University of West Bohemia in Pilsen

Faculty of Applied Sciences

Department of Computer Science and Engineering

DIPLOMA THESIS

Plzeň, 2003 Tomáš Smlsal

University of West Bohemia in Pilsen

Faculty of Applied Sciences

Department of Computer Science and Engineering

Diploma Thesis

Grafical Interface Direct X for C# within ROTOR Project

Plzeň, 2003

Tomáš Smlsal

originál zadání

Abstract

A new technology called .NET was recently introduced to wide public. Latest developments in computer graphics are showing popularity of MS DirectX on Windows platforms. Incorporation of such applications, particularly targeted to high performance gaming and multimedia, with .NET Framework environment brings a lot of benefits. A brief description of the DirectX interface is included as well as a short introduction to .NET environment. Also, specific tasks about Shared Source CLI (known as ROTOR) are presented. The main point is correct DirectX interface in .NET Framework implementation. Since only this might not be a problem due to a DirectX 9.0 Managed release, we can still find certain troubles when we need to solve some specific tasks. Solution to it is described in this work. The presented approach is based on COM technology, which allows us to simplify many steps. The idea of COM Interoperability will be briefly described as well. Reached results, advantages, and disadvantages of the selected approach are presented and discussed.

This work is a part of Microsoft Research Ltd. (U.K.): ROTOR project and was supported by the Ministry of Education of The Czech Republic – Project MSM 235200005.

Content

I hereby declare that this diploma thesis is completely my own work and that I used only the cited sources.

Plzeň, 15.th July 2003, Tomáš Smlsal.

....

1 Introduction

The purpose of this work is to provide design to implementation and implementation of DirectX graphical interface components for use in a C# language at the .NET Framework. A goal is to have such environment where the code for C# looks similar to the C++ unmanaged one, yet keeping the rules of .NET managed environment. This work is a part of project ROTOR, which is carried out by universities over the entire technical world, more detailed information is placed at [Cen03]. To better understand the expressions given above, it is essential to spend some time with documentation as [Vis03], or go to particular chapter of interest.

1.1 The Structure

This section contains information on considered topics and structure of this work. The following section *2. Knowledge Survey* introduces the Project ROTOR and development environment together with a DirectX interface, C# language, and the CLI. The *Bibliographic Search* chapter describes results of the literature search, when satisfactory literature had been finally found, as described there. The chapter *4. Possible approaches review* defines the theoretical foundation and prepares to understand the next implementation steps. With all the available knowledge, a very simple implementation design has been stated at the chapter *5. Implementation Design*, which immediately results from the previous chapter 4. Proving details to designed verification are given in chapter *6.1. Verification Design*. There is also explained, why the little testing has been enough to decide that the solution is correct. Then, the chapter *6.2. Verification* describes the own verification. Own DirectX interface implementation is described in chapter *7. Solution Description.* There is also a screenshot of demonstration application to force-feedback joystick. The chapter *8. Performance Evaluation* answers to performance issues. Then it follows up with the *Discussion* and *Constraints*. *Conclusion* summarizes how given aims were fulfilled.

1.2 DirectX Versions

Although the version 9 is adumbrated at the first point of the assignment, DirectX version 8.1b is mostly assumed, if not said explicitly. The reason for this is simple: at the time of beginning of this work the version 9 was unexpected to be released so early, and the difference between unmanaged versions 9 and 8.1b is not so significant, it is just stated in [MS03a]. For the thing itself, the principle of how to solve it is the same for both considered versions, and that is important.

2 Knowledge Survey

In this section will be given a short description of the most important programming tools and environments, which were abundantly used to create a described work. It will be introduced the ROTOR Project, described a C# language, and presented the DirectX interface.

2.1 Terms definition

Before clarifying a meaning of the ROTOR Project, it is necessary to define some needed terms.

.NET Framework is a platform that supports developing and running applications and therefore it is simpler to develop such applications. It is mentioned mainly for the distributed (Internet) applications.

Managed code is a code supplied by additional information, which is needed for some core services. These can be method metadata localization, walking a stack, handling exceptions, and storing and retrieving security information. An exemplary advantage is that developers do not need to care about memory allocation, memory release, and all other memory-management related tasks.

Managed data is data that is allocated and released automatically by the core of the .NET Framework, through a process called garbage collection, which is known e.g. from Java. Managed data can be accessed within the managed code only, but programs that are written in managed code can access both managed and unmanaged data.

CLR (Common Language Runtime) manages memory, thread execution, code execution, code safety verification, compilation, and other system services. These features are intrinsic to the managed code that runs on it [MS01a].

BCL (Base Class Library) is a library of classes, interfaces, and value types that are included in the .NET Framework. This library provides access to system functionality and is designed to be the foundation on which .NET Framework applications, components, and controls are built.

CLI (Common Language Infrastructure) is one of the fundamentals of the technology that supports the .NET Framework functionality. Simply, $CLI \sim CLR \cup BCL$. It provides a specification for executable code and the execution environment (the Virtual Execution System, or VES) in which it runs [MS01b]. Executable code is presented to the VES as modules. A module is a single file containing executable content in the format specified in [MS01c]. At the center of the CLI is a single type system, the Common Type System (CTS), which is shared by compilers, tools, and the CLI itself. It is the model that defines the rules the CLI follows when declaring, using, and managing types. The CTS establishes a framework that enables cross-language integration (language independence), type safety, and high performance code execution. Note that sometimes an acronym CLI is muddled with a CLR.

2.2 The .NET Framework

The .NET Framework is something like a (Java) virtual machine. It allows runtime environment functionality to any .NET application on whatever hardware platform or operating system, where the .NET Framework is implemented. The .NET is based on CLI technology, which ensures a right communication between independent application and specific hardware or operating system. The CLI is used by many libraries, which are extending it. These libraries are referred to as frameworks. For example, they provide application interfaces (APIs) or programming abstractions.

This platform should fulfill the following intents, as stated in [MS01a]:

- Full object orientation, with no relation between object code and a place where the code is executed.
- Minimum software deployment and versioning conflicts.
- Security of code execution.
- Elimination of scripted and interpreted environments performance problems.
- Unification of Windows- and Web-based applications development.
- All inner communication built on industry standards.

2.3 CLI

CLI is one of the basic .NET Framework components. It consists of Common Language Runtime (CLR) and Base Class Library (BCL). Each .NET application has to suit the CLI. A better clearness on .NET components gives the Fig. 2.1.

CLR is a runtime environment for .NET Framework applications. It provides many services such as code compilation and execution, application memory management, exceptions management, metadata access, intermediate language (MSIL: MicroSoft Intermediate Language) to native code conversion. BCL provides a wide set of classes, interfaces, and value types which provides access to system functionality and are designed to be the foundation on which .NET Framework applications, components, and controls are built. A significant treat of BCL library is a logical structure of namespaces, units, where all the classes, types, and interfaces are placed. To facilitate interoperability between languages, the .NET Framework types are CLS (Common Language Specification) compliant and can therefore be used from any programming language whose compiler conforms to the CLS.

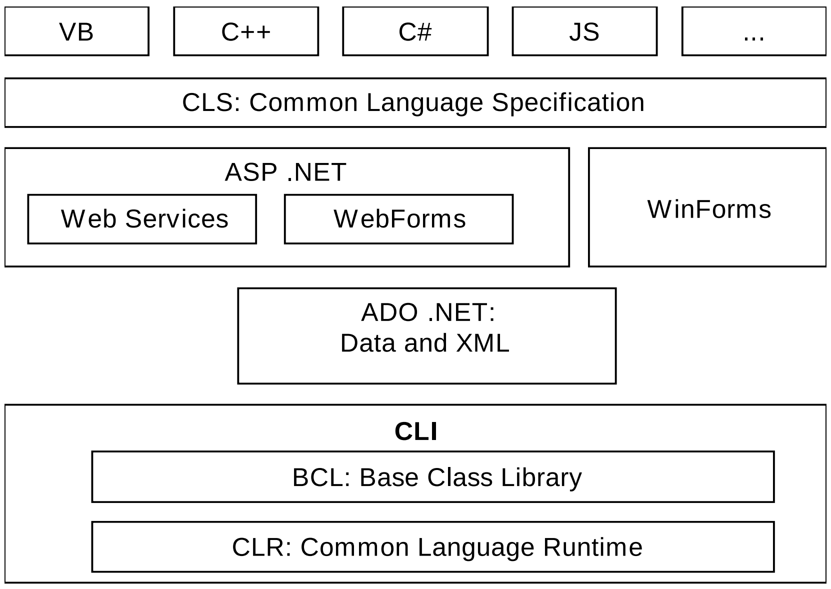


Fig. 2.1 - MS .NET Framework (courtesy of MS electronic archives).

The CLR includes several topics, which are very important for software development and can be represented in the chart at Fig. 2.2. The more significant are COM Marshaler and Garbage Collector for us.

The runtime of .NET Framework has some features, such as memory management, based on garbage collector (GC). It automatically controls the lifetime of existing objects, their location in memory to prevent fragmentation and removes them from memory since there is no reference to them. A code written for this managed environment can be called safe code and no pointers are allowed. Having a reference to an object, GC can shift the object in memory and the reference is still pointing to it. But once the pointer is initialized to some address, GC must keep away from the object lying there to avoid its possible shifting and invalidating the pointer. To switch to this unmanaged mode, where pointers are used, the unsafe code has to be used. To pass data into DirectX methods, pointers should be necessary as well as the unmanaged mode. But the managed one is preferred.

