University of West Bohemia

Faculty of Applied Sciences

Department of Computer Science and Engineering

Diploma Thesis

Visualization Toolkit for C# in Scope of ROTOR Project (VTK for C#)

Pilsen 2003 Milan Frank

Abstract

A new environment called .NET was recently introduced to a wide public. However, this environment does not contain libraries for advanced graphical output. Therefore it is necessary to make such libraries available to .NET. Here presented work describes implementation of Visualization Toolkit (VTK). The implementation allows straightforward and safety cooperation of VTK with .NET. Short description of VTK is included as well as a short introduction to .NET environment.

Table of Content

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

Pilsen, 23th May.2003 Milan FRANK,

Part 1 Introduction

This part gives the general overview about presented work and the structure of the whole text. Expected knowledge of the reader of this text is especially object-oriented programming principles, general knowledge of *C++* and possibly Java and Java Virtual Machine.

1.1 Structure of the Text

There are four main parts in the text. The Part 2 is aimed generally on the VTK library. There is a theoretical overview about the VTK structure, its object-oriented design and the most important principle - *visualization pipeline*. Some illustrating examples are given for better understanding. Also discuss about the VTK implementation is given with aim on the wrapping problematic.

The Part 3 discusses the *C#* and .NET environment technologies. It is mostly general description of the .NET Framework (or Common Language Infrastructure - CLI). It provides necessary background information for understanding of the presented work. The aim is on explanation of the .NET specific terms.

Finally, the Part 4 presents the VTK for .NET interface itself. It starts with the particular solution of the wrapping of VTK classes by the *MC++* wrap-classes. It is the main point of this work. The description of the data conversion follows. Next important section in this part describes the process that automates the *MC++* wrap-class creation. Finally discuss about results is presented.

1.2 Motivation

A new environment called .NET was recently introduced to wide public. However, this environment does not contain libraries for advanced graphical output. But the developing of an advanced computer graphic applications requires adequate tools. The VTK seems pretty reasonable for rapid developing of complex software and can be successfully used at the scientific field of application as well as in the commercial sphere. Also the new environment called .NET seems powerful with its programming safety and integration of wide sort of libraries in unified fashion. The goal of this work is to integrate the comfort and safety of .NET environment with power of the VTK.

1.3 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.

I wish to thank to my colleague Ivo Hanák for invaluable advices with .NET environment. Many thanks also belong to all students of KIV/GSVD course 2003 for testing large number of modules and their leaders ing. Martin Čermák and ing. Tomáš Hlavatý for the testing tasks preparation.

Part 2 Introduction to VTK

This part describes VTK (Visualization Toolkit) developed by Kitware Inc. as a free-source, object-oriented software. See [Kitware] web site. It is large and complex tool for data visualization. In version 4.2, it contains more than 800 classes for various purposes with more than 20000 public and protected class members. The concept of VTK makes the library easy to use once programmer learns about its object-oriented design and principles.

The VTK is an open-source software. That provides an advantage in large number of developers that can easily participate on the bug fixes and/or developing of new useful modules, tools and features.

The current version of the VTK can be installed on MS Windows and almost all UNIX-based systems. Therefore there is a good portability on source code level. Once developer writes a program that uses VTK only then he has good probability of running the program under Windows and UNIX operating system with recompilation only.

The Visualization Toolkit is aimed on scientific data visualization and image processing. It contains wide variety of Filters (algorithms), Importers, Exporters, Renderers and also set of classes for data representation. The main idea is visualization pipeline that uses the data flow principle. Therefore the VTK is relatively high-level library and so specialized for particular purpose. For example, hard to imagine is to make fast and flexible 3D game engine (as in the case of e.g. Quake with OpenGL) in VTK. On the other hand, the full power of VTK can be unleashed in the case of need e.g. volumetric data visualization with some filtering, isosurface extraction and triangle reduction. All necessary algorithms for this task are probably already available as VTK classes, renderer included. Therefore only necessary programming is to describe the visualization pipeline by instancing and interconnecting of some special objects and make some graphical user interface.

2.1 VTK Classes

VTK classes can be divided into following groups.

- Graphical pipeline objects
- Visualization pipeline objects
- Data objects
- Process objects
- Helper objects

The graphical pipeline objects are elements of the scene (geometry, textures, lights, cameras, etc.) and all other objects required for rendering (renderer, render window, interactor, etc.). Visualization pipeline objects belong to the data flow graph and its main task is to transform input data to some representation that can be rendered. Helper objects are any other objects as e.g. Matrix.

2.2 Graphical Pipeline

The pipeline is divided to graphical and visualization part probably because the graphical pipeline does not contain process and data objects strictly divided (in contrast to visualization pipeline). Actually, each object of graphical pipeline represents some data and process together. Another difference is that the graphical pipeline objects contain subclasses that are hardware dependent (means OpenGL-dependent, Mesa-dependent, etc.).

The graphical pipeline is responsible for rendering of data already prepared by visualization pipeline. Following list contains the most common classes that create a scene and renderer.

- vtkActor, vtkActor2D, vtkVolume
- vtkLight
- vtkCamera
- vtkProperty
- vtkMapper
- vtkTransform
- vtkRenderer
- vtkRenderWindow
- vtkRenderWindowInteractor

The *vtkProp* class represents things that we can "see" in the scene. Its commonly used subclasses are *vtkActor*, *vtkActor2D* and *vtkVolume*. Inheritance graph of the *vtkProp* class is given in Figure 2.1. The *vtkProp3D* represents objects that can be manipulated in the scene (e.g. have a general 4x4 transformation matrix). The *vtkVolume* is an actor specialized for volume rendering together with *vtkVolumeMapper*. The *vtkImageActor* used for 2D rendering in 3D scene and the *vtkActor2D* are utilized for two-dimensional data rendering without transformation matrix. Note the LODActor that means Level-Of-Detail that can automatically switch among number of geometry representations to maintain the frame rate.

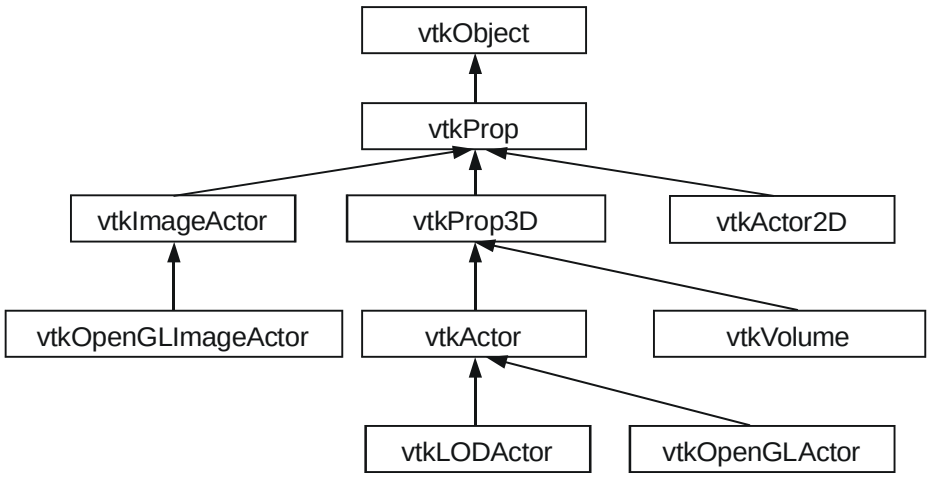

Figure 2.1 – The vtkProp inheritance graph

Actors with mappers have OpenGL subclasses that contain OpenGL code that is responsible for rendering of given data. When user creates an instance of *vtkActor*, a

hardware-dependent subclass is created (like *vtkOpenGLActor* instead of *vtkActor*). This approach is reasonable since the runtime can decide what particular graphical library will be used for rendering. Thus, the source code is hardware (graphical-interface) independent. The same principle is used for all graphical pipeline objects.

The mapper object provides interconnection between visualization and graphical pipeline. Together with actor, it is responsible for rendering of particular geometry/data.

Renderers and render windows are used to manage the interface between the graphics engine and the particular computer windowing system. The render window is a window that can be seen on display and where the renderer draws into. More than one renderer can be inserted into a single render window. The region that the renderer draws into is called *viewport*.

2.3 Visualization Pipeline

Good understanding of the visualization pipeline is probably the most important matter for efficient usage of the VTK. The visualization pipeline allows straightforward modular approach in application design.

The pipeline approach implicates two main class types and three subtypes.

- Data object
- Process object
  - Sources (outputs only, e.g. *vtkConeSource*)
  - Filters (inputs and outputs, e.g. *vtkDecimate*)
  - Mappers (inputs only, also Slinks or Terminals, e.g. *vtkPolyDataMapper*)

The data object represents some particular data structure, e.g. triangle mesh, volumetric data, image, etc. The process objects are workers that: produce, change, convert or consume the data objects. General example of visualization pipeline is given in Figure 2.2.

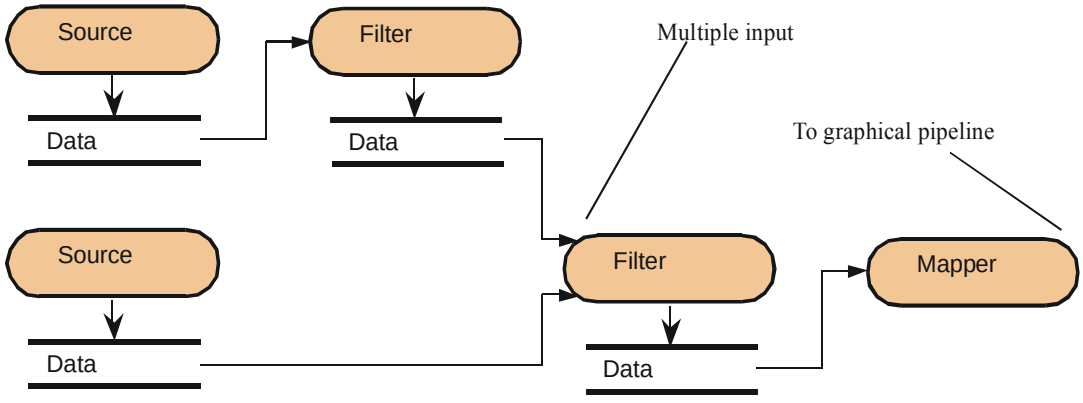

Figure 2.2 - Example of visualization pipeline with anonymous objects

2.3.1 Data Objects

A basic class for all data objects in VTK is *vtkDataObject*. This is a general representation of any data that can be exchanged among process objects. It serves to

encapsulate instance variables and methods for visualization network execution. Data that has a formal structure are called a *dataset*. Appropriate class is called *vtkDataSet* and is a subclass of *vtkDataObject*. A data object consists of geometric and topological information (points and cells), as well as associated attribute data such as scalars or vectors. The attribute data can be associated with the points and can represent various entities from scalar to *n*-dimensional tensor.

Following Figure 2.3 gives the common used data types as inheritance graph. This graph is incomplete and some objects are omitted.

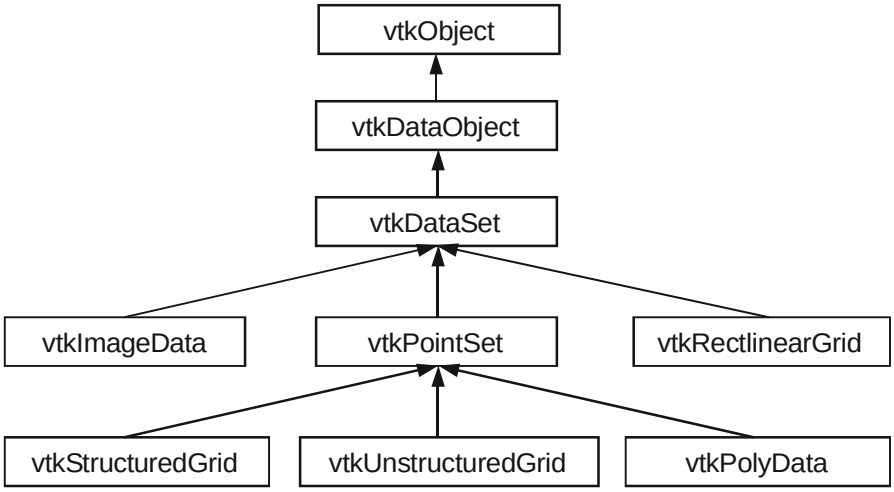

Figure 2.3 - Common data objects in inheritance graph. The super-class is in top

Probably the most useful data objects are in the bottom row. The most general is the *vtkUnstructuredGrid* object representing spatial and topological irregular set of points that can be topologically connected by *vtkCell* into lines, poly lines, polygons, tetrahedrons, etc. The *vtkPolyData* are similar but have some limitation in topological structure. Only vertices, lines, polygons and triangle strips can be topological elements of the *vtkPolyData*. Triangle mesh is typical example of *vtkPolyData*. The *vtkStructuredGrid* represents topological regular but spatial irregular data grid. Each point can be selected by [*i, j, k*] indices. Therefore neighbors are given implicitly but user of the object gives coordinates of each point explicitly. The *vtkRectilinerGrid* represents spatially and topologically regular data. This data type is useful for volumetric data in regular grid.

PolyData Example

This example shows the creation of polygonal data object and addition of one triangle into them. The *vtkPoints* object named *vertices* represents the spatial information. The *vtkTriangle* object named *triangle* represents the topological structure. It is a subclass of *vtkCell* object that is a general topological element. Following Code 2.1 gives example in *C#*.

```
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.Allocate(1, 1);
  polyData.SetPoints(vertices);
  polyData.InsertNextCell(triangle.GetCellType(), triangle.GetPointIds());
```

Code 2.1 – Example of polydata creation that contain one triangle

In Code 2.1, initially a field of points representing spatial information is created. Through that piece of code, three particular points are added. The meaning of *SetPoint()* method parameters is as follows: *index, x-coordinate, y-coordinate* and *z-coordinate*. As a next step, a topological structure is created. It might looks a little confusing, but the *GetPointIds()* returns a list of point indices and so the indices number can be set by *SetId()* method. The meaning of the parameters is following: an index in triangle indices field and corresponding index in vertices field. Finally the poly data are created and allocated for one cell. The geometrical information is set by *SetPoints()* method call. The topological information is setup by *InsertNextCell()*. Unfortunately the *vtkPolyData* doesn't contain direct method that has *vtkCell* parameter. Therefore *InsertNextCell()* must be called with presented parameters. It is probably due to limitation of the topological structure in *vtkPolyData* object.

It is need to point out that handling of polygonal data in this fashion is usually matter for developer of a new process object only. Because for most applications that use VTK, enough process objects exist and so no new ones are usually needed.

2.3.2 Process Objects

A process object operates on data objects to produce new data objects. They represent an algorithm. Process objects are connected to visualization pipeline (or data flow network). This principle implicates three types of process objects:

- Source
- Filter
- Mapper

The **source** is object that reads or generates some kind of data. This is the starting object of the visualization pipeline. The **filter** object is object that has some data object/objects as input and some object/objects as output. The input and/or output type can be the same or different. The **mapper** (or slink or terminal) object terminates the visualization pipeline. It usually represents the interconnection to graphical pipeline (or interface). Another function of mapper can be an export of given data to data file.

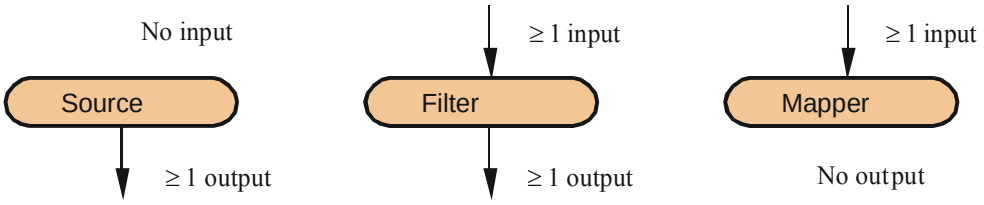

Figure 2.4 – Source, Filter and Mapper

Calling of *SetInput()/GetOutput()* method establishes an interconnection between process objects. The calling of *aProcObj.SetInput(bProcObj.GetOutput())* provides interconnection between *aProcObj* and *bProcObj*, with data flow direction from *b* to *a*. There is necessary to check the data type compatibility. The output has to be of the same type as expected input type is.

2.3.3 Pipeline Execution

The visualization pipeline is only executed when data is required for another processing (*lazy evaluation*). This execution is based on the internal modification time of each object via *Update()* method. See Figure 2.5.

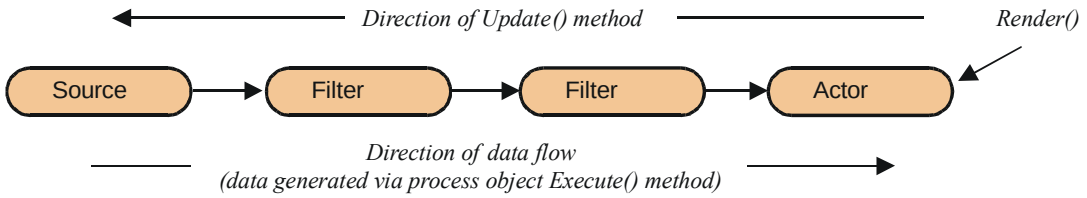

Figure 2.5 – Conceptual overview of pipeline execution

Each process object implements particular *Execute()* method that produces its output. The Update method calls the *Execute()* method only when last modification time of input objects is newer than modification time of this object or if any values of current object have changed. Therefore the *Exectute()* method call brings the current object up to date. This mechanism goes recursively through whole graphics and visualization pipeline.

Usually the method that executes the pipeline is *Render()* method. Then *Update()* method goes recursively up to the source objects that create data and propagate them through filters and graphical elements to the renderer. For instance, when just some rotation of the actor is encountered, there is no need to re-generate or filter data again. Therefore the *Update()* method checks whole pipeline and just the actor object is transformed (executed). Particular example follows.

