User Guide

Roslyn C#

Runtime C# compiler

Trivial Interactive

Version 1.9.x

Roslyn C# allows runtime loading of assemblies and C# scripts at runtime using the Roslyn compiler making it trivial to add modding support to you game by allowing your users to write C# scripts. In addition, Roslyn C# also allows strict security restrictions to be enforced as specified by the developer meaning that external code can run safely.

Limitations

- Requires full .Net 4.x or .Net Standard equivalent (Cannot run on legacy mono)
- AOT platforms such as IOS are not supported.
- Web GL is not supported due an IL2CPP build error but we have reported this error and if it gets fixed we may be able to add support.
- Scripts must be compiled before they can be executed.

Features

- Compile and run C# scripts at runtime.
- Fast Execution – once compiled, external scripts run as fast as game scripts.
- Allows for modding support to be added easily.
- Pre-load security checks mean that unsafe code can be identified and discarded.
- Support for loading assemblies and C# scripts.
- All scripts use custom namespaces to prevent clashing type names.
- Support for non-concrete interface using script proxies.
- Simple and clean API for accessing types and proxies.
- Cached member tables for quick reflection.
- Automatic construction of types using the appropriate method (AddComponent, CreateInstance, new)
- Comprehensive .chm documentation of the API for quick and easy reference.
- Fully commended partial C# source code included.

Support for PC, Mac, and Linux platforms. Roslyn C# may work without issue on other platforms however we will only offer support for the listed platforms.

Contents

Upgrade Guide

Upgrade to 1.5.0

If you are upgrading to Roslyn C# 1.5.x or newer, then there are a number of breaking changes which may cause a number of console errors upon completing the upgrade.

The following types have been renamed:

- **IScriptMemberProxy:** Has been renamed to **IScriptDataProxy**.

The following members have been renamed:

- **ScriptAssembly.RawAssembly:** Has been renamed to **ScriptAssembly.SystemAssembly**.
- **ScriptType.RawType:** Has been renamed to **ScriptType.SystemType**.Upgrade to 1.2.0

The property **ScriptAssembly.MainType** will now behave differently. The property will now return the first defined user class if one exists, or the first defined struct or enum if no classes are defined. This property will no longer return the internal '<Module>' type which may have been returned in the past.

Upgrade to 1.2.0

If you are upgrading from a Roslyn C# version older than 1.2.0 then it should be noted that a number of breaking changes were made which may cause compile errors upon upgrading to the latest version. The changes are mostly limited to the assembly referencing API:

- **IList<string> ScriptDomain.RoslynCompilerService.References:** This property has been removed and you will receive compiler errors if you have code which accessed this property. You should now use the alternate property 'ReferenceAssemblies' in order to add, remove or otherwise manage assembly references.
- **IList<Assembly> ScriptDomain.RoslynCompilerService.ReferenceAssemblies:** This property has been replaced with:
    - **IList<IMetadataReferenceProvider> ScriptDomain.RoslynCompilerService.ReferenceAssemblies.** The return value for the property has been changed and as a result you could see compiler errors if you access this property.

  Solution

  If you have any code which references the above properties then you will need to change the code as shown below:

  Reference Property

  It is likely that you used this property to add an assembly reference name or path to the compiler. To do this you will now need to use the method:

  ```
  IMetadatReferenceProvider AssemblyReference.FromNameOrFile(referenceName)
  ```

This will convert a string containing an assembly name or assembly path to an IMetadataReferenceProvider object which can then be added to the 'ReferenceAssemblies' collection.

ReferenceAssemblies Property

The return value for this property has been changed and you will now need to use the 'AssemblyReference' type to get a suitable IMetadataReferenceProvider object representing the assembly reference you wish to add. Previously this property was used for adding assemblies which were already loaded into memory. This can still be achieved by using:

```
IMetadataReferenceProvider AssemblyReference.FromAssembly(assembly)
```

This will convert a System.Reflection.Assembly instance to an IMetadataReferenceProvider which is compatible with the 'ReferenceAssemblies' collection. Note that the specified assembly must have a non-empty 'Location' property otherwise this will not work.

- **Params string[] parameters:** All compile methods of the ScriptDomain type which accept multiple source strings/filepath arguments no longer accept a varying amount of string arguments. Instead a fixed array must now be provided whereas previously you could provide a single string or multiple due to the params nature of the parameter.
  Solution

  If you want to pass multiple sources to the method in the form of a string array then no change is required however if you want to pass a single source string to a plural compile method such as 'CompileAndLoadFiles' then you should use the singular alternative method, in this case 'CompileAndLoadFile'.

Quick Start

This section will cover the steps required to get up and running as quickly as possible. More detailed information is provided later in the document.

1. **Install package**

   Open the project that you want to install Roslyn C# into (Unity 2018.1.0f2 and onwards are officially supported but other versions may work) and open the player settings 'Edit -> Project Settings -> Player'. Under the 'Other Settings' section find the 'Configuration' heading and ensure that 'Scripting Runtime Version' is set to '.Net 4.x Equivalent'.

   You can now install the package as you would do normally by downloading from the asset store.

