DXStudio Plugins „Hands On“

by miko (miko@mikoweb.de)

v1.0.1 / 27.09.2010 – 2nd Version “BETA”

1 Introduction 3
2 Terminology 3
3 What does a plugin do? 3
4 Prerequisites 3
4.1 DXStudio 3
4.2 Visual Studio 4
4.3 DirectX SDK 4
4.4 SDK Sample 5
5 Reading/Links 5
6 Basic Layout of a Plugin 6
6.1 The DXStudio Framework 6
6.2 Standard Plugin Structure 7
6.2.1 header.xml 7
6.2.2 myclassscript.h 8
6.2.3 plugin.h/.cpp 8
6.2.4 myclass.h/.cpp 8
6.2.5 thumbnail.jpg 8
6.2.6 buildpack.bat 8
6.3 Compile Procedure 8
6.4 Utilization by the DXStudio Player 8
7 Compiling the SDK sample 9
7.1 Preparing a working project 9
7.2 Adjusting for the first compile 10
7.2.1 Including the DirectX headers and libs 10
7.2.2 Adjust Compiler Output 12
7.2.3 Modifying buildpack.bat 13
7.2.4 Adjust GUID 14
7.3 Compile 15
7.4 Embed and Test 19
7.5 Workflow of the SDK sample 21
7.5.1 Get an Overview 21
7.5.2 Check Property Handlers 22
7.5.3 Check Method Handlers 24
7.5.4 Check Calls of exposed Dll Functions 25
8 Creating your own Plugin Base Project 30
8.1 header xml 30
8.2 buildpack.bat 30

8.3 myclassscript.h 31
8.4 plugin.h/cpp 32
8.5 myclass cpp/h 32
8.6 Cross-Checking 32
9 Writing a Plugin: GetComputerName_Plugin 33
9.1 Usage Target 33
9.2 Basic Preparations 33
9.3 Extending myclass 33
9.4 Adding Handlers 34
9.5 Embed and Test 35

1 Introduction

This step-by-step guide is intended to help you get your hands dirty with DXStudio plugins. It uses many infos that are available on the SDK wiki page – while adding my own experiences. Be aware, though, that it can not cover all the tools' usage in high detail. E.g., when you are new to Visual Studio or C++, there still is a lot of reading ahead.

2 Terminology

Plugins are refered to as "DXEffects" in DXStudio. Compiled plugins are named like, “myEffect.dxeffect” (and essentially are a zipped bunch of files that you add to your DXStudio document or to the …*DX Studio Documents\library\plugins* folder).
Below, I'll be using both terms, *plugin* and *dxeffect* alike.

3 What does a plugin do?

A plugin/dxeffect provides a dll and resources that can be accessed by a DXStudio document via the DXStudio player. As a lot of interface possibilities are provided, one can access DirectX and the Javascript engine inside DXStudio – aswell as Windows routines and a lot of other things. This way, options to extend the DXStudio functionality are vast.

4 Prerequisites

To create a plugin/dxeffect, we need

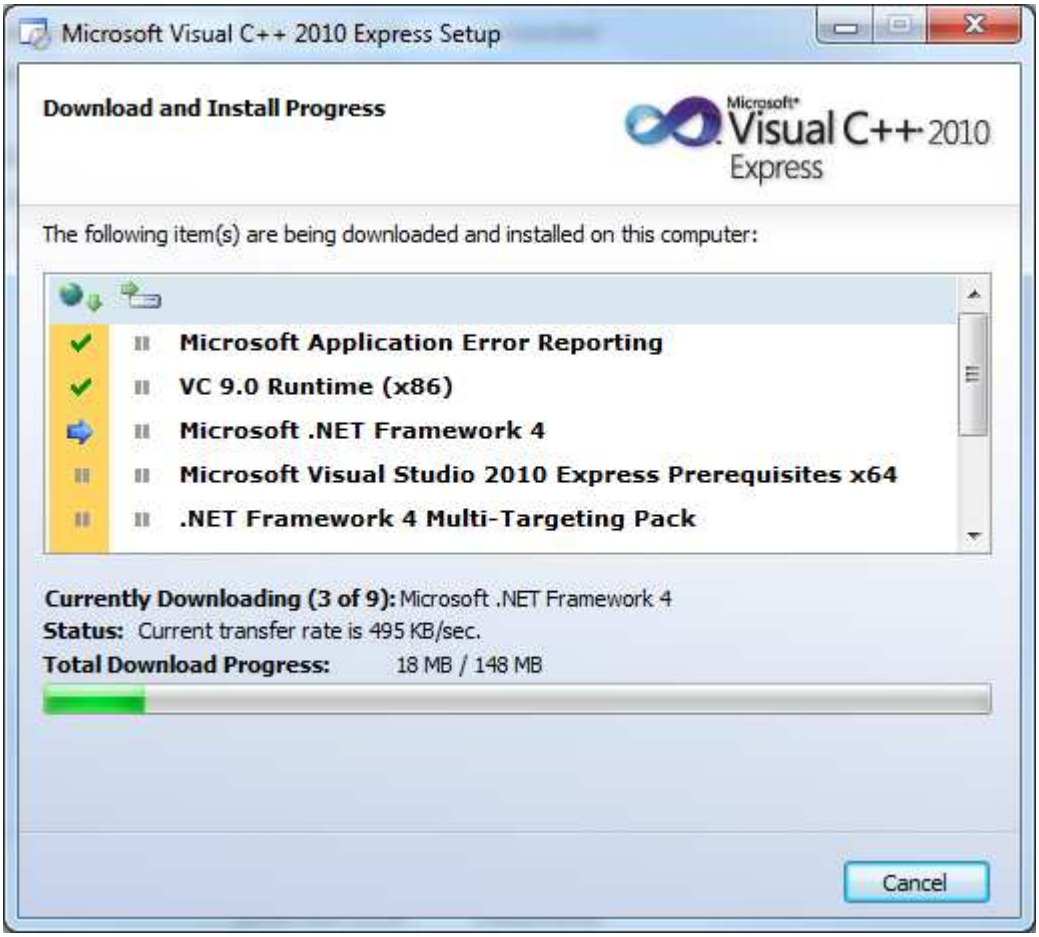
- DXStudio (ha, whoda thunk?)
- A C++ compiler (using free Visual Studio Express here)
- Microsoft's DirectX SDK
- Worldweaver's SDK sample as a starting point

4.1 DXStudio

DXStudio v3.2.47 or later is needed to use a plugin made with the SDK. At the time of this writing, v3.2.71 is available. Download the most recent version here:
http://www.dxstudio.com/
Actually, for just *creating* a plugin, one would not need the editor. To make a DXStudio doc and embed the plugin for using and testing though, you would.
Note that your custom plugins will work with all versions of DXStudio (Freeware/Std/Pro).

4.2 Visual Studio

Check for the newest Microsoft Visual Studio 2010 Express "C++" version here:
http://www.microsoft.com/express/Downloads/
Dowload and install.



Downloading and installing takes a while ;-)

After having installed VS2010Express, you might want to run WindowsUpdate and let it check for new files.

4.3 DirectX SDK

As DXStudio is linked against the DirectX November2008 SDK, you are strongly advised to get this one, too. Note that this does not mean that you have to use "old" DirectX drivers/runtime – just the SDK.
Actually, I had a lot of inconvenience when trying to use other/newer SDK versions. This is mainly because the DirectX helper file *d3dx9_40.dll*: It is delivered with DXStudio and when using Nov2008 SDK, your plugin links to that, too.
With newer SDKs, the index number - here: 40 - is counting up, and you will have to deliver a corresponding dll with your plugin (path finding issues ahead!) – or ensure it is present on the users' machines (force upgrade DirectX version).

Download and install the DirectX SDK November 2008 here:
http://www.microsoft.com/downloads/details.aspx?FamilyId=5493F76A-6D37-478D-BA17-28B1CCA4865A&displaylang=en

When firing up VS2010 after having installed the SDK, it might pause a while to integrate the SDK help. This (usually) happens only once, though.

4.4 SDK Sample

Worldweaver's *DXStudio Player SDK Sample* is available on the SDK Wiki page.
Download it here and store the zipped file for later use:
http://www.dxstudio.com/downloads/PlayerSDK_4_1_0.zip

It contains a visual Visual Studio 2008 project that we will use with VS2010 Express later on. Once you progress with plugin development, I'd strongly recommend you create your own custom "starter project" from the download. E.g. I removed the demo code and added some memory checks and definitions that I'm going to use in all further projects. Then, for a new plugin, just copy the base structure over and start…
The nice thing is that this download contains the includes of the JSAPI aswell. This is sort of an SDK that is needed to connect the C++ environment with the Javascript Engine.

5 Reading/Links

Listed here, some sources of informations that could be helpful.

