

User Guide

uMod 2.0

Modding support made easy

Trivial Interactive

Version 2.0.8

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

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

Features

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

Contents

CONTENTS	2
<u>INSTALLING.....</u>	<u>4</u>
INSTALL.....	4
UNINSTALL	4
UPDATING	4
PACKAGES	5
<u>MOD ESSENTIALS</u>	<u>6</u>
MOD STRUCTURE.....	6
INSTALL MODS	7
MOD PATH.....	8
MOD HOST.....	9
<u>MOD SETTINGS</u>	<u>11</u>
GLOBAL SETTINGS	11
GENERAL SETTINGS.....	11
ASSET SETTINGS	12
SECURITY SETTINGS.....	13
SCRIPT SETTINGS	13
<u>MOD LOADING</u>	<u>14</u>
INITIALIZE UMOD.....	14
LOAD MOD.....	14
COROUTINE MOD LOADING	17
DOWNLOADING MODS	17
COMMAND LINE LOADING	18
UNLOAD MOD	20
<u>MOD ASSETS.....</u>	<u>21</u>
LOAD MOD ASSETS	21
COROUTINE ASSET LOADING.....	23
UNLOADING ASSETS	23
SHARED ASSETS	24

MOD SCRIPTING	25
REQUIREMENTS	25
CONCEPTS	26
SCRIPT DOMAIN	26
SCRIPT ASSEMBLY	27
SCRIPT TYPE	27
SCRIPT PROXY	27
EXECUTING ASSEMBLIES	29
LOADING ASSEMBLIES	30
ACTIVATION.....	30
INTERFACE APPROACHES.....	32
GENERIC COMMUNICATION	32
INTERFACE COMMUNICATION.....	36

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.

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.*

Updating

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.

Packages

In order to keep the plugin organised and uncluttered uMod 2.0 distributes a number of packages that can be selectively installed as needed to add features to uMod. At present, there are only packages for offline documentation and scripting reference but in the near future packages such as steam workshop support will be included which can be selectively installed.

You are able to manage the installed packages by opening the packages window from 'Tools -> UMod 2.0 -> Packages' where you should see a window as shown below:

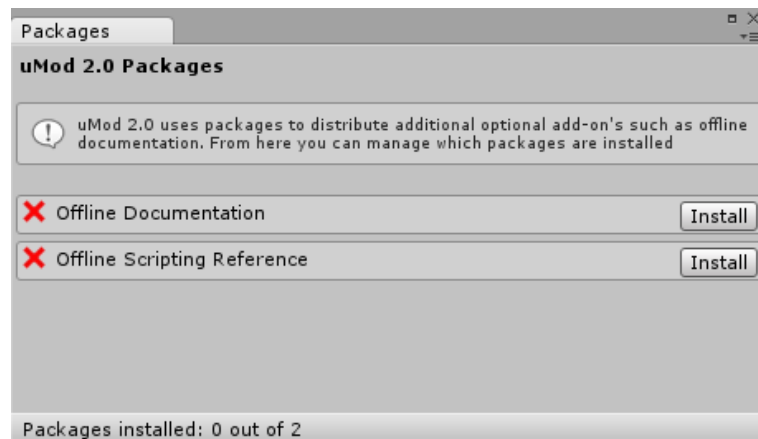


Figure 1

From this window you are able to install, uninstall and repair packages in order to extend the functionality of uMod 2.0. You are able to get a description of the package by hovering the cursor over the package name.

Note: All packages released for uMod 2.0 will be free. The packaging system simply acts as a filter allowing unused aspects of the package to be omitted thus reducing build / compile times and reducing the output size of your application.

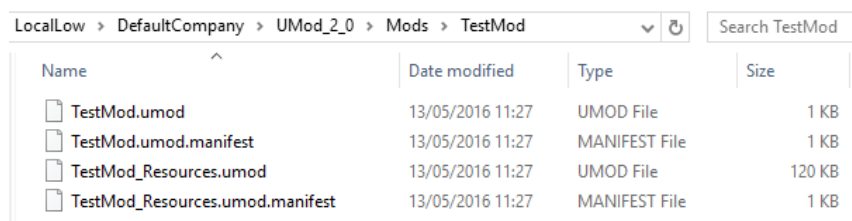
Mod Essentials

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

Mod Structure

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

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



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

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

Figure 2

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

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

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

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

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

```
1  using UnityEngine;
2  using UModHost;
3
4  public class ModPathExample : MonoBehaviour
5  {
6      private void Start()
7      {
8          // Get the app data path for the current operating system
9          string appDataPath = Application.persistentDataPath;
10
11         // Append a 'Mods' folder to the path
12         string modInstallPath = Path.Combine(appDataPath, "Mods");
13
14         // Create a mod path for 'Example Mod'
15         ModPath path = new ModPath(Path.Combine(modInstallPath,
16 "ExampleMod"));
17
18         // Begin loading the mod
19         Mod.LoadMod(path);
20     }
21 }
```

