMod now uses a single mod file instead of a folder and multiple file solution as used previously.
-

Code Changes

- Many types have been removed, replaced or moved to a different namespace. Take a look at the API scripting reference for details.
- Changed the structure of UModSettings.
- Mod.Initialize is no longer required.
- ModPath has been remove and ‘System.Uri’ is now used instead.
- Changed the BuildEngineService.BuildMod arguments. We also recommend that you now use ‘ModToolsUtil.StartBuild’ and ‘ModToolsUtil.StartBuildAndRun’ if you are building mods via code.
- Custom build engine processors will need to use the new attribute ‘UMod.BuildPipeline.UModBuildProcessorAttribute’.

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.

IL2CPP

IL2CPP is not currently supported by uMod 2.0 and you will receive errors when trying to build a player using this backend. Instead you should use mono with .Net framework or .Net standard.

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 defined 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

Major improvements have now been made to the uMod serialization system and we think that we have successfully mirrored the Unity behaviour for script, component and game object serialization and referencing. If you find a case where our asset differs from the expected Unity behaviour then please let us know by reporting a bug and we will look into it.

Unity Preview

This section will cover some of the preview features of the Unity engine and whether they are supported or not.

ECS

ECS is currently not officially supported however it can be used with a bit of work and will allow your modders to create mods using Unity ECS.

To allow ECS to work properly you will need to ensure that 'TypeManager.Initialize' or more specifically 'InitializeAllComponentTypes' is called after all mod load calls. This is an essential step because uMod will load the necessary assemblies for a mod during its loading process which will cause it to be activated in the current AppDomain. If the type manager initializes before a mod is loaded then that mod will not have had its types registered with the ECS system and will not work correctly. This is due to the fact that the type manager will search the current app domain during initialization to find all ECS related types and any types not registered beforehand will not be found.

Take a look at this useful forum post for more information: https://forum.unity.com/threads/all-componenttype-must-be-known-at-compile-time.766322/#post-5104754

You can ensure that a method is called before type initialization by registering a runtime initialize method which will be executed at startup. If mods are loaded within this method then everything ECS related should work as expected although that may change in future Unity version.

C# Code

```csharp
using UMod;
using UnityEngine;

class Example
{
[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.SubsystemRe
gistration)]
    static void OnRuntimeLoad()
    {
        Mod.Load(new Uri("C:/Mods/MyExampleMod.umod"));
    }
}
```

It may also be possible to delay the automatic world creation which would trigger the type manager to initialize using the define symbol '`#UNITY_DISABLE_AUTOMATIC_SYSTEM_BOOTSTRAP_RUNTIME_WORLD`' although this would need to be investigated further. Delaying may be desirable if you need to load a mod later in the startup procedure. There is some useful information in the ECS package docs relating to this which can be found here: https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/world.html?_ga=2.33221246.1123019591.1569075385-2077434320.1567177935

With thanks to Unity forum user vestigial

Quick Start

If you want to get uMod up and running as quick as possible then this following section will cover the bare minimum required in order to achieve this. Before continuing it is recommended that you take a look at the Mod Essentials section in order to better understand how uMod works although it is not essential.

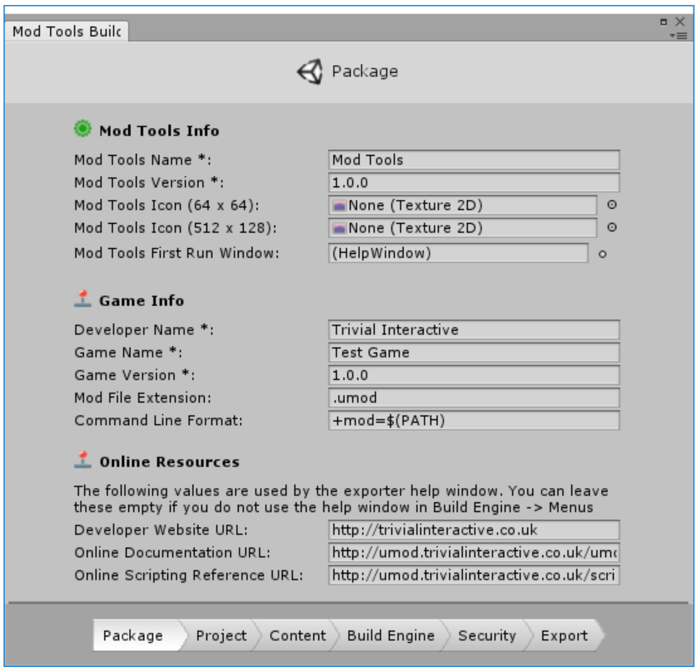
Import Package

Import the package from the asset store. See the previous section 'Installing' for details.

Build Tools Package

Once you have imported the uMod package into your project you will want to create a mod tools package that you will use to create mods for your game. In previous version of uMod you would make use of the free uMod Exporter tool also available on the asset store to create your mods but as of version 2.4 the exporter is no longer used.

First you will need to open the Mod Tools Builder window by going to the menu 'Tools -> uMod 2.0 -> Mod Tools Builder' which will open the following window:



Mod Tools Info

Mod Tools Name *:	Mod Tools
Mod Tools Version *:	1.0.0
Mod Tools Icon (64 x 64):	None (Texture 2D)
Mod Tools Icon (512 x 128):	None (Texture 2D)
Mod Tools First Run Window:	(HelpWindow)

Game Info

Developer Name *:	Trivial Interactive
Game Name *:	Test Game
Game Version *:	1.0.0
Mod File Extension:	.umod
Command Line Format:	+mod=$(PATH)

Online Resources

The following values are used by the exporter help window. You can leave these empty if you do not use the help window in Build Engine -> Menus

Developer Website URL:	http://trivialinteractive.co.uk
Online Documentation URL:	http://umod.trivialinteractive.co.uk/um
Online Scripting Reference URL:	http://umod.trivialinteractive.co.uk/scri

Figure 1

You will then need to fill out the required fields which are marked with '*' and continue through the wizard by clicking the next breadcrumb button in the window footer. Most settings can be left at default for testing purposes but if you need more information about the settings available then take a look at the Mod Tools Builder section.

When you get to the export section of the wizard you will need to specify the file location where the mod tools package will be created or you can tick the option to build the tools in the current project which will setup your current Unity project so that you can create mods. Simply press the export button when done and then wait for the process to complete and when finished you should end up with a mod tools .unitypackage.

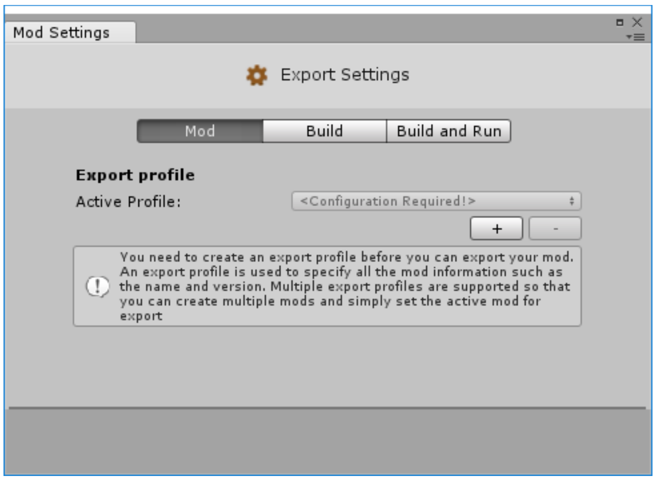
Note: You can recreate the mod tools package at any time using the mod tools builder

Export a Mod

Once you have built your mod tools package your next step will be to create a test mod to make sure everything is working as expected. If you did not select the option to build the tools in the current project in the previous step you will need to create a new Unity project and then import your mod tools package.

uMod 2.0 includes a couple of simple example mods which can be used for this testing. You can find the assets for these mods under the following folder 'Assets/UMod/Examples/ExampleModsRaw'. If you are using your mod tools from the current package you will need to move these raw mod assets outside of the UMod folder and to a more suitable location (usually the root assets folder). If you are using another Unity project to create your mods you will need to select the folder for the raw mod assets that you want to test with and right click to select export package. This will create another package that you can import into your modding Unity project. Once imported into the modding project make sure that the assets are not located inside the UMod folder.

You will now need to create an export profile for your test mod. Go to the menu 'Mod Tools -> Export Settings' and the following window should appear:



Export profile

Active Profile: <Configuration Required!>

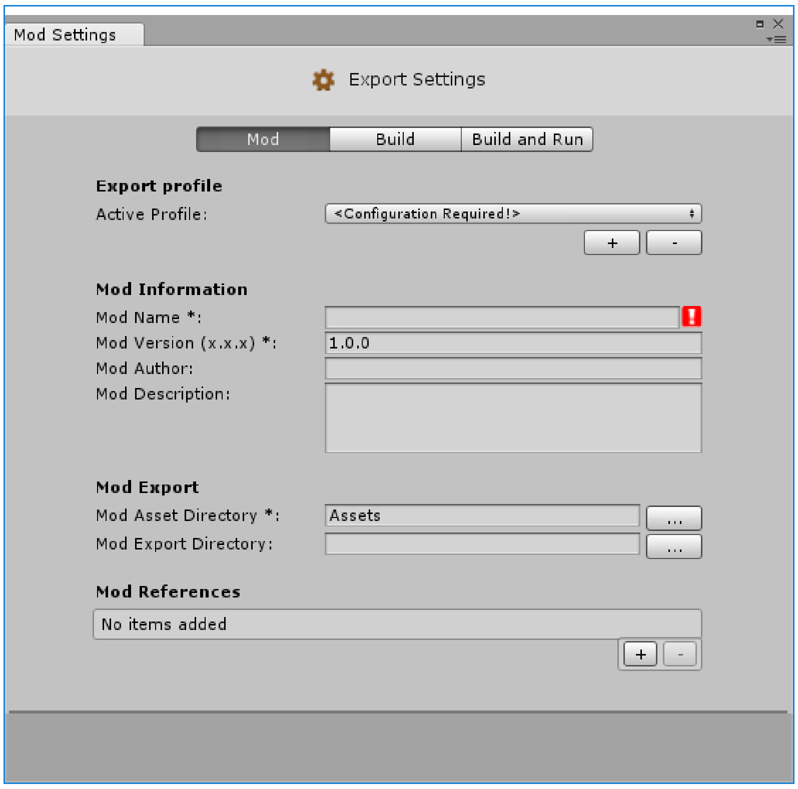
You need to create an export profile before you can export your mod. An export profile is used to specify all the mod information such as the name and version. Multiple export profiles are supported so that you can create multiple mods and simply set the active mod for export

Figure 2

To create an export profile click the '+' button under the 'Active Profile' item in order to create a new mod export profile. This will create cause new fields to be created where you can enter all the required Meta information about the mod. You will need to fill out the required fields as a minimum which are indicated with an '*'. If any data is invalid you will see an error icon appear to the right of the field.

For the 'Mod Asset Directory' field you should select the raw assets folder that we imported or moved previously as this is the folder which will be exported.

Once you have entered all the necessary information simply close the window to save the settings.



Export profile

Active Profile: <Configuration Required!>

Mod Information

Mod Name *:

Mod Version (x.x.x) *: 1.0.0

Mod Author:

Mod Description:

Mod Export

Mod Asset Directory *: Assets

Mod Export Directory:

Mod References

No items added

Figure 3

You are now ready to build the mod which can be done by going to the menu 'Mod Tools -> Build Mod'. The process will take a few seconds and upon completion the exported mod should be revealed in the file explorer or finder.

Load a Mod

After the mod has been built you are now read to load it into the game. Take a look at the 'Mod Loading' section to learn how to load and manage mods within your game.

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.

Install Mods

uMod 2.0 is able to load mods from any location, provided that the application has sufficient security privileges. It is partially up to the developers to determine where the best place to install mods should be. Typically this will be a specific folder located in either the local machine settings or under the games installation folder and once mods are placed inside that folder, they will be recognised by the game.

We recommend that the developer uses the persistent data path that is accessible via Unity at runtime along with a suitable sub folder. For example:

"C:/Users/<UserName>/AppData/LocalLow/<CompanyName>/<GameName>/Mods"

Note: The above path is valid on a windows desktop device and other platforms may vary. If you decide to use the persistent data location to store mods then the below example can be used to access a data location, regardless of the platform.

The following C# code shows how the persistent data path can be accessed at runtime using the Unity API:

C# Code

```csharp
using UnityEngine;
using UModHost;

public class ModPathExample : MonoBehaviour
{
    private void Start()
    {
        // Get the app data path for the current operating system
        string appDataPath = Application.persistentDataPath;

        // Append a 'Mods' folder to the path
        string modInstallPath = Path.Combine(appDataPath, "Mods");

        // Create a mod path for 'Example Mod'
        ModPath path = new ModPath(Path.Combine(modInstallPath,
"ExampleMod"));

        // Begin loading the mod
        Mod.Load(path);
    }
}
```

Mod Host

uMod 2.0 uses containers known as mod hosts to manage a single mod within the game which are represented as game objects.

A mod host is essentially a dedicated manager for a specific mod and is responsible for the entire lifetime of that mod from creation until it is unloaded. There may be any number of mod hosts in the scene at any time unless 'Allow Multiple Mods' has been disabled in the settings window, in which case only a single host can exist. Mod hosts may also be re-used to load a different mod and any subsequent calls to 'loadMod' will force the host to unload its current mod (if any).

Mod hosts are known as state objects as they can be in many different states depending upon the operations performed on them. Each state is outlined below:

- Unloaded (Default): The mod host contains no managing information about a specific mod. It is in a clean state and ready to take on management of a mod via a load call.
- Loading: The host is currently loading a mod and as such cannot perform operations such as load or unload.
- Loaded: The host has its assigned mod loaded and is currently managing it.

These are the main states of a mod host and the following diagram shows how a host may transition to these states. Note that the 'OnDestroy' event triggered by destroying the script or game object for the host will result in the host automatically unloading its current mod if one is loaded.

