

How to make a simple Renegade Server Plug-In for TT 4.0

A complete newbies guide

This tutorial should enable you to successfully create your own server plug-in that works with TT 4.0 from scratch. I am deliberately going to use very simple terms, and try to explain each step assuming little to no previous knowledge on the subject.

The objective here is to increase the damage applied when a player damages something.

It will cover the resources required, references to supporting documentation, simple programming elements, building the code into a usable file and finally deploying it on your server.

The tutorial has been written specifically as a response to a request from a community member, but I will try to make it as generic as possible for others that might be interested.

First of all, you are going to need an Integrated Development Environment. This is just a term used for the application that allows you to edit the source code and includes an automated build feature, to turn that code into a file.

For this tutorial, you are going to need Microsoft Visual Studio 2010. It's an expensive program to purchase, and as such suffers terribly from online pirates who put it on torrent sites. There is an express edition, but I have never tried using it. The TT team confirmed it should work with the source code, but I have never tried. It's available [here](#).

You're also going to need the source code that the TT has released. This source code is more than what you really need. They have released an entire solution, which is basically a bunch of projects in one big container. One of the projects in this solution is a template, standard plug-in that actually does very little other than print out some messages, this is the one that we will use as a base. However, the other plug-in's are interesting to look at to understand how other things are accomplished.

You can download the source code [here](#).

There is also a guide for making a new plug-in using this source on the TT wiki, [here](#).

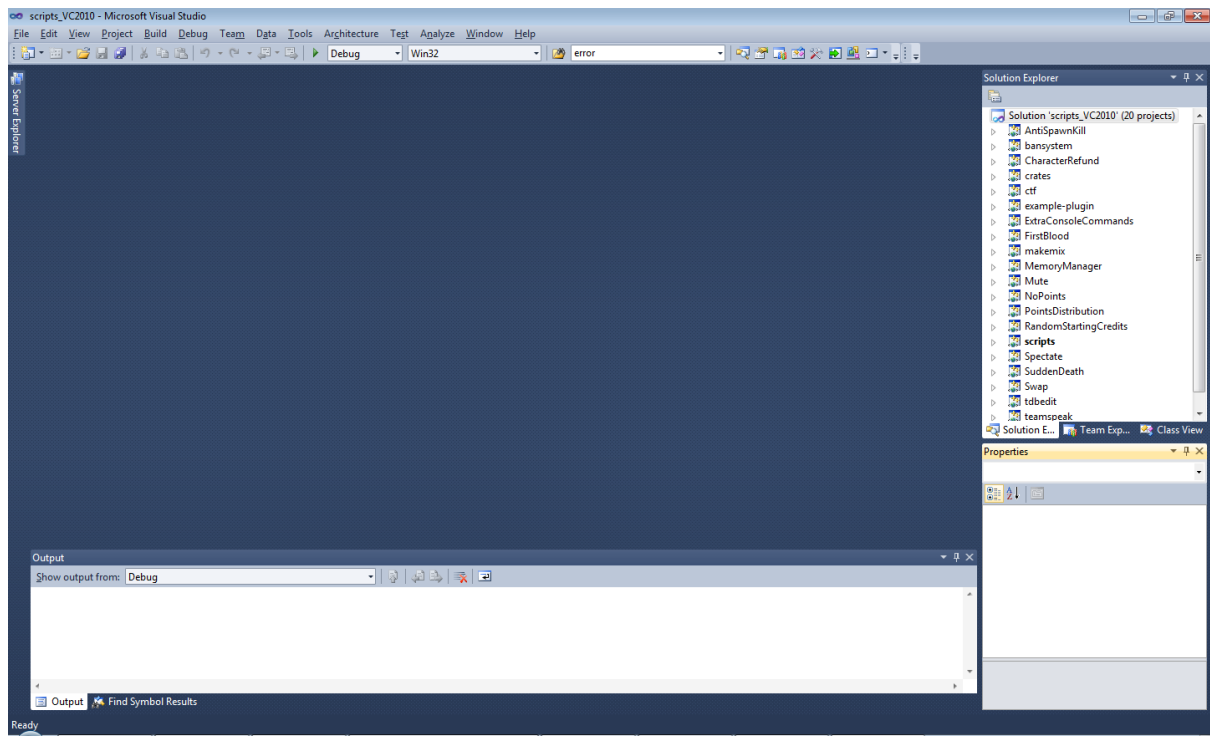
However, this guide is more about how to actually set-up your plug-in as a new project within the solution. It's complicated, manually intensive and pretty boring. It doesn't really get you making anything quickly and it doesn't really show you what's what.

This tutorial is aimed at the beginner, and as such, I feel no shame in taking a short-cut for you, and over-looking the part where you are encouraged to make a project within the solution, and just work in the existing example provided instead, and rename it to our choice.

Lots of waffle so far... Let's get cracking!

Assuming you've installed visual studio and downloaded the source code, open up the zip folder that should simply be called "tt-source-4.0.zip" and extract the folder called "source". It doesn't matter where you extract this folder, you can move it later if you like.

In the source folder you'll find a file called "scripts_VC2010.sln" you should open this file, and it should automatically open with your visual studio application, and look like this:

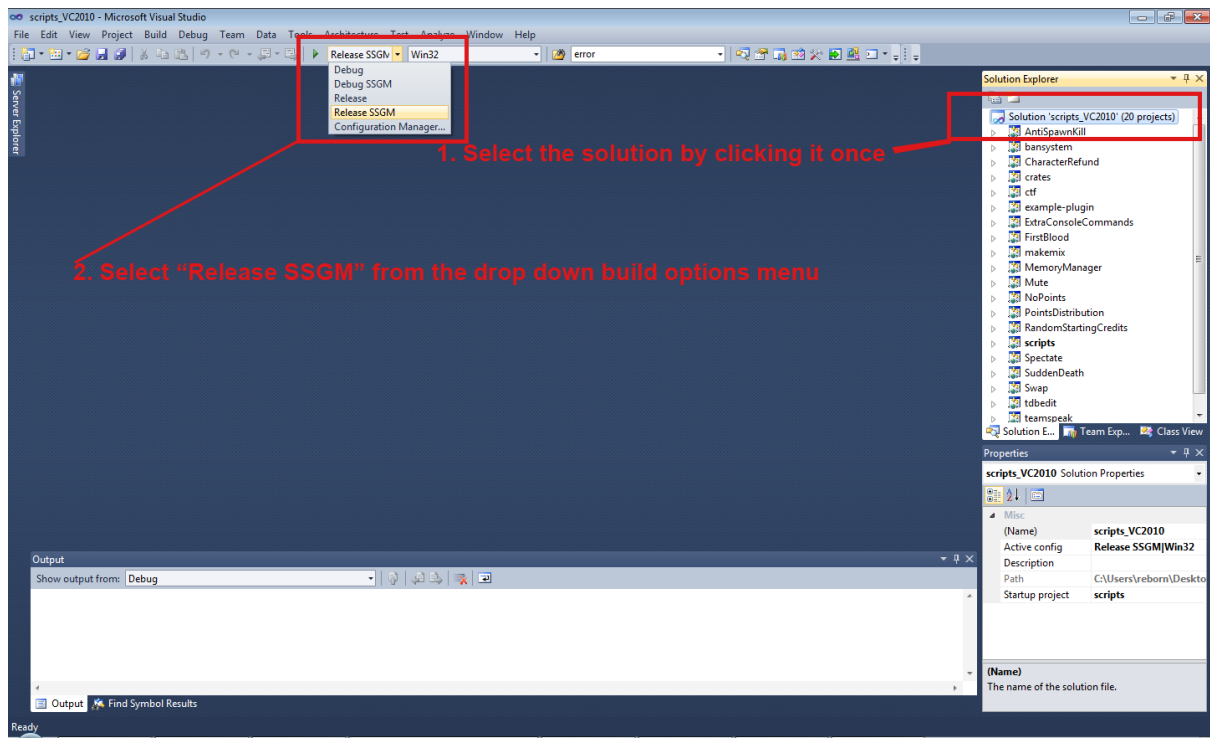


If it's the first time you've used the application, you will likely have some other prompts regarding your preferred set-up.

The first thing we are going to do is change the configuration of the example plug-in so that it isn't a debug version, but a regular SSGM release version. We do that because the SSGM release version has configuration settings that are needed to build an appropriate SSGM plug-in. To do that, follow these steps:

Select the solution at the top of the hierarchy in the solution explorer window called "scripts_VC2010" by clicking it once.