Mod Path

uMod 2.0 introduces the concept of Mod Paths which are used to load a mod from a specific location. Mod paths can be created from a simple string path and allows the developer to either load from the local file system, or from a remote server. Mod paths are also capable of storing other forms of path value such as relative paths and IP addressed paths pointing to server files.

Below are a few examples of valid mod paths (Assuming that the Mod Folder they point to exists):

- "C:/Mods/ExampleMod"
- "Mods/ExampleMod"
- "File:///C:/Mods/ExampleMod"
- "93.48.12.90/Mods/ExampleMod"

Note: These paths do not point to a specific file, but instead point to the root 'Mod Folder' as described in [Mod Structure](#). This folder will always have the same name as the mod and the files inside will automatically be discovered during loading.

The following C# code shows how to create a mod path from a raw string path value and then begin loading the mod:

C# Code

```
1 using UnityEngine;
2 using UMod;
3
4 public class ModPathExample : MonoBehaviour
5 {
6     private void Start()
7     {
8         // Create the mod path from the string path
9         ModPath path = new
10         ModPath("C:/ExampleGame/Mods/ExampleMod");
11
12         // Begin loading the mod
13         Mod.LoadMod(path);
14     }
15 }
```


Mod Host

uMod 2.0 uses containers known as mod hosts to manage a single mod within the game and 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.

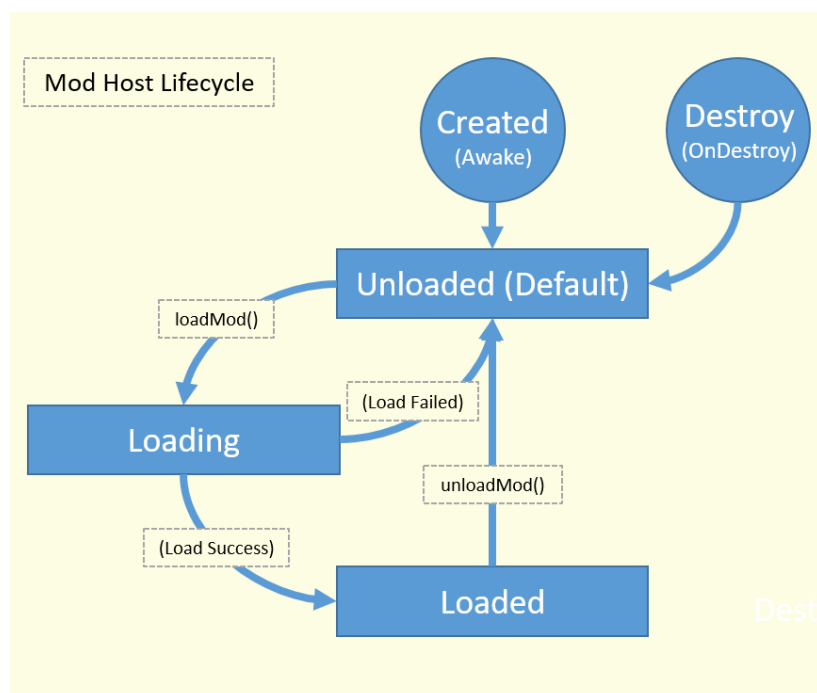


Figure 3

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

```
1 using UnityEngine;
2 using UMod;
3
4 public class ModHostExample : MonoBehaviour
5 {
6     private void Start()
7     {
8         // Create the mod host using the static method
9         ModHost host = ModHost.CreateNewHost();
10    }
11 }
```

Mod Settings

Global 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:

