User Guide

Roslyn C# 2.0

Next Generation Runtime C# compiler

Trivial Interactive

Code Samples

Version 2.1.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 desktop platform using Mono backend by default for code execution. IL2CPP and any platform support is possible using dotnow add-on (Free with MIT License on github).
- Async compilation is not supported on WebGL due to lack of threading support.
- Scripts must be compiled before they can be executed.

Features

- Compile and Load C# scripts at runtime
- Use the latest C# language features
- Fast execution - Once compiled, external code will run as fast as game code
- Simple and easy to use API for assembly, type and instance reflection
- Support for non-concrete communication using script proxies
- Automatic type construction using correct method (AddComponent, CreateInstance, new)
- Code security validation means that unsafe code can be identified and discarded
- Code security generates a detailed report upon failure containing information about illegal types used and every usage occurrence in the external code
- Execution security ensures that loaded code cannot crash the host game with long or infinite loops
- Fully commented C# source code included
- Comprehensive documentation of the API for quick and easy reference

Contents

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 and additionally via code samples hosted on GitHub.

1. **Install package**

   Open the project that you want to install Roslyn C# into (Unity 6.0.0 and onwards are officially supported but other versions may work) and install the package as you would do normally from package manager.

> **Note:** *The first time you enter play mode or trigger a domain reload, Roslyn C# will create a user settings asset in the project containing all Roslyn C# settings that you can modify as required. A message will be logged in the console when this occurs, but this is a subsequent auto-setup step which is worth mentioning.*

2. **Create a Domain**

   The first thing you will need to do to interact with Roslyn C# is create a Script Domain where all your external code will be loaded into. This is done from code, and a script domain is simply a container where all loaded code is stored and executed, plus it also provides access to the many available compile API's that you can use from your game. The following C# code shows how a domain can be created using the constructor:

```
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.        // Create the domain
13.        domain = new ScriptDomain();
14.    }
15. }
16.
```

> **Note:** *Usually, it is common practice to have 1 script domain object that is created on startup and exists for the duration of the game. For this reason, it is a good idea to store the domain in a field so that it can be conveniently accessed from other methods when needed.*

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'.

```
 1. using UnityEngine;
 2. using RoslynCSharp;
 3.
 4. public class Example : MonoBehaviour
 5. {
 6.     private ScriptDomain domain = null;
 7.
 8.     // The C# source code we will load
 9.     private string source =
10.   "using UnityEngine;" +
11.         "class Test : MonoBehaviour" +
12.         "{" +
13.         "   void SayHello()" +
14.         "   {" +
15.         "       Debug.Log(\"Hello World\");"    +
16.         "   }"                                  +
17.         "}";
18.
19.     void Start()
20.     {
21.         // Compile and load the source code
22.         ScriptType type = domain.CompileAndLoadMainSource(source);
23.     }
24. }
25.
```

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 as well as async variants for all to run the compilation in the background. Take a look at the included scripting reference for an overview of the available API's

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.

> **Note:** *Since the ScriptType (`Test` in this case) is derived from MonoBehaviour, we will need to pass in a GameObject as part of the CreateInstance call. Internally Roslyn C# will detect that the type is a MonoBehaviour and will then use `AddComponent` rather than a normal C# constructor.*

Previous code has been omitted to keep the examples short

```
 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