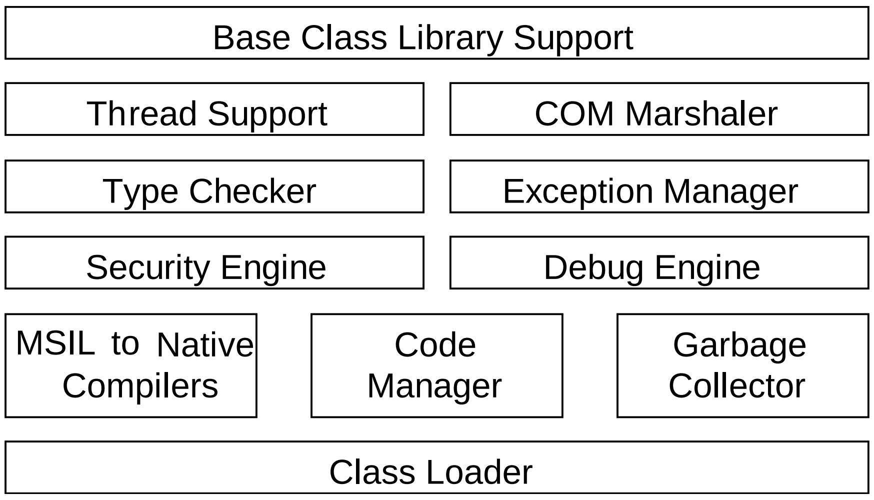


Fig. 2.2 - CLR components

2.4 C# Language

C# [read as "c sharp"] is a native programming language of the .NET Framework and has been standardized [ECM02]. It is very similar to Sun's JAVA in the sense of syntax and with some significant exceptions mentioned later, it is executed in the comparable way as a JAVA. It allows more inheritance, deriving and polymorphism to the business software development. Although .NET Framework provides a high level of language interoperability (every object code can be written in whatever language suiting the CLI), the C# language still remains the closest to the .NET by its idea.

Some beginners to the .NET have problems with correct understanding of **language interoperability**. It may seem that C# is the basic programming language of a whole .NET and that other languages are only something as extensions. In particular, .NET applications can be written in any language, which meets all the requirements given by specifications (exactly the Common Language Specification). The language interoperability is conditioned by some mechanisms as common data types system (CTS), data marshalling, etc. All the .NET languages at the same way share the CTS. It has to be noted that a source code is compiled into something similar to byte-code and additional type information is included as metadata. The result is mixture of intermediate code and metadata, which are still carrying a description of which types will be available at the runtime. In other words, the intermediate code contains complete information to be compiled into a native code of a used processor. Let us mention at least one advantage of the previously stated principle. It is possible for execution engine to verify the type safety and code correctness just before the execution is done.

It is important to understand this idea to be able to read the documentation provided by the Microsoft Company.

2.5 The ROTOR Project

Now the reader should be ready to understand what is a ROTOR. It is a code word for the Shared Source CLI (also known as SSCLI) Project. Shared Source CLI means that the CLI code (`CD-ROM:/public/references/sscli.code/sscli20020326.tgz`) for the .NET Framework platform has been released to a wide academic public for the improvement (and other, mainly experimental and educational) purposes.

It should be also mentioned that the difference between .NET Framework and SSCLI is in the missing support for the COM (Component Object Model) at the SSCLI side and at the OS existing implementations. This makes DirectX implementation impossible to the SSCLI. .NET is currently available only on the MS Windows, SSCLI both on Windows and BSD-Unix. As a notice, writing of own compilers is also a part of the ROTOR Project.

Later on, it will be clarified that the only successful implementation of DirectX is possible only on the platform, which supports both COM and HW (by drivers) and this platform is the .NET only. Therefore, it is not possible to implement it in ROTOR's SSCLI. For the other hand, the ROTOR exists because of the .NET academic openness, and from this point of view, this work can be treated as a part of the ROTOR Project.

The most known places to the author, where the ROTOR is carried out, are Microsoft Research, Cambridge, United Kingdom and University in Pisa, Italy. In Cambridge, there had been the first Rotor Workshop, the second one had been in Pisa. The idea of these workshops lies in progress reports presentation of Rotor Award winning groups on their activities and in interaction with other members from the Rotor and CLI teams, doing an active research, involving Rotor. Workshop format support invited speakers, rich project presentations and panel discussions

2.6 DirectX

The next definition is adopted from [MS02b]:

"Microsoft DirectX is a set of low-level application programming interfaces (APIs) for creating games and other high-performance multimedia applications. It includes support for two-dimensional (2-D) and three-dimensional (3-D) graphics, sound effects and music, input devices, and networked applications such as multiplayer games."

DirectX allows programmers to access the available hardware devices, such as graphical adapter, sound card, and so forth. It takes the advantage of device independent functions to simplify game related tasks, performed by the computer. It is decomposed into several components, each targeted to a particular area of usability. These components are

- Direct Graphics – graphical output interface, discussed in a detail later,
- DirectInput – input devices interface, supporting (in addition to standard peripherals) joysticks, game-pads and force-feedback devices,
- DirectPlay – multiplayer networking,
- DirectSound – high-performance audio applications dealing for example with capturing waveform audio,
- DirectMusic – software support for soundtrack based waveforms, MIDI,
- DirectShow – high-quality capture and playback of multimedia streams,
- DirectSetup – supports one-call installation of necessary components to DirectX,
- DirectX Media Objects – supports development and using data-streaming objects such as encoders, decoders, and effects.

As given in assignment, it will be considered (in the following text) the graphical part of DirectX – Direct Graphics, which is of the interest.

COM technology was used also at DirectX production, because it helps to outshine the DLL-hell problem. This problem resulted from sharing dynamically linked libraries by different applications, where one application could install its library with changed functionality over another existing library of other application, what caused for example crash of the previously running application.

2.7 Graphical Interface: Direct3D, DirectDraw

The DirectX API handles most of the I/O aspects which programmer needs at a very low-level, and therefore it will certainly pay off to not use the standard Windows I/O functions provided by the GDI in order to gain as much speed as possible. Consequently, managed memory is also a nice thought, but if working with high performance graphics, consideration of classic memory manipulation is also essential. In fact, this is an ideological problem, because the main idea of the .NET is an abstraction (even in memory area), compared to high performance DirectX, which mostly needs characteristic memory support at the developer side (e.g. while working with Vertex Buffer). Despite these difficulties, with some compromise, the problem can be solved, mainly if the performance issue is not the point.

Very important fact is that all this technology is based on a Component Object Model (COM), even if it is not mentioned immediately at the first line of the documentation. In other words, DirectX is a set of COM components, each providing some interfaces, which can be divided into subsets with a similar functionality. One of the subsets handles whatever about the graphics and is called DirectX Graphics. It combines previous 3D and 2D graphic components Direct3D and DirectDraw into one and the name Direct3D remained for both. (Now, the entire planar graphic must be done via 3D component.)

The .NET Framework seemed to be very interesting for people from the area of computer graphics that originators of this work decided to implement some of the well-known graphical interfaces for it. The DirectX Direct3D has been taken into account. This interface is widespread and having it prepared in the .NET Framework, it is easy to extend our *old* working algorithms with new features and functionality. For example, a developer used to write a code for DirectX8.0 can simply continue with a development with it, build it in .NET Framework and easily add whatever other network functionality he wants.

DirectX provides a low-level access to HW, what sometimes makes the code for beginners *hard to read*, especially in the case desiring the most performance from that hardware. Generally, many devices are supporting the DirectX well.

Example of graphical output is shown at Fig 2.3.

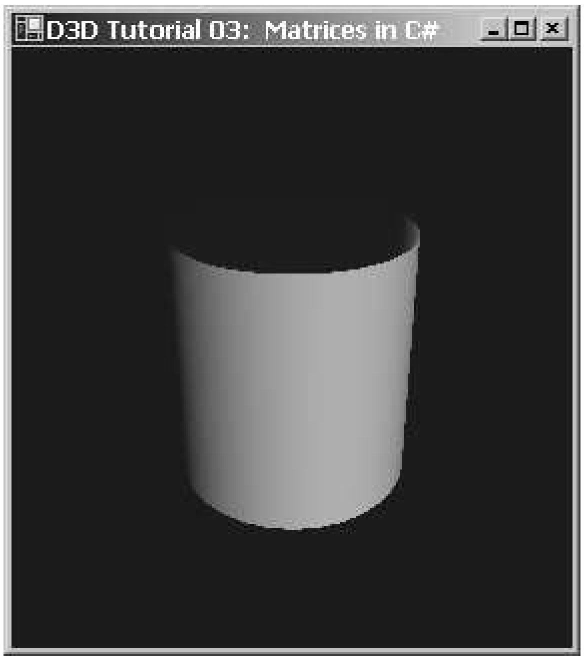


Fig. 2.3 - Example of graphical output. Implemented in C#.

The aim of this work is programming safety and comfort of the use of the ported interface. The idea of pure .NET look of the DirectX ported interface is being reached, i.e. avoiding unmanaged blocks of code in order to communicate with the interface.

3 Bibliographic search

The library of home university – University of West Bohemia in Pilsen – has been recently connected to several scientific bibliographic search databases offering very sophisticated access to worldwide-published literature. Among them, these particular systems have been tried and answered some results:

- Web of Science – `http://wos.cesnet.cz/`
- Eiffel Direct – `http://search.global.epnet.com/`
- Inspec (Dialog) – `http://dialog.cvut.cz/`
- Compendex (Dialog) – `http://dialog.cvut.cz/`
- IEEE Computer Society - Digital Library – `http://dialog.cvut.cz/`
- IEEE/ACM Transactions and Networking – `http://www.acm.org/ton/`
- Journal of the ACM – `http://www.acm.org/jacm/`
- Transactions on Programming Languages and Systems – `http://www.acm.org/toplas/`

However, the processed topic is so new and special, that it probably had rare chance to get into these systems and no useful stuff has been found. Let's suppose there is a developer, doing specific research as this one, demanding vital information on incorporating some COM components into .NET. The highly appreciated information place is certainly the Internet, if not directly the Microsoft site. On this reason, the well-known search engine Google (`http://www.google.com`) has been tested.

As it was expected, with one exception the only available site at the Internet, concerning the solved topic, was the Microsoft's MSDN (MicroSoft Developer Network) Library `http://msdn.microsoft.com`, both with several running discussions on .NET development. For a screenshot of MSDN, see Fig. 3.1. The exception stands for a C-Sharp Corner site `http://www.csharp-corner.com`, where occurred a few subscriptions on DirectX topic. As it was found later, the information lying there is just a rehash of MSDN.