2. **Create a Domain**

   The next thing you will need to do is create a Script Domain where all your external code will be loaded into. A script domain is simply a container where all loaded code is stored and executed. The following C# code shows how a domain can be created:

C# Code

```
1  using UnityEngine;
2  using RoslynCSharp;
3
4  public class Example : MonoBehaviour
5  {
6      // Our script domain reference
7      private ScriptDomain domain = null;
8
9      // Called by Unity
10     void Start()
11     {
12         // Should we initialize the compiler?
13         bool initCompiler = true;
14
15         // Create the domain
16         domain = ScriptDomain.CreateDomain("MyDomain",
initCompiler);
17     }
```

   The domain is created using the static 'CreateDomain' method. There are a number of arguments you can provide to this method but the default options will be suitable for most applications.

3. Compile / Load External Code

Once you have a domain, you are now ready to load or compile any external C# code or assemblies. There are a number of methods that you can use for loading assemblies or C# code. For a basic example, we have defined the C# code we want to load as a string named 'source'.

C# Code

```
using UnityEngine;
using RoslynCSharp;

public class Example : MonoBehaviour
{
    private ScriptDomain domain = null;

    // The C# source code we will load
    private string source =
        "using UnityEngine;" +
        "class Test : MonoBehaviour" +
        "{" +
        "    void SayHello()" +
        "    {" +
        "        Debug.Log(\"Hello World\");"    +
        "    }"                                  +
        "}";

    void Start()
    {
        // Create the domain - We are using C# code so we need the
compiler
        domain = ScriptDomain.CreateDomain("MyTestDomain", true);

        // Compile and load the source code
        ScriptType type = domain. CompileAndLoadMainSource' (source);
    }
}
```

The main load method here is the call to 'CompileAndLoadMainSource'. This method will invoke the Roslyn compiler which generates a managed assembly. The main type for that assembly (in this case 'Test') is then automatically selected and returned as a Script Type. There are many more load and compile methods that you can use. Take a look at the included scripting reference for an overview of the available API's

For more information on loading C# code and assemblies take a look at 'LoadingAssemblies' and 'LoadingScripts' section.

4. Create an instance of the type

Once you have a Script Type loaded, the next thing you will want to do is create an instance of that type (Assuming that the type is not static). There are a number of methods that allow you to do this but for this example we will use the basic 'CreateInstance' method of Script Type.

Previous code has been omitted to keep the examples short

C# Code

```
1  using UnityEngine;
2  using RoslynCSharp;
3
4  public class Example : MonoBehaviour
5  {
6      void Start()
7      {
8          // Compile and load the source code
9          ScriptType type =
   domain.CompileAndLoadScriptSource(source);
10
11         // We need to pass a game object because 'Test' inherits
   from MonoBehaviour
12         ScriptProxy proxy = type.CreateInstance(gameObject);
13     }
14 }
15
```

At this point you now have an external C# script attached to a game object as a component. All of the expected mono behaviour events will be called such as 'Start' and 'Update' and you can interact with the Unity API from the external script.

5. Call a Method

This next step is a simple example of how you are able to call custom methods to provide a similar event system as a mono behaviour. For example, let's say we want to call the 'SayHello' method when a script is constructed.

Previous code has been omitted to keep the examples short

C# Code

```
using UnityEngine;
using RoslynCSharp;

public class Example : MonoBehaviour
{
    void Start()
    {
        // Compile and load the source code
        ScriptType type =
domain.CompileAndLoadScriptSource(source);

        // We need to pass a game object because 'Test' inherits
from MonoBehaviour
        ScriptProxy proxy = type.CreateInstance(gameObject);

        // Call the 'SayHello' method
        proxy.Call("SayHello");
    }
}
```

We use the method 'Call' in this example but there is another method called ' SafeCall' which might be more appropriate in this scenario. The 'SafeCall' method will catch any exceptions throw by the target method which could prove useful when dealing with external code.

For more information about cross communication between external code and the application, take a look at the 'Interface Approaches' section.

6. **Congratulations, you have just compiled and loaded a basic C# script at runtime. Now you can go on to do awesome things!**

Escape The Maze Example

Roslyn C# includes an example programming based game to demonstrate some of the features and uses of the asset. The example game can be found by loading the scene at 'Assets/RoslynCSharp/Examples/EscapeTheMazeExample.unity'. The objective of the game is to write the decision-making code for a maze crawling mouse to reach the exit of the maze.

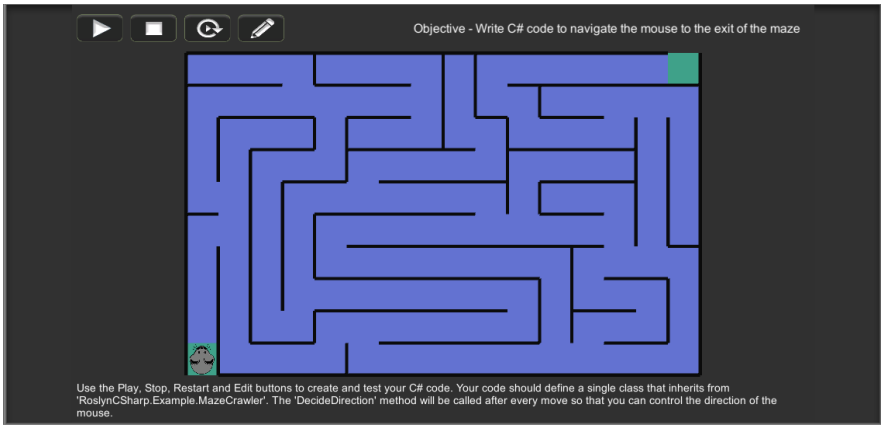


Figure 1

The game buttons shown in the top left of the game window are described below from left to right:

