Graphical Interfaces for C#*

Milan Frank[F], Ivo Hanák[H], Tomáš Smlsal[S], Václav Skala[@]
Department of Computer Science and Engineering
University of West Bohemia
Plzeň, Czech Republic

Abstract

A new environment called .NET was recently introduced to wide public. However, this environment does not contain libraries for advanced graphical output. Therefore it is necessary to make such libraries available to .NET. This paper describes our implementation of graphical library interfaces. The implementation allows cooperation of graphical interfaces with .NET. We have dealt with interfaces of libraries such as OpenGL, DirectX, and Visualization Toolkit (also known as VTK). A short description of libraries is included as well as a short introduction to .NET environment. We also present results, advantages, and disadvantages of our implementation. This paper contains neither comparison of the graphical libraries nor comparison of .NET environment and other environments.

Keywords: .NET, OpenGL, DirectX, Direct3D, VTK, C++ Managed Extension, C#

1 Introduction

This paper is focused on a pure implementation task rather than in explanation of a general theory or specific graphical algorithms. However, the described solution seems to be interesting and useful to people within a computer graphics community. The reason is simple: there are many people using the OpenGL, Visualization Toolkit [1] (VTK) or DirectX, who want to benefit from .NET Framework features. Simply, it is a runtime environment, which hides the operating system layer to the application and unifies single machine and network environments into one (see [2]). Later, we will describe what a .NET Framework stands for in more detail.

* Project supported by the Ministry of Education of the Czech Republic: Project MSM 235200005 and Microsoft Research Ltd.: Project ROTOR.

F mfrank@students.zcu.cz
H hanak@students.zcu.cz
S tscz@centrum.cz
@ skala@kiv.zcu.cz

1.1 The Basic Idea

The .NET Framework seemed to be very interesting for people from the area of computer graphics that we decided to implement some of the well-known graphical interfaces in it. The VTK, OpenGL and DirectX have been taken into account. These interfaces are widespread and having them 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 OpenGL can simply continue with a development with it, build it in .NET Framework and easily add whatever other network functionality he wants.

We have implemented the VTK, OpenGL and DirectX interfaces for use within the .NET Framework. It fulfills well our objectives given at the early beginning. Now, graphics developers can also work with the fully object oriented programming (OOP) language C#. It allows more inheritance, deriving and polymorphism into the computer graphics.

1.2 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 Common Language Infrastructure (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.

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. Actually, C# is really the native language of the .NET. However, .NET applications can be written in any language, which meets some requirements given by specifications (concretely 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 background information to be able to read the documentation provided by the Microsoft Company.

2 Graphical Interfaces

2.1 OpenGL

OpenGL is a graphic library based on commercial graphical system by SGI [3]. It is used in many applications including games and industry. It offers an interface and facilities for projecting 3D objects on 2D screen (or into image), but just the projection is not the only purpose of the library.

This library provides complete rendering pipeline, which handles lighting, texturing and transformations. The result of 3D world projection is then rendered to the given frame buffer or window of current (underlying) GUI (window system).

The object geometry can be specified using few basic render primitives, such as points, lines, triangles and quads. It includes support for triangle fans and strips. The library as a standard part of the rendering pipeline also provides light and lighting computation. It is possible to choose from common light types, such as point light, directional light and reflector and to set up their parameters. These light types are often supported by the hardware.

It is also possible to cover surface of every rendered primitive with user defined 2D or 1D texture. This includes support for multi-texturing and mip-mapping as a standard facility of the library.

Support for transformations is provided by composition of particular transformations, which are specified by matrices. These particular transformations can be set either by a user (e.g. in the form of matrix) or by standard library functions. Such functions offer a possibility to parameterize basic transformations, such as rotation around given axis, translation by a given vector and scaling by given coefficients.

OpenGL is aimed at visualization of 3D objects. It is also possible to use OpenGL for 2D output because the interface contains functions for setting and retrieving values of pixels inside a specified area (rectangle) at the target frame buffer. Unfortunately, these functions do not have good hardware support and their capabilities are not sufficient enough. E.g., they use nearest-neighbor approach when scaling due to which the visual result is not good. Due to that it is better to use simple 3D objects with texture mapping instead of these functions.

Interface is the most important part of a library. It is the only part visible to a user. In the case of OpenGL this interface consists of a group of functions and constants. These functions are not grouped into classes so it is also possible to use the library in a non-object-oriented languages (e.g., C).

Interface structure (i.e., contained functions and constants) is defined by specifications, which are open to the public. This specification also describes behavior and prescribed reaction of the library to calls of its interface functions. Important advantage of OpenGL interface is its full backward compatibility. Each new version neither adds a complete set of functions nor modifies existing ones. It just enlarges the existing set of functions by new ones. These additions usually follow features, which are implemented in the available hardware.

From inside view behavior of OpenGL is similar to a state machine. Each function (excluding function used to retrieve data and state) modifies the current state of the machine. This state then influences the result of the rendering.

Interface functionality and behavior are described by standardized specification. However, a real **implementation** is something a little bit different. It follows behavior described in the specification but in some cases (usually error handling) it slightly differs.

Some implementations provide robust and very stable background so they are able to absorb user's mistakes without any visible feedback while others strictly follow the specifications and in the case of such mistake they provide an unpredictable output.