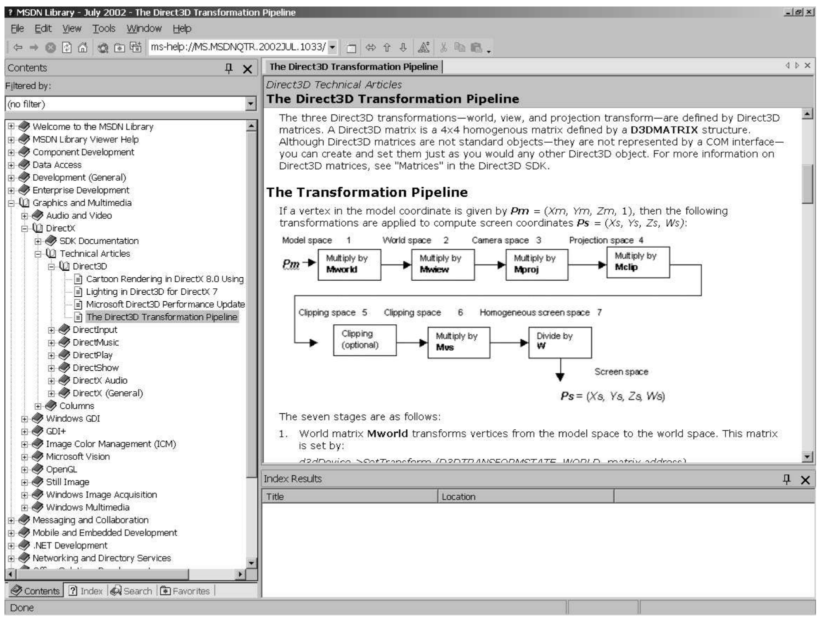
After some exploration of Microsoft pages, the following list of found expert books can be compiled:

- Bargen, Bradley and Peter Donnelly, **Inside DirectX**, Microsoft® Press®, 1998.
- Kovach, Peter J., **Inside Direct3D**, Microsoft Press, 2000.
- Thompson, Nigel, **3D Graphics Programming for Windows**, Microsoft Press, 1996.
- Rogerson, Dale E., **Inside COM**, Microsoft Press, 1997.

These sources are useful to better understand some insides, but for purpose of this work, it is insufficient. More or less surprisingly, this situation just reflects the following idea: any developer should be able to work even only with the generally accessible sources collected at one well known place. This place is the previously mentioned MSDN Library, available via Internet and on many Microsoft product installation CDs. Finally, it is possible to make a decision: only MSDN Library can be singled out as satisfactory source of related information.

Further literature to .NET programming is available also in different languages, as for example the [Kac03].

Results of this section had been lately briefly discussed with thesis supervisor.



Direct3D Technical Articles

The Direct3D Transformation Pipeline

The three Direct3D transformations—world, view, and projection transform—are defined by Direct3D matrices. A Direct3D matrix is a 4x4 homogenous matrix defined by a **D3DMATRIX** structure. Although Direct3D matrices are not standard objects—they are not represented by a COM interface—you can create and set them just as you would any other Direct3D object. For more information on Direct3D matrices, see "Matrices" in the Direct3D SDK.

The Transformation Pipeline

If a vertex in the model coordinate is given by ***Pm*** = (*Xm*, *Ym*, *Zm*, 1), then the following transformations are applied to compute screen coordinates ***Ps*** = (*Xs*, *Ys*, *Zs*, *Ws*):

The seven stages are as follows:

1. World matrix **Mworld** transforms vertices from the model space to the world space. This matrix is set by:

Fig. 3.1 - Screenshot of the MSDN Library: a characteristic layout.

4 Possible Approaches Review

After a closer investigation, there were found the three principal ways to solve the given problem. First, advancing from the COM foundations of the DirectX, is the solution based on a **COM interoperability** approach. This approach employs the `System.Runtime.InteropServices` namespace tools, which helps to build COM into .NET managed environment. Second way to solution is use of the **type library**, where is stored the essential information about COM object (and its interfaces) as types, enums, methods and so on. With having the type library, a lot of programming effort from the first approach is saved. Finally, the third solution is an existing solution released by Microsoft itself. It is called the **DirectX9.0 Managed**, contains nearly all the functions of DirectX8.1b (or DirectX9.0) with exception of DirectShow component and is prepared for immediate use. Unfortunately, this package had been released too late after assigning this topic. Notice that the lack of anything similar to DirectX 9.0 Managed had just been the motivation for this work.

4.1 COM Interoperability

To take the easiest decision in sense of expended programming effort, it is necessary to use the .NET Framework facilities to bring the DirectX functionality into .NET. These facilities are particularly called COM Interop (Component Object Model Interoperability) and expectably they are suitable to get-in those functions written in COM.

It is easy to ask a question, why not only to simply take existing DirectX dll's and wrap their functions into a .NET assembly as it was done in [Han03] with OpenGL. The short and the long of it is just the COM. Previously mentioned dll's contains only a few functions, but all DirectX functions are *packed* in there lying objects and are accessible via object interfaces only. Concluding it, the COM technology has to be taken into account, what seemingly complicates the entire work.

By [MS01d], *COM Interop provides access to existing COM components without requiring that the original component be modified*. A step to incorporate COM code into a managed application is to *import the relevant COM types by using a COM*

```
C:\ap\dp\dll.tools>tlbimp d3d8.dll
Microsoft (R) .NET Framework Type Library t
o Assembly Converter 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-20
01.  All rights reserved.

TlbImp error: The input file 'C:\ap\dp\dll.
tools\d3d8.dll' is not a valid type library
```

Fig. 4.1 - Screen copy. TlbImp.exe does not help.

Interop utility (TlbImp.exe) for that purpose. Once imported, the COM types are ready to use. After execution, the common language runtime marshals data between COM objects and managed objects as needed.

Unfortunately, in the case of DirectX, this does not work at all, as seen in Fig 4.1. As some investigation had been done with the `TLBIMP.EXE` (Type Library Importer), a command-line tool included in the .NET Framework SDK, it seems there is no type library included in the DirectX DLL's. The author suspects that it is simply from the efficiency reasons. It may be generally wrong to include a type library into a DLL supposing that a developer, even the program user, will need this functionality for some wrapping. Other question, which appears then, is why in contrast the quartz.dll can contain its type library.

Let us remind what particular facilities does .NET Framework provide to C# while performing COM Interop. C# has support for

- creating COM objects,
- determining if a COM interface is implemented by an object,
- calling methods on COM interfaces, and
- implementing objects and interfaces that can be called by COM clients. (Stated only for completeness reason. This would be used e.g. in case of writing DirectX component in C# and expecting its usage also in unmanaged C++ as a COM.)

Notice that The .NET Framework handles reference-counting issues with COM Interop so there is no need to call or implement AddRef() and Release() functions.

4.1.1 How To Create a COM Class Wrapper

For C# code to reference COM objects and interfaces is necessary to include a .NET Framework definition for the COM interfaces in the C# build. As known, the TlbImp.exe cannot help, so a COM type library conversion into .NET Framework metadata – effective creation of a managed wrapper that can be called from any managed language – has to be done a quite trickily. The matter is to manually define the COM definitions in C# source code using C# attributes. Once the C# source mapping has been created, all to do is *simply* compile the C# source code to produce the managed wrapper. Wrapper is an original entity being converted together with additional code to support functionality at new environment.

The following conversions have to be performed manually as well:

- **COM coclasses** conversion to C# classes with a parameterless constructor,
- **COM structs** (structures) conversion to C# structs with public fields.

A great way to check registered COM components as a feedback to our effort is to run the .NET Framework SDK command-line tool `Ildasm.exe` (Microsoft Intermediate Language Disassembler) to view the result of the conversion.

The main attributes needed to understand them to perform COM mapping are:

- **`ComImport`** - Marks a class as an externally implemented COM class.
- **`Guid`** – Used to specify a universally unique identifier (UUID) for a class or an interface.
- **`InterfaceType`** – specifies whether an interface derives from `IUnknown` or `IDispatch`.
- **`PreserveSig`** – specifies whether the native return value should be converted from an `HRESULT` to a .NET Framework exception.

Each of these attributes has own specific values, which should be *very clear* to everyone before using them.

4.1.2 Declaring a COM coclass

COM coclasses are represented in C# as classes. These classes must have the `ComImport` attribute associated with them. The following restrictions apply to these classes:

- The class must not inherit from any other class.
- The class must implement no interfaces.
- The class must also have a `Guid` attribute that sets the globally unique identifier (GUID) for the class.

The following example declares a coclass in C#:

```
// declare FilgraphManager as a COM coclass
//
[ComImport, Guid("E436EBB3-524F-11CE-9F53-0020AF0BA770")]
class FilgraphManager
{
}
```

The C# compiler will add a parameterless constructor that can be called to create an instance of the COM coclass.

4.1.3 Creating a COM Object

COM coclasses are represented in C# as classes with a parameterless constructor. Creating an instance of this class using the new operator is the C# equivalent of calling `CoCreateInstance()`. Using the class defined above, it is simple to instantiate the class:

```
class MainClass {
  public static void Main() {
    //
    // Create an instance of a COM coclass - calls
    //
    // CoCreateInstance(
    //   E436EBB3-524F-11CE-9F53-0020AF0BA770,
    //   NULL, CLSCTX_ALL,
    //   IID_IUnknown, &f)
    //
    // returns null on failure.
    //
    FilgraphManager f = new FilgraphManager();
  }
}
```

The short and the long of it is that the COM object is created automatically by .NET Framework runtime.

4.1.4 Declaring a COM Interface

COM interfaces are represented in C# as interfaces with `ComImport` and `Guid` attributes. They cannot include any interfaces in their base interface list, and they **must declare the interface member functions in the order that the methods appear in the COM interface**.

COM interfaces declared in C# must include declarations for all members of their base interfaces with the exception of members of `IUnknown` and `IDispatch` – the .NET Framework automatically adds these. COM interfaces which derive from `IDispatch` must be marked with the `InterfaceType` attribute.