```
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.Methods.Call("SayHello");
    }
}

```

Note: *The `SayHello` method in this case is marked as private (due to no explicit public, protected or internal keyword). Notice though that we can still call this private method from the game with no issues, in a similar way that Unity can call the `Start` or other methods even if they are private.*

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

What Next?

Now that you are up and running with the quickest of samples, take a look at these following topics to learn more about Roslyn C# and how you can use it in your project:

- **Demo**: Setup and play around with the included programming base demo game.
- **Concepts**: Read more about the concepts used by Roslyn C# such as script domains and proxies to get a better understanding of what they are for and how they can be used.
- Roslyn C# Settings: Check out the user settings that are available in Roslyn C# that can be changed to suit your needs.
- Code Samples (Github): Want to learn more about coding with Roslyn C#. The various code samples are the best place to start and show how many uses are achieved.

Escape The Maze Demo

This section covers the included `Escape the Maze` demo game that showcases some features of Roslyn C#.

> **Dependencies:** *Before continuing it should be noted that the included demo games have dependencies to Universal Render Pipeline and UGUI. Make sure those packages are installed beforehand to run the demo without issue. The core asset itself does not have any external dependencies.*

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 2.0/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.

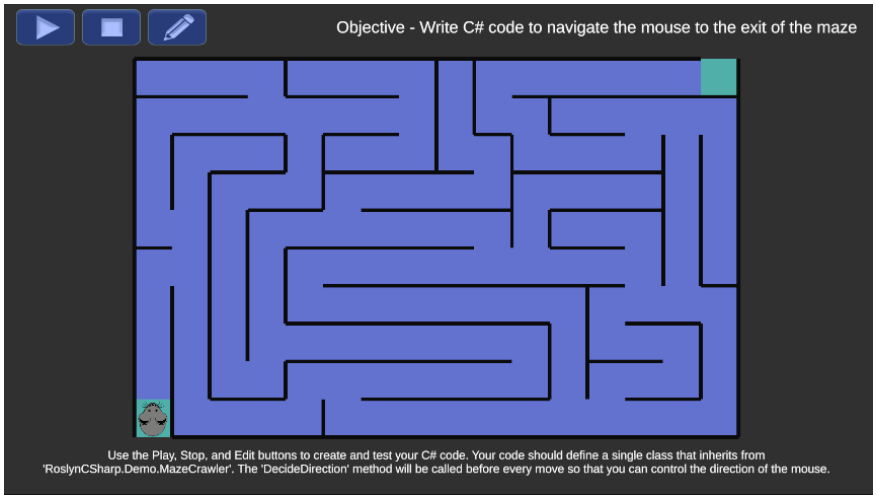


Figure 1

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. **Edit Code Button** – Press this button to open the in-game code editor

The code editor window is show below:

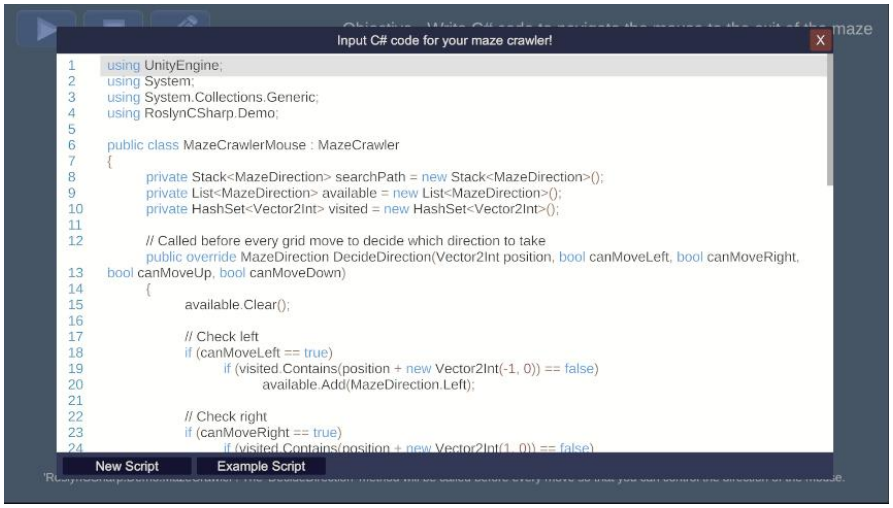


Figure 2

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 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:

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

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.

Running The Demo

To run the maze demo in editor, you can simply hit the `Run` button in the top left of the UI to start the maze crawler. By default, the game will load with solution code already applied so that the demo can be run without further steps, but of course you can make changes to that code and hit the `Run` button again to try your modified code.

Running in a standalone build is the same but you will need to add the 'EscapeTheMazeExample.unity' scene to the editor build settings in order for it to be available in a standalone build. The demo will run on any desktop mono platform in default form.

Can you escape the maze?

Concepts

This section will cover some of the concepts used by Roslyn C#. It is not essential to know about all of these concepts, but it will help better understand how the asset works and is intended to be used.

Script Domain

Roslyn C# uses the concept of Script Domains which you can think of as a container for any externally loaded code. Its 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 as from extensive testing it appears that Unity/Mono is not able to work nicely with additional app domains other than the default game domain.

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 criterion. 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 selected by a predefined set of rules by default, but you can change that behaviour if required. For more info about main type selection check out this sample.

If you need more control of the assembly, then you can use the 'SystemAssembly' 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 end. 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 or Component, 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.

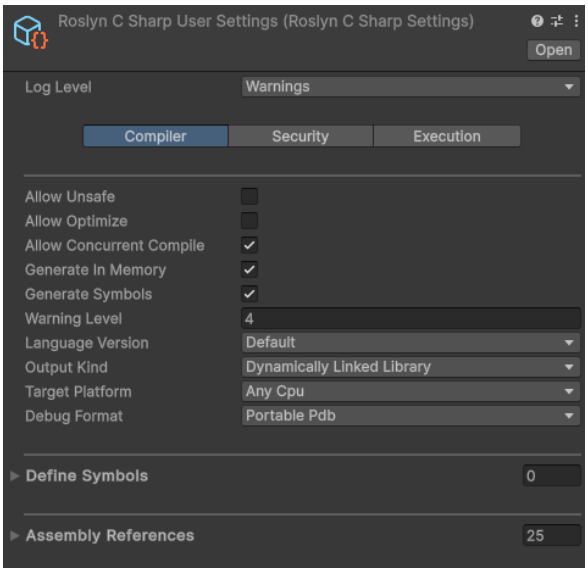
A Script Proxy implements the IDisposable interface which handles the destruction of the instance automatically based upon its type.

- 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 user settings that can be modified within the Unity editor by opening the following menu item 'Tools -> Roslyn C# 2.0 -> Settings'. This will cause the Roslyn C# project settings to be loaded and displayed in the settings window:



The settings are mostly grouped into tabs for better organization and the following section will cover all the options in each tab and what they do.

- **LogLevel:** This option is always displayed and allows you to set the logging detail output by Roslyn in the Unity console window. It means you can easily hide messages from Roslyn C# that you are not interested in, and the default value is `Warnings`:
    - **None:** Nothing will be logged at all. Be careful as critical errors will also be hidden.
    - **Errors:** Only error or exception messages will be logged in the console.
    - **Warnings:** Errors, exceptions and warnings will be logged in the console.
    - **Messages:** Errors, exceptions, warnings and info message will be logged in the console.