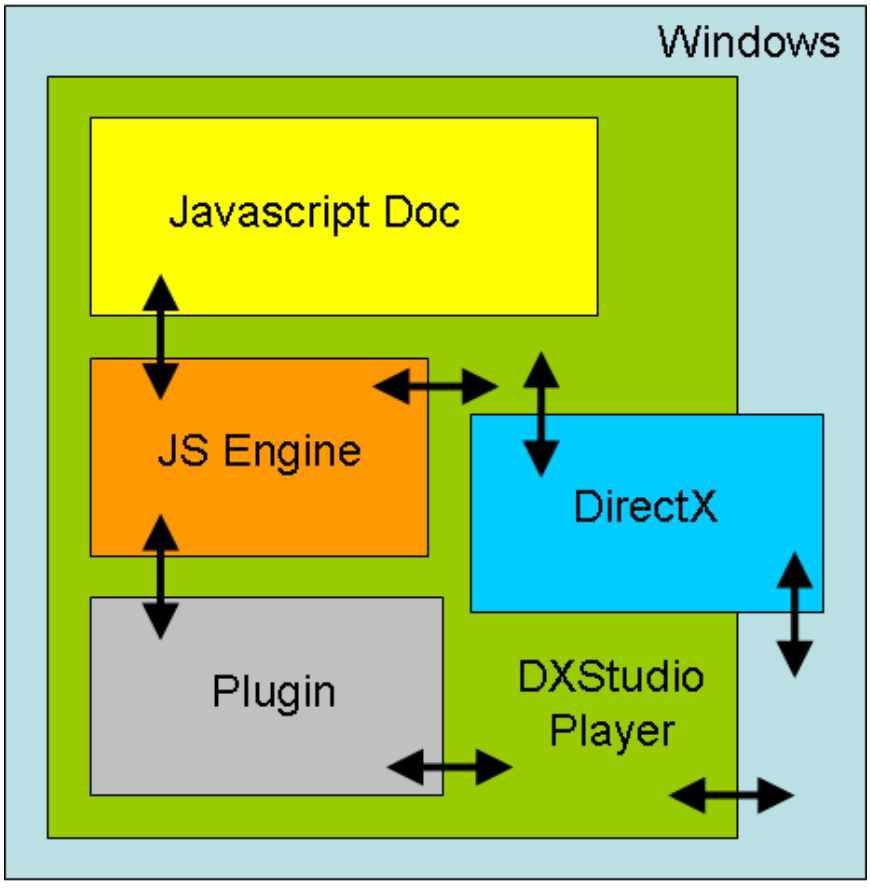
- DXStudio SDK Wiki
  http://www.dxstudio.com/guide.aspx?id=f84db734-b5e6-403c-acd7-9fb96cb30cde
- SDK Forum at the DXStudio boards
  http://www.dxstudio.com/forumbrowse.aspx?forumid=b6e6c10a-a92b-49c2-aaa8-57171531a316
- Tracemonkey (Spidermonkey), the JS engine used in the player
  https://wiki.mozilla.org/JavaScript:TraceMonkey
  http://www.mozilla.org/js/spidermonkey/
- Mozilla's JSAPI reference, for easy lookup of functions
  https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference

6 Basic Layout of a Plugin

Before actually starting, lets talk about the build-up and work flow of a plugin. You don’t need to do anything yet – just read. The “step-by-step” procedure will start in the next chapter.

6.1 The DXStudio Framework

This is how I imagine the interconnections between the different “main components” of the DXStudio player. It might not entirely be correct in technical terms, but helped me a lot when trying to get a grasp of how this all works.



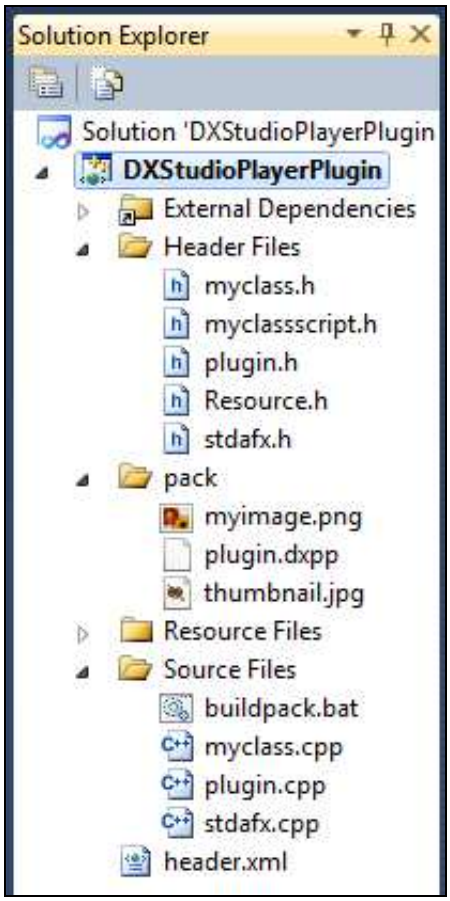
As we know, DXStudio’s “native” user language is Javascript. That is what we write our DXStudio document in. For handling the Javascript code, a JS engine is being used. This one – by itself – is written in C++, as well as the underlying DXStudio code.

So, when you e.g. add *scenes.scene_1.doSomething();* to your doc, what does happen upon execution in the player? (I’m just making this up as an example)
First, the javascript engine will parse through the script, then “route” the function call to *doSomething()* to a C++ function. Inside this function then, the actual action happens. Like, set some DirectX parameters, show an object, whatever.

The relationships and interactions can grow quite complex, but this example might do for a basic understanding. When writing plugins, remember that there are several “blocks” (JS engine, DirectX, Windows) that you need to communicate with.

6.2 Standard Plugin Structure

The structure as given in the SDK sample project looks like this (shown here as visible in the VS Solution Explorer). Below, some more details about the “standard” components/files.



6.2.1 header.xml

This is the “information file” that holds general data about the plugin. That would be things like

- Version of the effect
- Copyrights, description
- Unique Identification number
- Embed infos (like, “only for 2D scenes”)
- Execution infos (like, “call on update or on rendering”)
- Available properties and methods info and help text for the DXStudio editor

6.2.2 myclassscript.h

This file represents the “interfacing” between the Javascript Engine and your C++ plugin code. It tells what properties and methods are available and what code should be executed when e.g. a property is changed or a method of your dxeffect is being called.
E.g., a call to *scene.effects.myeffect.doSomething()* would first “hit” myclasscript on execution. There, you have to deal with that call then.

6.2.3 plugin.h/.cpp

Here, the standard plugin functions reside. Exported dll functions that are exposed show up here (e.g. *PLUG_API BOOL GetDLLInfo(…),* as well as data definitions of some (very) helpful structures (e.g. *SPlayerInfo* to access certain application members and data).
The exposed dll functions are automatically called by the DXStudio player (e.g., *PLUG_API BOOL DocumentLoaded(…)* is called when ..well.. the document is loaded).

6.2.4 myclass.h/.cpp

This is the class where (probably) most of your own code would be inserted. Usually, its functions are either called from *myclassscript.h* (as a response to a script call) or from *plugin.cpp*, when handling a ‘standard call’ like *onUpdate(…)*.
Look at *PLUG_API BOOL DocumentLoaded(…)* in *plugin.cpp*. This is called by the player, then by itself calls *myclass.DocumentLoaded()* – which routes to *BOOL CMyClass::DocumentLoaded()* – and here you’ll do some magic.

6.2.5 thumbnail.jpg

The thumbnail picture for the plugin, as defined in *header.xml*.

6.2.6 buildpack.bat

A helper batchfile that creates (actually, zips) the plugin from compiled code and resources and copies it over to your local plugin directory. You probably need to adjust the paths in there to make it work on your pc. We’ll do that below.

6.3 Compile Procedure

When compiling your plugin , the actual created dll (named *plugin.dxpp* here) is being placed into the *…PlayerSDK_4_1_0\Player\DXStudioPlayerPluginSample\pack* directory. There, other resources you need (like the thumbnail picture and *header.xml*) must reside aswell.
After that, the whole bunch is zipped by *buildpack.bat* and copied over to your …DX Studio Documents\library\plugins\ folder. Now, it will show up in the DXStudio effects list and you can add your new dxeffect via the DXStudio editor.

6.4 Utilization by the DXStudio Player

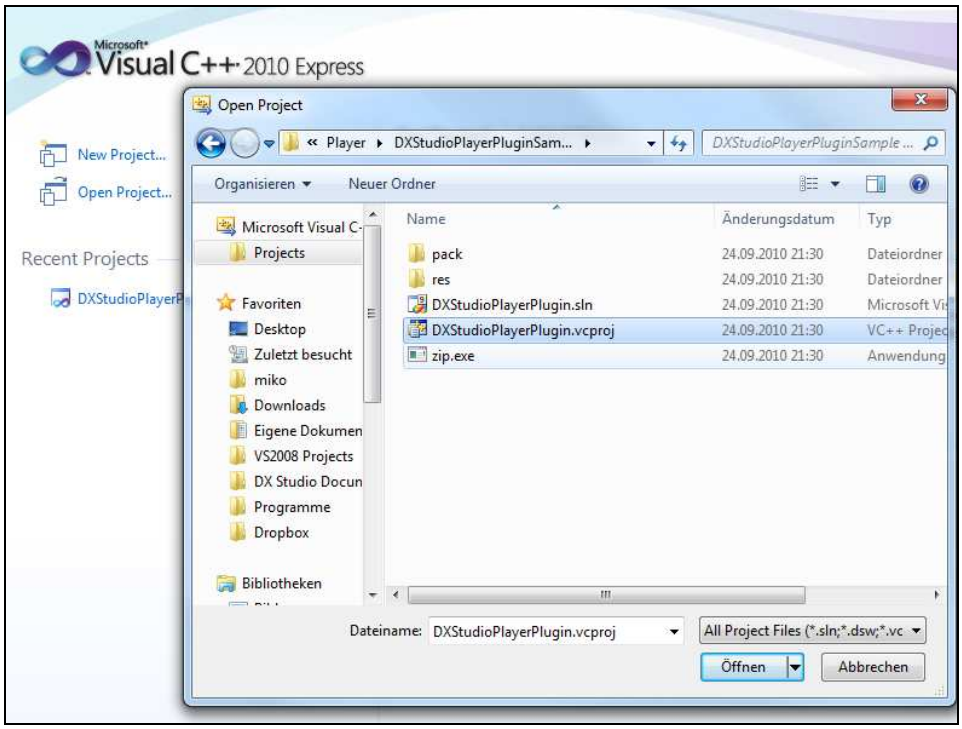
When a dxeffect is embedded into a DXStudio doc and the player runs this doc, it unzips the plugin to a temp directory. Then, the informations in *header.xml* are being used to determine how to call into the plugin’s dll (*plugin.dxpp* here). And in there, your added custom actions happen.