This depends also on the used graphical hardware and sometimes on the used version of the device drivers. It can lead to difficulties while debugging when user develops his/her application using robust implementation and then get strange output using another, less robust.

OpenGL provides functionality for rendering of basic primitives with defined properties (e.g., lights, texture, etc.). Unfortunately, this functionality is sometimes not sufficient enough or its use is too difficult. Therefore together with OpenGL there exist several libraries (GLU, GLUT) or add-ons (GL Extensions).

One of these is the **GLU library**. This library provides facilities for rendering and tessellation of parametric surfaces as well as useful functions for setting projection.

Another case of such library is the **GLUT library**. It aims at simplification and unification of OpenGL initialization and its cooperation with currently available GUI. This is because OpenGL interface itself is

standardized by specifications while its initialization and cooperation with current GUI is not. Also OpenGL does not contain any facilities for input, library just handles output. Therefore the GLUT library provides environment, which unifies these tasks and makes source code portable to different platforms.

Add-ons such as **GL Extensions** were mentioned last. These are part of the OpenGL library and provide a possibility to use the latest hardware features, although they are not available in specifications yet. This makes GL Extensions heavily hardware dependent – since they are not part of the specification, each graphical hardware vendor usually creates his own set.

The interface consists of a set of static functions and numeric constants. Thanks to that, it is possible to have OpenGL available in various programming languages. Unfortunately, such construction of the interface can sometimes lead to not very readable source code. Also the use of GL Extensions can be a source of difficulties due to its hardware dependency.

To show how an actual source code using OpenGL looks like, there is a short and **simple example** (see Figure 1). This example (in C language) should provide a possibility to compare OpenGL source code with other introduced graphical interfaces. It does not contain any code used to cooperate with current GUI.

```
glClearColor(0, 0, 0, 0);
glClear(GL_COLOR_BUFFER_BIT);
glColor3d(1, 1, 1);
glBegin(GL_TRIANGLES);
glVertex3d(-1.0, -1.0, 0.0);
glVertex3d(0.0, 1.0, 0.0);
glVertex3d(1.0, -1.0, 0.0);
glEnd();
glFlush();
```

Figure 1 - Example of OpenGL source code

The output of the example is a white triangle on black background. Fist function in the example sets color (black) for clearing the background. Then the background is cleared with such color. Afterwards the color of the triangle is set. The following block of the source code defines the triangle (i.e., sets coordinates of its vertices). The last function ensures that all functions above are actually performed (i.e., they do not stay waiting in the queue or buffer to be performed later).

2.2 DirectX Graphics = Direct3D

DirectX is a set of application interfaces (APIs), which take the advantage of device independent functions to simplify game related tasks, performed by the computer. The DirectX API handles most of the I/O aspects you need 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.

DirectX provides a low-level access to hardware functionality of available peripheral hardware devices, such as graphical adapter, sound card, etc. Very important fact is that all this technology is based on a Component Object Model (COM). 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 handle all 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.)

To get idea of DirectX usage, see Figure 2 (code snippet that draws a triangle). This code draws a triangle, each vertex in a different color. The inside area pixels are colored by interpolating colors in triangle vertices. First, we get a vertex buffer to draw all scene vertices on a graphic card. Second, an adequate stream is created for our vertex buffer, which we have to lock for our job. On following lines is only declaration of a three-element array that includes coordinates and opacity of the three vertices of our triangle. Finally, we have to send the vertices to a device and unlock the vertex buffer. Other code snippets are presented in [4].

```
VertexBuffer vb =
  (VertexBuffer) sender;
GraphicsStream stm =
  vb.Lock(0, 0, 0);
CustomVertex.TransformedColored[]
  verts = new CustomVertex.\
    TransformedColored[3];

verts[0].X=150; verts[0].Y=150;
verts[0].Z=0.5f; verts[0].Rhw=1;
Verts[0].Color =
  System.Drawing.Color.Aqua.ToArgb();

verts[1].X=150; verts[1].Y=150;
verts[1].Z=0.5f; verts[1].Rhw=1;
Verts[1].Color =
  System.Drawing.Color.Brown.ToArgb();

verts[2].X=150; verts[2].Y=150;
verts[2].Z=0.5f; verts[2].Rhw=1;
Verts[2].Color =
  System.Drawing.Color.Blue.ToArgb();

stm.Write(verts);
vb.Unlock();
```

Figure 2 - A triangle code snippet of DirectX

On December 2002, Microsoft has released the **DirectX 9.0 (Managed)** version of the DirectX, which should meet all the requirements stated at the beginning of this paper. Thus it is used as a reference for comparison to reached results. In the following sentences 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 matrix, clipping plane, quaternion, vector manipulation 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 8.1 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 Figure 3. Advanced information for DirectX .NET development is available in [5] and [6].

