CppInator

None

None

None

Table of contents

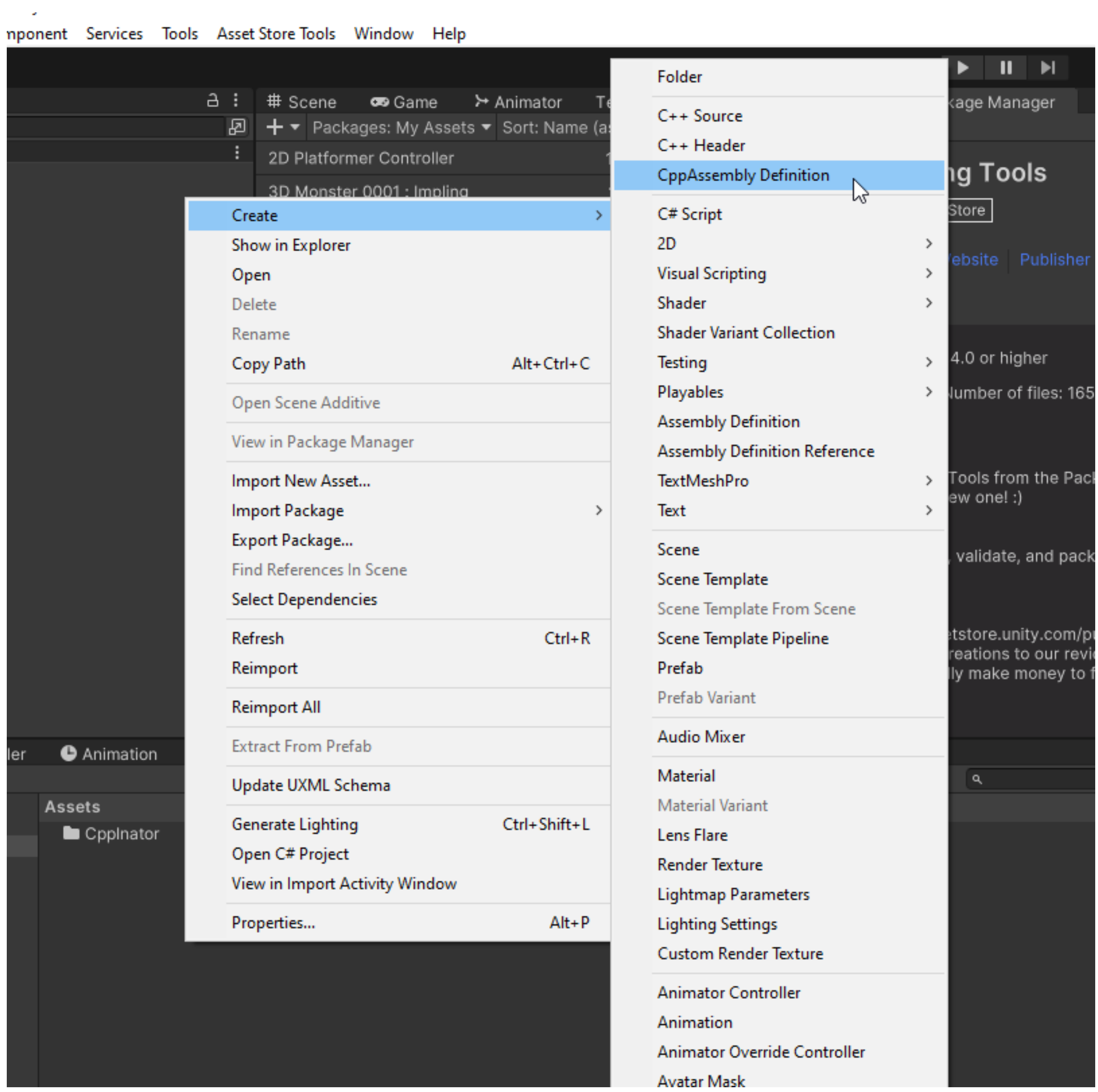
1. Getting Started

1.1 Installation

Download from the Asset Store.