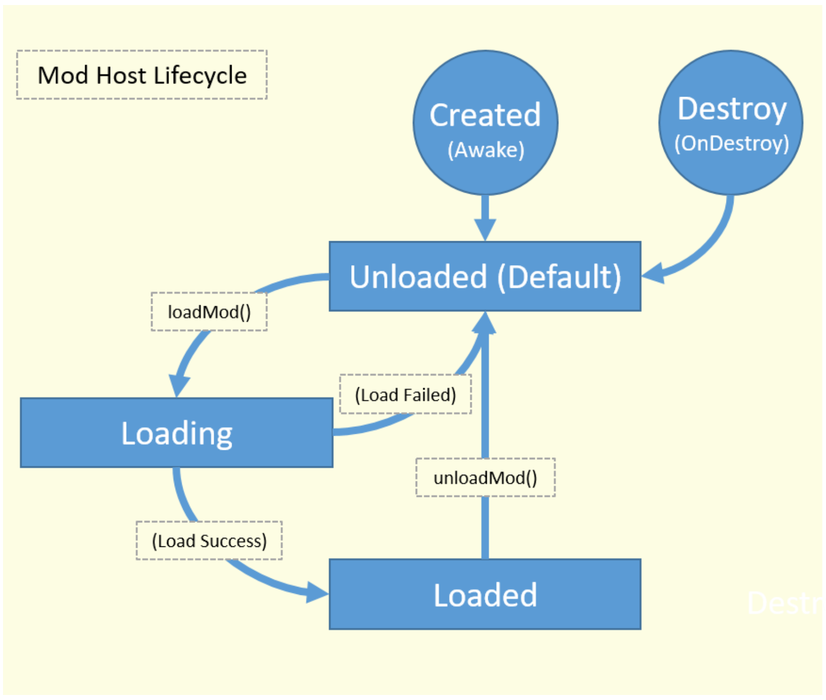


Figure 4

Note: *Mod hosts are recyclable objects and may be reclaimed if they are in their unloaded state as a direct result of a new host request. If you want to manage hosts manually and ensure that they are not 'collected' then you should set the 'CanBeReclaimed' property on that specific host to false.*

Mod hosts are implemented as unity components and as such can be attached to a game object in order to create an instance. While this is a valid method of creating a host, we recommend that hosts are created via the static 'CreateNewHost' method which will recycle any idle hosts as well as creating a dedicated object for the host and its sub systems.

The following code shows how a new host can be created via script:

C# Code

```csharp
using UnityEngine;
using UMod;

public class ModHostExample : MonoBehaviour
{
    private void Start()
    {
        // Create the mod host using the static method
        ModHost host = ModHost.CreateNewHost();
    }
}

```

Note: *Mod host creation will usually be managed by a load or async load request meaning that it is not necessary to create hosts manually unless you need more control over the load process.*

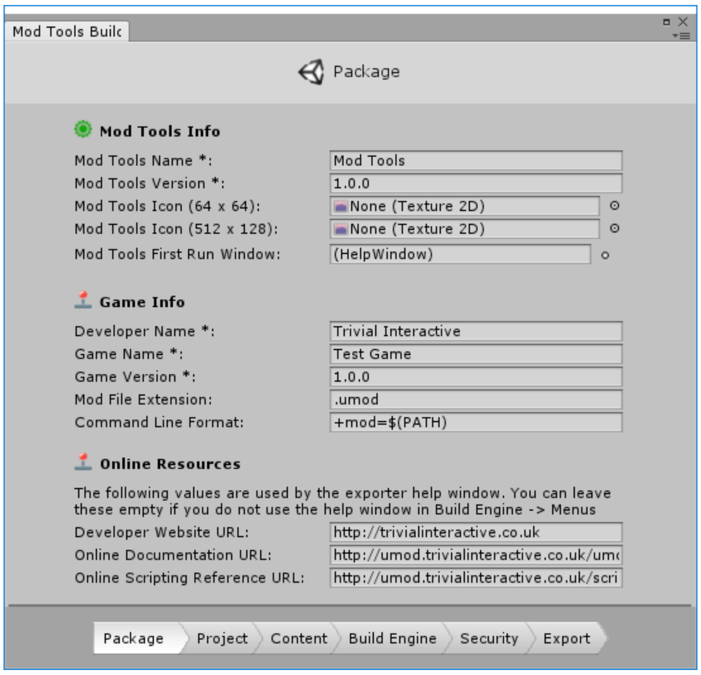
Mod Tools Builder

As of version 2.4.x, uMod now makes use of a dedicated mod tools builder wizard instead of using the generic uMod exporter package. The mod tools builder is able to create a custom .unitypackage which you will then need to distribute to your modders in any way you choose. This means that developers can have far more control over many aspects of the modding process by enforcing edit time security restrictions and other preferences. To open the mod tools builder simply go to the menu 'Tools -> uMod 2.0 -> Mod Tools Builder' and you should be presented with the following window:

The tools builder is setup as a wizard which will guide you through the various settings that you can change in order to customize your mod tools package. Each step deals with various related settings which are explained below:

Package

The package page deals with the main package settings, some of which are required in order to build a mod tools package. The information you enter here will be used throughout the mod tools package.



Mod Tools Info

Mod Tools Name *:	Mod Tools
Mod Tools Version *:	1.0.0
Mod Tools Icon (64 x 64):	None (Texture 2D)
Mod Tools Icon (512 x 128):	None (Texture 2D)
Mod Tools First Run Window:	(HelpWindow)

Game Info

Developer Name *:	Trivial Interactive
Game Name *:	Test Game
Game Version *:	1.0.0
Mod File Extension:	.umod
Command Line Format:	+mod=$(PATH)

Online Resources

The following values are used by the exporter help window. You can leave these empty if you do not use the help window in Build Engine -> Menus

Developer Website URL:	http://trivialinteractive.co.uk
Online Documentation URL:	http://umod.trivialinteractive.co.uk/um
Online Scripting Reference URL:	http://umod.trivialinteractive.co.uk/scri

Figure 5

Mod Tools Info

- **Mod Tools Name (Required Field):** The name of the mod tools package. The generated .unitypackage will use this name and the root folder in the package will also use this name.
- **Mod Tools Version (Required Field):** The current version of the mod tools package. The version should be 3 or more numbers separated by '.', For example: '1.2.4'.

- **Mod Tools Icon:** A small icon for your mod tools package that is used by the default uMod exporter windows. This icon should be 64x64 pixels in size for best results. If no icon is specified then the default uMod icons will be used.
- **Mod Tools Icon:** A large icon for your mod tools package that is used by the default uMod exporter windows. This icon should be 512x128 pixels in size for best results If no icon is specified then the default uMod icons will be used.
- **Mod Tools First Run Window:** A window that should be shown the first time the mod tools package is imported into a Unity project. This window will only be shown the very first time the Unity project is loaded. You can change the window by selecting the thumb button to the right of the field which will allow you to view all supported tool windows. You can also create your own tool windows if required which is described In the Custom Tools Window section.

Game Info

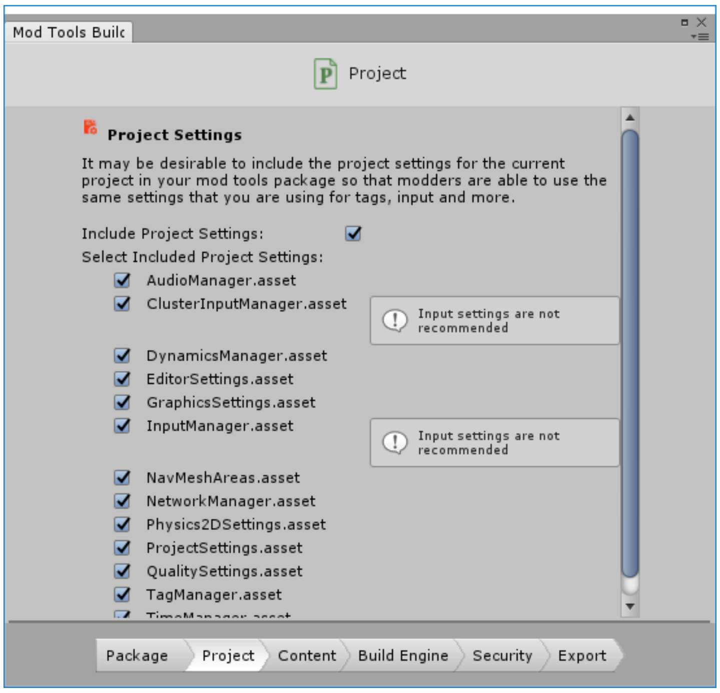
- **Developer Name (Required Field):** The name of the developer. This is usually your company name or personal name in the case of single developers. This value is used for game locking if enabled and crediting in the default uMod exporter about window.
- **Game Name (Required Field):** The name of the game you are creating. Usually this will be the same as your Unity product name which can be found in player settings. This value is used for game locking when enabled.
- **Game Version (Required Field):** The version of the game you are creating. The version should be 3 or more numbers separated by '.', For example: '1.2.4'. This value is used for game locking when enabled.
- **Mod File Extension:** The file extension that all uMod files will be given. By default this is set to '.umod' but you may like to change this to suit your game. Note that an extension must be provided in order for some of the runtime API's such as 'ModDirectory' to work correctly.

Online Resources

- **Developer Website URL:** This URL should point to your company or personal website and is used by the default uMod exporter help window. It is intended to allow your modders to have quick access to your company website.
- **Online Documentation URL:** This URL should point to any online modding documentation that you may create. We highly recommend creating your own modding documentation for your modders, especially with the introduction of tailored mod tools packages but you can always link the generic uMod modders documentation if needed.
- **Online Scripting Reference URL:** This URL should point to any online scripting references you may create for you modders which will usually contains an API overview. Again we recommend that you create your own scripting reference for your modders especially if you create any custom mod API's for mod-game communication.

Project

The project page is used to specify whether you would like to include any of your current project settings with the mod tools package. If you game relies on tags or layers for core mechanics then it may be desirable to include these project assets so that your modders will have access to the same tags and layers and other project settings that you include.



Project Settings

It may be desirable to include the project settings for the current project in your mod tools package so that modders are able to use the same settings that you are using for tags, input and more.

Include Project Settings: ✓

Select Included Project Settings:

- ✓ AudioManager.asset
- ✓ ClusterInputManager.asset — Input settings are not recommended
- ✓ DynamicsManager.asset
- ✓ EditorSettings.asset
- ✓ GraphicsSettings.asset
- ✓ InputManager.asset — Input settings are not recommended
- ✓ NavMeshAreas.asset
- ✓ NetworkManager.asset
- ✓ Physics2DSettings.asset
- ✓ ProjectSettings.asset
- ✓ QualitySettings.asset
- ✓ TagManager.asset

Figure 6

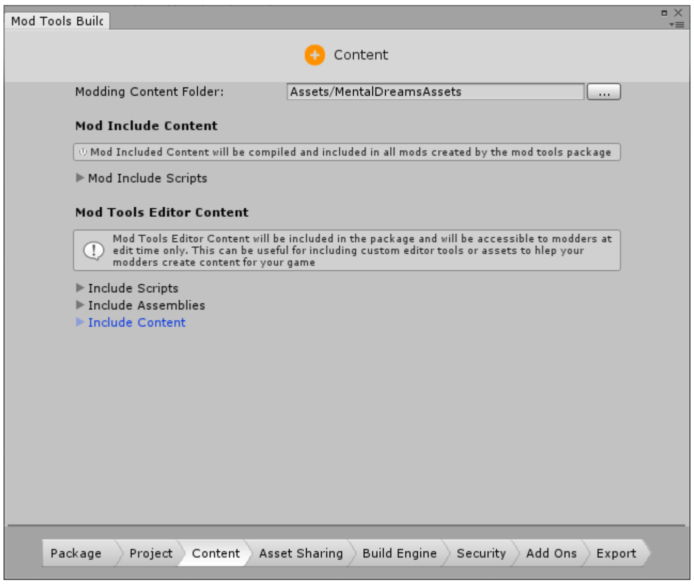
You can manually specify which project assets you include in your mod tools package however we do not recommend that you include input settings assets. If you would like to give your modders access to input then we would recommend that you create an intermediate interface assembly with suitable API's for game input. Take a look at the interface assembly scripting communication approach for information on creating an interface assembly.

Why do we recommend this?

- It forces your modders to conform to your input conventions.
- It allows you to add key remapping support and your modders can still use the same API's
- If you have support for game pads or other devices then you can handle that behind the scenes and not rely on modders implementing device support.

Content

The content page allows you to specify additional asset, script and assembly assets that you would like to include in your mod tools package. It may be desirable to include custom editor tools and scripts that will help your modders to create mods for your game and this page allows you to do that. You can also include additional scripts that will be compiled on a per mod basis allowing all mods to have access to certain types without requiring an interface assembly.



Modding Content Folder: Assets/MentalDreamsAssets ...

Mod Include Content

Mod Included Content will be compiled and included in all mods created by the mod tools package

▶ Mod Include Scripts

Mod Tools Editor Content

Mod Tools Editor Content will be included in the package and will be accessible to modders at edit time only. This can be useful for including custom editor tools or assets to hlep your modders create content for your game

▶ Include Scripts
▶ Include Assemblies
▶ Include Content

Figure 7

Modding Content Folder: Before you can add any content to your mod tools package, you will need to setup a modding content folder. A modding content folder is simply a folder managed by the developer typically located directly under the 'Assets' folder (for example 'Assets/Modding') where all content that you want to include in the mod tools package is located. This folder can also contain general game assets which will not be included and the content can exist at any folder depth. Once you have created this folder you can then select it from the content page of the build tools exporter wizard and from there you will be able to select additional content to add to the mod tools package.

Mod Include Content: The mod include content section allows you to add content that is built into each mod generated by the resulting mod tools package. At the moment only script assets are supported. To add a script simply expand the mod include scripts foldout and select the include assets via the asset hierarchy display.

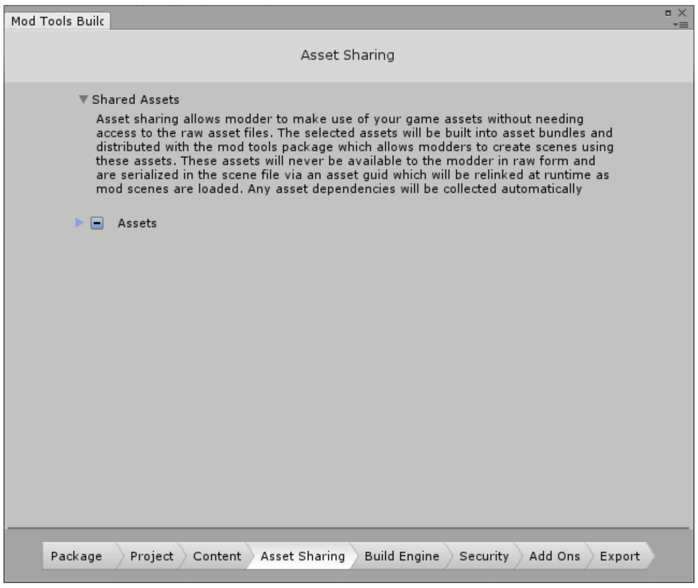
Mod Tools Editor Content: Content added under this section will be available at edit time only and cannot be used directly by mods created with the tools package. This content section is intended to allow you to add additional modding abilities or tools to help your modders create content for your game. A good candidate for mod tools editor content would be race track path creating tools to work seamlessly with your game. This could be in the form of a node-based path editor used to define the layout of a track for in game AI etc. This content could be included as scripts, assemblies or any other unity asset type added via the appropriate foldout category.

Asset Sharing

The asset sharing page can be used to add additional game assets to the mod tools package for modders to use in scenes. This is ideal if you want to allow your modders to create additional game levels via mods.

