Salutations, I'm Jesco and you're watching Game Dev Made Easy.

致敬，我是Jesco，您正在看Game Dev Made Easy。

In the video today, we will create a basic Game Plugin implementation in CryEngine.

在今天的视频中，我们将在CryEngine中创建一个基本的Game Plugin实现。

We will be using the results from this video in further CryEngine tutorials.

我们将在进一步的CryEngine教程中使用此视频的结果。

(Intro)

First and foremost, what is GamePlugin and what is the purpose of it?

GamePlugin is a means for you to do game scripting with C++ in CryEngine, it enables your C++ code to

be read by the engine and allows for your custom code to be executed during gameplay. In other words,

without this, you’d be forced to use either GameSDK exclusively or attempt to do gameplay with

FlowGraph without extra features provided by custom C++ code or GameSDK.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Background Information Prior to Continuing:

The very first thing to do is to make sure we have the correct Windows 10 sdk and cpp tools. I have C++ ATL for latest v 142 build tools for 32- and 64-bit operating systems. I also have the Windows 10 SDK for every version available to us in Visual Studio.

What Crytek recommends is Visual Studio 2017 and Visual Studio 2019. Windows 10 SDK version 10.0.10586.0.

The next step is to obviously create a new project. Once the blank C++ project has been created, navigate to the folder where your project is located.

Right click on game and select to generate solution. Now that the solution has been generated, we can open the project in visual studio by navigating to the Solutions and Win64 folder and selecting the name of the project dot sln file.

Header File:

Go ahead and open the GamePlugin dot h file. This is the header file. What we are going to do is delete everything from this file. The reason is that we are going to build our own version of game plugin that does not have everything that the version of GamePlugin that Crytek pushes with the engine. Which means we will be able to get rid of the default Components that comes with the engine.

Do the same thing to the GamePlugin dot cpp file as well for the same reasons specified above.

The first thing we are going to write is hashtag pragma once. This is a preprocessor directive that is designed to make the compiler include the header file only once when compiling a source code file.

We can now do our include statements. The first one will be hashtag include less than sign CrySystem forward slash ICryPlugin dot h greater than sign. We are including the ICryPlugin because it is designed and implemented to represent an engine plug-in that can be loaded as a dynamic library, or statically linked in. Without it, our GamePlugin will fail to compile.

The last include statement is hashtag include less than symbol CryGame forward slash IGameFramework dot h greater than symbol. IGameFramework exposes all the CryAction subsystems to us. Which means we have access to update, initialize and so on events for the engine.

With the includes out of the way, we can now define our class…

Write class CGamePlugin colon public Cry colon colon IEnginePlugin comma public ISystemEventListener with an opening and closing curly bracket along with a semi-colon. Whew, that was a mouthful (clip of a mouthful), try saying that five times fast.

Okay, so what is this and why are we doing it?

This is the entry point of the application. An instance of CGamePlugin is automatically created when the library is loaded. We inherit from IEnginePlugin. The reason for this is because on startup, the engine parses the Game.cryproject file in your project directory, which in turn contains a path to our game plugin DLL file.  
We also inherit from the ISystemEventListenser Interface so we can register events to the engine.

It is now time to create some public functions.

Write public colon. This says that we are creating public variables and functions in this section of the header file.

The first thing we will add inside is CRYINTERFACE underscore SIMPLE with the parameter being Cry colon colon IEnginePlugin. The reason we are doing this is because Plug-ins utilize the engine's extension framework. This is a form of reflection allowing us to query implementations based on a specific interface. In this case, we indicate that our implementation implements Cry colon colon IEnginePlugin.

The next thing to write is CRYGENERATE underscore SINGLETONCLASS underscore GUID.The parameter will be CGamePlugin comma Quote Blank end quote comma quote Your GUID that you generate end quote undercore cry underscore guid. We do this because we need to set the GUID for our plug-in, this should be unique across all used plug-ins. The way this can be generated in Visual Studio under Tools -> Create GUID.

Next up, we will define our virtual functions.

The first virtual function is virtual tilde CGamePlugin. This is the destructor for this class which we will define in the cpp file.

The next virtual function is going to be our initialize function.

Write virtual bool Initialize with the parameter being SSystemGlobalEnvironment with the ampersand symbol called env comma const SSystemInitParams with the ampersand symbol called initParams. Afterwards we will write override semicolon.

