proceedings of C# and .NET Technologies'2003 Int.Workshop, ISBN 80-90301-3-6, UNION Agency – Science Press, Plzen, Czech Republic, pp.25-34, 2003

OpenGL and VTK interface for .NET

Ivo HANÁK
University of West Bohemia
Univerzitní 8, BOX 314
Pilsen, Czech Republic
hanak@students.zcu.cz

Milan FRANK
University of West Bohemia
Univerzitní 8, BOX 314
Pilsen, Czech Republic
mfrank@students.zcu.cz

Václav SKALA
University of West Bohemia
Univerzitní 8, BOX 314
Pilsen, Czech Republic
skala@kiv.zcu.cz

ABSTRACT

VTK (visualization toolkit) is a large and useful object oriented library for data visualization. The current version (4.0) is also distributed for Win32 platform by means of dynamic linked libraries. Its native language is C++. An interface for Java, Python and TCL exists. These languages provide some subset of the VTK functionality only. Therefore the best use is with C++.

OpenGL is well known library for graphics output used in large scale of applications. Interface and behavior of the library is defined in specifications available to wild public. Inner implementation of the interface is matter of operating system and/or graphics hardware providers. Currently existing interface implementations are ready to use within various programming languages.

There is necessary to automate the process of wrap-class creation because of the VTK size (more then 700 classes). This process consists of two parts - parsing of C++ headers and generating of appropriate wrap-classes. The parser is distributed with VTK and it is used to generate Java, TCL and Python interfaces and so it is possible to use it in the case of C#.

It is not possible for the library user to use inheritance and polymorphism when using the manner described above. A possible way is to use two-level wrapping. The level-one wrap-class is unmanaged and provides calling of managed virtual methods and makes protected methods accessible for level-two wrap-class where the direct inheritance is used. The level-two wrap-class is managed and has the same functionality as described above.

One of the aims of this work is to find some way for straightforward use of VTK in C#. It has been done by means of wrap-classes written in *C++ Managed Extension*. Each VTK class has its own wrap-class. This wrap-class is managed and provides access into methods of unmanaged VTK class. Data conversion and memory management are also matters of wrap-class.

Second aim of this work is to create an OpenGL port to .NET environment by wrapping an existing interface. The goal of this part is to compare it with already existing OpenGL interface implementation called CsGL. This interface is fully functional and it is based on the similar principles as of this work. This work tries to go a little bit further to increase programming safety and user's comfort.

The presented paper is an introduction and description of this work's approach. The goal of this work is to try to find a reasonable way of VTK and OpenGL porting into .NET environment.

Keywords

VTK, visualization toolkit, OpenGL, CsGL, .NET, interface, C#, porting

1st Int.Workshop on C# and .NET Technologies on Algorithms, Computer Graphics, Visualization, Computer Vision and Distributed Computing
February 6-8, 2003, Plzen, Czech Republic.

ISBN 80-903100-3-6

1 INTRODUCTION

New .NET environment (also managed environment) offers to user a lot of possibilities how to create an application easily. Unfortunately some useful libraries, known at non-managed environment, do not exist in managed one. Difficulty of porting of non-managed libraries to the managed environment differs form case to case. It depends on internal structure and interface design of original libraries.

1.1 Porting

There are to ways how to achieve the needed port:

1. Port source code,
2. Port an interface.

Project supported by the Ministry of Education of The Czech Republic – Project MSM 235200005 and Microsoft Research Ltd.(U.K.)

1.1.1 Porting Source Code

Porting source code means to rewrite original library from non-managed code to the managed one. Unfortunately this means that there must be changes done at source code level – changes which are not trivial.

Not only syntax is needed to change but also it is needed to change structure of an implementation of the algorithm due to restrictions of managed environment in many cases. This task is not easy to automate.

Also these libraries are usually huge – so it would take too long to rewrite the code and it could happen that the next version of library would be released before the old one is ported. This approach is possible only for libraries whose source code is available to a developer.

Next problem of this approach is the fact that some libraries are heavily system-dependent or low-level (for example OpenGL). Therefore porting of such libraries on the source code level does not mean that these libraries would be possible to use on another platform (not only different CPU but also different operating system).