It is important to note that asset sharing assets will be built into asset bundles before being included in the mod tools package so that the modder does not have access to the raw asset files. This allows you to use 3rd party assets from sources such as the Unity asset store and allow modders to use these assets without providing them in raw form which would breach licence agreements. Since an asset bundle is considered an acceptable game distribution format, there will be no issues in distributing these assets.

Once a shared asset is built into the mod tools package, modders will be able to create scene instances of these assets via the UMod Game Assets window. Adding a shared asset to a mod scene will cause the necessary asset bundle to be loaded and a visual representation of the asset will appear in the scene. uMod will then serialize a guid value in the scene file for this asset which will cause the asset to be replaced with the actual game asset once the mod is loaded.



Asset sharing allows modder to make use of your game assets without needing access to the raw asset files. The selected assets will be built into asset bundles and distributed with the mod tools package which allows modders to create scenes using these assets. These assets will never be available to the modder in raw form and are serialized in the scene file via an asset guid which will be relinked at runtime as mod scenes are loaded. Any asset dependencies will be collected automatically

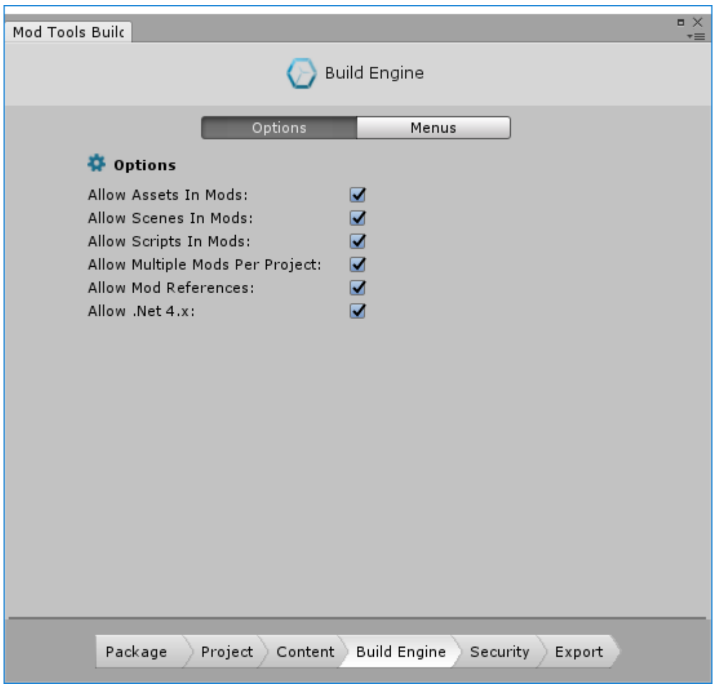
Figure 8

You can add a shared asset via the asset hierarchy display. Currently only prefab assets are supported for asset sharing.

As of version 2.7, you can now reference shared game assets from mod scripts using the 'ModGameAssetReference' type. You can simple declare a serializable field of type 'UMod.ModGameAssetReference' and then you will be able to select the desired game asset via the inspector using the game assets selection window.

Build Engine

The build engine page allows you to configure some of the uMod build engine options. Some of these options may determine the behaviour of the default uMod exporter windows.



Options

Allow Assets In Mods:	✓
Allow Scenes In Mods:	✓
Allow Scripts In Mods:	✓
Allow Multiple Mods Per Project:	✓
Allow Mod References:	✓
Allow .Net 4.x:	✓

Figure 9

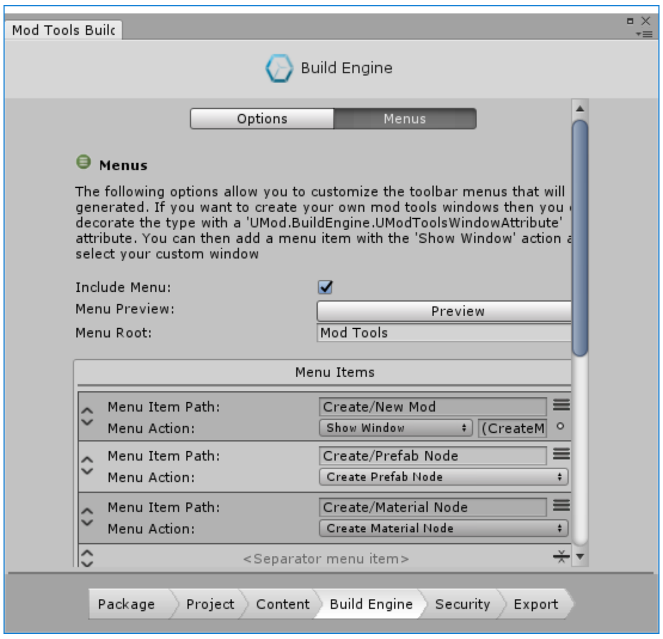
Options

- **Allow Assets In Mods:** Should the build engine allow modders to include shared assets in their mods. Shared assets are defined as any Unity asset that is not a scene, script or managed assembly. If modders attempt to include shared assets in a mod and this option is disable those assets will simply be excluded by the build engine.
- **Allow Scenes In mods:** Should the build engine allow modders to include scene assets in their mods. If modders attempt to include scene assets in a mod and this option is disable those assets will simply be excluded by the build engine.
- **Allow Scripts In Mods:** Should the build engine allow modders to include script content in their mods. Note that script content in this context includes both C# script source and pre-compiled managed assemblies. If you only want to restrict a certain type of script asset then take a look at the security settings page. If modders attempt to include script assets in a mod and this option is disable those assets will simply be excluded by the build engine.
- **Allow Multiple Mods Per Project:** This option is mainly for editor workflow. Multiple mods per project means that your modders will be able to create multiple mod export profiles all in the same Unity project and then select which mod they would like to export via the exporter window. If this option is disabled then all the default uMod exporter windows will only allow a single export profile.
- **Allow Mod References:** Should the build engine allow mods to reference other mods. Mod referencing is a good way of allowing common mod assets to be referenced by a number of other mods but you can prevent this behaviour by disabling this option.

- **Allow .Net 4.X:** Should the build engine allow scripts to be compiled using .Net 4.x. This option is intended for you to force compatibility with the target game. If you game is limited to legacy .Net for whatever reason then you can disable this option which will cause mod build to fail if the modder selects a .Net 4 API.

Menus

The menus section allows you to create your own menu bar entry with certain pre-defined actions without needing to create a custom script to handle it.



Mod Tools Builc

Build Engine

Options | Menus

Menus

The following options allow you to customize the toolbar menus that will
generated. If you want to create your own mod tools windows then you
decorate the type with a 'UMod.BuildEngine.UModToolsWindowAttribute'
attribute. You can then add a menu item with the 'Show Window' action
select your custom window

Include Menu: ✓
Menu Preview: Preview
Menu Root: Mod Tools

Menu Items

Menu Item Path: Create/New Mod
Menu Action: Show Window (CreateM

Menu Item Path: Create/Prefab Node
Menu Action: Create Prefab Node

Menu Item Path: Create/Material Node
Menu Action: Create Material Node

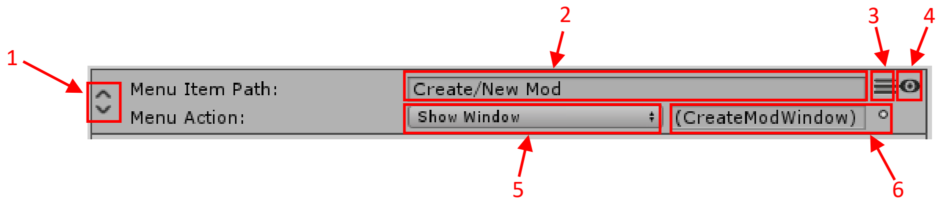
<Separator menu item>

Package | Project | Content | Build Engine | Security | Export

Figure 10

- **Include menu:** Should the mod tools package include a menu. This is highly recommended unless you are using your own editor windows or scripts to run the uMod build engine otherwise you will have no way of starting a mod build.
- **Menu Preview:** Click this button to show a popup context menu of the menu you have created.
- **Menu Root:** This is the label that will appear on the Unity menu bar and that all menu items will be listed under.

The menu items list allows you to create, remove and re-arrange menu entries that will appear in the menu bar as well as set the action which they will perform when clicked. The following image labels all the configurable menu item options:



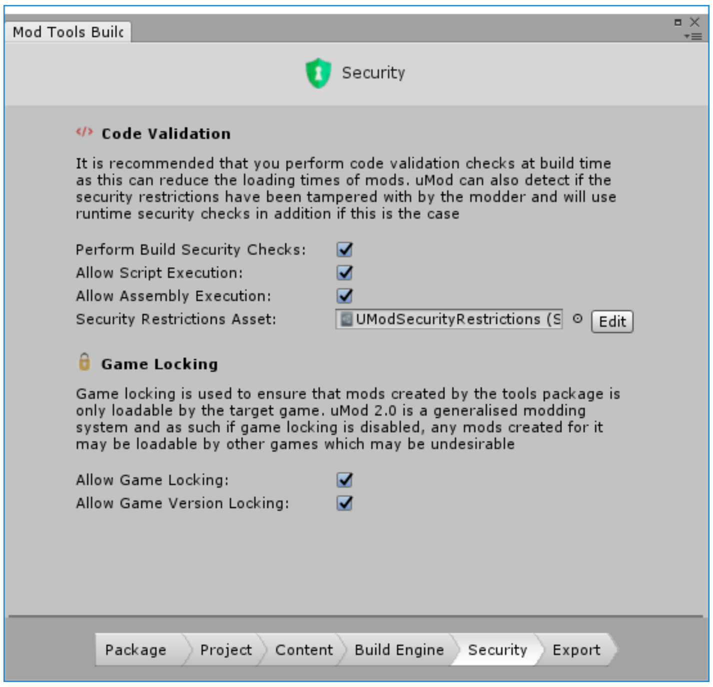
Menu Item Path: Create/New Mod
Menu Action: Show Window (CreateModWindow)

Figure 11

1. **Order Menu Item:** Use the up and down arrows to move the menu item higher or lower in the menu items listbox. This will cause the item to be displayed in a different order. Use the preview button to check that the menu item is shown correctly.
2. **Menu Item Path:** The name or path of the menu item which will appear in the menu bar under the main 'Menu Root' entry. Forward slashes can be used to create sub menus of any depth. You are also able to create menu shortcuts for a menu item using special characters to denote keyboard combinations. Take a look at the Unity documentation for 'UnityEditor.MenuItem' to see the available hotkey formats.
3. **Separator Toggle:** Click this icon to toggle between menu item and menu separator mode. By switching to separator mode the menu item will be displayed as a small separator bar in the preview menu so you can better arrange your menu items into groups.
4. **Visible Toggle:** Click this icon to toggle between visible and non-visible. A non-visible menu item will not show up in the preview menu or the finished menu but will remain in the menu item list so that you can quickly re-enable it if needed.
5. **Menu Action:** A drop down which determines what happened when the menu item is clicked. There are a number of useful present actions including build a mod and show an editor window. Note that clicking a menu item in the preview menu will have no effect.
6. **Menu Window Selector:** This field is only visible when 'Menu Action' is set to 'Show Window'. This field indicates the editor window that will be shown when the menu item is clicked. You can select an editor window to show by clicking the thumb button to the right of the field which will bring up a window select context menu. You can create your own editor windows that can be used by this menu. Take a look at the Custom tools window section for more information.

Security

The security page allows you to configure your code security policy for runtime and build time code validation as well as game locking.


Mod Tools Build

Security

Code Validation

It is recommended that you perform code validation checks at build time as this can reduce the loading times of mods. uMod can also detect if the security restrictions have been tampered with by the modder and will use runtime security checks in addition if this is the case

Perform Build Security Checks:	✓
Allow Script Execution:	✓
Allow Assembly Execution:	✓
Security Restrictions Asset:	UModSecurityRestrictions (S

Game Locking

Game locking is used to ensure that mods created by the tools package is only loadable by the target game. uMod 2.0 is a generalised modding system and as such if game locking is disabled, any mods created for it may be loadable by other games which may be undesirable

Allow Game Locking:	✓
Allow Game Version Locking:	✓

Figure 12

Code Validation

- **Perform Build Security Checks:** Should the build engine run code validation checks using the assigned uMod security restrictions at build time. This is highly recommended as it shifts the expensive code validation steps to edit time as opposed to runtime and can aid in faster mod loading times. It is possible that modders may tamper with the security checks put in place at edit time however uMod is able to detect when code was validated with a compromised security system and in that case it will use runtime code validation checks to ensure safety.
- **Allow Script Execution:** Should the build engine allow C# scripts to be included in mods. These scripts will be compiled at edit time into a managed assembly which will then be packaged with the mod. These scripts will be subject to code validation if enabled.
- **Allow Assembly Execution:** Should the build engine allow pre-compiled managed assemblies to be included in mods. These assemblies will be subject to code validation if enabled.
- **Security Restrictions Asset:** The security restrictions asset that describes that code restrictions that you want to enforce. These restrictions will be shared by your game project and the exported mod tools package so that both edit time and runtime code validation can be performed if necessary based on a number of settings. You can click the 'Edit' button next to the field in order to open a new window where you will be able to create and remove code restrictions.

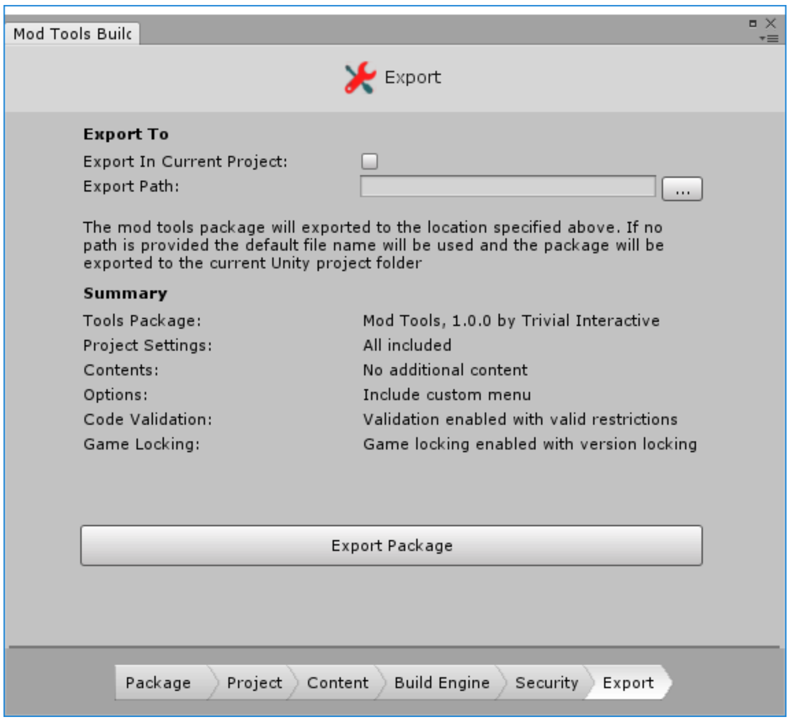
Game Locking

Game locking is a technique used to ensure that the mods created by your tools package will only be loadable by your game. Without game locking anyone else that uses uMod will be able to load your mod.