This function is called shortly after loading the plugin from disk and is where you would initialize any third-party APIs and custom code.

The third and final virtual function is our System Event listener.

Write virtual void OnSystemEvent with the parameter being ESystemEvent called event comma UINT underscore PTR called wparam comma UINT underscore PTR called lparam. Afterwards we write override semicolon.

This function is designed to be able give us access to the engine’s system events.

This completes our header file and we can move on to the cpp file implementation.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Cpp file Creation:

In the GamePlugin dot cpp file, we have six include statements to make.

The first one is hashtag include quote stdafx dot h end quote. StdAfx is a header file which contains core system header files and definitions required for the engine.

The second one is hashtag include quote GamePlugin dot h end quote. Which is the header file we created for this cpp file, so it defines the functions and allows us to access functions and properties from other files that were named in the include statements that we need to implement and utilize.

The third one is hashtag include less than sign Cry Schematyc forward slash Env forward slash IEnvRegistry dot h greater than sign. Which is used for registering and deregistering packages in the engine.

The fourth one is hashtag include less than sign Cry Schematyc forward slash Env forward slash EnvPackage dot h greater than sign. Which is used for creating packages for the engine.

The fifth one is hashtag include less than sign Cry Schematyc forward slash Utils forward slash shared string dot h greater than sign. Which is a special string class that Crytek made for working with strings in the engine.

The sixth one is hashtag include less than sign Cry Core forward slash Platform forward slash platform underscore impl dot inl greater than sign. Which is a class that designates the system implementation that needs to be used for your package.

With the headers complete we can now implement the three functions that we defined in the header.

The first function we will implement is the destructor for the Game Plugin.

Write CGamePlugin colon colon tilde symbol CGamePlugin with an opening and closing parenthesis.

In C++ we call the name of the header file followed by two colons to say that we are defining a function that is outside of the class. The tilde signifies that this is a destructor and is also part of the name of the function itself. The name of the two colons side by side is commonly called the scope resolution operator.

Inside of the function write gEnv dash greater than symbol pSystem dash greater than symbol GetISystemEventDispatcher opening and close parenthesis followed by dash greater than symbol RemoveListener opening parenthesis this closing parenthesis and semi-colon.

This function call will remove any registered listeners before this game plugin goes out of scope.

I should also note that the dash greater than symbol has a similar meaning to the dot notation for function calls in C++. The meaning is as followed. The (dot) operator and the (arrow) operator are used to reference individual members of classes, structures, and unions. From now on, instead of saying dash greater than sign, I will reference it as the arrow operator.

The last things we need to do in this function is an if statement.

Write if opening and closing parenthesis gEnv arrow operator pSchematyc. Inside of the if statement write gEnv arrow operator pSchematyc arrow operator GetEnvRegistry with an opening and closing parenthesis dot DeregisterPackage opening parenthesis CGamePlugin colon colon GetCID opening and closing parenthesis and finally the closing parenthesis with a semi-colon.

What this call does is deregister the CGamePlugin package. And we deregister the plugin by the CID, which we will specify later.

Now we can define our Initialize function.

Write bool CGamePlugin colon colon Initialize. Opening parenthesis SSystemGlobalEnvironment ampersand symbol env comma const SSystemInitParams ampersand symbol initParams and a closing parenthesis. Following that, add your opening and closing curly brackets.

I should now note the meaning of the ampersand symbol. If you use the ampersand symbol on the left-hand side of a variable declaration, it means that you expect to have a reference to the declared type. It can be used in any type of declarations (local variables, class members, method parameters). It is a pointer that points to a specific place in memory, not that they have the same value.

Inside of the function write, gEnv arrow operator pSystem arrow operator GetISystemEventDispatcher with an opening and closing parenthesis. Following that add the arrow operator RegisterListener opening parenthesis this comma quote CGamePlugin end quote, closing parenthesis and a semi-colon.

This call will Register for engine system events.

Write return true with a semi-colon. And with that, the second function has been completed.

Our third and final function to define from the header file is the OnSystemEvent function.

Write void CGamePlugin colon colon OnSystemEvent opening parentesis ESystemEvent called event, UINT underscore PTR called wparam comma Uint underscore PTR called lparam and a closing parenthesis. Make sure to go ahead and add the opening and closing curly bracket.