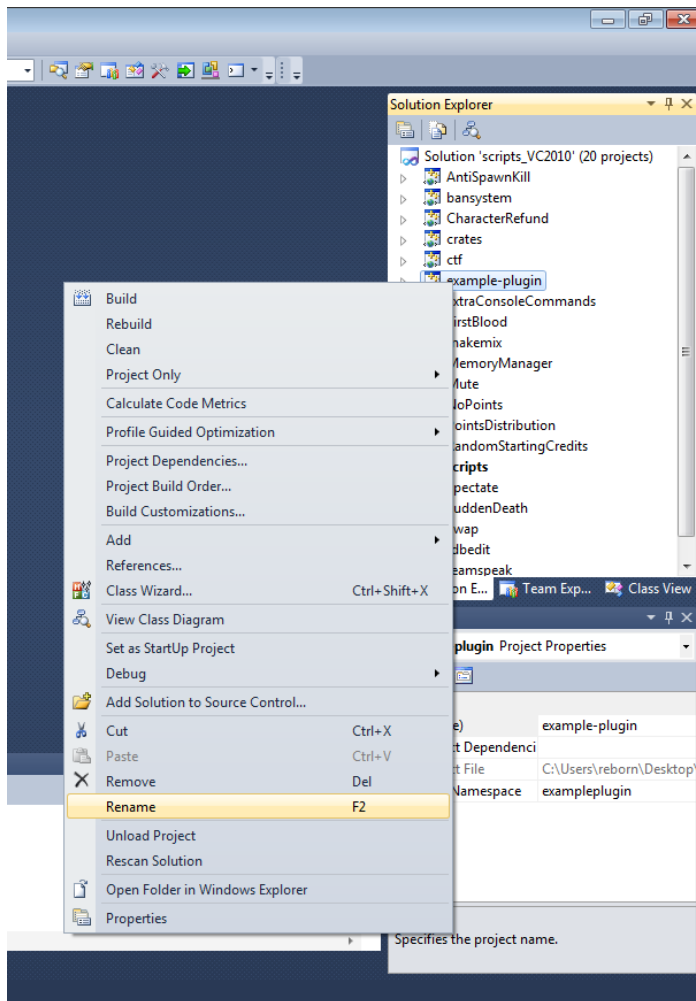
From the build configuration manager drop down menu select "Release SSGM".



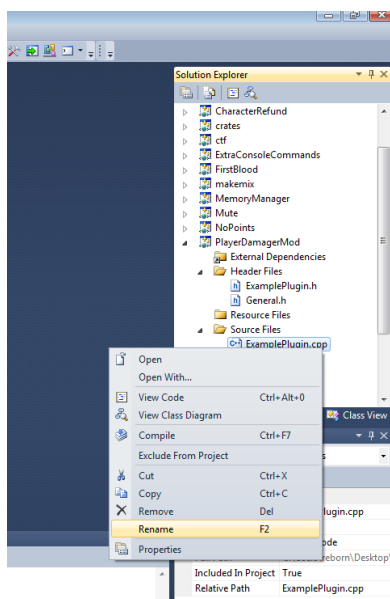
Now we are going to be sneaky and rename the example plug-in to our own plug-in name.

This is a bit of a cheat, technically you should be adding all your plug-in's to one solution, and maintaining them there. However, you may never get to that part, and it's always something you can do later. It takes fair amount of the boring part out of the job.

Simply right click on the "example-plugin" in the solution explorer window, and choose "rename". You can now call it whatever you like. Due to the request from the community member, I am going to call this plug-in PlayerDamagerMod.



You'll also want to rename the actual source files inside this project, to do that, just click the little triangle looking things next to your plug-in's name, and then also open up the folder called "Header Files" and the one called "Source Files". This should reveal ExamplePlugin.h and ExamplePlugin.cpp (as well as the General files). Rename the ExamplePlugin files to your projects name (but leave the file extensions .cpp and .h alone), like this:



Here's a little bit of information about the .cpp and .h files. It's irresponsible to just give instructions without really giving at least the smallest explanation as to why you are doing this.

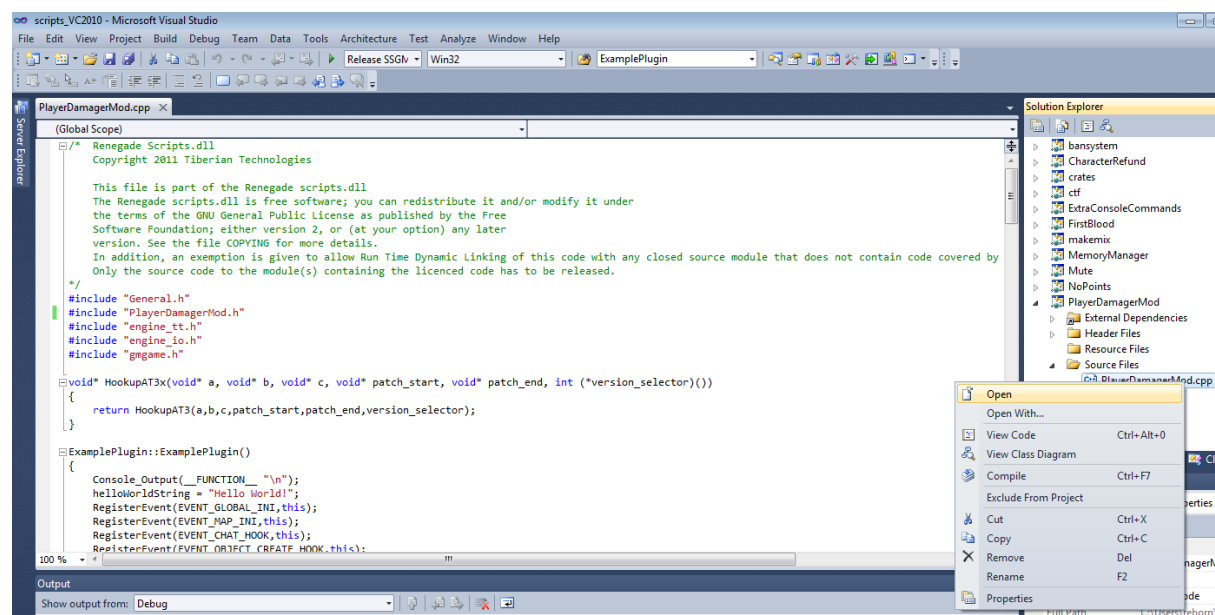
The solution is almost entirely written in C++, a programming language very commonly used to create desk top gaming titles (amongst other things). The plug-in is exclusively written in this language.

C++ is the language you are going to be using to write your modifications, and as such it makes use of **H**eader files (.h) and **C**Plus **P**lus source files (.cpp). The convention is that .h files contain declarations, and .cpp files contain definitions. This is an area that people could think I am being too basic on, but trying to keep it simple I'll give a little more depth to this.

The .cpp files contain the functions, the code that is actually run and executed. So say you write a function that gives a player some weapons, restores his health and changes his armor type to that of a tank. You might name that function "UberfyPlayer". The thing is, every function needs a declaration. So you write your function named UberfyPlayer, and then you declare that function in the header file (you can declare it inside the .cpp file, but that limits you to only being able to use that function inside that one single .cpp file). This allows you to use that function when you need to, and you can even use that function inside other .cpp files as long as you include that header file in the .cpp file.

You're now going to have to open the actual .cpp file (mine is called PlayerDamagerMod.cpp) and make some changes to it so that it will compile when you build it. We could spend some time here talking about linking, compiling and the build process, but I'll skim this by saying it's the process where we say "Ok, I've written my code, now turn this code into the finished product".

You can open the file by double clicking it, or right click and "open". You should get something that looks like this:



Welcome! This is where you really start to make a difference. Here is where you write the code that will change the way the server works.

You'll notice that there is some green writing at the top of the file in normal plain English. No, unfortunately it's not as easy as just typing in English and the application makes the program you want. However, you can add comments to your code that the application will ignore. It displays these comments in **green text**. You can add comments like this:

```
/* Anything I write
```

```
In-between these slash asterisk
```

```
Symbols will be ignored by the application when it compiles the code into my
```

```
Final product */
```

```
//I can write anything I like on a single line after a double slash and it will be ignored
```

This won't be ignored, and will cause an error **//But this will be ignored**

After the comments, you'll notice some **#include** references. This is what was discussed earlier regarding header (.h) files. There are functions that the plug-in needs to work, and these functions are included in other .cpp files, where the functions are declared in the header files you see listed at the top of the file.

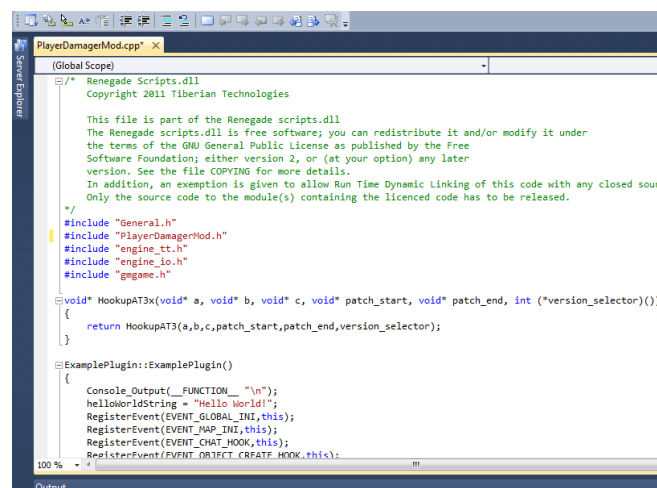
For our plug-in we will not need to add any more header files, but more complicated plug-ins almost certainly would.

You should also see some red squiggly lines under some of the lines of text. This isn't fool proof, but the application tries to pre-empt errors, and visually shows potential issues with these lines.