The Mace Example

This example shows some previously introduced features of the VTK on mace example. Resulting object of Visualization pipeline is a polygonal representation of sphere with cone on each vertex with the vertex normal orientation; see Figure 2.6. This data are rendered in window with simple interactor that allows some basic manipulation by mouse, like rotation and translation of the object.

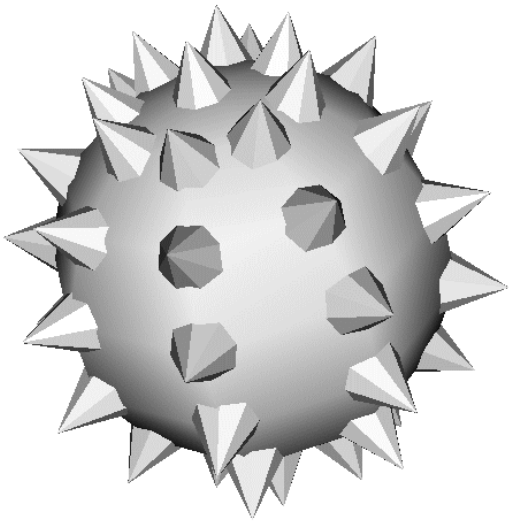

Figure 2.6 - The mace output

Figure 2.7 gives the data flow graph and the graphical interface. This graph is reasonable way of VTK application analysis.

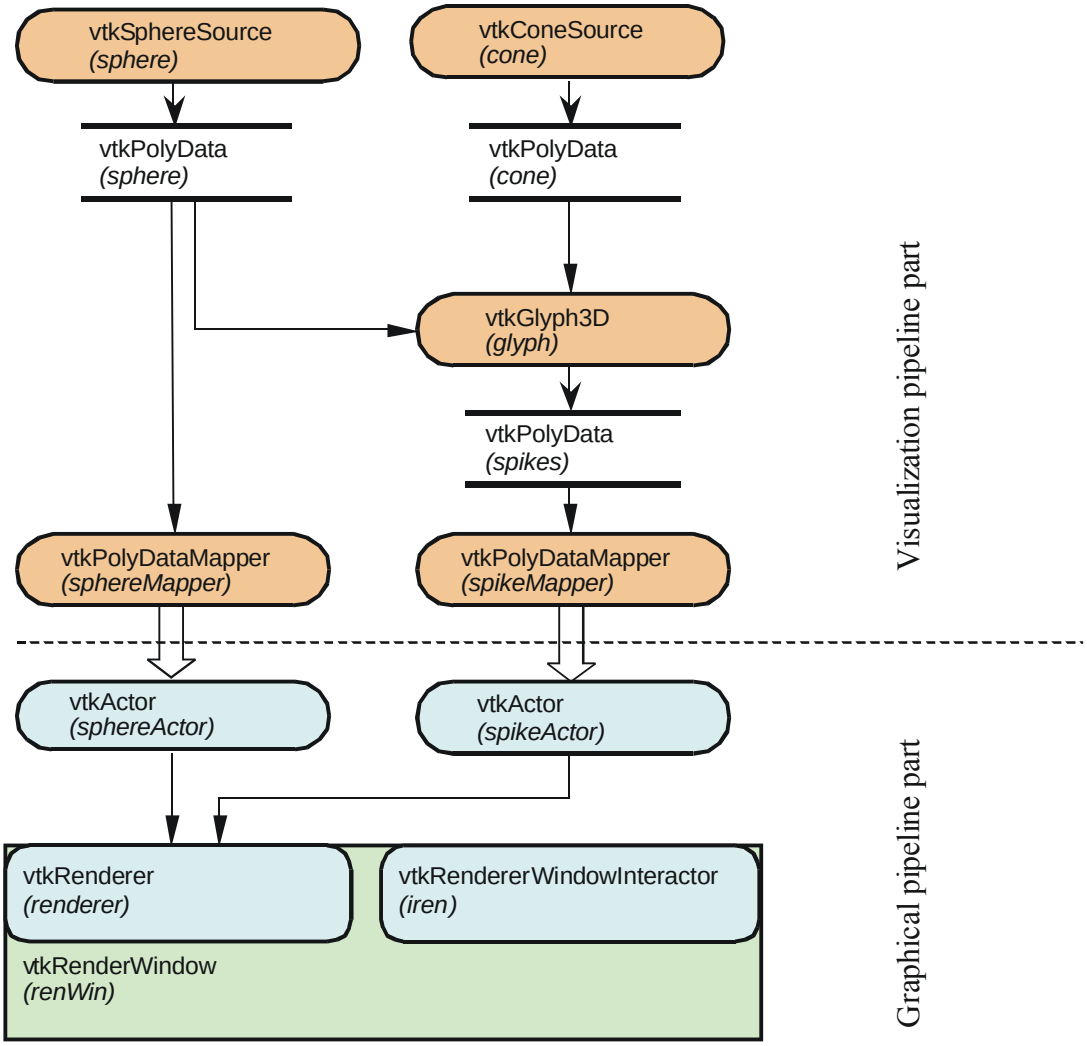

Figure 2.7 - The mace example as with visualization pipeline and graphical part

As usual, the visualization pipeline starts with source objects. In this case, there are two sources: *sphere* and *cone*. The *sphere* is an instance of the *vtkSphereSource* class and the *cone* is an instance of the *vtkConeSource*. They are simple sources of generated geometric data of *vtkPolyData* type. The *glyph* object is an instance of *vtkGlyph3D* class. It has two inputs. The first input is polygonal mesh and the second one is an object that is going to be mapped on each vertex. The output is a new polygonal object; here, it is called *spikes*. This object is example of filter with two inputs and one output. The *sphere* and *spikes* are mapped to *sphereMapper* and *spikeMapper*. The visualization pipeline ends in these two terminal objects.

The graphical pipeline starts with two actors that are the basic elements of the scene. These actors are linked with appropriate mapper objects. Actors are added to the renderer where they create a scene. The renderer is added to render window with interactor.

Used process objects are given in Figure 2.8 as a graph of inheritance. All process objects are derived from *vtkProcessObject*. Its subclass *vtkSource* is an abstract object that specifies behavior and interface of source objects. The source object is further specialized for polygonal data by its subclass *vtkPolyDataSource* that produce polygonal data only. *vtkConeSource* and *vtkSphereSource* are its typical direct subclasses that produce appropriate geometry.

The *vtkDataSetToPolyDataFilter* class is an abstract filter class whose subclasses take input of some dataset and generate polygonal data on output. It seems to be a little logical "jump" from source to filter. Our filter Glyph3D is a direct subclass of the *vtkDataSetToPolyDataFilter*.

A subclass for all mappers is *vtkAbstractMapper* that is derived from *vtkProcessObject*. There are two classes between the *vtkAbstractMapper* and our *vtkPolyDataMapper*.

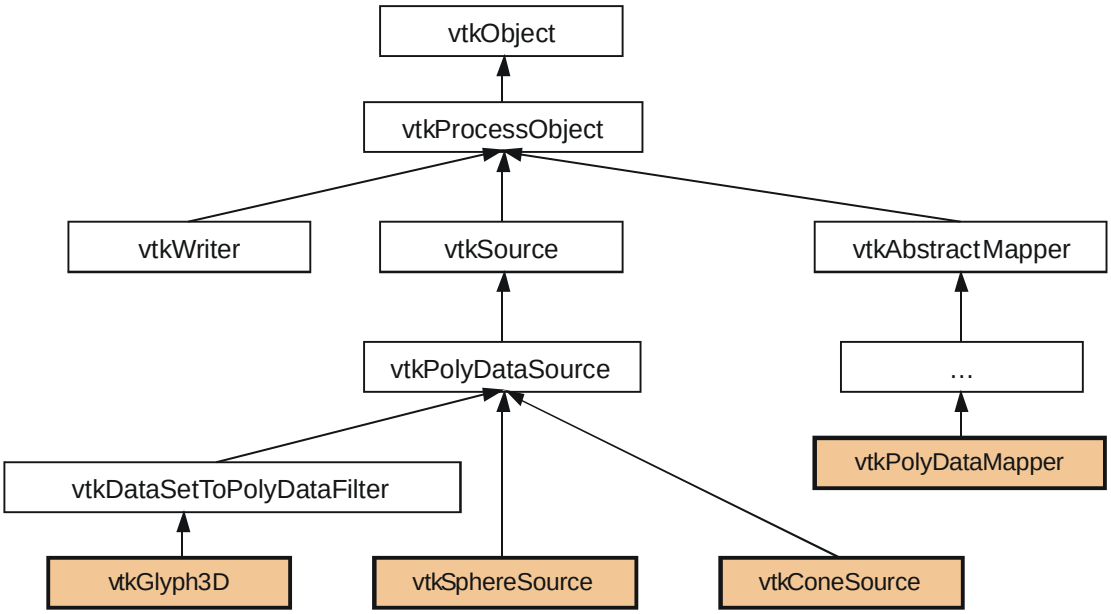

Figure 2.8 – Used process objects as inheritance graph. Instanced object are highlighted

An implementation of the example in *C#* is given in Code 2.2. The line 1 contains creation of the sphere source named *sphere*. The creation is done by static method *New()* of the *vtkSphereSoure*. It is a legacy from the *C++* implementation where it

is because of avoidance of static object creation by means of protected constructor. Lines 2 and 3 contain settings of the sphere *object*. The *Phi* and *Theta* resolution affect the number of meridians and parallels. Note the indentation of the setting lines, which are inserted to improve readability. Line number 4 contains *sphereMapper* creation. The connection between *sphere* and *sphereMapper* is in line 5. The *sphereActor* creation and assignment is in lines 6 and 7.

Following lines 8 and 9 contain the cone source creation. It is very similar to sphere source creation.

Line 10 contains *glyph* creation. The *SetInput* and *SetSource* methods, which are called in lines 11 and 12, implement the multiple input connection of the glyph filter. Note the multiple output of the sphere object that is done by interconnection of its output into two objects (mapper – line 5 and glyph – line 11). Actually, there is only one instance of the sphere data object and both consumers of the data can only read it. Additional setting, of how to map the source object on input object follows in lines 13, 14 and 15. Mapper (*spikeMapper*) of the spikes (glyph output) is created and interconnected in lines 16 and 17. Appropriate actor for spikes (*spikeActor*) is created and assigned in lines 18 and 19.

The renderer is created in line 20. It contains default camera and default lights. Therefore we do not need to add them in this simple example. Lines 21 and 22 contain an addition of sphere and spike actors. The white background of the renderer is set in line 23.

The render-window is created in line 24. The renderer is inserted in line 25. Note the *add()* method is used instead of *set()* method. It is due to that multiple of renderers can be in one render-window. The size of render window is set in line 26. Lines 27 and 28 contain creation and assignment of the interactor that allows basic camera manipulation in renderer. The render method is called in line 29. It invokes an update on the whole graphical and visualization pipeline. Therefore all computation start here, render-window opening included. Finally, in line 30, the event loop is started so the interactor starts the communication with mouse. It ends when the user closes the window.

```
1)     vtkSphereSource sphere = vtkSphereSource.New();
2)       sphere.SetThetaResolution(6);
3)       sphere.SetPhiResolution(6);
4)     vtkPolyDataMapper sphereMapper = vtkPolyDataMapper.New();
5)       sphereMapper.SetInput(sphere.GetOutput());
6)     vtkActor sphereActor = vtkActor.New();
7)       sphereActor.SetMapper(sphereMaper);

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

10)    vtkGlyph3D glyph = vtkGlyph3D.New();
11)      glyph.SetInput(sphere.GetOutput());
12)      glyph.SetSource(cone.GetOutput());
13)      glyph.SetVectorModeToUseNormal();
14)      glyph.SetScaleModeToScaleByVector();
15)      glyph.SetScaleFactor(0.25f);

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

18)    vtkActor spikeActor = vtkActor.New();
19)      spikeActor.SetMapper(spikeMapper);

20)    vtkRenderer renderer = vtkRenderer.New();
21)      renderer.AddActor(sphereActor);
22)      renderer.AddActor(spikeActor);
23)      renderer.SetBackground(1,1,1);

24)    vtkRenderWindow renWin = vtkRenderWindow.New();
25)      renWin.AddRenderer(renderer);
26)      renWin.SetSize(450,450);
27)    vtkRenderWindowInteractor iren = vtkRenderWindowInteractor.New();
28)      iren.SetRenderWindow(renWin);

29)    renWin.Render();
30)    iren.Start();
```

Code 2.2 – The mace example as C# source code

When the event loop starts, just a camera position is changed. Therefore there is no need of new execution of whole pipeline. The update method is propagated through whole pipeline with every new frame but there is no execution except the renderer.

2.4 System Architecture Overview

The Visualization Toolkit consists of two basic subsystems: a compiled core and a wrapper layer. See Figure 2.9. The core layer is a compiled *C++* class library. In case of MS Windows it is distributed as a set of dynamic linked libraries (.DLL), which are necessary for run any application that using VTK. The official[1] wrapper layers make possible to use VTK in the Java, TCL and Python programming languages. The goal of this work is to create wrapping layer for .NET platform.

[1] Standard free-source distribution of VTK available on [Kitware] web site.

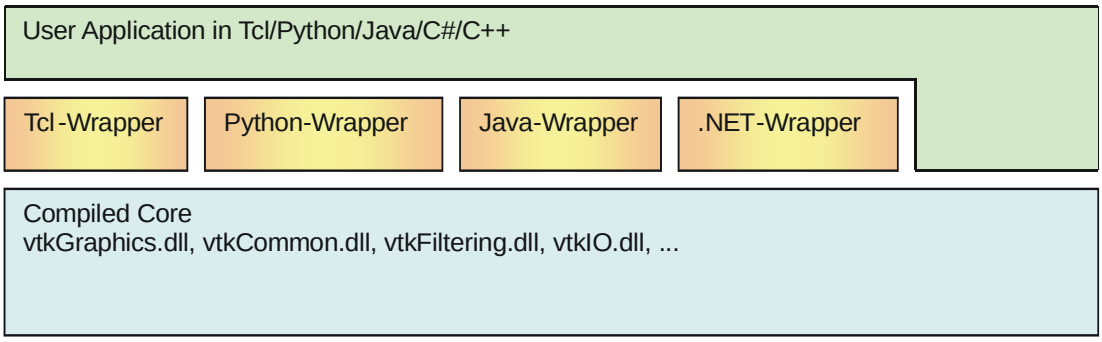

Figure 2.9 - Core, wrappers, and user application. Note the direct access possibility

The most efficient use of the VTK lies in *C++* programming language because the *C++* is "native" for the VTK and provides direct access to the VTK core. Therefore a full functionality, flexibility and speed efficiency can be achieved in final *C++* application. Also the *C++* is only "official" way of how to create new VTK modules. The main reason of this is probably the nonfunctional polymorphism between wrapped and wrapping classes.

In case of wrapper usage, the developer faces two problems. The minor one is a slowdown of resulting application. The second, mayor one, results from differences between *C++* and particular wrapping language because of fact that the *C++* is probably the most powerful programming language ever developed. For example, pointer manipulation is possible in *C++* but not in Java. Also some data types are difficult to transform between so different environments.

2.4.1 Existing Java Interface

The Java interface is considered as the reference interface for the goal .NET interface due to similarity of Java and .NET features (e.g.: ByteCode vs. MSIL, JVM vs. .NET Framework, garbage-collector, etc). Also the Java and .NET languages are semi-interpreted and JITers can be used.

The realization of the Java interface is done by *vtkXXXJava.dll* files that are wrapped by Java Native Interface objects compiled and packed into the *vtk.jar* file.

Source code of the Java wrappers is automatically generated by program *vtkWrapJava*. Its source code can be found in *vtk\wrapping\vtkWrapJava.cxx* file in the VTK source code distribution. This program uses a parser written by means of *Lex* and *Yacc* programs for syntax and lexical analysis. This parser works on *C++* header files and obtains information about classes, their methods and parameters of the methods. Additional information is taken from hint file that contains information about length of arrays since the *C++* source code does not contain the information (In *C++* is field usually pointer only). The resulting data is stored in parsers' internal data structures. These structures are given in Code 2.3.

```
typedef struct _FileInfo
{
  int   HasDelete;
  int   IsAbstract;
  int   IsConcrete;
  char *ClassName;
  char *FileName;

  char *SuperClasses[10];
  int   NumberOfSuperClasses;
  int   NumberOfFunctions;
  FunctionInfo Functions[1000];
  char *NameComment;
  char *Description;
  char *Caveats;
  char *SeeAlso;
} FileInfo;
```

```
typedef struct _FunctionInfo
{
  char *Name;
  int   NumberOfArguments;
  int   ArrayFailure;
  int   IsPureVirtual;
  int   IsPublic;
  int   IsProtected;
  int   IsOperator;
  int   HaveHint;
  int   HintSize;
  int   ArgTypes[MAX_ARGS];
  int   ArgCounts[MAX_ARGS];
  char *ArgClasses[MAX_ARGS];
  int   ReturnType;
  char *ReturnClass;
  char *Comment;
  char *Signature;
} FunctionInfo;
```

Code 2.3 – Internal structures of the parser

Mentioned parser is general (as we can see in its internal data structures) and is used for generating of all others interfaces (TCL, Python). Unfortunately, this parser omitting some parsed methods. May be it is not an error but the goal.

Part 3 .NET Environment

In this part the new .NET environment will be discussed. This text is not detailed description of the .NET, but it is a personal view of the author about this problematic and provides necessary background information for understanding of this work. The aim is on explanation of .NET specific terms and general overview.

3.1 Introduction

Under the .NET term can be comprehended number of things. First it should be initiative, which goal is to build a new generation of operating systems, application servers and number of developing tools. Second, it can be the *.NET Framework* that is the kernel of these products and these entire products depends on it.

3.1.1 .NET as a New Layer

Under the term .NET a new layer that continues with the unification and simplification tendency in software developing can bee seen. Operating system makes transparent a lot of hardware dependencies for programmers. Therefore he can access various hardware devices, without implementation details knowledge. Block diagram of this idea is given in Figure 3.1.