1. **Run Code Button** – Press this code to compile and run the code you created in the code editor.
2. **Stop Code Button** – Press this button to stop the mouse crawler and reset the maze
3. **Restart Code button** – Press this button to restart the maze and run the same code again
4. **Edit Code Button** – Press this button to open the in-game code editor

The code editor window is show below:

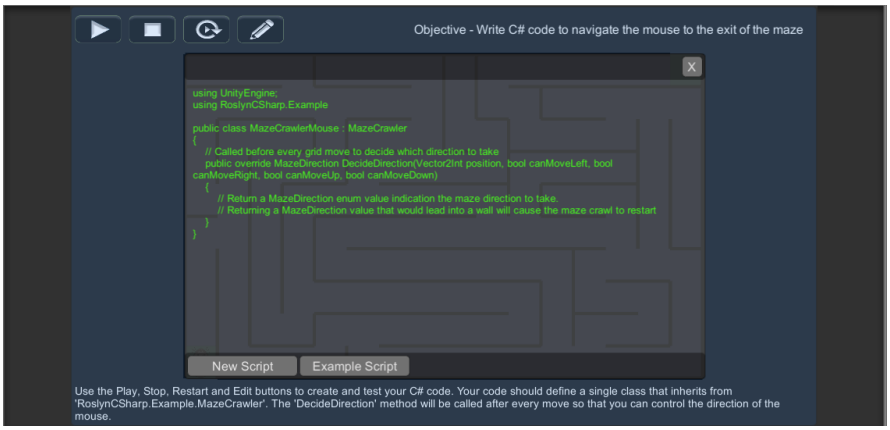


Figure 2

The code editor buttons shown at the bottom left of the code editor window are described below:

1. **New Script** – Press this button to load the blank template script
2. **Example Script** – Press this button to load the solution code for the maze crawler

Game Rules

- The maze is grid based and before every move you must specify the direction that the mouse should move given the current situation.
- Directing the mouse into a wall will cause the game to finish and reset.
- The mouse will drop a breadcrumb after every move so you can see which routes have been explored.
- There Is no limit to the amount of time that it takes for the mouse to exit the maze.
- The maze never changes but hardcoding the solution is frowned upon 😊

How it Works

As mentioned previously you will need to write the code that determines which direction the mouse will take before each move. To write this code you will use the in-game code editor which can be accessed by clicking the 'edit' icon. The code editor will load a template script which contains a basic class definition and an override method that you should provide the body for as shown below:

C# Code

```
1  using UnityEngine;
2  using RoslynCSharp.Example
3
4  public class MazeCrawlerMouse : MazeCrawler
5  {
6      // Called before every grid move to decide which direction to
7  take
8      public override MazeDirection DecideDirection(Vector2Int
9  position, bool canMoveLeft, bool canMoveRight, bool canMoveUp, bool
10 canMoveDown)
11     {
12         // Return a MazeDirection enum value indication the maze
13 direction to take.
14         // Returning a MazeDirection value that would lead into a
15 wall will cause the maze crawl to restart
16     }
}
```

This method will be called before every move around the maze and the state information for the mouse crawler will be passed including the current index position. From this state information you must choose the appropriate move for the mouse to take which is specified by returning a 'MazeDirection' enum value.

The same class instance will be used throughout the game so you are able to store state information such as visited indexes at class scope and their values will be persisted thorough the game. There is also an included solution to the game that will guide the mouse crawler successfully to the exit every time which can be loaded in the code editor window.

Can you escape the maze?

Plugin Conflicts

2019.3 or newer

In some cases, you may receive compilation errors on import about one or more missing assembly references, preventing the asset from working correctly. This is a rare occurrence but should be easily fixable in most cases. To resolve the problem:

1. Navigate to the folder at 'Assets/RoslynCSharp/Resources' and find the unity package asset named 'System.ValueTuple.unitypackage'.
2. Double click on this asset or 'Right Click -> Open' to import the package into your project.
3. At the import package window, click the 'Import' button.
4. Allow Unity to recolime scripts and see that the error no longer occurs.

If the problem still persists, then please contact us for further support. Contact information can be found at the end of this document.

2019.0 – 2019.2.x

If you use install packages from the package manager into your project or you use newer versions of Unity you may see one or more errors referring to conflicting assemblies which will stop the project for working properly. Here is an example of such as error:

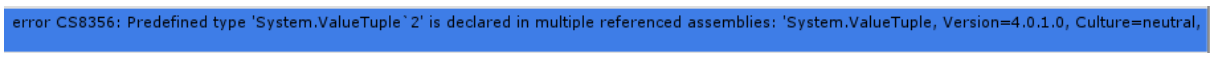


Figure 3

In order to fix this error you should first identify and locate the trouble assembly in the Roslyn CSharp asset folder. In this case, the conflicting assembly is named 'System.ValueTuple' as per the error message and it can be found inside the Roslyn CSharp plugins folder at 'Assets/RoslynCSharp/Plugins/System.ValueTuple.dll'.

