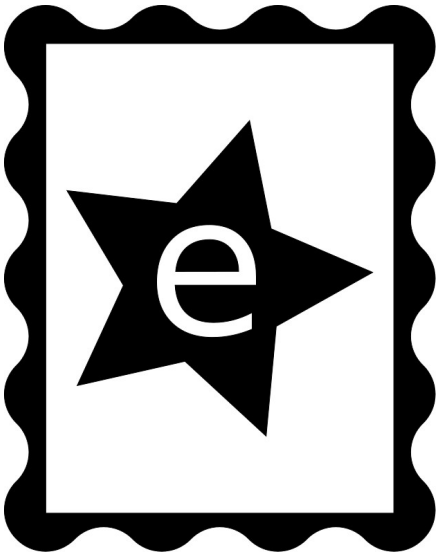


Make your own DLL from scratch (C++)

(eToile 2016)

Introduction

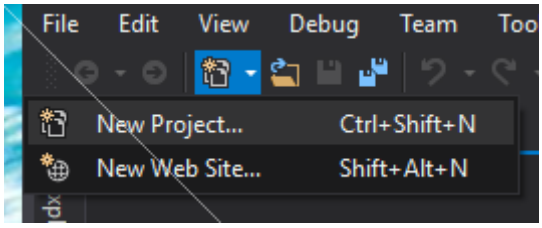
This package includes everything you need to know in order to deploy x86 and x64 DLLs for Unity integration. The included DLL project compiles under Visual Studio Community 2015, which can be downloaded for free from here:
https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx

You'll need to install the "C++ language" to be able to compile DLLs in Visual Studio.
In any case, eToile recommends to make a complete installation in order to avoid problems with the DLL example project and any further product development you may want to make.

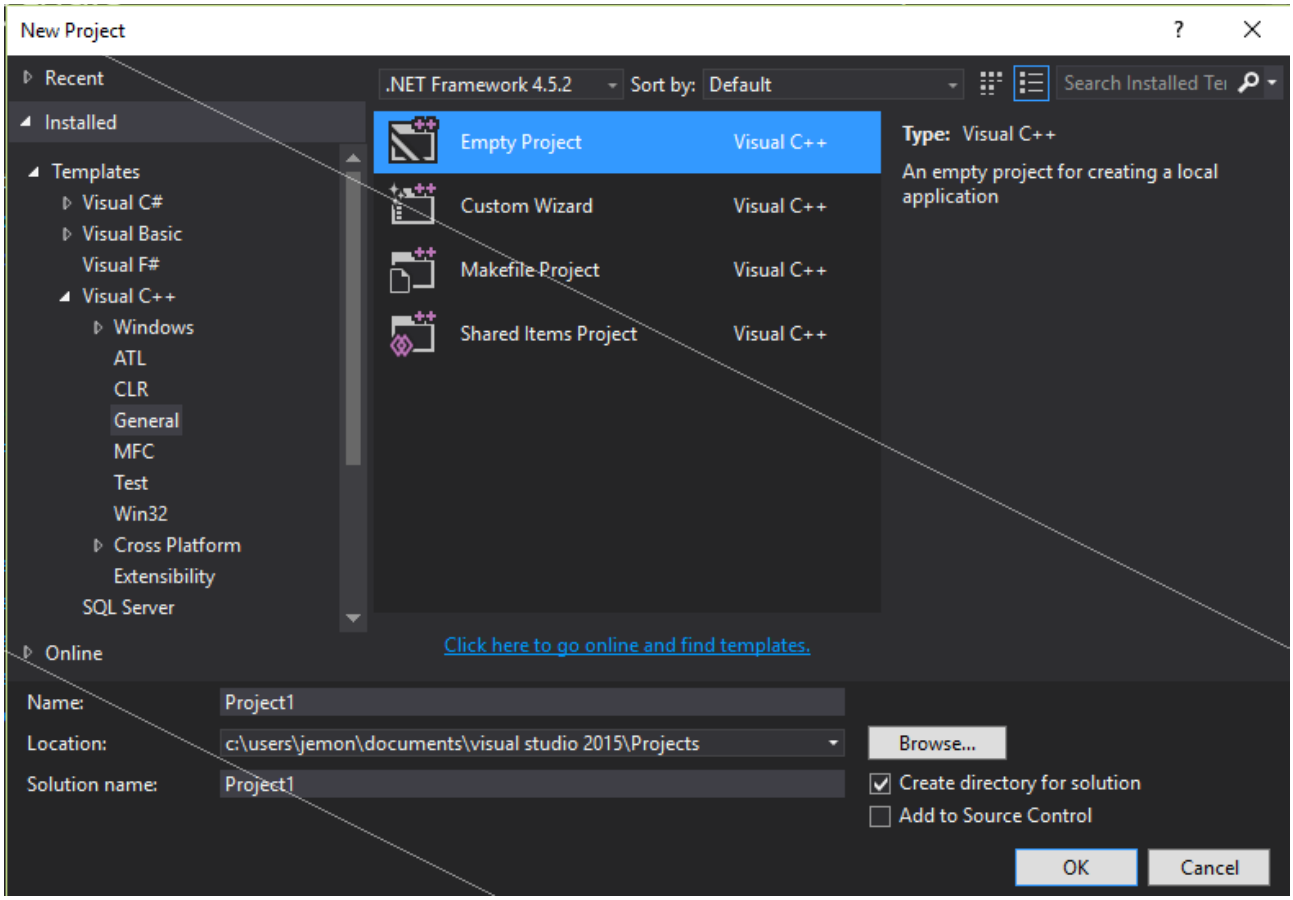
As you may expect, this DLL integration may also run on other development platforms because there is not any "Unity-Only" feature. This feature is not covered in this document.