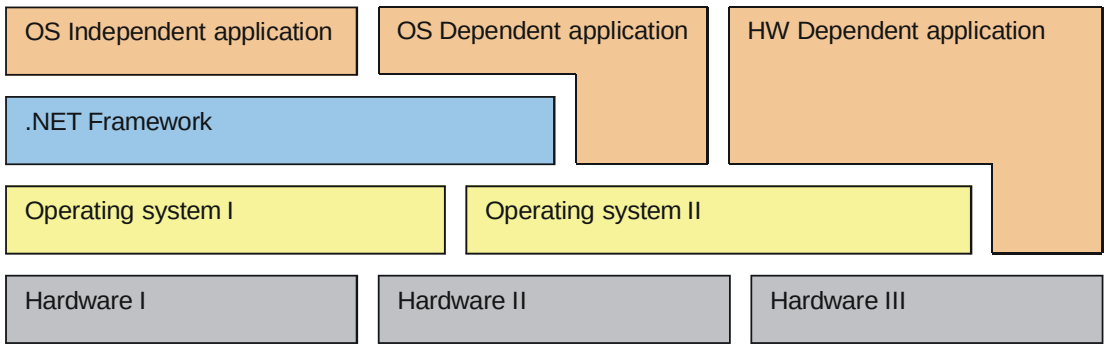

Figure 3.1 – Layers and dependencies

For example, number of sound cards exists. Each one has its specific interface how to access them. But, thanks to the operating system we do not have to write special code for each possible sound card (as in the case of MS-DOS). It is done by the operating system that provides unified sound card interface.

In case of .NET it is the whole operating system instead of any specific hardware part that is going to be unified (or transparent). The application written for the .NET platform can be fully functional without need of any dependency on specific operating system. This kind of "pure" application is portable at any platform for that the .NET Framework exists. This idea is used in Java runtime too.

3.2 .NET Framework (CLI)

The ECMA International Standard ECMA-334 and ECMA-335 (see references [ECMA02a] and [ECMA02b]) discuss the .NET Framework as Common Language Infrastructure (CLI). Both these expressions will be used in following text.

The kernel of the .NET platform is the .NET Framework, which is a new computing platform that simplifies and unifies the application development and deployment. The CLI is designed to fulfill the following objectives:[2]

- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that guarantees safe execution of code, including code created by an unknown or semi-trusted third party.
- To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.

To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

The .NET Framework has two main components: the common language runtime and the .NET Framework class library. The common language runtime (CLR) is a basic element of the .NET Framework. It can be imagine as a "virtual machine" that executes program code, provides services as a memory management, thread management, type controls and exception handling. The code that targets the common language runtime is known as a *managed code* and the code that do not targets the CLR is known as *unmanaged code*. The class library is a wide and compact collection of reusable classes that can be used for application development. It has wide range of functionality such as GUI, database connectivity, distributing application support etc.

Figure 3.2 presents general view on the .NET Framework. At the lowest level is the Common Language Runtime. Above the CLR is situated the Basic Class Library that provides wide functionality that can be used by user application. Above the Basic Class Library lie the Win-Forms and Web-Forms that contain classes necessary for GUI (Graphic User Interface) creation. The Web-Forms are designed for usage on Web and the Win-Forms are determined for desktop application. Both these libraries have unified API that simplifies the conversion from desktop application to web application and back. The higher horizontal level contains the potentially unlimited set of programming languages. The Common Intermediate Language (CIL or MSIL) is the target language (or platform) for all other languages. The CIL is interpreted (or executed) by CLR.

[2] This list is taken and modified from "Overview of the .NET Framework" article in [MSDN] resource.

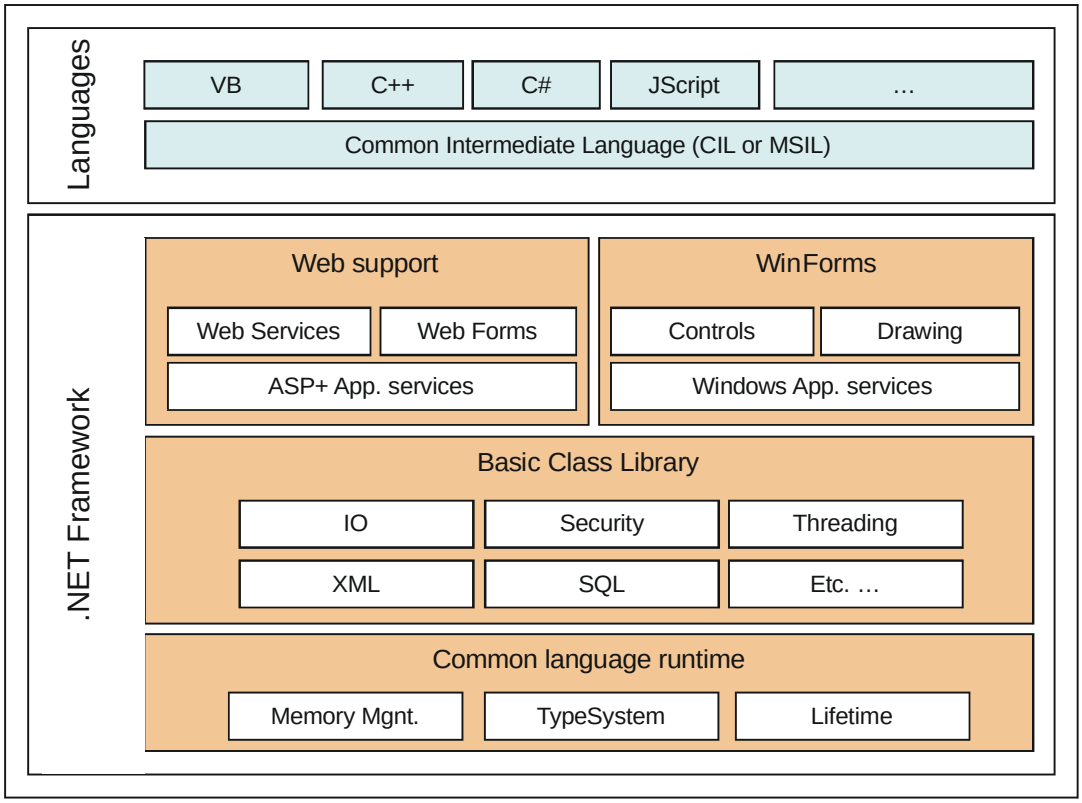

Figure 3.2 - .NET Framework structure with programming languages

3.2.1 Common Language Runtime (CLR)

The CLR is the "virtual machine" that executes the CIL code. It provides memory management (by *garbage collector*); thread execution, code execution, code safety verification, compilation (by *JITter*), and other system services.

The CLR provides memory management with garbage collector service. It means the object is automatically released from memory when becomes unnecessary. Therefore programmers do not have to carefully handle its object termination and the program developing is easier and so faster. Call of a deleted object method is often error that causes the program crash. The memory management can also move any managed data/object in memory (change its memory address) to prevent the memory fragmentation. The side effect is, the pointers are useless since any pointer usually contains relative address to some kind of data/object but the data/object can be moved without any note.

The just-in-time compiler (*JITter*) is responsible for compilation of the CIL code to the native code of particular processor. Therefore the *JITter* can optimize the CIL code for particular instruction set and so obtain extra performance with full CIL portability. Three types of just-in-time compiling exist: install-time compilation, runtime compilation, econo-compilation. The install-time means the code is compiled to the native code of given processor during installation of application. The advantage is that there is no delay during application start. The runtime compilation means the whole CIL code is compiled before program execution. The econo-compilation (also called econoJITter) compiles only the parts of code that are going to be used. It can also release unnecessary parts of compiled code from the memory. Thanks the *JITter* usage there is no real interpretation of the CIL code in the .NET environment.

The code access security is done by its trustworthiness. For example, user can trust that an executable embedded in a Web page can play an animation on screen, but cannot access their personal data, or file system.

The runtime also enforces code robustness by implementing a strict type and code verification infrastructure called the common type system (CTS). The CTS ensures that all managed code is self-describing. Managed code can consume other managed types and instances, while strictly enforcing type fidelity and type safety.

3.2.2 Common Intermediate Language (CIL)

The common intermediate language (also called as MSIL – Microsoft Intermediate Language) is programming language that is processed (or executed) by CLR. It is very close to a native processor assembler. In the terms of Java it is similar to Java byte-code. It is the target language for all (higher) .NET programming languages.

The CIL contains support for object oriented programming, such as virtual method invocation and exception handling.

One of the reasons for introduction of the intermediate language is to provide code description, which does not depend on particular processor instruction set. Program in the CIL is compiled by *JITter* for particular processor just before the runtime, thus it can use its special advantages. Simple example of CIL follows.

Example of CIL Code

Code 3.1 gives simple class with one static method. The *Main()* method has two local variables. They are sets to value one, added together, converted to the string, concatenated with informative string and printed to the console.

```
class Class1
{
  static void Main(string[] args)
  {
    int i, j;
    i = j = 1;
    Console.WriteLine("1 + 1 = " + (i + j));
  }
}
```

Code 3.1 – Simple C# source code

When the Code 3.1 is compiled into the *.exe* form ant then decompiled (by *Ildasm.exe*) into the text form of the CIL, we obtain the Code 3.2. Line 1 contains method header. Allocation of two local variables is on line 6. The instruction on line 7, pushes value one on the top of evaluation stack. The following *dup* instruction duplicates the topmost value. The instruction *stloc.1* pops the topmost value and stores it in the first local variable. The same is done with the zero local variable on line 10. The instruction *ldstr* in line 11 loads given string value to the top of the evaluation stack. Instructions *ldloc.1* and *ldloc.0* loads the local variables values at the top of the stack. The *add* instruction pops the two topmost values, add them together and stores the result at the top of the stack.

```
1)     .method private hidebysig static void  Main(string[] args) cil managed
2)     {
3)     .entrypoint
4)     // Code size       28 (0x1c)
5)     .maxstack  3
6)     .locals init (int32 V_0, int32 V_1)
7)     IL_0000:  ldc.i4.1
8)     IL_0001:  dup
9)     IL_0002:  stloc.1
10)    IL_0003:  stloc.0
11)    IL_0004:  ldstr      "1 + 1 = "
12)    IL_0009:  ldloc.0
13)    IL_000a:  ldloc.1
14)    IL_000b:  add
15)    IL_000c:  box        [mscorlib]System.Int32
16)    IL_0011:  call       string [mscorlib]System.String::Concat(object, object)
17)    IL_0016:  call       void [mscorlib]System.Console::WriteLine(string)
18)    IL_001b:  ret
19)    } // end of method Class1::Main
```

Code 3.2 – Code in Common Intermediate Language

Interesting instruction lies in the line 15. This instruction converts the topmost integer value to the object reference of System.Int32 type. Thus conversion to the string is being possible by *ToString()* method call. This is standard approach of the .NET environment for the primitive data types. Each one is usually processed as value type. When it is necessary its class type automatically boxes (or wraps) it. This approach is chosen due to performance. The value types are usually easier (and so faster) to handle than the class type. The line 16 contains call of string concatenation function. The resulting string is printed on the console by static method *WriteLine()* call. Finally the return from the function is done by the *ret* instruction.

3.2.3 Common Language Specification (CLS)

The CLS is a set of rules that guarantees cross-language interoperability. The application (or library) that is targeted as CLS-compliant can expose only types and features that are common to all other languages targeting CLS. By another words, it is a set of rules, which any CLS-compliant language have to accept and can expose. Actually, the common language specification rules are a subset of a common type system.

3.2.4 Metadata

Metadata is binary information describing any part of program code. Every type and member defined and referenced in a module or assembly is described by metadata that are stored in the special part of program file. Metadata consists of:[3]

- Description of the assembly.
  - Identity (name, version, culture, public key).
  - The types that are exported.
  - Other assemblies that this assembly depends on.
  - Security permissions needed to run.
- Description of types.
  - Name, visibility, base class, and interfaces implemented.

[3] This list is taken and modified from “Metadata overview” in [MSDN] resource.

- Members (methods, fields, properties, events, nested types).

- Attributes
  - Additional descriptive elements that modify types and members.

Metadata is the key to a simpler programming model, which eliminates the need for extra interface definition files e.g. header files, or any external method of component reference. Metadata allows .NET languages to describe themselves automatically in a language-neutral manner. Additionally, metadata is extensible through the use of attributes.

The attribute metadata can be associated with any program element. The class library provides wide variety of attribute classes that can be used. Alongside it, custom attributes can be created and used as well. Metadata are accessible in runtime by reflection mechanism.

Attribute and Metadata Example

Following source codes gives the attribute *Obsolete* example. The Code 3.3 gives the simple method that is marked obsolete by attribute associated with this method. If this method is used, the compiler writes warning with given string. It writes warning only (not error) due to *false* parameter.

```
class Class1
{
  [Obsolete("Deprecated method. Do not use it.", false)]
  void OldMethod()
  {
    Console.WriteLine("I am old method.");
  }
  ...
```

Code 3.3 – C# source code with Obsolete attribute.

Code 3.4 gives the decompiled version of previously introduced example. At the beginning of the method, the instance of *System.ObsoleteAttribute* is created. The parameter values are defined as a block of binary data. In decompiled form they are displayed as a set of hexadecimal numbers with commented string representation.

```
.method private hidebysig instance void  OldMethod() cil managed
{
  .custom instance void [mscorlib]System.ObsoleteAttribute::.ctor(string, bool)
       = ( 01 00 21 44 65 70 72 65 63 61 74 65 64 20 6D 65   // ..!Deprecated me
           74 68 6F 64 2E 20 44 6F 20 6E 6F 74 20 75 73 65   // thod. Do not use
           20 69 74 2E 00 00 00 )                            //  it....

  // Code size       11 (0xb)
  .maxstack  1
  IL_0000:  ldstr      "I am old method."
  IL_0005:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000a:  ret
} // end of method Class1::OldMethod
```

Code 3.4 – Decompiled .exe file that gives the attribute metadata example

3.2.5 Assembly

Assembly is a logical collection consisting of, one or more .EXE or .DLL files and resources with manifest. At the first point of view, assembly can looks like any other unmanaged .DLL (or module). But the managed assembly provides much more functionality like:

self-description – Assembly contains information about exported classes, functions and arguments.

number of version – Each assembly contains its version consists of *Major*, *Minor*, *Build* and *Revision* part. Some resources mix the Build and Revision order.

number of versions of used modules – If assembly requires any other assembly then contains numbers of their versions (and possibly hash code of the file content).

side-by side multi versions – In the .NET environment can exists simultaneously number of the same assemblies with different versions.

assembly is standalone – The assembly does not depend on registry or MTS/COM+ catalogue. This is usually the main difficultly during application deployment.

The Figure 3.3 gives the logical structure of single file and multi file assembly. The Assembly metadata means the metadata associated with the assembly (also called *manifest*).

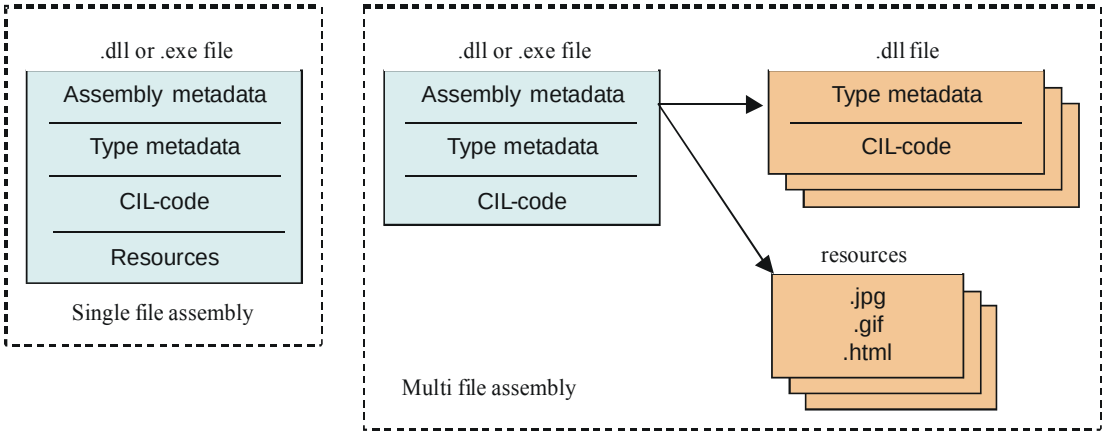

Figure 3.3 – Single- and multi- file assembly structure

3.2.6 Class Library

The class library is large collection of reusable classes with wide variety of functionality. It is designed to be consistent. It makes the class library easy to use and reduce time necessary to learn the new features.

The class library is common language specification compliant. Therefore it can be simply used in any language of .NET platform that is CLS compliant too. This is advantage, because in CLI exists only one class and programmers experience with one language is "portable" on any other language. To learn language specific libraries is probably the most time consuming part of learning of any new language.

The CLI specification guarantees the set of runtime classes that have to be present in any CLI implementation, therefore usage of such classes provides portable program.

3.3 Languages for .NET Platform

Number of languages of .NET platform is generally unlimited. Each language that has compilator targeting the CIL (managed environment) can be considered as a language for .NET platform. There are languages that target managed environment only (like Visual Basic) or managed environment and unmanaged together (like *MC++* and *C#*).

Following subsections contains brief introduction to the most important features of the languages used in presented work.