This one should be pretty obvious:

```
#include "General.h"
#include "ExamplePlugin.h"
#include "engine_tt.h"
```

We renamed ExamplePlugin.h to PlayerDamagerMod.h. Simply replacing that word should automatically remove the red squiggly line, try it now (you should notice the other red squiggly lines disappear too):

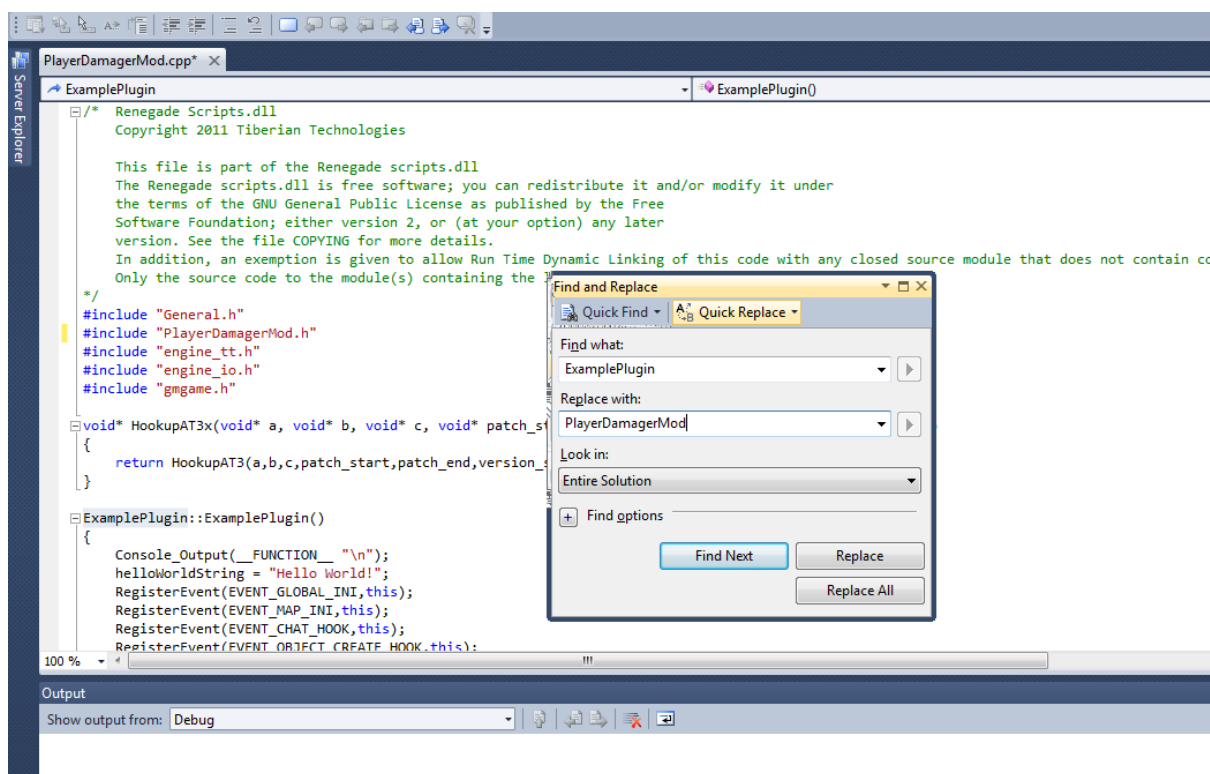


You'll notice that there are several references to "ExamplePlugin" in the source code, we will want to change that, too. It isn't really necessary to be honest, you could just leave it. However, it makes it a little bit more your own this way and should go a little way to help you understand how the plug-in works.

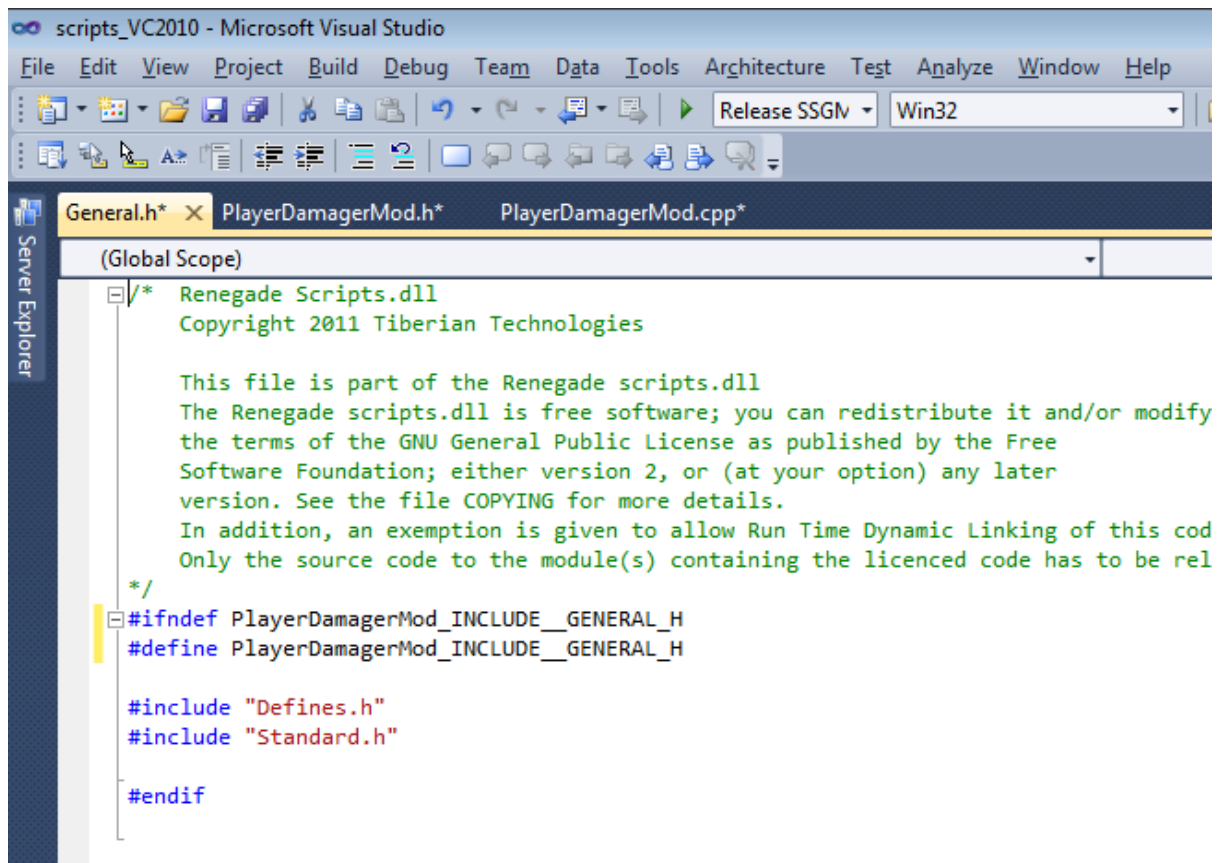
The easiest way to change these references is the Find & Replace function (pretty much the same thing across most Microsoft applications).

Just press the control key and F, then click on the "Quick Replace" tab.

Type "ExamplePlugin" into the "Find what" entry, and type "PlayerDamagerMod" into the "Replace with" entry. Just make sure you change the drop down option under "Look in" to Current project, that really important! You don't want to search and replace in the entire solution! The click on the "Replace all" button.



You should now see three tabs. This is because data in the those three tabs has been modified as part of your change. This is represented by the little star/asterisk on the tab. You'll need to save these changes. You can do that by going File->Save all, or individually right clicking each tab and then hitting save.



```
scripts_VC2010 - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Architecture Test Analyze Window Help
Release SSGM Win32
General.h* PlayerDamagerMod.h* PlayerDamagerMod.cpp*
(Global Scope)
/* Renegade Scripts.dll
Copyright 2011 Tiberian Technologies

This file is part of the Renegade scripts.dll
The Renegade scripts.dll is free software; you can redistribute it and/or modify
the terms of the GNU General Public License as published by the Free
Software Foundation; either version 2, or (at your option) any later
version. See the file COPYING for more details.
In addition, an exemption is given to allow Run Time Dynamic Linking of this cod
Only the source code to the module(s) containing the licenced code has to be rel
*/
#ifndef PlayerDamagerMod_INCLUDE__GENERAL_H
#define PlayerDamagerMod_INCLUDE__GENERAL_H

#include "Defines.h"
#include "Standard.h"

#endif
```

You should now have your own base for a plug-in!

It's difficult to continue at this point without mentioning programming lingo on the way, but I'll try to keep it to a minimum. It's also worth mentioning that programming inside this plug-in is not exactly the best introduction to programming... Normally you have a nice little example to fill out that will print the words "Hello World" to the console screen, followed up by an introduction to types of variables.

A quick word on "variables". Variables are like a storage location. Think back to maths at school where you have to solve an equation to work out what "x" equals, "x" is just a storage location.

With C++ these variables must be defined as a type. You can think of them like strict buckets, when you introduce your variable you must explain what type of variable it is. In the bucket analogy you might have a bucket specifically only for water, and buckets only for Oranges. In C++ you have variable types for integers (which are whole numbers: 1,2,3...), variable types for characters as well as many others. Below is a list of some of the common variable types.

