University of West Bohemia in Pilsen

Faculty of Applied Sciences

Department of Computer Science and Engineering

DIPLOMA THESIS

Pilsen, 2003 Ivo Hanák

University of West Bohemia in Pilsen
Faculty of Applied Sciences
Department of Computer Science and Engineering

Diploma Thesis

Graphical Interface OpenGL for C# in scope of ROTOR project

Pilsen, 2003 Ivo Hanák

Abstract

Graphical Interface OpenGL for C# in scope of ROTOR project

Few years back Microsoft released first release version of runtime environment called .NET. However, the default library that is contained in .NET lacks classes for high performance graphical output. OpenGL is worldwide known standard for hardware supported high-performance graphical output. The goal of this works is to connect these two together and provide comfortable environment for developer including improved debugging capabilities. Documentation contains brief introduction into .NET runtime environment, OpenGL, and languages used for both development and testing purposes. It describes problems that need to be solved in order to allow cooperation of OpenGL and .NET. It also contains list of possible solutions and explains which solution and why it was chosen. Finally, measurement of performance in comparison with existing solution is provided. It also contains set of examples that uses results of this work. Even thought this work does not provide complete OpenGL interface (i.e., all version and all add-ons such as GL Extensions), it describes possible approach, which forms a solution that allows use of OpenGL in .NET environment

Contents

Statement

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

Pilsen, May 20 2003, ……..

Ivo Hanák

1 Introduction

In year 2002, Microsoft has released the first non-beta version of runtime environment called .NET. It allows an application to run no matter of current underlying platform. This environment is aimed on distributed applications, i.e., applications where application parts are not present on machines on which they shall run. Therefore, it provides facilities that simplify this task.

However, .NET is not just environment that allows application to run. It also defines a framework for languages that can be used by any language whose compiler aims .NET. Such compiler then compiles source code to the special assembler that is called Common Immediate Language (CIL).

Language framework, CIL, execution system, and memory management, as well as other parts of .NET are **standardized** as ECMA standard (ECMA-335 CLI standard). This allow developers around the world to create their own implementations that follow these standards, e.g., Shared Source CLI (code name: “ROTOR”; see [SSCLI]), mono project (see [MONO]), etc.

Essential part of the .NET Framework is a **library** (.NET Framework Library) of classes that provide wide range of functionality. However, it has no advanced support for graphical output. The only output that is supported is based on common capabilities of the majority of windowing systems, i.e., it allows simple 2D output. The usual disadvantage of such output is its performance, i.e., it is rather slow. This fact is a **motivation** of this work.

Currently there are two major common graphical interfaces for **high performance graphical output** available: DirectX and OpenGL. Because DirectX is platform dependent, this work uses OpenGL in order to provides high performance graphical output that allows projecting of both 2D and 3D objects onto a screen. Even thought OpenGL is platform dependent similar to DirectX, opposite of that it is implemented on many platforms. Therefore, creating of class that allows .NET application to call OpenGL would lead to the platform independent high performance graphical output. This is **the goal** of this work.

Currently there exist projects that aim on introducing of OpenGL to .NET, i.e., porting of OpenGL. However, these projects are usually based on a mechanism that is provided by .NET and they lead to the exact copy of the original OpenGL interface. Therefore this work shall not just create another port of OpenGL but it shall also answer a question whether there is another reasonable approach that would allow to improve comfort of programming and simplify debugging of application, i.e., improved **programming safety**.

The result shall be a specification of framework rather then complete port of OpenGL, i.e., it shall not support both all version and all GL Extensions. This work shall summarize benefits and drawbacks of this approach in comparison to both original OpenGL and the most important port of OpenGL called **CsGL**. Comparing of results shall be performed from the viewpoint of both performance and comfort.

1.1 The Document

This document introduces basics of .NET and graphical interfaces to a reader. It describes difficulties, solutions, and results. It does not contain detailed description of .NET, OpenGL, and programming languages that are used for implementation and/or testing purposes. The reader shall be familiar with C++ language syntax.

First, there is an **introduction to .NET** that includes description of **major features** because knowledge of .NET features equals to knowledge of C# language capabilities due to C# was created especially for .NET in order to make all of its features available to the user. Then the document contains brief introduction of languages that are used for both implementation and testing purposes: C++ Managed extension (MC++) and C#. This introduction covers basics of syntax (for both MC++ and C#) and summary of major differences between MC++ and C++.

Next, the reader is introduced to **graphic output** possibilities available in .NET and basics of OpenGL. This chapter does not contain detailed description of both .NET Framework Library and OpenGL due to it is matter of specifications and reference manuals. After that, **the goal** of this work is described in high detail. The chapter also contains explanation of reasons for including particular sub-goals into the whole goal. Then, difficulties and possible approaches are introduced including description of their advantages and disadvantages.

In order to illustrate possible approaches, **existing solutions** are introduced to the reader. This includes solution description of previously mentioned difficulties. However, only the most important ports of OpenGL are mentioned due to creation of simple port is not difficult and therefore they may exist many implementations based on similar mechanism.

Important part of this document is description of **solution** itself and **results**. In this part, reasons for particular approach are explained and results are commented. It also contains brief introduction into a tool that was designed to simplify creation of a OpenGL port. Next, a verification of the interface functionality is introduced including description of particular test source code structure.

The document is closed with a conclusion that **summarizes** results and compares major features of this work to both CsGL and original OpenGL. It also contains recommendations for future work. All references including namespace, class, and class members reference is contained in appendices. These appendices also include examples of source code that uses this work and summary information for developers who might extend results of this work. Appendices do not contain reference for OpenGL functions, constants, and enumeration data types.

1.2 Text Formatting Conventions

- sans-serif for menu items or GUI control names (identifiers)
- monospace for source code, identifiers, language constructions
- **bold** for emphasis, important expression, or term
- *italics* for mathematical expressions and terms from figures
- sans-serif enclosed by brackets (e.g., <key>) for particular key of key combination on keyboard

2 .NET Framework

.NET Framework is an object oriented environment that provides facilities allowing execution of platform independent code. The environment is often called as **managed environment**, because it handles some tasks automatically (e.g. memory management). There are two major parts of .NET: Common Language Infrastructure (CLI) and .NET Framework class library, which is a large set of reusable classes that provide support for networking, multithreading, user interface, etc.

CLI (Common Language Infrastructure; often referenced as CLR: Common Language Runtime) is a foundation of the .NET Framework. It handles automatic memory management, thread execution, code verification, code execution and security. The CLI is aimed on distributed code, which can be called either locally or remotely. Due to that, a security system is integral part of the runtime. It provides support for security permission that may by based on a source of the code, i.e., code executed remotely may have larger restriction then code executed locally. The support for signed code is also provided by CLI.

Code that is generated for CLI and runs completely under managed environment is called **managed code** (managed application). Similar to the managed code, there exist **managed data**. CLI provides automatic allocation and deallocation of such data. Deallocation of managed data is performed by a process called **garbage collection** (see section 2.5).

Type system that is used by CLI is specified by Common Type System (CTS) and the code that conforms CTS is strict typed. Generated code also contains description of itself (i.e., it is self-describing). This feature is the simplification of code deployment and makes easy to use third part components.

