Intercept Calls to DirectX with a Proxy DLL

Posted by **Michael Koch** on **February 28th, 2006**

Introduction

Have you ever wanted to paint your own graphics (or text) on top of a DirectX application (for example, to show TeamSpeak information, or a self-created map, within a game)?

If so, the solution discussed below might help you. By using a "proxy DLL," calls to DirectX can be intercepted, data altered, and even new objects can be created and be shown within another application (read: the source code of this application does not need to be available).

This article's topics cover the creation of a basic proxy DLL for DirectX8/9 3D-calls (d3d8/d3d9.dll), including the full source. Based on your imagination and knowledge, you might enhance this for your very own applications and needs.

The following picture shows an example application, adding graphical objects to a game. Testing application: Krakout-Arkanoid (http://www.wegroup.org/Games/Arkanoid-games/Krakout-RE-Arkanoid.html). This was done using a DirectX8 proxy DLL. The yellow rectangle in the upper left corner and the cube in the middle aren't part of the original game; they were created within the proxy DLL and added.

Concept

To obtain DirectX functionality, an application needs to load certain dynamic link libraries (DLLs); for example, d3d9.dll. A "proxy DLL" names exactly like a DLL in question and manages to get loaded by the application instead of the "real, original" one. Once loaded, the proxy loads the "real" DLL by itself and passes all calls from the application on to it.

So, a "filter" is installed between the application and its calls to regular DLL functions. Obviously, your own, new functionality now can be added when intercepting or altering certain function calls.

Regular setup: Application <-> d3d9.dll (system)
Proxy setup: Application <-> d3d9.dll(proxy) <-> d3d9.dll (system)

By the way, this concept is very similar to one used by a Microsoft DirectX programming utility, named *D3DSpy*. Quote from the D3DSpy help file:

> "D3DSpy works by providing a proxy DLL, which the application connects to and treats like the real D3D9.DLL. It intercepts all the calls and gathers various information before making the real D3D call."

The proxy concept doesn't work with all applications due to several reasons. It has to be tested thoroughly with an application (game) before making it available to a wider range of users.

There are other approaches on how to draw your own stuff in a DirectX application (like "run-time code-injection," or even altering the code of an executable file). In case the proxy DLL concept doesn't work for you, you may want to look for one of those (not covered here).

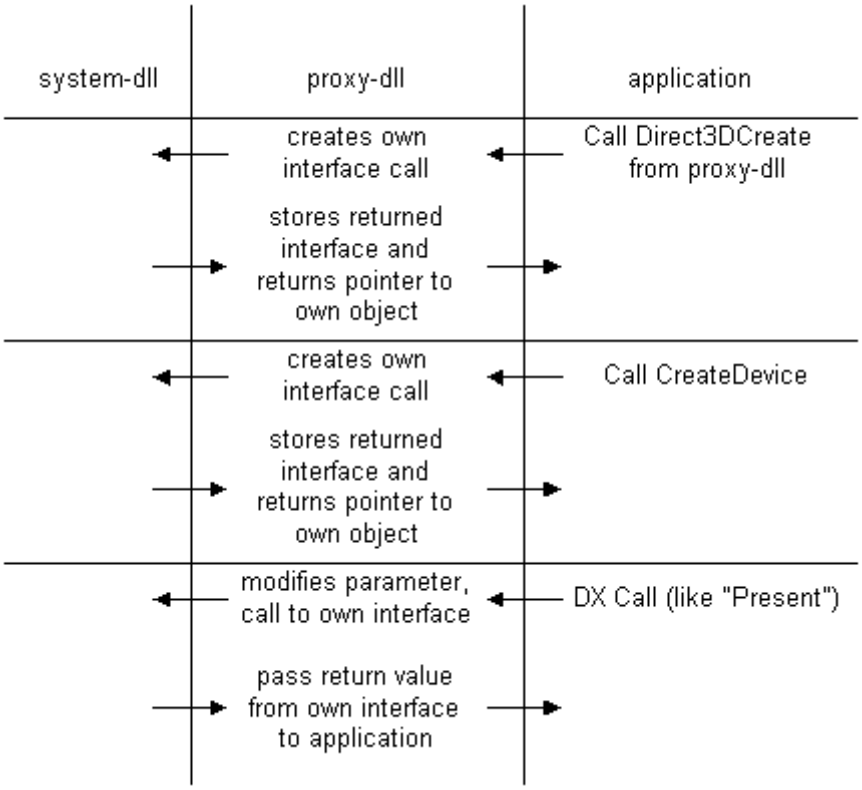
Realisation

I focus on Direct8/9 3D calls here. This said, you need to create a d3d8/d3d9.dll that will be called by the application instead of the "real" (system-provided) library.