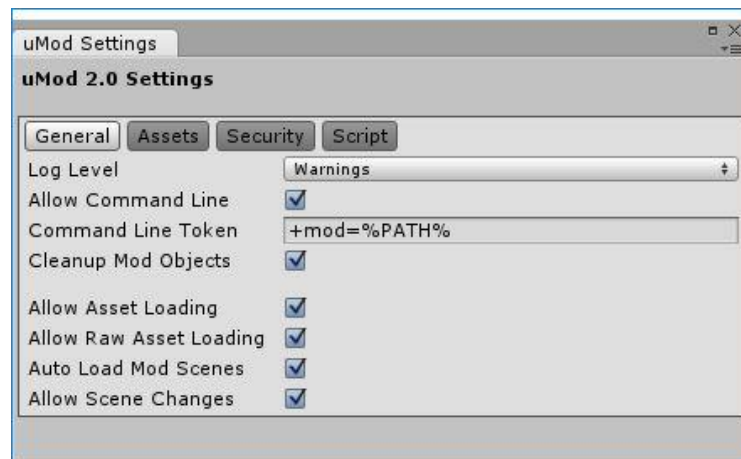


Figure 4

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 Settings

- **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.
- **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 Token:** 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%' value 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 placed on command line launching. It is recommended that this value remains at the default for maximum compatibility, however if you do decide to change it then you will need to inform your modding community of the format string as it is required by the exporter tool to correctly launch your game.

- **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.
- **Allow Asset Loading:** When enabled, mod scripts will be able to issue load requests for assets packaged within the mod. It is highly recommended that this value remains enabled otherwise modders may not be able to access any prefabs that are included in the mod.
- **Allow Raw Asset Loading:** When enabled, mod scripts will be able to issue raw asset load requests. A raw asset is defined as an external un-compiled asset that is provided in a common format such as '.png' for textures.
- **Auto Load Mod Scenes:** When enabled, mod hosts will automatically attempt to load a mod's default scene. If there is no default scene then the mod host will simply fail silently.
- **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 your own scene load handler.

Asset Settings

The asset settings tab is shown below:

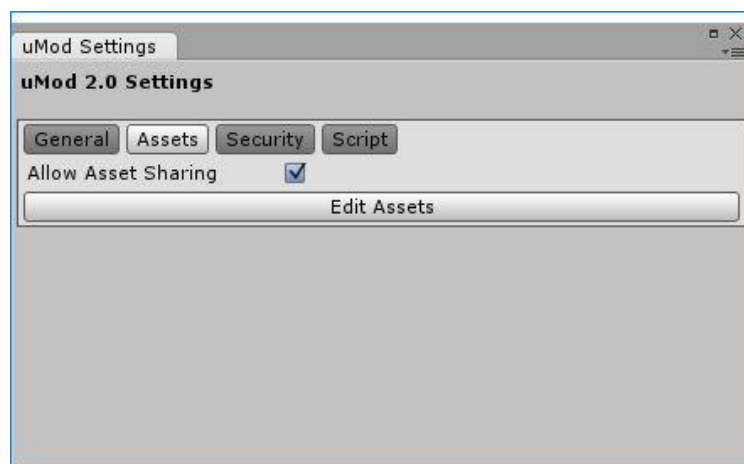


Figure 5

- **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.
- **Edit Assets:** By clicking this button, you will open up the asset collection window where all shared assets can be organised. For more information take a look at the [Shared Assets](#) section

Security Settings

To be added.

Script Settings

To be added

Mod Loading

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

Initialize UMod

Before you get carried away and start issuing loading calls to uMod there is an important step that should be done beforehand. Before you make any call into the API you should ensure that uMod is initialized by calling the static initialize method as shown below:

C#	Code
1	<code>using UnityEngine;</code>
2	<code>using UMod;</code>
3	
4	<code>public class ModInitializeExample : MonoBehaviour</code>
5	<code>{</code>
6	<code>private void Start()</code>
7	<code>{</code>
8	<code>// Initialize uMod</code>
9	<code>Mod.Initialize();</code>
10	<code>}</code>
11	<code>}</code>

You should call this method at the start of your game. A typical place to call this method would be a pre-menu splash screen or the start-up menu such as a main menu. Do however make sure that you call this method only once.

Note: This method may throw a 'ModNotSupportedException' when called from an unsupported platform. If you want to check whether the current platform is supported beforehand you can use the static property 'Mod.IsPlatformSupported' which will return true for desktop and editor platforms.

Load Mod

Loading a mod in uMod 2.0 is relatively straight forward and can be achieved using a single function 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. For more information about valid paths and formats take a look at the [Mod Paths](#) section.