Name	Description	Size*	Range*
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	Wide character.	2 or 4 bytes	1 wide character

My point I was trying to make though is that you are jumping in at the deep end, so don't get worried that things don't make lots of sense right now, or that this is hard.

Normally you wouldn't be working in an API (Advanced Programming Interface) for your first project, and you wouldn't have to be dealing with understanding the concept of hooks.

Calling it an API is being kind, essentially it's a giant hack, and as such, you're going to pick up some dirty habits.

The next thing you should see in the example is this:

```
void* HookupAT3x(void* a, void* b, void* c, void* patch_start, void* patch_end, int (*version_selector)())
{
    return HookupAT3(a,b,c,patch_start,patch_end,version_selector);
}
```

We're going to just walk on by this part and ignore it. You're not going to change it, and just need to know it's part of the engine to do with versioning.

After that you should see this:

```
PlayerDamagerMod:~PlayerDamagerMod()
{
    Console_Output(__FUNCTION__ "\n");
    helloWorldString = "Hello World!";
    RegisterEvent(EVENT_GLOBAL_INI,this);
    RegisterEvent(EVENT_MAP_INI,this);
    RegisterEvent(EVENT_CHAT_HOOK,this);
    RegisterEvent(EVENT_OBJECT_CREATE_HOOK,this);
    RegisterEvent(EVENT_LOAD_LEVEL_HOOK,this);
    RegisterEvent(EVENT_GAME_OVER_HOOK,this);
    RegisterEvent(EVENT_PLAYER_JOIN_HOOK,this);
    RegisterEvent(EVENT_PLAYER_LEAVE_HOOK,this);
    RegisterEvent(EVENT_REFILL_HOOK,this);
    RegisterEvent(EVENT_POWERUP_PURCHASE_HOOK,this);
    RegisterEvent(EVENT_VEHICLE_PURCHASE_HOOK,this);
    RegisterEvent(EVENT_CHARACTER_PURCHASE_HOOK,this);
    RegisterEvent(EVENT_THINK_HOOK,this);
}
```

This is part of a class. A class is like a data type, where you have a variable of type int. Only it's bigger in the sense that it can contain many variables and also functions.

This is a constructor, and while much can be [said about what a constructor is](#), all you really need to know right now is that the constructor sets everything up in the class and initialises the members of the class.

You might be able to intuitively recognise what is happening in this constructor...

Essentially, it's making calls to all those hacks that the boffins have written at TT, which hook into the server, and is watching for when certain things happen. You can see that they have included lots of events in the standard plug-in, like when a vehicle is purchased or a player joins the server.

There are other events that the TT guys have made available for you, but they have not registered them. The event that we really want is the damage event, so we're going to add it. We don't really need to "watch" the other events, so we could remove these registers and all the associated functions. We're not going to though, as you may want to expand on this plug-in, and they will help you understand how the server works when you see the function names triggered to appear on the FDS screen at different times.

Now we're going to register the damage events (plural). There are two damage events we need to watch; Those events sent from TT clients and damage events sent from non-TT clients. Due to TT changing the way that TT clients send damage data there are two events for this, I believe this is the only event that behaves this way.

Modifying both events ensures this is applied to non-TT players and TT players alike.

The events that are available in the standard example plug-in are defined in the gmplugin.h file. This file is under the "scripts" project, inside the SSGM folder. Here is the list and their explanations:

```
virtual float Get_Version() {return INTERFACE_VERSION;} //returns the version of the SSGM interface this plugin is intended to work with
virtual void OnLoadGlobalINISettings(INIClass *SSGMINi) {}; //called when ssgm.ini is parsed to read global settings
virtual void OnFreeData() {}; //called when data allocated for global settings is freed (i.e. on shutdown and before global settings are re-loaded)
virtual void OnLoadMapINISettings(INIClass *SSGMINi) {}; //called when ssgm.ini is parsed to read per-map settings
virtual void OnFreeMapData() {}; //called when data allocated for per-map settings is freed (i.e. on shutdown and before per-map settings are re-loaded)
virtual bool OnChat(int PlayerID, TextMessageEnum Type, const wchar_t *Message, int receiverID) {return true;} //called on chat message
virtual void OnObjectCreate(void *data, GameObject *obj) {}; //called on object create
virtual void OnLoadLevel() {}; //called on level load
virtual void OnGameOver() {}; //called on game over
virtual void OnPlayerJoin(int PlayerID, const char *PlayerName) {}; //called on player join
virtual void OnPlayerLeave(int PlayerID) {}; //called on player leave
virtual bool OnRefill(GameObject *purchaser) {return true;} //called on refill
virtual int OnPowerupPurchase(BaseControllerClass *base, GameObject *purchaser, unsigned int cost, unsigned int preset, const char *data) {return -1;} //called on beacon purchase
virtual int OnVehiclePurchase(BaseControllerClass *base, GameObject *purchaser, unsigned int cost, unsigned int preset, const char *data) {return -1;} //called on vehicle purchase
virtual int OnCharacterPurchase(BaseControllerClass *base, GameObject *purchaser, unsigned int cost, unsigned int preset, const char *data) {return -1;} //called on character purchase
virtual void OnThink() {}; //called once per frame
virtual bool OnRadioCommand(int PlayerType, int PlayerID, int AnnouncementID, int IconID, AnnouncementEnum AnnouncementType) {return true;} //called on radio command
virtual bool OnStockDamage(PhysicalGameObj* damager, PhysicalGameObj* target, float damage, uint warheadID) {return true;} //called on damage from clients with version <4.0
virtual bool OnTtDamage(PhysicalGameObj* damager, PhysicalGameObj* target, const AmmoDefinitionClass* ammo, const char* bone) {return true;} //called on damage from clients with version >=4.0
virtual void OnPreLoadLevel() {}; //called on level load but before the client is sent any network updates
```

Technically, you can write your own hooks and register other events. However, that's way beyond the scope of this tutorial.

You'll notice these two events:

```
virtual bool OnRadioCommand(int PlayerType, int PlayerID, int AnnouncementID, int IconID, AnnouncementEnum AnnouncementType)
{return true;} //called on radio command
virtual bool OnStockDamage(PhysicalGameObj* damager, PhysicalGameObj* target, float damage, uint warheadId) { return true; } //called
on damage from clients with version < 4.0
```

These are the ones we're going to additionally register in our plug-in.

Open up the **PlayerDamagerMod.h** and you should see the following:

```
class PlayerDamagerMod :
    public Plugin
{
    StringClass helloworldString;

public:
    PlayerDamagerMod();
    ~PlayerDamagerMod();

    virtual void OnLoadGlobalINISettings(INIClass *SSGMINi);
    virtual void OnFreeData();
    virtual void OnLoadMapINISettings(INIClass *SSGMINi);
    virtual void OnFreeMapData();
    virtual bool OnChat(int PlayerID, TextMessageEnum Type, const wchar_t *Message, int recieverID);
    virtual void OnObjectCreate(void *data, GameObject *obj);
    virtual void OnLoadLevel();
    virtual void OnGameOver();
    virtual void OnPlayerJoin(int PlayerID, const char *PlayerName);
    virtual void OnPlayerLeave(int PlayerID);
    virtual bool OnRefill(GameObject *purchaser);
    virtual int OnPowerupPurchase(BaseControllerClass *base, GameObject *purchaser, unsigned int
cost, unsigned int preset, const char *data);
    virtual int OnVehiclePurchase(BaseControllerClass *base, GameObject *purchaser, unsigned int
cost, unsigned int preset, const char *data);
    virtual int OnCharacterPurchase(BaseControllerClass *base, GameObject *purchaser, unsigned int
cost, unsigned int preset, const char *data);
    virtual void OnThink();
};
```

This is the entire class declaration, and where we need to add the declarations for the damage functions.