1.1.2 Porting of an Interface

Opposite of that is the second approach of porting of non-managed library to the managed environment. This approach does not need source codes of original library – it uses binary form of the original library. It is based on a fact that the only important thing for a developer/user is an interface of such library. The knowledge of inside mechanisms is not needed for a user using that library. User just needs to know how it behaves from the view of the outside world – i.e. to know interface.

Thanks to that the only thing that needs to be ported is interface. This approach is more flexible in comparison with the first one. It has one major advantage – it is possible to automate the task. It means that it is possible to create ports of new version of the library fast enough so the situation, where new version is released before the old one is port, should not occur.

Also low-level and heavily system-dependent libraries can be ported with this approach. Unfortunately this approach to port libraries has one major disadvantage – all ported libraries are system-dependent not only on CPU level but also on operating system level (like OpenGL or VTK).

It means that port of such library is able to run only on specific CPU/operating system platform. This disadvantage is quite problematical because every new platform (operating system/CPU) on which managed environment is running needs new port of the library. It is somehow against the idea of managed environment, where the compiled application created on one platform can run without any changes on another one, which supports managed environment.

Unfortunately not all libraries are easily portable and for those libraries is this approach the only one possible (as it was explained before). Nice example of such library is OpenGL.

Another example is VTK library, which is possible to port as it was described using the first approach, but source code of this library is quite huge and therefore there would be high probability that disadvantages described earlier would occur.

Opposite of OpenGL the VTK library is object oriented. It is not based on COM technology (like DirectX) and its porting is not as easy as it would be in the case of COM technology (as described in MSDN).

1.2 Introduction to Ported Libraries

1.2.1 OpenGL

OpenGL is a library used for low-level rendering of 3D graphics. Rendering 2D graphics is also possible but the aim of OpenGL library is displaying of 3D space. This library was created by SGI and it is based on commercial SGI's graphic library.

Library provides a user to render various graphics primitives like lines, points and polygons. It is also possible to render parametric surfaces (like NURBS) by OpenGL thanks to additional library called GLU. Lighting and transformations computation of the rendered primitives is also provided by OpenGL library. User can setup various light types (directions, point and spot), their color, location in 3D space and other specific properties of light. Also tool for combining of basic transformations (translation, rotation and scaling) of rendered primitives is provided by OpenGL library.

It is possible to apply a texture and material to rendered primitives. OpenGL provides tools to handle this task. User can set a texture itself and its properties. For example user can set whether the texture is repeated on the surface or not, how it is combined with rendered primitives and other textures (this is called multitexturing and it is for example used to add pre-rendered shadows to rendered surface), etc.

These are basics of the library. Other functions are provided according to output device capabilities.

Interface of OpenGL library is standardized by specifications available to wide public. It consists of set of static functions (including constants) used to render primitives and to setup rendering pipeline. Every new version of the specification guarantees backward compatibility with previous versions.

OpenGL inner structure is based on state machine. User can change states with functions provided by the interface of the OpenGL library. Implementation details depend on operating system/graphics hardware vendor.

This usually results in debugging difficulties. All implementations behave as described in the specification, however, their behavior can differ in a case of an error while calling interface function (incorrect parameters, forbidden combination of function calls, etc.). The result is that an application can have different outputs depending on used hardware.

It is possible to add new functionalities to the OpenGL without need of updating whole specification. These additions are called GL Extensions. With them hardware vendors provides support for the latest features. This means, that they depend on plugged device capabilities and drivers.

OpenGL library is worldwide known as a standard for graphics output. It is supported on various platforms and provides basic facilities to create graphical engines.

1.2.2 VTK

VTK is object-oriented library for data visualization that contains many useful algorithms, importers, exporters and renderers. The main idea of VTK is visualization pipeline where output of one module is connected to the input of another one. This scheme implicates existence of two main object categories (process objects and data objects). The process objects are further divided into data source objects (outputs only), filters (inputs and outputs) and slinks (inputs only). Except these main objects there are some help objects (like matrix). In general VTK 4.0 contains approximately 700 classes with approximately 16000 public and protected class members.