Specification of CLI and its parts is standardized by ISO/ECMA and therefore the compilers for various programming languages that targets CLI may be implemented. It makes possible to reuse the code that was created in programming language A in programming language B without any restriction. This leads to faster development because in order to use .NET there is no need to learn a new language: the developer may use language that he is used to. There is already support (compilers) for various languages, e.g., C#, C++ in a form of Managed Extension C++, Visual Basic, J# that is Java for .NET, Eiffel, and others. C# is a programming language that was created especially for .NET in order to make the most of features of CTS available to the user.

As it was mentioned, managed code is platform independent and uses Common Intermediate Language (CIL). CIL is aimed on just-in-time (JIT) compilation, i.e. code is compiled just before its execution. However, because of CLI construction it is possible to mix it with interpreted code and native code (non-managed). CLI contains support for

interoperability with native (binary; platform-dependent; non-managed) libraries with support for COM technology.

.NET Framework provides support for web application in form of ASP.NET. This is completely object-oriented approach and the output of ASP.NET application is plain HTML with only few bits of Javascript (client-side script).

Currently .NET Framework is running on Windows, Linux (Unix), and Mac platform. Linux platform support is provided by mono project by Ximian (see [MONO]). This project is based on ECMA standards and it is possible to run managed applications compiled under Windows on Linux platform. Mono project also provides compilers for C# and Visual Basic .NET. All these three platforms are supported by SSCLI (see [SSCLI] project). However, this project currently lacks few parts such as GUI, web services.

2.1 Common Type System

Common Type System (CTS) is the basic part of CLI and it provides support for data handling. Because of it provides support not only for object oriented programming (OOP) but also for procedural and functional programming there exist two types of entities: objects (see Figure 2.2) and values (see Figure 2.1).

Values are stored in form of bit patterns and they can be used as representation for basic or simple data types (e.g. integers, floats, etc.). They are defined by their type, which describes not only storage and meaning of those bit patterns but also operations that are allowed.

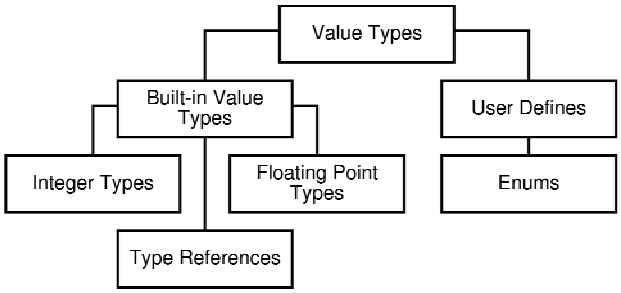


Figure 2.1: Value data types of CTS

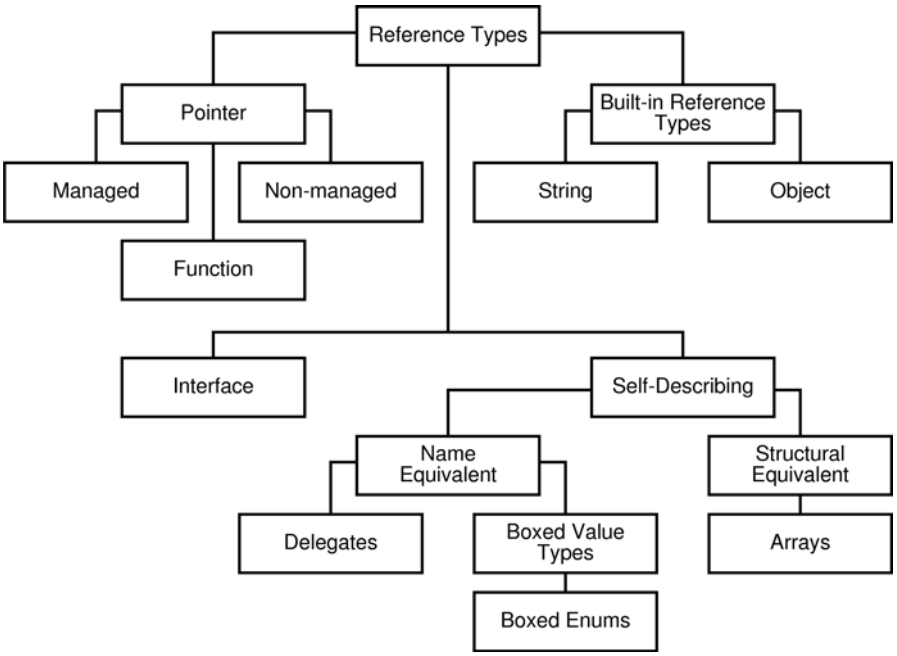


Figure 2.2: Reference data types of CTS

Objects, on the other hand, are more then values. Their type is explicitly stored in its representation and it contains slot, which can be occupied by other entities (values or objects). Object has an identity, which distinguishes it from others. This identity remains the same even through the contents of the slot changes.

Type is often used as description of data representation. In CTS, a type means not only representation of data but also behavior or possible operations. Two types are considered the same only if they both have compatible representation and behavior. This makes possible to substitute instance of the base type with instance of inherited type. Due to the way the CTS is designed, it is possible to handle different approaches of several OOP languages. CTS does not support “typeless programming”. It is not possible to call non-static function (object member) without knowledge of the object's type.

2.1.1 Value and Reference Types

As it was mentioned, a value is described by type, which specifies not only its representation but also allowed operations. Currently there exist two kinds of types: value types and reference type. Value types describe values that are represented as sequence of bits and reference types describe values that are represented as location. There are four kinds of types: object, interface, pointer and build-in type.

Object type is self-describing value. In some cases (such as abstract classes), it describes itself only partially. **Interface type** is similar to an object, but it describes itself only partially. **Pointer type** is reference to a location (machine address) in memory. **Built-in type** is integral part of CTS (i.e. basic types) with direct support from executional system (VES; Virtual Executional System. Every value has only one **exact type** that provides full description of operations available for the value and description of its representation. This is also why is not possible to use interface type as exact type of the value because of it provides description only for its functionality without description of its representation.

Every type can be used to describe representation of the value. This relation is transitive only for object type (a reference type). On the other hand, this relation is **not** a transitive relation in the case of value types. It is not possible to extract the exact type from an actual value of the Value. For example built-in integer type fully describes its value (i.e. it is an exact type) but it is not possible to determine the exact type just from its Value (i.e. sequence of bits).

2.1.2 Boxing and Unboxing

It is possible to handle value type as a reference type. Every value type has a special reference type defined that is called **boxed type**. Every value type also contains an operation called **box**, which creates the corresponding instance of boxed type. This instance contains bit copy of the original value and provides operation called **unbox** that is reversal to the **box** operation. This makes possible to handle value types the same way as reference type. However, use of above described operations leads to a slowdown and boxed values are not CLS-compliant (Common Language Specification; see section 2.2).

2.1.3 Identity and Equality

One of the basic binary operation defined for values (for both reference and value types) is **identity** and **equality**. Both of these operations return Boolean as a result and both are mathematical equivalence operators, i.e. they are reflexive, symmetric and transitive. It is also true, that if two values are identical, they are equal in the same time.

To explain these two terms let us consider two variables of reference type. They are both identical only if they refer to the same object, it also implies their equality. Identity is implemented on `System.Object` via `ReferenceEquals` method.

On the other hand to have these objects equal, a reference to the same object is not needed, just the value of the object shall be the same. If either of operands is a boxed value, the equality could be computed by unboxing boxed operands and comparing of resulting values. Equality is implemented on `System.Object` via `Equals` method. For floating points values, when comparing two NaNs, this method returns **true**, which differs from a standard (see [IEC]). In order to satisfy the standard an override of the method is required.