Generating your C++ DLL from scratch

Assuming that you have already installed Unity (x86 or x64) and Visual Studio Community 2015, open Visual Studio and create a new project:



In the project creation window, you must select "Templates>Visual C++>General>Empty Project":



Enter the name you wish for your project in this window and click the "OK" button (this example will use the default project name Project1).

Once your project has been created, use the "Solution Explorer" window to create the source files for your dll as follows:

1. Right click in the "Header Files" folder and chose "Add > New Item…".
2. Select the "Visual C++ > Header File (.h)" option and enter a name for your file. This example will use the default name (Header.h). Click "Add" button.
3. Right click in the "Source Files" folder and chose "Add > New Item…".
4. Select the "Visual C++ > C++ File (.cpp)" option and enter a name for your file. This example will use the default name (Source.cpp). Click "Add" button.
5. Right click in the "Source Files" folder and chose "Add > New Item…".
6. Select the "Visual C++ > Code > Module-Definition File (.def)" option and enter a name for your file. This example will use the default name (Source.def). Click "Add" button.

Now open the ".cpp" file. Here is where the main code will be written. We will implement a simple interface to test. So, the ".cpp" file should look like this:

```
#include "Header.h"

// Function definitions:
extern "C"

int __stdcall GetInt(int n)
{
	return 2 + n;
}
```

Pay close attention to the `__stdcall` definition, this determines the function interpretation. Now our interface is defined.
You can also declare variables out the curly braces and that stored data will be persistent, so it can be retrieved through an interface at any time.

Note that the `Header.h` file must be included in order to compile correctly.

Open the ".h" file. This file should look like this:

```
#pragma once

// Custom function declarations goes here:
#ifdef __cplusplus
extern "C"
{
#endif
       // Just argument type of functions must be declared (no names):
       int __stdcall GetInt(int);

#ifdef __cplusplus
}
#endif
```

Note that arguments only include variable types, not variable names.
The `extern "C" {}` definition must contain every dll interface.

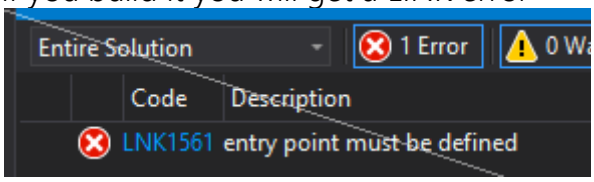
Open the ".def" file. This file declares the public interfaces, also called "EntryPoints".
This file should look like this:

```
; Source.def : Declares the module parameters for the DLL.

LIBRARY

EXPORTS
    ; Explicit exports can go here
       GetInt
```

Do not forget to include the `LIBRARY` statement, or Unity will throw EntryPoint exceptions when accessing the dll.

When you add the ".def" file to your project, Visual Studio adds the build reference automatically, if you need to modify it open the project properties dialog and check the "Configuration Properties > Linker > Input > Module Definition File" option.

The project is ready, but if you build it you will get a LINK error



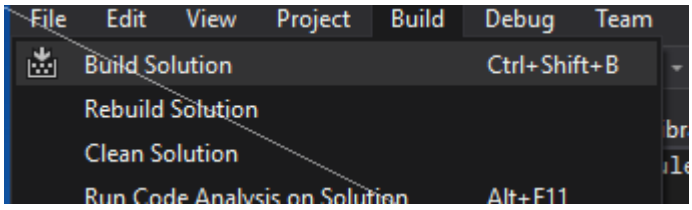
In order to fix this you have to set the type of project you are trying to compile, so open the project properties right clicking the project (Project1 in this example) in the "Solution Explorer" dialog and selecting "Properties".