- **Allow Game Locking:** Should the mod tools package only create mods that are hard locked to your game. If enabled this will cause the mod tools builder to generate a mod game lock asset in your project which is essential for loading mods created by your tools package. It acts as a key which is able to unlock your mods so you should not delete it.
- **Allow Game Version Locking:** Should the mod tools package only create mods that are hard locked to this specific version of your game. This is intended to prevent compatibility errors that may be caused via mismatched game and mod version.

Export

The export page is the final stage in the mod tools builder wizard and allows you to confirm a summary of all previous settings as well as specify the location where the mod tools package will be exported to.



Export To

Export In Current Project: ☐
Export Path: [ ] ...

The mod tools package will exported to the location specified above. If no path is provided the default file name will be used and the package will be exported to the current Unity project folder

Summary

Tools Package:	Mod Tools, 1.0.0 by Trivial Interactive
Project Settings:	All included
Contents:	No additional content
Options:	Include custom menu
Code Validation:	Validation enabled with valid restrictions
Game Locking:	Game locking enabled with version locking

Figure 13

- **Export in Current Project:** You should enable this option when you want your current game project to be able to build mods. The tools builder will generate the necessary content in the current project under a root folder named after your tools package name. You could also export the package to file and then re-import that package however this method exports almost instantly while exporting to file takes a lot longer.
- **Export Path:** The output file path where the mod tools package will be created. The target file should have a '.unitypackage' extension in order for Unity to recognise it.

When you are happy with the settings simply click he 'Export Package' button to begin building the mod tools package. This may take a little while depending upon the content included but a progress bar will be displayed so you can monitor the progress.

Custom Tools Windows

The mod tools package builder allows you to change many settings in order to tailor the package to suit your game however you will still have the default uMod exporter windows to choose from when it comes to setting up a menu. These included windows are intended to be generic and support all configurations that the mod tools builder can generate however you may want to create your own editor window set so that you can further customize your mod tools.

In order to create an editor window that uMod can use you will simply need to create an editor window as you would normally using the Unity API. This window can be implemented as either an editor script or as part of a managed editor assembly but it must exist inside an editor folder. For information on creating an editor window using the Unity API take a look at the Unity editor documentation. We have also included the uMod exporter source code as part of the uMod project so that you can modify it to suit your needs.

Once you have created your editor window you will then need to link it with your tools package. To do this you will need to make the window into a uMod tools window which can be achieved by simply adding the '[`UModToolsWindow`]' attribute to the editor window class declaration. You can then open the mod tools builder wizard and navigate to the 'Build Engine -> Menus' tab where you should now be able to select your editor window in one of the menu item 'Show Window' actions.

C# Code

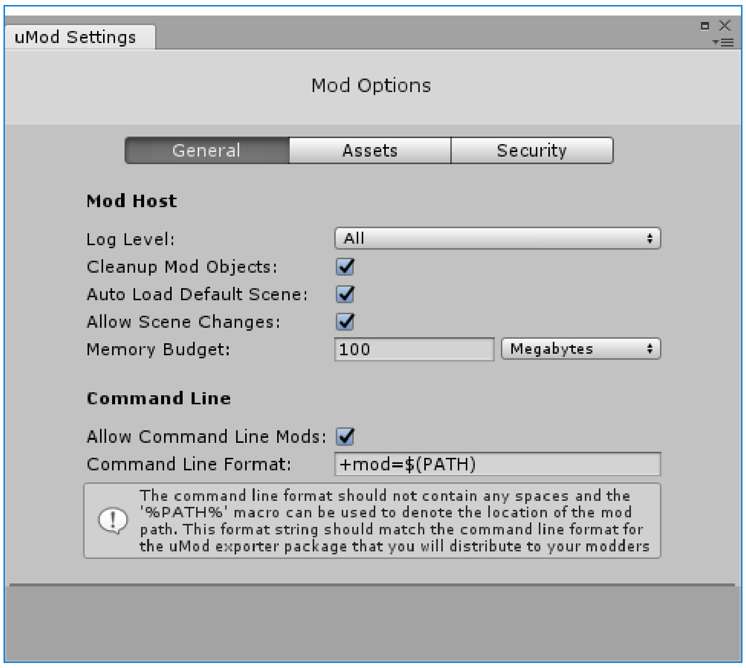
```
using UMod;
using UnityEditor;

[UModToolsWindow]
public class ExampleModToolsWindow : EditorWindow
{
    public void OnGUI()
    {
        GUILayout.Label("Hello World");
    }
}
```

Window Considerations

- If you decide to implement your editor window as a C# script instead of an assembly then you will need to manually include the necessary scripts in your mod tools package. You can do this via the mod tools builder 'Content' page where you can specify a number of C# scripts to be included. If you are using an editor assembly to contain your editor windows then the assembly will automatically be included if one of its types is referenced in the wizard window.
- You can include any number of editor windows and tools to help your modders create content for your game. It does not have to be limited to the default exporter windows. For example: for a racing game it may be desirable to include a number of track creation editor tools to help your modders setup racing tracks.
- You can interact with the uMod build engine from your custom editor windows so that you can create mods without using the default exporter window. You will mainly want to use the 'UMod.BuildEngine.ModToolsUtil' class to access the same actions as the default exporter. Take a look at the scripting reference for API details or take a look at the included uMod exporter source code to see how the default exporter works.

Mod Settings

uMod 2.0 includes a number of global settings that are applied to all mod hosts and can be edited from the Unity editor for convenience. You can edit these settings by navigating to the following menu: 'Tools -> uMod 2.0 -> Settings', after which you should see a window that looks similar to the following:



Mod Host

Log Level: All
Cleanup Mod Objects: ✓
Auto Load Default Scene: ✓
Allow Scene Changes: ✓
Memory Budget: 100 Megabytes

Command Line

Allow Command Line Mods: ✓
Command Line Format: +mod=$(PATH)

The command line format should not contain any spaces and the '%PATH%' macro can be used to denote the location of the mod path. This format string should match the command line format for the uMod exporter package that you will distribute to your modders

Figure 14

Note: *Any settings that are modified in this window are automatically saved when the window is closed.*

You will note that there are a number of tabs in the window to keep the settings organised. This following section will cover the settings located within each tab and what they do.

General

The general tab contains the most used settings which apply to all mod hosts.

Mod Host

- **Log Level:** This value determines how much detailed will be included when loading mods. The default value is set to 'Warnings' meaning that only log messages that are warnings or more severe will be logged.
- **Cleanup Mod Objects:** When enabled, mod hosts will take responsibility for all objects they create so that when a host is unloaded or destroyed, all of these objects will also be destroyed. You should be sure to manage references to mod assets appropriately since unloading mods can cause those existing references to become null.
- **Auto Load Default Scene:** When enabled, mod hosts will automatically attempt to load a mods default scene. If there is no default scene then the mod host will do nothing.

- **Allow Scene Changes:** When enabled, mods will be able to request scene changes which will be handled by the mod host. If you want to prevent modders from switching scenes then you should either disable this value or implement you own scene load handler.
- **Memory Budget:** This value can be used to specify the maximum amount of mod resource data that can be loaded into memory by each mod host. For smaller mods it may offer load time benefits to load the entire contents into memory but for larger mods this will not be practical. If a mods resources size is greater than this value then the content will be streamed from file which may require decompression to occur. If the mod is loaded on a remote server then it will first be downloaded to a temporary location and then streamed from file. Note that this limit does not account for mod host overhead.

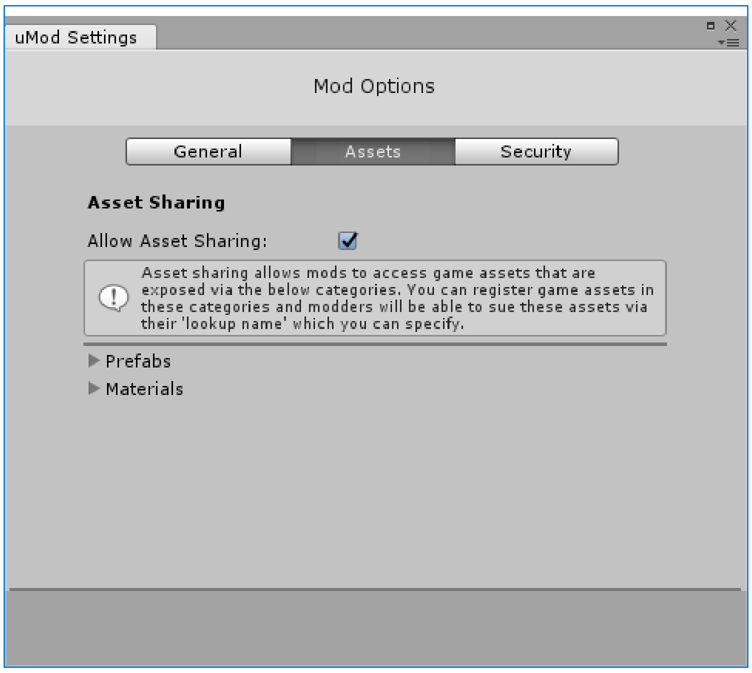
Command line

- **Allow Command Line:** When enabled, uMod will automatically parse the application command line on startup and will identify any mod paths specified using the 'Command Line Token' settings as a search string. Supporting command line launching is highly recommended as the accompanying exporter tool has built in Build and Run functionality that uses the command line to launch a specific mod.
- **Command Line Format:** This value is only used when the 'Allow Command Line' setting is enabled and represents a formatted string that specifies the argument format when launching the game from the command line. The '$(PATH)' constant represents a macro variable and must be present somewhere in the string as it represents the position at which the actual mod path will be injected. It is recommended that this value remains at the default for maximum compatibility, however if you do decide to change then you will need to ensure that your mod tools package and your uMod settings have the same command line format.

Asset

As of version 2.5 the assets tab has been removed in favour of asset sharing via the tools builder window.

The asset settings tab allows you to setup asset sharing where mods can access certain game assets such as prefabs or materials that you decide to share.



Asset Sharing

Allow Asset Sharing: ✓

Asset sharing allows mods to access game assets that are exposed via the below categories. You can register game assets in these categories and modders will be able to sue these assets via their 'lookup name' which you can specify.

▶ Prefabs

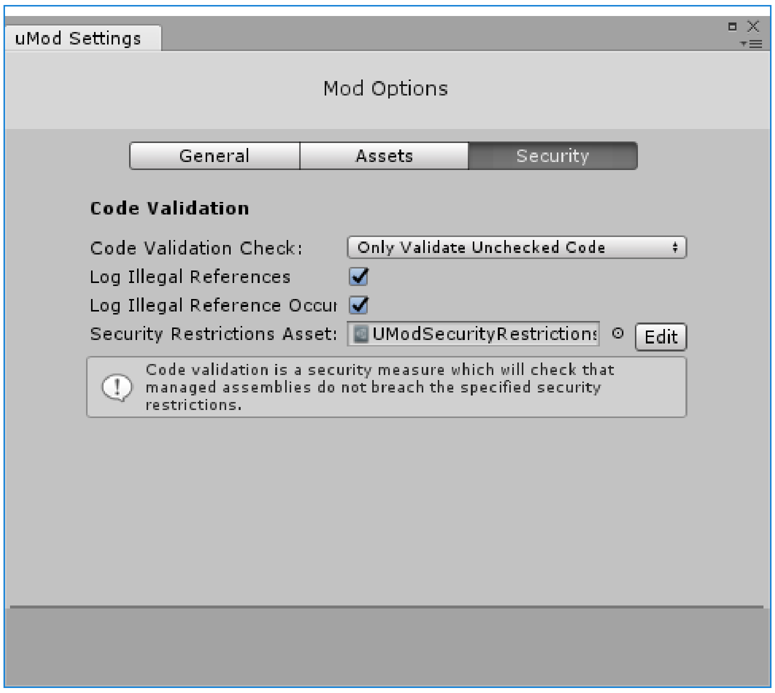
▶ Materials

Figure 15

- **Allow Asset Sharing:** When enabled, game prefabs may be accessed by modders for use in scenes or scripts. A good example would be allowing the modder to place the player prefab into a modded scene. Don't worry though, as the developer you get to decide which prefabs are allowed to be accessed and which prefabs cannot.
- **Prefabs:** You can drag prefab assets to the drag and drop area under the prefabs foldout in order to register the prefab for asset sharing. This will allow modders to access the object by name using a prefab node.
- **Materials:** This is much the same as prefabs only this allows for certain game materials to be shared. Adding materials to this list will provide access by modders via a material node.

Security Settings

The security tab allows you to setup a code security policy for any modded code that may be included with mods. Using 3rd party code that you have no control over could potentially be a security risk as modders may have malicious intent and could do almost anything using the C# language. Using code validation is highly recommended to significantly reduce the risk that unknown code may have by only allowing code which passes a number of validation checks to be executed.



Code Validation

Code Validation Check: Only Validate Unchecked Code
Log Illegal References ✓
Log Illegal Reference Occur ✓
Security Restrictions Asset: UModSecurityRestrictions

Code validation is a security measure which will check that managed assemblies do not breach the specified security restrictions.

Figure 16

- **Code Validation Check:** This value determines what type of code validation checks uMod will perform on any code included with a mod. Options are:
    - **No Code Validation:** uMod will not perform any runtime code validation checks. If you enabled 'Perform Build Security Checks' when building your tools package then the build engine will perform code validation when building mods.
    - **Only Validate Unchecked Code:** This option means that modded code will be validated at build time if possible by the uMod build engine and will then be stamped with a security hash. If the security hash is missing or invalid then uMod will also run the code validation checks at runtime. Shifting the code validation checks to build time will help to reduce the overall load time of a mod containing code assets.
    - **Always Validate Code:** This option will force uMod to always run code validation when loading mods. This may decrease loading performance of mods as code security checks can take some time depending upon the amount of code included.
- **Log Illegal References:** When enabled uMod will log each code validation error to the console so you can see what restrictions the mod is breaking. This is useful for debugging purposes.
- **Log Illegal Reference Occurrences:** When enabled uMod will log every occurrence of illegal assembly, namespace or type access to the console so you can see exactly which code is

failing validation. Note that uMod performs code validation at the CIL level so source code information such as line and column numbers will not be available.

- **Security Restrictions Asset:** The security restrictions asset that describes that code restrictions that you want to enforce. These restrictions will be shared by your game project and the exported mod tools package so that both edit time and runtime code validation can be performed if necessary based on a number of settings. You can click the 'Edit' button next to the field in order to open a new window where you will be able to create and remove code restrictions.

Moddable

Take a look at the 'Moddable Content' section for more information.