2.1.4 Location and Type Conversion

Every value is stored in a **location**. Location is typed and it contains only single value at the time. This type specifies the usage of value loaded from the location. It means that the allowed operations are the only ones specified by the location type even though the stored type has larger capabilities.

When assigning to a location the assigned value have to be **assignment compatible**. In the most of the cases, it is possible to perform this check during the compilation time. How ever in few cases, involving objects and interfaces the check cannot be performed during compilation due to the compiler is not able to determine the type, which would be assigned. Type is always assignment compatible with itself.

In some cases is possible to assign a type that is not assignment compatible by performing conversion. It is possible to convert via two operations: coercing and casting.

In the case of **coercing** the conversion is performed by creating of value of destination type with an equal meaning to the original value. There are two kinds of coercing, widening, and narrowing. **Widening** is an operation, which does not loose any information, and it is often provided as implicit conversion. Compared to that, **narrowing** usually leads to information loss and explicit conversion is usually needed in order to perform narrowing. Both of these operations lead to modification of both value and type and do not preserve identity of objects.

Casting, on the other hand, uses the fact that value can be of more than one type. Performing casting operation means to cast value to one of its types (e.g. casting an object to implemented interface). This operation does not modify type and value and preserves identity of objects.

2.1.5 Compound Types

In a previous text, there was a description of types and their properties. Nevertheless, a creation of structured types is allowed. These are called as **compound types** and they are composed of array types or fields, whose can differ by their type.

Fields are named sub-values. If these sub-values are accessible by an indexing expression, they are called as **array elements**. **Array type** is a type composed of array elements of same type. Both fields and array elements are typed and their type cannot be changed.

2.1.6 Methods

Type can specify available operations. These operations are called as methods and they had a signature (see subsection 2.1.11) that defines types for its argument and type of the return value, if any. CTS supports static method, which is associated with the type itself, instance method, which is associated with an instance of such type and virtual methods.

The difference between virtual and instance method is based on their implementation. Virtual methods are allowed to change its implementation in inherited classes by overriding. Decision about the implementation, which will be invoked, is made during run-time.

For virtual methods, it is possible to invoke them same way as in the case of instance method by specifying a class and method within class. If there is no instance of the class, the CTS allows object passed as **this** to be **null**. In other cases, **this** refers to instance of the type or inherited type. In the case of virtual method, this is often used to invoke implementation of the method inside the parent class. However, a virtual method can be invoked by a mechanism called as **virtual call**, which chooses implementation of the method based on the dynamically detected type of the instance. For detailed descriptions see [CLI-I].

2.1.7 Enumeration Types

CTS provides support for **enumeration type** (enum). Enum is a value type, which is assignment compatible with underlying type. This type is defined as an integer and characteristics of this type are defined by CLS-rules.

As it was mentioned this type is a value type, however there are some restrictions, e.g., no type members are allowed with exception of single instance field that defines underlying type. Also all enums have to inherit from `System.Enum` class and they cannot be further inherited (i.e., they are sealed).

As defined by CLS-rules, there are two kinds of enums. First kind is similar to the enum in C++; i.e. its values are similar to integer constants. Second kind consists of single bit flags rather then integer constants; i.e. values of this enum can be combined in order to create group of flags.

2.1.8 Names

Each entity has to have a name. This name is used when referencing to such entity. Entity of type system shall have exactly one name. Comparison performed on names is so called **code-point comparison**, i.e. it is case-sensitive and locale-independent. The permitted form of such names is described by CLS rules (see section 2.2). It uses Unicode and therefore languages, whose characters are not contained in ASCII (7bit), are allowed. Form of the identifier is described at Figure 2.3.

```
      <identifier> ::= <identifier_start>(<identifier_start>|<identifier_extend>)*
<identifier_start> ::= [{Lu}{Ll}{Lt}{Lm}{Lo}{Nl}]
<identifie_extend> ::= [{Mn}{Mc}{Nd}{Pc}{Cf}]
```

Figure 2.3: Valid identifier name

The first character of the identifier can be an uppercase letter, lowercase letter, title-case letter, modifier letter, other letter, or letter number. As subsequent characters, any of those mentioned above plus non-spacing marks, spacing combining marks, decimal numbers, connector punctuator, and formatting codes are allowed. Before storing or comparing formatting codes shall be filtered out. For details see [Dav99].

2.1.9 Scopes

As it was mentioned in previous text, every entity has to have a name. However, these names are usually not unique. CTS allows same name to be used for multiple entities as long as these entities differs in their kind (e.g. methods, fields, etc.). Such use is allowed within a **scope**, where scope is a group of names.

Each entity can be fully identified by a **qualified name**. The qualified name consists of both scope and name of the entity. When referencing to a member of compound type, scope contains also name of enclosing type. When these entities are types, situation is similar except that types are grouped into **assembly scopes**. Only top-level types (i.e. not nested types) are scoped by the assembly (see section 2.7).

2.1.10 Visibility and Accessibility

When referencing to an entity, it has to be visible. An access is then granted only if referenced type is visible, referenced member of the type is accessible and all security demands are satisfied. Both visibility and accessibility are relations between referent and referenced entity. Visibility is property only of type names and there are three categories:

- **Exported** type is a type that can be visible outside the enclosing assembly. However, such visibility depends on assembly configuration, which determines if the type is really exported.
- **Nonexported** type is a type that is hidden from the outside world.
- **Nested** type visibility depends on visibility of enclosing type.

Accessibility, on the other hand, is a property of all entities and depends on visibility of referenced type and a scope of referent. While inheriting, there is a possibility to modify accessibility of inherited virtual member methods by access "widening". This means that the inherited virtual method shall either have the same accessibility or permit more access.

- It is the same rule as in other languages (e.g., C++). Different approach of accessibility modification would lead to a possibility to gain access to hidden virtual methods by casting to a base class. To prevent overriding of virtual method in inherited class a use of **final** (see subsection 2.1.15) is recommended. CTS provides support for seven categories of accessibility:
- **Compiler-Controlled** members are accessible only through use of definition not reference and they are accessible only within a single compilation unit under control of compiler.
- **Private** members are accessible only for referents from within an implementation of the same type that defines the referenced members.
- **Family** members are accessible for referents from an implementation of the same exact type or type derived from such type. This access needs runtime check in some cases.
- **Assembly** members are accessible for referents in the same assembly that contains implementation of such type.
- **Family-and-Assembly** members are accessible for those referents which fulfills both family and assembly access requirements.
- **Family-or-Assembly** members are accessible for those referents which fulfills either family or assembly access requirements.
- **Public** members are accessible for all referents.

Security permissions and demands also influence access. There exist two types of demands: inheritance and reference demand. It is not allowed to inherit them and it is possible to attach only one kind of the demand to a single item. Attaching it to the item attaches the same security demand to all nested types and type members unless another demand of same kind is attached to the item. This approach to security is called **Declarative Security** (see section 2.8).

Inheritance demand influences overriding of the method and inheritance of the type. Type that wishes to inherit from the type or method that wishes to override the virtual method, have to meet required security permission. **Reference demand** influences references to the type. A referent needs have to have all required security permission in order to refer such type.

Nested types have a full access to the members of enclosing type and family access to members of the type from which it inherits. In order to access field, array element or function that uses nested type (as a parameter or return value) such nested type shall be both visible and accessible to the referent.