At the top of the "Properties" dialog chose the "All Configurations" option in the "Configuration" drop down and the "Platform" drop down too.

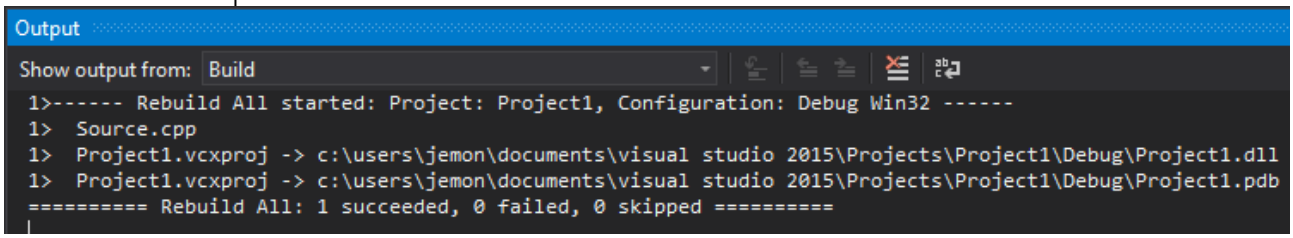
Set the option "Configuration Properties > General > Configuration Type" to "Dynamic Library (.dll)" instead of "Application (.exe)".
Then apply and close the Properties dialog.

Pay close attention that you need to set the "Configuration Type" to both processor types separately (x86 and x64), chose "All Configurations" and "All Platforms" at the top of this dialog.

Build the project and see if there are any errors. If you experiment issues check the tutorial up to this point.



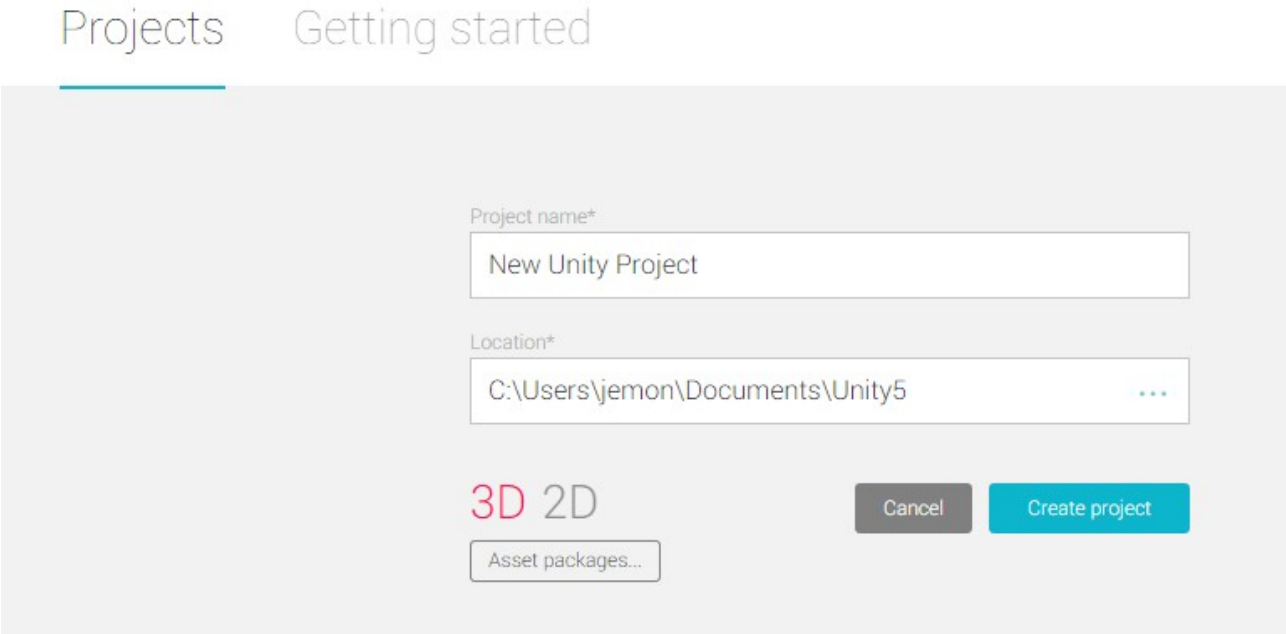
The correct compilation should look like this:



Here you can see the path of your generated dll file. The path depends on solution configuration and platform.
During development, it is recommended to use the "Debug" compilation, but once the code is running and needs to be shipped within the final product, the "Release" compilation must be used. The "Release" compilation has code optimizations and eliminates "Debug" dependencies.
Your DLL file will have the same name as your project, in this case: "Project1.dll".