Alright, let’s get to defining the inner function.

We will start off by creating a switch statement.

Write switch opening parenthesis event, closing parenthesis followed by an opening and closing curly bracket.

We will have two cases that we will want to switch on. ESystem underscore Event underscore Register underscore Schematyc underscore Env and ESystem underscore Event underscore Game underscore Post underscore Init.

Let’s go ahead and implement the first one.

Be sure that the Events are written in all capital letters.

Write case ESystem underscore Event underscore Register underscore Schematyc underscore Env with a colon at the end. Add your opening and closing curly bracket.

Inside, the first thing we want to do is to register all components that belong to the game plugin.

To do this, write auto staticAutoRegisterLambda equals sign and opening and closing bracket which is immediately followed by a opening parenthesis. Now write Schematyc colon colon IEnvRegistrar ampersand registrar with a closing parenthesis.

The auto keyword directs the compiler to use the initialization expression of a declared variable, or lambda expression parameter, to deduce its type. Think of it like var in JavaScript or C Sharp.

Add an opening and closing curly bracket and a semi-colon.

Inside of the opening curly bracket write the following.

Detail colon colon CStaticAutoRegistrar with a less than sign Schematyc colon colon IEnvRegistrar with the ampersand symbol, following that a greater than symbol colon colon InvokeStaticCallbacks with an opening parenthesis registrar closing parenthesis and a semi-colon.

This a call that will call all static callbacks that are registered with the Cry underscore Static underscore Auto underscore Register underscore with underscore param.

Outside of this lambda expression, we will make another if statement.

Write if opening parenthesis gEnv arrow operator pSchematyc closing parenthesis with an opening and closing curly bracket.

A basic if check is being employed here.

Inside of the if statement, write gEnv arrow operator pSchematyc arrow operator GetEnvRegistry opening and closing parenthesis dot RegisterPackage with an opening parenthesis.

To continue, write stl colon colon make underscore unique greater than symbol Schematyc colon colon CEnvPackage less than sign with an opening parenthesis.

The five parameters are going to be as followed.

CGamePlugin colon colon GetCID with an opening and closing parenthesis comma quote EntityComponents end quote comma quote Crytek GmbH end quote comma quote Components end quote comma staticAutoRegisterLambda with a closing parenthesis and another closing parenthesis with a semi-colon.

Whew. That was a long one. So, what is this doing? It is registering the GamePlugin and all the code that will reside in the Components folder to be part of the GamePlugin. And all of them will be uniquely identified as to make sure any changes or renaming doesn’t cause issues when the engine loads the plugin and custom code.

We still have one more case statement to write.

Write case ESystem underscore Event underscore Game underscore Post underscore Init with a colon.

Add the obligatory opening and closing curly brackets and let’s begin the inner definition.

Write if opening parenthesis exclamation point gEnv arrow operator IsEditor with an opening and closing parenthesis, followed by another closing parenthesis and the opening and closing curly brackets.

The exclamation point means not. So, this is saying if it is not the editor, do something.

Inside the if statement, write gEnv arrow operator pConsole arrow operator ExecuteString opening parenthesis quote map example end quote comma false comma true, closing parenthesis and a semi-colon.

This statement is saying to the console to load the map named example without silent mode being run and we want the execution to delay execution by wait seconds and wait frames commands.

Outside of the if statement, write break with a semi-colon.

We want to break out of this case.

All three functions have been defined and there is one last thing to do.

The final function will be Cryregister underscore singleton underscore class opening parenthesis CGamePlugin closing parenthesis.

This will register the factory that can create this plugin instance. This has to be done in a source file that is not included anywhere else.

Now, if we were to build this as I am doing now, it will fail. The reason for this is the example code is tightly coupled with the plugin code. To fix this, close visual studio, and navigate back to the folder that houses the project.

Navigate to the code folder, delve down into the components folder and delete the player dot h and player dot cpp file. We will not be using this version of the player class in any future code tutorials.

Once you’ve done this, regenerate the solution and open it in Visual Studio.

We can now build the plugin and it will compile successfully. However, there isn’t much point at this time as we don’t have any custom components yet.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In Closing:

In the next CryEngine tutorial, we will actually create some custom code that will utilize this GamePlugin.

This has been GameDevMadeEasy, and I’ll see you in the next one.