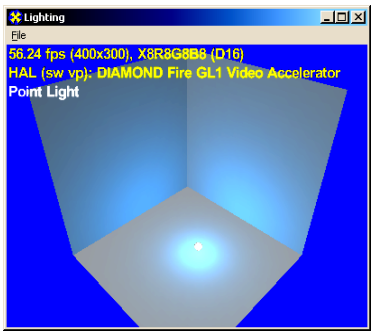

Figure 3 - The Lighting sample

2.3 VTK

The VTK (*Visualization Toolkit*) is an object-oriented library aimed at data visualization. As a free source project it is developed and supported by Kitware Inc. On the web [7], full version with documentation can be downloaded. The VTK contains wide variety of algorithms, exporters, importers, renderers and also classes for data representation. In contrast to OpenGL and Direct3D, the VTK is higher-level and more specialized on the data visualization. E.g. hard to imagine is to make a fast 3D game engine (as in the case of Quake) in VTK.

The current version of VTK can be installed on MS Windows and almost all UNIX based systems. Therefore, there is a good portability on source code level. Developers can also select from number of programming languages to use. C++ is the VTK native language and so programs developed in this language are the most efficient. On the other hand, Java, TCL and Python can be used by means of appropriate wrappers that are part of the distribution. These wrappers have some limitations due to differences between C++ and wrapper languages (C++ is the most general).

The main idea of the VTK is the **visualization pipeline**. It means that there are some sources of data, which are passed into some kind of filter that processes it and finally the data flows to exporter and/or renderer as the output. Better view of this idea can be given by the following example.

The mace example is similar to well known ”Hello world” application for the VTK. The output is a sphere in polygonal representation with a cone on each vertex normal; see Figure 5. It is displayed in renderer with interactor that allows simple manipulation (rotation, translation, etc.) of the resulting mace by mouse. The appropriate visualization pipeline is given as a graph on Figure 4.

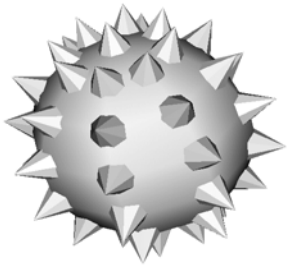

Figure 5 - The mace output

In Figure 4 we can see two sources of polygonal data, the sphere source and the cone source. Both pass their data to the vtkGlyph3D that creates copies of the cone on each vertex of the sphere with orientation of the appropriate vertex normal. The resulting ”spikes” as polygonal data and also the sphere data are passed to the sphere and spike mapper. Mappers are terminal objects that prepare the data for data for rendering (actually they