3.3.1 C# Programming Language

C# (pronounced C Sharp.) is a simple, modern, object oriented, and type-safe programming language. It is designed especially for .NET platform. Therefore it is the most complying language for the CIL and CLI features. Is based on well-known *C++* programming language. Number of ideas is taken from Java programming language. Brief list of important features follows:

- Support single inheritance only
- Support interfaces (special case of class that does not have attributes/data) instead of multiple inheritance
- Support property (possible replacement for *Get/Set* methods)
- Implicitly support managed environment
  - No pointers
  - Garbage collector
- Explicitly support unmanaged environment (by *unsafe* block)
  - Pointers available
  - No garbage collector
- Support events and exception handling

Hallo World Application in C#

Following example is taken and modified from [ECMA02b] resource. The "hello world" program can be written as follows in Code 3.5:

```
using System;
class Hello
{
  static void Main()
  {
    Console.WriteLine("hello world");
  }
}
```

Code 3.5 – Hallo world application in C#

The source code for a *C#* program is typically stored in one or more text files with a file extension of *.cs*, as in *hello.cs*. Using a command-line compiler, such a program can be compiled with a command line like:

csc hello.cs

Which produces an application named *hello.exe*. The output produced by this application when it is run is:

hello world

Close examination of this program follows:

- The *using System;* directive references a namespace called *System* that is provided by the Common Language Infrastructure (CLI) class library. This namespace contains the Console class referred to in the *Main* method. Namespaces provide a hierarchical means of organizing the elements of one or more programs. A using-directive enables not fully qualified use of the types

that are members of the namespace. The *hello world* program uses *Console.WriteLine* as shorthand for *System.Console.WriteLine*.

- The *Main()* method is a member of the class *Hello*. It has the static modifier, and so it is a method on the class *Hello* rather than on instances of this class.
- The entry point for an application (the method that is called to begin execution) is always a static method named *Main()*.
- The "hello world" output is produced using a class library of CLI.

For *C* and *C++* developers, it is interesting to note a few things that do *not* appear in the "hello world" program.

- The program does not use a global method for *main*. Methods and variables are not supported at the global level; such elements are always contained within type declarations (e.g., class and struct declarations).
- The program does not use either *::* or -> operators. The *::* is not an operator at all, and the -> operator is used in only a small fraction of programs (which involve unsafe code). The separator . is used in compound names such as *Console.WriteLine()*.
- The program does not contain forward declarations. Forward declarations are never needed, as declaration order is not significant.
- The program does not use *#include* to import program text. Dependencies among programs are handled symbolically rather than textually. This approach eliminates barriers between applications written using multiple languages. For example, the *Console* class need not be written in *C#*.

3.3.2 C++ Managed Extension (*MC++*)

C++ programming language is probably the most general and powerful programming tool ever developed. Even this, the CIL is sufficient to be a target platform for *C++* compiler. However once a *C++* program is compiled to CIL, it may utilize the powerful features of the runtime (such as garbage collection, cross-language cooperation, etc.) but the *C++* needs some added tools. This is the reason for the *Managed Extension* of the *C++*.

Code 3.6 gives simple "hallo world" managed application in *MC++*.

```
#using <mscorlib.dll>
using namespace System;

void main()
{
  Console::WriteLine("hello world");
}
```

Code 3.6 – The MC++ hallo world application

- The source code is usually stored in text file with *.cpp* extension. Can be compiled by *C++* compiler with */clr* command line option. The *#using <mscorlib.dll>* command, imports the CLI class library to be used. The rest is similar to previously presented "hallo world" in *C#*. The only major difference is that the *C++* does not need to encapsulate the *main()* method to a class.

- The */clr* compiler option does not alter the semantics of an existing *C++* program. For example, *C++* classes do not become garbage collected unless they are modified. Such features are only provided for *Managed Extensions classes*.
- The *MC++* programming language is powerful tool that allows very general mixing of unmanaged code with managed code. Alongside it, the *MC++* maintains full backward compatibility with any existing *C++* code. It makes it perfect tool for porting of existing code.

Managed Extension Keywords

Following list of keywords provides access to managed extension features.

__abstract __box __delegate __event __gc __identifier __interface __nogc __pin __property __sealed __try_cast __typeof __value

Managed Extension Classes

As mentioned before, ordinary definition of class in *MC++* does not provide benefits of the CLI. The class requires to be declared as garbage collected by *__gc* keyword. Any class marked by the *__gc* keyword is called as garbage collected or managed class. It takes all advantages and limitations provided by the CIL. Such as:[4]

Extensions:

- A declaration of a __gc class shall always have the *__gc* keyword.
- A __gc class can have a data member that has type *pointer-to* any unmanaged type.
- A __gc class can contain a user-defined destructor.
- Operator *delete* can be called on a pointer-to a *__gc class* in order to force the destructor to run immediately.
- A __gc class can implement any number of *__gc interfaces*.
- A __gc class can contain properties.
- A __gc class can be marked with the *__abstract* keyword.
- A __gc class can be marked as "sealed".
- A __gc class can declare a static class constructor.
- A __gc class can declare a constructor.
- A __gc class can have a visibility specifier.

Limitations:

- A __gc class shall not inherit from an unmanaged class.
- A __gc class shall not have an unmanaged class derived from it.
- A __gc class shall not inherit from more than one managed class.
- A __gc class shall not declare a user-defined copy constructor.

[4] This List is taken and modified from "Managed extension for C++ specification" article. Stored in file *managedextensionsspec.doc* file, embedded with the Visual Studio .NET distribution.

- A __gc class shall not declare or define friend classes or functions.
- A __gc class shall not declare or define a *new* or *delete* operator.
- A __gc class shall not contain a *using* declaration.
- The calling convention of a member function of a *__gc* class cannot be redefined to a native *C++* calling convention, for example, to *__cdecl*.

There are two another keywords that can mark managed class: *__value* and *__interface* keyword. The *__interface* keyword marks classes that are called *interface* in the terms of CIL, *C#* and Java with all its limitation and advantages. The *__value* keyword is designated to represent small, short-lived data items for which full garbage collection would be too costly.

Number of any other types known from the CIL specification can be used in *C++* such as a *__value enum*, *__gc pointer*, and *__gc reference*. They are similar to their unmanaged counterparts.

3.4 Existing Code Cooperation

Support for cooperation with already existing code is essential for the success of .NET platform. Large amount of useful unmanaged software (or libraries) already exists. Therefore the use of this software is usual in any new environment.

The .NET environment provides support for managed and unmanaged code cooperation. This cooperation can be in both directions: the managed code can call unmanaged code and also the unmanaged code can call managed code. How complicated it is, it depends on unmanaged technology we wish to use, e.g. the easiest way is probably in the case of COM technology (in both directions).

3.4.1 COM Cooperation

This kind of cooperation is the easiest way how to cooperate between managed and unmanaged environment. The creation of COM-callable wrapper (makes managed class accessible as COM module) and Runtime-callable wrapper (makes COM module accessible from managed environment) is fully automated by using programs embedded in Visual Studio .NET.

3.4.2 Unmanaged DLL Cooperation

Standard way for cooperation with ordinary (unmanaged) .DLL module is by *Platform Invocation Service* (*PInvoke*). With the *PInvoke* mechanism it is possible the direct calling any Win32 API function or function exported from arbitrary dynamic-link library (DLL). Everything we have to do is to use the *DllImport* attribute. Simple example of a Win32 API function call follows in Code 3.7.

```
class Demo
{
  [DllImport("User32.dll", EntryPoint="MessageBox", CharSet=CharSet.Auto)]
  public static extern int MsgBox(int hWnd, String text, String caption, uint type);

  static void Main(string[] args)
  {
    MsgBox(0, "Called from C#", "PInvoke", 0);
  }
  ...
}
```

Code 3.7 – The PInvoke example in C#

Disadvantage of *PInvoke* service is the fact that there is no straightforward way how to make accessible object-oriented libraries (especially non-static class methods). This is crucial disadvantage due to the VTK is fully object-oriented.

3.4.3 Managed and Unmanaged Code Mixing

In this approach is used the power of *MC++*, which is "backward compatible" with ANSII *C++* (in following text marked as *AC++*) and provides the CIL functionality as well. This is called It-Just-Works (IJW) mechanism. Therefore it looks like perfect tool for flexible interface creation.

It means we can create managed class, which uses unmanaged features of any library in standard *C++* way. The difficultly is to correctly convert managed types to unmanaged types and back. The garbage collector, useful service in managed environment, causes serious problems with unmanaged code cooperation. When we wish to pass any data from managed to unmanaged environment there is necessary to "convince" the garbage collector to do no moving or no de-allocation of the data. Fortunately, the CLI class library provides the *GCHandle* class that provides required functionality. By calling of *Allocate()* method we can fix the data in garbage-collected heap and then release them by *Free()* method call. By means of the *AddrOfPinnedObject().ToPointer()* method call we can obtain real pointer to the garbage-collected heap.

Part 4 VTK for .NET

This part describes our original solution of the straightforward usage of the VTK in the .NET environment. This is done by interfacing layer between managed applications and unmanaged compiled core of the VTK. The interfacing layer is managed assembly that uses functionality of VTK *C++* API. The managed interface seems similar to its unmanaged version with few limitations and differences.

4.1 Goals

The main goal of this work is to make VTK easily accessible from .NET environment. It covers number of matters.

- Offer managed equivalents of unmanaged VTK classes.
- Convert ANSII *C++* data types to .NET (CTS) data types. (E.g. *char** that represents zero terminated string in ANSII *C* should be *System.String* of the .NET environment.)
- Avoid unsafe blocks in resulting application. (User does not have to learn the managed and unmanaged code cooperation)
- Use garbage collector. (User does not have to call the *Delete()* method of VTK objects)

Next sub-goal is to use the original VTK precompiled dynamic linked libraries. It means, no changes in original VTK source code. Implemented solution is a layer between unmanaged DLLs of the VTK and a pure managed application as given in Figure 4.1. This layer is named *vtkDotNetWrap*.

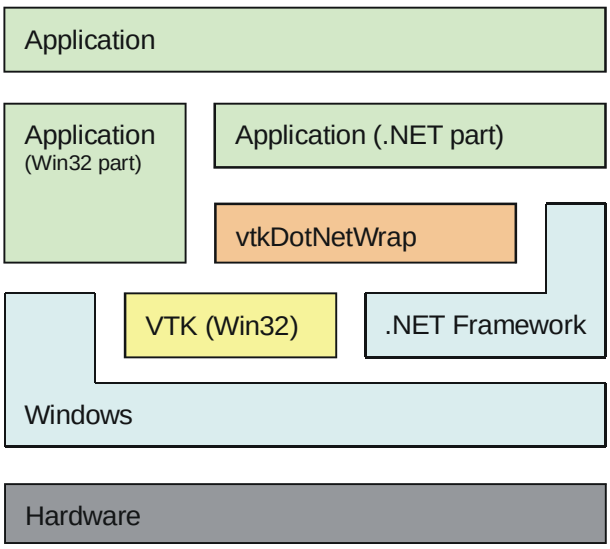

Figure 4.1 – Placement of interfacing layer in resulting application

4.2 Approach

The solution of the interface is realized as a managed layer between (possibly) pure managed application and the unmanaged VTK library. This layer consists of managed

wrap-classes (or proxy-classes), which provides functionality of VTK classes. Each VTK class has right one wrap-class. These wrap-classes can be easily instantiated and used in managed environment.

The wrap-class contains internally wrapped-class or more precisely instance of wrap-class contains instance of wrapped-class. The instance of the wrap-class is responsible for the instance of the wrapped-class (allocation, deallocation and accessing its functionality). Graphical demonstration of the wrapping is given in Figure 4.2.

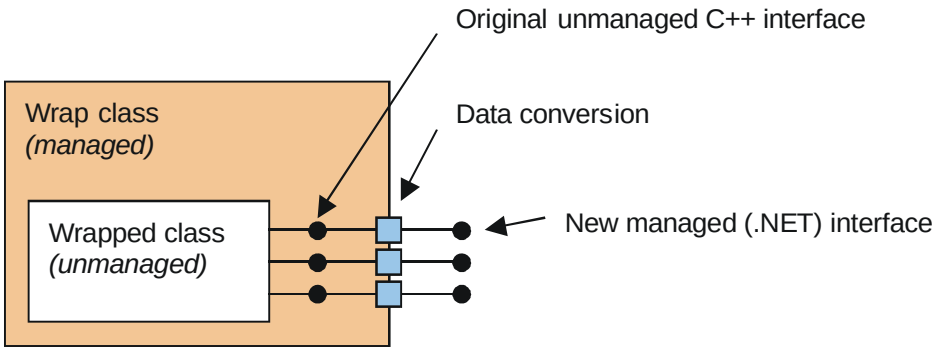

Figure 4.2 – Wrap-class contains wrapped class and suits its interface into .NET manner

The most problematic part is the data conversion, due to different characteristics of managed and unmanaged environment. The target .NET environment is relatively type strict in comparison with the ANSII *C++* (VTK native). Next problem arises from completely different memory management. The garbage collector can release memory (when there is no reference) or even move the block in memory (change its address) without any note. This process is completely undeterministic. Thus, the wrap-class has to pine the memory block before passing it to the unmanaged environment, etc.

4.3 Realization

For implementation of the interfacing layer is required a tool (programming language) that allows use of already existed unmanaged library (DLL) and utilization of managed features simultaneously. The most flexible tool for this is probably *C++* Managed Extension (*MC++*). Thanks its "backward compatibility" with unmanaged environment we can simply access the functionality of the VTK. Thanks its "managed extension" we can mark some of our classes as garbage-collected and so accessible from managed environment. This feature of *MC++* is called "It Just Works" (IJW).

4.3.1 Composition vs. Inheritance Discuss

This subsection is little speculation about object-oriented principles and possibilities. It is not very important for whole work. In fact, here proposed consideration is probably "blind alley".

There are two ways how to (hierarchically) join functionality of two classes in object-oriented programming, the composition and the inheritance. There are two general questions that help to decide what to use:

- Does the class *B* contain class *A*? If it is so then use composition.
- Is the class *B* (a specialization of) class *A*? If it is so then use inheritance.

Literature usually contains simple example. Does *car* contain *engine*? It is definitely true, so the composition should be used and the object *car* should contains object *engine*. Example for inheritance: Is *roadster* a specialization of *car*? The answer is yes. Then, we inherit the class *roadster* from class *car*.

In the case of wrapping we can possibly answer both questions "yes". The wrap-class could be specialization of the wrapped class (adds the managed features) and also the wrap-class can contain the wrapped class.

From the first point of view, the inheritance solution seems more simply. The managed wrap-class would derive all functionality of the unmanaged wrapped class. The wrap-class would access all protected members of the wrapped class. The polymorphism would be simply functional.

Unfortunately, the inheritance between managed and unmanaged classes is not possible and therefore we have to use the second solution - the composition.

The solution by composition results in two main difficulties. Protected members of wrapped class are not accessible from wrap-class and polymorphism does not automatically work. These problems are the main reason why user can create VTK modules in *C++* only.

This idea continues in section 4.3.4 Double Wrapping Discuss. In following subsection is presented implementation of the solution by composition.

4.3.2 MC++ Wrap-class

The realization of wrap-class in *MC++* (with composition) is simple. We can use managed and unmanaged paradigms in one piece of source code. An instance of managed 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 call of appropriate wrapped-class method with data conversions. There are some exceptions especially in constructors. Source Code 4.1 is a representative part of *MC++* 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());
  }
  // ...
```

Code 4.1 – Part of MC++ wrap-class source code

The *w* member variable contains pointer to the unmanaged wrapped class. When any method of the wrap-class is called then is called appropriate method of the wrapped class by this pointer. Note the call of *GetClassName()* method with data conversion. There is creation of *String* object from zero-terminated string returned from unmanaged method.

The wrapped object is created in *New()* static method of the wrap-class. See Code 4.2. The wrapped-class is deallocated in destructor of wrap-class that is automatically called by garbage collector.

```
vtkSphereSource * vtkDotNetWrap::vtkSphereSource::New()
{
  vtkSphereSource * ret;
  ret = new vtkSphereSource(::vtkSphereSource::New());
  return ret;
}
```

Code 4.2 – Creation of wrap-class and wrapped-class by static method New

4.3.3 Data Conversion

In this subsection we will discuss the data conversions between managed and unmanaged environment as they are implemented in presented interface.

There is number of set of data types that have to be handled in different manners. List of successfully wrapped data types follows:

- Primitive data type (*int, float*, …)
- Reference type (*int &, float &*, …)
- Pointer/field data type (*int *, float **, …)
- Zero terminated string (*char **)
- Pointer to VTK class (*vtkObject **)
- *const* modification of previously presented data types
- Pointer to function (*void (*f) (void *)*)

There also exists number of special cases that are not successfully wrapped. The list is similar to list of unwrapped data types as in case of Java, TCL and Python wrappers distributed with VTK. This list follows:

- ANSII *C++* specific data types (*FILE, iostream, ...*)
- Pointer to function different from *void (*f) (void *)*
- Value and reference type of VTK class
- Void pointer as argument

Primitive Data Types

Simple (or primitive) data types can be passed as an input argument and/or as a return value. In this case there is no problem and the IJW mechanism of the *MC++* solves transparently all necessary conversions.

All these data types have straightforward equivalent in managed environment. For example, the ANSII *C++ int* has straightforward equivalent in CTS named *System.Int32* with *int* alias. Thus, all is intuitive and transparent. Example of this case follows in Code 4.3 (method argument) and Code 4.4 (return variable).

```
// void SetRadius (float );  codes: 2(1)
void vtkDotNetWrap::vtkSphereSource::SetRadius(float arg0)
{
  w->SetRadius(arg0);
}
```

Code 4.3 – Simple data type is passed directly to unmanaged environment

```
// float GetRadiusMaxValue ();  codes: 1()
float vtkDotNetWrap::vtkSphereSource::GetRadiusMaxValue()
{
  float ret;
  ret = w->GetRadiusMaxValue();
  return ret;
}
```

Code 4.4 – Primitive data type as a return variable

Reference Types

Reference types are slightly more complicated than value types. Common use of reference types is to pass the change of the argument out from the method. By another words, the argument of a method can be used for output from the method.

Equivalent in CTS (*C#* syntax) is the *ref* type. In *MC++* is the syntax identical but alias (*int* instead of *System.Int32*) cannot be simply used.

We have to create unmanaged temporary variables, because there is not possible to simply obtain reference to managed type. Example of particular solution is given in Code 4.5.