Mod Loading

This next section will cover the loading and unloading of mods in detail.

Load Mod

Loading a mod in uMod 2.0 is relatively straight forward and can be achieved using a single method call, we do however need to know the location of the mod before we can load it. uMod is able to load mods from both local and remote locations and will determine the location of the path during loading.

Note: *If mods are located on a remote server then the loading time can drastically increase since the mod must first be downloaded. The actual time will depend on a few factors such as the size of the mod and the download speed but you are able to retrieve download statistics and progress while loading.*

The following code shows how a mod can be loaded using a local mod path.

C# Code

```csharp
using System;
using UnityEngine;
using UMod;

public class ModLoadExample : MonoBehaviour
{
    private void Start()
    {
        // Create a uri object from a string path
        Uri path = new Uri("file:///C:/Mods/TestMod.umod");

        // Issue a mod load request
        Mod.Load(path);
    }
```

Note: *An important thing to note when specifying a mod path is that unlike previous uMod versions, you now need to specify the path to the mod file including extension.*

It is very easy to issue a mod load request as you have seen but how do we actually know that the mod loaded successfully? It is good practice to implement some error checking code and to not assume that the mod will always load correctly without errors. uMod aims to be as simple and intuitive to use and as a result you can check whether a load failed by accessing a single property. Of course, you can also get much more information about the loading error if needed to help with debugging or error reporting.

First off, you may have already noticed that the return value of the load method is a mod host which is essentially a manager object that is responsible for managing the loaded mod. For detailed information take a look at the Mod Host section. The host is a state object and has a number of variables that represent the current state of the host, such as: 'IsLoading' and 'IsLoaded'. These values can be accessed to determine what state the host is in, for example: 'IsLoaded' can be used to determine whether the mod host has a valid mod loaded.

The following code shows the above example that has been modified to log a message when the host has finished processing a load request. The message changes depending upon the load status of the mod. For more information on the variables that are available take a look at the dedicated scripting reference as well as the example scripts included in the package.

C# Code

```csharp
using System;
using UnityEngine;
using UMod;

public class ModLoadExample : MonoBehaviour
{
    private void Start()
    {
        // Create a mod path from a string path
        Uri path = new Uri("file:///C:/Mods/TestMod.umod");

        // Issue a mod load request
        ModHost host = Mod.Load(path);

        if(host.IsLoaded == true)
        {
            Debug.Log("The mod is loaded");
        }
        else
        {
            Debug.Log("The mod could not be loaded - " + 
host.LoadResult.Message);
        }
    }
```

You are now able to issue mod load requests however you may observe that calling a loading method that can take an undefined amount of time to complete and is not the best practice in a responsive application like a game. It is often good practice to display a loading screen with appropriate progress values that inform the user what is happening and how long they can expect to wait. Take a look at the next section which shows how to implement asynchronous mod loading which takes place on a background thread.

Async Mod Loading

In order to make uMod easier to use within Unity, we tried to closely mimic the async loading method that Unity uses. As a result, uMod includes a number of custom yieldable objects that are returned by all async load methods which contain information such as the current load progress as well as the end result. As you would expect, all of the objects are yieldable and can be used I coroutines.

The following C# code shows a basic example where a mod is loaded asynchronously. Note that the return value of the load method if now a ‘ModAsyncOperation’.

C# Code

```csharp
using System;
using UnityEngine;
using UMod;

public class ModCoroutineExample : MonoBehaviour
{
    private IEnumerator Start()
    {
        // Create a mod path
        Uri path = new Uri("file:///C:/Mods/ExampleMod.umod");

        // Create a request
        ModAsyncOperation<ModHost> request = Mod.LoadAsync(path);

        // Wait for the task to complete
        yield return request;

        // Check the status of the load
        Debug.Log("Loaded = " + request.IsSuccessful);
    }
```

Note: *The actual loading will be done on a background thread for the most part so you are able to run other tasks while the request is being processed. Be aware however that there is a small activation portion of the load which must be performed on the main thread so you may see a slight load spike.*

That is all quite easy and shifts all the heavy loading away from the main thread however we have no way of knowing how much progress the load has made. This would be very useful to know if we wanted to provide the user with a nice loading bar or progress indicator. Luckily we can access the current loading progress of an async load request by modifying the above code slightly:

C# Code

```csharp
using System;
using UnityEngine;
using UMod;

public class ModCoroutineExample : MonoBehaviour
{
    private IEnumerator Start()
    {
        // Create a mod path
        Uri path = new Uri("file:///C:/Mods/ExampleMod.umod");

        // Create a request
        ModAsyncOperation<ModHost> request = Mod.LoadAsync(path);

        while(request.IsDone == false)
        {
            // Display the loading progress
            Debug.Log("Load progress: " + request.Progress);

            // This is very important - it allows the main thread
to continue and return here in the next frame
            yield return null;
        }

        // Check the status of the load
        Debug.Log("Loaded = " + request.IsSuccessful);
    }
}
```

As you can see, the 'ModAsyncOperation' object contains a property call 'Progress' which returns a normalized value between 0 and 1 representing the current load progress where 0 represent no progress and 1 represents a completed load. We can use this value to update a progress bar or we can even display the load value as a percentage using the 'ProgressPercentage' property.

You can use the 'Result' property of a 'ModAsyncOperation' to get the load result which in this case would be a mod host.

For more information take a look at the included example scripts which show these async load operations in detail.

Downloading Mods

Depending upon the path specified when loading a mod, the host may be required to download the mod from a remote server. Loading from a server can be achieved simply by creating a mod path using a URL path as opposed to a local file path, For example:

"trivialinteractive.co.uk/Mods/ExampleMod.umod"

When downloading is involved, the loading time can be significantly increased because uMod must first download all of the mod files from the server before it can begin the loading process. The duration of the download will be influenced by many factors including the size of the mod and the download speed for the network, but in most cases you can expect it to take longer than a few seconds.

In games, it is often common practice to display a loading screen to the user when they have to endure any lengthy process and as such uMod has a number of useful features that you can use to display a loading type screen while any downloads are processed in the background. The following C# example code shows how the progress of a download can be accessed:

C# Code

```
using System;
using UnityEngine;
using UMod;

public class ModProgressExample : MonoBehaviour
{
    private ModHost host = null;

    private void Start()
    {
        // Create a mod path from a url
        Uri path = new
Uri("trivialinteractive.co.uk/Mods/ExampleMod.umod");

        // Issue a mod load request
        host = Mod.loadMod(path);

        // Add an event listener for progress updates
        host.onModLoadProgress += onModLoadProgress;
    }

    private void onModLoadProgress(ModLoadProgressArgs args)
    {
        // This will be true if the host is downloading from a
server
        if(args.IsDownloading == true)
        {
            Debug.Log("Download Progress: " + args.Progress);
            Debug.Log("Download Speed: " + args.Speed);
        }
```

The above code will receive progress events throughout the loading progress which can be used to display a downloading screen. There are two important values in the progress update which can be used to measure the progress of a download:

- Args.Progress: This is a normalized float value that represents the progress of a download. The value will be 0 if no bytes have been downloaded an 1 if all bytes have been downloaded. This value can be used to create a progress bar or similar loading visual.
- Args.Speed: This represents the current speed of the download and can be measured either in bytes, kilobytes or megabytes depending upon the most fitting unit.

Command Line Loading

uMod 2.0 supports command line loading out of the box which can be used by the exporter tool to allow build and run behaviour. Build and run allows the exporter tool to export a mod based on its current settings and then automatically launch the target game with the mod loaded. This is achieved by launching the game process and passing command line arguments containing the path to the newly exported mod.

By default the command line is not parsed automaticallyparsed when ‘Mod.initialize’ is called however any mods paths specified will not be loaded automatically. Instead uMod will trigger the ‘ModCommandLine.onModCommandLine’ event for every mod it finds in the command line. This allows the developer to decide whether to load these mods or not. If you always want to load command line mods automatically then you are able to pass ‘true’ as an argument to ‘Mod.initialize’ which will cause any specified mod paths to be loaded automatically.

In order for uMod to detect when a mod path has been specified on the command line it uses a predefined format string which defines how the input should look. This format string can be modified via the settings window in order to be in keeping with any other command line arguments that your game may receive. By default, the command line format string is:

```
+mod=$(PATH)
```

The ‘$(PATH)’ value in the string is a macro value and will be replaced by the full mod path during parsing. A typical command line string with 2 mods specified may look like the following:

```
C:/mygame.exe +mod=C:/Mods/ExampleMod.umod +mod=C:/Mods/AnotherMod.umod
```

When uMod has finished parsing the command line it will attempt to load a mod from ‘C:/Mods/ExampleMod.umod’.

Unload Mod

Once you have successfully implemented mod loading one of the next things you may want to do is unload mods which is also a trivial task. Mods may be unloaded at any time as you the developer sees fit and will cause the host to discard all mod related data resulting in a clean host instance that may be re-used.

Note: *Due to the limitations of the CLR, any mod scripts that have been loaded into memory may remain after the host has unloaded the mod however, they will be treated as dead scripts and will not receive any events or context data from the host.*

Unloading a mod is as simple as calling a single method as shown below:

C# Code

```
using System;
using UnityEngine;
using UMod;

public class ModUnloadExample : MonoBehaviour
{
    private ModHost host = null;

    private void Start()
    {
        // Create a mod path from a string path
        Uri path = new Uri("file:///C:/Mods/TestMod.umod");

        // Issue a mod load request
        host = Mod.Load(path);

      if(host.IsLoaded == true)
        {
            // Unload the host when the load has successfully
completed
            host.unloadMod();
        }
    }
}
```

Note: *If the host that you unload is a dependency of another mod host then that mod will also be unloaded to prevent the dependency chain from being broken.*

You may already be aware that the mod host object derives from mono behaviour and as a result may be destroyed as you would normally destroy a Unity object. Destroying a host using Object.Destroy will indeed cause the managed mod to be unloaded as you might expect.

Mod References

As of version 2.1.5 uMod supports mod referencing which allows a mod to reference one or more other mods. Adding a reference to a mod makes all the assets scenes and public scripts accessible to the parent mod meaning that you can share common assets scenes and code instead of storing the same content in multiple mods.

All mod references are defined at export time by the modder and cannot be changed without rebuilding the mod. Take a look at the modder documentation for information on adding and managing mod references.

Resolving References

In order for a mod that references other mods to be loaded successfully, all reference dependencies must be resolvable during the loading process. This means that all referenced mods and indirectly referenced mods must be either loaded or loadable. In order to be loadable, a mod must exist in either the ModDirectory or in a location that is resolved by implementing the 'OnResolveReference' event. If the referenced mods are loadable then uMod will automatically load them during the loading process.

The following code shows how to manually handle reference resolve requests:

C# Code

```
using UnityEngine;
using UMod;

public class ModReferenceResolveExample : MonoBehaviour
{
    private void Start()
    {
        // Add an event listener
        Mod.OnResolveReference += OnResolvePath;
    }

    private Uri OnResolvePath(IModNameInfo reference)
    {
        // Create the full path
        string path = Path.Combine("C:/Mods", reference.ModName +
".umod");

        // Create the mod path
        return new Uri(path);
    }
`
```

Note: *The 'OnReferenceResolve' event will only be triggered if the reference cannot be found by uMod. It is not used as a complete replacement resolver.*

Reference Dependencies

When a mod references another mod, uMod will need to ensure that the reference mod is loaded prior to loading the specified mod. This creates a recursive dependency chain where reference mods may also reference other mods which can be represented as a tree structure. When a referenced mod is loaded by uMod it is marked as a dependency meaning that it is required by one or more other mods. Due to this requirement, unloading a mod that is required by other mods will cause all dependants to also be unloaded meaning that a single unload call could potentially unload a number of mods depending upon the dependency structure. This may not be an issue to you as the developer but it is just something to be aware of.

Reference Names

It may be desirable as a developer to get an array of reference names for a specific mod which is easily achievable. A reference name is defined as a name and version for a mod and can be accessed both prior to loading and after loading of a mod. The following C# code example shows how you can use the Mod Directory to get mod reference information for a specific mod:

C# Code

```
using UnityEngine;
using UMod;

public class ModReferenceResolveExample : MonoBehaviour
{
    private void Start()
    {
            // Create a mod directory for specified folder
            ModDirectory directory = new ModDirectory(“C:/Mods”);

            // Find an installed mod
            IModInfo info = directory.GetMod("TestMod");

            // Iterate all references
            foreach(IModNameInfo referenceInfo in info.ReferenceInfo)
            {
                // Print the name and version
                Debug.Log(referenceInfo.ModName + ", " +
referenceInfo.ModVersion);
            }
```

Once a mod has been loaded it is also possible to access an array of mod hosts that are dependent upon the first host. You can then access the reference information directly from the host dependencies as shown In the following example:

C# Code

```csharp
using UnityEngine;
using UMod;

public class ModReferenceResolveExample : MonoBehaviour
{
    private void Start()
    {
            // Iterate all references for the host
            foreach(IModNameInfo referenceInfo in host.ReferencedMods)
            {
                // Print the name and version
                Debug.Log(referenceInfo.ModName + ", " +
referenceInfo.ModVersion);
            }

            // Iterate all host dependencies
            foreach(ModHost referenceHost in host.HostDependencies)
            {
                // Get the mod info
                IModInfo referenceInfo = referenceHost.CurrentMod;

                // Print the name and version
                Debug.Log(referenceInfo.NameInfo.ModName + ", " +
referenceInfo.NameInfo.ModVersion);
            }     }
}
```

Reference Assets

If a loaded mod references one or more other mods then you will be able to access any assets that are included in that referenced mod from the loaded mod. This applies for both the developer and the modder, the only difference being that the developer has the option of explicitly accessing assets for the current mod or referenced mods. This is one of the main benefits of mod referencing and allows common mod assets to be built into a dedicated mod and the referenced by a number of other mods reducing export times and individual mod sizes.

In order to access these referenced mod assets you can use the 'ModSharedAssets' property of a mod hosts which will allow access to assets built into the mod and all referenced mods reccursivley. The following example show the usage:

C# Code

```csharp
using UnityEngine;
using UMod;

public class ModReferenceAssetsExample : MonoBehaviour
{
    private void Start()
    {
        // We assume this host is already valid and loaded
        ModHost host;

        // Load an asset which is located in a referenced mod
        host.ModSharedAssets.Load("Cube");
    }
}
```

The ‘ModSharedAssets’ property returns the same ‘IModAssets’ interface used to load assets only included in the current mod. Take a look at the Mod Assets section for more information about the usage.

Mod Assets

uMod 2.0 allows any Unity asset type to be included within mods and these assets are accessible to the game at runtime. As the developer you are able to load assets from mods in a similar way that you would use the 'Resources' folder in Unity. The following sections will cover the loading of assets and any special requirements.