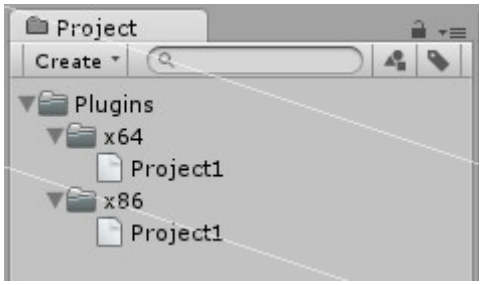
Creating the Unity project for integration

The next step is to create a Unity project and then add this DLL to it. To do so, open Unity and create a new empty project:



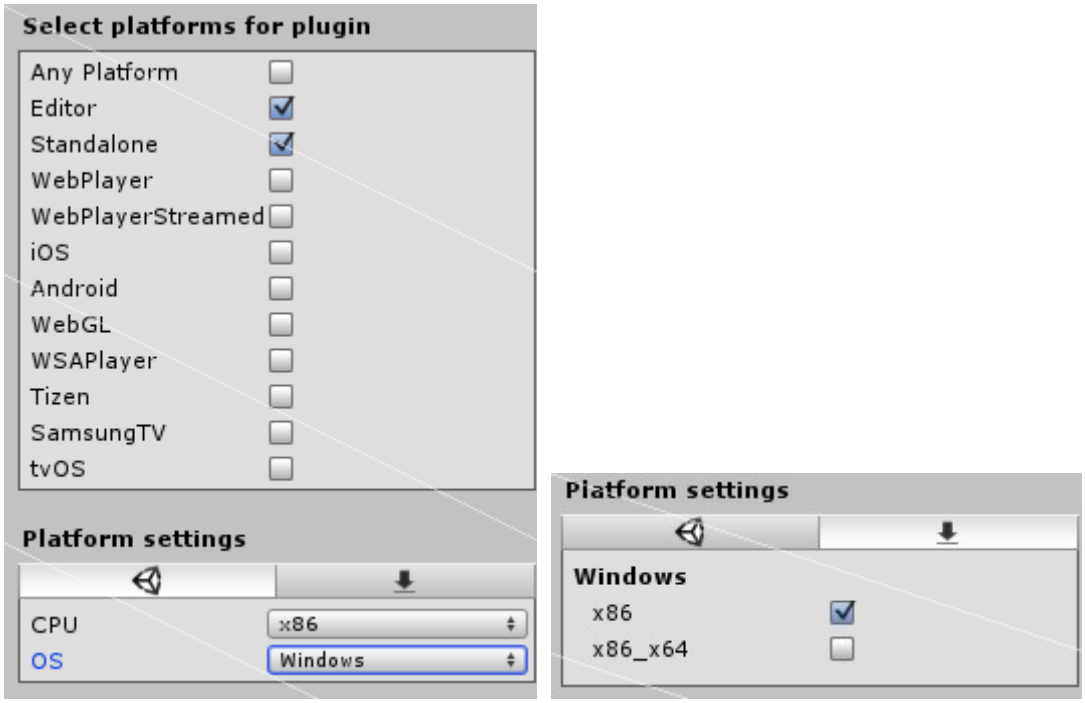
This example will use Unity 5, which imports the DLLs through an importer (older versions obligated a "Plugins" folder).
For compatibility, we will create the "Plugins" folder, but this is not compulsory.
Inside the "Plugins" folder, also create the "x86" and "x64" folders, so you can add both versions of your plugin and compile for both platforms.
Drag and drop each generated DLL file into its corresponding folder.
NOTE: You don't need to compile both versions of the DLL, but it is a good practice for compatibility and this way you will learn how to do it.