```
// virtual void ViewToWorld (float &wx, float &wy, float &wz);  codes: 2(101, 101, 101)
void vtkDotNetWrap::vtkRenderer::ViewToWorld(System::Single & arg0
                                        , System::Single & arg1, System::Single & arg2)
{
  float tmp0 = arg0;
  float tmp1 = arg1;
  float tmp2 = arg2;
  w->ViewToWorld(tmp0, tmp1, tmp2);
  arg0 = tmp0;
  arg1 = tmp1;
  arg2 = tmp2;
}
```

Code 4.5 – Passing of reference data types

Pointer/Field Data Types

Pointer in ANSII *C++* usually means field. Fortunately, in case of VTK there are all pointers to primitive data type a field. As a managed equivalent of the pointer the *System.Array* has been chosen.

The field that is passed as argument from managed environment is allocated on the garbage-collected heap (can be moved in memory without any note). Therefore there is necessary to fix the field in memory and then get the pointer, which can pass to unmanaged environment. Example follows in Code 4.6.

```
// void SetCenter (float  a[3]);  codes: 2(301)
void vtkDotNetWrap::vtkSphereSource::SetCenter(float arg0 __gc [])
{
  GCHandle tmp0 = GCHandle::Alloc(arg0, GCHandleType::Pinned);
  w->SetCenter((float *)(tmp0.AddrOfPinnedObject().ToPointer()));
  tmp0.Free();
}
```

Code 4.6 – Pinning of field in managed memory

When pointer (field) is a returned from an unmanaged method than it is necessary to allocate appropriate array in managed memory and copy all elements. Because a pointer does not contain information about field length, this information is passed as argument of appropriate conversion subroutine. Example follows in Code 4.7.

```
// float *GetOrientation ();  codes: 301()
float vtkDotNetWrap::vtkTransform::GetOrientation() __gc []
{
  float ret __gc [];
  ret = wgPtr2Field_float(w->GetOrientation(), 3);
  return ret;
}
```

Code 4.7 – Conversion from pointer to field

String Data Types

Each pointer to char is considered a zero terminated string. As an equivalent has been chosen the *System.String*. The conversion from the zero terminated string to *System.String* is straightforward. The *System.String* contains constructor that accepts the zero terminated string. Example is given in Code 4.8.

```
// const char *GetClassName ();  codes: 1303()
System::String * vtkDotNetWrap::vtkSphereSource::GetClassName()
{
  System::String * ret;
  ret = new System::String(w->GetClassName());
  return ret;
}
```

*Code 4.8 – Conversion from char * to System.String*

The conversion in other direction is more complicated. There is one global block of (unmanaged) memory allocated by the *vtkDotNetWrap* interface called *tmpStr*. An address of this memory is always passed instead of the related string. The *wgStr2Char()* function simply copies char by char the content of the *System.String* object to the *tmpStr* memory block. Example is given in Code 4.9.

```
// int IsA (const char *name);  codes: 4(1303)
int vtkDotNetWrap::vtkSphereSource::IsA(System::String * arg0)
{
  int ret;
  ret = w->IsA(wgStr2Char(arg0));
  return ret;
}
```

*Code 4.9 – Conversion from System.String to char **

Pointer to VTK Class

A pointer to any VTK class can be argument as well as return variable. Thus, there is necessary to know how to wrap and unwrap any class at application run-time.

The unwrapping is simple. The pointer to unmanaged class is internally (for the *vtkDotNetWrap* namespace) accessible. So, after null pointer check it can be simply used. Example follows in Code 4.10.

```
// void Concatenate (vtkLinearTransform *transform);  codes: 2(309)
void vtkDotNetWrap::vtkTransform::Concatenate(vtkLinearTransform * arg0)
{
  w->Concatenate(((arg0 == NULL) ? NULL : arg0->w));
}
```

Code 4.10 – Conversion from managed class to unmanaged class (unwrapping)

The second direction is a little bit more complicated. In this case we have to create instance of appropriate wrap class with given wrapped class. Example follows in Code 4.11.

```
// vtkImageData *GetInput ();  codes: 309()
vtkImageData * vtkDotNetWrap::vtkTexture::GetInput()
{
  vtkImageData * ret;
  ret = new vtkImageData(w->GetInput());
  return ret;
}
```

Code 4.11 – Conversion from unmanaged class to managed class (wrapping)

Pointer to Function

Pointer to function is used for setting of a callback method. The first equivalent, which we can see in CTS, is probably "delegate" but its use is not really straightforward. In this work is used completely different approach for callbacks. The disadvantage is that only callbacks with *void* pointer parameter can be implemented.

The *vtkDotNetWrap* layer offers interface called *wgICallback*. It is simple managed interface with just one method (called *Callback()*). When user wishes to create its callback method then he implements the *wgICallback* interface in arbitrary class. The overridden *Callback()* method is considered (and called) as the registered pointer to function. Equivalent for data passed in ANSII *C++* (as the void pointer) are the member variables of the given user class (which implements the *wgICallback* interface).

This solution is not very complicated for implementation. Instead of pointer to data is passed packed (converted to void pointer) *GCHandle* of the *wgICallback* class. Instead of pointer to function is passed pointer to one general function that unpacks the data void pointer to *wgICallback* class and then calls its *Callback()* method. Example of wrapping of a registration method is given in Code 4.12.

```
// void SetUserMethod (void (*func)(void *) , void *arg);  codes: 2(5000)
void vtkDotNetWrap::vtkRenderWindowInteractor::SetUserMethod(wgICallback * arg0)
{
  w->SetUserMethod(wgCallbackInternalFunction, wgPackToVoid(arg0));
}
```

Code 4.12 – Callback registration method and conversion from wgICallback interface to void-packed GCHandle

Unimplemented Data Type Conversion

There is number of data types that are not successfully converted. This set is similar to other wrappers (Java, TCL, Python). In general it is minority of data types and VTK can be effectively used without these data types.

First group is ANSII *C++* specific complex data types, such as *FILE* and *iostream*. These data types have no straightforward equivalent in CTS. They would have to be wrapped similarly to other VTK classes.

Probably the most serious unimplemented data type is the nonparametric callback. Due to the chosen system of callback conversion there is no straightforward way how to implement them. It could be considered as work for possible continuator.

VTK has some useful rules, such as "all VTK classes can be only used by pointer". Unfortunately there is minor number of exceptions and some methods expect reference or value type of a VTK class. With regard to implemented system of dynamic wrapping there is no straightforward way how to implement these exceptions. Fortunately, it is only minor number of cases.

Final minor unimplemented data type is the void pointer as argument. It is pointer to unspecified data set and it is not easy to say what equivalent should be used in CTS. Fortunately, it is really rare case in VTK.

4.3.4 Double Wrapping Discuss

As was mentioned before, the main disadvantage of the simple wrapping approach is probably the inability of effective usage of polymorphism. In this subsection we will discuss possible solution by "double wrapping". This solution is not implemented in resulting application and this subsection is here only as a suggestion for possible continuator on this work.

The double wrapping means to use two kinds of wrappers. The first one unmanaged and the second one managed. Let us call them level-one wrapper (L1-Wrapper) and level-two wrapper (L2-Wrapper).

The L1-Wrapper is unmanaged and is derived from wrapped class. It contains handle to the L2-Wrapper. It overrides all virtual methods of the base class with code that calls the same method in L2-Wrapper. From another point of view, it wraps all virtual methods of the L2-wrapper.

The L2-Wrapper is managed and contains the L1-Wrapper. Its purpose and structure is very close to previously presented simple wrap-class.

Possible way how to implement the double wrapping, is given on the following example.

Let us have unmanaged Win32 class (called *Win32Class*) in DLL that we cannot modify. This class contains two methods whose implementation is not important. This is equivalent of a VTK class. See Code 4.13.

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

Code 4.13 – The unmanaged base class we wish to wrap

The L1-wrapper (Code 4.14) is also unmanaged and provides access to protected methods and is responsible for correct calling of virtual method of L2-wrapper. Note that the direct inheritance is used.

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

Code 4.14 – The unmanaged L1-wrapper

Things are going to be more complicated. Following method in Code 4.15 unpacks the *GCHandle* from *void* pointer and calls virtual method of the L2-wrapper. This method is actually the one, which is called by the polymorphism mechanism.

```
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;
}
```

Code 4.15 – The L1-wrapper calls the L2-wrapper

L2-wrapper is very similar to the simple wrap-class presented above. It is managed and makes L1-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()");
    }
}
```

Code 4.16 – The managed L2-wrapper

This approach seems to be a possible way to make inheritance and polymorphism functional, thus the VTK modules could be created in *C#*. Simple examples based on this works well. But implementation for whole VTK is beyond scope of this work. There are two main problems:

- Automatically overridden virtual method can cause infinite cycle.
- There is necessary to review all data type conversions because we wrap managed method by unmanaged (it is all upside-down).

4.4 Wrap-class Generating Process

With regard to VTK size (~800 classes, ~9MB of wrap-class source code, ~19MB of HTML reference manual) there is necessary to automate the processes of wrap-class generating. Our presented generator has following features:

- The output from the generator is the *MC++* source code of wrap-class that can be compiled without any additional modification.
- Secondary output of the generator is a reference manual.
- The input for the generating process is complete set of *C++* header files of the VTK. These header files contains enough information about classes that are going to be wrapped.

The generating process is divided into two parts. At the first part it is necessary to obtain information about VTK class, which is going to be wrapped. This is matter of parser. The second part is the generator that uses the output from the parser. For data exchange between the parser and the generator the intermediate files (IF-files) are used.

It is a simple text file that contains appropriate information about a class its methods and parameters. Schematic view of the whole process is shown in Figure 4.3.

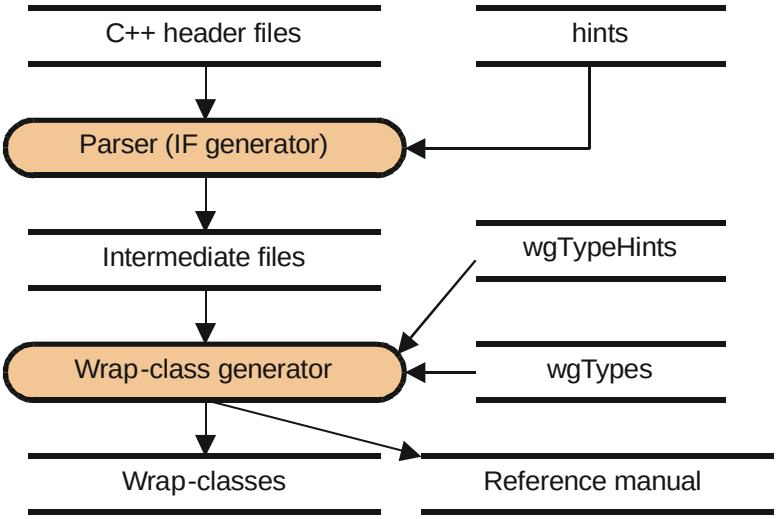

Figure 4.3 – Generating process scheme

4.4.1 The Parser

Fortunately, VTK already contains number of wrappers (Java, TCL, Python). Therefore its structure is already prepared for wrapping. The VTK source code distribution contains parser that is introduced in section 2.4.1 Existing Java Interface. For this work is this parser assumed as a suitable source of information about classes. The inputs of the parser are the *C++* header files and additional *hint* file with field length information.

The already mentioned structures of the parser (see Code 2.3) can be easily processed by implementation of the *vtkParseOutput()* function (see Code 4.17). The *fp* variable contains the output file descriptor and the *data* variable contains the internal structures of the parser, which are going to be processed (printed out).

```
/* print the parsed structures */
void vtkParseOutput(FILE *fp, FileInfo *data)
{
```

Code 4.17 – The parses output function

The only activity that is done by implementation of this function is to print these internal structures as simple as possible to some kind of text file. This text file should be easily readable for program as well as for human by text editor. Sample of this intermediate text file follows in Code 4.18.

```
ClassName vtkAbstractMapper
NameComment  vtkAbstractMapper - abstract class specifies interface to map data
Description  vtkAbstractMapper is an abstract class to specify interface between data
and $ graphics primitives or software rendering techniques. Subclasses of $
vtkAbstractMapper can be used for rendering 2D data, geometry, or volumetric$ data.$$
SeeAlso  vtkAbstractMapper3D vtkMapper vtkPolyDataMapper vtkVolumeMapper$
HasDelete No
IsAbstract No
IsConcrete Yes
NumberOfSuperClasses 1
SuperClass vtkProcessObject
NumberOfFunctions 25
  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
  Comment None
```

Code 4.18 – Sample of intermediate text file – the output of the parser

As we can see, this file contains important information such as, class name, class description comment (can be used for generated reference manual), list of methods with list of arguments, etc.

This intermediate text file is invaluable for debugging of the generating process. Because any programmer can easily see what is the exact output of the parser. Sometimes the parser does not work exactly as we could expect.

Data Type Codes

The parser represents data types by an integer value (type code). Complete list of these codes is in Appendix C. Example of some type codes follows:

- 1 – float
- 3 – char
- 4 – int
- 9 – VTK class
- 10 – unsigned modification
- 100 – reference modification
- 300 – pointer modification
- 1000 – const modification

For example code 1314 means constant pointer to unsigned integer (*const unsigned int* *). This system is designed well and generally does not need any changes for purpose of this work. Some minor problems are outlined in following subsection.

4.4.2 The Generator

Implementation of the generator is the focal point of this work. Simply said, it translates the information from the intermediate text file to *MC++* source code of a

wrap-class. The secondary output of the generator is the reference manual in HTML that contains list of classes with methods and brief description.

This program is not user friendly and is not designed for public use but only for "programmers use". Details how to use and modify this generator is described in Appendix B – Programmers Manual.

The generator starts with loading of intermediate text files (Code 4.18) and creates list of all classes and its methods (in memory). During this process are some unsuitable data type codes overridden, some implicitly (directly wired in generators code) some explicitly by information given in *wgTypeHint.txt* file. If some unknown data type is found than it is written to *UnknownCode.txt* file. Afterwards starts generating itself.

First is generated the main *vtkDotNetWrap.cpp* file that contains inclusion of all others *.cpp* files generated later[5]. This file also contains some statistic information such as, total number of classes, its methods and the number of successfully and unsuccessfully wrapped methods.

Then is generated wrap-class source code one by one: headers (*.h*) and implementations (*.cpp*) files. In this final part of generating is very important the *wgTypes.txt* file, which contains information how to convert data types to managed environment and back. This information is stored in form of a set of conversion macros. The generator expands these macros at appropriate placement.

Parsers Type Codes Overwriting

As was mentioned before, some type codes from parser are unsuitable. Here proposed solution adds some new type codes. Their value is 5000 and more. Complete list follows

- 5000 general pointer to function
- 5001 *vtkIdType* – replace the *long* object ID value
- 5002 *vtkIdType * field*
- 5003 *FILE* * - problematic type
- 5004 *vtkIdType &*
- 5005 *wgICallback* - implemented version of (void *) parametric callback function

Two ways of overriding the type code are used: implicit and explicit. The explicit overriding is defined by enumeration in *wgTypeHints.txt* file. The data type for overriding is defined by class name, method name, argument position and original data type. The final item in this record is the new (required) data type code. Example follows in Table 4.1.

ClassName	MethodName	ArgPos	Ori	New
vtkByteSwap	Swap4BERange	0	306	5002
VtkCell	Initialize	1	306	5002
vtkCellArray	ReplaceCell	2	306	5002

Table 4.1 – Sample of wgTypeCode.txt file, which explicitly overrides some type codes

[5] This approach is much faster then to compile each *.cpp* file extra and then links them together.

The second way of overriding is the implicit overriding. This overriding is defined directly in the source code of generator. It overrides the data type code when defined features are satisfied. For example, a code 5000 (general pointer to function) is substituted for code 5005 (pointer to function with parameter) when the appropriate function signature contains the *","* comma character.

Conversion Macros

Conversion macros contains information how to handle a data type. They are stored in tab-separated text file called *wgTypes.txt*. List of particular macro types with its meaning follows:

- **Code** – code of data type. Given by VTK parser and possibly overridden by generator
- **ACppDecl** – How to declare in unmanaged *C++* (ANSII *C++*)
- **MCppDecl** – How to declare in managed *C++*
- **ToACpp** – How to convert from managed to unmanaged (ANSII)
- **ToMCpp** – How to convert from unmanaged to managed
- **TmpDecl** – How to declare and initialize temporary variable
- **TmpRet** – How to finalize temporary variable
- **IsProblematic** – If yes every function containing this type is commented out

Each data type (which has code number) has these entire macros (as a row in the table). Some items in the record can be void, such as in the example in Table 4.2. Note the *IsProblematic* item. It is useful for possible manual modification of resulting wrap class-source code. The unwrapped method is part of the source code but it is commented out.

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

Table 4.2 – Example of conversion macro for zero terminated string

Symbols for expansion are inside angle brackets. Complete list follows:

- **<var>** - variable name (e.g. *arg0*)
- **<class>** - class name (e.g. *vtk3DSImporter*)
- **<field>** - filed size (e.g. *8*)
- **<tmp>** - temporary variable (e.g. *tmp0*)

These symbols are replaced for appropriate variable name (or class name, or temporary variable, etc.). Complete list of conversion macros is given in Appendix C.

The placement of conversion macros in method wrapping is given in Code 4.19. It is a "pseudo-code" for general overview. The angle brackets contain the "argument" of a macro.

```
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;
}
```

Code 4.19 – General method wrapping with macros

This system of data type conversion is flexible enough to define all necessary data type conversion without need of any manual modification of generated source code.

4.4.3 Generated Wrap-classes

All generated wrap-classes are stored in output directory as well as the main file and replicated help file. They do not need any manual modification before compilation. The main file can be compiled with the *MC++* compiler. Microsoft Visual Studio .NET (version 7.0) was used for testing as well as for compilation the release version. The compilation process takes several minutes.

As it was mentioned before, the unwrapped methods are presented in the source but commented out. If some user really need some of them and knows how to convert a particular data type then he/she can simply uncomment, modify and compile the interfacing layer again.