7 Compiling the SDK sample

Now, after all that theory mumbling, lets proceed to actual action! First, we'll prepare for compiling the SDK sample doc, then create it, test it and do a bit of deeper analyzing what happens inside.
Note that after importing the given SDK sample into VS2010, some adjustments need to be done. When using VS2008, there is a lot less to do.

7.1 Preparing a working project

First, locate the (already downloaded) zipped SDK sample (*PlayerSDK_4_1_0.zip*) and unzip it to a directory of your choice (I use *C:\Users\miko\Documents\Visual Studio 2010\Projects*).
Now, fire up Visual Studio Express, click on "Open Project…" and select the *.vcproj on the plugin *project's PlayerSDK_4_1_0\Player\DXStudioPlayerPluginSample* folder.



This will bring up the Conversion Wizard, as the project is VS2008, and we are using 2010 here. Press "Finish", confirm a Security Warning, and you should see the "Conversion Complete" message.
Once you expand the project's view in the Solution Explorer to the left, you'll see the different project files.

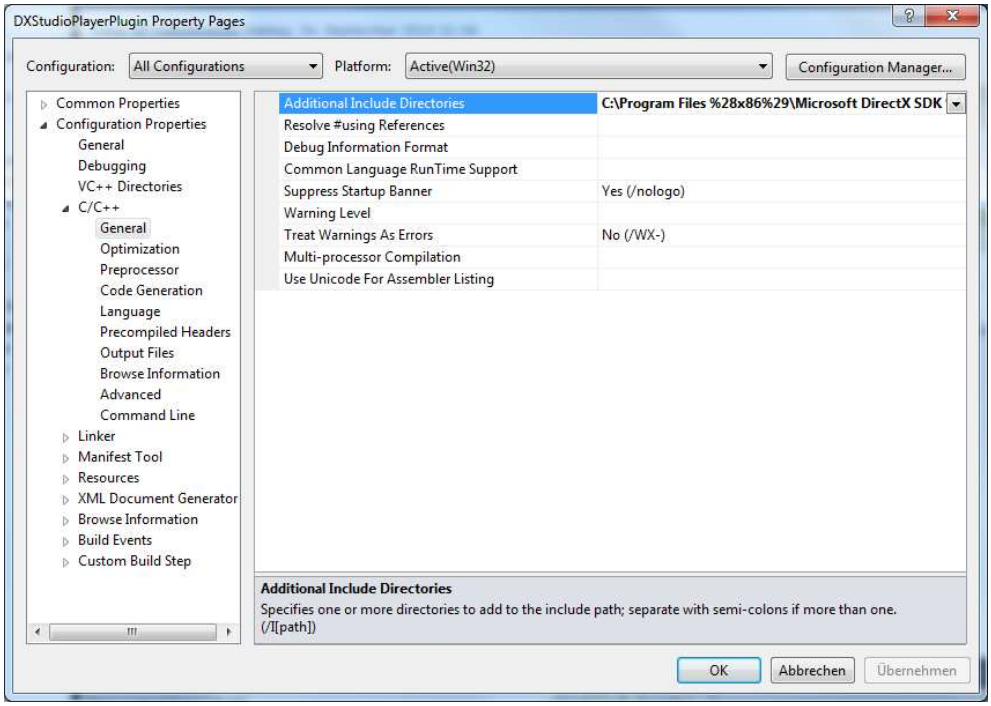
7.2 Adjusting for the first compile

After having successfully imported the SDK sample project into VS2010, we need to adjust some of the presets that are in there. Creation of your very first plugin will not succeed otherwise.

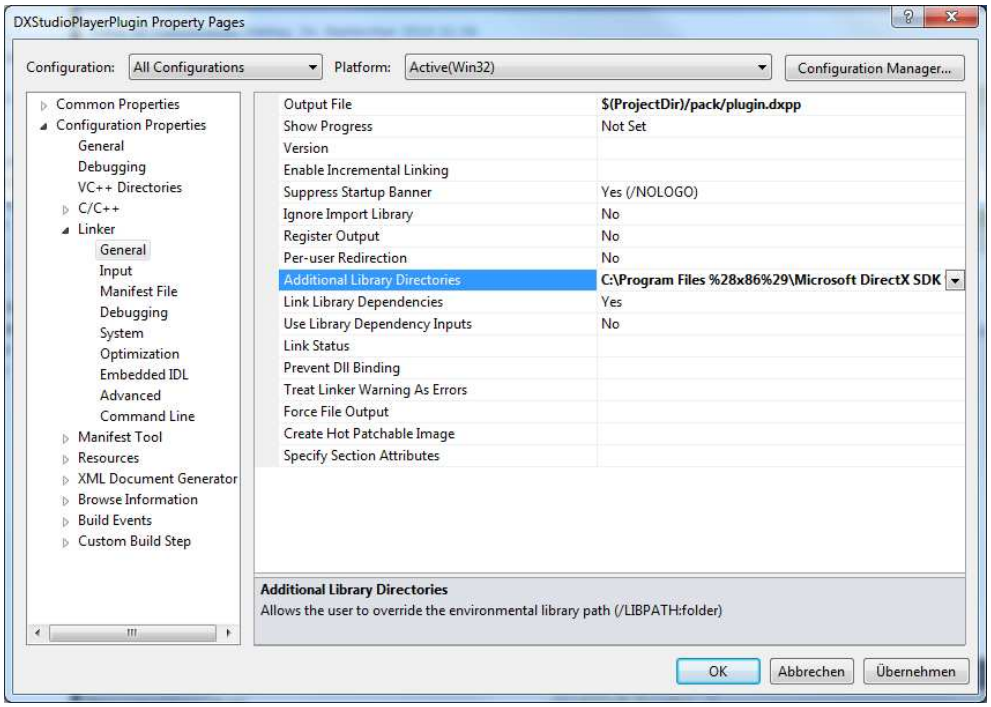
7.2.1 Including the DirectX headers and libs

Right-click on *DXStudioPlayerPlugin* in the Solution Explorer and bring up the properties page. Adjust Configuration to “All Configurations” (so that the changes apply to Debug and Release build alike).

In *C/C++/General*, add the path to the DirectX SDK’s include folder as an additional include directory. In my case, this is looking like *C:\Program Files %28x86%29\Microsoft DirectX SDK %28November 2008%29\Include*



In Linker/General, add the path to the DirectX SDK’s lib(32) folder as an additional include directory. In my case, this is looking like *C:\Program Files %28x86%29\Microsoft DirectX SDK %28November 2008%29\Lib\x86*.

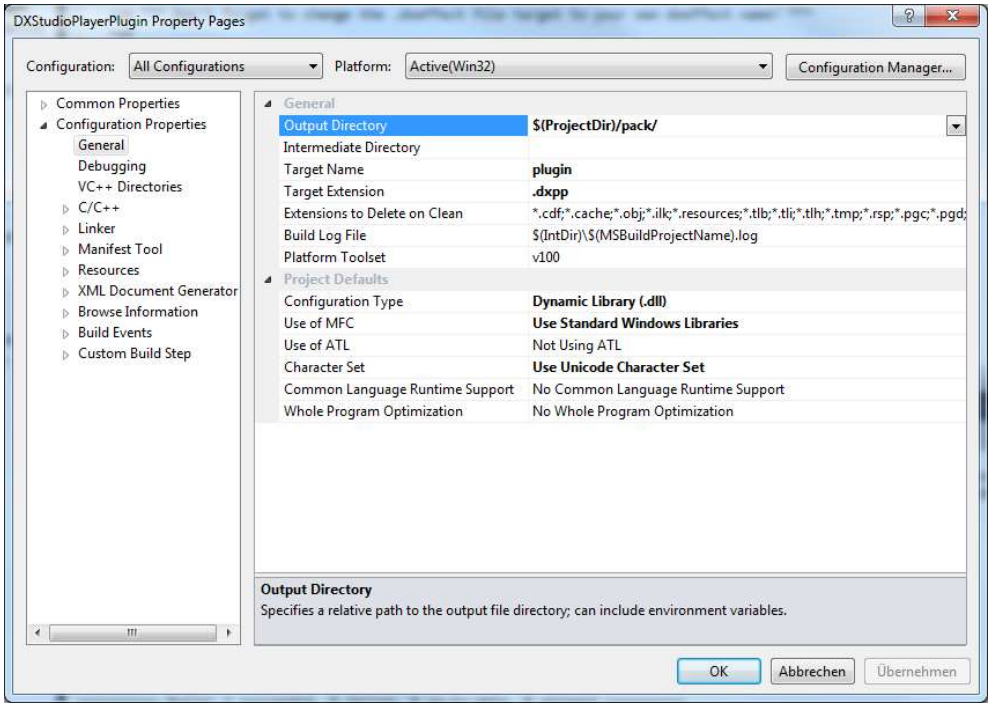


With these changes, the compiler should be able to find the needed DirectX headers and libs.