1.2 Creating a C++ assembly asset

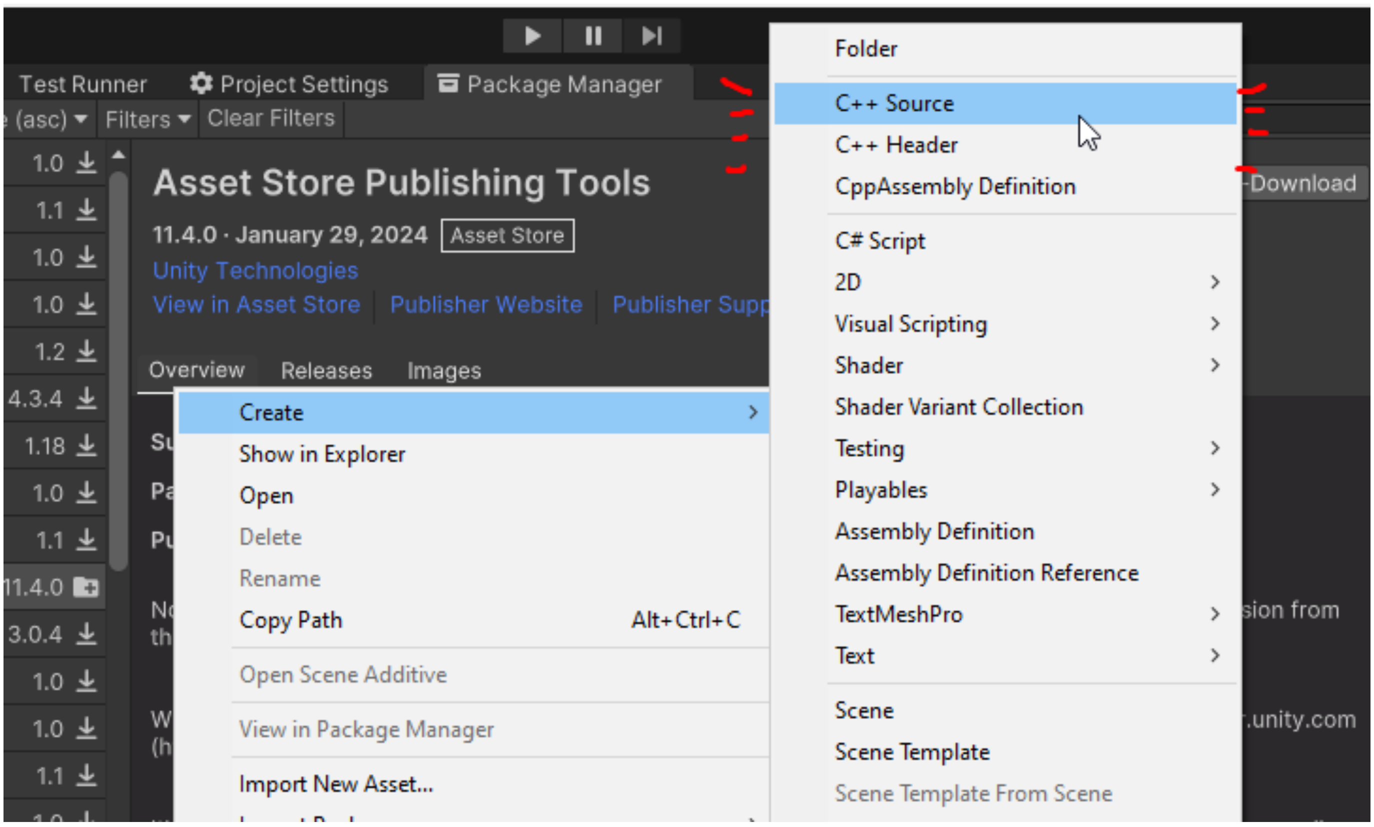
For CppInator to detect and compile your .cpp and .h files, you need to create a C++ Assembly asset. To do this, right-click in the Project window and select `Create > CppAssembly Definition`. This will create a new C++ Assembly asset in your project.



This is similar to a C# Assembly Definition, but it is used to define a C++ assembly. Any .cpp or .h files that are in the same directory as the C++ Assembly asset will be compiled into a shared library and linked to your Unity project.

1.3 Creating a C++ script

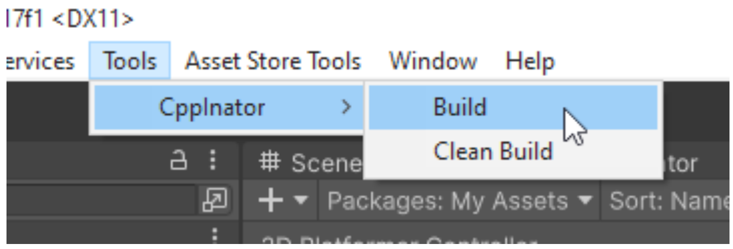
To create a new C++ script, right-click in the Project window and select `Create > C++ Source` or `Create > C++ Header`. This will create a new .cpp or .h file in your project.