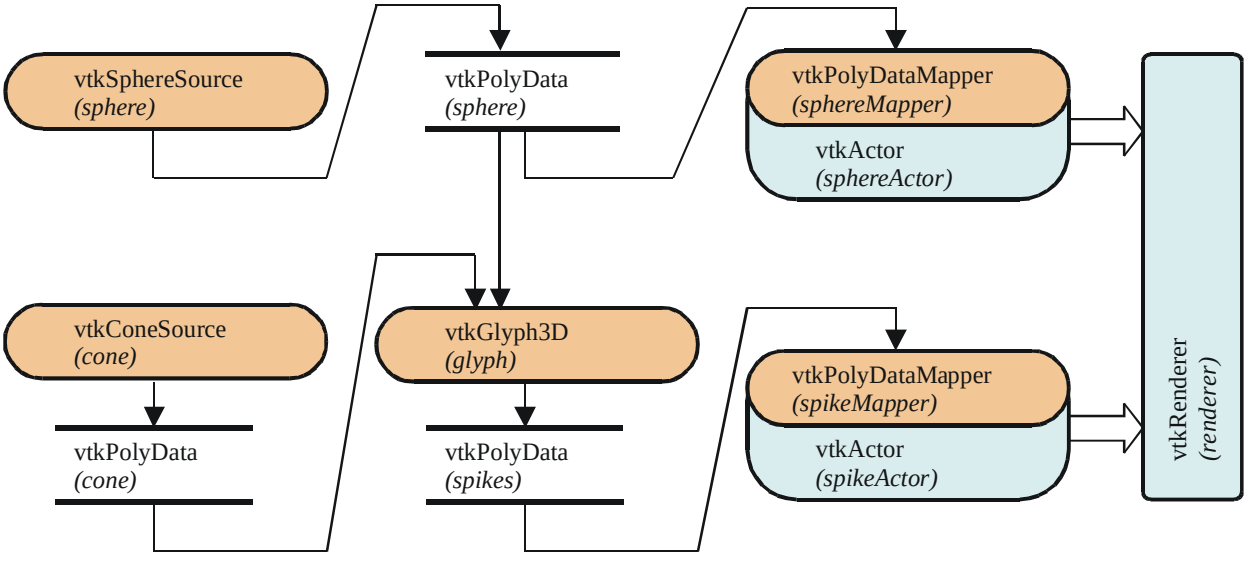

Figure 4 - The mace visualization pipeline given as graph

have no output). But renderer itself can visualize actors only. Therefore actors wrap mappers and actors are passed to the renderer. Fully functional source code in C# of this example is given on Figure 6.

```
vtkSphereSource sphere
        = vtkSphereSource  .New();
  sphere .SetThetaResolution(  6);
  sphere .SetPhiResolution(  6);
vtkPolyDataMapper sphereMapper
        = vtkPolyDataMapper  .New();
  sphereMapper  .SetInput(sphere  .GetOutput());
vtkActor sphereActor   = vtkActor .New();
  sphereActor  .SetMapper(sphereMapper);

vtkConeSource cone   = vtkConeSource  .New();
  cone .SetResolution(  6);

vtkGlyph3D glyph   = vtkGlyph3D  .New();
  glyph .SetInput(sphere  .GetOutput());
  g lyph .SetSource(cone  .GetOutput());
  glyph .SetVectorModeToUseNormal();
  glyph .SetScaleModeToScaleByVector();
  glyph .SetScaleFactor(  0.25f );

vtkPolyDataMapper spikeMapper
        = vtkPolyDataMapper  .New();
  spikeMapper  .SetInput(glyph  .GetOutput());

vtkActor spik  eActor  = vtkActor .New();
  spikeActor  .SetMapper(spikeMapper);

vtkRenderer renderer   = vtkRenderer  .New();
  renderer  .AddActor(sphereActor);
  renderer  .AddActor(spikeActor);
  renderer  .SetBackground(  1,1,1);

vtkRenderWindow renWin
        = vtkRenderWindow  .New();
  renWin .AddRenderer(renderer);
  renWin .SetSize( 450,450 );
vtkRenderWindowInteractor iren
        = vtkRenderWindowInteractor   .New();
  iren .SetRenderWindow(renWin);

// interact with data
renWin .Render();
iren .Start();
```

Figure 6 - The mace C# source code (VTK)

In Figure 6, first we can see creation of the sphere-source and its mapping as the *sphereMapper*. The *sphereMapper.SetInput(sphere.GetOutput())* method call makes the interconnection between sphere source and sphere mapper. Appropriate actor *(sphereActor)* is created and assigned just behind the mapper. The creation of cone-source object follows. Its output is connected as a source input of the glyph. The second input of the glyph is output from sphere-source. The spike-mapper maps the output from the glyph with spike-actor assigned. Finally the renderer with render-window and interactor are created. The command *renWin.Render();* causes the pipeline execution and appropriate result is displayed in renderer. The command *iren.Start()* contains the event loop handling and therefore an interaction of user with the renderer is possible.

Here, we would like to point out the ”lazy execution”. Until the render command is called, no computation is executed. When some output is required, the process object asks its sources if there are any changes in their state and possibly updates its own state. It passes recursively through all visualization pipeline elements.

Now we present **the triangle** example to maintain the integrity with other parts of this article. As we mentioned before, the VTK is higher-level library than OpenGL or Direct3D. On the other hand, it is an interesting comparison between such different approaches that solve the same problem. The main part of the source code in C# can be seen in Figure 7. Only the renderer and actor creation is omitted.

```
vtkPoints vertices = vtkPoints.New();
  vertices.SetNumberOfPoints(3);
  vertices.SetPoint(0, 0.0, 0.0, 0.0);
  vertices.SetPoint(1, 1.0, 0.0, 0.0);
  vertices.SetPoint(2, 1.0, 1.0, 0.0);

vtkTriangle triangle = vtkTriangle.New();
  triangle.GetPointIds().SetId(0, 0);
  triangle.GetPointIds().SetId(1, 1);
  triangle.GetPointIds().SetId(2, 2);

vtkPolyData polyData = vtkPolyData.New();
  polyData.SetPoints(vertices);
  polyData.Allocate(1, 1);
  polyData.InsertNextCell(
        triangle.GetCellType(),
        triangle.GetPointIds());

vtkPolyDataMapper mapper =
        vtkPolyDataMapper.New();
  mapper.SetInput(polyData);
```

Figure 7 - The triangle C# source code (VTK)

First there is a creation of a vertex array with definition of coordinates called *vertices*. The *triangle* object itself is actually array of indices to the vertices array. The creation of the triangle is just after the vertices creation. Next the polygonal data are created and vertices with the triangle indices are passed into the object. Finally these polygonal data are mapped to the mapper that can be processed as mappers in previously presented example.

The triangle object is only one from a set of possible polygonal data elements. Actually it can be any correctly derived class from *vtkCell* class. But detailed description of the data representation in VTK is beyond the scope of this article.

3 Implementation and Approach

This section contains description of our approach for graphical interfaces mentioned in previous section. One of the possible approaches for porting a library to .NET is the use of **wrappers**. 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).

3.1 OpenGL

This subsection contains description of the OpenGL approach. First, an existing solution (CsGL) is described to show current state of the art. Next is our approach and its advantages and disadvantages. At the end of this subsection a short example of source code is included.

OpenGL library has currently an **existing solution** for .NET environment. It is called CsGL [8], it is an Open Source library. First version of CsGL library was released at August 14, 2001.

It is a wrapper of an interface and is implemented in C and C#. The C source code provides connectivity to underlying GUI and C# code provides OpenGL/GLU functionality.

It uses PInvoke mechanism – i.e. .NET environment handles data sharing and function calling between itself and outside world. The only things, which need to be specified, are function headers.

As it is notified in documentation, this combination of languages should simplify porting to another platform.

OpenGL/GLU functions and constants are static members of one class. This approach is based on the fact that a recommended use of CsGL is via inheritance. It means that user has to inherit his own classes from CsGL class to gain full access to all OpenGL functions. Ported OpenGL code then looks quite the same as it would be in plain C.

Currently CsGL has full implementation of OpenGL up to version 1.4, a complete GLU and approximately 50 GL Extensions including tool for their porting.