When calling a COM interface method from C# code, the common language runtime must marshal the parameters and return values to (from) the COM object. For every .NET Framework type, there is a default type that the common language runtime will use to marshal when marshaling across a COM call. For example, the default marshaling for C# string values is to the native type `LPTSTR` (pointer to `TCHAR` char buffer). You can override the default marshaling using the `MarshalAs` attribute in the C# declaration of the COM interface. The exact manner of marshalling particular arguments is not so important as long as the own process of marshalling is straightforward (mentioned the marshalling at runtime), because interface definitions just exist. The problem arises earlier, at time of manual rewriting COM interface methods into C# code, when it must be exactly known how the argument types have to be substituted. See later in *Chapter 6: Solution Description.*

In COM, a common way to return success or failure is to return an HRESULT and have an out parameter marked as retval in MIDL (Microsoft Interface Definition Language) for the real return value of the method (in syntax of IDL):

```
HRESULT _stdcall MyMethod(
                   [out, retval] InMyFace** ReturnVal);

HRESULT _stdcall MyOtherMethod(
                   [out, retval] VARIANT_BOOL* ReturnVal);

HRESULT _stdcall CreateDevice(
                   [in] UINT Adapter,
                   [in] D3DDEVTYPE DeviceType,
                   [in] HWND hFocusWindow,
                   [in] DWORD BehaviorFlags,
                   [in] D3DPRESENT_PARAMETERS*
                          pPresentationParameters,
                   [out, retval] IDirect3DDevice8**
                          ppReturnedDeviceInterface);
```

In C# (and the .NET Framework), the standard way to indicate an error has occurred is to throw an exception. By default, the .NET Framework provides an automatic mapping between the two styles of exception handling for COM interface methods, which are called by the .NET Framework:

- The return value changes to the signature of the parameter marked retval (void if the method has no parameter marked as retval).
- The parameter marked as retval is left off of the argument list of the method.
- Any non-success return value will cause a System.COMException exception to be thrown.

The next example taken from [MS01d], and shortened, shows a COM interface declared in MIDL and the same interface declared in C# (note that the methods use the COM error-handling approach).

The original MIDL version of the interface:

```
[ odl,
  uuid(56A868B1-0AD4-11CE-B03A-0020AF0BA770),
  helpstring("IMediaControl interface"),
  dual,
  oleautomation
]
interface IMediaControl : IDispatch {
...
  [id(0x60020006), propget]
  HRESULT FilterCollection(
                   [out,retval] IDispatch** ppUnk);
  [id(0x60020007), propget]
  HRESULT RegFilterCollection(
                   [out,retval] IDispatch** ppUnk);
  [id(0x60020008)]
  HRESULT StopWhenReady();
};
```

Here is the C# equivalent of this interface:

```
using System.Runtime.InteropServices;

// Declare IMediaControl as a COM interface which
// derives from the IDispatch interface.
[Guid("56A868B1-0AD4-11CE-B03A-0020AF0BA770"),
    InterfaceType(ComInterfaceType.InterfaceIsDual)]
interface IMediaControl // cannot list any base interfaces
                        // here
{
  // Note that the members of IUnknown and Interface
  // are NOTlisted here
  //
...
```

```
    [return : MarshalAs(UnmanagedType.Interface)]
    object FilterCollection();

    [return : MarshalAs(UnmanagedType.Interface)]
    object RegFilterCollection();

    void StopWhenReady();
  }
```

Note how the C# interface has mapped the error-handling cases. If the COM method returns an error, an exception will be raised on the C# side. To prevent the translation of `HRESULT`s to `COMExceptions`, attach the `PreserveSig(true)` attribute to the method in the C# declaration. For details, see `PreserveSigAttribute` Class in documentation.

4.1.5 Using Casts Instead of QueryInterface

A C# coclass would be not very useful until it could access an interface that it implemented. In C++, developer would navigate an object's interfaces using the `QueryInterface()` method on the `IUnknown` interface. In C#, the same thing is possible by explicit casting the COM object to the desired COM interface. If the cast fails, then an invalid cast exception is thrown:

```
// Create an instance of a COM coclass:
MyCOMCoclass myCOMCC = new MyCOMCoclass();

// See if it supports the IMyCOMInterface COM interface.
// Note that this will throw a System.InvalidCastException
// if the cast fails. This is equivalent to QueryInterface
// for COM objects:
IMyCOMInterface iMyCOMI = (IMyCOMInterface) myCOMCC;

// Now call a method on a COM interface:
iMyCOMI.MyMethod();
```

This kind of approach seems to be a little puzzling. Even though the COM interface functionality is needed, and interface methods have to be declared firstly, why to complicate it by defining coclasses and cast them to interface? Why not to directly use the interface only? It is very probably that it is a step needed in general case, but in this work it does not seem to be useful.

4.1.6 COM Interfaces

Once the COM interface is declared in C#, all its methods can be called as pleased. But here is a hidden problem: how to retrieve all necessary attribute values for interface declaration (mainly `Guid`) and attributes for arguments marshalling (`In`, `Out`)? Note that there exists an IDL (Interface Definition Language), which syntax supports description capabilities of COM interface. See the IDL part of IDirect3D8 interface, which was obtained by OLE/COM Object Viewer (TypeLib Viewer) tool from a `Dx8vb.dll` Type Library (the *original* IDL file was unavailable!):

```
[
  odl,
  uuid(1DD9E8DA-1C77-4D40-B0CF-98FEFDFF9512),
  helpcontext(0x00014453)
]
interface Direct3D8 : IUnknown {
    [helpcontext(0x00014460)]
    HRESULT _stdcall RegisterSoftwareDevice(
                    [in] void* InitializeFunction);
    [helpcontext(0x0001445a)]
    int _stdcall GetAdapterCount();
...
    [helpcontext(0x00014459)]
    HRESULT _stdcall EnumAdapterModes(
                    [in] int Adapter,
                    [in] int Mode,
                    [in, out] D3DDISPLAYMODE* DisplayMode);
...
    [helpcontext(0x0001445e)]
    long _stdcall GetAdapterMonitor([in] int Adapter);
```

```
        [helpcontext(0x0001446b)]
        HRESULT _stdcall CreateDevice(
                        [in] int Adapter,
                        [in] CONST_D3DDEVTYPE DeviceType,
                        [in] long hFocusWindow,
                        [in] CONST_D3DCREATEFLAGS
                          BehaviorFlags,
                        [in] D3DPRESENT_PARAMETERS*
                          PresentationParameters,
                        [out, retval] Direct3DDevice8**
                          ppReturnedDeviceInterface);
    };
```

If compared to description of the exactly same interface contained in header `d3d8.h`, it is possible to see some similarity:

```
DECLARE_INTERFACE_(IDirect3D8, IUnknown)
{
...
  /*** IDirect3D8 methods ***/
  STDMETHOD(RegisterSoftwareDevice)
    (THIS_ void* pInitializeFunction) PURE;
  STDMETHOD_(UINT, GetAdapterCount)(THIS) PURE;
...
  STDMETHOD(EnumAdapterModes)
    (THIS_ UINT Adapter,
     UINT Mode,
     D3DDISPLAYMODE* pMode) PURE;
...
  STDMETHOD_(HMONITOR, GetAdapterMonitor)
    (THIS_ UINT Adapter) PURE;
  STDMETHOD(CreateDevice)
    (THIS_ UINT Adapter,
     D3DDEVTYPE DeviceType,
     HWND hFocusWindow,
     DWORD BehaviorFlags,
     D3DPRESENT_PARAMETERS* pPresentationParameters,
     IDirect3DDevice8** ppReturnedDeviceInterface)
```

```
            PURE;
        };
```

As it was stated, there are no available IDL files for considered components of DirectX, what little complicates the situation, because it is necessary the resolve the component structure directly from the DirectX header files. In the `d3d8.h`, there are defined 12 interfaces, including totally about 260 COM interface function declarations.

4.2 Type Library

.NET Framework metadata lying in the Type Library are included in a C# build via the /R compiler option, or as reference addition (reference to the COM type library) at the Visual Studio development environment. The main conversion is done automatically.

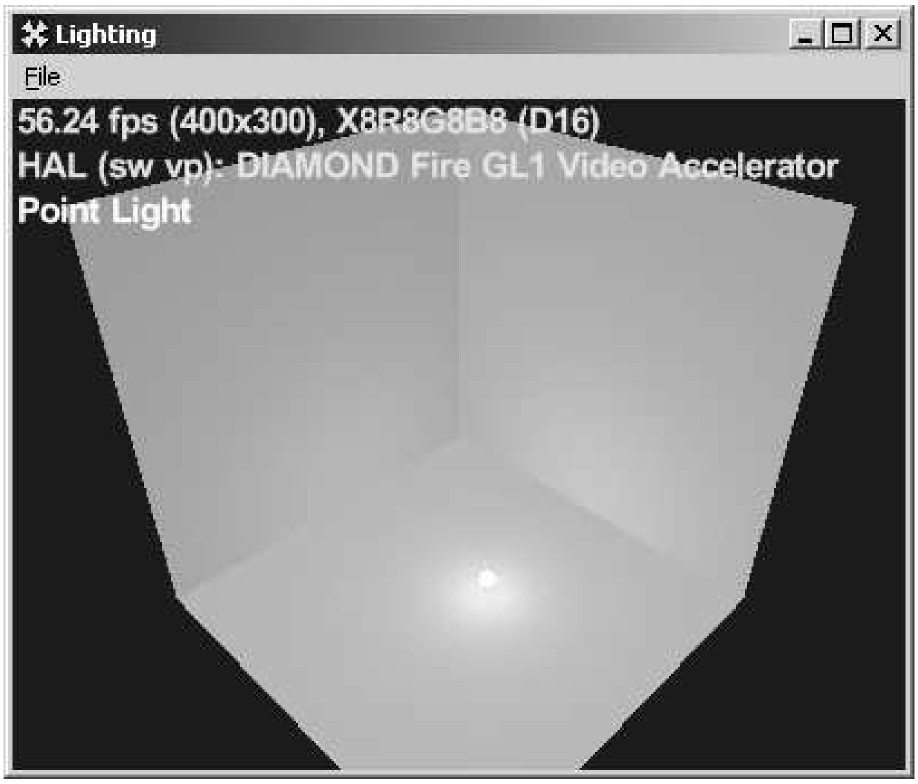
To demanding readers' satisfaction, type library (.tlb, .dll) is a binary file that stores information about a COM or DCOM object's properties and methods in a form that is accessible to other applications at runtime. Using a type library, an application or browser can determine which interfaces an object supports, and invoke an object's interface methods. This can occur even if the object and client applications were written in different programming languages. The COM/DCOM run-time environment can also use a type library to provide automatic cross-apartment, cross-process, and cross-machine marshaling for interfaces described in type libraries. The type library is generated from a special file (see IDL later), which syntax is based on an ODL [MS02c]. The only problem connected to this is a missing support to a `module` *type* (?) at the .NET side. It results in particularly missing functions, e.g. utilizing mathematical functions with vectors, matrices, etc. If there was found a way in which to bring the module-functions to life, the necessity of own implementation in the helper assembly `DxVBLibA1` would be void. Details about the `module` are described in [MS02d].