For better view there is a simple example. Following code generates sphere colored by elevation. We can see the visualization pipeline diagram on (Figure 1). The *vtkSphereSource* (source) object generates *vtkPolyData* (data object). This data object passes through *vtkElevationFilter* (filter) that colors it to the *vtkDataSetMaper* (slink).

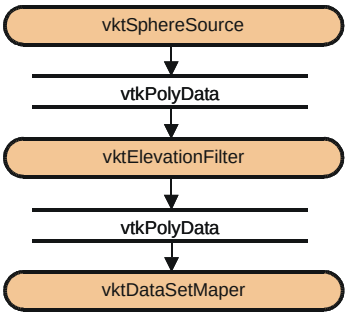

Figure 1. Example of visualization pipeline

Part of C# source code that implements visualization pipeline Figure 1 follows. Part of source code that creates renderer and rendering window was omitted.

```
vtkSphereSource sphere
       = vtkSphereSource.New();
sphere.SetThetaResolution(12);
sphere.SetPhiResolution(12);
vtkElevationFilter elevationFilter =
       vtkElevationFilter.New();
elevationFilter.SetInput(
       sphere.GetOutput());
elevationFilter.SetLowPoint(
       0.0f, 0.0f, -1.0f);
elevationFilter.SetHighPoint(
       0.0f, 0.0f, 1.0f);
vtkDataSetMapper mapper =
       vtkDataSetMapper.New();
mapper.SetInput(
       elevationFilter.GetOutput());
```

Graphical result of this example is shown on Figure 2

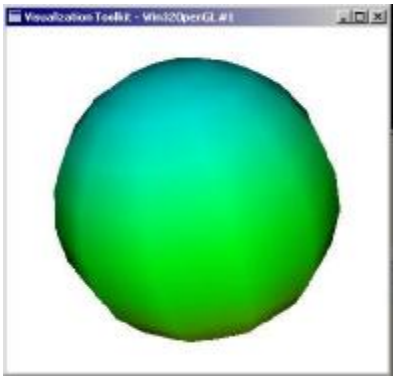

Figure 2. The resulting sphere

In general VTK is compact software with many useful rules that make user's life easier but there are some exceptions that make life sometimes harder.

1.3 The Goal

The goal of this project is to try to create not an implementation but an interface to existing OpenGL and VTK library in managed environment. This implementation should be able to maintain the highest compatibility with an original interface in non-managed environment. Also the important thing was to decrease delay of communication between non-managed and managed environment to the lowest possible value and to increase safety of programming.

2 EXISTING SOLUTIONS

2.1 CsGL

While there is probably no working port of VTK library to the managed environment, for OpenGL it already exists one. It is called CsGL and it is based on an approach described as a second one. It means that it is port of the interface of library in binary form.

This port is open source. It was implemented using plain C# with use of PInvoke mechanism and plain C language. As it is mentioned in CsGL documentation, this fact can simplify porting of the library to another platform. While porting, there is a need to modify only the C code. This C code is heavily system dependent and it is used to initialize OpenGL Render Context.

The use of PInvoke mechanism to port functions (their interfaces) is one of the main reasons why porting can be easily automated. This means that creating of interface implementation of new OpenGL Extension is just matter of running prepared script or application that deals with it. Together with a fact that creation of such application/script is not difficult this leads to quite important advantage.

Next advantage of CsGL is that porting of existing OpenGL application is quite simple with an exception of framework of an application, which needs to be modified due to restrictions of managed environment. Similar situation is by OpenGL initialization provided by CsGL library.

It is possible because of inheritance. All OpenGL/GLU functions and constants are static members of one class (from a view of CsGL user). Because of that the user needs to inherit his/her class in order to use OpenGL from CsGL class and modify source code (C/C++ source code) to compile it with C# compiler (or another compiler for managed environment). After that the code is able to run in managed environment using OpenGL.

This possibility is quite useful because of the fact that the result (OpenGL source code) looks quite the similar compared to original C code. Unfortunately this approach also means that user has no chance to use functions of specific OpenGL/GLU interface specification.

Safety of the programming is quite important task and it is not handled by CsGL. It means there is no control mechanism able to check validity of function parameters. For example it is possible to pass smaller array of values than required by a function. It is similar situation as in original C code and it can lead to bugs, which are not easy to find.