Load Mod Assets

Depending upon the type of modding you want to support you way need to be able to load a number of assets from mod once it has been successfully loaded. uMod 2.0 allows you to do this in a similar way to the 'Resources' folder in Unity and can return any asset that derives from 'UnityEngine.Object'.

Just like the resources folder, you are able to load assets by using their name. A common method of modding is to allow modders to create assets with special predefined names. Upon loading the mod you search for assets that match these names and you can then replace the in game assets with the modded assets.

Note: *An important thing to note when loading game objects is that they must be 'Instantiated' just like you would when loading from the 'Resources' folder in order from them to appear in the scene. You should take special care with this as there appears to be a bug in Unity that causes the engine to freeze if a gameobject loaded from a mod is modified without instantiating it.*

There are three main methods that you can use to load assets from mods and the method you choose will depend on how many simultaneously loaded mods you want to support. For some games you may only want to support the loading of a single mod at a time which will help to eliminate conflicts and clashes. For other games you may need support for any number of mods running simultaneously in which case you will take a different approach.

- **ModAssets**: 'ModAssets' is a static class which can be used by the developer to load assets from at runtime. Typically you would only chose this option if you want to support more than one loaded mod at a time as it will search all loaded mods for an asset with the specified name.
- '**Assets'**: An alternative method is to use the 'Assets' property of the appropriate mod host in order to access the same loading API as the above method. It is important that you ensure that the host has loaded a mod before accessing this API otherwise an exception will be thrown.
- **SharedAssets**: The final method of accessing mod assets is by using the 'Sharedassets' property of the appropriate mod host. Shared assets contain all assets that are included in the mod as well as all referenced mod assets recursively. That means that any assets in any referenced mod or indirectly referenced mod will be accessible. It is important that you ensure that the host has loaded a mod before accessing this API otherwise an exception will be thrown.

Note: *You are able to make any number of load calls with the same asset name and the asset will be cached on the first call to prevent reloading when it is unnecessary.*

There are three main calls within the assets API which have a number of overloads in order to provide similar behaviour as the 'Resources' class. All of the following method will allow for asset caching meaning that multiple calls to a load method with the same asset name will result in loading taking place on the first call only. The second call will simply return a cached instance.

1. **Load**: All load methods will attempt to load an assets from the mod with the specified name. These calls will cause loading to take place on the main thread any will block until they have completed. The return value will always be of type 'UnityEngine.Object' or a derived type such as 'Texture2D' and you are able to use the generic overload to specify the type of the return value.
2. **LoadAsync**: All LoadAsync methods will have the same functionality of 'load' methods, however the loading will take place on a background thread allowing the main thread to continue execution. As a result, async method return a yieldable ModAsyncOperation object which can be used to access load progress information.
3. **Instantiate**: When loading game objects from the mod, the return value must be instantiated in order for it to appear in the scene. This can be done with a subsequent call to 'Object.Instantiate' or you can use this 'instantiate' method which will load the asset beforehand.

The following C# code shows basic usage of the assets API. For more detailed examples take a look at the example scripts included with the package. The following code assumes that a mod has already been loaded in order to keep the code concise.

C# Code

```
using UnityEngine;
using UMod;

public class UModAssetExample : MonoBehaviour
{
    private ModHost host = null;

    private void Start()
    {
        // This example assumes that the 'host' already has a valid
mod loaded.

        if(host.Assets.Exists("MyAsset") == true)
        {
            // Load the asset from the mod
            GameObject go = host.Assets.Load("MyAsset") as
GameObject;

            // Create an instance in the scene
            Instantiate(go, Vector3.zero, Quaternion.identity);
        }
    }
}
```

Async Asset Loading

uMod 2.0 includes a custom yield instruction for loading assets from mods called 'WaitForAssetLoad'. The loading call will be issued immediately after creating an instance of the instruction so you should not cache references to the type but instead create a new instance for every load request. The following C# code shows a basic example where the coroutine will yield until the asset has been loaded.

C# Code

```csharp
using UnityEngine;
using UMod;

public class ModAsyncAssetExample : MonoBehaviour
{
    private ModHost host = null;

    private IEnumerator Start()
    {
        // This example assumes that the 'host' already has a valid
mod loaded.

        // Create a request
        ModAsyncOperation request =
host.Assets.LoadAsync("MyAsset");

        // Wait for the task to complete
        yield return request;

        // Get the result as a game object
        GameObject go = request.Result as GameObject;

        // Create an instance in the scene
        Instantiate(go, Vector3.zero, Quaternion.identity);
    }
}
```

Note: *The actual loading will be done on a background thread for the most part so you are able to run other tasks while the request is being processed.*

For more information take a look at the included example scripts which shows async asset loading in detail.

Unloading Assets

Once assets have been loaded by a mod host they are considered to be owned by that host and cannot be unloaded until the host is unloaded. By calling 'unloadMod' on a mods host, all of the mods loaded assets will also be unloaded. This is only true for shared asset references such as textures or prefabs, however any instantiated assets can be destroyed as you would expect in Unity by using the 'Object.Destroy' method.

Shared Assets

uMod 2.0 allows for a concept called asset sharing where assets included in the game can be shared with loaded mods which allows modders to access game assets. Typically the modder will use special game objects called prefab nodes to mark the position and orientation in the scene where a game asset should be placed and at runtime this reference will be resolved based on the assets name. A good example of this is to mark the start position of a player character in a modded scene.

As the developer you will always have full control over what can and cannot be accessed by modders and as a result you must explicitly select the assets that are allowed to be shared with mods. Adding or removing shared assets can be done via the 'Mod Assets' window which can be opened from the settings window. On the assets tab there is a button called 'Edit Assets' and once you click this you should see the following window appear:

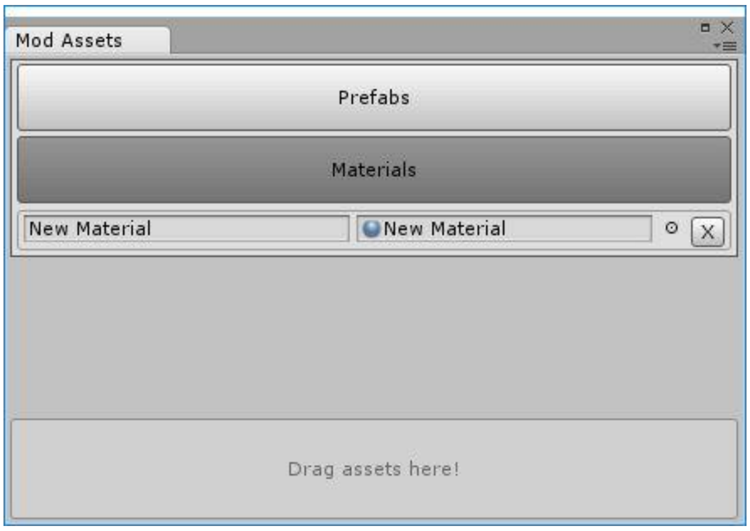


Figure 17

Note: *At the moment, only material and prefab assets are allowed to be shared but this may expand at a later date.*

The asset collection window allows you to drag assets that you want to share into the collection where they will be categorised by type and assigned a default lookup name. As you can see, once an asset has been assigned, the lookup name is automatically assigned as the asset name but can be changed to anything you like. The lookup name is a case sensitive string that modders will use to reference the asset. It is important that the name is not overly complex to the point where spelling mistakes could be easy to make, and that the name is unique in the appropriate category (Materials and Prefabs may share the same name as they are in different categories). If multiple assets in the same category share the same name then uMod will simply use the first asset added to the collection whenever the modder references it.

Moddable Content

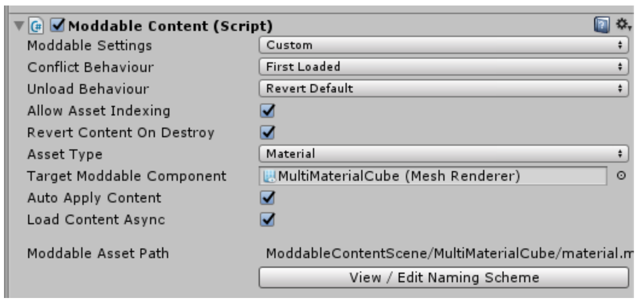
uMod version 2.6 adds the moddable content system which allows you to quickly and easily setup an in-game object as moddable. Making an object moddable means that the object can now have one or more of its components modified so that source assets are now referenced from mods instead of game assets. This means that modders will be able to create new asset content such as materials, textures, meshes and more and have this content be applied to a specified game object when the mod is loaded.

How does it work?

In order to make a scene object or a prefab asset a moddable entity, you will need to add the 'ModdableContent' component to the game object. This component will cause a unique asset path/name to be generated in the UMod settings which will be the final location where modders should create a new asset in order to mod this particular component. At runtime the moddable content component will detect when a new mod is loaded and automatically search for any matching mod content at the location defined in the naming scheme. The content will be loaded if found and applied to the component which is targeted by the moddable content.

Moddable Content Component

The moddable content component is the heart of the automatic content modding system and is responsible for loading, applying and managing mod content for a specific game object component.



Moddable Content (Script)

Moddable Settings	Custom
Conflict Behaviour	First Loaded
Unload Behaviour	Revert Default
Allow Asset Indexing	✓
Revert Content On Destroy	✓
Asset Type	Material
Target Moddable Component	MultiMaterialCube (Mesh Renderer)
Auto Apply Content	✓
Load Content Async	✓
Moddable Asset Path	ModdableContentScene/MultiMaterialCube/material.m

View / Edit Naming Scheme

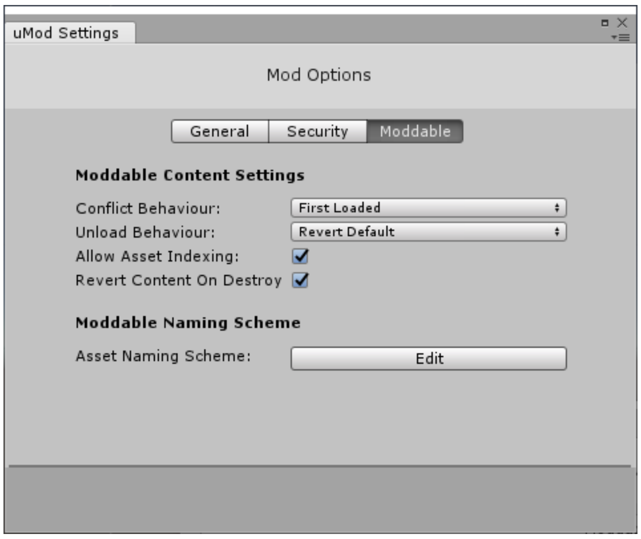
Figure 18

- **Moddable Settings:** Choose whether common settings for the moddable content component are inherited from the global moddbale settings or whether they are defined on this component.
    - **Use Settings:** Common moddable settings values should be taken from the global moddable settings. See 'Moddable Settings' section for more information.
    - **Custom:** Common moddable settings values should be taken from this component.
- **Conflict Behaviour (Moddable Settings = Custom):** What should happen when one or more mods with identical content are loaded into the game.
    - **First Loaded:** The mod that was loaded first should take modding priority and have its content used to replace game assets. Note that when loading mods asynchronously they may not load in the order requested depending upon the load time for the mod.
    - **Last Loaded:** The mod that was loaded last should take modding priority and have its content used to replace game assets. Note that when loading mods

    asynchronously they may not load in the order requested depending upon the load time for the mod.
  - **Revert Default:** The moddable content should not apply any mod content or should revert back to the original game asset when a conflict is detected.
- **Unload Behaviour (Moddable Settings = Custom):** What should happen when the owning mod of any applied moddable content will be unloaded from the game.
  - **Revert Default:** Any applied moddable content should be reverted to use the original game asset.
  - **RevertConflictingModOrDefault:** Any applied moddable content should be replaced by content from any previously conflicting mods or reverted to the default game asset if no other conflicting mods are loaded.
  - **Do Nothing:** Take no action. Note that this may cause some asset references to be broken causing potential game issues.
- **Allow Asset Indexing (Moddable Settings = Custom):** Can moddable assets include array indexing syntax at the end of their file name indicating which array element it should be applied to. For example: 'myMaterialAsset[1].mat' would cause the asset to be applied at array index '1' of the target component. This is useful for components that accept more than one asset of a particular type such as renderer components.
- **Revert Content On Destroy (Moddable Settings = Custom):** Should the moddable content be reverted to the default game asset when the moddable content component has been removed from a game object. Note that once a moddable content component has been removed from a game object, conflict and unload handling can no longer take place.
- **Asset Type:** The type of asset that can be modified on the target component. Note that the asset type and the target component must be compatible.
- **Target Moddable Component:** The game object component that can be modified. If you want to allow an objects material or texture to be modified then you could assign a Renderer component to this field. Note that the target component must be compatible with the asset type.
- **Auto Apply Content:** Should moddable content be applied automatically when the game starts or a new mod is loaded. Disabling this option will mean that you will need to manually call 'ApplyModdableContent()' in order to load and apply any suitable moddable content.
- **Load Content Async:** Should moddable content be loaded asynchronously in the background or should the main thread wait for the content to load.
- **Moddable Asset Path:** The full asset path where modders can create new mod content to replace the assets of the target component. This path is relative to the mod asset folder path and folder structures are denoted by '/' forward slash. This value is auto generated based on game object context at edit time but can be modified via the moddable asset naming scheme accessible via the UMod settings window. Alternatively, you can click the 'View / Edit Naming Scheme' to directly open the moddable asset naming scheme window.

Moddable Settings

The global moddable settings can be edited via the uMod settings window under the 'Moddable' tab. Go to 'Tools -> uMod 2.0 -> Settings' to open the settings window and then switch to the 'Moddable' tab in order to edit these options.


uMod Settings

Mod Options

General | Security | Moddable

Moddable Content Settings

Conflict Behaviour: First Loaded
Unload Behaviour: Revert Default
Allow Asset Indexing: ✓
Revert Content On Destroy ✓

Moddable Naming Scheme

Asset Naming Scheme: Edit

Figure 19

The moddable content settings displayed in this window will be applied to all moddable content components which have the 'Moddabe Settings' option set to 'Use Settings'. This allows you to quickly and easily change important moddable settings for all moddable content in the game project.