4.3 Managed DirectX9.0

On December 2002, Microsoft has released the **DirectX 9.0 Managed** version of the DirectX, which should meet all the requirements stated at the previous pages. Thus it is used as a reference for comparison to reached results. At the next few paragraphs only its significant graphic namespaces will be shortly described: Microsoft DirectX, Direct3D and DirectDraw.

The namespace **Microsoft.DirectX** provides utility operations and data storage for DirectX application programming, including exception handling, simple helper methods, and structures used for matrices, clipping planes, quaternion, vector manipulations and so forth. **Microsoft.DirectX.Direct3D** enables to manipulate visual models of 3-D objects and take advantage of hardware acceleration and **Microsoft.DirectX.DirectDraw** that provides functionality across display memory, the hardware blitter, hardware overlay support, and flipping surface support. It seems that small inconsistency appeared because Direct Graphics 8.1b should combine both D3D and DDraw into one, but in the version 9.0 it is formally divided again.

This is the best solution, which provides a complete DirectX functionality in the style of .NET Framework. An example demonstrating DirectX lighting is at the Fig. 4.2. Advanced information for DirectX .NET development is available in [Csc03] and [Vis03].



56.24 fps (400x300), X8R8G8B8 (D16)
HAL (sw vp): DIAMOND Fire GL1 Video Accelerator
Point Light

Fig. 4.2 - DirectX9.0 Managed: Lighting Sample.

4.4 Wrapping in detail

The advantage that DirectX is a COM based is highly welcome. The .NET Framework runtime environment can save a lot of work to developer in a wrapping task because of its runtime callable wrappers feature. The functionality of GC can be used although the pointers are needed as well. Each time the method of a COM is called, the runtime callable wrapper (RCW) is automatically created for accessing the unmanaged code of that COM. It is created every time that the call occurs. This could seem to be unacceptably high overhead cost, but, if considering the fact that for e.g. rendering 10 or 10 billions facets takes only one call and one RCW build, it is feasible. And how it works?

The common language runtime exposes the COM objects through a proxy called as runtime callable wrapper (RCW). Although the RCW appears to be an ordinary object to other .NET clients, its primary function is to marshal calls between a .NET client and COM object, as given in [MS01a].

The runtime creates exactly one RCW for each DirectX COM object, regardless of the number of references that exist on that object. Any number of managed clients can hold a reference to the COM objects that expose some interfaces. The runtime maintains a single RCW for each object.

Using metadata derived from a type library, the runtime creates both the COM object being called and a wrapper for that object. Each RCW maintains a cache of interface pointers on the COM object it wraps and releases its reference on the COM object when the RCW is no longer needed. The runtime also performs garbage collection on the RCW.

Among other activities, the RCW marshals data between managed and unmanaged code, on behalf of the wrapped object, which is essential to this work. Specifically, the RCW provides marshaling for method arguments and method return values whenever the client and server have different representations of the data passed between them.

The standard wrapper enforces built-in marshaling rules. For example, when a .NET client passes a String type as part of an argument to a managed object, the wrapper converts the String to a BSTR type. Should the COM object return a BSTR to its managed caller, the caller receives a String. Both the client and the server send and receive data that is familiar to them. Other types require no conversion. For instance, a standard wrapper will always pass a 4-byte integer between managed and unmanaged code without converting the type, what is very useful.

When created as an early-bound object, the RCW is a specific type. It implements the interfaces that the COM object implements and exposes the methods, properties, and events from the object's interfaces. In the illustration, the RCW exposes the INew interface but consumes the IUnknown and IDispatch interfaces. Further, the RCW exposes all members of the INew interface to the .NET client.

5 Implementation Design

This chapter is a direct follow-up to the previous one. The next paragraph stark proposition can be formed only after a very serious consideration of previously given facts. Moreover, some intensive investigations had to be done, in Visual Studio .NET and its tools, to recover that simple principle. The following result has been found.

Wrapping task can be defined as a process when migrating some functionality from foreign development environment into ours without changes at the original source code. In the other words, it can be also named as porting as in [Han03]. To create a port of some dynamically linked library (.dll) means to somehow provide headers of all necessary functions and to do all the necessary steps for the .dll import. But having the original functionality in a COM, it is simple to let the .NET Framework runtime to do everything automatically. The runtime has methods for handling components written in an unmanaged mode and its basic idea is described in the next paragraph.

Forgetting whatever possible solution $\boldsymbol{\pi}$ exists or not, suppose that a very effective and robust solution $\boldsymbol{\zeta}$ to our problem is presented. Bearing in mind the facilities of .NET Framework for COM technology, it is easy to expect that $\boldsymbol{\zeta}$ will be based on the COM Interoperability. Now, the implementation task is reduced to interface declarations only. But re-declaring of interfaces (same as re-implementing COM interfaces) again, if they are once declared in type library, is like the saying about selling coals to Newcastle. From the stated facts it reasonably implies that

$\boldsymbol{\pi} \approx \boldsymbol{\zeta}$.

In other words, to save the programming effort, the implementation of DirectX graphical interface is best done via the type library approach, which is highly similar to COM Interoperability. Particularly, the type library lies at file `dx8vb.dll` and is named **DirectX 8 for Visual Basic Type Library**. If developer knows the Visual Basic well (i.e. types representation, array indexing, etc.), he is able to easy use this library in C# too. Only those functions requiring some nonstandard techniques as callbacks or memory manipulations have prepared their improved versions to work fine, which could be treated as a small exception to the previous idea.

Example can be the available devices enumerating. In C++, a callback is necessary to this procedure, while in C# is used a function that by default accesses given enumerated device by its order. Though, nearly always the first device will be right (index set to zero), when the e.g. 101st device will cause an exception (if there is not 101 available graphical devices).

6 Solution Correctness

Before continuing in reading, it should be noticed that author is not a software engineer expert. It means that there probably exists a standard and certified ways of software packages verification and validation procedure, but this thesis is completed by a person from the field of computer graphics, who asks for a pardon if anything is not so correct. However, even with the qualification author has, it will be tried to provide a good proof of solution correctness.

6.1 Verification Design

After some approach being done, it is now right to state the important thesis: in the way the implementation is done, only the standard recommended programming techniques are used. That means, if we have correct declarations, we can expect some problem while calling the interface method, e.g. error in marshaler, impossible type-casting, etc. This kind of error would arise just by first run of the application, but once working, it should work forever. All other problems, which could appear, would arise on the side of DirectX, which in principle cannot be handled, or on the side of .NET Framework, where it is again out of the author responsibility.

The previous suggestions are valid for general case of COM Interfaces approach. Considering the fact, that the implementation is done via type library approach, where we can expect that it was verified (it has been released by Microsoft), the necessity of verification is void.

6.2 Verification

The verification of solution correctness has been tested on selected functions. As it was stated in the previous section, we can expect that the type library `dx8vb.dll` is not erroneous, because even after half a year of using, there has been found neither mistake nor error in this library.

For complete picture on tested functions, see the content of directory `programs` on the attached medium. For its largeness, it would be inefficient to include all the program listings into this text or to the attachments.

7 Solution Description

This chapter contains description of selected approach implementation for graphical interface mentioned in previous sections. One of the terms used for porting a library to .NET is the **wrapper**. To use a wrapper or to wrap a library means to create a set of functions (or objects) that shall make interface accessible from particular environment. These functions usually perform system dependent task and call a wrapped function (i.e., particular function of the original library).

The implementation of the graphical interface DirectX has been done with use of type library, originally designated to a Visual Basic programming language. After a deep exploration, there have been found rules for correct incorporation of included functions. It is beyond author strength to provide a complete list of these rules. Probably, it would be also inefficient. Reader should rather see the code of provided samples at the attached medium, which is much more intuitive.

7.1 Implementation

Own implementation consists of a type library `DxVBLibA` from `dx8vb.dll` and additional functions exported in a namespace `DxVBLibA1`. Both `DxVBLibA` and `DxVBLibA1` represent the provided solution.

Note: `DxVBLibA1` assembly is contained in directory called helper by each project.

7.2 Functionality Demo

To support the stated theses about solution correctness, there has been prepared a demonstration application. It is a very simple game, where the quality topic is not the important one. It provides a picture of incorporating some DirectX components in one application: DirectDraw, DirectInput and DirectSound. As it is required in assignment, the application uses a force-feedback device Microsoft Sidewinder Force Feedback II joystick, which is also required for running of this application.

The screenshot is plotted to Fig. 7.1.

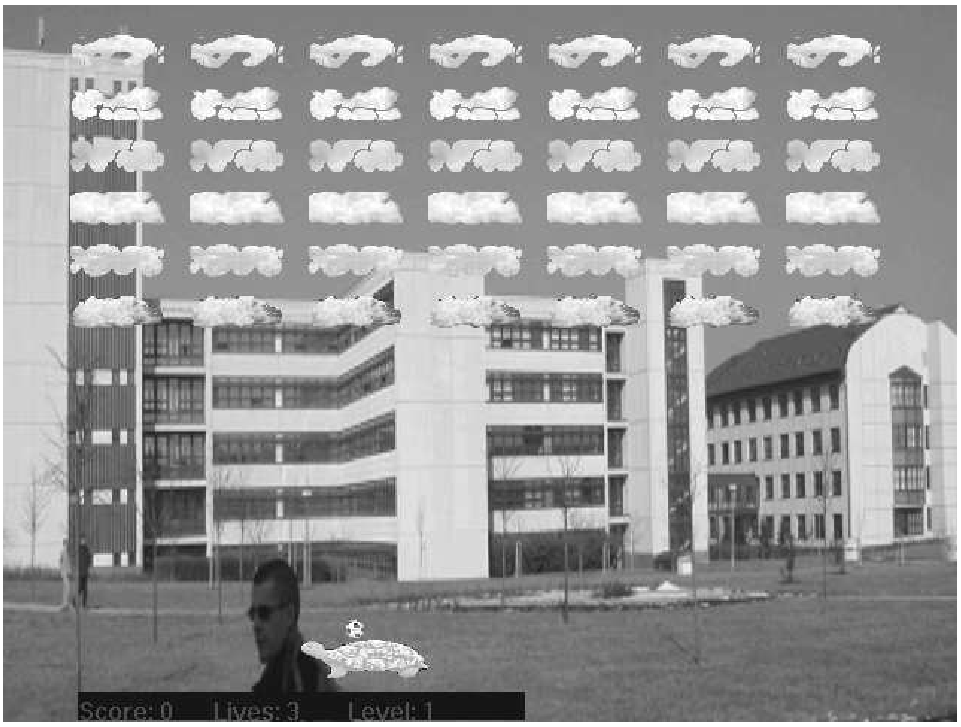


Fig. 7.1 - The screenshot of force-feedback application.