7.2.2 Adjust Compiler Output

Right-click on *DXStudioPlayerPlugin* in the Solution Explorer and bring up the properties page. Adjust Configuration to “All Configurations” (so that the changes apply to Debug and Release build alike).

In *General*, change the path of the *Output Directory* to *$(ProjectDir)/pack/*. Also, change the *Target Name* to *plugin* and the *Target Extension* to *.dxpp* (note the dot before dxpp).



7.2.3 Modifying buildpack.bat

In buildpack.bat, check for the line

copy ..\temp.zip "%userprofile%\My Documents\DX Studio Documents\library\plugins\sample.dxeffect“

and adjust the path to your local DXStudio’s plugin folder. This is where the plugin will be copied after compilation (and under the name given here).

Also, I found it convenient to have removed some output that is stored to the *pack* folder but not needed for the plugin afterwards. So, I changed the *buildpack.bat* from

```
@echo off
@echo Building effect...
cd pack
..\zip ..\temp.zip *.*
copy ..\temp.zip "C:\Users\MiKo\Documents\DX Studio Documents\library\plugins\sample.dxeffect"
del *.pdb
del *.ilk
del ..\temp.zip
cd..
@echo Done.
```

To

```
@echo off
@echo Building effect...
cd pack
del *.pdb
del *.ilk
del *.exp
del *.lib
..\zip ..\temp.zip *.*
copy ..\temp.zip "C:\Users\MiKo\Documents\DX Studio Documents\library\plugins\sample.dxeffect"
del ..\temp.zip
cd..
@echo Done.
```

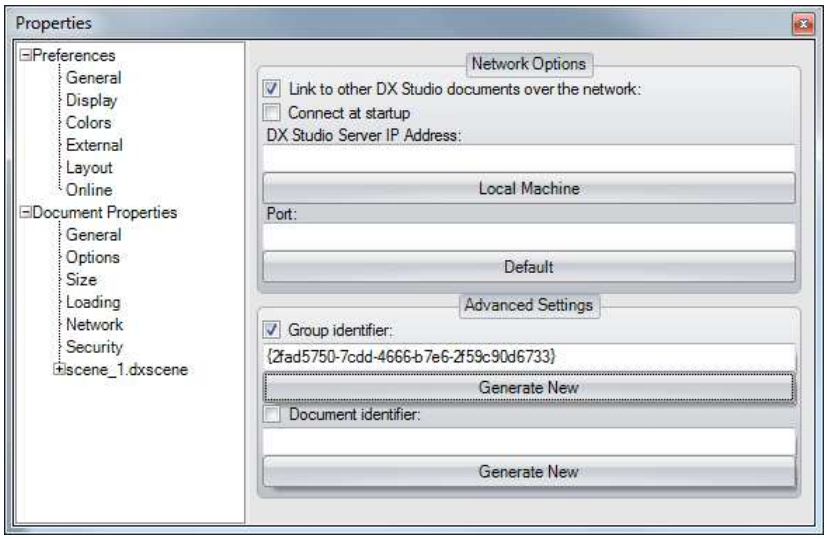
7.2.4 Adjust GUID

In *header.xml*, you will notice a line like

```
<version id="{b2bda7a1-9a9f-4d0f-99d0-f7ba5d4ba007}" friendlyversion="1.0.5" />
```

The version id given here must be unique for each plugin. Although not needed for compilation of “sample.dxeffect”, we’ll change the id here for training purposes.#

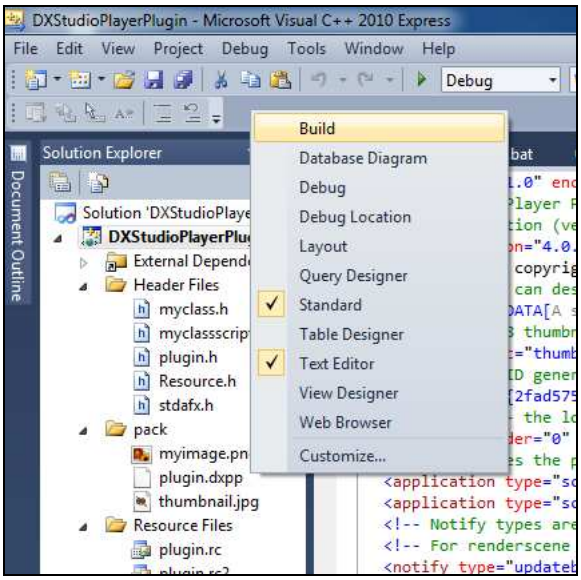
Fire up an empty DXStudio document, go to *Properties* and create a *Group Identifier* inside the *Network Options*. Copy the numbers inside the { } brackets and paste into the id definition in *header.xml*.



Rinse and save the VS project (well, just save).

7.3 Compile

To ease up building the project in VS2010, you'd better activate the Build toolbar. To do so, right-click in the toolbars section and activate Build.