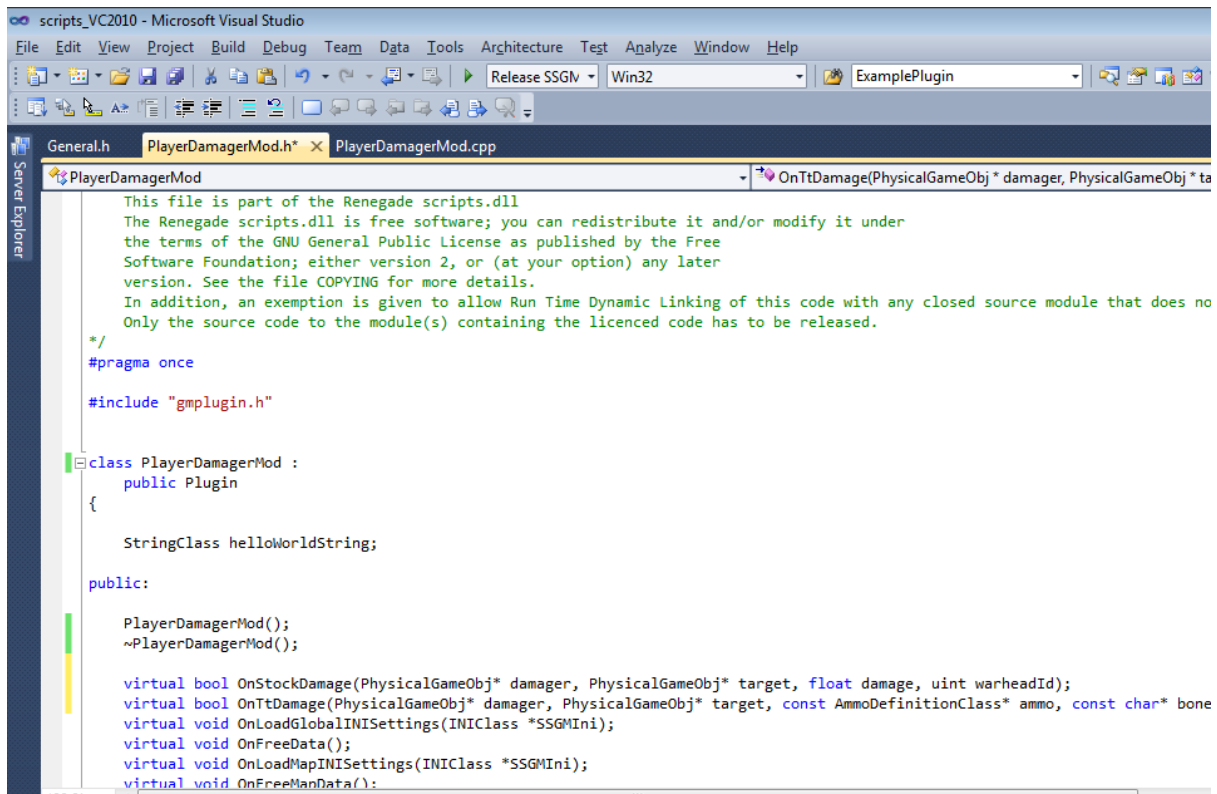
Add the following code here:

```
virtual bool OnStockDamage(PhysicalGameObj* damager, PhysicalGameObj* target, float damage, uint
warheadId);
virtual bool OnTtDamage(PhysicalGameObj* damager, PhysicalGameObj* target, const AmmoDefinitionClass*
ammo, const char* bone);
```

Just above the line:

```
virtual void OnLoadGlobalINISettings(INIClass *SSGMINi);
```

Save this file, so you'll have something that looks like this:



Now that the functions are declared, you have to actually define the functions and you need to register the event so that the functions are actually called when the plug-in detects that the event you're watching for has happened.

To do that, open up PlayerDamagerMod.cpp. Then you need to modify the constructor to include the event register, make yours look like this:

```

PlayerDamagerMod::PlayerDamagerMod()
{
    Console_Output(__FUNCTION__ "\n");
    RegisterEvent(EVENT_STOCK_DAMAGE_HOOK, this);
    RegisterEvent(EVENT_TT_DAMAGE_HOOK, this);
    helloWorldString = "Hello World!";
    RegisterEvent(EVENT_GLOBAL_INI, this);
    RegisterEvent(EVENT_MAP_INI, this);
    RegisterEvent(EVENT_CHAT_HOOK, this);
    RegisterEvent(EVENT_OBJECT_CREATE_HOOK, this);
    RegisterEvent(EVENT_LOAD_LEVEL_HOOK, this);
    RegisterEvent(EVENT_GAME_OVER_HOOK, this);
    RegisterEvent(EVENT_PLAYER_JOIN_HOOK, this);
    RegisterEvent(EVENT_PLAYER_LEAVE_HOOK, this);
    RegisterEvent(EVENT_REFILL_HOOK, this);
    RegisterEvent(EVENT_POWERUP_PURCHASE_HOOK, this);
    RegisterEvent(EVENT_VEHICLE_PURCHASE_HOOK, this);
    RegisterEvent(EVENT_CHARACTER_PURCHASE_HOOK, this);
    RegisterEvent(EVENT_THINK_HOOK, this);
}

```

The part you need to add is:

```

RegisterEvent(EVENT_STOCK_DAMAGE_HOOK, this);
RegisterEvent(EVENT_TT_DAMAGE_HOOK, this);

```

The "EVENT_STOCK_DAMAGE_HOOK" and TT version are found in gmpugin.h.

Very basically, what you have just done is told your plug-in to watch out for when damage occurs.

It would be nice to just set the damage value of the base defence, in which case we wouldn't have to watch out for damage events and apply additional logic, but the damage that is applied is calculated client side, so we have to accomplish the effect this way.

The class however also contains a destructor, this is basically a nice clean way of closing down the class object elegantly and cleanly. This is where you tell the server to stop watching out for the events you registered previously.

The destructor looks very similar to the constructor, but you'll notice the "~" symbol.

Unregister the event like this, then save the file.

```
PlayerDamagerMod::~PlayerDamagerMod()
{
    Console_Output(__FUNCTION__ "\n");
    UnregisterEvent(EVENT_STOCK_DAMAGE_HOOK, this);
    UnregisterEvent(EVENT_TT_DAMAGE_HOOK, this);
    UnregisterEvent(EVENT_GLOBAL_INI, this);
    UnregisterEvent(EVENT_MAP_INI, this);
    UnregisterEvent(EVENT_CHAT_HOOK, this);
    UnregisterEvent(EVENT_OBJECT_CREATE_HOOK, this);
    UnregisterEvent(EVENT_LOAD_LEVEL_HOOK, this);
    UnregisterEvent(EVENT_GAME_OVER_HOOK, this);
    UnregisterEvent(EVENT_PLAYER_JOIN_HOOK, this);
    UnregisterEvent(EVENT_PLAYER_LEAVE_HOOK, this);
    UnregisterEvent(EVENT_REFILL_HOOK, this);
    UnregisterEvent(EVENT_POWERUP_PURCHASE_HOOK, this);
    UnregisterEvent(EVENT_VEHICLE_PURCHASE_HOOK, this);
    UnregisterEvent(EVENT_CHARACTER_PURCHASE_HOOK, this);
    UnregisterEvent(EVENT_THINK_HOOK, this);
}
```

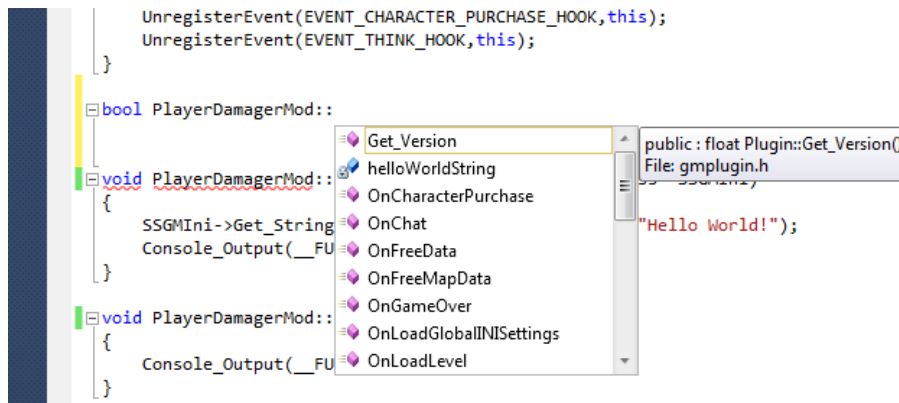
This is just a graceful way of closing it all down.

You're now going to write a function that is called when that registered event actually happens, and apply some additional logic when that takes place.

You now need to add the actual function, to do this start typing `bool PlayerDamagerMod::` just above the function here:

```
void PlayerDamagerMod::OnLoadGlobalINISettings(INIClass *SSGMINi)
{
    SSGMINi->Get_String(helloWorldString, "General", "Hello World!");
    Console_Output(__FUNCTION__ "\n");
}
```

You should notice that Visual Studio actually tries helping you now. Because you have included the function declaration in your class already, it will work similar to predictive cell phone messaging (and by cell phone, I really mean mobile phone), like this:



Both of the functions should look like this:

```
bool PlayerDamagerMod::OnStockDamage(PhysicalGameObj* damager, PhysicalGameObj* target, float damage,
uint warheadId)
{
    return true;
}
```

```
bool PlayerDamagerMod::OnTtDamage(PhysicalGameObj* damager, PhysicalGameObj* target, const
AmmoDefinitionClass* ammo, const char* bone)
{
    return true;
}
```

So you should have something that looks like this:

```
player
{
    UnregisterEvent(EVENT_VEHICLE_PURCHASE_HOOK,this);
    UnregisterEvent(EVENT_CHARACTER_PURCHASE_HOOK,this);
    UnregisterEvent(EVENT_THINK_HOOK,this);
}

bool PlayerDamagerMod::OnStockDamage(PhysicalGameObj* damager, PhysicalGameObj* target, float damage, uint warheadId)
{
    return true;
}

bool PlayerDamagerMod::OnTtDamage(PhysicalGameObj* damager, PhysicalGameObj* target, const AmmoDefinitionClass* ammo, const char* bone)
{
    return true;
}

void PlayerDamagerMod::OnLoadGlobalINISettings(INIClass *SSGMINi)
{
    SSGMINi->Get_String(helloWorldString, "General", "Hello World!");
    Console_Output(__FUNCTION__ "\n");
}
```