2.1.11 Contract and Signatures

Contracts are shared assumption on set of signatures between implementers and users and are names. They define what shall be implemented and provide a possibility for verification by checking implementation of enforceable parts of contracts. There are five kinds of contracts: class, interface, method, property and event contract.

First is a **class contract**, which is specified by a class definition. It provides specification of value representation of given class type and it also specifies what shall be implemented. Class supports class contract. When a class supports a class contract of another class then it means that the class inherits from that class type.

Interface contract is similar to class contract with exception that it does not support class contract because of interface is not an exact type and contains only operation specifications.

Next is **method contract**, it is a description of implementation of named operation and specification of contracts of its parameters and return value, if any. It is always part of contract of another type.

Property contract is a specification of set of method contracts that shall be implemented by a type that supports given property contract.

The last is **event contract**, which is a specification of operations that manages given event. This includes three standard methods, one for registering an event listener, one for revoking of listener’s registration and one for invoking the event.

Signature adds constants that are limitation on use or list of allowed operation on values and locations. Every value and location has a signature and while assigning, his compatibility (including compatibility of constraints) is required.

Properties of constraints are defined by CLS-rules (see section 2.2). All types in signature have to be CLS-compliant and whenever member is visible itself, all of types in member's signature have to be visible too. There are five kinds of signature: type, location, parameter and method signature.

Type signature is limitation and constraint on usage of the type. Type signature of the value is determined neither by the value itself nor by the type, but is determined by knowledge of the location signature where the value is stored.

Location signature is similar to type signature. It adds further restriction by **location constraints**. Currently there are two location constraints: init-only and literal constraint. **Init-only constraint** ensures that value of the location is set before the first use (i.e. during initialization) and the value cannot be changed afterwards. This constraint can be applied only to instance or static fields of compound types. **Literal constraint** means that all references to such field are replaced by field's value at compilation time. It is applicable only to static field of compound types and only build-in type values are permitted.

Local signature is similar to location signature and can be applied only to local variables. It adds **byref** constraint, which means that either content of location is managed pointer (if possible) or content of location is handled by copy-in/copy-out mechanism (in the rest of cases).

Together with local signature there exists special local signature: **typed reference**, e.g., a local variable, which that states to be typed reference, contains managed pointer and runtime representation of the type that is possible to store to the location. It provides dynamical type info and it cannot be combined with other constraints. It is also limited only to built-in types and can be used only for parameters and local variables. Boxing is also not allowed as well as use as type of a field. Typed reference is **not** CLS-compliant.

Parameter signature is similar to local signature. It provides information on how are parameters passed during method invocation. **Method signature** is composed of calling convention, list of parameter signatures (if any) and type signature of a return value. This includes additional constraint **varargs constraint**, that states all following perimeters to be optional only. Varargs constraint is **not** CLS-compliant.

2.1.12 Type Safety and Verification

Type specifies contracts. If the code implements enforceable parts of the contract (the names signatures), the code can be considered **typesafe**. Typesafe code stores values that are described only by a type signature in location. This signature shall be assignment compatible with location signature. In addition, operations that are not defined by exact type are forbidden for typesafe code. Only visible and accessible locations are accessed. In the case of typesafe code, exact type of value cannot be changed.

Verification is a mechanical process, which verifies an implementation to be typesafe or not. It may fail for typesafe implementation but never success for implementation that is **not** typesafe. In general, process of the verification cannot be performed in finite time with no errors. Code marked as unsafe cannot be verified and therefore it has higher security needs (see section 2.8). For more detailed description of verification see [CLI-I, CLI-II].

2.1.13 Type Definition

Type definers construct new type from an existing type. In the case of **implicit types**, a type is defined when it is used because of implicit type signature provides complete description of the type. Implicit type does not need user-supplied name.

Opposite of that, all other types needs to be defined explicitly and they need user-supplied names. Explicit definers are interface definitions and class definition that can be used to create either object types or value type including boxed version. It is good to note that not all types defined by a class definition are objects (e.g. value types).

Array types are defined by specifying element type, number of dimensions (rank) and lower and upper bounds for each dimension. These bounds shall be integer and their lower bound for all dimensions shall be zeros (defined by CLS-rules). The signature may specify information on lower bound, upper bound, or both bounds at compile time. Zero-dimensional arrays as well as location signatures for array elements are not allowed.

All array elements shall be laid out in row-major order. The actual storage allocated for each element may be platform-specific (i.e. a different padding of elements may appear on different platforms).

Arrays are objects and are inherited from type `System.Array` (abstract class), which represents all arrays regardless of element type, ranks, or bounds. Arrays are created automatically when they are required. Operation defined by CTS on arrays provides array

allocation, indexing, value read/write operations, computation of an element address (a managed pointer), and querying for either rank, bounds, or total number of stored array elements.

Unamanaged pointer types (also known as "pointer type") is defined by specifying a location signature for the location to which pointer references. Because of signature of pointer types includes location signature, no further definition of the pointer type is needed. Pointer types are reference types but their values are not objects. CTS provide basic typesafe operations on pointer types: loading a value from a location specified by the pointer and storing the value to such location. Pointer arithmetic is also provided by CTS. Pointer types are not CLS-compliant.

Delegates are object-oriented and typesafe version of function pointers. Each delegate contains method named `Invoke`, which invokes a method associated with the delegate. This associated method may be both static method or instance method. All delegates are inherited from `System.Delegate` and they may optionally contain other static or instance methods.

Interface type is an incomplete description of a value, i.e. it is not an exact type, class type or object type. It contains set of methods, locations and other contracts. Only static fields are allowed and only virtual or static methods are allowed. However only static methods are possible to implement inside interface because of they are associated with the interface itself rather then with any value of the type. Interface implementing static fields or methods are not CLS-compliant. Events and property contracts are both allowed. The use of such contracts follows the same rules as those of methods.

Only object types are allowed to support an interface. Support of the interface can be declared but existence of all implementations that particular interface requires does not imply support of that interface. The support of an interface means to provide complete implementation of all methods, locations, and other contracts defined by the interface. CLS-compliant interfaces shall not require implementation of non-CLS compliant methods.

All interface members shall be fully visible and accessible (i.e. public). No security permission shall be attached to any of them. This is because of interface only defines what shall be implemented rather then the implementation itself.

Class type is an exact type due to it provides complete specification of both value representation and representation of all contracts that are supported by the class type. Contracts, which can be supported by the class type, are class, interface, method, property, and event contract. Support for a class contract is synonymous with the object type inheritance (see subsection 2.1.15). With an exception of **abstract object type**, a class type provides definition as well as implementation for all contracts supported by the class type.

Not all classes require class definition, e.g., array types. Explicit class definition defines either an object type or value type. Explicit class type definition also contains definition of class type name. It implicitly assigns the class type to a scope (i.e., assembly), defines class contracts of the same name, defines representation/operations of class members, supplies implementation for supported contracts, and declares visibility for the type. Such visibility can be either public or assembly.

Class type definition may also specify initialization method of the type and type members. A type can be marked as **BeforeFieldInit**, i.e., initialization method can be executed anytime before first access to any static field defined for that member. When the type is not marked as **BeforeFieldInit** then type initialization method execution is triggered by first access to any (static or instance) field or (static, instate, or virtual) method. Such execution

of the initialization method does not trigger any initialization method of the base type. If this is a requested behavior for particular language, special (hidden) static field and code in class constructor, which touches that field of the parent class, shall be implemented.

Object type describes the physical structure of the instance and all allowed operations. Object type is set when the object is created and all instances of the same object type have same structure and allows same set of operations. Object type definition is class type definition and therefore it specifies assembly as its scope.