4.4.4 Generated Documentation

The secondary output of the generator is a reference manual. See Appendix E for sample. This documentation is in HTML format. The generator of this documentation has the same input as the generator of wrap-class source (it is parts of the same program), thus the documentation is always consistent with generated wrap-class source code.

Documentation for each class contains list of wrapped and unwrapped methods. Two versions of method signature are given. One is original ANSII *C++* signature and second one is the new managed signature accessible from .NET environment. The unwrapped methods are highlighted and contain the explanation why they are not wrapped. Each class and method also contains brief description if it is present in parsed *C++* header file as method or class comment.

4.5 Results

Resulting interfacing layer for VTK has approximately the same functionality as the official distribution of VTK interface for Java. The slowdown caused by this layer is approximately 30%. Comparable applications in Java have slowdown approximately 85%[6]. This interface is well tested by students of KIV/GSVD course. And as they said it is "Easy to use".

4.5.1 Verification and Time Measuring

[6] These numbers will be explained later.

The best way how to verify any product is to use it. Students of KIV/GSVD course 2003 did the main verification process. Testing assignments (classes – programmer) and results are given in Appendix D. With their invaluable help there were fixed number of bugs in the interface.

Each testing task consists of three comparable VTK programs in three different environments (*C#*, Java, Win32 – native *C++*). These VTK programs are as close as possible. It is important for time comparison among entire environments. Each task contains overall time counter, which can be printed to given text file. This text file is processed and overall time results are published in the Appendix D as a table and graph.

Most of resulting slowdown is approximately 20% (in comparison with direct use of VTK in unmanaged *C++*). It is expected slowdown caused by data conversion and versification. But, there are some interesting exceptions on both sides.

The extreme **slowdown** (~1000%) is caused by the relatively low VTK core time consumption in comparison with the wrapping layer time consumption. Note that only short total running time tasks have this extreme slowdown. On the other hand the long running time tasks have the slowdown close to 0%.

Even a **speedup** of *C#* programs can occurs. At the first point of view it seems really strange but the answer is probably in memory management. The large number of small objects in memory can be de-allocated faster by garbage collector (all in one time) than one by one with *C++* delete operator.

Part 5 Conclusion

Here presented implementation of the VTK interface allows straightforward and safety programming with VTK in .NET environment. It has approximately the same functionality as the official VTK interface for Java and significantly better speedup.

There are two ways how to use results of this work. The first one is direct use of precompiled assembly, which is ready to use for efficient application development. The disadvantages of this way of use can be the inability to use some method, which are not wrapped and the fact that for each distribution (version) of the VTK has to be a special compilation of the interface. The second way is to use only a necessary wrap-class source code, modify it and use it. It provides very high flexibility and can suit probably any needs.

Appendix A

Users Manual

This appendix discuss the usage of compiled version of *vtkDotNetWrap.dll* assembly "as is". This assembly is compiled for just one version of the VTK and cannot be used with another one. The recompilation of the assembly should be sufficient in case of minor changes (bug-fixes). The regeneration of all wrap-class source code could be necessary in case of VTK major changes. In this case, please refer the [Herakles] website for actual version of the interface.

Installation and System Requirements

Correct function of the *vtkDotNetWrap* requires .NET Framework and VTK of particular version[7]. Core installation of VTK with DLLs for *C++* is sufficient. The VTK DLLs have to be in executable directory. The *vtkDotNetWrap.dll* assembly has to be in local directory of current project because of the interfacing assembly cannot be installed in GAC (Global Assembly Cash).

C# Programming

The use of the interface with *C#* in Visual Studio .NET is straightforward. Please follow these instructions:

- Make sure the environmental variable PATH contains the directory where all VTK-DLLs are installed. (e.g. *C:\Program Files\vtk42\bin*)
- Make sure the version of *vtkDotNetWrap.dll* suits the installed version of VTK.
- Make a local copy of the *vtkDotNetWrap.dll* for current *C#* project.
- Set the project reference to the local *vtkDotNetWrap.dll* file.
- Each *C#* source code, which uses VTK should contain the `using vtkDotNetWrap;` directive.

At the beginning of using presented interface is probably the best first step try to compile and to run some examples distributed with the interface. The version of particular *vtkDotNetWrap* assembly and targeting version VTK is part of manifest and can be accessible by ildasm.exe program embedded with Visual Studio .NET.

Reference Manual

The *vtkDotNetWrap* interface consists of managed proxy classes (wrap-classes) for each VTK class. These proxy classes do not match exactly their VTK originals. These differences (and limitations) are documented at the reference manual, which is automatically generated with the *vtkDotNetWrap* source code.

Sample of method documentation follows:

[7] The particular number of required VTK version is part of assembly metadata and can be accessible by *ildasm.exe*. This number is in particular reference manual too.

GetClassName

OK

C++ Signature	`virtual const char *GetClassName () const;`
MC++ Signature	`virtual System::String * GetClassName();`

Comment:

Return the class name as a string. This method is defined in all subclasses of vtkObjectBase with the vtkTypeRevisionMacro found in vtkSetGet.h.

The most important information is in the two-row table. The first row contains signature of original VTK method. The second row contains a .NET-adapted version of the method in *C++ managed extension*. Note the *System.String* object replaced the zero terminated string (*char **).

If a method is unwrapped than it is marked in the reference with the reason why it is unwrapped. Sample follows:

PrintSelf

unwrapped problematicReason: arg0 arg1

C++ Signature	`void PrintSelf (ostream &os, vtkIndent indent);`
MC++ Signature	`void PrintSelf( arg0, vtkIndent arg1);`

Comment:

None

This method is unwrapped because of the first and the second argument is not successfully converted between managed and unmanaged environment, thus this method cannot be used in .NET environment. On the other hand, this method is part of the *MC++* source code of the particular wrap-class and is commented out.

Appendix B

Programmers Manual

Purpose of this text is to provide better orientation at the whole source code of this work (especially in wrap-class generator). Programmer should be able to use the wrap-class generator with here presented information.

All projects are situated in one solution of Visual Studio .NET (7.0) named *vtkDotNet*. Full list of projects follows:

- CppTests
- GSVDResultParser
- IFFGenerator
- ParseBatGenerator
- TestSet01
- TestSet02
- vtkDotDWrap
- vtkDWGenerator

CppTests

CppTests is a simple *C++* Win32 project that contains a set of testing examples in ANSII *C++*. It serves as a reference testing program for comparisons of *C#* functionality and effectiveness. This project actually contains only one *C++* source code at one time because each example contains main entry function. The switching between examples has to be done by insertion of required source code and exclusion all the others.

GSVDResultParser

This simple *C#* program processes a text file that is an output from testing programs created within the frame of KIV/GSVD - 2003 course. The input for this parser is a text file of following type:

```
# Delaunay triangulation in 2D
task  0103
input_file  data.pts
environment Win32
overall_time      375.000000

# Delaunay triangulation in 2D
task  0103
input_file  data.pts
environment Win32
overall_time      391.000000
…
```

Note both these records are output of the same program executed more than once for more accurate time measuring.

The output of this program is simple text file that contains average overall time of each task in three environments (.NET, Java, Win32). It is tab-separated text file that can be easily processed in MS Excel (for example). The first column contains code number of a task. The next three columns contain the average time in *C#*, Java and unmanaged *C++* respectively. Sample of one line follows:

```
103    427,3333    459    385,6667
```

Structure of this 250-lines program is very simple and understandable directly from source code.

IFFGenarator

This project contains the first part of the wrap-class generator, the parser. Files *vtkParse.tab.c* and *vtkParse.h* are assumed from the VTK source code distribution without any modification. File *vtkIFFPrint.c* implements the *vtkParseOutput* function that prints the given internal data structures of the parser to the intermediate text file.

The structure (and usage) of this program is also assumed. The entry main function is in the *vtkParse.tab.c*. Usage of the resulting program is by command line only. Explanation of command line parameters follows:

IFFGenerator.exe <input_h_file > <hint_file > <output_txt_file >

Example:

IFFGenerator.exe .\vtk\vtk3DS.h hints.\output\vtk3DS.txt

Due to intermediate file format (IF), there has to be a special char conversion. Each item in IF has to be at one line. Unfortunately, comments can contains the new line and career return chars, thus they can take more than one line. The implemented solution is straightforward, each \n and \r char is replaced by *$* char. It is replaced back in IF reader (part of generator).

ParseBatGenerator

This is very simple program that generates one special batch file. This *.bat* file contains call of the *IFFGenerator* for each *.h* file in the given directory. The first parameter is the input directory where all *.h* files are situated. The second parameter is the output directory where all resulting IF *.txt* files are going to be stored. Example of call follows:

ParseBatGenerator.exe .\vtk .\output

This command creates *parse.bat* file in current directory. A two sample lines of the batch file follows:

IFFGenerator.exe .\vtk\vtk3DS.h hints .\output\vtk3DS.txt

IFFGenerator.exe .\vtk\vtk3DSImporter.h hints .\output\vtk3DSImporter.txt

Etc.

TestSet01 TestSet02

These two *C#* projects contain a set of testing programs. They were used for testing and time measuring of here presented interfacing layer.

vtkDotNetDWrap[8]

This project contains the *vtkDotNetWrap* assembly itself. The *vtkDWGenerator* program generates following list of files automatically.

- *vtkDotNetWrap.cpp*
- *wgHelp.cpp*
- *wgHelp.h*
- all files in *include* directory

Only *AssemblyInfo.cpp*, *vtkDotNetWrap.cpp* and *wgHelp.cpp* files are marked as a part of this project. All the other necessary files (*.cpp* files of all wrap-classes) are included in *vtkDotNetWrap.cpp* file by *#include* directive. At the first point of view it does not seem to be very clean. But this approach is much faster for compilation than inclusion and compilation of each wrap-class *.cpp* file independently.

Compilation of the *vtkDotNetWrap.dll* assembly requires reference to *.lib* files distributed with VTK for *C++* developing.

vtkDWGenerator[8]

This project contains the *C#* code of the wrap-class generator. It is the most complicated program in this work.

The entry point is the *MainClass* and its *main()* method. Its content serves as a kind of "user interface"[9]. It contains an absolute path names, thus it has had to be modified before utilization of the generator. The essential part of the main function source code follows.

```
wgIFReader reader = new wgIFReader("...\\IFFiles\\output\\", "*.txt");
wgCppWriter writer = new wgCppWriter();
wgHtmlWriter htmlWriter = new wgHtmlWriter();

writer.AddClasses(reader.Classes);
htmlWriter.AddClasses(reader.Classes);

writer.Write("...\\vtkDotNet\\vtkDotNetDWrap\\", false);
htmlWriter.Write("...\\reference\\", false);
wgType.PrintUnknownTypes("UnknownCodes.txt");
```

The whole program consists of 10 classes. Important classes (used in main function) are marked as a "high-level".

- *MainClass* – already mentioned entry point
- *wgArgument* – represents particular method argument. Its constructor reads all necessary information from the given IF-file. It also contains methods for writing the *C++* code.
- *wgClass* – represents a VTK class. Its constructor reads all necessary information from the given IF-file. It also contains methods for writing the *C++* code.

---

[8] The "D" in *vtkDotNetDWrap* and *vtkDWGenerator* means "double". Because the *vtkDWGenerator* is already prepared for implementation of the double wrapping but it is not finished.

[9] This approach has been chosen because of developing effectiveness of the wrap-class generator. As it was mentioned before, the generating process is not for user but for developer of the interface.

- *wgCppWriter* – a high-level class that writes all given classes (*wgClass*) as *MC++* wrap-classes.
- *wgExclusiveList* – derived from *Hashtable*. It overloads the *Add()* method and provides exclusive addition of particular item (same item only once). The second functionality is to count the number of addition of the same object.
- *wgFunction* – represents a method of a class. Its constructor reads all necessary information from the given IF-file. It also contains methods for writing the *C++* code.
- *wgHtmlWriter* – similar to *wgCppWriter*. It writes the reference manual for the given classes instead of *C++* source code.
- *wgIFReader* – a high-level class that reads all IF-files from the given directory with the given extension. The class list of the reader can be passed to the *wgHtmlWriter* and/or *wgCppWriter*.
- *wgType* – represents a data type, which is recorded (one line) in the *wgTypes.txt* file.
- *wgTypeHint* – responsible for implicit and explicit overloading of data type codes.

For additional information please refer the commented source code.

Appendix C

The List of Conversion Macros

This appendix contains complete list of conversion macros with data type codes. The wrap-class generator processes this list.

Table of primitive data type conversion.

Code	ACppDecl	MCppDecl	IsProblematic
-1	<unknown> <var>	<unknown> <var>	Yes
1	float <var>	float <var>	No
2	void <var>	void <var>	No
3	char <var>	char <var>	No
4	int <var>	int <var>	No
5	short <var>	short <var>	No
6	long <var>	long <var>	No
7	double <var>	double <var>	No
8	code_8 <var>	code_8 <var>	Yes
9	::<class> <var>	<class> <var>	Yes
13	unsigned char <var>	unsigned char <var>	No
14	unsigned int <var>	unsigned int <var>	No
15	unsigned short <var>	unsigned short <var>	No
16	unsigned long <var>	unsigned long <var>	No

Table of reference type conversion

(The first part)

Code	ACppDecl	MCppDecl	ToACpp	ToMCpp
101	float & <var>	System::Single & <var>	<tmp>	<var>
103	char & <var>	System::SByte & <var>	<tmp>	<var>
104	int & <var>	System::Int32 & <var>	<tmp>	<var>
105	short & <var>	System::Int16 & <var>	<tmp>	<var>
106	long & <var>	System::Int32 & <var>	<tmp>	<var>
107	double & <var>	System::Double & <var>	<tmp>	<var>
113	unsigned char & <var>	System::Byte & <var>	<tmp>	<var>
114	unsigned int & <var>	System::UInt32 & <var>	<tmp>	<var>
115	unsigned short & <var>	System::UInt16 & <var>	<tmp>	<var>
116	unsigned long & <var>	System::UInt32 & <var>	<tmp>	<var>

(The second part)

Code	TmpDecl	TmpRet	IsProblematic
101	float <tmp> = <var>	<var> = <tmp>	No
103	char <tmp> = <var>	<var> = <tmp>	No
104	int <tmp> = <var>	<var> = <tmp>	No
105	short <tmp> = <var>	<var> = <tmp>	No
106	long <tmp> = <var>	<var> = <tmp>	No
107	double <tmp> = <var>	<var> = <tmp>	No
113	unsigned char <tmp> = <var>	<var> = <tmp>	No
114	unsigned int <tmp> = <var>	<var> = <tmp>	No
115	unsigned short <tmp> = <var>	<var> = <tmp>	No
116	unsigned long <tmp> = <var>	<var> = <tmp>	No

Table of pointer data type conversion

(The first part)

Code	ACppDecl	MCppDecl
301	float * <var>	float <var> __gc []
302	void * <var>	??? <var>
303	char * <var>	System::String * <var>
304	int * <var>	int <var> __gc []
305	short * <var>	short <var> __gc []
306	long * <var>	long <var> __gc []
307	double * <var>	double <var> __gc []
308	code_8 * <var>	code_8 * <var>
309	::<class> * <var>	<class> * <var>
313	unsigned char * <var>	unsigned char <var> __gc []
314	unsigned int * <var>	unsigned int <var> __gc []
315	unsigned short * <var>	unsigned short <var> __gc []
316	unsigned long * <var>	unsigned long <var> __gc []

(The second part)

Code	ToACpp	ToMCpp
301	(float *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_float(<var>, <field>)
302	<var>	<var>
303	wgStr2Char(<var>)	new System::String(<var>)
304	(int *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_int(<var>, <field>)
305	(short *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_short(<var>, <field>)
306	(long *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_long(<var>, <field>)
307	(double *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_double(<var>, <field>)
309	((<var> == NULL) ? NULL : <var>->w)	((<var> == NULL) ? NULL : new <class>(<var>))
313	(unsigned char *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_uchar(<var>, <field>)
314	(unsigned int *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_uint(<var>, <field>)
315	(unsigned short *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_ushort(<var>, <field>)
316	(unsigned long *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_ulong(<var>, <field>)

(The third part)[10]

Code	TmpDecl	TmpRet	IsProblematic
301	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No
302			Yes
303			No
304	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No
305	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No
306	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No
307	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No
308			Yes
309			No
313	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No
314	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No
315	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No
316	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No

Table of constant simple data type conversion

Code	ACppDecl	MCppDecl	IsProblematic
1001	const float <var>	const float <var>	No
1003	const char <var>	const char <var>	No

[10] The full-qualified name is replaced by locally qualified name. (only *GCHandle* instead of *System::Runtime::InteropServices::GCHandle*). This replacement is valid for all the other tables.