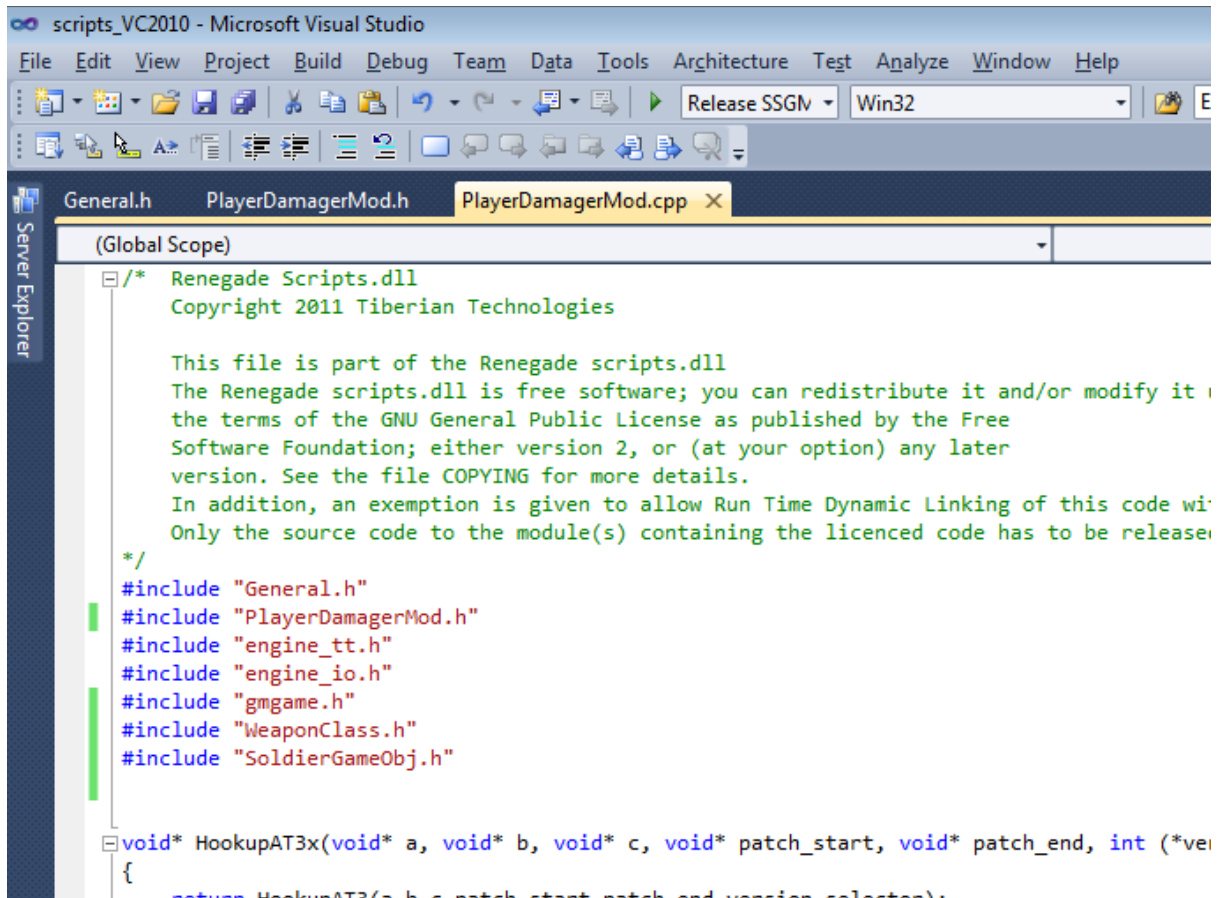
What you basically have here are two functions that are called. OnStockDamage is called (activated/executed/run) when a player who is not running TT causes some damage to occur in-game, and onTtDamage is called when a TT player causes some damage in-game.

In order to make our mod we're going to need to include some other classes, this is because we are going to use some of the function parameters, and some of these parameters contain types that are declared in classes outside of our plug-in. They are not the standard int/bool/char etc type variables, but something entirely bespoke.

We need to include:

```
#include "WeaponClass.h"
#include "SoldierGameObj.h"
```

So at the top of your PlayerDamagerMod.cpp file make the changes to look like this:



```
scripts_VC2010 - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Architecture Test Analyze Window Help
Release SSGN Win32
Server Explorer
General.h PlayerDamagerMod.h PlayerDamagerMod.cpp X
(Global Scope)
/* Renegade Scripts.dll
Copyright 2011 Tiberian Technologies

This file is part of the Renegade scripts.dll
The Renegade scripts.dll is free software; you can redistribute it and/or modify it
the terms of the GNU General Public License as published by the Free
Software Foundation; either version 2, or (at your option) any later
version. See the file COPYING for more details.
In addition, an exemption is given to allow Run Time Dynamic Linking of this code wi
Only the source code to the module(s) containing the licenced code has to be release

*/
#include "General.h"
#include "PlayerDamagerMod.h"
#include "engine_tt.h"
#include "engine_io.h"
#include "gmgame.h"
#include "WeaponClass.h"
#include "SoldierGameObj.h"

void* HookupAT3x(void* a, void* b, void* c, void* patch_start, void* patch_end, int (*ver
{
    return HookupAT3(a, b, c, patch_start, patch_end, version_selector);
}
```

Now back to your functions and a little explanation...

```
bool PlayerDamagerMod::OnStockDamage(PhysicalGameObj* damager, PhysicalGameObj* target, float damage,
uint warheadId)
```

You'll see that your first function starts with the word "bool" in blue. The blue is just the applications way of letting you know it's a type. This means that your function returns a value, and that value is type bool.

Not all functions return values. Some functions make changes inside the program and do not return any value at all.

But say you had a program that solved a maths problem. The program started up and the application asks the user to input the value of "x". The function then took that value "x" and multiplied it by 10.

You're going to want that function to return the final value!

This function is type bool though, we could go into the history of the name, but all you really need to know is that bool can only hold two possible values; either 1 or 0 (although it's easier to think of this as either "yes or no" or "true or false").

That bool value (Boolean) in this case represents whether or not the damage is allowed to be applied or not. Meaning you could return the value "0" or "false" and no damage would be applied. Pretty cool!

The reason why this bool value represents whether damage is allowed to be applied or not is because TT have hooked the damage event. This is not normal coding as you would first be introduced to, and actually relies on hacks. We're not going to go into this, but it was worth a little explanation so you didn't just think you can conjure up variable types and their relationship to in-game events.

After the function name you have some parentheses "()", and stuff inside them. Each thing in the parentheses (separated by the coma) is called a function parameter.

These are the things that you use when calling the function.

So for example, you might have a function like this:

```
Int MultiplyByTen(int x)
{
    int Value = 0;
    Value = x * 10;
    return Value;
}
```

The above function allows you to call it, and you can specify any value (or other variable) of type int as x. So it will take any whole number, times it by ten, and then return that value.

You'll notice that each parameter also has a variable type. Half of these types are pretty specific to the Plug-In API, and you won't find them anywhere else. This is why trying to learn how to program and starting directly with the Renegade API can be confusing.

The point here is though, that these parameters can be used inside the function itself.

Because these are hooks, you are not calling the functions, they are called by the server itself when the events are triggered, but you can alter the return value, perform functions inside this function and use the parameters that are passed to the function.

Basically, you get to a pointer to the actual thing that caused the damage, the thing that's damaged, the amount of damage applied and the warhead that was used.

The TT version is similar, but gives different parameters that are more useful and granulated.

After the parentheses you have a curly brace "{" symbol. Blocks of code are enclosed between these braces, so it allows you to add statements (the body of your function) that is to be executed inside them.

Here is how you apply the additional damage any time damage occurs that is calculated and sent by the client (building damage is calculated server side). Luckily there is already a function for that:

```
void (* Apply_Damage)( GameObject * object, float amount, const char * warhead_name, GameObject * damager);
```

You have to pass these function parameters for it to actually work, so you use it like this:

```
Commands->Apply_Damage(target, damage, ArmorWarheadManager::Get_Warhead_Name(warheadId), damager);
```

We've basically used all the existing parameters from the original function in our statement here.

The first parameter is saying what should we apply this damage to. We want to apply the damage to the sucker that's getting owned.

The second parameter is an actual amount, a number. It's not an integer, but a floating point number. Really though, it's just the amount of damage we want to apply. I've put in here the same amount of damage that was originally applied, so effectively you have doubled the damage. This is important, because we will expand on this a little bit to make the plug-in more useful and dynamic.

The only one that's a little different is the third one, the function is looking for the warhead name, but our parameter in the original function gives us the warhead ID. We simply look up the name of the warhead by using the warhead ID, by calling another function.

The fourth parameter is just who is applying the damage, and we keep the damager the same (the player).