There are 2 options to resolve the conflict:

1. Select the plugin asset in the Unity editor and in the inspector window disable all platforms so that the assembly is no longer in use.

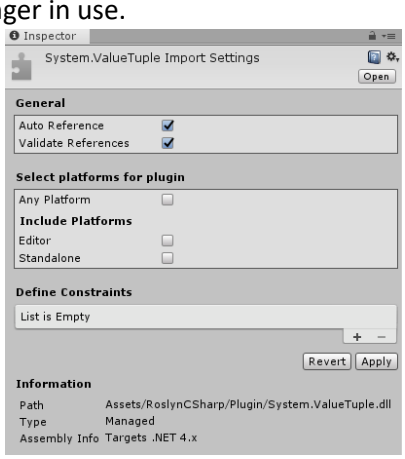


Figure 4

2. Delete the plugin asset from the project. This should fix the conflict issue and can be reverted by reimporting the asset via the asset store.

Repeat these steps for any other conflicting assemblies and you should now be able to use Roslyn C# as normal. If you are still having difficulties then please contact us for help. Contact information can be found at the end of this document.

If the problem still persists, then please contact us for further support. Contact information can be found at the end of this document.

Concepts

Script Domain

Roslyn C# uses the concept of Script Domains which you can think of as a container for any externally loaded code. It's main purpose is to separate any game or runtime code that gets loaded automatically by Unity when your game starts so that when calling any 'Find' methods it will only return externally loaded code. You must create a Script Domain before you are able to load any external code and all subsequent loading will be handled by that domain.

If you are particularly adept in C# then you may be familiar with 'AppDomains'. It is worth noting that Roslyn C# does not use a separate AppDomain for external scripts by default but can do if necessary. If you do need to use a separate app domain then you can create and pass that app domain when creating a script domain and it will then be used for all external code. You will however need to manually manage that domain by loading core libraries and resolving references as a result of load requests.

As well as acting as a container, a Script Domain is also responsible for the loading and compiling 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 or 'Compile' 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.

> **Note:** *Any assemblies or scripts that are loaded into a Script Domain at runtime will remain until the application ends (Unless a separate app domain is passed when creating a script domain). 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 (Equivalent of using the ‘new’ operator) 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.

A Script Proxy implements the IDisposable interface which handles the destruction of the instance automatically based upon its type.

- For instances inheriting from MonoBehaviour, the Script Proxy will call ‘Destroy’ on the instance to remove the component
- For instances inheriting from ScriptableObject, the Script Proxy will call ‘Destroy’ on the instance to destroy the data.
- For instances that implement the ‘IDisposable’ interface, the script proxy will call ‘Dispose’ on the wrapped instance.
- 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.*

Roslyn C# Settings

Roslyn C# uses a number of global settings that can be modified within the Unity editor by opening the following menu item 'Tools -> Roslyn C# -> Settings'. This will cause the Roslyn C# project settings to be loaded and displayed in the settings window:

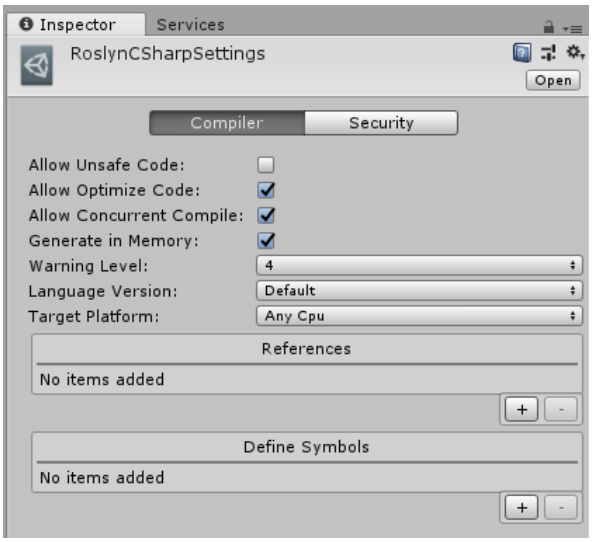


Figure 5

There are two main tabs in the settings window to categorize the settings. This following section will cover all the options in each tab and what they do.

Compiler

The compiler settings tab contains a number of settings relating to the Roslyn compiler:

- **Allow Unsafe Code:** When enabled, the Roslyn compiler will allow unsafe code to be compiled.
- **Allow Optimize Code:** When enabled the Roslyn compiler will optimize the compiled code.
- **Allow Concurrent Compile:** When enabled the Roslyn compiler may use multiple threads to compile the code. Disabling this value will force the compiler to use a single thread.
- **Generate In Memory:** When enabled the Roslyn compiler will not write the compiled assembly to file. It is recommended that you leave this option enabled for best compatibility across platforms because writing to file requires IO access in certain locations.
- **Warning Level:** The compiler warning level from 0 – 4.
- **Language Version:** The C# language version that should be supported. You can use this option to limit the C# language features that external code can make use of.
- **Target Platform:** The platform that the compiled code will target. It is recommended that you leave this option at 'Any CPU'.
- **References:** A list of reference names or paths the compiled code will reference. For example: 'UnityEngine.dll'.
- **Define Symbols:** A list of scripting define symbols that external code will be compiled with. For example: 'UNITY_EDITOR'.