Question is whether this means a disadvantage or not. Because of the lack of function parameters checking, an application (or OpenGL code) runs faster than with parameters checking. This is an advantage for highly experienced user because speed is quite important in graphical applications.

Opposite of that, for less experienced user who is working on larger project, the safety of programming is more important than the speed. When passing invalid function parameters a run-time error will occur. The run-time error is not easy to invoke again and therefore debugging of such application is not an easy task. It costs a lot of time to locate and repair it.

CsGL interface is full functional. It contains OpenGL version 1.1, 1.2, 1.3 and 1.4, GLU and at least 50 OpenGL Extensions. It also includes tool to port new extension (user just need C language header files). From this view CsGL is ready to use.

2.2 VTK Wrappers

VTK already supports number of programming languages. Native language of VTK is C++ so it is apparent that the most efficient use is again in C++. Wrap-classes in different languages usually have only subset of functionality and higher time and memory consumption. These problems are unavoidable in our solution too.

3 ANALYSIS

3.1 OpenGL Approach

Approach that was chosen for an OpenGL port to managed environment was the second one which was described before – i.e. porting of an interface.

OpenGL is low-level library because of being too close to operating system and device drivers. This means that it is impossible to port this library by rewriting source code and it also means that this library is heavily system-dependent.

This implicates that the only way how to implement OpenGL in managed environment is to implement an interface of this library. It is approach used by CsGL version of OpenGL port. In order to use this approach a creation of wrappers for all functions is needed. Inside of wrapper's body there is code for handling data sharing between managed and non-managed environment.

Fortunately OpenGL interface consist of group of functions and constants which are not members of any class. Because of this and the fact that OpenGL is available as C-style header files it is possible to automate this task.

Usually this wrapper's body consists of call of original OpenGL function but in some cases there is need of special handling of passed data due to differences between managed and non-managed environment.

Also data validity checking can be placed inside of such wrapper's body. This checking should prevent user from passing arrays of invalid size. For example common error is passing of shorter array then it is needed by OpenGL to overwrite it with data. That is caused by a fact that the only thing that is passed to the system is a pointer without any information about the length of this array.

According to that system can rewrite some parts of memory situated behind such array. This could lead to application's crash. This crash depends on current memory content. This bug can sometimes cause application's crash and such mistake is not easy to find during debugging.

3.1.1 Difficulties

The main problem that needs to be solved is Garbage Collector (alias GC). It is part of managed environment which is quite useful. It handles memory deallocation and fragmentation. It means that user of managed environment does not need to be aware of memory management. Object created by a user is simply removed from the memory when there is no active reference to such object.

This advantage could be a disadvantage while sharing data (in the form of an array) with non-managed environment. In the case of OpenGL this situation can occur when the user passes data (in a form of an array) to an interface. These data are then passed through and stored inside of OpenGL library in form of pointer in non-managed environment and can be used somewhere in a future.

Consider this situation happens in a function where the array was allocated and the only reference, which exists, is local variable. Due to the fact that pointer to data is stored inside OpenGL library in form of plain pointer, GC does not know that there still exists a reference to an array.

After exit from the function and release of local variables such object is next candidate for deallocation because there is no reference pointing at it. This memory cleaning does not occur immediately after an object looses all of its references but later (depending on construction of CG). This situation occurs while using functions like *glVertexPointer*, etc.

As it was mentioned, GC also handles memory fragmentation. This means GC can move block of memory to prevent memory fragmentation. Unfortunately it can happen anytime. Because of that, every function, which works with an array, needs special data handling inside of wrapper's body. This handling should prevent GC from moving or deallocation of passed data.

CsGL implementation solves this problem by use of PInvoke mechanism and *IntPtr* data type. It usually means that user is responsible for preventing GC from doing its duty on desired objects.

This approach leads to higher speed because of the fact that the user (developer) is the only one, who can decide when and how long should be data protected against GC. Unfortunately the user has to have the knowledge about non-managed and managed environment interaction. Implementation described here tries to handle this interaction on its own so the user does not need to have such knowledge.

Not only arrays are problematic. Also callback functions are source of difficulties. PInvoke mechanism offers a possibility to pass delegate that is used as a callback function. This solution is used by described implementation and by CsGL.