Object type can be marked either **abstract** or **concrete**. Abstract object types may provide definition for method contracts without implementation. It is similar to an interface with possibility to use contracts, which are in the case of interface forbidden. Some of the methods may have implementation in abstract object type. However, it is not allowed to create instances of abstract object type, but it is possible to inherit a type and then create instance of the inherited type. Such type shall provide implementation of all abstract methods, i.e., it is concrete object type.

Every object may support zero or more interface contracts, where the support means implementation of required set of methods (required by the interface contract). If two interfaces have a method with identical method contract, then they share its implementation. Class (value type or interface), that implements non-CLS-compliant interface, is not CLS-compliant.

Value types are types defined by class definition. However, they are not object types. When defining a value type, both unboxed (value type) and boxed type are defined. Boxed type supports interface contracts and have a base type (opposite of value type, which does not). Base type of a boxed type shall not have any fields. Value types do not requite a constructor (see subsection 2.1.14) to be defined and called in order to create instance of the value type. Instead of that, a special code shall be provided to initialize type members to zero or null.

When a non-static method of value type is invoked, **this** pointer is filled with either managed reference to the instance (for unboxed) or object reference (for boxed). Virtual method receives **this** pointer filled with object reference no matter if it is value type or boxed type.

2.1.14 Type Members

Object type definition contains also **member** definitions. These members are fields, methods, properties, and events. All names of members of the type are scoped to the type.

Fields of the object type are used to store value and specify representation of values of object type. They are named and typed via location signatures. There shall be no two fields with the same name and type contained within one object.

Fields may be marked as static and in such case they are locations associated with the object type itself. These locations are created when the object type is loaded and initialized together with the enclosing type. Otherwise, locations for non-static fields are created and initialized during construction of new of the given type. It is also possible to mark field as **serializable**. Such field is then considered part of persistent state of a type value and is serialized.

Method specifies allowed operation on values of the type and they have a method signature. Method definition consists of name, method signature, and optionally an implementation of the method. All methods in the object type shall differ by name and/or signature.

Methods may be marked as **static** in order to create a static method. Static methods are operations associated with the whole type, while non-static methods are operations with values of the object type. When calling non-static method, value of **this** (or **this pointer**) is passed as an implicit parameter. If the method does not contain its implementation, it shall be marked as **abstract**. Abstract Methods are allowed only in abstract object types and interface types. It is also possible to provide support for modification of particular method in derived type. Such method shall be both non-static and marked as **virtual**.

Properties defines accessing contract of a value. They are set of operations that provide an access to the value. Operations are defined in a form of methods (accessors) that are used to either store (`setter`) or retrieve (`getter`) the value. These methods are named and are typed via method signature. Return value of the `getter` shall be the same as the last parameter of the `setter`. The accessibility of accessors shall be equal to accessibility of the property. Property and its accessors shall all be either static, virtual, or instance.

Events type specifies named state transitions in which subscribers register their interests in the event via accessors. Accessors are methods that provide a possibility for the subscriber either to register interest in the event (`add`) or to revoke the registration (`remove`). Accessibility of the event and its accessor shall be identical, both accessors shall each take one parameter, whose type defines the event, and it shall be derived from `System.Delegate`, i.e. it shall be delegate. Firing an event is similar to invoking all methods whose delegates are registered to such event. Both accessor methods are named and are typed via method signature.

Constructor is a method used to create a new instance of an object type. It is an instance method defined by a special method signature and it is a part of the object type definition. Before the constructor is invoked a space for new value of the object type is allocated, VES data structures of the new value are initialized, and user-visible memory is zeroed. A constructor shall call constructor of the base class before the first access to inherited instance data. Every object type, with exception of value types, shall define at least one constructor method. There is no need for such method to be public.

Similar to constructors, there exist methods that are used when the instance is no longer accessible. These methods are called as **finalizers** and they are used to free allocated resource that is non-managed. However, their execution does not occur immediately after instance is no longer accessible due to memory management (see section 2.5). Limited control over finalizer execution is provided by `System.GC` class. It is possible to create a finalizer for value type, however such finalize will be run only for boxed instances of the value type.

2.1.15 Inheritance

Inheritance of a type means that derived type guarantees support for all contracts of the base type, i.e., interface contracts, class contracts, event contracts, method contracts, and property contracts. Also all locations defined in the base type are defined in a derived type. The derived type also inherits all implementations of the base type and may extend, override, and/or hide them. Because of that, it is possible to use value of the derived type instead of base type value.

All **object types** have to declare support for exactly one object type, i.e. they have to be derived from such type. This is a significant rule with one exception of `System.Object`. This object type is the only root of graph of the inheritance hierarchy. It is CLS-compliant class and all classes have to inherit at least from this class. To prevent deriving from a particular type, such type shall be marked as **sealed**. Any CLS-compliant class has to derive from a CLS-compliant class.

Unboxed form of a **value type** does not inherit from any class. However, boxed value types do have a base class. This base class is either `System.ValueType` or `System.Enum` (for enumeration only). Even through boxed version of value type is object type, there are more restrictive rules that are applied to it. The base type shall have no fields defined and boxed value type is implicitly marked as sealed, i.e., no further deriving from such type is allowed.

Interface types may inherit from multiple interface types. Types that implement support for interface types have to provide support for all inherited interface types. Interface type inheritance is similar to specification of additional contracts that shall be supported by an implementing object type, i.e. it is possible to specify which interface types shall be supported in object type in order to provide support for inherited interface type.

Only object types are allowed to inherit implementations, i.e. inherit all kinds of **type members** (fields, methods, properties and events). In order to allow instances of the derived object type to be used whenever instances of the base type are expected, object type may inherit only non-static fields of the base type. Object may also inherit all instance and virtual methods. Constructor methods are not inherited. It is possible to hide a non-virtual method of the base type by providing a new method definition with the same name or name and signature. Both methods may be invoked because type that contains the method also qualifies the method reference.

Object type may also inherit virtual methods and provide new implementation of such method. This includes possibility to modify **accessibility** of such method. Accessibility of inherited virtual method shall either be the same or permit wider access, e.g., it is possible to make new implementation of a family virtual method to be public, not private one. If the virtual method is marked as **final**, then it shall not be overridden, i.e., no new implementation of such virtual method is allowed in derived object type.

Properties and events are not directly supported by VES and therefore rules for both name hiding and inheritance depend on source language compiler. The generated code shall directly access methods named by the events and properties.

When deriving a new type there is a possibility to modify the layout of the instance. CTS provides support for it in a form of **hiding** and **overriding**. While it is allowed to hide every kind of class members, modification of layout is possible only through instance fields and virtual methods. Each member of the class type may be marked **hide by name**. This means that members of a same kind (fields, methods, etc.) in the base class with the same name will not be visible in the derived class. If a member of the derived class is marked as **hide by name-and-signature**, then all members of the same kind in the base class with the same name and either type (for fields) or signature (for methods) are hidden in derived class.

As it was mentioned **overriding** (i.e. modification of the layout) is available only for instance fields and virtual methods. When an overriding member of derived class is marked as **new slot**, such member always get new slot in layout of derived class. This means that overridden method or field of the base class is available in the derived class by the use of qualified reference. Qualified reference combines name of the base type, name of the member, and either its type or its signature. If the overriding member of derived class is marked as **expect existing slot**, an existing slot of the corresponding member (i.e. same name, same king and same type) of the base class is reused. If there is no such slot then a new slot is created.