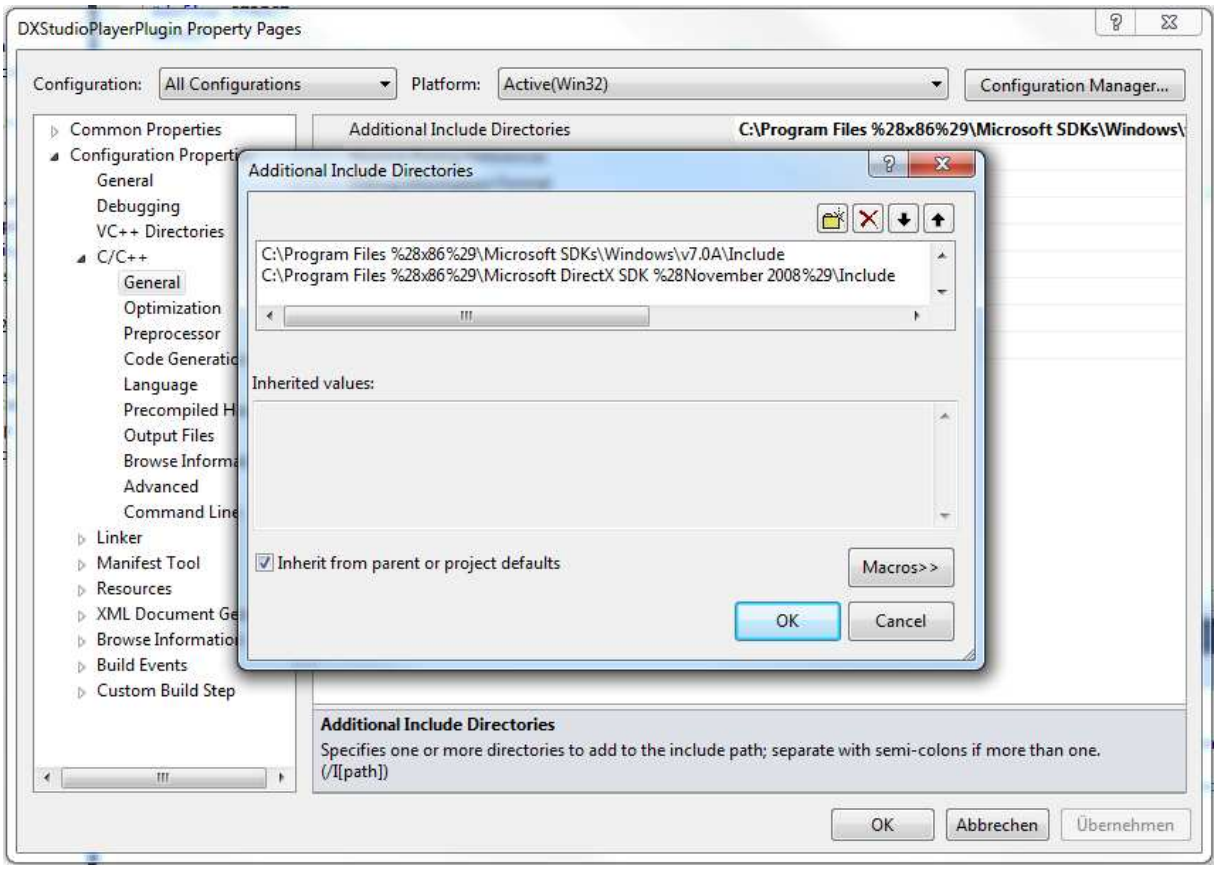


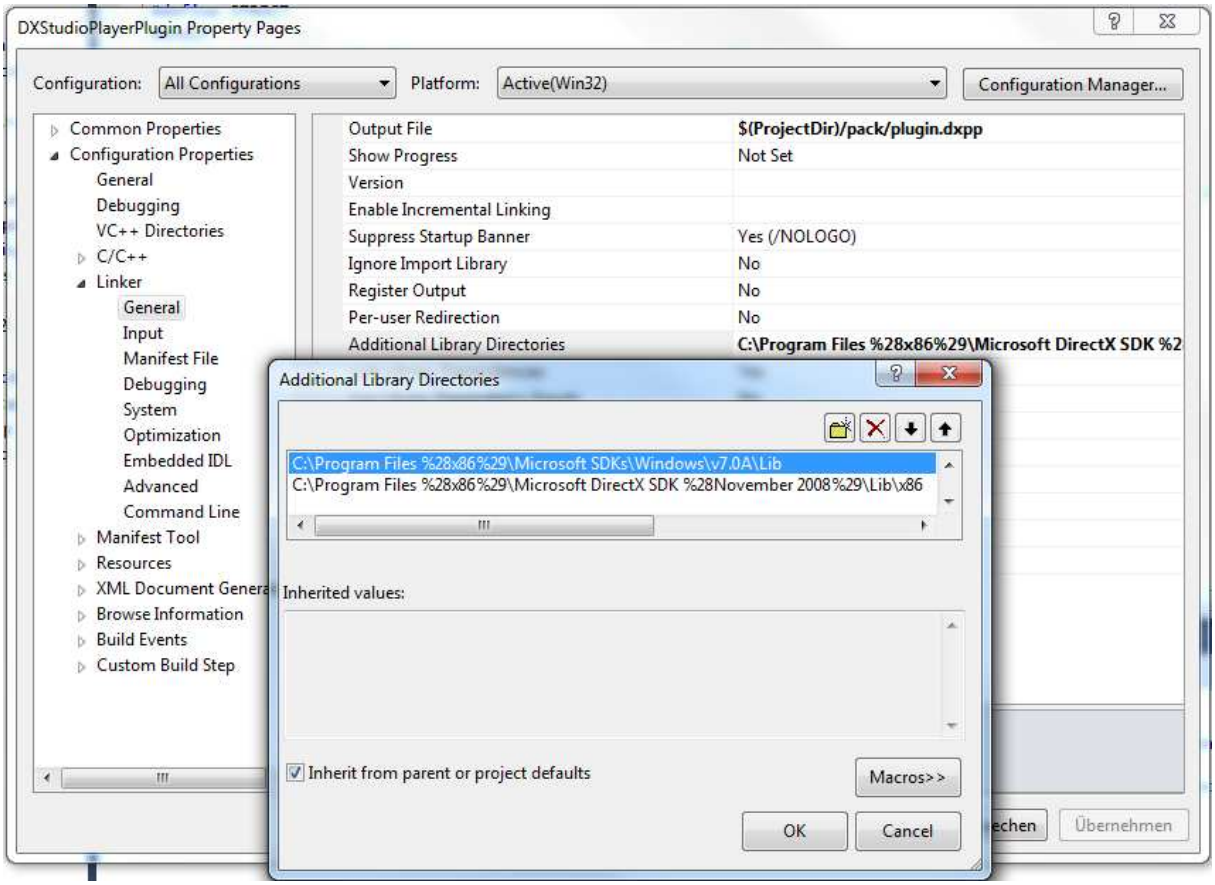
Actually (try to) compile by pressing the "Build DXStudioPlayerPlugin" icon.

Does the compile succeed? If not, check for these possible errors, which occurred during test runs:

```
c:\program files (x86)\microsoft sdks\windows\v7.0a\include\objidl.h(11280): error C2061:
syntax error : identifier '__RPC__out_xcount_part'
```

To solve this, you must include the Widows v7 headers and libs *before* the SDK headers/libs. So do as shown above (7.2.1), add the Windows SDK references and ensure they are ON TOP of the list.






```
plugin.rc(10): fatal error RC1015: cannot open include file 'afxres.h'
```

Right-click the file plugin.rc in the Solution Explorer, select View Code.
In the code, replace all occurrences of `"afxres.h"` with `"windows.h"` (there should be two of them).

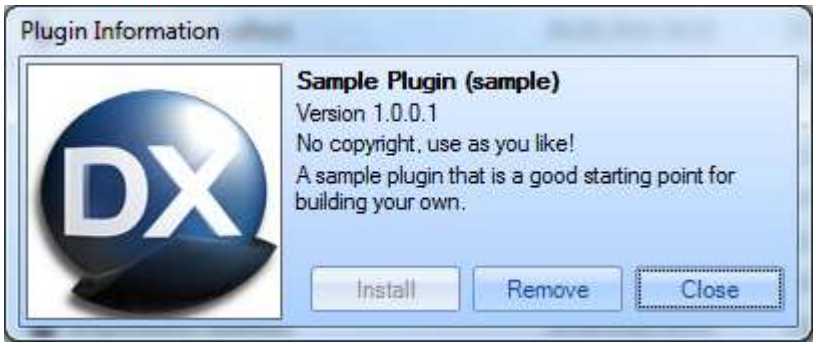
```
plugin.rc(114): fatal error RC1015: cannot open include file 'afxres.rc'.
```

Right-click the file plugin.rc in the Solution Explorer, select View Code.
In the code, comment out all occurrences of `" #include "afxres.rc"` (there should be two of them).

Finally, the Output Window should show something like this:

```
1>------ Build started: Project: DXStudioPlayerPlugin, Configuration: Debug Win32 ------
1>     Creating library C:\Users\MiKo\Documents\Visual Studio
2010\Projects\PlayerSDK_4_1_0\Player\DXStudioPlayerPluginSample\/pack/Plug_In_1.lib and
object C:\Users\MiKo\Documents\Visual Studio
2010\Projects\PlayerSDK_4_1_0\Player\DXStudioPlayerPluginSample\/pack/Plug_In_1.exp
1>  DXStudioPlayerPlugin.vcxproj -> C:\Users\MiKo\Documents\Visual Studio
2010\Projects\PlayerSDK_4_1_0\Player\DXStudioPlayerPluginSample\/pack/plugin.dxpp
1>  Building effect...
1>    adding: header.xml (164 bytes security) (deflated 56%)
1>    adding: myimage.png (164 bytes security) (deflated 2%)
1>    adding: plugin.dxpp (164 bytes security) (deflated 68%)
1>    adding: thumbnail.jpg (164 bytes security) (deflated 8%)
1>          1 Datei(en) kopiert.
1>  Done.
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

To check for the newly built plugin, navigate to your …*DX Studio Documents\library\plugins* folder, locate the *sample.dxeffect* file (as this is what we named it in *buildpack.bat*). Double-click it to bring up the DXStudio DXEffect info.

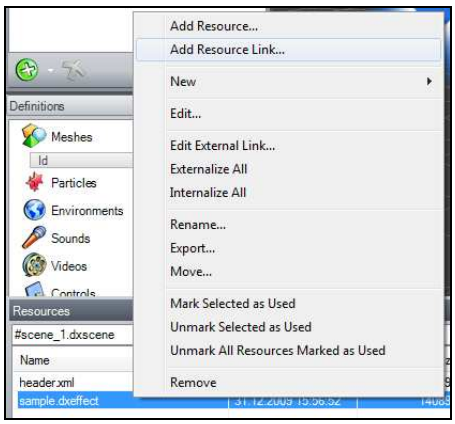


7.4 Embed and Test

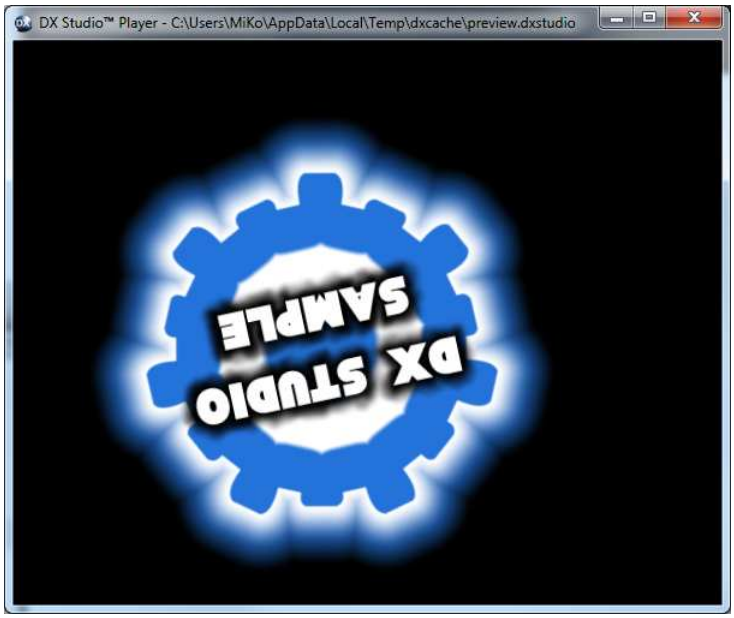
Start DXStudio, and create a new document with a 2D scene. Add the new DXEffect to the scene via the Document Properties.

Tip

Note that after having added the DXEffect to the scene, you can change it from “embedded” to “resource link”. This makes life a lot easier when developing plugins, as each newly compiled version is directly available in the document. Just don’t forget to change this before document release.



When starting the document (actually, in the editor already as well), you'll see the demo's spinning bitmap.



7.5 Workflow of the SDK sample

After having compiled the SDK sample, lets take a look under the hood and see what exactly it does – and how.

As said above, there are mainly two ways to carry out an action inside a plugin:

- Calling a property or method from the Javascript code
  (like *scene.effects.myeffect.doSomething()*)
  or
- Handling one of the automated calls into exposed functions (like *DocumentLoaded()*)

7.5.1 Get an Overview

First, lets check *header.xml*, to get an impression what a plugin is about.

```
  <application type="scene3d" />
  <application type="scene2d" />
```

So, this plugin can be used in 2D and 3D scenes. Good to know. Depending on that data, a plugin will (or will not) show up in the selection list when adding DXEffects to a document.