The **main goal of our implementation** is the same as for CsGL: to create an OpenGL library wrapper. However, unlike CsGL, there is an aim on programming safety. To achieve it, parameter checking will be added inside the wrapper. This will prevent user from passing invalid data structures or arrays (e.g. arrays of wrong length).

Next, our implementation will avoid use of *IntPtr* data type that has quite the same meaning as void pointer in non-managed environment. This will prevent user from passing data structures or arrays of invalid data type (e.g., passing an array of references instead of an array of doubles or integers).

Next, enumeration data types (the enums) will replace constants. This will prevent user from passing invalid constants without need of additional code inside functions wrappers. Also it will increase a comfort of the interface because it is a little self-documenting: the user does not need to exactly know, which constant is needed, because he have to choose only from a group of the constant specified by the function parameter data type.

An important aim of our interface implementation is to provide full comfort of managed environment. The result will be interface where using it does not need any knowledge about managed and non-managed environment cooperation.

This comfort will be achieved by completely avoiding *IntPtr* data type, by using enums instead of constants and by adding additional data structures. These additional structures will serve as a replacement of non-managed code capability to look on one block of a memory with different views.

Next important goal of the implementation is to make the slowdown (due to wrapping) as low as possible because the features described above (e.g., parameter checking) lead to additional code inside the wrapper.

Everything that was described above will create an interface of OpenGL/GLU library, which is comfortable, has high programming safety (especially for non-experienced users) and is as close to the original OpenGL/GLU specification as possible.

Let us describe briefly **the most important difficulties** of creating implementation described above. The major difficulty is data sharing. Sharing data between managed and unmanaged code means to pass non-managed pointer to a memory block outside the managed environment, and a problem can happen, when the garbage collector removes or moves with the referenced block of memory. After that, the passed pointer becomes invalid and operations with memory using such pointer can result in an application crash.

Next difficulty is due to void pointer. In this case it is also a matter of programming safety and a replacement needs additional helpful data structures.

Implementation is performed in C++ Managed Extensions (MC++), which provide better cooperation among non-managed, managed code and them.

Implementation follows goals and solves difficulties described above. The structure of the interface implementation is designed to fulfill the described goals. Also possibility for users to use only that version of OpenGL/GLU that is sufficient for his/her needs influenced the design.

The result contains four groups of classes, which provide:

- Underlying GUI and OpenGL library connectivity.
- OpenGL/GLU functions. Each version is placed into a separate class that is inherited from the previous version.
- Internal data structures. These classes are internal and therefore transparent for a user. They are the only one, which need to be modified when adding a new version of OpenGL/GLU interface.
- Other helpful classes and data structures.

The most important ones are classes containing OpenGL/GLU functions. These classes inherit from classes of previous OpenGL/GLU versions to maintain backward version compatibility. These classes contain two sets of functions (constants), which differ only by their names.

First one is similar to original OpenGL interface; second one is the modified. This modification is based on removal of '*gl*' (functions) and '*GL_*' (constants) prefix to

give the source code better look as it is shown in examples (see Figure 8).

A
```
glClearColor(0, 0, 0, 0);
glClear(GL_COLOR_BUFFER_BIT);
glColor3d(1, 1, 1);
glBegin(GL_TRIANGLES);
glVertex3d(-1.0, -1.0, 0.0);
glVertex3d(0.0, 1.0, 0.0);
glVertex3d(1.0, -1.0, 0.0);
glEnd();
glFlush();
```

B
```
GL.glClearColor(0, 0, 0, 0);
GL.glClear(GL.GL_COLOR_BUFFER_BIT);
GL.glColor3d(1, 1, 1);
GL.glBegin(GL.GL_TRIANGLES);
GL.glVertex3d(-1.0, -1.0, 0.0);
GL.glVertex3d(0.0, 1.0, 0.0);
GL.glVertex3d(1.0, -1.0, 0.0);
GL.glEnd();
GL.glFlush();
```

C
```
gl.glClearColor(0, 0, 0, 0);
gl.glClear(GL.GL_COLOR_BUFFER_BIT);
gl.glColor3d(1, 1, 1);
gl.glBegin(GL.GL_TRIANGLES);
gl.glVertex3d(-1.0, -1.0, 0.0);
gl.glVertex3d(0.0, 1.0, 0.0);
gl.glVertex3d(1.0, -1.0, 0.0);
gl.glEnd();
gl.glFlush();
```

D
```
gl.ClearColor(0, 0, 0, 0);
gl.Clear(GL.COLOR_BUFFER_BIT);
gl.Color3d(1, 1, 1);
gl.Begin(GL.TRIANGLES);
gl.Vertex3d(-1.0, -1.0, 0.0);
gl.Vertex3d(0.0, 1.0, 0.0);
gl.Vertex3d(1.0, -1.0, 0.0);
gl.End();
gl.Flush();
```

Figure 8 - Source code example.

The code in the example above does exactly the same as code in Figure 1. It shall provide a possibility to compare original C code (also CsGL code using inheritance) (A), CsGL code without use of inheritance (B), our implementation using first set of names (C), and the same with the second set of names (D). The *gl* variable contains reference to an instance of *GL* class.

Recommended use of the interface is via composition. Functions are not static members of the class so there is a need to create class instance in order to use OpenGL/GLU functions.