Now your project should look like this:



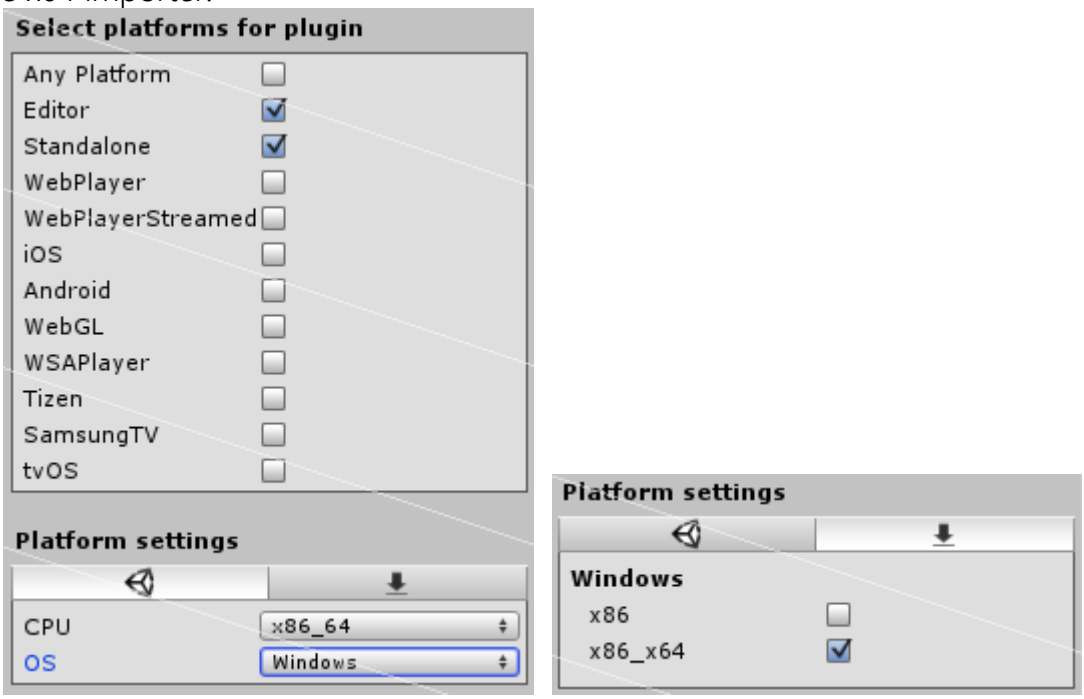
Both DLLs has different platform, so the importer must be different given that the code is not managed but native. You must configure the importers as follows.

Here the x86 importer:



If you are working on Windows, you can run the DLL in the Unity editor.

Here the x64 importer:



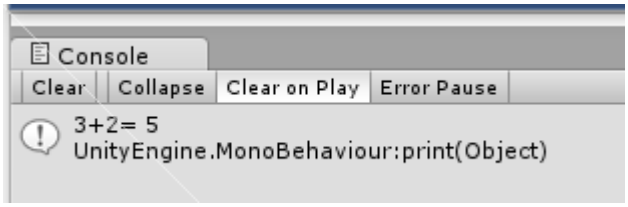
Don't forget to do the correct importing of DLL files, otherwise Unity will detect duplicated DLLs and will not compile correctly, given that both DLLs have the same name.
In the "Build settings" dialog, you can select the platform you wish to export, and Unity will chose the correct DLL to be included within the final product.

Create an empty C# source file, and name it "DllAccess.cs".
Your source code should look like this:

```
using UnityEngine;
using System.Runtime.InteropServices;

public class DllAccess : MonoBehaviour {

    [DllImport("Project1")] public static extern int GetInt(int n);

    // Use this for initialization
    void Start () {
        print("2+2= " + GetInt(2).ToString());
       }

       // Update is called once per frame
       void Update () {

       }
}
```

Attach this source code to the "Main Camera", dragging the ".cs" file from the "Project" view to the "Hierarchy" view, and dropping it over the "Main Camera" GameObject.

Run your project and the console should display this:



The `InteropServices` library is needed to access the `DllImport` directive that makes loading and accessing the DLL dynamically possible.

Unity loads the DLL files into memory at the first run, so you must restart Unity in order to update a DLL file. It is recommended to open the container folder ("Assets/Plugins/x86" or "Assets/Plugins/x64") and to overwrite the DLL file while Unity is closed.
Once replaced, start Unity and open your project again.

Feel free to experiment with the interfaces included in the Visual Studio example project to access some more interesting data types.
Normally, strings and byte arrays are the most important and complex types to be accessed.
For example: with a byte array, you can dynamically read an image or a video frame from a DLL and paint it into a Unity Texture2D in the fastest way.

Ending

If you find errors in the package or document, please contact me at jmonsuarez@gmail.com and product will be updated. Any comments, critics or improvements are also very welcome.

Hope you enjoyed the tutorial.

eToile, 2016