    - **All:** Same as `Messages` but included additionally for clarity.

Compiler

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

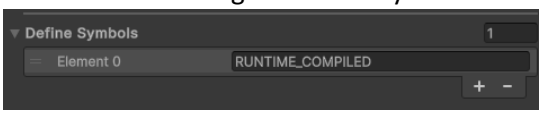
- **Allow Unsafe:** When enabled, the Roslyn compiler will allow unsafe code to be compiled.
- **Allow Optimize:** When enabled the Roslyn compiler will optimize the compiled code.
- **Allow Concurrent Compile:** When enabled the Roslyn compiler may use multiple threads to compile the code. Disabling 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.
- **Generate Symbols:** Should debug symbols be produced with the output when compiling.
- **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.

- **Output Kind:** The kind of output to produce, usually either .exe or the default of .dll.
- **Target Platform:** The platform that the compiled code will target. It is recommended that you leave this option at 'Any CPU'.
- **Debug Format:** The type of debug symbols to emit if `Generate Symbols` is enabled. For the most part Unity can work with portable pdb files.

- **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'.

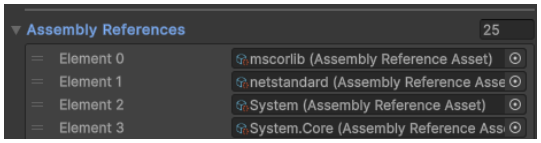
Define Symbols

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 at edit time, you might add the define symbol 'UNITY_EDITOR'.
A new define symbols can be added in the same way as any other Unity list item, simply by clicking the `+` button to add a new slot and then entering the define symbol text in the element input field.



Assembly References

The assembly references list can be used to add and remove assembly reference assets that are used by default (Any compile request that does not pass in a custom CompileOptions object will use the references added here).
The elements added to this list are of type `AssemblyReferenceAsset`, which are scriptable objects that can be created in the project window. You may notice though that Roslyn C# includes a number of pre-setup reference assets for common references you might use for things like the C# runtime and Unity modules. These existing reference assets can be found inside the Roslyn C# project folder: `RoslynCSharp 2.0/References/`.
A new reference asset can be added in the same way as any other Unity object list item, simply by clicking the `+` button to add a new slot and either dragging an asset into the slot or using the asset browser to select one.



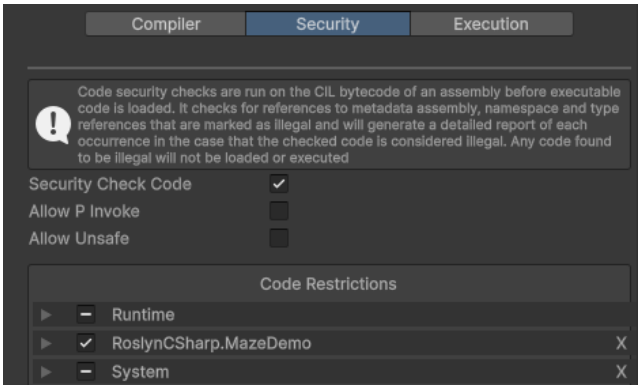
Security

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.

The code security feature in Roslyn C# is used to check all loaded code for potentially harmful or unsafe code that could potentially harm the device. This is useful if you are compiling code that has come from external sources in the case of modding, or for simply disallowing certain code for any reason.

This system was redesigned from the ground up for version 2.0 and works by scanning the CIL metadata and bytecode contained within the compiled .dll file (Runs after successful compilation) to check for any references that are considered illegal, as defined by the security settings that you can setup. If any illegal reference is found to an assembly, namespace, type or other member, then the code will not be allowed to load in an executable state, and a detailed code security report will be generated indicating where the problem is so that the user can address the issues and retry.

This system is enabled by default with suitable starting restrictions setup that allow access to common and safe runtime and Unity Api's. Unsafe or untrustworthy api's such as System.IO, System.Reflection etc are blacklisted by default.

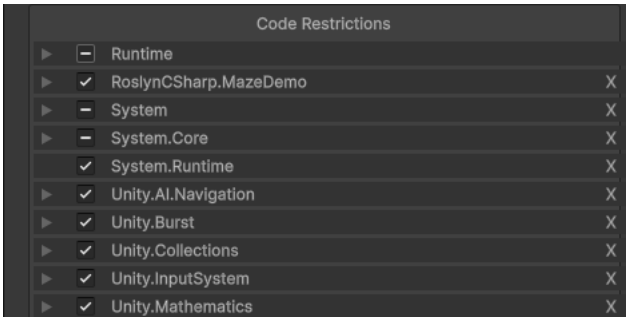


- **Security Check Code:** When enabled, Roslyn C# will attempt to validate all code before it is loaded into the script domain. If the security checks fail, then the code will not be loaded. It is highly recommended that this option remains enabled as you may not have any control over the external code being loaded.
- **Allow PInvoke:** Should platform invoke calls be allowed to call into native code. This is disabled by default and not recommended as there is no way to security check any native code that may be called via a PInvoke.
- **Allow Unsafe:** Should unsafe code conventions such as pointers and unsafe memory management be allowed. Not recommended in most cases as unsafe code can allow for memory access violations or access to data via pointers that could crash or the host game in some cases.

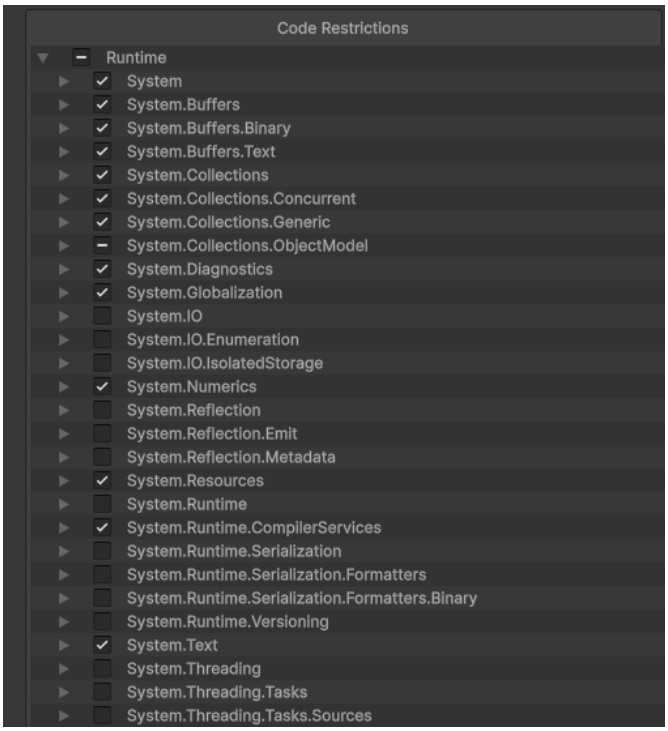
Code Restrictions

The code restrictions section is where you can specify which API's are allows to be used by compiled code, and which should not be allowed (Considered whitelisted or blacklisted respectively). The system is an opt-in approach only meaning that any assembly and associated members not explicitly added to this section will be considered illegal and will cause code security checks to fail if referenced.

You can see below that a number of assemblies are wholly or partially whitelisted by default, and this allows compiled code to access safe runtime API's as well as common Unity api's that might be required to interact with the game.