It is possible to use inheritance, but it is not possible to gain both OpenGL and GLU functionality (constants) by using this approach because of managed environment allows only one parent per class.

Currently there is a full implementation of OpenGL and GLU version 1.1. Higher versions and GL Extensions will be implemented using a generator, which is under development. This generator will provide possibility to automate the task of porting of new GL Extensions and new OpenGL versions.

3.2 DirectX

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.

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 [9]. 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.

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.

All that developer has to do is a COM interfaces **registration**. Therefore the problem of wrapping is not as difficult as in [9] and it is not necessary to deal with some specific problems. In the following paragraphs, important procedures of how to do it will be described.

COM coclasses is the first general approach that uses the COM interoperability. Essentials about COM interoperability can be found in [10]. At first, a list of all component interfaces is taken and for each interface is done registration as in the Figure 9.

```
using System;
using System.Runtime.InteropServices;

namespace Sample {

[ComImport,
Guid("3BBA0080-2421-11CF-A31A-\
      00AA00B93356")]

class IDirect3D {}

class Test {
  static void oldMain() {
    //Create a COM object wrapper..
    IDirect3D iDirect3D =
      new IDirect3D();
  }
}
...
```

Figure 9 - COM registration

Immediately after a declaration, the COM object is ready for initialization and use. Instantiating an interface instance causes a corresponding COM instantiation and all its methods are called via the interface reference. The list of necessary GUIDs (Globally Unique IDentifiers) can be retrieved from header files of DirectX SDK, downloadable from [2].

This is the most general way of obtaining DirectX functionality in the sense that only necessary parts of the interface are included. Another significant reason for this strategy can be higher level of freedom while mixing components from several versions. However, it is not possible to combine different versions of one component, but it is possible to mix different components, each one from only one version. If, for some reason, a developer needs graphical capabilities of Direct3D version 9.0 and sound from DirectX 6.0, the instructions given above will help him to solve this task efficiently.

Compared to previous approach, a type library (DxVbLib) solution gives a complete functionality of DirectX by a single command. All to do here is to add a reference to Visual Basic DirectX Type Library named DxVBLib.dll and since it is done, the whole functionality is available through instantiating the needed objects and their references.

In the following example (Figure 10) it is shown on creating a Direct3D8 object that supports enumeration and allows the creation of Direct3DDevice8 objects.

```
using DxVBLib;
// also add manually a ref.
//in options

namespace Sample {
...
public class MyClass {
  ...
  private Direct3D8 g_pD3D;
  ...
  g_pD3D =
    this.DirectX.Direct3DCreate8(
      D3D_SDK_VERSION);
  ...
}
...
}
```

Figure 10 - snippet for type library use

Until the version DirectX 9.0 was released, this was the simplest method how to implement DirectX in .NET Framework.

3.3 VTK

Our main goal with VTK is to allow the user its straightforward use in the managed (.NET) environment together with programming safety and comfort of the managed environment. It is necessary to make the unmanaged (Win32) classes accessible from the managed environment and to provide correct data type conversions between these two different worlds. As a reasonable solution the interfacing layer between managed application and unmanaged libraries seems. Simply said, the application uses the interface and the interface passes its request (with appropriate data conversions) to the unmanaged libraries. A possible application scheme is given in Figure 11.

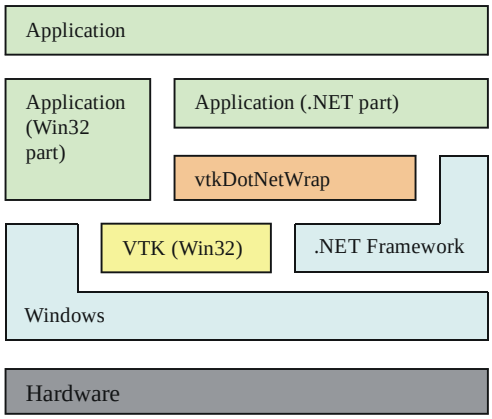

Figure 11 – A possible scheme of a VTK application in .NET environment

In the following paragraphs we present a short overview how the interface is created. The knowledge of that could be good for more efficient use of the interface.

To make unmanaged classes accessible from the managed environment, it is necessary to make managed **wrap-class** for each unmanaged class we want to access. The managed wrap-class contains an appropriate instance of unmanaged class. Its methods call methods of the unmanaged class. See a simplified part of wrap-class source code in Figure 12.

```
public __gc class vtkAbstractMapper :
public vtkProcessObject // wrap-class
{
  // wrapped-class
  ::vtkAbstractMapper *w;

  // wrapped method
  System::String * GetClassName()
  {
    return new System::String(
               w->GetClassName());
  }
```

Figure 12 - Simplified MC++ wrap-class source code

Each wrap-class is written in C++ Managed Extension (MC++). The MC++ is extended C++ to allow programmer use the new features of managed environment. The keyword *__gc* is typical. It marks the class as garbage-collected. In the simplest meaning, delete operator need not be called. Things are much more complicated than this but as the first overview it should be enough.

The generator or another process that automates wrap-class creation has to be used due to the VTK size (Approximately 700 classes with 16000 public and protected class members).