7.3 Implementation Notes

There exist several C# compilers, but for purpose of this work the Visual C# .NET `csc.exe` with the Microsoft Visual Studio .NET IDE had been selected and used as the most convenient.

Some problems encountered while looking up the GUIDs for needed objects. Particularly, there were non found for the Direct3D object coclasses (in headers).

Also, there was a problem with correct understanding with the meaning of returning `null` value on function call failure. Originally, this wrong way was used:

```
a = methodCalling(...);
if (a == null)
  failure_message();
```

It will never reach the line with `failure_message()`, because failure means exception! Instead, use approach as in this example:

```
// Create a DInput object
try {
  di = dx.DirectInputCreate();// Create the dinput device
  if (di == null) {
    MessageBox.Show("dx.DirectInputCreate() Failed.");
    return false;
  }
}
catch (COMException e) {
  MessageBox.Show(e.Message+",
    HResult:0x"+e.ErrorCode.ToString("x"));
  return false;
}
catch (Exception e) {
  MessageBox.Show(e.Message);
  return false;
}
```

While discussing the failures, the interesting question arises. In C++ style, the failures are reported as `HRESULT` values. It can be treated as exception, because something unexpected – unwanted – happened. So it should be implemented in exception style in C#. But for the other hand, it is a used practice to place such a call in an infinite loop, where the program stays until the function has been called with success (e.g. waiting for receiving exclusive access to a specific device, hold by another application). And this contradicts the idea that exceptions must be used only in the last resort, in other words not so often.

It would be interesting to compare the approaches of different error handling – one based on the true `HRESULT` value returning and the second based on nonsuccess `HRESULT` value to exception conversion. Which is better in performance? The way to do it begins with experimenting with the `PreserveSig` attribute value.

If the type library would be unusable from some reason, the approach of own interface methods declaration would be necessary. Then, more attention will have to be given to function arguments marshalling (see the marshalling attributes).

7.3.1 HRESULT in Detail

The `HRESULT` data type is a 32-bit value that is used to describe an error or warning.

```
typedef LONG            HRESULT;
```

On 32-bit platforms, the `HRESULT` data type is the same as the `SCODE` data type. On 16-bit platforms, an `SCODE` value is used to generate an `HRESULT` value.

An `HRESULT` value is made up of the following fields:

- A 1-bit code indicating severity, where zero represents success and 1 represents failure.
- A 4-bit reserved value.
- An 11-bit code indicating responsibility for the error or warning, also known as a facility code.
- A 16-bit code describing the error or warning.

8 Performance evaluation

The performance issue is always a crucial one. Since there are even approximately 260 functions only in the Direct3D component, it was not possible to test in performance all of them. There are also many influences, which gives the total performance. If the `HRESULT` return value function is called in C++, it always returns some code in very similar time interval. In C#, the exception handling presents some delays, which are not caused by the implementation itself, but makes one function sometimes faster and sometime slower, depending on whether exception occurred or not.

Other reason for stating the performance issue so generally is that every function (even interface) needs own specific comprehension to be able to call it. How to correctly prepare the input arguments, when it can be called and so on.

It is also important to keep in mind that DirectX performance highly depends on existing HW support on machine, where it is running. It is nice to provide some particular measurements done in software emulation, but in time when many computers support it by HW, it would be meaningless.

For purity, only some significant graphical operations have been tested (see Tab 8.1, Fig 8.1). From experience of the author, there hadn't appeared any significant performance gap between C++ version and the .NET one.

C# .NET	C++	Function type
27,9	23,2	billboarding
10,3	9,4	clipping
15,6	14,0	vertex shader
9,1	6,6	enhanced mesh
17,0	23,4	lights
7,2	6,3	vertex shader

Tab. 8.1 - Time [ms] to render the tested scene.

Each time the method of a COM is called, the runtime callable wrapper (RCW) is automatically created for accessing the unmanaged code of that COM. It is created every time that the call occurs. This could seem to be unacceptably high overhead cost, but, if considering the fact that for e.g. rendering 10 or 10 billions facets takes only one call and one RCW build on initialization, it is possible to suppose that the performance is not so significantly influenced by wrapping DirectX in .NET.

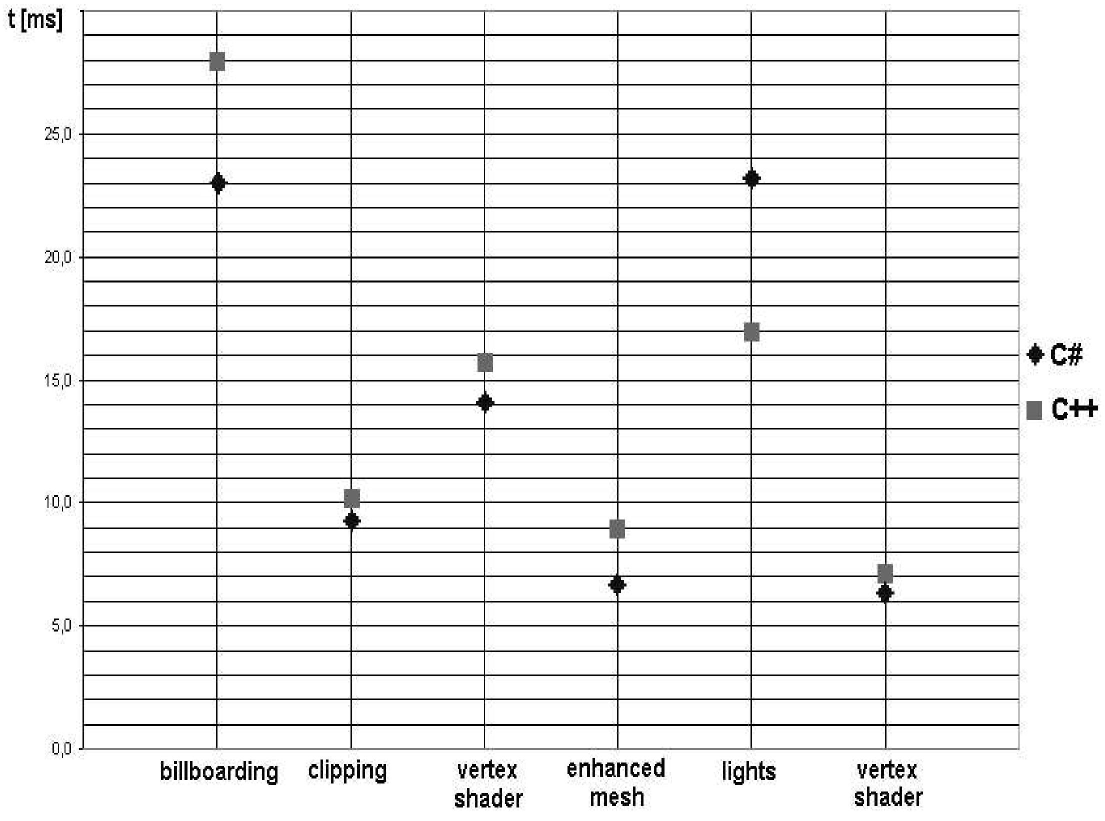


Figure 8.1 - Time [ms] to render scene.

9 Discussion

It has been implemented the DirectX graphical interface for use within the .NET Framework. It fulfills well the objectives given at the early beginning. Now, graphics developers can also work with the fully object oriented programming (OOP) language C#. Advantage of the described solution is a general investigated approach. Not only DirectX, but also any COM software can be handled by the same strategy now.

Three methods of DirectX implementation in C# were introduced and described. Until a version 9.0 has been released in December, the only suitable way for C# developers was the second method based on type library import. Since it has been released, the only recommended way is the third one, DirectX9.0 (managed version). With C# and this version can be reached all features of managed runtime .NET Framework environment and OOP even with reasonable overhead compared to C++.

However the problem seems to be solved, there is still some kind of feeling that the purely correct solution can be provided even much more easily. As it results from some exploration of MSDN documentation about MIDL, shortly IDL (Interface Definition Language) and COM topic, it is possible to generate a type library directly from an IDL file. Also, all COM objects implement one or more interfaces. When a custom COM object is created, the creator must describe the interface or interfaces in an IDL file, which is the needed one. Having the IDL files, for any COM, means a very high possibility of having a type library, from which it is easy to generate an assembly as well. Assumption of Microsoft generating DirectX9.0 Managed assemblies at the previously stated way seems to be probable.

The future work could be aimed at functionality improving, and stability and safety of the implementation. Currently this implementation has been tested due to given capabilities and furthermore, functionality is improved. It is already usable, but shall not be considered to be absolutely error-proof.

10 Constraints

The Type Library approach is constrained to DirectX versions up to 8.1b. It does not support mainly functions operating with memory (`VertexBuffer8::Lock()`) and callbacks. Instead, improved versions of these functions, originated for use in Visual Basic, are incorporated.

The COM Interfaces approach is generally valid, not only for DirectX, but for any software written in COM. But here the constraint is given by knowledge of interface description.

11 Acknowledgements

This work is a part of Microsoft Research Ltd. (U.K.): ROTOR project and was supported by the Ministry of Education of The Czech Republic – Project MSM 235200005.