You should now have something like this:

```
bool PlayerDamagerMod::OnStockDamage(PhysicalGameObj* damager, PhysicalGameObj* target, float damage, uint warheadId)
{
    Commands->Apply_Damage(target, damage, ArmorWarheadManager::Get_Warhead_Name(warheadId), damager); //This statement is essentially applying the same amount of damage to the target again.
    return true;
}
```

The only thing here is that it's pretty static. It makes buildings apply twice as much damage, but you might want to a different amount of damage. You could do that like this:

```
bool PlayerDamagerMod::OnStockDamage(PhysicalGameObj* damager, PhysicalGameObj* target, float damage, uint warheadId)
{
    Commands->Apply_Damage(target, damage/2, ArmorWarheadManager::Get_Warhead_Name(warheadId), damager); //This statement is essentially applying the same amount of damage to the target again.
    return true;
}
```

By changing the second parameter to "damage/2" you are applying another 50% worth of damage instead of another full 100%.

You could alter this quite a lot, like "damage*3" meaning the damage that is applied in total is 4 times the original.

But do you really want to have to change this value, compile it, and put it on the server when you are trying to find the right balance?

Instead of a static value like 2, you could use a variable.

So you have damage*x, where x is a floating point number and it could actually be 0.5 or 5.0.

Instead of calling it x, we will call this “Modifier”, and we’re going to let the .ini file define what value is assigned to “Modifier”.

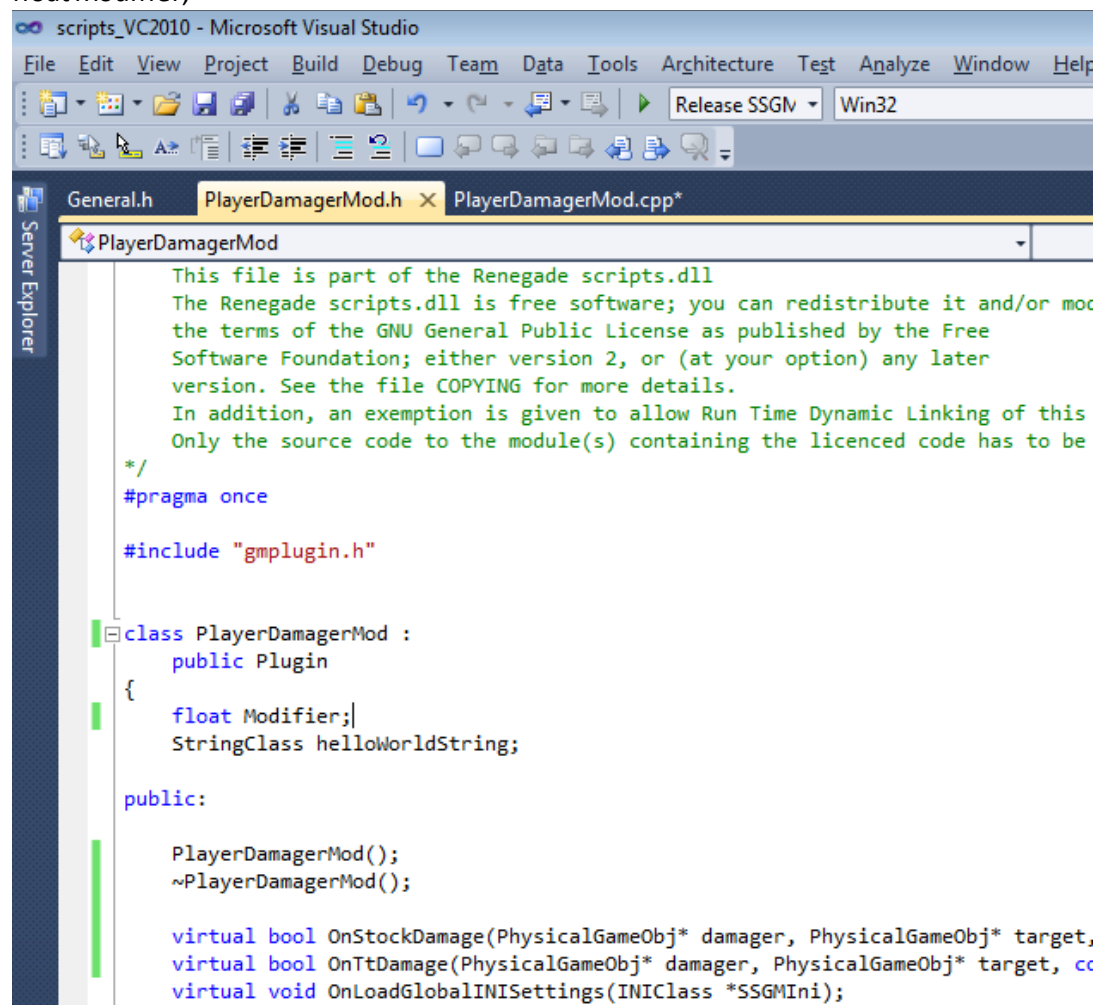
As a side note, you could make the base defence damage become less and less over time in a marathon server as a gimmick. All you would need to do is “return false” inside your code block of curly braces, so no normal damage is applied, and apply the damage yourself using the code, so that over time it goes from 3 times as power to applying no damage.

I’m not sure whether that would be a good idea or not, but it should make you realise that you can change the value of your variables at any time, and do not need them to be static.

This is how you make an entry in the .ini file read from the plug-in.

Firstly you need to declare the variable in your class, so open your .h file, and add this:

float Modifier;



```
scripts_VC2010 - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Architecture Test Analyze Window Help
Release SSGM Win32
General.h PlayerDamagerMod.h PlayerDamagerMod.cpp*
Server Explorer
PlayerDamagerMod
This file is part of the Renegade scripts.dll
The Renegade scripts.dll is free software; you can redistribute it and/or modify
the terms of the GNU General Public License as published by the Free
Software Foundation; either version 2, or (at your option) any later
version. See the file COPYING for more details.
In addition, an exemption is given to allow Run Time Dynamic Linking of this
Only the source code to the module(s) containing the licenced code has to be
*/
#pragma once
#include "gmpplugin.h"

class PlayerDamagerMod :
public Plugin
{
float Modifier;
StringClass helloWorldString;

public:
PlayerDamagerMod();
~PlayerDamagerMod();

virtual bool OnStockDamage(PhysicalGameObj* damager, PhysicalGameObj* target,
virtual bool OnTtDamage(PhysicalGameObj* damager, PhysicalGameObj* target, cc
virtual void OnLoadGlobalINISettings(INIClass *SSGMINi);
```

Then save the file.

Now go back to the .cpp file and find the following:

```
void PlayerDamagerMod::OnLoadGlobalINISettings(INIClass *SSGMINi)
{
SSGMINi->Get_String(helloWorldString, "General", "Hello World!");
Console_Output(__FUNCTION__ "\n");
}
```

This is where the ini files should be loaded and the values read.

You’ll notice that the example plug-in already loads a string type variable, and then has a Console_Poutput function.

The string “Hello World” isn’t doing us any harm at all, we can leave it there as an example you might like to use later on. The console output can actually be confusing to newbies, so I’ll explain it a little here, because you’ll see it often throughout the plug-in.

Console_Output is a function that allows you to print text to the FDS console itself. It’s not seen by players, just printed to the screen.

It normally contains text, or a text variable that is printed. In this case though, the `__FUNCTION__` actually passes the function name as a parameter.

The reason for this is so that the function name is printed to the console screen when that function is called.

It’s designed to help beginners understand when the events are being called. You should leave them all there, as this will help you learn more how things are working.

This is where we are going to use a pre-existing function and pass in our parameters to get our ini value from the ini file, like this:

```
void PlayerDamagerMod::OnLoadGlobalINISettings(INIClass *SSGMINi)
{
    Modifier = SSGMINi->Get_Float("General", "DamageModifier", 2.0);
    SSGMINi->Get_String(helloWorldString, "General", "Hello World!");
    Console_Output(__FUNCTION__ "\n");
}
```

We’ve already declared “Modifier” in the class, so we can use that variable here, and it will know exactly what we’re referring to.

What this now means is that the default modifier value is 2.0. You have to remember that this essentially makes the total damage 3 times more, not two.

You could set it to 0.5 for an additional 50% damage.

You also need to make sure that under the [General] settings in the ini file, that the entry “DamageModifier=2.0” is present. Of course, you can change that value to anything you like, the server defaults to 2.0 in case you forget to set it in the ini file. The ini file is what will set the value to.

Now we have this variable that can be modified from the ini file, we need to update our damage hook, like this:

```
bool PlayerDamagerMod::OnStockDamage(PhysicalGameObj* damager, PhysicalGameObj* target, float damage,
uint warheadId)
{
    Commands->Apply_Damage(target, damage * Modifier,
ArmorWarheadManager::Get_Warhead_Name(warheadId), damager); //This statement is essentially applying
the same amount of damage to the target again.
    return true;
}
```

We changed the stock damage function, but we now need to go back and apply almost exactly the same code to the TT version.

It’s not precisely the same, because the TT version has deeper details of what’s occurring with damage, so we’ll need to modify it a bit, like this:

```
bool PlayerDamagerMod::OnTtDamage(PhysicalGameObj* damager, PhysicalGameObj* target, const
AmmoDefinitionClass* ammo, const char* bone)
{
    float damage = ammo->Damage();
    int warheadId = ammo->Warhead();
    Commands->Apply_Damage(target, damage * Modifier,
ArmorWarheadManager::Get_Warhead_Name(warheadId), damager); //This statement is essentially applying
the same amount of damage to the target again.
    return true;
}
```

All that's happened here is that the damage and warheadId variables have had to be extracted from the different parameters that are available on the TT damage hook.