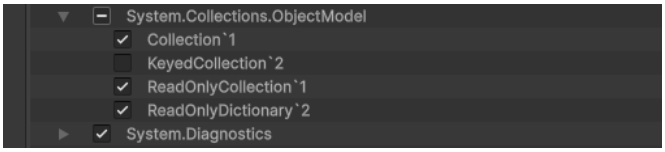


These top-level entries represent specific assemblies which have been added, and of course you can add your own assemblies if required as shown below. We can expand each assembly in the list to reveal the tree structure of members defined in the assembly.



As you can see here, the `Runtime` assembly is partially whitelisted as denoted by the `-` mark in the checkbox next to the assembly's name. With the tree view expanded we can see that certain namespaces are whitelisted such as `System` and `System.Numerics` as denote by the tick in the checkbox, and other namespaces which are considered unsafe such as `System.IO` are blacklisted as denoted by an unticked checkbox.

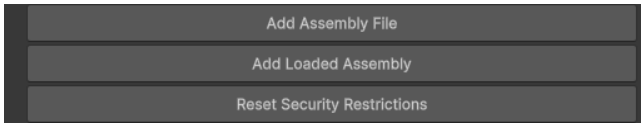
In this particular case we can also see that `System.Collections.ObjectModel` is also partially whitelisted, meaning that one or more child node is blacklisted. On expanding further, we can indeed see that `KeyedCollection`2' is blacklisted causing the mixed result.



The point is that you can easily spot nodes which are fully whitelisted by a tick, partially whitelisted nodes by `-`, and blacklisted nodes by an unchecked box.

Add or Reset Restriction

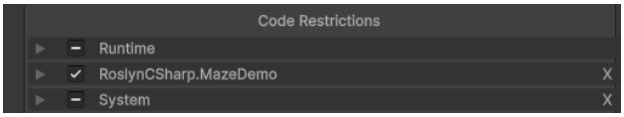
As mentioned previously, you can freely modify these restrictions as required to suit your game, whether you want that to be more restrictive or less restrictive. Towards the bottom of the inspector immediately after the code restrictions list, you will find the following buttons.



- **Add Assembly File:** This will show an open file dialog where you can select a .dll file from disk. Once a valid assembly path is selected, it will be scanned and added to the code restrictions list assuming that it is not already added.
- **Add Loaded Assembly:** This will bring up a context menu listing all assemblies that are currently loaded into the editor, and you can easily select an assembly to add from this list.
- **Reset Security Restrictions:** This will reset the code restrictions to default minimal values, containing only the runtime assembly which is required as a minimum

Remove Restriction

Removing an existing entry from the code restrictions can be done by clicking the 'X' icon to the right of the assembly's name. An important thing to note is that the `Runtime` node cannot be deleted entirely but only modified. This is because the runtime is always required from security checks to run properly, and depending upon the scripting backend in use, the `Runtime` will represent either `mscorlib` or `System.Private.CoreLib`.



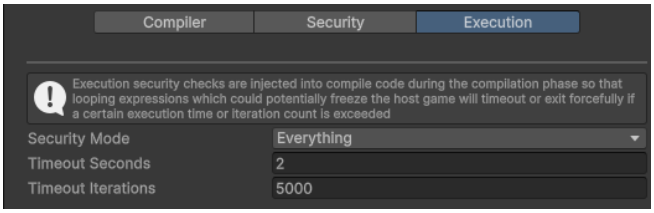
Execution

The execution tab is newly added for version 2.0 and covers all settings relating to code execution security. Code execution security is an additional dynamic security layer on top of the static code restrictions layer which comes into play once the compiled code is executed.

Roslyn C# will inject additional security checks into source code during the compile phase that can safely detect and handle execution issues which could cause problems with or crash the host game. This means that Roslyn C# will check for looping constructs in the compile code where there is potential for an infinite or very long loop to execute, which in the case of Unity could crash the host game, or at the very least freeze the game until it completes.

You may ask why not call such compiled code from a different thread to prevent this issue? That is a valid and suitable solution IF and only if that code does not need to access the Unity API is anyway, because of course it is unfortunately not possible to call any (Except a very small subset) Unity Api for a thread other than main.

For this reason, Roslyn C# will inject checks by default that can throw an exception to break out of a loop if a preset amount of time or number of iterations are exceeded.



- **Security Mode:** Determines which execution security checks are included by default.
    - **None:** Do not inject any execution security checks. Not recommended because the compiled code when executed will have the potential to crash the host game with an infinite loop.
    - **Execution Timeout:** This will allow time-based checks to be injected so that a loop can be forcefully aborted once it has been executing longer that the specified amount of time.
    - **Execution Iterations:** This will allow iteration counter checks to be injected so that a loop can be forcefully aborted if it executes more that the specified max iteration count.
    - **Everything:** Causes all of the above execution checks to be injected into compiled code.
- **Timeout Seconds:** The amount of time in seconds before a loop should be forcefully aborted, only applicable if security mode has the `Execution Timeout` option set.
- **Timeout Iterations:** The maximum number of iterations a loop can run before it should be forcefully aborted, only applicable if security mode has the `Execution Iterations` option set.

Assembly References

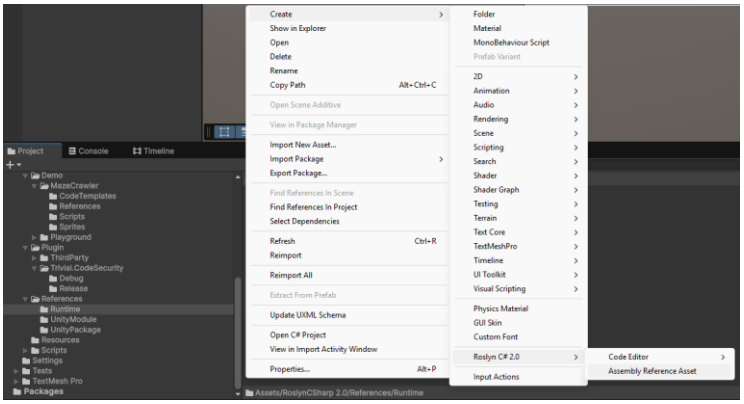