Great thanks also belong, for a strong support, motivation and experience, to the supervisor of this thesis, Prof. Ing. Václav Skala, CSc.

12 Conclusion

Originally, the assignment of this diploma thesis seemed to be very easy. Problems arrived, when there was not found the first function declaration in its DLL. Unexpectedly, an extra effort had to be devoted to the COM technology, what in return helped to find a very elegant solution in this task.

At this work, it is provided in the second chapter the introduction with project ROTOR, C# language, CLI and both DirectX 8.1b and DirectX 9.0.

The bibliographic search gave very poor results, but satisfactory literature had been finally found, as described in the third chapter. The lesson is that only the Microsoft's MSDN is the most convenient source.

Thanks to a very strong investigation in programming manuals and developers guides, supported by uncountable experiments done in Visual Studio .NET, the very nice solution could be found and implemented with the minimal effort of routine slavery work. However it has its weaknesses – in one person it is an unimaginable deal.

With all the available knowledge, a very simple implementation design has been stated at the chapter *Implementation Design*, which immediately results from the previous chapter 4.

A designed verification is not so strong, because of missing adequate experience in software engineering, but is still sufficient. Proving details are given in chapter 6.1. There is also explained, why the little testing has been enough to decide that the solution is correct – some procedures had to remain observed to reach it. Hence, the chapter 6.2 describes the verification required in 6th point of the assignment.

DirectX interface implementation is described in chapter 7. There is also a screenshot of demonstration application to force-feedback joystick as compelled at point 5. Very valuable notes to implementation are stated here, every developer should be familiar with them.

The answer to point 7 is as general as it covers very heterogeneous software unit. Even though some proximal investigation had been carried out and with certain effort, several graphical functions were tested. It is justified why.

The user and program documentations are placed at appendixes part, source code is reasonably commented.

The implementation, documentation and source codes are marked as a freeware as a part of project ROTOR.

The discussion can be found in chapter 9.

Work on this thesis has been valuable to the author even from the reason that all the essential knowledge had to be collected while processing this job: COM objects, interfaces, Visual Basic, DirectX, .NET Framework. Before assigning the topic, all of these keywords were a quite mysterious to the author. Fortunately, it was found what benefits these modern technologies bring and understood in which points to be more careful.

Advantage of the described solution is a generality of investigated approach. Not only DirectX, but also any COM software can be handled by the same strategy now.

Useful Acronyms

API	Application Programming Interface
BCL	Base Class Library
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CLS	Common Language Specification
CTS	Common Type System
COM	Component Object Model
DLL	Dynamically Linked Library
GC	Garbage Collector
GDI	Graphics Device Interface
GUID	Globally Unique Identifier
IDE	Interactive Development Environment
IDL	Interface Definition Language
MIDL	Microsoft Interface Definition Language
MSDN	Microsoft Developer Network
MSIL	Microsoft Intermediate Language
ODL	Object Definition Language
OLE	Object Linking and Embedding
OS	Operating System
RCW	Runtime Callable Wrapper
SDK	Software Development Kit
UUID	Universally Unique Identifier
VES	Virtual Execution System

References

[Cen03] ***Centre of Computer Graphics and Data Visualisation.***
`http://herakles.zcu.cz/research.php`

[Csc03] ***C# Corner.***
`http://www.c-sharpcorner.com/Directx.asp`

[Han03] Hanák, I., Frank, M., Skala, V.: ***OpenGL and VTK interface for .NET.*** In C# and .NET Technologies 2003 proceedings, UNION Agency, Science Press, Plzeň, 2003.

[ECM02] ECMA TC39/TG2: ***C# Language Specification.*** Final Draft, ECMA Technical Committee 39 (TC39) Task Group 2 (TG2), (electronic resources), 2002.

`CD-ROM:/public/materials/references/manuals/C-Sharp/CSharp.zip`

[Kac01] Kačmář, D.: ***Programming .NET applications..*** (in Czech) Computer Press, Praha, 2001.

[MS01a] Microsoft Corp.: ***Overview of the .NET Framework.*** .NET Framework Developer's Guide, *MSDN* (electronic resources), 2001.

`http://msdn.microsoft.com/library/`
`CD-ROM:/public/materials/references/Overview of the _NET Framework.htm`

[MS01b] Microsoft Corp.: ***Common Language Infrastructure (CLI) Partition I: Concepts and Architecture***. Microsoft .NET Framework SDK Tool Developer's Documentation, Microsoft Corporation, 2001.

`http://msdn.microsoft.com/library/`
`CD-ROM:/public/materials/references/CLI/docs/Partition I Architecture.doc`

[MS01c] Microsoft Corp.: ***Common Language Infrastructure (CLI) Partition II: Metadata Definition and Semantics***. Microsoft .NET Framework SDK Tool Developer's Documentation, Microsoft Corporation, 2001.
`http://msdn.microsoft.com/library/`
`CD-ROM:/public/materials/references/CLI/docs/Partition II Metadata.doc`

[MS01d] Microsoft Corp.: ***COM Interop Part 1: C# Client Tutorial***. C# Programmer's Reference, MSDN (electronic resources), 2001.
`http://msdn.microsoft.com/library/`
`CD-ROM:/public/materials/references/COM Interop Part 1 C# Client Tutorial.htm`

[MS02a] Microsoft Corp.: ***The .NET architecture review.*** Microsoft's promotional material (Daniel Rubiolo et al), 2002.

[MS02b] Microsoft Corp.: ***Microsoft DirectX 9.0.*** Microsoft Corporation, 2002.
`http://msdn.microsoft.com/library/`
`CD-ROM:/public/materials/references/DirectX9/DirectX9_c.chm`

[MS02c] Microsoft Corp.: ***ODL File Syntax.*** Platform SDK: Automation, MSDN (electronic resources), 2002.
`http://msdn.microsoft.com/library/`
`CD-ROM:/public/materials/references/ODL File Syntax.htm`

[MS02d] Microsoft Corp.: ***module.*** Platform SDK: Automation, MSDN (electronic resources), 2002.
`http://msdn.microsoft.com/library/`
`CD-ROM:/public/materials/references/module.htm`

[MS03] Microsoft Corp.: ***Microsoft COM Technologies.*** 2003.
`http://www.microsoft.com/com/`

[MS03a] Microsoft Corp.: ***What's New in DirectX 9.0.*** 2003.
`http://msdn.microsoft.com/library/en-us/directx9_c/directx/graphics/whatsnew.asp`

[Sml03] Smlsal, T., Skala, V.: ***DirectX in C#.*** In C# and .NET Technologies 2003 proceedings, UNION Agency, Science Press, Plzeň, 2003.

[Vis03] ***Visual Studio .NET Documentation.*** `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vsstartpage.asp`