We divided the process of generating into two parts, parsing and generating. The parsing gets appropriate information about VTK classes from its C++ header files. The generator uses the information from the parser and generates wrap-classes source codes.

4 Results

4.1 OpenGL

OpenGL interface implementation is currently in the state of testing and further development. It has been tested on a simple function to test the interface functionality.

We expected a slowdown due to the method that was used to create the port of the interface library, i.e. wrappers. But not only because of that, also due to the fact that the wrapper contains an additional code to improve the programming safety.

To prove our expectation, we performed three tests. These tests measure slowdown of the **CsGL** and **our implementation**, relatively to the **original OpenGL** library (i.e., common OpenGL Win32 application).

The **first test** was calling a function *glVertex2d*. This is an example of the function with value data types as parameters only. We expected such functions to be called most often. As you can see at Figure 13, our implementation is a little bit faster then CsGL and the slowdown is a very small.

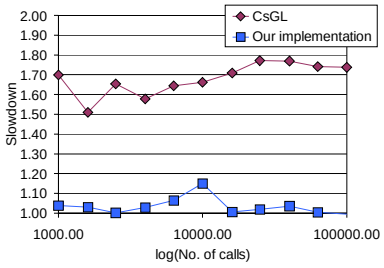

Figure 13 - Results of the glVertex2d test

The **second test** deals with functions that have an array type as a parameter and such passed array is not stored inside the OpenGL. For this test the function *glTexImage2D* was used. Again, it is compared to original OpenGL library.

As you can see at Figure 14, slowdown of our implementation is higher then CsGL. This slowdown is caused by the parameter checking because without it the results are close to CsGL.

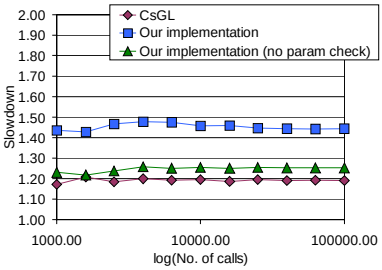

Figure 14 - Results of the glTexImage2d test

The third test was aimed at a function with an array type as a parameter that is stored inside OpenGL library in the form of pointer for a future use. An example of such a functions is *glVertexPointer*.

As you can see at Figure 15, the slowdown is quite significant even for a version without parameter checking. This is caused by the fact that our implementation uses generic collections for storing internal data.

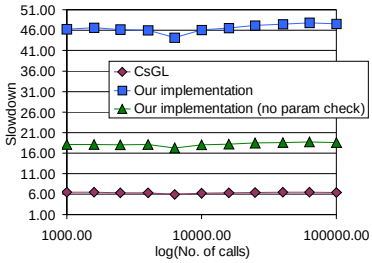

Figure 15 - Results of the glVertexPointer test

However, the functions of such kind are not called, in comparison with the previous ones, too often. Due to this, the future whole code should not be so significantly slow.

Unlike the CsGL code, using our implementation does not need unsafe code blocks. Application without unsafe code blocks is considered from a viewpoint of .NET to be more secure. The absence of unsafe blocks also increases programming comfort (i.e., no pointers). In C# language this means that only references to objects are needed.

4.2 DirectX

To provide some information about speed performance, sample codes of original DirectX8.1b in C++ have been compared to our modified DirectX9.0 .NET Framework version. The machine configuration was as follows: two Intel Pentium III / 500MHz, 1GB ECC SDRAM, Diamond Fire GL1 Video Accelerator PCI, OS Windows2000, 400×300×32 window mode.

The method of measurement was determining the number of rendered frames per second. Then, from the

average for each test, we calculated the time in milliseconds with precision provided by the number of decimal digits; see results in Table 1.

Sample type	C# .NET	C++
Billboarding	27,9	23,2
Clipping	10,3	9,4
Vertex shader	15,6	14,0
Enhanced mesh	9,1	6,6
Lights	17,0	23,4
Vertex shader	7,2	6,3

Table 1 - Time [ms] to render the tested scene

Plotted to Figure 16, it is obvious that overhead of C# is acceptable in most cases with as with the exception of the billboarding and enhanced mesh tests, where the results point, to better C++ compiler. It was surprising that DirectX in C# was faster at the lighting test.

During the tests even some MS SDK samples crashed. In **future work**, we want to discover why some errors occurred, e.g. presenting error exceptions, immediate quitting, machine deadlocks (without any notice), and some automatic machine reboots. Finally, about 60% of samples worked well.

4.3 VTK

The result of our work on VTK is one interfacing managed assembly (dll) that can be easily added to any managed (C#, etc.) project. This interfacing assembly calls the original VTK libraries. As we mentioned before, the presented interface provides only a subset of the VTK functionality as any other interfaces to other environment and/or programming languages.