```
  <notify type="updatebegin"/>
  <notify type="updateend"/>
  <notify type="renderprepare"/>
  <notify type="renderbegin"/>
  <notify type="renderend"/>
  <notify type="renderscenebegin" passes="0"/>
  <notify type="rendersceneend" passes="0"/>
  <notify type="rendersceneobject" passes="0"/>
```

These are the different exposed plugin dll routines that will be periodically called. E.g., the plugin will be called upon *UpdateBegin*, *UpdateEnd*, etc.
Remember? The routines that will be called reside in *plugin.cpp* – and there, we can add our own stuff. We'll look into that in detail later on.

```
<method id="reset" help="reset//A sample function called 'reset' that ..."/>
<property id="speed" type="float" help="speed//A float multiplier ..." default="1.0"/>
```

Aha, the plugin exposes a method called "reset" (without arguments, as it seems) and a property called "speed" (which is a float). So, we can expect to see handling of those two in *myclassscript.h*. Again, we will check there what is going on (see below).

7.5.2 Check Property Handlers

Taking a closer look at *myclassscript.h* now.

The properties being handled by the the plugin must be known to the Javascript engine. To do so, an array called *myclass_props* is filled with data. In the sample, this looks like

```
static JSPropertySpec myclass_props[] = {
  {"enable",  0,  JSPROP_ENUMERATE, myclass_getEnable, myclass_setEnable},
  {"url",     1,  JSPROP_ENUMERATE, myclass_getURL, myclass_setURL},
  {"width",   2,  JSPROP_ENUMERATE, myclass_getWidth, myclass_setWidth},
  {"speed",   3,  JSPROP_ENUMERATE, myclass_getSpeed, myclass_setSpeed},
  {0}
};
```

Note that – after reviewing the *header.xml* - we were only expecting to see “speed” here. For educational purposes, the demo contains some more. We’ll only look at “speed” here.

```
  {"speed",   3,      JSPROP_ENUMERATE, myclass_getSpeed, myclass_setSpeed},
```

Each line contains the name of the property (as it is known to the JS engine), a counting number, a constant, the C++ function to be called when the property is SET and the C++ function when it is GET (retrieved).
So, when you have something like *scene.effects.myEffect.speed = 5;* in your doc’s code (this is SET), inside the plugin, *myclass_setSpeed(…)* is being called.
Likewise, for *var a=scene.effects.myEffect.speed,* the *myclass_getSpeed(…)* is invoked.

myclass_getSpeed and *myclass_getSpeed* are both located in *myclassscript.h*, too. All these getter/setter functions need to have the same calling conventions. Parameters (passed in by the DXStudio framework) are the Javascript Context (pointer to access the Javascript engine), an Object (pointer to the Javascript object we are working on), an id number and a pointer to a Javascript value. The last one is where put in (or retrieve) the value that will be passed over to DXStudio script.

```
static JSBool myclass_getSpeed(JSContext *cx, JSObject *obj, jsval id, jsval *vp)
```

For *myclass_getSpeed*, the handler code looks like this:

```
(1)  CMyClass* myclass=(CMyClass*) JS_GetPrivate(cx, obj);
(2)  double val=(double) myclass->speed;
(3)  jsval thisval;
(4)  JS_NewNumberValue(cx,val,&thisval);
(5)  *vp=thisval;
(6)  return JS_TRUE;
```

(1) This is an automatism to get a pointer to the *CMyClass* object (defined in *myclass.h/.cpp*). We need this to access our own routines.
(2) Get the current value *speed* from our class (that we have the pointer to now)
(3,4) Create a new variable inside the Javascript engine to return. Remember the call like *var a=scene.effects.myEffect.speed; ?* The variable “a” must be created somewhere to be returned – and we do this here.
(5) Fill in the newly created variable to be returned to Javascript.
(6) Return TRUE (always)

For *myclass_setSpeed*, the handler code looks like this (rather similar to the GETter):

```
     CMyClass* myclass=(CMyClass*) JS_GetPrivate(cx, obj);
(1)  jsdouble dbl;
(2)  JS_ValueToNumber(cx, *vp, &dbl);
(3)  myclass->speed = (FLOAT) dbl;
(4)  myclass->WriteConsoleString(L"Speed set");
     return JS_TRUE;
```

(1,2) Convert the passed in value to a number (as it could be anything; imagine *scene.effects.myEffect.speed = “huh”;*)
(3) Set the new value in our custom code
(4) Use a helper routine in our code to send some text to the DXStudio player’s console

7.5.3 Check Method Handlers

Continuing with *myclassscript.h.*

The methods definitions are very similar to those of the properties. Again, there is an array – this time, named myclass_methods. In the demo, it looks like

```
static JSFunctionSpec myclass_methods[] = {
  {"reset",  myclass_reset,  0},
  {0}
};
```

There is only one method defined here. There is the name of the method (as known to the JS engine, the C++ function to be called, and the number of arguments to expect (none in this case).

Again, there is a common calling convention for all C++ functions that do handle a Javascript method call. It looks like

```
static JSBool myclass_reset(JSContext *cx, JSObject *obj, uintN argc, jsval *argv,
jsval *rval)
```

Parameters (passed in by the DXStudio framework) are the Javascript Context (pointer to access the Javascript engine), an Object (pointer to the Javascript object we are working on), the number of arguments passed in, pointer to a list of arguments.

The handler code is rather straightforward then.

```
(1)  CMyClass* myclass=(CMyClass*) JS_GetPrivate(cx, obj);
(2)  myclass->WriteConsoleString(L"Reset called!");
(3)  myclass->speed=1.0f;
(4)  *rval = BOOLEAN_TO_JSVAL(JS_TRUE);
(5)  return JS_TRUE;
```

(1) This is an automatism to get a pointer to the *CMyClass* object (defined in *myclass.h/.cpp*). We need this to access our own routines.
(2) Use a helper routine in our code to send some text to the DXStudio player’s console
(3) Just execute in our code whatever needs to be done to “reset”
(4) Place a result to return to the caller (this will be given back by the Javascript call to e.g. *var bRet = scene.effects.myEffect.reset*)
(5) Return TRUE (always)

7.5.4 Check Calls of exposed Dll Functions

Lets move on to *plugin.h/.cpp*.
In the chapters before, we did already see how the methods and properties handles of the sample plugin look like. Now, we're going to review calls to the exposed dll functions.

7.5.4.1 DeviceLost

```
PLUG_API BOOL DeviceLost(DWORD instancehandle)
```

This is called by the player when the DirectX device received status "lost" (maybe, it was being minimized?), to give you a possibility to react accordingly on your own DirectX objects. In here, *myclass.DeviceLost()* is being called.

```
void CMyClass::DeviceLost()
{
  if (mysprite) mysprite->OnLostDevice();
  lostdevice=TRUE;
}
```

As the plugin contains a D3DXSPRITE ("mysprite"), the method *OnLostDevice()* is just being called on this. That way, DirectX can do on this object whatever it needs, when a device gets lost. Additionally, an internal flag is set to remember that the current device is lost.

7.5.4.2 DeviceRestore

As for *DeviceLost*, we see similar behaviour for a device being restored again (like, when you e.g. maximize the player window from minimized state).

```
PLUG_API BOOL DeviceRestore(DWORD instancehandle)
```

This calls *myclass.DeviceRestore().*

```
void CMyClass::DeviceRestore()
{
  if (mysprite) mysprite->OnResetDevice();
  lostdevice=FALSE;
}
```

Here, OnResetDevice() is being called on the D3DXSPRITE ("mysprite"), and the internal flag is reset – showing that the DirectX device is (should be) operational again.

7.5.4.3 DocumentLoaded

This function is called once when the DXStudio document has been loaded into the player. Here is a good place to initialize your own objects, as the player should already be set up (like, DirectX device created, helper structures filled, etc.).

```
PLUG_API BOOL DocumentLoaded(DWORD instancehandle)
```

This calls *myclass.DocumentLoaded()*.

```
     BOOL CMyClass::DocumentLoaded()
     {
(1)   WCHAR pathbuf[MAX_PATH];
      wsprintf(pathbuf, L"%s%s", g_pInfo->plugincachefolder, L"myimage.png");
      if( FAILED( D3DXCreateTextureFromFile( g_pInfo->pDevice,
                                             pathbuf,
                                             &mytexture ) ) )
      {
       MessageBox(NULL,
                  L"Could not find 'myimage.png'",
                  L"Effect Compilation Error", MB_OK);
      }
      D3DXCreateSprite(g_pInfo->pDevice, &mysprite);

(2)   WriteConsoleString(L"Document loaded function called!");

(3)   EnterCriticalSection(g_pInfo->pJSCS);

(4)   CDXVector* v = new CDXVector(10,10,10);

(5)   JSObject* vobj=JS_NewObject(g_pInfo->pJSContext,
                                  g_pInfo->pJSC_Vector,
                                  NULL, NULL);

(6)   jsval rval;
      const WCHAR* script = L"system.timer";
      const char* filename = "[plugin script]";
      uintN lineno=0;
      if (JS_EvaluateUCScript(g_pInfo->pJSContext,
                              JS_GetGlobalObject(g_pInfo->pJSContext),
                              (const jschar*) (const TCHAR*) script,
                              wcslen(script),
                              filename, lineno, &rval))
      {
        if (JSVAL_IS_NUMBER(rval))
        {
          jsdouble dbl;
          JS_ValueToNumber(g_pInfo->pJSContext, rval, &dbl);
          FLOAT fps = (FLOAT) dbl;
          WCHAR info[256];
          swprintf(info, 256, L"Read system.timer as %g", fps);
          WriteConsoleString(info);
        }
      }