Annex A - Source Code to Force Feedback Support

```
// Tomas SMLSAL, 2003
// Supporting class to Joystick with Force-Feedback
//

using System;
using System.IO;
using System.Windows.Forms;

//using System.ComponentModel;
using DxVBLibA;
using System.Runtime.InteropServices;

namespace SpaceBreakout {
  /// <summary>
  /// Summary description for DInputFF.
  /// </summary>
  public class DInputFF {
    public const int TRUE = 1;
    public const int FALSE = 0;

    //-----------------------------------------------------------------------------
    // Global variables
    //-----------------------------------------------------------------------------
    DirectX8Class             DX8C         = new DirectX8Class(); //Whole Class
    DirectX8                  dx           = new DirectX8(); //DirectX 8 object
    DxVBLibA.D3DX8            g_pD3DX      = new DxVBLibA.D3DX8();

    public DirectInput8 di;                     //DirectInput object
    public DirectInputDevice8 diJoystick;       //DirectInput device object
    public DirectInputEnumDevices8 enumDevice; //DInput enumeration for devices object
    public DIDEVCAPS Caps;                      //store capabilities of the diJoystick
    public DirectInputEffect diEffect;          //Store the FF effect
    public DirectInputEffect diEffectLeft;      //Store the FF effect
    public DirectInputEffect diEffectRight;     //Store the FF effect
    public DIJOYSTATE2 diJoyState2;             //Joystick state.

    // do nothing in constructor..
    public DInputFF (){}


    //-----------------------------------------------------------------------------
    // Name: InitDirectInput()
    // Desc: Initialize the DirectInput variables.
    //-----------------------------------------------------------------------------
    public bool InitDirectInput(System.IntPtr hDlg) {
      int j;                                  // Count variable
      DIPROPLONG prop = new DIPROPLONG();     // Device property structure
      DirectInputEnumDeviceObjects diedo;     // Holds the collection of individual
                                              // objects on a device
      DirectInputDeviceObjectInstance didoi; // Holds the instance of an object on a
                                              // device
      int FFAxisCount = 0;                    // Holds the number of axis that support FF


      // Setup the g_EffectsList circular linked list
      //g_EffectsList = new ArrayList();

      // Create a DInput object
      try {
        di = dx.DirectInputCreate();// Create the direct input device
    if (di == null) {
      MessageBox.Show("dx.DirectInputCreate() Failed.");
      return false;
    }
  }
  catch (COMException e) {
    MessageBox.Show(e.Message+", HResult:0x"+e.ErrorCode.ToString("x"));
    return false;
  }
  catch (Exception e) {
    MessageBox.Show(e.Message);
    return false;
  }

  // Get the first enumerated force feedback device
  try {
    //di.CreateDevice((); // Enumerate all joysticks that are attached to the system
    enumDevice = di.GetDIDevices(CONST_DI8DEVICETYPE.DI8DEVCLASS_GAMECTRL,
      CONST_DIENUMDEVICESFLAGS.DIEDFL_ATTACHEDONLY
      | CONST_DIENUMDEVICESFLAGS.DIEDFL_FORCEFEEDBACK);
    if (enumDevice == null) {
      MessageBox.Show("di.GetDIDevices() Failed.");
      return false;
    }
    diJoystick = di.CreateDevice(enumDevice.GetItem(1).GetGuidInstance());
    if (diJoystick == null) {
      MessageBox.Show("di.CreateDevice() Failed.\nNo force feedback device found.");
      return false;
    }
    diJoystick.GetCapabilities(ref Caps); //Get the capabilites of the device
    // Get info about all the axis on the device
    diedo = diJoystick.GetDeviceObjectsEnum(CONST_DIDFTFLAGS.DIDFT_AXIS);
    if (diedo == null) {
      MessageBox.Show("diJoystick.GetDeviceObjectsEnum() Failed.");
      return false;
    }

    // This loops through to make sure that there
    // are at least two axis that support FF
    for (j=1; j<=diedo.GetCount(); j++){
      didoi = diedo.GetItem(j);
      if ((didoi!=null)
        && (( didoi.GetFlags() &
               CONST_DIDEVICEOBJINSTANCEFLAGS.DIDOI_FFACTUATOR )!=0)
        )
        FFAxisCount++;
    }

    if (FFAxisCount>1){
      // Set the format of the device to that of a joystick..
      diJoystick.SetCommonDataFormat(CONST_DICOMMONDATAFORMATS.DIFORMAT_JOYSTICK2);
      // Set the cooperative level of the device as an exclusive
      // background device, and attach it to the form's hwnd
      diJoystick.SetCooperativeLevel(hDlg.ToInt32(),
        CONST_DISCLFLAGS.DISCL_BACKGROUND
        | CONST_DISCLFLAGS.DISCL_EXCLUSIVE);

      prop.lData = 0;
      prop.lHow = (int)CONST_DIPHFLAGS.DIPH_DEVICE;
      prop.lObj = 0;
      IntPtr ip = (IntPtr)null;
      //          diJoystick.SetProperty("DIPROP_AUTOCENTER", ip);  // Turn off
      // autocenter
      diJoystick.Acquire(); // Make sure to aquire the device
      //          diEffect = diJoystick.CreateEffectFromFile("..//..//click1.ffe",
      //                         (int)CONST_DIFEFFLAGS.DIFEF_MODIFYIFNEEDED,
      //                                           "hsth");
      //          diJoystick.RunControlPanel(hDlg.ToInt32());

    }
    else {
      MessageBox.Show("Less than 2 force feedback axes.");
      return false;
    }

    //turn OFF the autocentering by playing a test-effect
    diEffect = diJoystick.CreateEffectFromFile("reset.ffe",
      (int)CONST_DIFEFFLAGS.DIFEF_MODIFYIFNEEDED,
      GetFirstFFENameFromFile("reset.ffe"));
    diEffect.Start(1, (int)CONST_DIESFLAGS.DIES_SOLO);

    //left bound
    diEffectLeft = diJoystick.CreateEffectFromFile("leva.ffe",
      (int)CONST_DIFEFFLAGS.DIFEF_MODIFYIFNEEDED,
      GetFirstFFENameFromFile("leva.ffe"));
    diEffectLeft.Start(-1, 0);
    //right bound
    diEffectRight = diJoystick.CreateEffectFromFile("prava.ffe",
      (int)CONST_DIFEFFLAGS.DIFEF_MODIFYIFNEEDED,
      GetFirstFFENameFromFile("prava.ffe"));
    diEffectRight.Start(-1, 0);


    //ok, download the needed effect
    diEffect = diJoystick.CreateEffectFromFile("crash.ffe",
      (int)CONST_DIFEFFLAGS.DIFEF_MODIFYIFNEEDED,
      GetFirstFFENameFromFile("crash.ffe"));


  }
  catch (COMException e) {
    MessageBox.Show(e.Message+", HResult:0x"+e.ErrorCode.ToString("x")
      +"\n FF joystick initialization failed.");
    return false;
  }
  catch (Exception e) {
    MessageBox.Show(e.Message
      +"\n FF joystick initialization failed.");
    return false;
  }
  return true;
}


//-----------------------------------------------------------------------------
// Name: FreeDirectInput()
// Desc: Frees the DI
//-----------------------------------------------------------------------------
public void FreeDirectInput(){
  // Release any DirectInputEffect objects.
  if (diJoystick != null){
    diJoystick.Unacquire();
    diJoystick = null;
  }

  // Release any DirectInput objects.
  di = null;
}
```

```
  //------------------------------------------------------------------------------
  // Name: PlayEffect()
  // Desc: Plays a FF file.
  //------------------------------------------------------------------------------
  public void PlayEffect(){
    try {
      if (diEffect != null)
        diEffect.Start(1, 0);
    }
    catch (COMException e) {
        MessageBox.Show(e.Message+", HResult:0x"+e.ErrorCode.ToString("x"));
    }
    catch (Exception e) {
      MessageBox.Show(e.Message);
    }
  }

  // retrieve necessary info from FFE file (an effect name)
  public string GetFirstFFENameFromFile(string Filename) {
    string effectName = null;
    try {
      StreamReader sr =
        new StreamReader(new FileStream(Filename,
                                        FileMode.Open,
                                        FileAccess.Read,
                                        FileShare.Read
                                        ));
      char[] buffer = new char[sr.BaseStream.Length];
      sr.Read(buffer, 0, (int)sr.BaseStream.Length);
      for (int i=0; (effectName == null) && (i<buffer.Length); i++) {
        if (  (buffer[i]=='e')&&(buffer[i+1]=='f')
            &&(buffer[i+2]=='c')&&(buffer[i+3]=='t')
          )
          for (int j=0; (buffer[i+4+j]!='\0')&&(j+i+4<buffer.Length); j++) {
            effectName += buffer[i+4+j];
          }
      }
      sr.Close();
    }
    catch (IOException e) {
      effectName = null;
      MessageBox.Show(e.Message);
    }
    return effectName;
  }
 }
}
```

Annex B - User Manual

Run-Time Requirements

DirectX 8.1 can be used in the Microsoft Windows® 98, Windows Me, Windows 2000, and Windows XP environments.

Description

The type library *DirectX 8 Visual Basic Type Library* is used as follows:

- First, add reference in references settings to this library, which has to be selected.
- Add the following line to the code: `using DxVBLibA;`
- To add required classes, interfaces or types, work with the namespace `DxVBLibA.`

For the exact parameters usage, see programs in the mediums `programs` directory.

The only suggested documentation is the MSDN – Graphics Development – DirectX – DirectX 8.1 (Visual Basic). Types are translated as given in .NET Framework documentation.

Annex C - Deployment Manual

It is necessary to install the .NET Framework, where are all necessary tools supporting runtime. The deployment itself is from principal done by copying an application. It is also clear that the correct version of DirectX has to be installed. Some developers find even difficult if the SDK version of DirectX runtime is missing.

Annex D - Program Manual (Developer Guide)

Since author did not find how to access functions hidden in type library modules, it is necessary to implement these supporting routines in a helper class. While using the type library approach, the implementation of this library is just prepared. If the library is not available, it is essential to re-declare the COM interfaces as follows in the IDirect3D8 example. Nearly all the time has been spent by trying attributes values, so there did not remained time to compose a handbook that would certainly specify the rules for translating the declarations from a header files, which is unfortunately manual:

```
// Declare I.. as a COM interface which
// derives from ??IDispatch interface:
[Guid("1DD9E8DA-1C77-4d40-B0CF-98FEFDFF9512")
,
InterfaceType(ComInterfaceType.InterfaceIsDual)]
public interface IDirect3D8 {   // Cannot list any base interfaces here
  // Note that IUnknown Interface members are NOT listed here:

  /*** IDirect3D8 methods ***/
  //void RegisterSoftwareDevice( [In] void* pInitializeFunction) ;
  void RegisterSoftwareDevice( [In] ref Object pInitializeFunction);//deprecated!!!
  uint GetAdapterCount();
  void GetAdapterIdentifier( [In] uint Adapter, [In] uint Flags, [Out] out
    DxVBLibA.D3DADAPTER_IDENTIFIER8 pIdentifier);
  uint GetAdapterModeCount( [In] uint Adapter) ;
  void EnumAdapterModes( [In] uint Adapter, [In] uint Mode, [In, Out] ref
    DxVBLibA.D3DDISPLAYMODE pMode) ;
  void GetAdapterDisplayMode( [In] uint Adapter, [In, Out] ref
    myDXTypeLib.D3DDISPLAYMODE pMode);
  void CheckDeviceType(uint Adapter, DxVBLibA.CONST_D3DDEVTYPE
    CheckType,DxVBLibA.CONST_D3DFORMAT DisplayFormat, DxVBLibA.CONST_D3DFORMAT
    BackBufferFormat,bool Windowed) ;
  void CheckDeviceFormat(uint Adapter, DxVBLibA.CONST_D3DDEVTYPE
    DeviceType,DxVBLibA.CONST_D3DFORMAT AdapterFormat,uint
    Usage,DxVBLibA.CONST_D3DRESOURCETYPE RType,DxVBLibA.CONST_D3DFORMAT CheckFormat) ;
  void CheckDeviceMultiSampleType(uint Adapter, DxVBLibA.CONST_D3DDEVTYPE
    DeviceType,DxVBLibA.CONST_D3DFORMAT SurfaceFormat,bool
    Windowed,DxVBLibA.CONST_D3DMULTISAMPLE_TYPE MultiSampleType) ;
  void CheckDepthStencilMatch(uint Adapter, DxVBLibA.CONST_D3DDEVTYPE
    DeviceType,DxVBLibA.CONST_D3DFORMAT AdapterFormat,DxVBLibA.CONST_D3DFORMAT
    RenderTargetFormat,DxVBLibA.CONST_D3DFORMAT DepthStencilFormat) ;
  void GetDeviceCaps(uint Adapter, DxVBLibA.CONST_D3DDEVTYPE
    DeviceType,DxVBLibA.D3DCAPS8 pCaps) ;
  IntPtr GetAdapterMonitor(uint Adapter) ;
  uint CreateDevice(uint Adapter,DxVBLibA.CONST_D3DDEVTYPE DeviceType,
                  IntPtr hFocusWindow,
                  uint BehaviorFlags,
                  ref DxVBLibA.D3DPRESENT_PARAMETERS pPresentationParameters,
                  ref DxVBLibA.Direct3DDevice8 ppReturnedDeviceInterface) ;
}
```

As long as it looks complicated, the most important is to preserve the order of functions of the interface.

Hereby I declare an agreement with this thesis to attendance loaning in university library at University of West Bohemia in Pilsen, for academic purposes.

Tomáš Smlsal: ....