C# Code

```
1 using UnityEngine;
2 using UMod;
3
4 public class ModLoadExample : MonoBehaviour
5 {
6     private void Start()
7     {
8         // Create a mod path from a string path
9         ModPath path = new ModPath("C:/Mods/TestMod");
10
11        // Issue a mod load request
12        Mod.LoadMod(path);
13    }
14 }
```

Note: An important thing to note when specifying a mod path is that the path does not actually point to a file, but instead points to a folder called the 'Mod Folder'. This is very important as uMod will automatically identify the contents of the folder when loading. For more information take a look at the [Mod Structure](#) section.

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 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.

uMod 2.0 does all the heavy work involved with mod loading from a background thread leaving the main thread free to do anything you may desire. This does beg the question, "How do I access the loaded mod and how do I know when loading is complete?"

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 assigned mod. For detailed information take a look at the [Mod Host](#) section. The host is a state object and has a number of events that are triggered when its state is changed, for example from loading to loaded. These events are implemented as C# delegates and can be subscribed to by listener method that act upon these state changes.

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

C# Code

```
1 using UnityEngine;
2 using UMod;
3
4 public class ModLoadExample : MonoBehaviour
5 {
6     private void Start()
7     {
8         // Create a mod path from a string path
9         ModPath path = new ModPath("C:/Mods/TestMod");
10
11        // Issue a mod load request
12        ModHost host = Mod.LoadMod(path);
13
14        // Add an event listener
15        host.OnModLoadComplete += OnModLoadComplete;
16    }
17
18    private void OnModLoadComplete(ModLoadCompleteArgs args)
19    {
20        if(args.IsLoaded == true)
21        {
22            Debug.Log("The mod is loaded!");
23        }
24        else
25        {
26            Debug.LogError("The mod could not be loaded - " +
27                args.ErrorStatus);
28        }
29    }
30 }
```


Coroutine Mod Loading

In order to make uMod easier to use within Unity, there are a number of custom yield instructions that can be used inside coroutines to wait for specified events. One of these yield instructions is 'WaitForModLoad' which can be used to wait for a mod to be loaded before continuing execution. The following C# code shows a basic example where the coroutine will yield until the mod load request has finished before continuing.

C# Code

```
1  using UnityEngine;
2  using UMod;
3
4  public class ModCoroutineExample : MonoBehaviour
5  {
6      private IEnumerator Start()
7      {
8          // Create a mod path
9          ModPath path = new ModPath("C:/Mods/ExampleMod");
10
11         // Create a request
12         WaitForModLoad request = new WaitForModLoad(path);
13
14         // Wait for the task to complete
15         yield return request;
16
17         // Check the status of the load
18         Debug.Log("Loaded = " + request.WasLoadSuccessful);
19     }
20 }
```

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. The loading will begin as soon as the yield instruction is created with the 'new' keyword so you should not create new cached instances at class level.

For more information take a look at the included example scripts which show these custom instructions 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"
```

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

```
1  using UnityEngine;
2  using UMod;
3
4  public class ModProgressExample : MonoBehaviour
5  {
6      private ModHost host = null;
7
8      private void Start()
9      {
10         // Create a mod path from a url
11         ModPath path = new
12         ModPath("trivialinteractive.co.uk/Mods/ExampleMod");
13
14         // Issue a mod load request
15         host = Mod.loadMod(path);
16
17         // Add an event listener for progress updates
18         host.onModLoadProgress += onModLoadProgress;
19     }
20
21     private void onModLoadProgress (ModLoadProgressArgs args)
22     {
23         // This will be true if the host is downloading from a
24         server
25         if(args.IsDownloading == true)
26         {
27             Debug.Log("Download Progress: " + args.Progress);
28             Debug.Log("Download Speed: " + args.Speed);
29         }
30     }
31 }
```

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 and 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 parsed 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 +mod=C:/Mods/AnotherMod
```

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

Unload Mod

Once you have successfully implemented mod loading one of the next things you will 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

```
1  using UnityEngine;
2  using UMod;
3
4  public class ModUnloadExample : MonoBehaviour
5  {
6      private ModHost host = null;
7
8      private void Start()
9      {
10         // Create a mod path from a string path
11         ModPath path = new ModPath("C:/Mods/TestMod");
12
13         // Issue a mod load request
14         host = Mod.loadMod(path);
15
16         // Add an event listener
17         host.onModLoadComplete += onModLoadComplete;
18     }
19
20     private void onModLoadComplete(ModLoadCompleteArgs args)
21     {
22         if(args.IsLoaded == true)
23         {
24             // Unload the host when the load has successfully
25             // completed
26             host.unloadMod();
27         }
28     }
29 }
```

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 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 may need to be able to load a number of assets from a 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 for 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 two 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 choose 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':** The 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.