> **Note:** To test the proxy DLL, I used *Krakout-Arkanoid* (DX8) and *Flatspace Demo* (DX9), amongst others. Both are nice games, and they were on hand at that time. For debugging purposes, *DebugView* is a nice tool. Ah, and check out *dependency walker* for the examination of DLLs.

First, take a look at an application that uses DirectX 3D calls. It enables D3D support by loading the d3d8/d3d9.dll, and then obtains the IDirect3D8/9 interface via *Direct3DCreate8/9* and creates a IDirect3DDevice8/9 with *CreateDevice*.

If you can manage to get your d3d8/d3d9.dll loaded by the application, thus getting the call to Direct3DCreate8/9 redirected to your code, you could create a IDirect3D8/9 Interface object by yourself. The same goes for the IDirect3DDevice8/9 Interface. You then would offer all funcionality like the original interfaces to the caller's code (in fact, just by redirecting each call to a IDirect3D/IDirect3DDevice you secretly created of your own). This gives you the possibility to change data and/or add new things.



Any drawbacks on that? Yes.

Normally, an application looks for a DLL within its working directory first, and then by browsing the system's path. Applications may prevent that by directly targetting the system directory (this is especially true for online games; they could consider a proxy DLL being a "hack"). And, you can't just replace the system's d3d8/d3d9.dll because you need its functions as described above.

Having said this, I strongly advise to not touch the original system DLL. You might destroy your DirectX installation. Furthermore, an application might implement its own texture manager (or even use undocumented calls). Because you will not be able to handle such calls (obviously), the app possibly would not work with a proxy DLL.

Finally, you know what you need:

- a file named d3d8/d3d9.dll that will be placed within the application's working directory (and hopefully get loaded by the app)
- a function named "Direct3DCreate8/9" being exposed by that DLL
- an interface provided by the DLL that looks exactly the same as the "real" IDirect3D interface
- an interface provided by the DLL that looks exactly the same as the "real" IDirect3DDevice interface

Luckily enough, some header files that come with the DX SDK point out what calls should be handled by your "stub" Interfaces.

The Demo Project

My "basic proxy DLL" implementation consists of these items:

- global routines for the DLL handling (entry point, load the system d3d8/d3d9.dll, and so on)
- an object called myIDirect3D8/9, derived from Direct3D8/9, exposing over a dozen functions
- an object called myIDirect3DDevice8/9, derived from IDirect3DDevice8/9, exposing about 100 functions

Basically, all functions within the "my..." objects pass parameters on to the original system DLL, providing the possibility to change or add data. A few routines need special attention, though.

::Release (both Objects)

In case the application releases the interface (and no more references are present), you destroy your own object as well.

::QueryInterface (both Objects)

If the application queries for certain interfaces, you do of course pass the address of your very own routine, not passing the system DLL response this time.

myIDirect3D9::CreateDevice

Once this is called, you create your own D3DDevice (internally) because you need it to process the application calls. Then, you respond by sending your own object's address.

For testing purposes, the *myIDirect3DDevice8/9::ShowWeAreHere* function was added to the demo source code. It is called within *myIDirect3DDevice8/9::Present* and does nothing but create a yellow rectangle in the upper left screen. This might give you an impression of how to add you own content.

This is a compilation from my work on http://www.proxy.mikoweb.de
rev1.1: added DX8 source, corrected typos

About the Author

Michael Koch

I'm a hobby programmer, using VC++ .net 2003

Downloads

- proxydll_8_src.zip
- proxydll_9_src.zip

Comments

- **d3d9.dll side effects**

  Posted by *gho* on *02/15/2017 08:28am*

  Oh, noo! GHO again.... As a matter of fact, DxWnd implements Coerellian method (see previous post) hooking in-memory the IAT functions and the following COM methods addresses. But a strange thing happens: In certain games (not all, but it would be interesting to understand which and why) some COM hooks get "unlinked" from the proxy interfaces and return back to original implementation, but if you interpose a void d3d9.dll in the middle, all hooks stay magically in place. I put the proxy method apart because I thought I fixed the problem by repeating the hook on certain events, and this fixes most situations, but not all. The "Drakensang demo" executable for instance needs the proxy to keep in place the hook to textures. Could anyone immagine why?

- **Hook the application import table instead**

  Posted by *Corellian* on *03/01/2006 01:27am*

  If you force process X to load "yourhookdll.dll", look up the import table in memory, and overwrite the desired function pointers it will achieve the same results without requiring you to tamper with the system .dll in any way. It's generally a good idea to save the old function ptrs so you can return the the import table back to its previous state if you unload "yourhookdll.dll" at some point before the application terminates. I've used this technique for hooking winsock when writing packet sniffers with great success.

  - **Hook the application import table**

    Posted by *prodigygroupindia* on *06/23/2007 10:03am*

    Is it also applicable for COM function calls? I do not see the calls that are made out of COM instances in the IMPORT TABLE at all. Do you have any clue? venkat at prodigygroupindia dot com.