References

The references list box can be used to add and remove assembly references. Click the '+' button to add a new entry and a dialog input window will allow you to enter the assembly name. Assembly names should be the name of the assembly file with extension. For example, 'UnityEngine.dll'.
To remove an existing reference, click on the item in the list and see that it is now highlighted. Click the '-' button to remove the active selection.

Define Symbols
The define symbols list allows you add compiler define symbols that can be used to activate or deactivate code features using pre-processor directives. For example, If you want to compile unity code for use ad edit time, you might add the define symbol 'UNITY_EDITOR'.
A new define symbol can be added by clicking the '+' button and entering the desired define symbol name in the resulting input dialog.
An existing item can be removed by clicking on the item so that it becomes highlighted, and then clicking the '-' button to remove the active selection.

Security

The security tab contains all settings related to code security and validation and is where you can setup restrictions to ensure that external code does not access undesirable API's such as System.IO.

- **Security Check Code:** When enabled, Roslyn C# will attempt to validate all code before it is loaded into the script domain. If the security checks fail then the code will not be loaded. It is highly recommended that this option remains enabled as you may not have any control over the external code being loaded.
- **Illegal References:** The assembly reference restrictions list allows you to add any number of reference restrictions which will be checked when external code is loaded. A reference restriction is simply the name of an assembly that must not be referenced by loaded code. A good candidate for a restriction would be 'UnityEditor.dll' as it is not available at runtime.
- **Illegal Namespaces:** The assembly namespace restrictions list allows you to specify individual namespaces that should not be accessed by external code. Simply specify the full namespace name and any loaded code that references that namespace will fail verification causing the code to be discarded. A good candidate for namespace restriction would be 'System.IO' to prevent access to the file system. It is also highly recommended that 'System.Reflection' is added to the restrictions because it could potentially be used as a workaround for already restricted namespaces. You are also able to use wildcards to exclude all child namespaces by using the format 'System.IO.*'.
- **Illegal Types:** The type references that are not allowed to be used by external code. This is useful for specifying certain types without excluding an entire namespace.

Hot Reloading

The hot reloading tab contains all features relating to the runtime hot reloading feature for C# scripts. This feature allows you to make changes to C# source files while in play mode, and Rolsyn C# will automatically compile, load and execute the modified script when saved. See the **Hot Reloading** section for more information.

- **Enable Hot Reloading:** The hot reloading feature can be fully disabled if it is not needed for your project. Disabling this option may give a slight improvement on the enter play-mode time.

- **Copy Serialized Fields:** Should all public and serialized fields of the script be replaced with the original values when hot reloading occurs. This is recommended to keep maintain the state of such variables during a hot reload.
- **Copy Non-Serialized Fields:** Should all other fields that are not serializable be replaced with the original values when hot reloading occurs. This is recommended to maintain the internal state of the script during hot reloading so that execution can continue based on the same variable data. With this option disabled, any member fields will be reset to their default or initializer values.
- **Destroy Original Script:** When this option is enabled, the original script that is being replaced as part of the hot reload will be removed completely from the scene. It is recommended to keep the scene well organised and uncluttered.
- **Disable Original Script:** When this option is enabled, the original script that is being replaced as part of the hot reload will remain in the scene, but will be disabled. This means that events such as 'Update' will not be called, but the script will still exist in the scene.

Assembly References

Compiling C# source files can be a powerful feature, but almost all external code you want to compile will need to work with other types found in external assemblies in order to do anything useful. This is where assembly referencing come in.

You will need to add the necessary references to the compiler before you attempt to compile source files, otherwise you may receive compilation errors regarding missing types or namespaces. Roslyn C# aim to make this process as simple as possible but there are still a few ways that you can add a reference:

Setting References

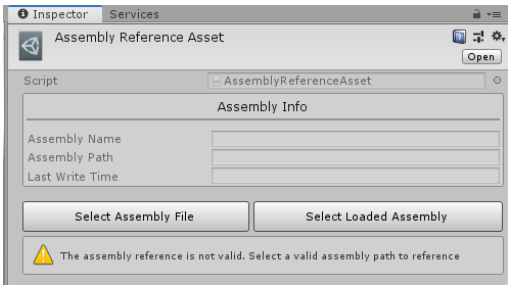
You can add global references via the Roslyn C# settings window. Open the settings window by going to 'Tools -> Roslyn C# -> Settings' and find the list box named 'References'. Clicking the '+' button to add a new reference will prompt you to enter the name or path of an assembly to reference. This assembly reference will then be added to all 'ScriptDomain' instances that are created at runtime.

Assembly Reference Asset

As of version 1.3.x, Roslyn C# now supports assembly reference assets which are now the preferred way to add an assembly reference. An assembly reference is simply a Unity asset file which contains referencing information about a particular assembly. You will need to create 1 assembly reference asset for each assembly file that you need to reference.

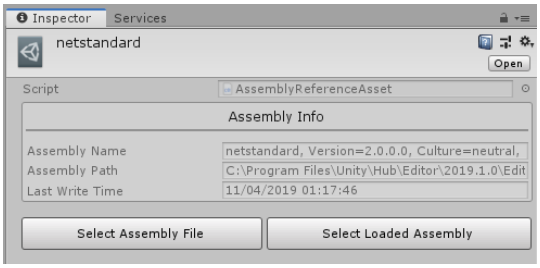
In order to create a new assembly reference asset, simply right click at a suitable location in the project window and select 'Create -> Roslyn C# -> Assembly Reference Asset'. This will create a new asset in the active folder and will prompt you to enter a name. We recommend that you name the asset after the assembly you intend to reference for clarity. For example: if you want to reference 'netstandard.dll', then a suitable assembly reference asset name might be 'NetStandard'. This is purely for the sake of clarity and the name can be anything you like.