The problem is not callback itself but its parameters. As long as these parameters are simple data types the PInvoke mechanism (together with data marshaling) handles this task. A problem can occur in case of passing user object as a parameter of the callback (for example user data). This can be solved either using *IntPtr* data type or handling it inside of wrapper's body.

Using *IntPtr* solves the problem completely. This data type can be used to store an index to user defined array or a key to hash table. This solution does not need any support from interface because it is handled similar to any other simple value data type. This approach is used by CsGL library – it is easy to implement and not difficult to use because it is quite similar to the void pointer (non-managed environment).

With no use of *IntPtr* interface has to handle user defined data itself and provide data structures for storing such data. This approach simplifies use of such callbacks (from a user's point of view) because user does not need to take care about configuring his/her own data to pass them to callback. User can pass an object (user data) to the callback setup function and get it in form of callback's parameter.

It hides parts of code that is created by user in the case of *IntPtr*. Due to that it leads to larger wrapper's source code, additional data structures and increased memory requirements. It is also solution used by interface described here.

Last major difficulty of an interface implementation is void pointer. That is a very useful language construction. A function, which uses void pointer as a parameter, can easily get various data structures via void pointer without need of explicit data conversion. The rest of the parameters provide description of such data. In managed environment a data type *IntPtr*

has a quite similar meaning (from user's view) as void pointer in non-managed one.

It is easy to implement and it is not difficult to use for an experienced user. Probably this is why CsGL uses it. Unfortunately this can lead to pass invalid data to a function via this data type. An inexperienced user can pass array of objects instead of array of value data types quite easily.

This implementation avoids use of *IntPtr* as an attempt to increase programming safety. It is sure this approach is more complicated than the previous one because of need to specify data type and it can also lead to slowdown.

3.2 VTK

We will discuss our solution of VTK interface for .NET in this section. In the first subsection we will show wrap-class principle on simple example. In the second subsection problems with inheritance and polymorphism will be discussed. Possible solution of it could be done by two-level wrapping. This solution is currently under investigation.

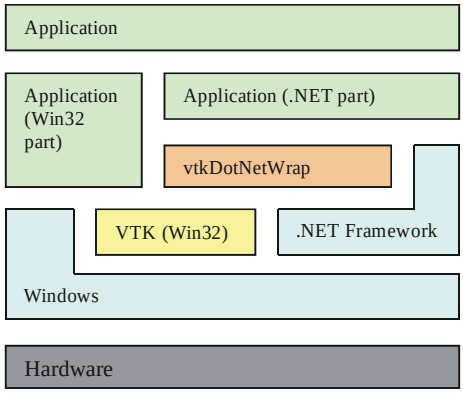

Figure 3. Possible scheme of VTK application in .NET environment.

The possible placement of managed dynamic-linked library with wrappers is shown on Figure 3. It should be between .NET applications and unmanaged dynamically linked libraries of VTK.

3.2.1 C++ Managed Extension Wrap-class

Each wrap-class is written in C++ Managed Extension and provides access to one unmanaged VTK class. The wrap-class is garbage-collected (managed) and so accessible from .NET environment. An instance of wrap-class contains one instance of unmanaged VTK class. The wrap-class has the same public methods as wrapped-class. Calling of wrap-class method usually causes only calling of appropriate wrapped-class method and some data conversions. There are some exceptions especially in constructors. Following source code is a part of wrap-class source code.

```
public __gc class vtkAbstractMapper : public
vtkProcessObject       // wrap-class
{
public:
  ::vtkAbstractMapper *w;  // wrapped-class
  vtkAbstractMapper(
       ::vtkAbstractMapper *_w):
       vtkProcessObject(_w) { w = _w;}
  // const char *GetClassName ();  1303 ()
  System::String * GetClassName()
  {
    return new System::String(
       w->GetClassName());
  }
```

Note the data conversion in *GetClassName* method. There is creation of *String* object from zero-terminated string returned from unmanaged method. This is typical example.

All these wrap-classes are compiled into one managed dynamic linked library file that uses unmanaged libraries of VTK.

3.2.2 Difficulties

There are some difficulties in this way of porting. In the following subsection there is described the most interesting problem with inheritance. In the second subsection there are outlined some minor problems together with explanation why only subset of methods can be successfully wrapped.