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 methods 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 asset from the mod with the specified name. These calls will cause loading to take place on the main thread and 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, a delegate must be passed to the load method which will be invoked when the load has finished.
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

```
1  using UnityEngine;
2  using UMod;
3
4  public class UModAssetExample : MonoBehaviour
5  {
6      private ModHost host = null;
7
8      private void Start()
9      {
10         // We assume that the 'host' already has a valid mod
11         loaded.
12
13         if(host.Assets.exists("MyAsset") == true)
14         {
15             // Load the asset from the mod
16             GameObject go = host.Assets.load("MyAsset") as
17             GameObject;
18
19             // Create an instance in the scene
20             Instantiate(go, Vector3.zero, Quaternion.identity);
21         }
22     }
```

Coroutine 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

```
1  using UnityEngine;
2  using UMod;
3
4  public class ModAssetCoroutineExample : MonoBehaviour
5  {
6      private IEnumerator Start()
7      {
8          // Create a request
9          WaitForAssetLoad request = new WaitForAssetLoad("MyAsset");
10
11         // Wait for the task to complete
12         yield return request;
13
14         // Check the status of the load
15         Debug.Log("Loaded = " + request.WasLoadSuccessful);
16     }
17 }
```

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. The loading will begin as soon as the yield instruction is created with the 'new' keyword so you should not create new cached instances at class level.

For more information take a look at the included example scripts which shows this custom instruction 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:

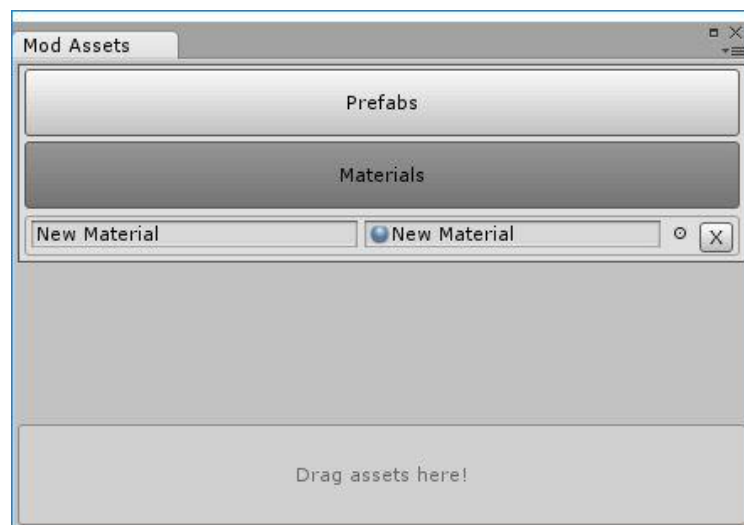


Figure 6

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.

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

```
1  using UnityEngine;
2  using UMod;
3
4  // Required for access to the scripting api
5  using UMod.Scripting;
6
7  public class Example : MonoBehaviour
8  {
9      ModHost host;
10
11      void Start()
12      {
13          // This example assumes that 'host' has been initialized
14          // and has a mod loaded
15          ScriptDomain domain = host.ScriptDomain;
16
17          // Print the name of each assembly
18          foreach (ScriptAssembly assembly in domain.Assemblies)
19              Debug.Log(assembly.Name);
20      }
21  }
```

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 construction 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 `MonoBehaviour`s 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: *You 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

```
1 using UnityEngine;
2 using UMod;
3
4 // Required for access to the scripting api
5 using UMod.Scripting;
6
7 public class Example : MonoBehaviour
8 {
9     // This example assumes that 'host' has been initialized and
10    loaded.
11    ModHost host;
12
13    void Start ()
14    {
15        // Get the execution context for the mod
16        ScriptExecutionContext context =
17        host.ScriptDomain.ExecutionContext;
18    }
19 }
```