2.2 Common Language Specification

Common Language Specification (CLS) is a part of CLI and is set of rules, which shall support language interoperability. Types generated for execution on a CLI implementation have to conform to the CLI specification and additionally to the CLS rules. These additional rules apply only to either types visible from the outside of the assembly or members that are accessible outside the assembly, i.e., members with accessibility of public, family, and family-or-assembly. It shall provide guidelines for writing high-level programming language tool (e.g. compilers). It is possible to look the CLS and the rules contained within from three possible viewpoints, which are called a framework, consumer and extender.

Framework is a library, which contains CLS-compliant code. This means that, such library is possible to use in wider range of programming languages than it would be in the case of library with non-CLS-compliant code contained within. Framework should also avoid use of names, which are usually considered keywords in common programming languages. In addition, implementations of methods of the same name and signature in different interfaces shall be independent and it should be not assumed that value types are initialized automatically.

A **consumer** is a tool that allows an access to features supplied by the framework, i.e. compilers. Consumer may have an ability to create CLS-compliant framework, but it is not necessary. Also capability of metadata initialization for fields and parameters excluding static literal fields is not required and consumers are allowed to ignore metadata of anything but static literal fields.

The last possible viewpoint on CLS is the viewpoint of **extender**. An extender is tool that provides a functionality of consumers plus makes extending of CLS-complaint frameworks possible. Therefore, all rules and properties of the consumer are also applied to extender. Except of these, the extender shall have a capability to extend any non-sealed CLS-compliant class and to implement any CLS-compliant interface.

2.2.1 CLS Compliance

CLS defines a set of rules, which controls the properties of the visible of accessible entities from the outside of the CLS unit, i.e. assembly. Inside the unit, there are no restrictions on the programming techniques, which can be used. This is similar to the first rule of the CLS. Complete list of the CLS rules and specification of their impacts on frameworks, consumers and extenders can be found in [CLI-I].

A part of the assembly, which is CLS-compliant, shall be marked as CLS-compliant with an attribute: `System.CLSCompliantAttribute`. This attribute (see section 2.10) makes possible to explicitly specify CLS-compliance of a type. A type inherits this attribute from either the enclosing type or enclosing assembly (for top-level types), but it is possible to mark it individually. Members of the non-CLS-compliant types shall not be marked as CLS-compliant (CLS Rule 2).

2.3 Common Intermediate Language

Common Intermediate Language (CIL) is specification for a code that can be executed by executional system (Virtual Executional System; VES; see section 2.4). CLI code generator that claims a conformance to standards specified by CLI ([CLI-I, CLI-II, CLI-III]) shall produce output valid to CLI. The generator may also claim to generate verifiable code.

Validation is a test that checks file format, metadata, and CIL for its self-consistency. **Verification** of a code means to test it for access outside program's logical space. It shall ensure that the only resources (including memory) that are accessed are those with appropriate access permissions, i.e. no code shall be able to corrupt the system by accessing non-accessible resources.

The time, in which the test shall be performed, is not specified, as well as behavior in the case of the test failure. However, it is possible to run unverifiable code (i.e., code that did not passed the verification). In such case, administration security and trust control shall not trust the unverifiable code. For a relationship between validated and verified visualized see Figure 2.4.

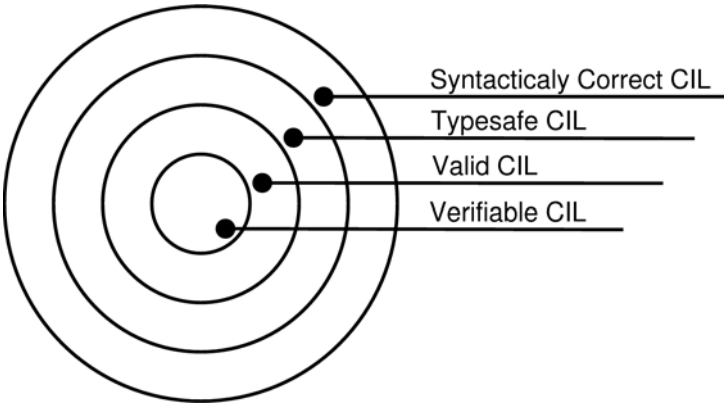


Figure 2.4: Relationship between validated and verified code

One of the important properties of CLI is **type safety**. This shall prevent code from corrupting of memory contents by writing inappropriate data. Everything is typed and due to declaration is part of metadata a compiler (or interpreter) can check code's type safety. All references are typed and both location and assigned object shall be assignment compatible.

Compiler produces metadata while compiling. Metadata contain additional information (i.e., declaration) and they are used for metadata driven-execution. This makes possible to:

- mix JITted code (i.e., code that is compiled into platform native code before its execution), interpreted code, native code (i.e., native for particular platform), and legacy code. It also allows use of uniform debugging and profiling tools for such mixture of codes.
- provide support for serialization and inter-operability with existing unmanaged (native) code.
- perform better optimization due to lack of physical offset, layouts, and sizes. This makes optimization for current platform possible.
- handle versions of assemblies, i.e., the system searches for a version that satisfies the current needs at the most (see subsection 2.7.2).

CIL specifies code that can be executed by VES. This means that CIL defines instructions that perform allowed operations. Data types that are used by instructions including description of their restrictions are defined by CTS (see section 2.1). CIL instruction does not have (with a few exceptions) specified operands. Both its operands and result of the operation are stored on a stack and CLI automatically keeps tracks of operand types. Figure 2.5 shows short example.

```
1:      .assembly extern mscorlib {}
2:      .assembly hello {}
3:      .method static public void main() cil managed
4:      {.entrypoint
5:      .maxstack 1
6:      ldstr "Hello world!"
7:      call void [mscorlib]System.Console::WriteLine(class System.String)
8:      ret
9:      }
```

Figure 2.5: CIL code example

The example above is similar to the example in [CLI-II]. It is an example of simple output, i.e, it writes ”Hello world” to the console. First it specifies assembly that is used (line 1) and current assembly (line 2). The entry point of assembly (line 4) is specified in method `main` (line 3). String "Hello world" is pushed on the stack (line 6) to be used as parameter for output method call (line 7). After that, the method `main` exits (line8). Example shall show syntax of CIL rather then to explain the instructions and rules of CIL. For detailed information and list of CIL instructions see [CLI-II, CLI-III].

2.4 Virtual Executional System

This section contains brief overview of the Virtual Executional System (VES), for more details see [CLI-I]. The purpose of the VES is to execute CIL instructions (refer to section 2.3) generated by a compiler. It has direct support for built-in types, i.e., integers (8 bits up to 64 bits), floating-point types (32 bits or 64 bits), object references, and managed pointers. Built-in types are the only types that are supported by CIL instructions. These instructions use evaluation stack for store their arguments.

2.4.1 Built-in Types

The built-in types mentioned above are supported by VES and it is possible to store such types inside a memory. However, evaluation stack, which is used for operations, supports only a subset of built-in types: 32-bit integers, 64-bit integers, native integers, and native float-point numbers that are uses for internal purposes only.

Even thought the evaluation stack (i.e. instructions) does support only 32-bit or 64-bit integers, it is possible to use **short integers** (8-bit, 16-bit). Such integers need to be converted into requested format, i.e. they are either widened or narrowed. Narrowing operation is performed without overflow tests. The time of conversions and conversions itself are CLI implementation dependent, so the behavior may vary.