We're also going to remove one of the printed messages that occurs on the "think event", if you do not remove it, it will cause considerable "spam" on your FDS console.

Simply change this code:

```
void PlayerDamagerMod::OnThink()
{
    Console_Output(__FUNCTION__ "\n");
}
```

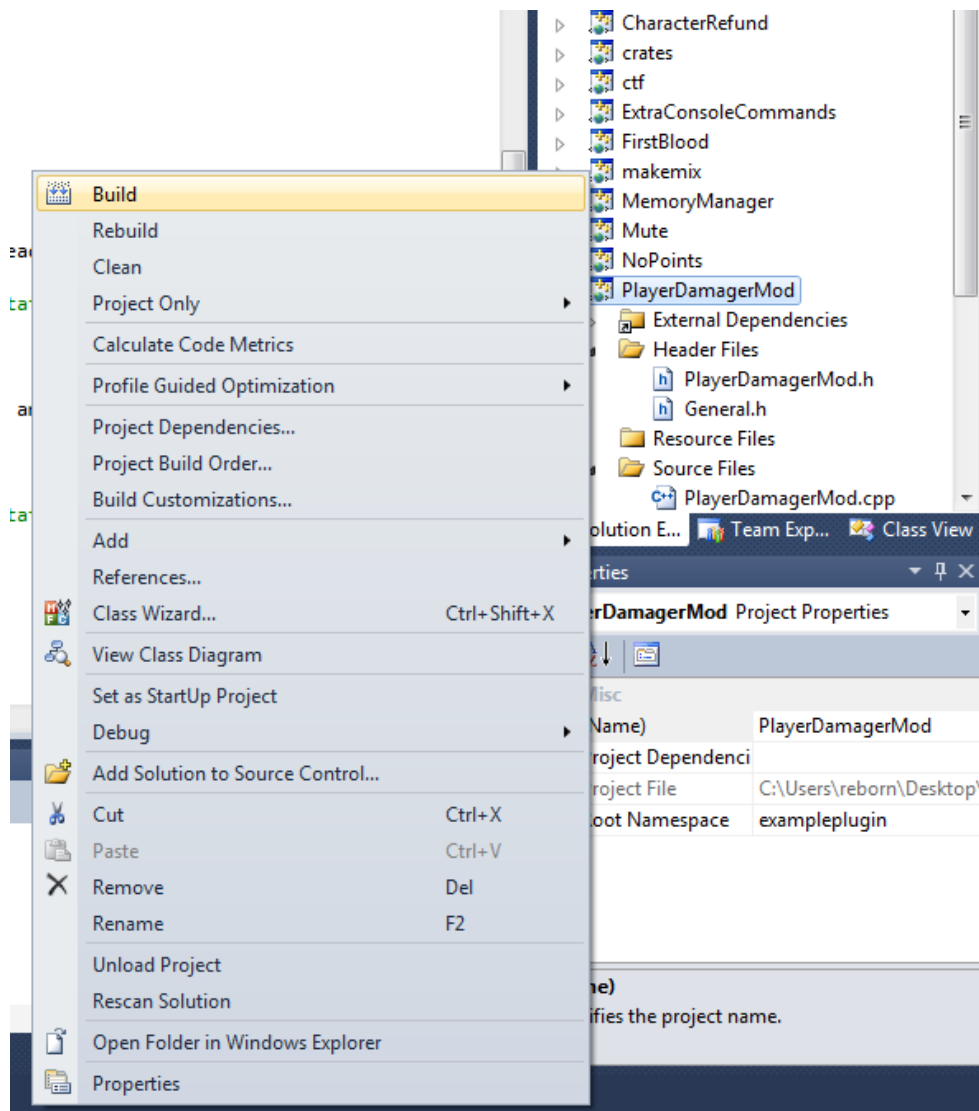
To read like this:

```
void PlayerDamagerMod::OnThink()
{
    //Console_Output(__FUNCTION__ "\n");
}
```

With that done, you are now ready to compile your plug-in. It's ready!

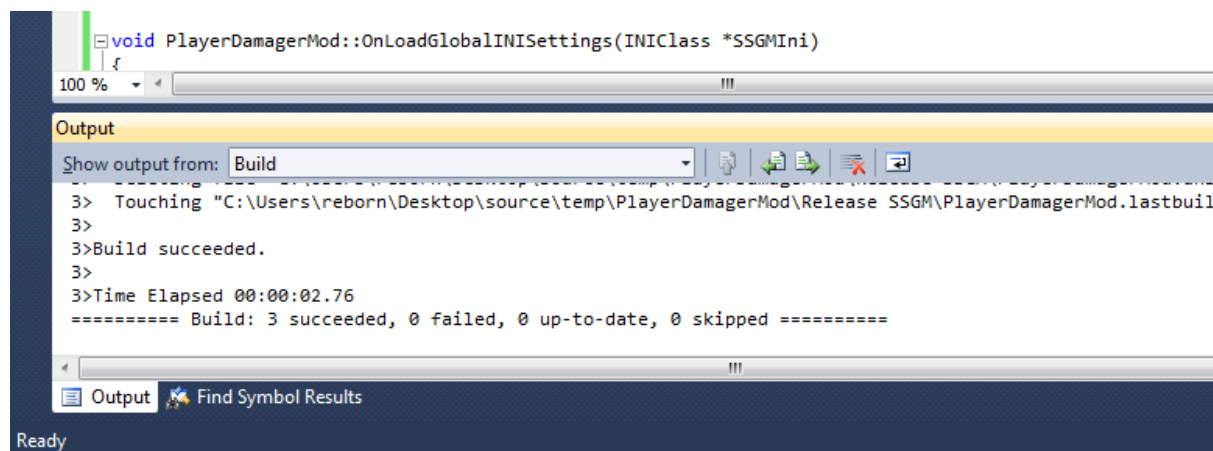
The first time you compile it will take a while. The plug-in makes use of many other files that are not pre-built, so depending on your PC, this could take anywhere between 10 seconds to several minutes for the first time.

I would like to go into linking and compiling in detail, but essentially, the application can take care of this for you, so all you need to do is right click the project file in the solution explorer, and select "build":

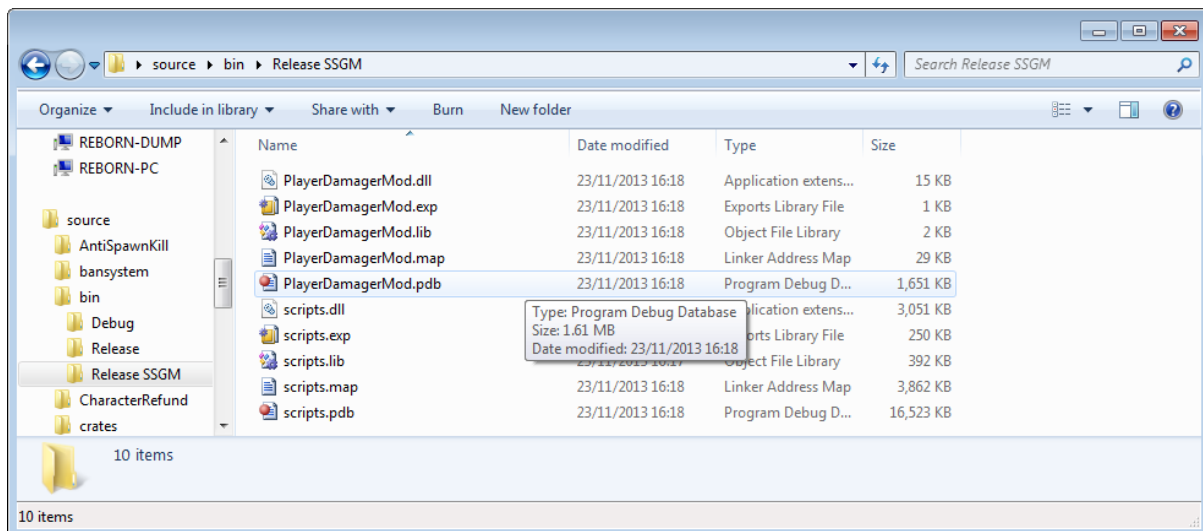


If you have entered everything correctly, you should get a message at the bottom of the application identifying that:

"===== Build: 3 succeeded, 0 failed, 0 up-to-date, 0 skipped ====="



Now the file should be ready! Browse to the folder source\bin\Release SSGM (where ever you placed "source"), and sitting nicely in your folder should be your compiled .dll file:



The next time you “build” should be much quicker. However, if you “build-all” it will take a significantly longer time, as it will build all the projects in the solution (that means all the plug-ins too)!

To install place 'PlayerDamagerMod.dll' inside the root FDS folder and add an entry for it under [Plugins] in SSGM.ini.

You should now load up your server and check it out!

Kind regards,

Spencer 'reborn' Elliott