3.2.2.1 Inheritance and Polymorphism

The main disadvantage of this approach is probably the inability of effective usage of inheritance and polymorphism. It is caused due to composition was used instead of inheritance between wrapped-class and wrap-class. For user it means there is no straightforward way to create its own VTK module in C#.

For example, consider we want to create our own source of *vtkPolyData* (for example Sierpinsky fractal). Standard approach in *C++* is to derive our own class (*vtkSierpinSource*) from *vtkPolyDataSource* and override the virtual method *Execute* (and other methods). After that we have fully functional source of polygonal data. We can connect it into any visualization pipeline that accepts polygonal data object. With C# it is not possible due to inability to override any virtual method and to access protected methods/attributes.

Fortunately there is a way to go around it. We call it double wrapping. On the following example there is shown the main idea of this approach. Let us have unmanaged Win32 class (called *Win32Class*) in DLL that we cannot modify. This class contains two methods which implementations are not important.

```
class EXPORT Win32Class
{
public:
  void PrintSelf();
protected:
  virtual char *Info();
};
```

The level-one wrapper is also unmanaged and provides access to protected methods and is responsible for correct calling of virtual method of level-two wrapper. Note that the direct inheritance is used.

```
class L1Wrapper : public Win32Class
{
public:
  void * l2w; //packed GCHandle
                    of level-two wrapper
  char * Info();
};
```

Things are going to be more complicated. Following method unpacks the GCHandle from void pointer and calls virtual method of level-two wrapper.

```
char * L1Wrapper::Info()
{
  IntPtr intPtr
        = IntPtr::op_Explicit(this->l2w);
  GCHandle handle
        = GCHandle::op_Explicit(intPtr);
  L2Wrapper::L2Wrapper * managed
= dynamic_cast <L2Wrapper::L2Wrapper *>
        (handle.Target);
  char * ret
        = String2Chars(managed->Info());
  return ret;
}
```

Level-two wrapper is very similar to the single wrap-class presented above. It is managed and makes level-one wrapper easily accessible from .NET environment.

```
__gc public class L2Wrapper
{
  public:
    ::L1Wrapper *w;

    L2Wrapper()
    {
      w = new ::L1Wrapper;
      GCHandle handle =
GCHandle::Alloc(this, GCHandleType::Weak);

      IntPtr intPtr = (IntPtr) handle;
      w->l2w = (void *) intPtr;
    }

    ~L2WBase()
    {
      handle.Free();
      delete w;
    }

    void PrintSelf()
    {
      w->PrintSelf();
    }

    virtual System::String * Info()
    {
      return new
         System::String("L2Wrapper::Info()");
    }
}
```

This approach looks like a reasonable way to make a possible to create fully functional VTK modules in C#. Nowadays it is in current research interest. Simple examples based on this works well.

3.2.2.2 Other Minor Difficulties

Other difficulties with porting are some inconsistencies in VTK library. For example in some cases there is a difference in class instancing. Many classes can be only in heap memory (because constructors are protected) so they are accessible only by pointer but there are some classes that can be anywhere. Different approach of wrapping has to be used. Also sometimes C structure FILE is used in the place of C++ stream, etc. Also callback functions are tough and some system dependent variables like window handles are not already fully implemented.

4 IMPLEMENTATION

4.1 OpenGL

Described solution is implemented in Managed Extension C++ because of the possibility to mix managed and unmanaged code and to use unmanaged APIs directly (as described in MSDN). Also C style preprocessor macros help to simplify implementation of parameters-checking.

The goal of our implementation is to create an interface which is as close as possible to the original OpenGL API (described in specification) while being easy to use. Also possibility to choose specific version of OpenGL is one of the important parts of the goal.

Designed and implemented interface consists of three main classes: class representing OpenGL's Rendering Context, class used for storing internal data and class containing interface functions and constants.

Opposite of CsGL, functions are not static members of the class – i.e. an instance of such class have to be created in order to use OpenGL/GLU functions. It is possible to use this class in two slightly different ways.

First one is similar to the approach recommended in CsGL library. User has to inherit his own class from a class containing OpenGL/GLU functions (constants). However it is not possible to use both libraries OpenGL and GLU with this approach because they are placed in separate classes.