1004	const int <var>	const int <var>	No
1005	const short <var>	const short <var>	No
1006	const long <var>	const long <var>	No
1007	const double <var>	const double <var>	No
1008	const code_8 <var>	const code_8 <var>	Yes
1009	const <class> <var>	const <class> <var>	Yes
1013	const unsigned char <var>	const unsigned char <var>	No
1014	const unsigned int <var>	const unsigned int <var>	No
1015	const unsigned short <var>	const unsigned short <var>	No
1016	const unsigned long <var>	const unsigned long <var>	No

Table of constant pointers conversion

(The first part)

Code	ACppDecl	MCppDecl
1301	const float * <var>	float <var> __gc []
1303	const char * <var>	System::String * <var>
1304	const int * <var>	int <var> __gc []
1305	const short * <var>	short <var> __gc []
1306	const long * <var>	long <var> __gc []
1307	const double * <var>	double <var> __gc []
1308	const code_8 * <var>	code_8 * <var>
1309	const <class> * <var>	<class> * <var>
1313	const unsigned char * <var>	unsigned char <var> __gc []
1314	const unsigned int * <var>	unsigned int <var> __gc []
1315	const unsigned short * <var>	unsigned short <var> __gc []
1316	const unsigned long * <var>	unsigned long <var> __gc []

(The second part)

Code	ToACpp	ToMCpp
1301	(float *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_float(<var>, <field>)
1303	wgStr2Char(<var>)	new System::String(<var>)
1304	(int *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_int(<var>, <field>)
1305	(short *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_short(<var>, <field>)
1306	(long *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_long(<var>, <field>)
1307	(double *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_double(<var>, <field>)
1309	((<var> == NULL) ? NULL : <var>->w)	new <class>(<var>)
1313	(unsigned char *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_uchar(<var>, <field>)
1314	(unsigned int *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_uint(<var>, <field>)
1315	(unsigned short *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_ushort(<var>, <field>)
1316	(unsigned long *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_ulong(<var>, <field>)

(The third part)

Code	TmpDecl	TmpRet	IsProb
1301	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No
1303			No
1304	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No
1305	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No
1306	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No
1307	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No
1308			Yes
1309			No
1313	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No
1314	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No
1315	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No
1316	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()	No

Additional simple data type conversion

Code	ACppDecl	MCppDecl	IsProblematic
2001	float <var>	float <var>	No
2002	void <var>	void <var>	No
2003	char <var>	char <var>	No
2004	int <var>	int <var>	No
2005	short <var>	short <var>	No
2006	long <var>	long <var>	No
2007	double <var>	double <var>	No

Table of additional pointer data types conversion

(The first part)

Code	ACppDecl	MCppDecl	IsProblematic
2301	float * <var>	float <var> __gc []	No
2303	char * <var>	System::String * <var>	No
2304	int * <var>	int <var> __gc []	No
2305	short * <var>	short <var> __gc []	No
2306	long * <var>	long <var> __gc []	No
2307	double * <var>	double <var> __gc []	No
2309	::<class> * <var>	<class> * <var>	No
2703	char * <var>	System::String * <var>	No
3303	char * <var>	System::String * <var>	No

(The second part)

Code	ToACpp	ToMCpp
2301	(float *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_float(<var>, <field>)
2303	wgStr2Char(<var>)	new System::String(<var>)
2304	(int *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_int(<var>, <field>)
2305	(short *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_short(<var>, <field>)
2306	(long *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_long(<var>, <field>)
2307	(double *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_double(<var>, <field>)
2309	((<var> == NULL) ? NULL : <var>->w)	((<var> == NULL) ? NULL : new <class>(<var>))
2703	wgStr2Char(<var>)	new System::String(<var>)
3303	wgStr2Char(<var>)	new System::String(<var>)

(The third part)

Code	TmpDecl	TmpRet
2301	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()
2303		
2304	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()
2305	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()
2306	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()
2307	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()

Table of added data type conversion

(The first part)

Code	ACppDecl	MCppDecl	IsProblematic
5000	void (* <var>)(void *)	wgICallback * <var>	No
5001	vtkIdType <var>	long <var>	No
5002	vtkIdType * <var>	long <var> __gc []	No
5003	FILE * <var>	FileStream * <var>	Yes
5004	vtkIdType & <var>	System::Int32 & <var>	No
5005	void (* <var>)(void *)	wgICallback * <var>	Yes

(The second part)

Code	ToACpp	ToMCpp
5000	wgPackToVoid(<var>)	
5001	(vtkIdType) <var>	(long) <var>
5002	(vtkIdType *)(<tmp>.AddrOfPinnedObject().ToPointer())	wgPtr2Field_long((long *)<var>, <field>)
5003		
5004	<tmp>	<var>
5005	wgPackToVoid(<var>)	

(The third part)

Code	TmpDecl	TmpRet
5000		
5001		
5002	GCHandle <tmp> = GCHandle::Alloc(<var>, GCHandleType::Pinned)	<tmp>.Free()
5003		
5004	vtkIdType <tmp> = (vtkIdType) <var>	<var> = (long) <tmp>
5005		

Appendix D

KIV/GSVD Course Testing

This appendix contains materials related with the GSVD course testing. Resulting programs are part of embedded CD-ROM (in Czech only).

Following Table 0.2 contains overall average times. Each time is result after three times run of each program. Computer configuration is given in Table 0.1.

OS Name	Microsoft Windows 2000 Professional
Version	5.0.2195 Service Pack 3 Build 2195
OS Manufacturer	Microsoft Corporation
System Name	NYMPH7
System Manufacturer	Dell Computer Corporation
System Model	Precision WorkStation 410 MT
System Type	X86-based PC
Processor	x86 Family 6 Model 7 Stepping 3 GenuineIntel ~447 Mhz
BIOS Version	Phoenix ROM BIOS PLUS Version 1.10 A13
Windows Directory	C:\WINNT
System Directory	C:\WINNT\System32
User Name	NYMPH7\Administrator
Time Zone	Central Europe Daylight Time
Total Physical Memory	1 048 108 KB
Available Physical Memory	844 576 KB
Total Virtual Memory	2 202 796 KB
Available Virtual Memory	1 893 712 KB
Page File Space	1 154 688 KB
Page File	C:\pagefile.sys

Table 0.1 – Testing computer configuration

Task Code	C# [ms]	Java [ms]	Win32 [ms]	C#/Win32	Java/Win32
103	427	459	386	1,108	1,190
106	234	360	170	1,378	2,116
107	116	190	78	1,485	2,419
108	281	275	146	1,927	1,886
109	338	396	328	1,032	1,208
110	427	542	437	0,977	1,239
111	84	463	31	2,699	14,946
113	1568	1739	1552	1,010	1,120
114	57	59	57	1,005	1,043
115	416	516	349	1,194	1,479
203	99	99	63	1,577	1,571
205	110	266	63	1,746	4,222
206	172	270	60	2,867	4,506
208	177	208	52	3,382	3,981
209	233	332	172	1,359	1,936
210	48833	48916	48588	1,005	1,007
213	250	328	203	1,230	1,616
214	1714	1708	2052	0,835	0,832
215	38516	60201	37702	1,022	1,597
303	922	937	844	1,092	1,111
305	922	937	844	1,092	1,110

306	511	614	443	1,152	1,386
308	473	452	344	1,376	1,314
309	93	171	52	1,800	3,310
310	604	677	536	1,127	1,263
311	1224	901	1156	1,059	0,779
313	656	760	610	1,075	1,246
314	1745	1729	1646	1,060	1,051
315	25797	25937	25760	1,001	1,007
403	661	729	610	1,085	1,196
405	656	734	610	1,075	1,203
406	1855	1969	1428	1,298	1,379
407	13286	13401	13233	1,004	1,013
408	504	505	380	1,326	1,329
409	121800	106593	121457	1,003	0,878
410	1042	1120	1011	1,031	1,108
411	3640	3765	3578	1,017	1,052
413	969	1067	922	1,051	1,158
414	86	97	87	0,994	1,110
415	143077	148172	142031	1,007	1,043
503	218	219	188	1,161	1,165
505	218	219	188	1,160	1,165
507	250	312	203	1,234	1,539
508	610	614	542	1,126	1,134
509	260	390	229	1,137	1,706
510	172	271	141	1,225	1,927
511	698	880	651	1,072	1,353
513	177	407	120	1,478	3,389
514	552	572	411	1,342	1,391
515	2906	3026	2900	1,002	1,043
603	151	193	99	1,523	1,940
605	141	203	94	1,500	2,160
607	744	838	697	1,067	1,201
608	3292	3296	3255	1,011	1,012
609	16421	16582	16588	0,990	1,000
610	552	672	485	1,140	1,387
613	2287	3109	2245	1,019	1,385
614	17417	17312	17162	1,015	1,009
615	149000	150833	148582	1,003	1,015
702	1370	1568	1422	0,963	1,102
703	5333	16000	0	#DIV/0!	#DIV/0!
705	0	16000	0	#DIV/0!	#DIV/0!
706	449	578	380	1,181	1,521
707	93	213	52	1,788	4,096
708	922	932	797	1,157	1,169
709	510	640	463	1,103	1,383
710	906	1047	735	1,234	1,425
711	901	1021	875	1,030	1,167
714	589	620	484	1,216	1,280
715	536	1125	479	1,120	2,350
807	685	987	616	1,111	1,602
809	1384	1479	1286	1,076	1,150
810	458	651	380	1,206	1,713
811	1828	1922	1792	1,020	1,073
813	192	255	141	1,364	1,811
814	271	250	115	2,354	2,174
815	1952	5672	505	3,866	11,232

Table 0.2 – Average overall time measuring

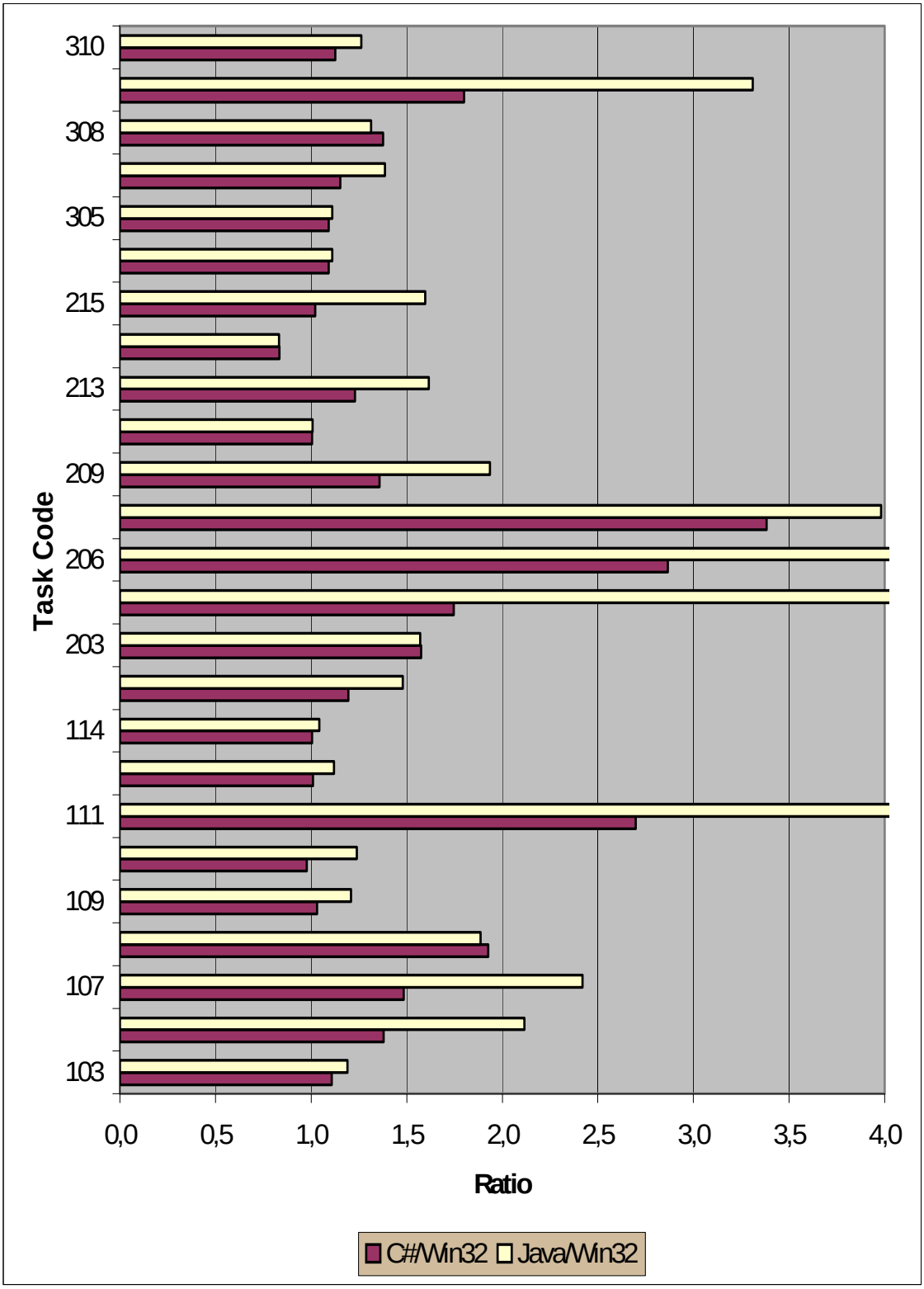

Figure 0.1 – Time measuring graph – the first part

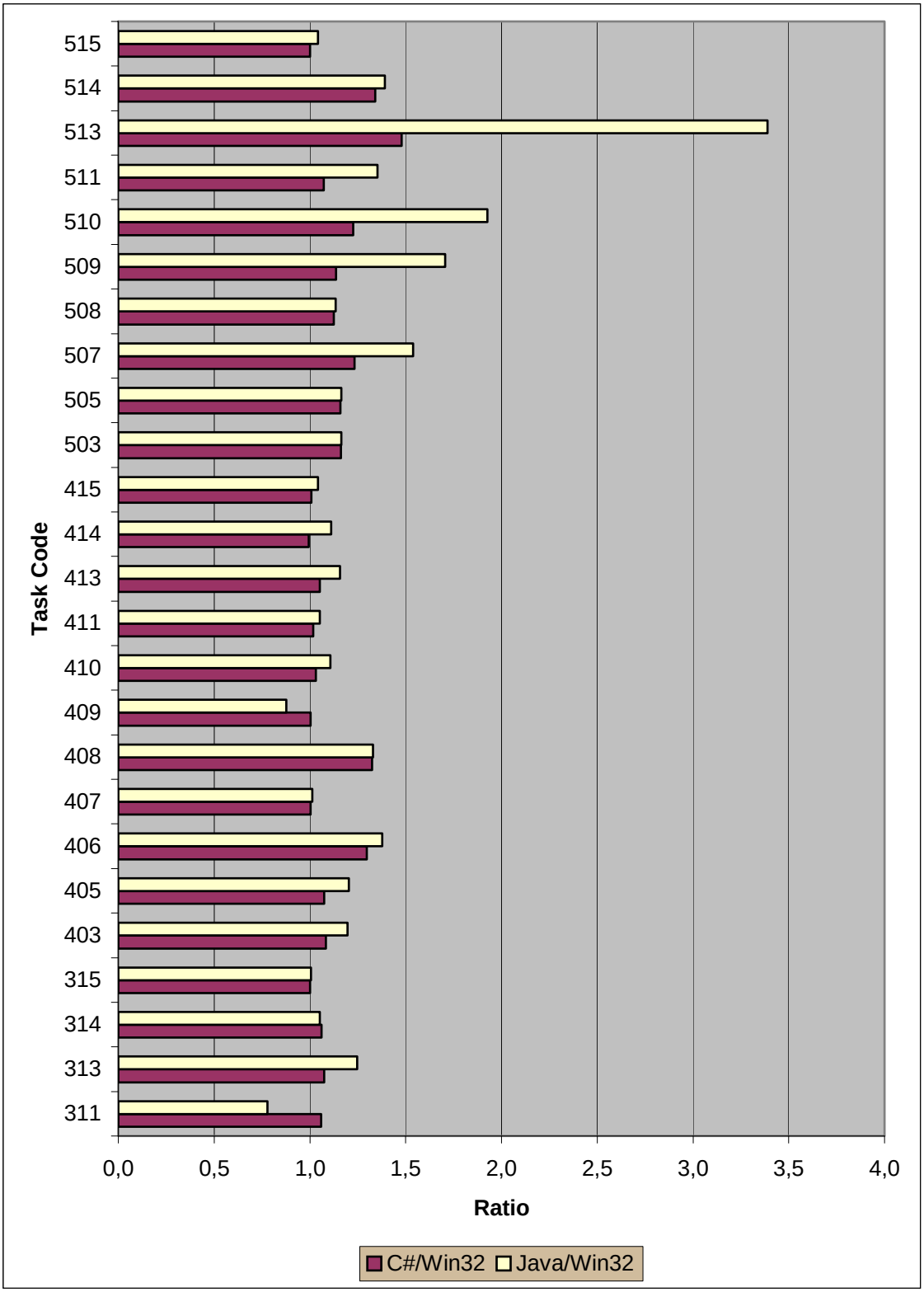

Figure 0.2 – Time measuring graph – the second part

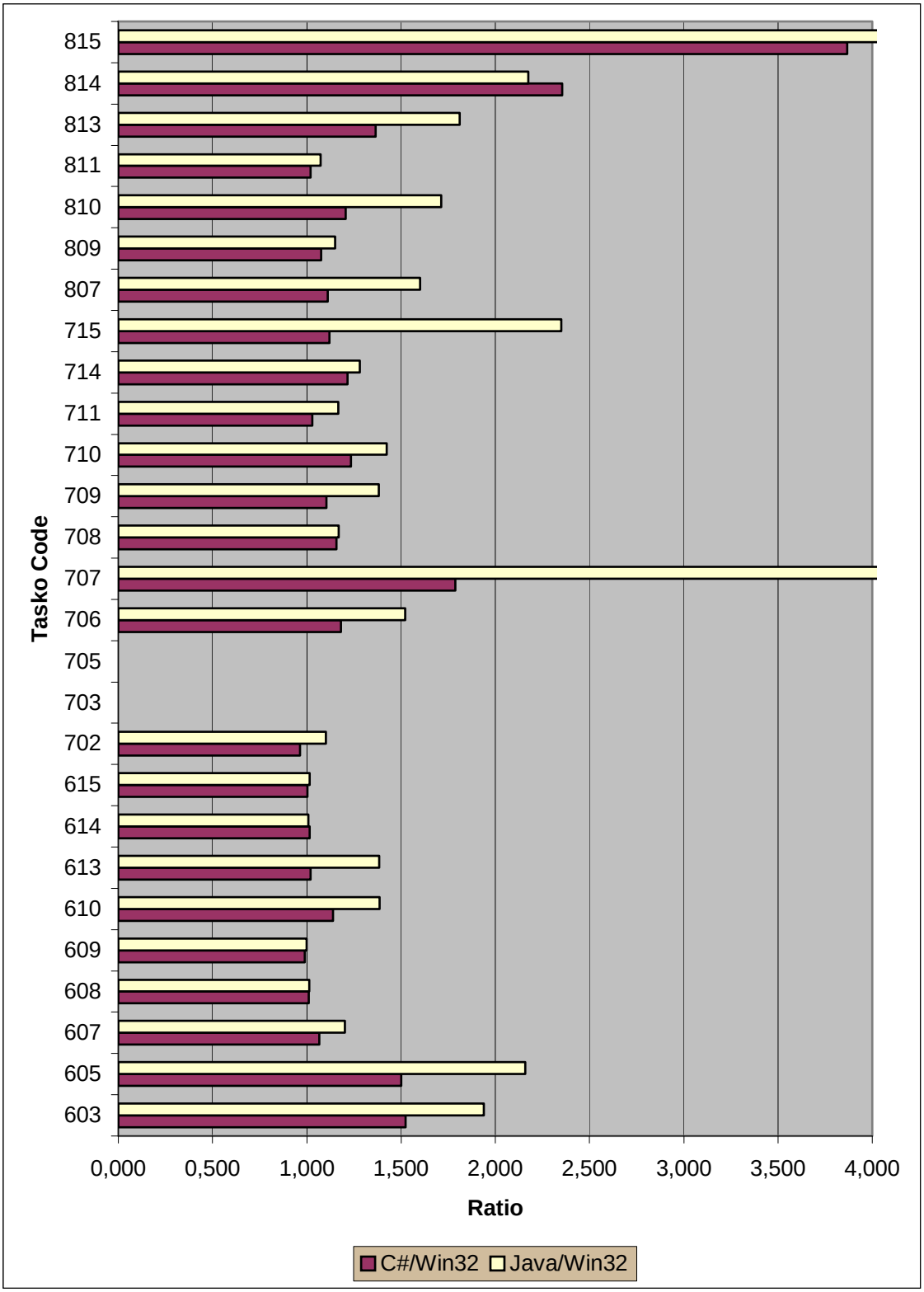

Figure 0.3 – Time measuring graph – the third part

Submissions for students, which class they shall test:

GSVD - zadání 1. úlohy

ID	Příjmení	Zadané třídy	Popis
01	ANTON	vtkArrowSource vtkAxes	
02	BENDA	vtkCursor3D vtkOutlineSource	
03	DO	vtkDelaunay2D	
04	DOUBEK	vtkSphereSource vtkConeSource vtkCubeSource vtkCylinderSource vtkDiskSource	
05	JANDA	vtkEarthSource	Planetarni system, Slunce, Merkur, Venuse, Zeme+mesic, Mars
06	KOŽINA	vtkLineSource vtkPlaneSource vtkPointSource	
07	LANG	vtkSuperquadricSource vtkTexturedSphereSource	
08	MATOUŠEK	vtkTextSource vtkVectorText	
09	MIKŠÍČEK	vtkBYUReader	
10	NOVOTNÝ	vtkSTLReader	
11	PARUS	vtkPLOT3DReader	
13	SMLSAL	vtkDelaunay3D	
14	VAIS	vtkCamera	Prulet kratkou umele vytvorenou scenou
15	VÁŠA	vtkLight	

GSVD - zadání 2. úlohy

ID	Příjmení	Zadané třídy	Popis
01	ANTON	vtkRenderWindow	demonstrace vlastnosti tridy na jednoduche scene
02	BENDA	vtkInteractorStyle	odzkouseni stavajicich + navrh vlastniho ovladani
03	DO	vtkActor	
04	DOUBEK	vtkFollower vtkLODActor vtkMesaActor vtkOpenGLActor	 - demonstrace rozdilu
05	JANDA	vtkTexture	
06	KOŽINA	vtkTextureMapToPlane vtkTextureMapToSphere vtkTextureMapToCylinder	
07	LANG	vtkPicker (vtkCellPicker, vtkPointPicker)	
08	MATOUŠEK	vtkAssembly	hierarchicka struktura - rameno robota
09	MIKŠÍČEK	vtkCoordinate	test jednotlivych souradnicovych systemu, prevody mezi nimi
10	NOVOTNÝ	vtkProp3D, vtkLODProp3D	demonstrace transformaci v 3D prostoru, kratka animovana sekvence
11	PARUS	vtkActor2D vtkMapper2D vtkProperty2D	
12	RÁDLOVÁ	vtkTextMapper	
13	SMLSAL	vtkAxisActor2D vtkCaptionActor2D	
14	VAIS	vtkXYPlotActor vtkLegendBoxActor	
15	VÁŠA	vvtkTransformFilter vtkTransformPolyDataFilter	 - rozdily, demonstrace

GSVD - zadání 3. úlohy

ID	Příjmení	Zadané třídy	Popis
01	ANTON	vtkTextMapper	
02	BENDA	vtkRendererSource	
03	DO	vtkCubeAxesActor2D vtkParallelCoordinatesActor	
04	DOUBEK	vtkScalarBarActor vtkScaledTextActor	
05	JANDA	vtkLabeledDataMapper	
06	KOŽINA	vtkLookupTable	namapovat napr. teplotu na barevne spektrum
07	LANG	vtkContourFilter	
08	MATOUŠEK	vtkGlyph3D	nacist napr. STL soubor a zobrazit napr. normaly trojuhelniku jako sipky
09	MIKŠÍČEK	vtkTubeFilter	vstupni soubor bude obsahovat ridici body lomenne cary a ukolem bude zobrazit potrubi o danem prumeru (parametry budou ulozeny v konfiguracnim souboru)
10	NOVOTNÝ	vtkRuledSurfaceFilter, vtkStreamLine	
11	PARUS	vtkCutter, vtkImplicitFunction	orezavani teles implicitni funkci, test na nekolika vstupnich STL souborech, nekolik jednoduchych implicitnich funkci
12	RÁDLOVÁ	----	
13	SMLSAL	vtkMergeFilter	ukazka spojovani datovych vstupu, geometrie, normaly, skalarni hodnoty, atd.
14	VAIS	vtkProbeFilter	rovinne rezy volumetrickymi daty
15	VÁŠA	vtkExtractGeometry	spoluprace s vtkImplicitFunction

GSVD - zadání 4. úlohy

ID	Příjmení	Zadané třídy	Popis
01	ANTON	----	
02	BENDA	vtkArrowSource vtkAxes	
03	DO		vtkClipPolyData - spoluprace s vtkImplicitFunction, vstup STL soubor
04	DOUBEK	----	
05	JANDA	vtkGeometryFilter	
06	KOŽINA		vtkPolyDataNormals - nacist STL soubor, spocitat normaly a odzkouset i ostatni vlastnosti tridy
07	LANG		vtkDecimatePro - strucny a jasny popis algoritmu + demonstracni program na STL vstupnim souboru
08	MATOUŠEK		vtkDecimate - strucny a jasny popis algoritmu + demonstracni program na STL vstupnim souboru
09	MIKŠÍČEK		vtkQuadricDecimation - strucny a jasny popis algoritmu + demonstracni program na STL vstupnim souboru
10	NOVOTNÝ		vtkQuadricClustering - strucny a jasny popis algoritmu + demonstracni program na STL vstupnim souboru
11	PARUS		vtkSmoothPolyDataFilter - strucny a jasny popis algoritmu + demonstracni program na STL vstupnim souboru
12	RÁDLOVÁ	----	
13	SMLSAL		vtkWindowedSincPolyDataFilter - strucny a jasny popis algoritmu + demonstracni program na STL vstupnim souboru
14	VAIS		vtkRenderWindow - demonstrace vlastnosti tridy na jednoduche scene
15	VÁŠA	vtkStructuredGridGeometryFilter	

GSVD - zadání 5. úlohy

ID	Příjmení	Zadané třídy	Popis
01	ANTON	----	
02	BENDA	vtkImageData vtkImageViewer	
03	DO	vtkExtractVOI	zobrazovani detailu vol. dat
04	DOUBEK	----	
05	JANDA	vtkWarpScalar	
06	KOŽINA	vtkImageViewer vtkImageActor	
07	LANG	vtkImageCanvasSource2D	
08	MATOUŠEK	vtkImageEllipsoidSource vtkImageGaussianSource vtkImageGaussianSmooth	
09	MIKŠÍČEK	vtkImageNoiseSource vtkImageSinusoidSource vtkImageGridSource	
10	NOVOTNÝ	vtkImageGradient vtkImageGradientMagnitude	
11	PARUS	vtkImageAccumulate	nacist barevny obrazek + zobrazit histogramy vsech barevnych slozek RGB
12	RÁDLOVÁ	----	
13	SMLSAL	vtkImageLogic	pouziti cfg. souboru pro nastaveni typu logicke operace a vstupnich obrazu
14	VAIS	vtkImageReslice vtkImageResample	
15	VÁŠA	vtkImageFlip vtkImageClip vtkImageChangeInformation	

GSVD - zadání 6. úlohy

ID	Příjmení	Zadané třídy	Popis
01	ANTON	----	
02	BENDA	vtkColorTransferFunction vtkPiecewiseFunction	- demonstrace pouziti (barva, pruhlednost)
03	DO	vtkSphereSource vtkConeSource vtkCubeSource vtkCylinderSource vtkDiskSource	
05	JANDA	vtkVolumeMapper (i podtridy)	- orezavani: demostrace pouziti, k cemu se hodi, atd.
06	KOŽINA	vtkVolumeMapper (i podtridy)	- 3D texturovani: demostrace pouziti, k cemu se hodi, atd.
07	LANG	vtkVolumeMapper (i podtridy)	- normala, gradient: demostrace pouziti (popsat vtkEncodedGradientEstimator)
08	MATOUŠEK	vtkVolumeRayCastIsosurfaceFunction vtkVolumeRayCastCompositeFunction	
09	MIKŠÍČEK	vtkFollower vtkLODActor vtkMesaActor vtkOpenGLActor	- demonstrace rozdilu
10	NOVOTNÝ	vtkScalarBarActor vtkScaledTextActor	
11	PARUS	vtkLinearExtrusionFilter vtkRotationalExtrusionFilter	- demostrace modelalovacich technik, vytvorit a zobrazit nekolik modelu
13	SMLSAL	vtkSurfaceReconstructionFilter	- vstup mnozina bodu v souboru, format domluvit se cvicicimi
14	VAIS	vtkVoxelModeller	- namodelovat par objektu + demonstrovat moznosti
15	VÁŠA	vtkImplicitModeller	- namodelovat par objektu + demonstrovat moznosti

GSVD - zadání 7. úlohy

ID	Příjmení	Zadané třídy	Popis
01	ANTON	----	
02	BENDA	vtkCurvatures	konzultace s kolegou Hlavatym
03	DO	vtkArcPlotter vtkBandedPolyDataContourFilter	
04	DOUBEK	----	
05	JANDA	vtkDepthSortPolyData	demonstrovat na pouziti algoritmu malire pro reseni viditelnosti
06	KOŽINA	vtkExtractPolyDataGeometry vtkFeatureEdges	
07	LANG	vtkTriangleFilter vtkGLUTesselatorTriangleFilter	otestovat obe tridy na vstupnim netrivialnim polygonu nacitanem ze vstupniho souboru ve formatu: pocet vrcholu + jejich vycet v textovem souboru (navrh konzultovat s cvicicimi)
08	MATOUŠEK	vtkHull	demonstrovat na vstupnim STL souboru, vytvorit konvexni obalku s nastavitelnym poctem rovin v config souboru
09	MIKŠÍČEK	vtkPolyDataConnectivityFilter	
10	NOVOTNÝ	vtkReverseSense vtkRibbonFilter	
11	PARUS	vtkShrinkPolyData vtkSelectPolyData	
12	RÁDLOVÁ	----	
13	SMLSAL	vtkSpline vtkSplineFilter	
14	VAIS	vtkStripper	stripifikace ze vstupnich trojuhlenikovy dat, konzultace vstupnich souboru s kolegou Vaneckem (pet@kiv.zcu.cz), ktery se danou problematikou zabyva
15	VÁŠA	vtkVoxelContoursToSurfaceFilter	rekonstrukce povrchu z paralelnich rezu, konzultace vstupnich souboru s kolegou Svitakem (rsvitak@kiv.zcu.cz), ktery se danou problematikou zabyva

GSVD - zadání 8. úlohy

ID	Příjmení	Zadané třídy	Popis
01	ANTON	----	
02	BENDA	vtkEdgePoints vtkDividingCubes vtkRecursiveDividingCubes	- principy + porovnani
03	DO	vtkImageToPolyDataFilter	
04	DOUBEK	----	
05	JANDA	vtkMarchingCubes	- test na implicitni funkci + zakladni princip algoritmu
06	KOŽINA	vtkProgrammableGlyphFilter vtkCellCenters	- vytvorit vlastni glyph - nacist objemova data a zobrazit glyfy ve stredu voxelu odpovidajici jejich hodnote
07	LANG	vtkHedgeHog vtkHyperStreamline	
08	MATOUŠEK	vtkMarchingContourFilter	- otestovat pro 2D data
09	MIKŠÍČEK	vtkMaskPoints vtkOutlineCornerFilter vtkOutlineFilter	
10	NOVOTNÝ	vtkSelectVisiblePoints vtkThresholdPoints vtkBrownianPoints	
11	PARUS	vtkTensorGlyph	
12	RÁDLOVÁ	----	
13	SMLSAL	vtkArrayCalculator	
14	VAIS	vtkProgrammableFilter	- vytvorit prahovy filter, prahem bude smer vektoru a odchylka, na vystupu budou jen ty bunky, ktere splnuji definovane kriterium
15	VÁŠA	vtkCellDerivatives	

Appendix E

Sample of Generated Documentation

This appendix contains a sample of generated documentation. This kind of documentation is generated for each wrap-class by the wrap-class generator.

vtkSphereSource

Parent Class :

vtkPolyDataSource

Name comment :

create a polygonal sphere centered at the origin

Description:

vtkSphereSource creates a sphere (represented by polygons) of specified radius centered at the origin. The resolution (polygonal discretization) in both the latitude (phi) and longitude (theta) directions can be specified. It also is possible to create partial spheres by specifying maximum phi and theta angles. By default, the surface tessellation of the sphere uses triangles; however you can set LatLongTessellation to produce a tessellation using quadrilaterals.

SeeAlso:

None

Methods:

OK GetClassName
OK IsA
OK NewInstance
OK SafeDownCast
unwrapped PrintSelf
OK New
OK SetRadius
OK GetRadiusMinValue
OK GetRadiusMaxValue
OK GetRadius
OK SetCenter
OK SetCenter
OK GetCenter
OK SetThetaResolution
OK GetThetaResolutionMinValue
OK GetThetaResolutionMaxValue
OK GetThetaResolution
OK SetPhiResolution
OK GetPhiResolutionMinValue
OK GetPhiResolutionMaxValue
OK GetPhiResolution
OK SetStartTheta
OK GetStartThetaMinValue
OK GetStartThetaMaxValue

OK GetStartTheta
OK SetEndTheta
OK GetEndThetaMinValue
OK GetEndThetaMaxValue
OK GetEndTheta
OK SetStartPhi
OK GetStartPhiMinValue
OK GetStartPhiMaxValue
OK GetStartPhi
OK SetEndPhi
OK GetEndPhiMinValue
OK GetEndPhiMaxValue
OK GetEndPhi
OK SetLatLongTessellation
OK GetLatLongTessellation
OK LatLongTessellationOn
OK LatLongTessellationOff
unwrapped vtkSphereSource
unwrapped vtkSphereSource
unwrapped Execute
unwrapped ExecuteInformation
unwrapped vtkSphereSource
unwrapped None

GetClassName

OK

C++ Signature	`const char *GetClassName ();`
MC++ Signature	`System::String * GetClassName();`

Comment:

None

IsA

OK

C++ Signature	`int IsA (const char *name);`
MC++ Signature	`int IsA(System::String * arg0);`

Comment:

None

NewInstance

OK

C++ Signature	`vtkSphereSource *NewInstance ();`
MC++ Signature	`vtkSphereSource * NewInstance();`

Comment:

None

SafeDownCast

OK

C++ Signature	`vtkSphereSource *SafeDownCast (vtkObject* o);`
MC++ Signature	`vtkSphereSource * SafeDownCast(vtkObject * arg0);`

Comment:

None

PrintSelf

unwrapped problematicReason: arg0 arg1

C++ Signature	`void PrintSelf (ostream &os, vtkIndent indent);`
MC++ Signature	`void PrintSelf( arg0, vtkIndent arg1);`

Comment:

None

New

OK

C++ Signature	`static vtkSphereSource *New ();`
MC++ Signature	`static vtkSphereSource * New();`

Comment:

Construct sphere with radius=0.5 and default resolution 8 in both Phi and Theta directions. Theta ranges from (0,360) and phi (0,180) degrees.

SetRadius

OK

C++ Signature	`void SetRadius (float );`
MC++ Signature	`void SetRadius(float arg0);`

Comment:

Set radius of sphere. Default is .5.

List of Figures Tables and Source Codes

Index

References

[Hana03a] 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.

[Hana03b] Hanák, I. (2003). *Graphical Interface OpenGL for .C#.* M. A. thesis, University of West Bohemia in Pilsen, Pilsen.

[Kacm01] Kačmář, D.: *Programujeme .NET aplikace.* Computer Press, Praha, 2001.

[Race98] Racek, S. Kvoch, M.: *Třídy a Objekty v C++*. Kopp, České Budějovice, 1998.

[Schr98] Schreder, W., Martin, K., Lorensen, B.: *The Visualization Toolkit.* Prentice Hall, New Jersey, 1998.

[Schr01] Schreder, W., Avila, L., Martin, K., Hoffman, W., Law, C.: *The VTK User's Guide.* Prentice Hall, New Jersey, 2001.

[ECMA02a] *Common Language Infrastructure (CLI).* Standard ECMA-335, December 2002.

[ECMA02b] *C# Language Specification.* Standard ECMA-334, December 2002.

[MSDN] *Microsoft Development Network*. http://msdn.microsoft.com/library/.

[Kitware] *Home pages of VTK.* http://public.kitware.com/vtk/.

[Herakles] *Home pages of CCGDV and vtkDotNetWrap.* http://herakles.zcu.cz.