(7)   LeaveCriticalSection(g_pInfo->pJSCS);

(8)   return FALSE;
     }
```

Whohoo, a lot of interesting stuff in there! Lets break it down into smaller bits.

(1) Remember that a plugin is unzipped into a temp folder by the player on execution? This is how you could gain access to this folder. *g_pInfo* already provides some preset info that you can use. Here, a path *to myimage.png* is created (you need to ship that file with your plugin/dxeffect. Read: Have it in the /pack folder when the final plugin is zipped up after compilation).
Using DirectX routines, a texture (“mytexture”) is created from the image (show Windows message box on error), as well as D3DXSPRITE (“mysprite”).

Note: As we did see above, *OnDeviceLost* is only being called on the sprite, not on the texture. This is because D3DXCreateTextureFromFile places the texture into D3DPOOL_MANAGED (where things happen automatically).

(2) Use a helper routine in our code to send some text to the DXStudio player’s console (we had that already before)

(3**) (IMPORTANT)** You’ll find and use this line quite often – and not using it where required will bring you a lot of trouble.
Remember the DXStudio environment is a multithreaded build with several “blocks” interacting. One block is the Javascript Engine. To preserve state, only one thread is allowed to access the Javascript engine at a time.
To do so, you must “lock” the engine (read: wait until others finished their work, then do your own things while preventing others to do theirs, then leave. If you want, imagine it working like a public toilet ;-P). *EnterCriticalSection()* would help you there. It waits, then locks. So, whenever you access the Javascript engine, use *EnterCriticalSection()* and the (7) corresponding *LeaveCriticalSection()*.

(4,5) These are code examples for creating a “DXStudioVector-like” object in the C++ environment and for creating a new Vector object in the Javascript engine. They are not used here, so we skip that (actually, I think the “*new*” might cause some memory garbage).

(6) Now this is a fine one. By using JS_EvaluateUCScript(), you can execute Javascript inside the JS engine (like as you would have typed it into the DXStudio document). The actual JS code to execute is located in variable “script”, while “filename” and “lineno” contain info that will be shown in the player’s console in case of an error (“syntax error in…” or so).
Here, “system.timer” is executed in JS (which means “retrieve the value of system’s timer property), returning the value of the timer as a Javascript object in “rval”.
After that, it is checked IF the returned value actually IS a number (theoretically, it could be everything), then the value is converted to a float and again sent to the player’s console in an info string.

(8) Actually, I’m not sure why to return FALSE here (tests with TRUE didn’t change anything for me). So, we just leave it as it is.

7.5.4.4 EngineStage

This function is being called several times per render pass, depending what is defined in header.xml (see 6.2.1).

```
PLUG_API BOOL EngineStage(DWORD instancehandle, ES stage, int pass, SEngineInfo* ei)
```

Out of the different possibilities, only the *ESRenderSceneBergin* and *ESRenderSceneEnd* stages are handled and forwarded to *myclass.xxxxx*. This way, our code in myclass gets called upon the START and the END of a rendering cycle.

```
void CMyClass::RenderSceneBegin(SEngineInfo* pEI, int pass)
{
  if ((g_pInfo->pDevice==NULL) || (lostdevice)) return;
  if (pass != 0) return;
}
```

Well, basically, this does achieve nothing. It just shows how we could test for a valid device, the right pass and for a lost device. Lets move on to RenderSceneEnd, hoping there would be more of an action.

```
     void CMyClass::RenderSceneEnd(SEngineInfo* pEI, int pass)
     {
       if ((g_pInfo->pDevice==NULL) || (lostdevice)) return;

(1)    mv=pEI->mview;
       mp=pEI->mproj;
       D3DXMatrixInverse(&mvi, NULL, &mv);
       viewposition=D3DXVECTOR3(mvi._41,mvi._42,mvi._43);
       viewrot=pEI->camera_rot;
       D3DXMatrixIdentity(&matworld);
       LPDIRECT3DDEVICE9 pDevice=g_pInfo->pDevice;
       pDevice->SetTransform( D3DTS_WORLD, &matworld );
       pDevice->SetTransform( D3DTS_VIEW, &mv );
       pDevice->SetTransform( D3DTS_PROJECTION, &mp );

(2)    mysprite->Begin(D3DXSPRITE_ALPHABLEND);
       pDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
       pDevice->SetRenderState(D3DRS_ALPHATESTENABLE, FALSE);
       pDevice->SetRenderState(D3DRS_ZWRITEENABLE,FALSE);
       D3DXVECTOR3 cen(0.0f, 0.0f, 0.0f), pos(512.0f, 512.0f, 0.0f);
       D3DXMATRIX mat, mattemp;
       D3DXMatrixRotationZ(&mat, rotangle);
       D3DXMatrixTranslation(&mattemp, -256.0f, -256.0f, 0.0f);
       D3DXMatrixMultiply(&mat,&mattemp,&mat);
       D3DXMatrixTranslation(&mattemp, 256.0f, 256.0f, 0.0f);
       D3DXMatrixMultiply(&mat,&mat,&mattemp);
       mysprite->SetTransform(&mat);
       mysprite->Draw(mytexture, NULL, NULL, NULL, 0xffffffff);
       mysprite->End();
     }
```

(1) This code block shows some example of how to set up projection and view matrices and device states. I believe it is not needed for the sample plugin to work.

(2) This code block is “pure DirectX” and draws a sprite (actually, the texture “mytexture” that was created earlier on, by *using* a sprite). As we are in *RenderSceneEnd*, this is drawn on top of all the other things in the scene (as those have already been drawn).

7.5.4.5 Update

This function is called once per frame to update any calculations in the plugin.

```
PLUG_API BOOL Update(DWORD instancehandle, FLOAT seconds)
```

This calls *myclass.Update().*

```
void CMyClass::Update(FLOAT seconds)
{
 rotangle+=seconds*speed;
}
```

Well, here, a value for “rotangle” is calculated (which is then used for rotation when drawing the plugin’s custom texture/sprite in *myclass. RenderSceneEnd*)

8 Creating your own Plugin Base Project

By ripping some parts out of the (already modified) sample project, we can create a base project and use this when starting with a new plugin. First of all, *make a copy* of the existing project we worked on before, so you have a backup available. Now we are going to make it a clean “starter project”.

8.1 header xml

Select a fancy name and id placeholder (need to adapt for each new plugin)

```
<dxeffect version="4.0.0" name="Sample Plugin" id="sample">
```

Put in whatever your standard copyright term is

```
<copyright>No copyright, use as you like!</copyright>
```

Put in whatever your standard copyright term is

```
<summary>
  <![CDATA[A sample plugin that is a good starting point for building your own.]]>
</summary>
```

Once again, change the (unique) guid (need to adapt for each new plugin)

```
<version id="{A0C49A8E-BDB3-4c22-BB32-1E1DD999601E}" friendlyversion="1.0.5" />
```

Personally, I like to keep in some dummy lines for a method and a property. Just makes it easier to duplicate new ones later on.

```
<method id="dummy1" help="dummy1(a)//A dummy1 method. Parameter a is a value."/>
<property id="dummy2" type="float" help="dummy2//A dummy property." default="1.0"/>
```

8.2 buildpack.bat

In buildpack.h, change to target copy name to a fancy dummy name (need to adapt for each new plugin).

```
...
copy ..\temp.zip "C:\Users\MiKo\Documents\DX Studio Documents\library\plugins\mydummy.dxeffect"
...
```

8.3 myclassscript.h

You might want to change the internal name of JSClass myclass_class from “ZDXPLUGIN-sample” to something else, just to be consistent.

```
static JSClass myclass_class = {
  "ZDXPLUGINdummy", JSCLASS_HAS_PRIVATE,
  generic_addProperty,  generic_delProperty,  generic_getProperty,  generic_setProperty,
  generic_enumerate,    generic_resolve,      generic_convert,      generic_finalize
};
```

Delete all *myclass_setXXX* and *myclass_getXXX* functions, add two new (dummy) ones:

```
static JSBool myclass_getDummy2(JSContext *cx, JSObject *obj, jsval id, jsval *vp)
{
  CMyClass* myclass=(CMyClass*) JS_GetPrivate(cx, obj);

  double val = (double) 1.0f;
  jsval thisval;
  JS_NewNumberValue(cx,val,&thisval);
  *vp=thisval;

  return JS_TRUE;
}

static JSBool myclass_setDummy2(JSContext *cx, JSObject *obj, jsval id, jsval *vp)
{
  CMyClass* myclass=(CMyClass*) JS_GetPrivate(cx, obj);

  return JS_TRUE;
}
```

Adapt the content of *JSPropertySpec myclass_props* to the new dummy property handlers

```
static JSPropertySpec myclass_props[] = {
    {"dummy2",        0,      JSPROP_ENUMERATE, myclass_getDummy2, myclass_setDummy2},
    {0}
};
```

Change the one and only method handler to a dummy type and adjust *JSFunctionSpec myclass_methods* accordingly.

```
static JSBool myclass_dummy1(JSContext *cx, JSObject *obj, uintN argc, jsval *argv, jsval *rval)
{
  CMyClass* myclass=(CMyClass*) JS_GetPrivate(cx, obj);

  *rval = BOOLEAN_TO_JSVAL(JS_TRUE);

  return JS_TRUE;
}

static JSFunctionSpec myclass_methods[] = {
    {"dummy1",            myclass_dummy1,       1},
    {0}
};
```

8.4 plugin.h/cpp

In *plugin.cpp*, delete the calls to *myclass.RenderSceneBegin* and *myclass.RenderSceneEnd* inside the *EngineStage* function (at least, I did so for my “utility plugins”. When directly going for DirectX usage, leave them in). We’ll leave the other calls to members of *myclass* in place, as they could be handy for a new project.

8.5 myclass cpp/h

Clean up the header file, so that only a few functions and variables remain.