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

```
1 using UnityEngine;
2 using UMod;
3
4 // Required for access to the scripting api
5 using UMod.Scripting;
6
7 public class Example : MonoBehaviour
8 {
9     // This example assumes that 'host' has been initialized and
10    loaded.
11    ModHost host;
12
13    void Start()
14    {
15        // Get the exection context for the mod
16        ScriptExecutionContext context =
17        host.ScriptDomain.ExecutionContext;
18
19        // Load an assembly from a file path
20        ScriptAssembly assembly =
21        host.ScriptDomain.LoadAssembly("C:/Examples/Example.dll");
22
23        // Activate the assembly
24        ScriptProxy[] proxies = context.ActivateAssembly(assembly);
25
26        // All of these proxies have now been activated and are
27        // receiving mod events
28        foreach(ScriptProxy proxy in proxies)
29            Debug.Log(proxy.ScriptType);
30    }
31 }
```

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 inevitably 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 methods 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

```
1 using UnityEngine;
2 using DynamicCSharp;
3
4 public class Example : MonoBehaviour
5 {
6     // This example assumes that 'proxy' is created before hand
7     ScriptProxy proxy;
8
9     void Start ()
10    {
11        // Call the method 'OnScriptStart'
12        proxy.SafeCall("OnScriptStart");
13    }
14
15    void Update ()
16    {
17        // Call the method 'OnScriptUpdate'
18        proxy.SafeCall("OnScriptUpdate");
19    }
20 }
```

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 it is 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

```
1 using UnityEngine;
2 using DynamicCSharp;
3
4 public class Example : MonoBehaviour
5 {
6     // This example assumes that 'proxy' is created before hand
7     ScriptProxy proxy;
8
9     void Start()
10    {
11        // This example assumes that 'testField' is an int
12
13        // Read the value of the field
14        int old = proxy.Fields["testField"];
15
16        // Print to console
17        Debug.Log(old);
18
19        // Set the value of the field
20        proxy.Fields["testField"] = 123;
21    }
22 }
```

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

```
1 using UnityEngine;
2 using DynamicCSharp;
3
4 public class Example : MonoBehaviour
5 {
6     // This example assumes that 'proxy' is created before hand
7     ScriptProxy proxy;
8
9     void Start()
10    {
11        // This example assumes that 'testProperty' is an int
12
13        // Read the value of the property
14        int old = proxy.Properties["testProperty"];
15
16        // Print to console
17        Debug.Log(old);
18
19        // Set the value of the property
20        proxy.Properties["testProperty"] = 456;
21    }
22 }
```

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
1  using UnityEngine;
2
3  public interface IExampleBase
4  {
5      void SayHello();
6
7      void SayGoodbye();
8  }
```

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

```
1 using UnityEngine;
2
3 public class Test: IExampleBase
4 {
5     void SayHello()
6     {
7         Debug.Log("Hello");
8     }
9
10    void SayGoodbye()
11    {
12        Debug.Log("Goodbye");
13    }
14 }
```

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

```
1 using UnityEngine;
2 using UMod;
3
4 // Required for access to the scripting api
5 using UMod.Scripting;
6
7 public class Example : MonoBehaviour
8 {
9     // This example assumes that 'host' has been initialized and
10    loaded.
11    ModHost host;
12
13    void Start()
14    {
15        // Look through all assemblies loaded in the mod domain
16        foreach (ScriptAssembly assembly in
17        host.ScriptDomain.Assemblies)
18        {
19            // This method will find all types in the assembly
20            that implement our interface
21            ScriptType[] types =
22            assembly.FindAllSubtypesOf<IExampleBase>();
23        }
24    }
25 }
```

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 thing we can do is call those methods on each type:

```
C# Code
1  using UnityEngine;
2  using UMod;
3
4  // Required for access to the scripting api
5  using UMod.Scripting;
6
7  public class Example : MonoBehaviour
8  {
9      // This example assumes that 'host' has been initialized and
10     loaded.
11     ModHost host;
12
13     void Start()
14     {
15         // Look through all assemblies loaded in the mod domain
16         foreach (ScriptAssembly assembly in
17             host.ScriptDomain.Assemblies)
18         {
19             // This method will find all types in the assembly
20             // that implement our interface
21             ScriptType[] types =
22                 assembly.FindAllSubtypesOf<IExampleBase>();
23
24             // Create an instance of all types
25             foreach (ScriptType type in types)
26             {
27                 // Create a raw instance of our type
28                 IExampleBase instance =
29                     type.CreateRawInstance<IExampleBase>();
30
31                 // Call its methods as you would expect
32                 instance.SayHello();
33                 instance.SayGoodbye();
34             }
35         }
36     }
37 }
```

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.