There is support for 32-bit and 64-bit **floating-point numbers**. The format of the numbers is specified in [IEC]. This includes definition for NaN, +infinity, --infinity values. NaN is considered unordered for comparison operations. All operations do not generate an exception in the case of unusual condition (e.g. overflow, invalid operand) rather they produce infinity (for operations with value in limit) or NaN value. However, it is possible to check the result for infinity and/or NaN. Such check is able to generate an exception if it fails. Internal representation of the float-point value is implementation dependent and shall have the same or greater precision then that of the variable that it is representing.

2.4.2 Pointers and References

Managed pointers and object references are directly supported by the VES. However, the size of the pointer is not possible to determine during the compilation, i.e. they are not fixed size. This makes the code portable to platform that uses different address width (e.g., 32 bits per address).

For **unmanaged pointers** VES uses native signed integers and they are similar to pointers known in for example C language. **Object reference** is a pointer outside the object or pointer to the whole object and it provides only limited set of operations. **Managed pointer** is similar to object reference and it points to object members or elements of an array. It is similar to by-ref type (i.e. managed pointer contains type description) and it is possible to point unmanaged memory with it. Managed pointer shall not outlast the life of the location that is pointing to.

2.4.3 Compound Value Types and Value Types

Compound value types are types that have sub-components and that are passed by copying the value. Sub-components themselves do not have such restriction (i.e., they can be managed pointers, object references, etc.). Properties of the compound value type are similar to the value type as it was described in previous sections (see subsection 2.1.1). This means that such type can be considered either as boxed or as unboxed. The boxed version carries full run-time information (similar to instance of `System.Object`) and it is allocated on heap. Unboxed version does not contain run-time information at all and it is never allocated on heap.

2.4.4 Machine State and Evaluation Stack

Machine state is a state of the machine and evaluation stack. It consists of global state and method state. Global state consists of several threads of control, which can be through of as a singly linked lists of method state, state of multiple managed heaps, and state of a shared memory address space.

Evaluation stack is part of the machine state and it is not addressable. Therefore, CIL instructions operate only with the top of the stack. Return values of the CIL instructions are also stored onto the stack. It is possible to store any data type, including unboxed instance of a value type. However, due to restrictions (see subsection 2.4.1) a narrowing and widening are sometimes needed. For detailed information, refer section 12.3 in [CLI-I].

2.5 Memory Management

Memory management (similar to resource management) is crucial part of an application. It can be source of unpredictable bugs. These are usually caused by access of deallocated block of memory or creation of memory leaks. Such bug can occur only time-to-time and cause an unpredictable application crash. To solve it, there exist two major approaches: to use either tools that help to debug the application or a facility called **garbage collection**. The last approach that was mentioned above is the one used by the managed environment. The subsections below contain overview of memory management, for full descriptions see [Ric00, CLI-II].

2.5.1 Garbage Collection

Garbage collection is a task performed by **garbage collector** (GC). It provides mechanism for memory management (i.e., allocation, deallocation, and optimization of memory blocks), which is transparent to the user. All objects that are managed by GC are stored on a **managed heap**.

Similar to non-managed environment (e.g., C language), all blocks on the heap are linked by pointers. However, in the case of non-managed environment, an allocation means to walk through the linked list of block and search for free block that is large enough to suit the needs. Such block is then split and linked list of blocks is modified.

In the case of managed heap, there is no linked list search. The managed memory is assumed infinite and new objects are simply added to the tail of the list. This is faster then searching of linked list. However, it needs additional mechanism mentioned above, i.e., garbage collection. This includes need of knowledge of pointer types. It means the language shall not be able to perform unrestricted cast of pointer type from one type to another. This is also explains it is not possible to implement garbage collection for languages such as C/C++ and maintain the result to be able to perform all operations described by specifications.

As it was mentioned, GC performs garbage collection. Currently there exist several algorithms that solve the GC and that are tuned for optimal performance depending on a used platform. Next text contains overview of GC implementation.

To make garbage collection possible, each application contains set of roots. These are references to storage location, e.g., local variables, static object pointers, pointers to object on managed heap, etc. This set is fully accessible to GC in order to allow modifications of roots.

Only unused objects are deallocated, i.e., there is not reference to such objects. When GC starts, it assumes that all objects on heap are garbage (i.e., are not referenced). It performs recursive search of references starting at application roots. Every object is examined and searched only once per garbage collection. This means that if the object was examined while searching previous root and it is found again, its references are not searched again. It shall solve infinite referencing loops.

All objects that were not referenced (i.e., they were not examined while searching the graph) are considered for removal from memory. To perform it, GC walks linearly the heap and searches for blocks that were previously owned by non-referenced objects. When such block is found, blocks above are shifted down the heap to compact the memory. This shifting means that references to the shifted blocks are no longer valid and GC has to modify the application roots and references between objects. Compaction described above is not performed on larger memory blocks due to the high CPU-time costs.

Use of memory compaction leads to higher memory needs and may cause possible slowdown. However, this occurs only when the heap is full. Otherwise, the allocations are faster then for non-managed heap. Garbage collection provides higher comfort and decreases number of unpredictable bugs in managed code based on invalid use of the memory.

2.5.2 Finalizers

Some objects, however, allocate resources (e.g., network connections, output/input device communication, unmanaged memory, etc.) that need special handling while they are released. To make possible to release them, GC allows the user-specified code in form of a method to be called while the object is garbage collected. This process is called a **finalization** and the method that contains user-specified code is called **finalizer**.

Finalizers are methods that are called at garbage collection of the object. These methods are not similar to destructors (C++) even thought they perform similar task. The major difference between destructors and finalizers is that the user has no control over the time of finalizer's execution. Order of finalizer calls is not specified and therefore it is not recommended to access inner, member objects in it. It may also happen, that the finalizer is not called at all in order to make the application to exit as fast as possible. To prevent this happen, it is possible to force GC to execute the finalizer before the application exit. However, it may change garbage collection behavior.

Use of finalizers leads to GC performance loss while allocationg and deallocationg objects, e.g., garbage collection of an array of object with finializers leads to a call of finalizer for every object stored in the array. It is because of garbage collection of an object usually leads to garbage collection of all referenced objects. Use of finalizer leads to unwanted prolonging of a life of the object.

Internally are finalizers implemented by two queues: **finalization queue**, which contains objects with finalizers, and **freachable queue**, which contains objects waiting for their finalizers to be called. Dealocation of an object with finalizer means to move an object from finalization queue to freachable queue. Freachable queue is similar to application roots and therefore objects inside this queue cannot be removed from memory. This is also the cause for unwanted prolonging of the life of the object mentioned above. Finalizers of objects that are waiting for finalization are executed by a special thread. These thread calls the finalizer and removes the object from freachable queue, i.e., object is ready to be removed from the memory during next garbage collection session.

However, it is possible to perform a **forced cleanup** of an object. To perform it a special method needs to be created. This method is usually called `Dispose` (member of `IDisposable` interface) or `Close` and is called manually by the user. Managed environment also provides a possibility (`GC.SupressFinalize`) to avoid execution of the finalizer even thought it is specified for the object. This is often used while performing forced cleanup manually.

One of the side effects of finalizers is a possibility to **resurrect** a finalized object, i.e., to create a new reference to the object during execution of the finalizer. However, it is not recommended to use this feature due to the object (and usually some of referenced objects) is already finalized and therefore it needs to be registered for finalization again by calling `GC.ReRegisterForFinalize` method. It is important to note that the call of this method does not cancel the effect of `CG.SupressFinalize` call.