Second approach is suitable for this interface. User has to create an instance of desired classes and than call their member functions. Unfortunately source code, which is the result of use of this approach, can look ugly compared to CsGL source code.

To solve this disadvantage, a second set of functions and constants was included into each class. This set differs only in the name of functions (constants). In the case of functions the "*gl*" prefix is removed from a function name. In the case of constants the situation is similar – the "*GL_*" prefix is removed form a constant's name.

This together with intelligent names for class instances results into clear source code quite similar to original C (or CsGL) code.

```
glClearColor(0, 0, 0, 0);
glClear(GL_COLOR_BUFFER_BIT);
glBegin(GL_POINTS);
glVertex2d(0, 0);
glEnd();
glFlush();
```

```
gl.glClearColor(0, 0, 0, 0);
gl.glClear(GL.GL_COLOR_BUFFER_BIT);
gl.glBegin(GL.GL_POINTS);
gl.glVertex2d(0, 0);
gl.glEnd();
gl.glFlush();
```

```
gl.ClearColor(0, 0, 0, 0);
gl.Clear(GL.COLOR_BUFFER_BIT);
gl.Begin(GL.POINTS);
gl.Vertex2d(0, 0);
gl.End();
gl.Flush();
```

Figure 4. Example of use of C/CsGL code (on the top), described interface (middle and bottom).

Example in Figure 4 displays use of C/CsGL code (on the top) and described interface. Identifier "*gl*" is name of instance of "*GL*" class, which contains OpenGL functions and constants. The code on the bottom uses function set with removed prefixes. You can see that the code on the bottom is quite similar to the code on the top.

As it was mentioned above, OpenGL and GLU functions (constants) are placed in separate classes. To achieve a possibility for the user to choose specific version of OpenGL/GLU, every version is placed in separate class. Such class inherits from a class of previous version. For OpenGL this means that there exists (or will be created as described in chapter Future work) a class for version 1.1 (*GL11*), version 1.2 (*GL12*), etc. Class *GL12* inherits from class *GL11*.

To simplify use of these classes a special class is added. This class is inherited from highest version of OpenGL specification which is implemented. Therefore this class provides simple use of constants and the user does not need to know which version of OpenGL interface contains desired constant.

To add new OpenGL versions it is required to do some modifications of specific parts of the interface's implementation.

- The parent of *GL* class needs to be updated to reflect the latest implemented version.
- The class, which contains internal data structures, needs to be modified by adding (not changing) members (if needed).

All of these modifications are not major and they are not visible for user. Therefore older applications can run without any modifications. GLU interface implementation is handled in similar way, same as OpenGL.

4.2 VTK

Because of VTK size there is necessary to automate the processes of wrap-class generating. This process consists of two parts. At the first part it is necessary to obtain information about each VTK class we want to wrap. The second part is generator itself that use the output from parser. For data exchange between the parser and the generator intermediate files are used. It is a simple text file that contains appropriate information about class, methods and method parameters. Schematic view on the whole process is shown in Figure 5.