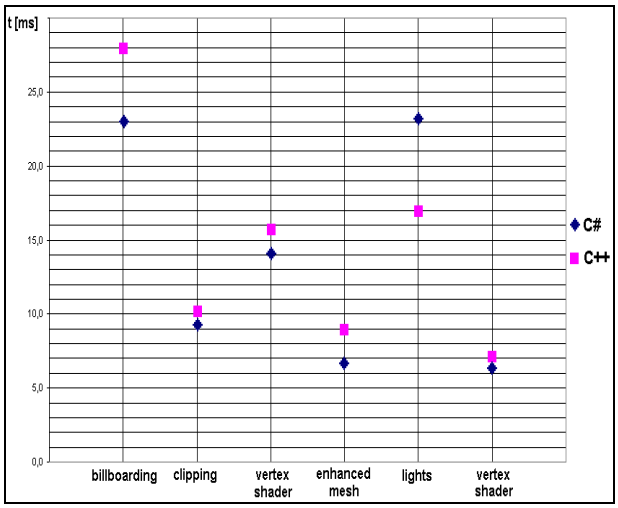

Figure 16 - Time [ms] to render scene.

We already **tested** the presented interface on a part of testing samples that are part of the VTK distribution and no serious troubles have been encountered.

Currently large testing process is in progress. With 15 colleagues, we are preparing a set of examples from all main parts of VTK application. It is a practical work in one of the courses supervised by the Center of Computer Graphics and Data Visualization on the University of West Bohemia in Plzeň, Czech Republic.

It is clear the added interfacing layer produces some slowdown in the application run. Here we present results of the **slowdown measuring**. There is comparison between the managed application in C# that uses our interface and the unmanaged application in C++ that calls the VTK libraries directly. Absolute times are given in Table 2.

Program	Time C# [s]	Time C++ [s]
Mace	0.29	0.31
FrustrumClip	0.54	0.38
ExpCos	2.96	2.28
PointLocator	0.59	0.30
Rgrid	0.34	0.34
IFlamigm	0.65	0.94
VolProt	6.48	7.27

Table 2 - Slowdown measuring between managed C# and native C++

As we can see, the slowdown is about 20%. In our opinion it is not crucial because in the VTK we are hunting the developing speed and easiness, not the execution speed.

The tested samples are usually taken from the VTK source codes and manuals. The first one is the same mace as the one presented in the section 2.3. The FrustrumClip is a kind of 3D clipping. The ExpCos example is a 300x300 grid waved by the ”Mexican-hat” function. The PointLocator is a kind of the 3D nearest-neighbor search. The RGrid is ”manually” created regular grid displayed as a lines. Test of 3DS file importing and rendering provide the IFlamigm example. Finally, the VolProt tests wide variety of direct volume rendering in one window. The graphical outputs can be seen in Figure 17.

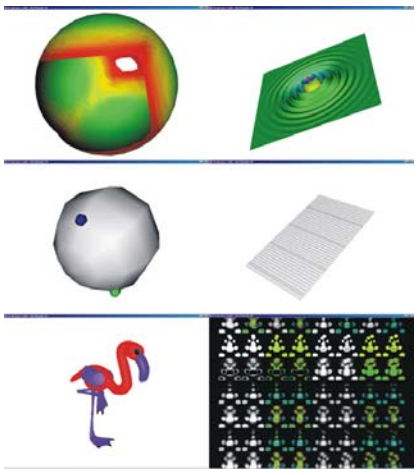

Figure 17 - Testing examples: FrustrumClip, ExpCos, PointLocator, RGrid, IFlamigm, VolProt

As any other VTK interfaces, also the presented one does not provide a straightforward inheritance possibility. So the user cannot simply derive his own class from any VTK class in .NET environment. For the user it means he can only use already created VTK modules and cannot create his own modules. Finally we would like to point out that (as far as we know) it is possible in unmanaged C++ only.

As a **future work** we would like to finish the testing process with students. We are considering to allow the user to make his own functional VTK objects in managed environment by means of two-level wrapping. Full description of the problem can be found in our paper [9].

5 Conclusion

Our goal is not to mutually compare interfaces to tell which one is the best. What we want to achieve is to create ports of the presented libraries to .NET environment or to try a little bit different approach than existing ones (as for OpenGL).

The aim of our work is programming safety and comfort of the use of the ported interfaces. We want to make our ported interface to have pure .NET look, i.e. to avoid using of unmanaged blocks of code in order to communicate with the interface.

The future work, as it was mentioned in previous sections, is aimed at improving functionality, stability and safety of implementations. Currently our implementations are in state of testing and further improving functionality. They are already usable, but shall not be considered to be completely error-proof yet.

Presented works are our diploma thesis. Release versions are expected at the beginning of June. All the presented project will be published as OpenSource because it is a part of the ROTOR project [12].

Acknowledgements

This work is a part of Microsoft Research Ltd. (U.K.): ROTOR project [12] and was supported by the Ministry of Education of The Czech Republic – Project MSM 235200005.

We also wish to thank doc. Ivana Kolingerová for advises and emendation.

References

[1] Schreder, W., Martin, K., Lorensen, B.: *The Visualiasation Toolkit.* Prentice Hall, New Jersey, 1998.

[2] Kačmář, D.: *Programujeme .NET aplikace.* (in Czech) Computer Press, Praha, 2001.

[3] *SGI: OpenGL 1.3 specification.* http://www.opengl.org/

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

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

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

[7] *Home pages of VTK.* http://www.kitware.com/vtk/

[8] *CsGL project documentation*. http://csgl.sourceforge.net/

[9] 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.

[10] *Microsoft COM Technologies.* http://www.microsoft.com/com/

[11] *MSDN* (electronic resources). http://msdn.microsoft.com/library/

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