Compiling C# source files can be a powerful feature, but almost all external code you want to compile will need to work with other types found in external assemblies in order to do anything useful. This is where assembly referencing comes in.

Roslyn C# 2.0 aims to make referencing as simple as possible, and as a result there are already references setup for common runtime and Unity assemblies that you might like to use from compiled code. It means you can use runtime libraries for things like Linq and collections, as well as most of the Unity API you would expect without adding those references manually. This means you will only need to add a reference to assemblies that you add to the project yourself either as a .dll asset file or via Unity package manager, or also if you create code in your game that you want to be usable from compiled code.

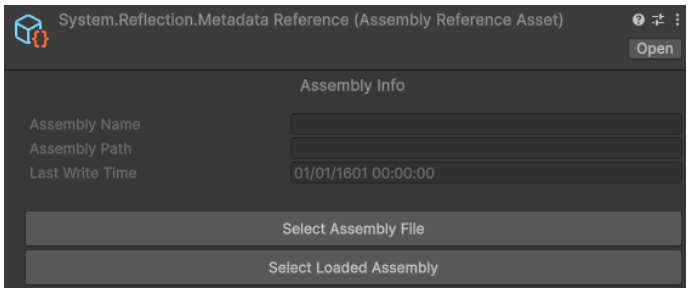
Assembly Reference Asset

In order to add a reference, you will need to first create an assembly reference asset. An assembly reference asset is just a regular unity scriptable object asset which stores metadata about a particular assembly which can be used by the compiler to create a reference, allowing any public API to be used from compiled code.

To create a new reference asset, simply right click in the project folder and select `Create -> Roslyn C# 2.0 -> Assembly Reference Asset`:



Once created you can give the asset a suitable name (Usually the same name as the assembly you want to reference for clarity) and then the inspector window should appear something like this:

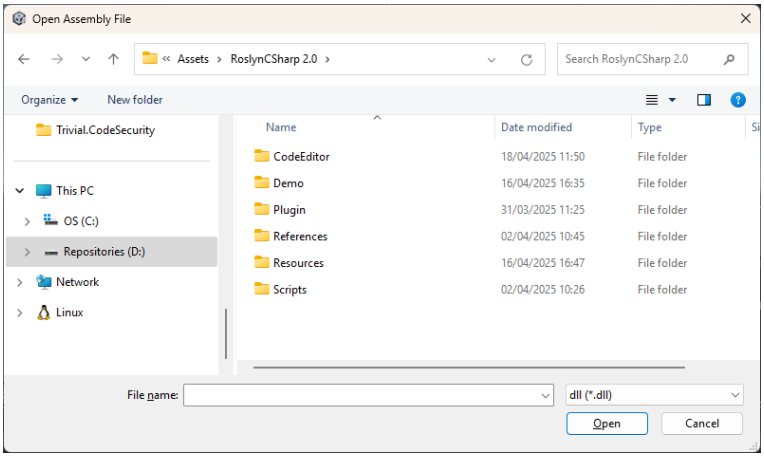


You can see here that we have an empty reference asset because the name, path and last write time values are all invalid. To make this reference valid we need to assign an assembly so that the necessary metadata can be loaded into the asset for referencing purposes.

Select Reference Assembly

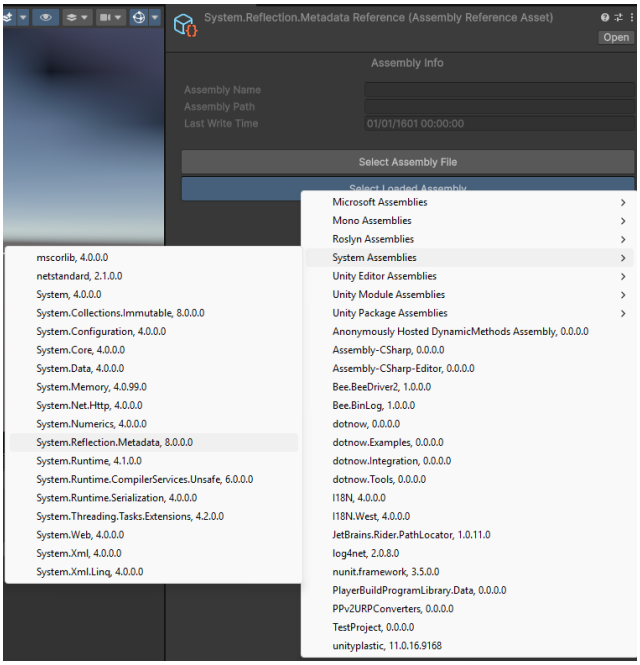
In order to make the assembly reference asset usable, we will need to assign an associated reference assembly to it, and there are 2 ways that this can be done:

1. **Select Assembly File:** First, we can select and assembly file (managed assembly with .dll extension) if the location is known. To do this simply click the `Select Assembly File` button to open a file selection dialog, find and select the correct assembly and choose open to attempt to load and initialize the reference asset.

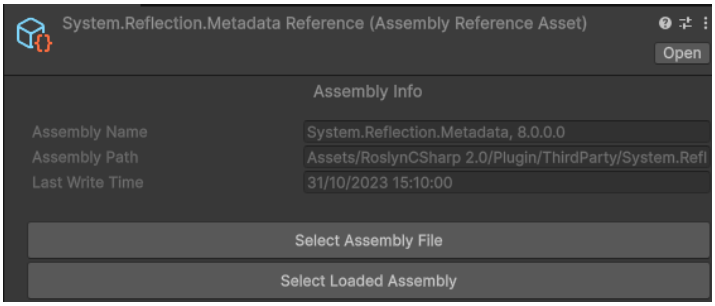


1. **Select Loaded Assembly:** This option is usually easier and quicker as it will show a list of all assemblies that the Unity editor has loaded into memory, and in most cases, you will be able to find the assembly you want to reference unless it has not been added to the project.

   For convenience, the assemblies are sorted by categories so you can find them easier, but in most cases any assembly that you have added to the project should appear in the root menu. To add a reference simply select one of the assemblies from the menu:

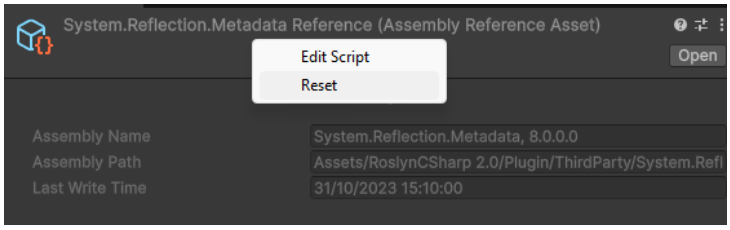


Once you have selected an assembly that you wish to reference, you should see that the assembly reference asset is now updated to reflect that, with valid assembly name, path and last write time values:



Reset Reference Assembly

If you need to clear the reference assembly and reset it back to default, then this can be done with the normal Unity `Reset` option which is available when right clicking on the asset header:



Alternatively, you can simply select an assembly again as shown in the previous section, and the current assembly information will be overwritten.

Auto Refresh

Assembly reference assets are designed to be able to auto-update when a change is detected to the source assembly. This process if fully automatic and the last write time of the assembly is used to detect changes that may have been made. If the source assembly has a more recent write time, then the assembly reference asset will update all metadata to ensure that the reference info s correct for the latest version. This is important to ensure that newly added or removed public API's can be used correctly without needing to recreate the reference asset at each occurrence, which may be the case if you are referencing one of your own asmdef's for example that is part of your game project code.

Reference From Code

In some cases, you may wish to add additional compiler references from code. While this is possible, it is not recommended in almost all cases, since it is generally less portable due to requiring file paths (Best to test that the reference works correctly in a standalone build).

To create a reference from code you can simply use the `AssemblySource` class which implements the `ICompilationReference` interface, allowing it to be passed as a compiler reference. To create an instance, it is simply a case of selecting and using a suitable helper method as shown below.

For more detailed code referencing samples, check out the online samples repo.

Path Reference

You can create a reference object for an assembly path using the following method. Note that the path can be relative or absolute, but must exist or the compiler will generate an error:

```
1. AssemblySource source = AssemblySource.FromFile(“My/Assembly/Path.dll”);
2.
```

Memory Reference

Alternatively, if you have an assembly image stored in memory as a byte array, it is possible to create a reference from this data. An assembly image in this case means the raw contents of a managed .dll file (If you used `File.ReadAllBytes` or similar), and not the actual loaded memory in the AppDomain, as there is no easy way to add a reference to an assembly in that case. You can use the following method in that case:

```
1. AssemblySource source = AssemblySource.FromMemory(assemblyBytes);
2.
```

Script Interfaces

Once you have loaded an assembly or compiled code into a Script Domain, the next step you will likely want to take is to communicate with the types in some way by calling methods or accessing members.

There are 3 main types of communication that you can use to interact with external scripts and assemblies which are interface communication, generic communication and compiled-to-game communication. Each approach has pros and cons, and in many cases, you may wish to combine 1 or more approaches to achieve your goal, but for the most part we would recommend interfaced based communication and compiled-to-game communication because they both ensures type safety and will be as performant as normal game code.

Here is a brief overview of each approach:

- **Interface Communication:** Allows game code to call and access compiled code.
  - + Ensures type safety and will cause compilation errors if Api's are incompatible
  - + Full JIT performance. Code runs as fast as game code.
  - - A bit more complex to understand and implement.
- **Compiled-To-Game Communication:** Allows compiled code to call and access game code.
  - + Ensures type safety and will cause compilation errors if Api's are incompatible
  - + Full JIT performance. Code runs as fast as game code.
  - + When combined with interface communication, allows for full 2-way type safe communication with compiled code.
  - - A bit more complex to understand and implement.
- **Generic Communication:** Provides an alternate means for game code to call and access compiled code.
  - + No need to implement interfaces or any other type to aid communication
  - – Communication requires using member names which should be known ahead of time or can be scanned via reflection.
  - – Using reflection internally means that performance is quite a bit slower per call/access than the interface approach.

Interface Communication

Interface communication provides a way for game code to call or access members that are defined in compiled code in a type safe and very performant way. It is a little bit more advanced than generic communication because it requires defining one or more base classes or interfaces in your game code which the compiled code can then implement. It essentially comes down to dependency injection, where the game provides an interface that can be called, and the compiled code provides the implementation in this case.

Define an interface

The first thing to do is define a suitable public interface as part of your game that can be called when you want to decide some game logic, or really at any point that suits your needs. In this case we will just create the simplest interface to display a greeting when the game loads.

With that in mind we can end up with something like this, but alternatively it could be an abstract or virtual class that can be inherited instead, whichever is more suited to your application:

```
1. public interface IGreeting
2. {
3.     string SayHello();
4. }
5.
```

Note: You can create any number of base classes or interfaces with more complex Api's if required. You can use any public type that is defined in your game assembly or is part of the Unity/runtime as parameters or return types. For example, it is possible to pass Vector3's, Transforms, even Action<int> etc.

Note: For a real-world example of this approach in action, check out the source code for the included `Escape the Maze` demo game, which uses this same approach for calling compiled code to get the direction that the mouse crawler should take for each move.

Add Reference

Now that we have some code to be used externally, we will now need to add an additional compiler reference to the Roslyn C# settings so that any code we compile can have access to this `IGreeting` API. For this example, we will assume that this code is defined within the standard `Assembly-CSharp` module, meaning that we did not create a separate assembly definition for this code. A separate assembly definition is indeed possible and may be preferable in some cases, and in such a case the steps are almost identical except for the assembly's name.

To add a reference, we will need to create a separate assembly reference asset in the project to represent this interface and then add it to the project settings. Both topics were covered in previous sections so will not be repeated here to keep this guide concise.

1. To create a new assembly reference asset for `Assembly-CSharp`, simply follow the guide here.
2. To open the Roslyn C# 2.0 user settings window, follow the instructions here.
3. To add the new assembly reference asset to the Roslyn C# settings, check the following section.

Compiled Implementation

With a compiler reference added, the next step is to compile and load the source code which will contain the implementation. For this example, is means compiling and loading some C# source code which implements the `IGreeting` interface to return a suitable greeting message:

```
using UnityEngine;
using RoslynCSharp;

void Example : MonoBehaviour
{
	string source = @"public class CustomGreeting : IGreeting
{
	string SayHello()
	{
		return ""Hello from compiled code!"";
	}
}";

	ScriptDomain domain;
	void Start()
	{
		// Create domain
		domain = new ScriptDomain();

		// Compile and load as assembly
		ScriptAssembly asm = domain.CompileAndLoadSource(source);

		// Now find the type which implents `IGreeting` - `CustomGreeting` in this case
		ScriptType customGreetingType = asm.FindSubTypeOf<IGreeting>();
	}
}

```

Call Interface

Finally, once we have compiled, loaded and found a suitable ScriptType that implements the `IGreeting` interface, we are now ready to create an instance of this class and call its method via the interface. To do that we can make use of the `CreateInstanceAs` method which will return a new instance of a specified ScriptType as any generic type you provide.

```
using UnityEngine;
using RoslynCSharp;

void Example : MonoBehaviour
{
	void Start()
	{
		// Previous parts of the code were omitted
		// Now find the type which implents `IGreeting` - `CustomGreeting` in this case
		ScriptType customGreetingType = asm.FindSubTypeOf<IGreeting>();

		// Create instance as
		IGreeting greeting = customGreetingType.CreateInstanceAs<IGreeting>();

		// Finally we can get the greeting string which will call the compiled code, and just log it
		Debug.Log(greeting.SayHello());
	}
}

```

Compiled-To-Game Communication

Compiled-To-Game communication is similar to interface communication but provides a means for compiled code to access public members from the game code. Depending upon your game and what you intend to achieve, it is entirely possible that you may like to combine both this and interface communication to achieve 2-way communication between the game and compiled code. Similarly, this approach is also type safe and just as performant as your game code would be.

The idea behind this approach is that you create a public Api' for your game code which you will allow compiled code to use. This public Api can be part of your base game, or in more advanced cases you can create a separate Api specifically dedicated as an Api' that compiled code will use. This is usually done by creating a separate assembly definition in the Unity project where you will define code that should be used by the compiled code, and then if required this interface assembly can work directly with the game code.

Interface API

To keep this example simple, we will cover the first case where we will have a single game Api (Either use Assembly-CSharp as the interface or create a separate assembly definition), and we can simply control which aspects are available to compiled code by using C# access modifiers. Only members marked as public will be usable by compiled code.

Now we can take a look at a simple code example. In this case we have a simple agent class which represents a character that should be controllable by an external implementation, and the objective of the compiled code will be to control the movement of this agent.

```
 1. using UnityEngine;
 2.
 3. public class Agent : MonoBehaviour
 4. {
 5.       [SerializeField]
 6.       private float moveSpeed = 5;
 7.
 8.       public void Move(Vector3 amount)
 9.       {
10.                transform.position += amount * Time.deltaTime;
11.       }
12. }
13.
```

You can see from the code that we can choose to only expose certain Api's as public allowing them to be used from compile code (The `Move` method in this case), and we can keep the implementation hidden.

Add Reference

This step is the same as in the interface communication section, so can be skipped if that was already completed.

Implementation

With the assembly reference setup, we can now compile and load C# source code which uses this, Api. For this example, we will create an additional component which should be attached to the agent game object and should call the `Move` method any time the agent should be moved to a new position.

Here is a short example of that implementation and how it can be compiled, loaded and instantiated as a component on a game object.

```
 1. using UnityEngine;
 2. using RoslynCSharp;
 3.
 4. void Example : MonoBehaviour
 5. {
 6.       string source = @"using UnityEngine;
 7. public class AIAgent : MonoBehaviour
 8. {
 9.       Agent agent;
11.       void Start()
12.       {
13.                 agent = GetComponent<Agent>();
14.       }
15.       void Update()
16.       {
17.                 // Move to the right
18.                 agent.Move(new Vector3(0.1f, 0, 0));
19.       }
20. }";
21.
22.       ScriptDomain domain;
23.       public Agent agentPrefab;
24.       void Start()
25.       {
26.                 // Create domain
27.                 domain = new ScriptDomain();
28.
29.                 // Compile and load
30.                 ScriptType aiAgentType = domain.CompileAndLoadMainSource(source);
31.
32.                 // Create an instance of an agent
33.                 aiAgentType.CreateInstance(Object.Instantiate(agentPrefab));
34.       }
35. }
36.
```

Now when this code is run, it will instantiate an agent prefab which in this case is assumed to be a simple prefab of a cube or anything really with an `Agent` component added from the base game. Then it will add the compiled `AIAgent` component in addition which will automatically startup and will then simply move the agent to the right at a steady speed.

It is a very simple example but proves the concept that the compiled code is able to call game code with absolute type safety and full performance. Of course this can now be expanded with the same principal for virtually any use case to suit your game needs.

Generic Communication

Generic communication is an alternate way for game code to communicate with compiled code and 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, a bit similar to how Unity `Start` or `Update` methods are called by the engine, although technically not the same:

```
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.Methods.Call("OnScriptStart");
13.    }
14.
15.    void Update()
16.    {
17.        // Call the method 'OnScriptUpdate'
18.        proxy.Methods.Call("OnScriptUpdate");
19.    }
20. }
21.
```

The above code shows how a method with the specified name can be called at runtime using the ‘Call’ method. As you might expect, this will look for and attempt to call a method with the specified name, and it does not matter about the visibility of the method (It can be public, private, protected or internal and will still be found).

Of course, you can also pass arguments when calling a method by using the optional `args` parameters which is an array of objects representing the arguments to pass to the method. It is very similar to normal C# reflection in that regard.

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' provider of the proxy to read and write values as required.

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

```
 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.     {
13.         // Read the value of the field
14.         int old = (int)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 should be all the more careful. For example, trying to set a property which does not define a setter will result in an exception being thrown. The following code shows how a property called 'testProperty' can be accessed. The method is very similar with fields and properties.

```
 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.     {
13.         // Read the value of the property
14.         int old = (int)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. }
```

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 itself 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# 2.0 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/