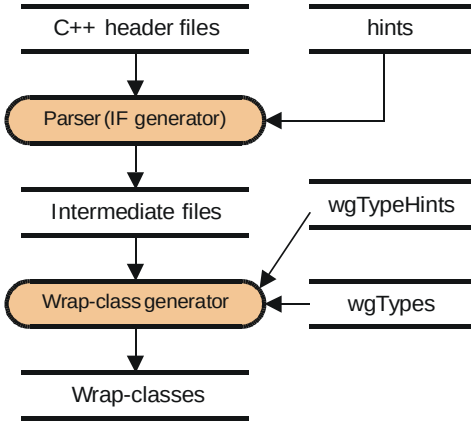

Figure 5. Scheme of automated wrap-class generating process

4.2.1 Parsing of C++ Headers

To obtain information about VTK classes parsing of C++ headers was chosen. This is the easiest way because parser for it already exists. This parser has been already used for Java, TCL and Python interfaces. The parser is written by means of Yacc and Lex. There was no problem to add simple exporter from inner structures of the parser to some text file. This text file is called intermediate file and an example of it follows. Note this is part of the same class as in example in sec. 3.2.1.

```
ClassName vtkAbstractMapper
HasDelete No
IsAbstract No
IsConcrete Yes
NumberOfSuperClasses 1
SuperClass vtkProcessObject
NumberOfFunctions 18
  FunctionName GetClassName
  FunctionSignature const char *GetClassName
       ();
  NumberOfArguments 0
  ArrayFailure No
  IsPureVirtual No
  IsPublic Yes
  IsOperator No
  HaveHint No
  HintSize 0
  ReturnType 1303
  ReturnClass None
  FunctionName IsA
  FunctionSignature int IsA (const char
       *name);
```

```
  NumberOfArguments 1
    ArgType 1303
    ArgCounts 0
    ArgClasses None
  ArrayFailure No
  IsPureVirtual No
  IsPublic Yes
  IsOperator No
  HaveHint No
  HintSize 0
  ReturnType 4
  ReturnClass None
```

There are some difficulties in this approach. Probably the most serious is coding of data types. The parser uses coding of data types where some ambiguities are. This must be patched in generator with explicit retyping. The creation of the explicit retyping data file is time consuming.

4.2.2 Wrap-class Generator

The generator itself is one program written in C# that converts intermediate files into wrap-classes source codes. It has three types of input files. The first one is intermediate file with class description (presented above). The second one is file ***wgTypeHints*** where some data type codes are explicitly overridden. The third one is file ***wgTypes*** where data type conversions are defined by means of macros. An example of this is given in Table 1.

Code	303
ACppDecl	char * <var>
MCppDecl	System::String * <var>
ToACpp	wgStr2Char(<var>)
ToMCpp	new System::String(<var>)
TmpDecl	
TmpRet	
IsProblematic	No

Table 1 Example of macros for System.String conversion.

Usage example of conversion macros follows in pseudo-code. It presents how and where these macros are expanded during wrap-class generation.

```
MCppDecl<methodName(MCppDecl<arg0>,
MCppDecl<arg1>)>
{
  TmpDecl<arg0, tmp0>;
  MCppDecl<ret>;
  ret = ToMCpp<w->methodName(
        ToACpp<tmp0, arg0>, ToACpp<arg1>)>;
  TmpRet<arg0, tmp0>;
  return ret;
}
```

In these subsections the process of wrap-class generation has been briefly presented . This process is quite complicated and is not designed for user to do it and we presented it here only for better view how we deal with the size of VTK.

5 FUTURE WORK

5.1 OpenGL

As it was mentioned above future work will be aimed to increase the programming safety, to add parameter validity checking and enumerate types (to replace constants). Also implementation of newer OpenGL/GLU versions and OpenGL Extensions are goals of the future work.

Future work will concern an attempt to create tests to measure the slowdown of specific parts of interface in comparison to CsGL implementation. The C code will be used as reference. This should prove whenever our interface implementation is suitable for use in practise or not.

5.2 VTK

The main goal for the future work is to implement the double wrapping to make possible inheritance and polymorphism and so allow writing fully compatible algorithms as VTK objects in C#. Also some data types not currently implemented should be added.

As a vision we are considering the possibility of graphical programming with VTK modules similarly to MVE 1.0 developed by our workgroup. See http://herakles.zcu.cz.

6 CONCLUSION

Described OpenGL interface was not meant to replace existing CsGL implementation. It is an attempt to create an interface where user can fully enjoy the comfort and safety of managed environment while experiencing the lowest slowdown as possible.

Currently this interface is in status of beta version and so it is not yet recommended for serious use. Interface of OpenGL 1.1 and GLU 1.1 is fully implemented without mentioned parameter checking.

Presented solution of VTK interface looks reasonable and follows the same strategy as OpenGL interface according to comfort and safety of managed environment. In fact it suffers from the same problems as interfaces of VTK for other languages (not full set of accessible methods).

7 REFERENCES

[1] SGI: OpenGL 1.3 specification
[2] MSDN (electronic resource)
[3] CsGL project documentation
[4] Schreder, W., Martin, K., Lorensen, B.: The Visualiasation Toolkit, Prentice Hall, New Jersey, 1998
[5] Kačmář, D.: Programujeme .NET aplikace (in Czech), Computer Press, Praha, 2001