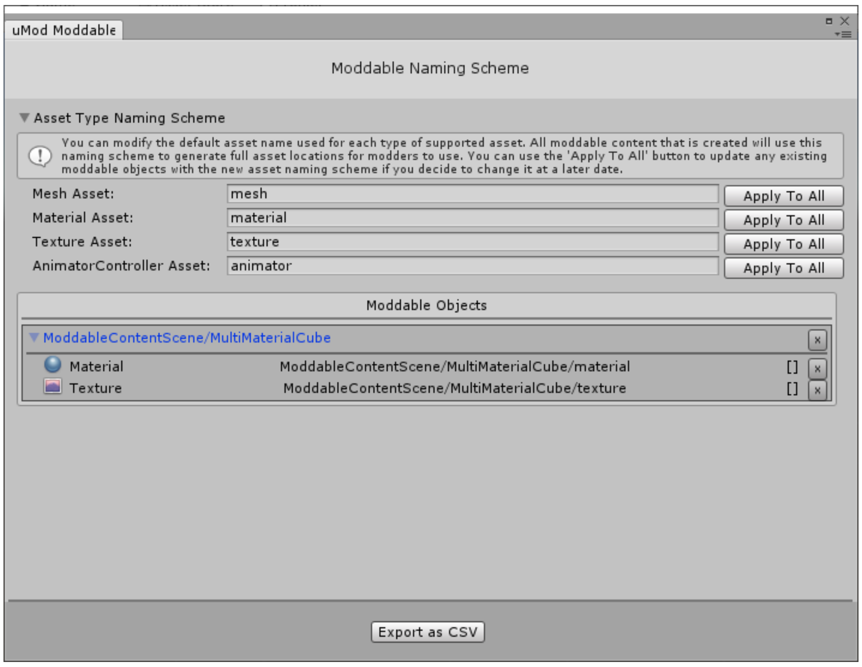
- **Conflict Behaviour:** What should happen when one or more mods with identical content are loaded into the game.
    - **First Loaded:** The mod that was loaded first should take modding priority and have its content used to replace game assets. Note that when loading mods asynchronously they may not load in the order requested depending upon the load time for the mod.
    - **Last Loaded:** The mod that was loaded last should take modding priority and have its content used to replace game assets. Note that when loading mods asynchronously they may not load in the order requested depending upon the load time for the mod.
    - **Revert Default:** The moddable content should not apply any mod content or should revert back to the original game asset when a conflict is detected.
- **Unload Behaviour:** What should happen when the owning mod of any applied moddable content will be unloaded from the game.
    - **Revert Default:** Any applied moddable content should be reverted to use the original game asset.
    - **RevertConflictingModOrDefault:** Any applied moddable content should be replaced by content from any previously conflicting mods or reverted to the default game asset if no other conflicting mods are loaded.
    - **Do Nothing:** Take no action. Note that this may cause some asset references to be broken causing potential game issues.
- **Allow Asset Indexing:** Can moddable assets include array indexing syntax at the end of their file name indicating which array element it should be applied to. For example:

'myMaterialAsset[1].mat' would cause the asset to be applied at array index '1' of the target component. This is useful for components that accept more than one asset of a particular type such as renderer components.

- **Revert Content On Destroy:** Should the moddable content be reverted to the default game asset when the moddable content component has been removed from a game object. Note that once a moddable content component has been removed from a game object, conflict and unload handling can no longer take place.
- **Asset Naming Scheme:** Click this button to open the moddable naming scheme window where you can view and edit the asset naming scheme for all moddable content.

Moddable Naming Scheme

The moddable naming scheme window allows you to view and edit the naming scheme used for mod assets which should be mapped to a moddable game object or prefab.



You can modify the default asset name used for each type of supported asset. All moddable content that is created will use this naming scheme to generate full asset locations for modders to use. You can use the 'Apply To All' button to update any existing moddable objects with the new asset naming scheme if you decide to change it at a later date.

Mesh Asset:	mesh	Apply To All
Material Asset:	material	Apply To All
Texture Asset:	texture	Apply To All
AnimatorController Asset:	animator	Apply To All

Moddable Objects

ModdableContentScene/MultiMaterialCube

Material	ModdableContentScene/MultiMaterialCube/material	[]
Texture	ModdableContentScene/MultiMaterialCube/texture	[]

Figure 20

Asset Type Naming Scheme

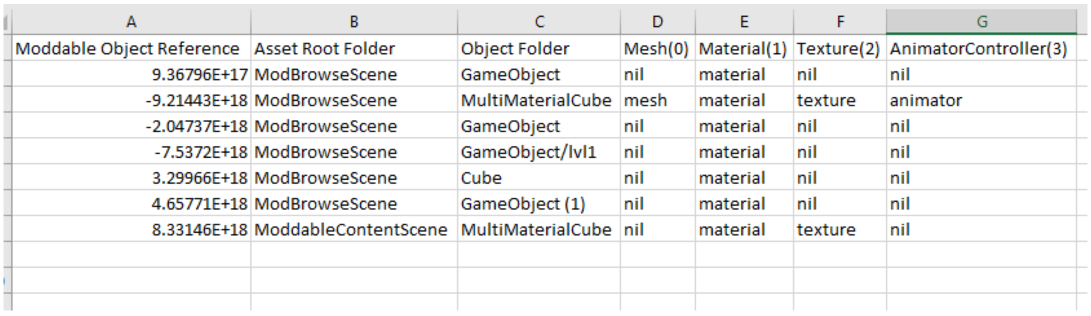
This section of the moddable naming scheme window allows you to specify the individual asset names used when a new moddable object is added to the naming scheme. You can see that the registered moddable object 'MultiMaterialCube' has '2' moddable asset types register and each asset path uses the corresponding name that was defined. You can change the naming scheme for individual assets at any time however it will only be applied to newly create moddable objects. You can forcefully update all registered moddable objects to use the name asset naming scheme by clicking the 'Apply To All' button for the asset type. Note this this will overwrite all registered asset names for the target asset type.

Note that only the supported asset types are displayed in this section. Support for more types of asset may be added I the future.

Moddable Objects

The moddable objects section contains a list view of all registered moddable content which was either added to a scene game object or to a prefab. The display name for the moddable object will be the folder structure where modders should place moddable content for this asset. Expanding the foldout will reveal all allowed types of moddable assets and will provide the full file path (without extension) relative to the mod export folder where modders should create specified assets. For example: If a modder wanted to create a material mod for the 'MutliMaterialCube' object then they could create a material asset at the path 'Assets/<modexportfolder>/ModdableContentScene/MultiMaterialCube/material.mat'. This asset would the be detected, loaded and applied automatically once the mod has been loaded.

Export as CSV: There is also the option to export the moddable object naming scheme as a comma separated .csv file which may be useful for generating suitable modding documentation for your modders so they can understand which objects are moddable and where they can create new assets to map to certain moddable objects.



A	B	C	D	E	F	G
Moddable Object Reference	Asset Root Folder	Object Folder	Mesh(0)	Material(1)	Texture(2)	AnimatorController(3)
9.36796E+17	ModBrowseScene	GameObject	nil	material	nil	nil
-9.21443E+18	ModBrowseScene	MultiMaterialCube	mesh	material	texture	animator
-2.04737E+18	ModBrowseScene	GameObject	nil	material	nil	nil
-7.5372E+18	ModBrowseScene	GameObject/lvl1	nil	material	nil	nil
3.29966E+18	ModBrowseScene	Cube	nil	material	nil	nil
4.65771E+18	ModBrowseScene	GameObject (1)	nil	material	nil	nil
8.33146E+18	ModdableContentScene	MultiMaterialCube	nil	material	texture	nil

Figure 21

Custom Object Naming Scheme

If you would like more control over the auto-generated asset naming scheme for a particular object, it is possible to override the generation process by adding a 'ModdableContentNamingSchemeOverride' component to manually specify the desired paths. This can be setup via the inspector window or by code if you have a lot of objects to apply overrides to.

1. Identify the moddable content scene object that you want to setup and make sure that you have already added the 'ModdableContent' component as normal.
    a) In the editor, find the target object in the scene hierarchy and ensure that the object is selected.
    b) Via code, you could search the active scene for all 'UMod.Moddable.ModdableContent' components and target the game object for each found result. You could also search by name or tag if you only want to affect a particular object.
2. Add the 'ModdableContentNamingSchemeOverride' component to the game object. Note that it is possible for a game object to have multiple 'ModdableContent' components attached but only one instance of 'ModdableContentNamingSchemeOverride' can be added and will apply to each moddable content component.
    a) In the inspector window of the selected object, go to 'Add Component -> Scripts -> UMod.Moddable -> ModdableContentNamingSchemeOverride' to add the component.

b) Via code, add the component to the moddable game object as shown:

C# Code

```csharp
using UMod.Moddable;

public void Example()
{
	GameObject targetObject = ...;

	targetObject.AddComponent<ModdableContentNamingSchemeOverride>();
}
```

3. Fill out the override properties with the necessary values. These properties are:
   - **ModdableOverrideContentFolder:** This is the parent folder location relative to the mod export folder where mod content should be placed in order to be applied to the target game object.
   - **ModdableOverrideContentName:** This is the name of the end folder which when appended to the 'ModdableOverrideContentFolder' path gives the final output directory for mod content which can be applied to the target game object.
   - **ModdableOverrideAssetNames:** This is a lookup for each supported type of moddable asset which allows you to override the final asset name that the mod should use.

These properties are then used to calculate the final path location where mod content should be created in order to be applied to the target game object. The resulting override path can be calculated like so:

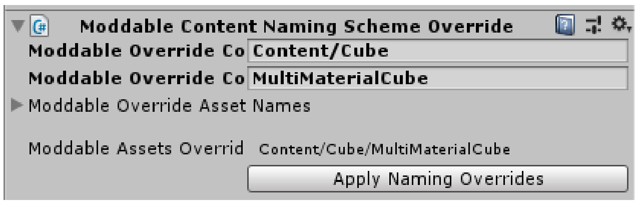
ModdableOverrideContentFolder + "/" + **ModdableOverrideContentName** + "/" + **ModdableOverrideAssetNames[ModdableAssetType.Material]**

For example, if the properties are set to:

- **ModdableOverrideContentFolder** = "Models/Props/Explosive"
- **ModdableOverrideContentName** = "ExplosiveBarrel"
- **ModdableOverrideAssetNames[ModdableAssetType.Material]** = "CustomMaterial"

The final output path would be *'Models/Props/Explosive/ExplosiveBarrel/CustomMaterial.mat'*

a) You can set these properties via the inspector window if working from the Unity editor and after making any changes you should hit the 'Apply Naming Overrides' button to apply the changes to the central modding naming scheme.



Moddable Content Naming Scheme Override
Moddable Override Co: Content/Cube
Moddable Override Co: MultiMaterialCube
Moddable Override Asset Names
Moddable Assets Overrid: Content/Cube/MultiMaterialCube

Figure 22

   b) From code, you can make changes via identically named properties of the 'ModdableContentNamingSchemeOverride' type and then call the method 'ApplyOverrides' to save the changes made.

4. After applying the naming scheme overrides you should see that the displayed output path for all 'ModdableContent' components attached to the same game object are updated to reflect the changes. This displayed path is now the actual path that your modders should use when creating moddable content for that particular game object.

Mod Scripting

uMod 2.0 allows mods that contain C# scripts to be created and used at runtime. These scripts will often execute in the same way as a traditional Unity script but there are a few things that may be different. This section will cover the entire scripting system for uMod 2.0 and how it works.

Requirements

The scripting system can only be used in the full version of uMod 2.0 and is completely disabled in the trial version meaning that script references attached to prefabs may fail to load correctly since the managed assembly is never loaded.

Basic Scripting Support

uMod 2.0 provides basic scripting support straight out of the box allowing modded scripts to be loaded and executed automatically without you needing to do anything. In order to achieve this, uMod provides an interface to the modder that can be used to create entry point scripts that will be initialized when a mod is loaded.

Concepts

The following section will cover the key concepts used by the uMod 2.0 scripting system. These concepts are fairly straightforward but play a key role in the system.

Script Domain

uMod 2.0 uses the concept of Script Domains which you can think of as a container for any externally loaded code. If a mod contains any code then a script domain will be created automatically during the loading procedure. Once this domain is created then any modded code will be automatically security checked and loaded into the domain provided that there are no security errors. You can access the script domain for a mod as shown below:

C# Code

```csharp
using UnityEngine;
using UMod;

// Required for access to the scripting api
using UMod.Scripting;

public class Example : MonoBehaviour
{
    ModHost host;

    void Start()
    {
        // This example assumes that 'host' has been initialized
and has a mod loaded
        ScriptDomain domain = host.ScriptDomain;

        // Print the name of each assembly
        foreach(ScriptAssembly assembly in domain.Assemblies)
            Debug.Log(assembly.Name);
    }
}
```

If you are particularly adept in C# then you may be familiar with 'AppDomains'. It is worth noting that uMod 2.0 does not create a separate AppDomain for mod scripts but instead loads all code into the current domain. This is due to a number of limitations with the mono implementation of .Net and as a result means that once loaded, mod code cannot be unloaded until the game exits. This is not a problem however as the code will sit idle in memory once a mod has been unloaded and will no longer be used. In general, mods will be loaded once at startup and may run until the game exits so this limitation is no real issue.

A Script Domain simply acts as a filter allowing only external code to be visible to the user instead of all code loaded code (including game code). As well as acting as a container, a Script Domain is also responsible for the loading of C# code or assemblies, as well as security validation to ensure that any loaded code does not make use of potentially dangerous assemblies or namespaces. For example, by default access to ‘System.IO’ is disallowed.

Script Assembly

A Script Assembly is a wrapper class for a managed assembly and includes many useful methods for finding Script types that meet a certain criteria. For example, finding types that inherit from UnityEngine.Object.

In order to obtain a reference to a Script Assembly you will need to use one of the ‘LoadAssembly’ methods of the Script Domain class. Depending upon settings, the Script Domain may also validate the code before loading to ensure that there are no illegal referenced assemblies or namespaces.

Script Assemblies also expose a property called ‘MainType’ which is particularly useful for external code that defines only a single class. For assemblies that contain more than one types, the MainType will be the first defined type in that assembly.

If you need more control of the assembly then you can use the ‘RawAssembly’ property to access the ‘System.Reflection.Assembly’ that the Script Assembly is managing. This can be useful if you need to access other assembly information that is not exposed in the script assembly.

Note: *Any assemblies or scripts that are loaded into a Script Domain at runtime will remain until the application ends. Due to the limitations of managed runtime, any loaded assemblies cannot be unloaded.*

Script Type

A Script Type acts as a wrapper class for ‘System.Type’ and has a number of unity specific properties and methods that make it easier to manage external code. For example, you can use the property called ‘IsMonoBehaviour’ to determine whether a type inherits from MonoBehaviour.

The main advantage of the Script Type class is that it provides a number of methods for type specific construction meaning that the type will always be created using the correct method.

- For types inheriting from MonoBehaviour, the Script Type will require a GameObject to be passed and will use the ‘AddComponent’ method to create an instance of the type.
- For types inheriting from ScriptableObject, the Script Type will use the ‘CreateInstance’ method to create an instance of the type.
- For normal C# types, the Script Type will make use of the appropriate constructor based upon the arguments supplied (if any).

This abstraction makes it far simpler to create a generic loading system for external code.

Script Proxy

A Script Proxy is used to represent an instance of a Script Type that has been created using one of the ‘CreateInstance’ methods. Script Proxies are a generic wrapper and can wrap Unity instances such as MonoBehaviours components as well as normal C# instances.