Once you have created a new assembly reference asset, you will notice that it will have an invalid configuration by default:



Use one of the 'Select Assembly' buttons in order to load a valid assembly reference configuration.

- **Select Assembly File:** Click this button to browse for an assembly file to reference. This will open a file browser dialog when you can select a suitable assembly. Accepting the dialog with a valid assembly file selected will cause the assembly reference asset configuration to be generated and you will notice that the 'Assembly Info' fields are now populated.
- **Select Loaded Assembly:** Click this button to browse though all currently loaded assemblies in the Unity engine app domain. The assembly list will be presented as a sorted context list where you can select the assembly you require, if it is in fact loaded.



Note: Assembly reference assets store the assembly image data as an embedded asset meaning that the filepath may not be used at runtime if it cannot be found. In this case, the reference will use the embedded data for referencing purposes meaning that platforms like Android can be supported since the data is no longer external. Any time a referenced assembly file is changed, the assembly reference asset will auto update its embedded image version so that the reference data remains current.

Once you have setup a valid assembly reference asset, you can then pass it to the compiler in one of 2 ways. At startup, you can add it the compiler references list for a particular script domain. The following example code shown how this can be achieved:

C# Code

```
1  using UnityEngine;
2  using RoslynCSharp;
3
4  public class Example : MonoBehaviour
5  {
6      // Assign in inspector
7      public AssemblyReferenceAsset referenceAsset;
8
9      void Start()
10     {
11         ScriptDomain domain = ScriptDomain.CreateDomain("Example");
12
13         domain.RoslynCompilerService.ReferenceAssemblies.Add(
   referenceAsset);
14     }
15 }
16
```

Alternatively, you can pass in one or more assembly reference assets when you compile source code as shown in this example:

C# Code

```
using UnityEngine;
using RoslynCSharp;

public class Example : MonoBehaviour
{
    string sourceCode = ...;

    // Assign in inspector
    public AssemblyReferenceAsset referenceAsset;

    void Start()
    {
        ScriptDomain domain = ScriptDomain.CreateDomain("Example");

        domain.CompileAndLoadMainSource(sourceCode,
ScriptSecurityMode.UseSettings, new IMetadataReferenceProvider[]{
referenceAsset });
    }
}
```

Hot Reloading

As of version 1.6.2, Roslyn C# now supports runtime hot reloading of C# scripts by utilizing the compiler in the editor. The hot reload system is intended for quick iteration/testing of script changes while in play mode. The hot reloading feature makes this possible by observing all C# source files for any changes or modifications while in the mode. When a change is detected, the compiler is triggered to compile and load the modified script into memory. The script will then go though the 'Mod Script Replacer' pipeline to replace the original component, along with any field values (depending upon settings). At the end of this stage, the original script will have been replaced with the new 'Hot reloaded' script which will continue execution using the same script state (serialized and non-serialized field values).

This system works well for decoupled scripts and can improve iteration time massively (Only modified scripts will be recompiled). There are some limitations with this feature though:

Limitations

- Scripts that reference other user components may fail to compile at the moment. The hot reloading feature will not compile all scripts in the project as a batch at the moment for improved reload time. This can result in errors if the script has dependencies on other user scripts. We will be looking to fix this in a future version.
- Other components in the scene that hold references to a script that you want to hot reload will become null. We have not found any solution to this problem at the moment so it is why we only recommend the feature for well decoupled components/standalone behaviours.
- The feature may not perform as expected if you are using assembly definition files. The compile is not setup to scan asmdef files for references at the moment, which may result in compilation errors when the hot reload feature attempts to recompile source files.

Auto Refresh

To get the full speed benefits of the auto reload feature, you will need to disable the Unity auto refresh feature. By default, Unity will detect script changes when you modify and save a source file, and it will then perform a complete runtime teardown, compilation and initialization sequence which can take a lot of time, especially on larger projects. You can disable auto-refresh to eliminate this stage and have the full benefits of hot reloading.

To disable auto refresh, go to 'Edit -> Preferences' and disable the option named 'Auto Refresh' under the 'General' section.

Script Changes While Playing

If disabling auto-refresh is not an option for your project, then you can also use the 'Script Changes While Playing' option that is available in the same preferences window. Simply set this option to 'Recompile After Finished Playing' to prevent scripts from being reloaded during play mode. This will mean that assets and scripts will auto refresh as normal while outside of play mode, but you will still get the improved reload speed while in play mode.

Events

It may be useful to know when your script is going to be hot reloaded. You can implement the 'RoslynCSharp.Modding.IModScriptReplacedReceiver' interface to receive a callback just before a script is about to be hot reloaded. The replacement script instance will be passed into this method so that you can perform any necessary initialization or setup.

Note that this event will run after the Unity 'Awake' event due to the design of the Unity events system.

Loading Assemblies

It is possible to load external code as compiled assemblies using Roslyn C#. 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 must be created in order to use Roslyn C#.