You can then edit the .cpp and .h files in your favorite text editor or IDE. When you save the files, CppInator will automatically detect the changes and recompile the shared library.

Visual Studio Code is a great choice for editing C++ files, as it has built-in support for C++ syntax highlighting and IntelliSense. CppInator also configures Visual Studio Code to use the correct include paths and compiler options for your project.

1.4 Forcing a recompile or a clean rebuild of the shared library



1.5 Building for different platforms

You can build for any platform that supports IL2CPP, including Windows, macOS, iOS, and Android.

2. Calling a C++ function from C

In unity you would call the function just as you would for any other internal library using PInvoke. The only difference is that you need to use the `Native.Invoke` method to call the function.

Well, you don't NEED to but you risk crashing your editor if you don't. `Native.Invoke` will properly handle hot-reloading and other edge cases that can cause your editor to crash.

Here is an example of calling a C++ function from C#:

```
using CppInator.Runtime;

[DllImport("__Internal")]
static extern int PowerFunction(int value, int power);

void Start()
{
    int result = Native.Invoke(PowerFunction, 2, 3);
    Debug.Log(result); // Output: 8
}
```

And here is the C++ code that the C# function is calling:

```
#include <unityengine.h>
#include <cmath>

EXPORT(int) PowerFunction(int value, int power)
{
    return pow(value, power);
}
```

3. Calling a C# function from C++

This is a simple example of calling a C# function from C++.

First you need to send the function pointer from C#:

```
using CppInator.Runtime;

delegate void MyFancyCallback(int number);

[DllImport("__Internal")]
static extern void CallMeBaby(MyFancyCallback callback);

void Start()
{
    Native.Invoke(CallMeBaby, (MyFancyCallback)callback);
}

// MonoPInvokeCallback is required for the callback to work on iOS
[MonoPInvokeCallback(typeof(MyFancyCallback))]
void WhenCalled(int value)
{
    Debug.Log("Omg, her number is: " + value);
}
```

Then you need to call the function from C++:

```
#include <unityengine.h>

typedef void (*MyFancyCallback)(int);

EXPORT(void) CallMeBaby(MyFancyCallback callback)
{
    callback(8675309);
}
```

4. Logging inside C++ code

Here is an example of how to use `Debug.Log` in C++ code:

```
#include <unityengine.h>

void LogMessage()
{
    DebugLog("Hello from C++!");
    DebugWarning("This is a warning message.");
    DebugError("This is an error message.");
}
```

You can also use `Debug.Log` to log formatted messages using `std::to_string` or other string formatting functions. Here is an example of logging a formatted message in C++ code:

```
#include <unityengine.h>
#include <string>

void LogFormattedMessage()
{
    DebugLog("Hello from C++! " + std::to_string(42));
}
```

5. Threading in C++

CppInator includes support for the pthreads library, which provides a simple API for creating and managing threads in C++. To use pthreads in your C++ scripts, you need to include the `pthread.h` header file and link against the `pthread` library.

Windows is also supported.

Here is an example of creating a new thread in C++ using pthreads:

```
#include <pthread.h>

void* myThread(void* arg)
{
    pthread_exit(NULL);
}

int main()
{
    pthread_t thread;
    pthread_create(&thread, nullptr, myThread, nullptr);
    pthread_join(thread, nullptr);
    return 0;
}
```

You could for example callback a C# function from the new thread:

```
#include <unityengine.h>

typedef void (*MyFancyCallback)(int);


void* myThread(void* arg)
{
    CallMeBaby((MyFancyCallback)arg);
    pthread_exit(NULL);
}

EXPORT(void) StartAThreadMethod(MyFancyCallback callback)
{
    pthread_t thread;
    pthread_create(&thread, nullptr, myThread, (void*)callback);
}
```

Be careful when calling Unity API from a new thread, as Unity API is not thread-safe. You should only call Unity API from the main thread. To call a Unity API from a new thread, you can use the `UnityMainThreadDispatcher` class provided by CppInator.

```
void* myThread(void* arg)
{
    auto callback = (MyFancyCallback)arg;
    int calculateNumber = 0;

    for (int i = 0; i < 1000000; i++)
        calculateNumber += i;

    QueueToMainThread([]() {
        callback(calculateNumber);
    });

    pthread_exit(NULL);
}
```

QueueToMainThread is a function provided by CppInator that queues a lambda function to be executed on the main thread. This allows you to safely call Unity API from a new thread.