One of the main uses of a script proxy is to communicate with modded code by invoking methods or accessing fields and properties of mod scripts. This communication is possible without knowing the type of the modded code because the communication is string based. This simply means that in order to call a method you will specify the name as a string value instead of calling the method as you would normally. If you are familiar with unity's 'SendMessage' function then you will be right at home. For more information on mod communication take a look at the 'Interface Approaches' section later in this document.

A Script Proxy also implements the IDisposable interface which handles the destruction of the instance automatically based upon its type. Again this is done for ease of use and a unified method of destroying scripts.

- For instances inheriting from MonoBehaviour, the Script Proxy will call 'Destroy' on the instance to remove the component
- For instance inheriting from ScriptableObject, the Script Proxy will call 'Destroy' on the instance to destroy the data.
- For normal C# instances, the script proxy will release all references to the wrapped object allowing the garbage collector to reclaim the memory.

Note: Y*ou are not required to call 'Dispose' on the Script Proxy. It is simply included to provide a generic, type independent destruction method.*

Executing Assemblies

In order to implement scripting support you may need to find all of the executing scripts at some point (Probably after loading) so that you can call an event method or similar. By the time a mod is loaded, there may already be a number of executing scripts depending upon the mod content and various settings. You can access running scripts at any time by accessing the execution context for the mod:

C# Code

```
using UnityEngine;
using UMod;

// Required for access to the scripting api
using UMod.Scripting;

public class Example : MonoBehaviour
{
    // This example assumes that 'host' has been initialized and 
loaded.
    ModHost host;

    void Start()
    {
        // Get the exection context for the mod
        ScriptExecutionContext context = 
host.ScriptDomain.ExecutionContext;
    }
}
```

The execution context allows more control over scripting and also allows you to get an array of currently executing scripts as script proxies. You are also able to 'kill' the execution context which will immediately destroy all running mod script instances making sure to trigger the 'OnModUnloaded' event where necessary. This can be useful if you don't want mod code to execute any more but still want to access the asset content for the mod.

Loading Assemblies

It is possible to load managed assemblies into a script domain at any time but note that any code included with a mod will be automatically security checked and loaded. This section applies only to external code that is not packaged directly in the mod. There are a number of load methods that are provided that allow you to achieve this, all of which will perform additional security verification checks if enabled. All of these methods are called directly on an instance of a Script Domain which can be accessed from the 'ModHost' that loaded the mod.

The assembly loading methods are as follows:

- **LoadAssembly(string):** Attempts to load a managed assembly from the specified file path.
- **LoadAssembly(AssemblyName):** Attempts to load a managed assembly with the specified assembly name. Note that due to limitations, this method cannot security check code so it is recommended to use another 'Load' method were possible.
- **LoadAssembly(byte[]):** Attempts to load a managed assembly from its raw byte data. This is useful if you already have the assembly in memory or are downloading it from a remote source or similar.
- **LoadAssemblyFromResources(string):** This is a Unity specific load method any will attempt to load an assembly from the specified TextAsset in the resources folder.

All of these load methods return a Script Assembly which can be used to access Script Types using a number of 'Find' methods.

For more information on loading methods, take a look at the separate API documentation included with the package.

Activation

You may already know that uMod includes a basic modding interface for modders to use which provides appropriate call-backs such as 'OnModLoaded' and 'OnModUnloaded'. This interface also allows the modder to do many other useful things such as load assets, request scene changes and much more. Activation is defined as the process of initializing mod types that use this interface so that they receive the call-backs as expected. By default, any code that is included within the mod is automatically activated allowing the mod code to run as soon as a mod is loaded, however it can be very useful to load external code from another source and also activate the types within. You are able to manually activate a type at any time using the uMod scripting api. The following example shows how to activate an external managed assembly:

C# Code

```csharp
using UnityEngine;
using UMod;

// Required for access to the scripting api
using UMod.Scripting;

public class Example : MonoBehaviour
{
    // This example assumes that 'host' has been initialized and
loaded.
    ModHost host;

    void Start()
    {
        // Get the exection context for the mod
        ScriptExecutionContext context =
host.ScriptDomain.ExecutionContext;

        // Load an assembly from a file path
        ScriptAssembly assembly =
host.ScriptDomain.LoadAssembly("C:/Examples/Example.dll");

        // Activate the assembly
        ScriptProxy[] proxies = context.ActivateAssembly(assembly);

        // All of these proxies have now been activated and are
receiving mod events
        foreach(ScriptProxy proxy in proxies)
            Debug.Log(proxy.ScriptType);
    }
}
```

Interface Approaches

Once you have loaded an assembly or script into a Script Domain and created a Script Proxy instance, the next step you will likely want to take is to communicate with the types in some way

If you want to support scripts in user mods then uMod 2.0 allows you to do that relatively easily. In fact, all the hard work is already done and you don't need to do anything for basic scripts support. By default uMod will compile and build scripts into the mod during export which will be security checked, loaded and activated at runtime by uMod. This means that any scripts using the uMod interface will receive all mod events such as 'OnModLoaded' and that all mono behaviour mod scripts will be created when mod objects are instantiated.

This may be enough scripting support in some situations however for meaningful scripting support you will inevitable need to communicate between the game and the mod at some point. This requirement introduces a number of issues, namely how does the mod know what types are available in the game code and how they can be used. uMod 2.0 has 2 main method of cross communication which can be used in unison if desired. These communication types are known as 'Generic Communication' which uses string based type interaction, and 'Interface Communication' where the developer provides an additional interface assembly describing the types that can be used by the modder.

Generic Communication

Generic communication is considered as a non-concrete type of communication meaning that the type you want to communicate with is not known at compile time. This poses a few issues because you are unable to simply call a method on an unknown type. Fortunately uMod 2.0 includes a basic generic communication system that works using reflection and allows any class member to be accessed without knowing the runtime type. This method does however require that you know the name of the type and or member before you can communicate with it. At this point it is up to the developer to provide suitable modding documentation outlining the names or types and members that can be used with this communication method. For example, Unity provides documentation for the 'magic' events of the monobehaviour class such as 'Update' or 'Start'.

A Script Proxy is used to communicate with external code using string identifiers to access members. The following example shows how to create your own magic method type events:

C# Code

```csharp
using UnityEngine;
using DynamicCSharp;

public class Example : MonoBehaviour
{
    // This example assumes that 'proxy' is created before hand
    ScriptProxy proxy;

    void Start()
    {
        // Call the method 'OnScriptStart'
        proxy.SafeCall("OnScriptStart");
    }

    void Update()
    {
        // Call the method 'OnScriptUpdate'
        proxy.SafeCall("OnScriptUpdate");
    }
}
```

The above code shows how a method with the specified name can be called at runtime using the 'SafeCall' method. The following methods can be used to call methods on external scripts:

- Call: The call method will attempt to call a method with the specified name and upon error will throw an exception. Any exceptions thrown as a result of invoking the target method will also go unhandled and passed up the call stack so it is recommended that you use 'SafeCall' unless you want to implement your own error handling.
- SafeCall: The SafeCall method is essentially a wrapper for the 'Call' method and handles any exceptions that it throws. If the target method is not found then this method will fail silently.

When calling a method is it also very useful to be able to pass arguments to that method. uMod 2.0 allows any number of arguments to be passed provided that the passed types are known to both the game and the external script beforehand. A good candidate for this Unity types such as Vector3 or primitive types like int, string, and float. The target method must also accept the same argument list otherwise calling the method will fail.

uMod 2.0 also includes a way of accessing fields and properties of external scripts provided that their name is known beforehand. Again communication is achieved via the proxy but instead of calling a method you use either the 'Fields' or 'Properties' property of the proxy.

1. **Fields**

   Fields can have their values read from or written to so long as the assigned type matches the field type. If the types do not match then a type mismatch exception may be thrown.

   Unlike methods, there is no safe alternative for accessing fields using this method. If you want to be safe when accessing fields then you should catch any exceptions thrown.

   The following code shows how a field called 'testField' can be modified:

C# Code

```csharp
using UnityEngine;
using DynamicCSharp;

public class Example : MonoBehaviour
{
    // This example assumes that 'proxy' is created before hand
    ScriptProxy proxy;

    void Start()
    {
        // This example assumes that 'testField' is an int

        // Read the value of the field
        int old = proxy.Fields["testField"];

        // Print to console
        Debug.Log(old);

        // Set the value of the field
        proxy.Fields["testField"] = 123;
    }
}
```

2. Properties

Properties are a little different to fields because they are not required to implement a get and a set method. This means that certain properties cannot be written to or read from which means you have to be all the more careful.

If you attempt to read from or write to a property that does not support it, then a target exception may be thrown. As with fields, there is no safe alternative for accessing properties. If you want to be safe when accessing properties then you should catch any exceptions thrown.

The following code shows how a property called 'testProperty' can be accessed. The method is very similar with fields and properties.

C# Code

```csharp
using UnityEngine;
using DynamicCSharp;

public class Example : MonoBehaviour
{
    // This example assumes that 'proxy' is created before hand
    ScriptProxy proxy;

    void Start()
    {
        // This example assumes that 'testProperty' is an int

        // Read the value of the property
        int old = proxy.Properties["testProperty"];

        // Print to console
        Debug.Log(old);

        // Set the value of the property
        proxy.Properties["testProperty"] = 456;
    }
}
```

Interface Communication

The second communication method is fairly more advanced than the previous method however it will allow for concrete type communication as opposed to loose string based communication. It should also offer improved runtime performance since it does not rely on reflection to access types and members.

The implementation involves creating a shared interface containing any number of base classes or C# interfaces that all external code must inherit from. The best way to do this would be to create a separate managed assembly containing these shared base types and make it available to the modded code. In order to ensure that modders can access this shared assembly it is recommended that you create your own modified exported package that includes this assembly. By doing this, any scripts created by the modder will automatically reference this assembly.

Providing a guide for creating a separate interface assembly is beyond the scope of this documentation however there are a number of very useful Unity specific tutorial online to cover this. You should however ensure that the assembly targets .Net 3.5 framework as higher versions will result in 'TypeLoadExceptions' being thrown.

Once you have defined your interface then you are able to load and call the external code as if it were part of your game.

As an example, we will use the following C# interface to show how the process would work:

C# Code

```csharp
using UnityEngine;

public interface IExampleBase
{
    void SayHello();

    void SayGoodbye();
}
```

As you can see the interface contains 2 methods which must be implemented and for now we will assume that this interface is defined in an assembly called 'ExampleInterface.dll'. As mentioned before, this assembly must be accessible to both the game code and the mod code which means that you should distribute this interface to modders.

We will now require our modded code to implement this interface in order for us to load it into the game. If it does not then we will simply ignore the code and assume that it is either irrelevant or have been activated by uMod automatically. Our example mod code is simply as follows:

C# Code

```csharp
using UnityEngine;

public class Test: IExampleBase
{
    void SayHello()
    {
        Debug.Log("Hello");
    }

    void SayGoodbye()
    {
        Debug.Log("Goodbye");
    }
}
```

As you can see, the example code is very basic and will simply print to the Unity console when one of its two methods are called.

At this point we will now assume that we have exported a mod containing this ‘Test’ class using the uMod 2.0 exporter. The next step then is to identify and load the modded code so that we can call the ‘SayHello’ and ‘SayGoodbye’ methods from the game code. Since the code has been included in the mod, it will be automatically loaded by uMod 2.0 when the mod is loaded meaning that the assembly has already been loaded. As a result we can access the assembly in memory instead of loading it manually. The following C# code shows how we can access all types that inherit from our ‘IExampleBase’ interface that we created earlier.

C# Code

```csharp
using UnityEngine;
using UMod;

// Required for access to the scripting api
using UMod.Scripting;

public class Example : MonoBehaviour
{
    // This example assumes that 'host' has been initialized and
loaded.
    ModHost host;

    void Start()
    {
        // Look through all assemblies loaded in the mod domain
        foreach(ScriptAssembly assembly in
host.ScriptDomain.Assemblies)
        {
            // This method will find all types in the assembly
that implement our interface
            ScriptType[] types =
assembly.FindAllSubtypesOf<IExampleBase>();
        }
    }
}
```

As you can see, we now have an array of Script Types, all of which implement our ‘IExampleBase’ interface. That means we can be sure that all of these types implement both methods defined in the interface so the next thig we can do is call those methods on each type:

C# Code

```csharp
using UnityEngine;
using UMod;

// Required for access to the scripting api
using UMod.Scripting;

public class Example : MonoBehaviour
{
    // This example assumes that 'host' has been initialized and
loaded.
    ModHost host;

    void Start()
    {
        // Look through all assemblies loaded in the mod domain
        foreach(ScriptAssembly assembly in
host.ScriptDomain.Assemblies)
        {
            // This method will find all types in the assembly
that implement our interface
            ScriptType[] types =
assembly.FindAllSubtypesOf<IExampleBase>();

            // Create an instance of all types
            foreach(ScriptType type in types)
            {
                // Create a raw instance of our type
                IExampleBase instance =
type.CreateRawInstance<IExampleBase>();

                // Call its methods as you would expect
                instance.SayHello();
                instance.SayGoodbye();
            }
        }
    }
}
```

As you would expect, before we can use the type we need to create an instance of it which is done using the ‘CreateRawInstance’ of the Script Type. The main difference this method has when compared with the ‘CreateInstance’ method is that the concrete type is returned as opposed to a managing Script Proxy. This means that we can access the result directly as our ‘IExampleBase’ interface and the conversion will work fine. After that we now have an instance of the ‘Test’ class defined earlier stored as the ‘IExampleBase’ interface meaning that we can now call the methods directly.

Although the interface approach requires more setup to get working, it is worth the effort as you gain type safety as well as extra performance when compared with the proxy communication method. This is due to the fact that proxies rely on reflection under the hood in order to call methods and access members which will always be slower than simply calling a method.

Report a Bug

At Trivial Interactive we test our assets thoroughly to ensure that they are fit for purpose and ready for use in games but it is often inevitable that a bug may sneak into a release version and only expose its self under a strict set of conditions.

If you feel you have exposed a bug within the asset and want to get it fixed then please let us know and we will do our best to resolve it. We would ask that you describe the scenario in which the bug occurs along with instructions on how to reproduce the bug so that we have the best possible chance of resolving the issue and releasing a patch update.

http://trivialinteractive.co.uk/bug-report/

Request a feature

uMod 2.0 was designed as a complete runtime modding solution, however if you feel that it should contain a feature that is not currently incorporated then you can request to have it added into the next release. If there is enough demand for a specific feature then we will do our best to add it into a future version.

http://trivialinteractive.co.uk/feature-request/

Contact Us

Feel free to contact us if you are having trouble with the asset and need assistance. Contact can either be made by the contact options on the asset store or buy the link below.

Please attempt to describe the problem as best you can so we can fully understand the issue you are facing and help you come to a resolution. Help us to help you :-)

http://trivialinteractive.co.uk/contact-us/