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 where 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.

Loading Scripts

One of the main features of Roslyn C# is that it allows C# source code to be compiled at runtime meaning that you can easily add features such as modding support without relying on slower interpreted languages such as Lua. Since the code is compiled before use, you will have the same performance as if the script was included in the game in the first place.

There are a number of useful methods that you can use. The following methods can compile and load C# source code at runtime:

- **CompileAndLoadScriptFile(string):** Attempts to compile the C# source code in the specified file and load the resulting Script Type into the Script Domain.
- **CompileAndLoadScriptFiles(params string[]):** Attempts to compile all of the specified C# source files as a batch and loads the resulting assembly into the Script Domain.
- **CompileAndLoadScriptSource(string):** Attempts to compile the C# source code specified in the string argument and load the resulting Script Type into the Script Domain.
- **CompileAndLoadScriptSources(params string[]):** Attempts to compile all of the specified C# source code and loads the resulting Script Assembly into the Script Domain.

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

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

There are 2 main types of communication that you can use to interact with external scripts and assemblies:

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, Roslyn C# includes a basic but flexible generic communication system that works using reflection and allows any class member to be accessed without knowing the runtime type.

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 RoslynCSharp;
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     }
}
```

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. Roslyn C# 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 as

compiled scripts are set to reference the UnityEngine.dll by default. The target method must also accept the same argument list otherwise calling the method will fail.

Roslyn C# 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

```
using UnityEngine;
using RoslynCSharp;

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 writ 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 fields 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

```
using UnityEngine;
using RoslynCSharp;

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 type 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 external code, however it is possible (although not advised) to create these base classes directly within your Unity project and allow the external code to reference the 'Assembly-CSharp' assembly containing all of your game code. The reason this method is not recommended is that it might allow external code to use your game scripts as well as the base classes which may be undesirable.

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.

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'.

In order for runtime compiled code to access this assembly we will need to add it to the assembly references in the setting windows. See the 'Roslyn C# Settings' section for information on how to do this.

Once the assembly has been added to the references then we are ready to compile and load our external code. We will now require our external code to implement this interface in order for us to load it into the game. If it does not then we will treat it as if there are no valid defined types. Our external code is simply as follows:

C# Code

```
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.

The next step it to compile and load the code into a Script Assembly using one of the many load method of our Script Domain. As in the previous section, we will store the C# code from the above example directly in a string in order to keep the example simple but you can load the source code from any location you need. The following code will compile and load the source into a Script Domain:

C# Code

```
using UnityEngine;

class Example : MonoBehaviour
{
    private string source =
        "using UnityEngine;"                +
        "public class Test : IExampleBase"  +
        "{"                                 +
        "    void SayHello()"               +
        "    {"                             +
        "        Debug.Log("Hello");"       +
        "    }"                             +
        "    void SayGoodbye()"             +
        "    {"                             +
        "        Debug.Log("Goodbye");"     +
        "    }"                             +
        "}";

    void Start()
    {
        // Create the domain
        ScriptDomain domain =
ScriptDomain.CreateDomain("TestDomain", true);

        // Compile the source code
        ScriptAssembly assembly =
domain.CompileAndLoadScriptSources(source);
    }
}
```

At this point, we now have our source code compiled and loaded into our Script Doman and the next step is to make use of our interface and find all types that inherit from it. The Script Assembly class contains a number of useful methods for accessing Script Types that inherit from other types. The following code shows how we can access all types that inherit from our 'IExampleBase' interface that we created earlier.

Previous code has been omitted to keep the examples short

C# Code

```
using UnityEngine;

class Example : MonoBehaviour
{
    void Start()
    {
        // Compile the source code
        ScriptAssembly assembly =
domain.CompileAndLoadScriptSources(source);

        // Find all types that implement the 'IExampleBase'
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.

Previous code has been omitted to keep the examples short

C# Code

```
using UnityEngine;

class Example : MonoBehaviour
{
    void Start()
    {
        // Find all types that implement the 'IExampleBase'
interface
        ScriptType[] types =
assembly.FindAllSubtypesOf<IExampleBase>();

        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 you the type we need to create an instance of it which is done using the ‘CreateRawInstance’ of the Script Type. The main different this method has when compared with the ‘CreateInstance’ methods 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.

Broadcast Messages

As of version 1.6, Roslyn C# now includes the domain broadcast feature which is a useful 1-way communication method with external scripts. Much like the Unity 'BroadcastMessage' method, this feature is intended to call a method with the specified name for all script components or instances that are currently executing in a ScriptDomain. This can be useful if you wanted to create your own magic event methods similar to Unity's 'Start' or 'Update' methods.

The broadcasting feature that Roslyn C# offers is however much more versatile since it can support static methods, instance methods for both mono behaviour and non-mono behaviour instances, and is not limited to game object hierarchies. You can in fact broadcast to a particular scene or even all scenes if required in order to filter out behaviour components.