```
class CMyClass
{
  public:
    CMyClass();
    ~CMyClass();
    BOOL GetJSClass(JSClass** ppJSClass, JSPropertySpec** ppJSProps,
                    JSFunctionSpec** ppJSMethods, void** ppPrivate);
    BOOL DocumentLoaded();
    void DeviceLost();
    void DeviceRestore();
    void Update(FLOAT seconds);
    void WriteConsoleString(const WCHAR* str);

    BOOL lostdevice;
};

extern CMyClass myclass;
```

Same goes for the source file. Remove the functions that are not present in the header anymore, and clean up the remaining blocks of demo code. I’m not going to list the changes in detail here. You might know what to do already (if not: see next chapter, hehe).

8.6 Cross-Checking

To see if the recent “cleanup” didn’t break anything, make a test compile. Fix any errors that might occur (like, usage of a deleted variable etc.). The compiled plugin should work now, but not really do anything useful.
Save this newly created “base project” to a secure location and copy it over for any new projects.

9 Writing a Plugin: GetComputerName_Plugin

Now, on to your very own first plugin. It will expose a method that delivers the computer name (as known to windows) back as a string for usage in your DXStudio doc. Here, we want to utilize both, a property and a method – both doing the same thing in slightly different ways.

9.1 Usage Target

We want to use the plugin in DXStudio script like this (accessing it by both, a method and a property. This is kind of redundant, but will show different ways to achieve your goal):

```
var aString = Scenes.scene_1.effects.GetComputerName.getComputerName();
var bString = Scenes.scene_1.effects.GetComputerName.sComputername;
```

while this should not have any effect (obviously):

```
Scenes.scene_1.effects.GetComputerName.sComputername = “duh”;
```

Internally, the plugin shall use a Windows function to retrieve the computer’s name. Then, convert it to an appropriate string and return it to Javascript.

9.2 Basic Preparations

Copy over your fancy plugin base project and prepare it for creating a new DXEffect by

- Changing the name and GUID in *header.xml*
- Adjusting the target file name in *buildpack.bat*

9.3 Extending myclass

Before adding the interfacing to Javascript, lets first create the main “worker routine” in *myclass*. This is where we retrieve the computer’s name and return it as a string. After implementing this, we can call it via our “Javascript handlers”. I’m using this code and make it a public function in the header:

```
WCHAR* CMyClass::myGetComputerName()
{
	static WCHAR wcsCNAME[MAX_COMPUTERNAME_LENGTH+1];
	memset(wcsCNAME,0,sizeof(wcsCNAME));

	DWORD dwSize = MAX_COMPUTERNAME_LENGTH;
	if (GetComputerName(wcsCNAME,&dwSize)==false)
	{
		swprintf(wcsCNAME,MAX_COMPUTERNAME_LENGTH,L"n/a");
	}

	return(wcsCNAME);
}
```

Note that transferring strings over to the Javascript engine needs them to be wide characters. In case you retrieve multi-byte from somewhere, you'd need to do a conversion before (this is not the case here).

9.4 Adding Handlers

When adding handlers, we start with the *header.xml*. I'm going to replace the "dummy" method and property by something like this:

```
<method id="getComputerName" help="getComputerName()//Returns the computers name (as
known to Windows) as a string."/>
<property id="sComputername" type="string" help="sComputername//The computer's name
(as known to Windows). Read only." default=""/>
```

Now, in *myclasscript.h*, we are going to adjust values in the *myclass_props* and *myclass_methods* arrays, so they mirror what we have added to *header.xml* just before:

```
static JSPropertySpec myclass_props[] = {
    {"sComputername", 0, JSPROP_ENUMERATE, myclass_getComputername, myclass_setComputername},
    {0}
};
```

As *sComputername* shall be read only, we could add *null* instead of an actual pointer to *myclass_setComputername* here (where we do just nothing). Also, using flags like JSPROP_READONLY could serve here. Personally, I like the way - as shown - best.

```
static JSFunctionSpec myclass_methods[] = {
    {"getComputerName", myclass_getComputerNameMethod, 0},
    {0}
};
```

The one and only method does not take any parameters, so the last value is 0.

Now, coming to the handler functions themselves. As we put 3 names into the *myclass_xxx* arrays, we need 3 functions.

```
static JSBool myclass_setComputername(JSContext *cx, JSObject *obj, jsval id, jsval *vp)
{
   // Doing nothing. Read only
   return JS_TRUE;
}
```

Now, this one is easy, eh?

```
static JSBool myclass_getComputername(JSContext *cx, JSObject *obj, jsval id, jsval *vp)
{
   CMyClass* myclass=(CMyClass*) JS_GetPrivate(cx, obj);

   WCHAR* wcsComputername = myclass->myGetComputerName();
   size_t iSize = wcslen(wcsComputername);

   JSString* jsStrMyRet = JS_NewUCStringCopyN(cx, (jschar*) wcsComputername, iSize);

   *vp = STRING_TO_JSVAL(jsStrMyRet);

   return JS_TRUE;
}
```

and

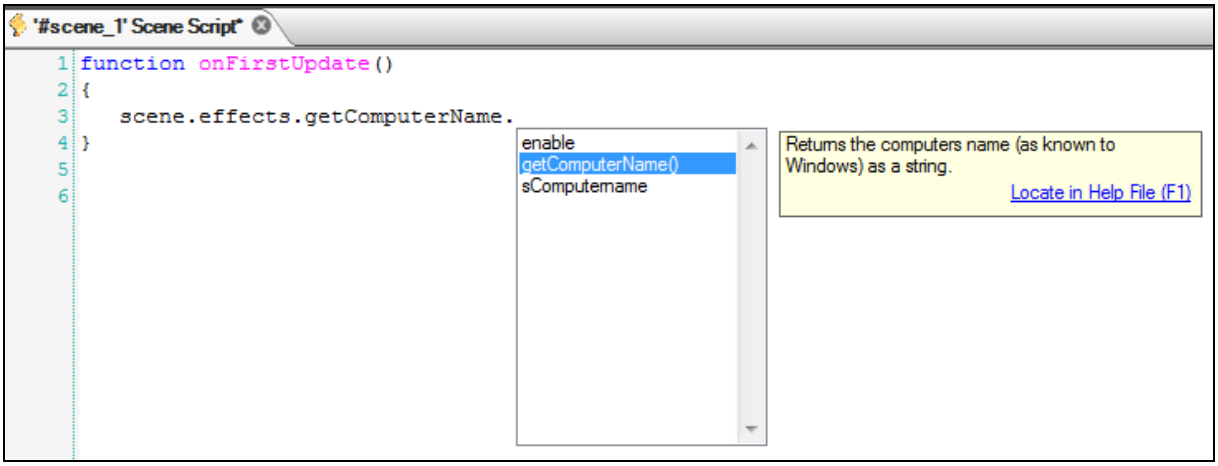
```
static JSBool myclass_getComputerNameMethod(JSContext *cx, JSObject *obj, uintN argc, jsval *argv,
jsval *rval)
{
   CMyClass* myclass=(CMyClass*) JS_GetPrivate(cx, obj);

   WCHAR* wcsComputername = myclass->myGetComputerName();
   size_t iSize = wcslen(wcsComputername);

   JSString* jsStrMyRet = JS_NewUCStringCopyN(cx, (jschar*) wcsComputername, iSize);

   *rval = STRING_TO_JSVAL(jsStrMyRet);

   return JS_TRUE;
}
```

The code for returning the propery value and the method result look pretty much the same. First, the computer name string is retrieved from our “worker function” (wide character), then its length is determined. Next, we create a new Javascript string from that C string (using a helper function from the JS SDK that comes with the project). Then, convert the string to a JSVAL (the universal format for all JS variables) and store it as the return value.

9.5 Embed and Test

Now, compile your very first own project. Hopefully, it will proceed without any errors. Now, navigate to the …*DX Studio Documents\library\plugins* folder and locate the newly generated DXEffect (“getComputerName.dxeffect”). Double-click to rise the plugin’s info window.

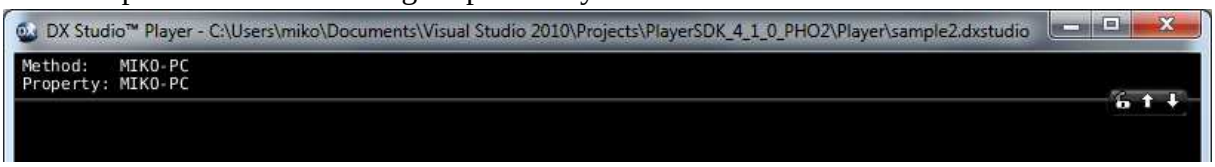
Now, embed the effect into a new DXStudio document and see how the help text you added integrates into the editor:



Then, you can use lines like these to test it:

```
function onFirstUpdate()
{
      print("Method:   "+scene.effects.getComputerName.getComputerName());
      print("Property: "+scene.effects.getComputerName.sComputername)
}
```

…which produces the following output on my machine:



This (rather basic) example concludes the “Hands-On” manual, hoping to give you a head start for own ventures into the DXStudio plugin world. Have fun!