Static Broadcasts

Static broadcasts will as the name suggests, attempt to invoke all static methods on all types loaded into a ScriptDomain with a matching method name. Any number of arguments of any type are also supported, although the argument list must match exactly otherwise the broadcast call will fail silently. Here is an example of a static broadcast:

C# Code

```
1  using UnityEngine;
2
3  class Example : MonoBehaviour
4  {
5      void Start()
6      {
7          ScriptDomain domain =
8  ScriptDomain.CreateDomain("Example");
9
10         // Compile and load some code that defines a static method
11 named 'SayHello'
12         domain.CompileAndLoadSource("class Test { static void
13 SayHello() { Debug.Log("Hello"); } });
14
15         // Send the broadcast
16         domain.StaticBroadcast("SayHello");
17     }
}
```

You can also pass any number of arguments as an 'object[]' using the overload method if you want to pass data when invoking the method.

Broadcast Behaviour Instances

Using a similar API, you are also able to broadcast to behaviour instances, or script components that derive from 'MonoBehaviour' if you prefer. The only real difference is that you will need to decide if you want any receiver filtering using scenes. You can pass a Unity scene to the broadcast method to only invoke the method on behaviours that exist inside that scene. You can also broadcast to all scenes if you do not need any filtering.

There is also one other filtering mechanism in place for instance methods. You can opt to provide a 'System.Type' object as the 'baseType' arguments using one of the overloads. This type argument is

used to only invoke the broadcast method on instances that derive from the type. As a result, this type should always be a subtype of 'MonoBehaviour' otherwise the broadcast will do nothing.

C# Code

```
using UnityEngine;

class BroadcastBase : MonoBehaviour {}

class Example : MonoBehaviour
{
    void Start()
    {
        ScriptDomain domain =
ScriptDomain.CreateDomain("Example");

        // Compile and load some code that defines a static method
named 'SayHello'
        ScriptType type = domain.CompileAndLoadMainSource("class
Test : BroadcastBase { void SayHello() { Debug.Log("Hello"); } });

        // An instance needs to be created to receive broadcasts
        type.CreateInstance(gameObject);

        // Send the broadcast to the active scene
        domain.BroadcastActiveScene("SayHello");

        // Send the broadcast to all scenes
        domain.BroadcastAllScenes("SayHello");

        // Send the broadcast to the active scene - Instance must
derive from 'BroadcastBase'
        domain.BroadcastActiveScene(typeof(BroadcastBase),
"SayHello");
    }
}
```

Broadcast Non-Behaviour Instances

This is much the same as broadcasting to behaviour instances, although there is no filtering by scene, since normal C# objects are not bound to any scene. Instead, you will just use the same 'baseType' argument as the only filtering option. You can elect to broadcast to all non-behaviour instances by passing 'typeof(object)' as the base, since all managed objects derive from this type. It should also be noted that only instances constructed using 'ScriptType.CreateInstance' can receive broadcasts. Instances created from external code using the 'new' operator will not receive broadcasts, even if they do define a suitable broadcast receiver method.

C# Code

```
class BroadcastBase {}

class Example : MonoBehaviour
{
    void Start()
    {
        ScriptDomain domain =
ScriptDomain.CreateDomain("Example");

        // Compile and load some code that defines a static method
named 'SayHello'
        ScriptType type = domain.CompileAndLoadMainSource("class
Test : BroadcastBase { void SayHello() { Debug.Log("Hello"); } });

        // An instance needs to be created to receive broadcasts
        type.CreateInstance(gameObject);

        // Send the broadcast to all instances
        domain.BroadcastInstance(typeof(object), "SayHello");

        // Send the broadcast to all instances - Instance must
derive from 'BroadcastBase'
        domain.BroadcastInstance(typeof(BroadcastBase),
"SayHello");
    }
}
```

Android

Android platforms are not officially supported. However, a number of users have been able to get the asset to work well on Android platforms, and were kind enough to report to us the steps they took to achieve this. The following section will list some crucial steps to get the asset up and running on Android:

Requirements

The most important factor for Android platforms is that Assembly Reference Assets must be used (Requires version 1.4.0 and newer). This is because assembly reference assets allow you to reference assemblies when compiling code without requiring file paths, which can be an issue on Android. Roslyn C# includes a number of Assembly Reference Assets that you can use which can be found at 'Assets/RoslynCSharp/AssemblyReferences/'. You can also create your own reference assets for any other assembly you may like to use.

Configuration

There is a little bit of setup required for Android platforms. First, open the Roslyn C# settings window by going to 'Tools -> Roslyn C# -> Settings' and preforming the following:

1. Enable the option named 'Generate In Memory' if it is not already. This will cause the compiler to emit assemblies in memory streams as opposed to file streams and is quite important.
2. Remove All default compiler references that are displayed in the 'References' collection. These assembly references are auto-added to the compiler via file path only, which can cause issues on Android. If you need to reference any of these assemblies in your external code, then you can add an equivalent Assembly Reference Asset. Take a look at the 'Assembly References' section for information on how to create a reference asset, or alternatively, you can use an already existing reference asset if one exists.
3. Remove All In-code reference expressions such as:

C# Code

```
1  AssemblyReference.FromNameOrFile(...)
2  AssemblyReference.FromAssembly(...)
```

These reference expressions will attempt to reference assemblies via file paths, which can be an issue. The following expressions are OK since they do not use file paths:

C# Code

```
1  AssemblyReference.FromStream(...) // Cannot be FileStream
2  AssemblyReference.FromImage(...)
```

Hopefully by following this mini guide, you can get the asset running on your target Android platform.

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

Roslyn C# was designed as a complete runtime coding 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/