2.5.3 Optimization of Performance

To improve the performance of the garbage collection, GC uses **generation** mechanism. It assumes that newer objects have shorter lifetime and are frequently accessed. Therefore, every new object is marked to be of generation 0. After it survives garbage collection it is marked as member of generation *n+1*, where *n* is generation number. Garbage collection then occurs when generation 0 is full. If garbage collection of generation *n* does not create block of free memory large enough, garbage collection is performed for generation *n+1*.

Next possibility how to improve the performance of an application is to use **weak references**. In order to access an object a strong reference is needed. When the object is referenced by the weak reference, it is allowed to perform garbage collection over the object. However, it is still possible to retrieve the strong reference to it until its garbage collection.

It is useful for large memory structures that are used only time to time. By using weak references, these structures can be removed when a memory is needed, i.e., in the case of low free memory. Weak references are represented by `System.WeakReference` class.

2.6 Exception Handling

To provide comfortable way of handling errors and exceptional situations, the CLI supports exceptions and their handling. Only class instance is allowed for using as exception object, i.e., while it is possible to use boxed type for such purpose, use of pointer or unboxed type is forbidden. Class, which is used as exception object, shall be either

instance of `System.Exception` or instance of derived class. The user is allowed to create its own exceptions by deriving a class from either `System.Exception` class or another class that is derived from `System.Exception`. This subclassing of exceptions may used to provide more information for raised exception.

The support for exception handling is provided in form of protected blocks of code (also called as ”try block”) and exception handlers. A single protected block shall have exactly one handler. This handler can be associated with finally handler, fault handler, type-filtered handler, and user-filtered handler.

Finally handler is executed whenever the block exists no matter if the exception was thrown or not. Opposite of that, **fault** handler is executed only in the case of exception being thrown. **Type-filtered** handler handles exception of specified class or exceptions that are derived from specified class. **User-filtered** handler is used to determine whether the exception is handled, ignored, or passed to the next protected block.

An exception can be raised either by the user or by CLI. In the case of CLI, exceptions are usually raised when an instruction is executed. The exact time of a throw is not specified but the exception shall be raised before an execution of the instruction that caused the exception.

The CLI has its own set of exceptions (e.g., `ArithmeticException`, `DivideByZeroException`, `SecurityException`, etc.), which are instruction dependent (i.e., they can be raised only by a specific type of instruction). However, a special exception can be raised by all instruction. It (i.e., `ExecutionEngineException`) is raised whenever inconsistency of CLI occurs. Only unverified code can cause this exception to be thrown and it is usually caused by corrupting a memory. Inconsistency is detected before such instruction is executed and `ExecutionEngineException` is general way of handling it.

Next important exception is **resolution exception**. This exception occurs in the case of using invalid or mismatched reference to an interface, a class, a base class, a method, or a field. The time of a throw is implementation dependent. It is possible to raise the exception during initialization of a type. In such case, the static initializer is not executed. It is also possible to raise this exception at installation time or type loading time. In this case, the type load may fail and appropriate exception is thrown, e.g., when a type fails to load, a `TypeLoadException` is thrown. Another example is when the required method is accessible, but violates declared security policy. It this case a `SecurityException` is thrown.

The last possibility to throw this exception is before the instruction is executed. Such exception has the highest priority possible, i.e. if there is a need to raise the resolution exception, no other exception may be thrown. Further execution of instruction that passed the test (i.e., no resolution exception was thrown while the instruction was executed for the first time) shall not throw the resolution exception.

Exception handling is performed via table of handlers. Each method has such table that contains list of handlers of specified types. The order of handlers is important. When an exception is raised, CLI searches for a handler that suits raised exception, i.e., handler that handles the exception. If the handler is found, an exception object that contains a description of the exception is created and appropriate handler is executed. Both finally and fault handlers are called before the corresponding exception handler is executed.

If there is no appropriate handler, table of calling method is searched. If it is a top-level method (i.e., there is no calling method) and still there is no exception handler, CLI dumps

a stack trace and aborts a program. A debugger can be used to inspect the contents of the stack before any stack unwinding is performed.

2.7 Assemblies

Assembly is fundamental deployment unit that is part of managed environment. It is a scope and security boundary for types contained within. It consists of modules and other files. Module is a single file that contains executable content and it may contain description of the assembly.

Assembly may be either static or dynamic. **Static assemblies** are stored in form of a file and are loaded and then executed. Opposite of that, **dynamic assemblies** are created in a memory during runtime. It is possible to store dynamic assembly outside the memory. Both static and dynamic assembly shall contain its descriptions.

Assembly provides its description. Opposite of COM technology, the assembly does not need additional registration or registry record. This fact simplifies installation, uninstallation, and replication of an application.

2.7.1 Manifest

Each assembly contains description of self in form of a manifest. Only one manifest is allowed per assembly and it is possible to read the information during runtime. Manifest contains **assembly name**, **strong name** information (i.e., public key from publisher), **list of all files** that are part of the assembly, **cryptographic hash**, **culture information**, **originator public key**, and **version** number.

All files of the assembly shall be stored in the same directory as module that contains the manifest. **Cryptographic hash** for the contents of the file applies only to assemblies that consist of more then one file. It is coded by SHA1 algorithm and all CLI implementation shall use it to provide compatibility with other implementations.

Culture information informs about the specific culture for which is the assembly customized. It is case-insensitive string and follows format described in [RFC1766].

Originator public key is a public part of the key for RSA algorithm that is possible to use to encrypt cryptographic hash. This key is then used while assembly loading to check whether is a loaded assembly similar to an assembly that used while compilation.

Version number provides information about version of the assembly. It consists of four 32-bit integers. These integers represent:

- **major version**: This number shall be changed only in the case of large modifications (e.g., complete rewriting of assembly) to the assembly. Assemblies with different major version are not interchangeable, i.e., they are not backward compatible.
- **minor version**: Change of this number while maintaining the same major version number means that even though some significant enhancement were made, the assembly is still backward compatible.
- **build**: This number is increased every time the new build from same sources is made. Appropriate use of this number is to detect change of compiler or platform.
- **revision**: This number is changed every time a modification to source files is done. It means that even thought some bug fixes or optimization to code were done, the assembly is still fully interchangeable with previous version (revision).

Standardized libraries do have last two numbers zeroed. First two numbers (major and minor version) detects whichever functionality and additional featured of virtual machine is needed. It may also detect the needed version of virtual machine. Non-standard libraries shall either ignore it or fill it with appropriate information. Version information is used for versioning (see subsection 2.7.2).

2.7.2 Versioning

In non-managed (e.g., win32) environment users are sometimes experiencing problems with different versions of DLLs. It usually happens when some application during installation rewrites the current version of DLL with either older or newer version. This can then lead to a crash of the other application due to different versions of DLL usually do not guarantee backward compatibly, i.e., they have modified interface. In addition, applications usually expect only one version to be installed on a machine. This can lead to problems with overwriting of DLLs described above. This is a problem of maintaining consistency between set of components, which were used to build the application, and components currently present at run-time.

Assembly versioning is an attempt to solve it. It is possible that the same assemblies, which differ only by a version, will coexist at one machine without side effects. However, versioning is available only for assemblies with strong names (i.e., signed assemblies; see section 2.8).

When an assembly is requested to be loaded, system decides whichever version will be picked. First system retrieves the information about needed assembly from manifest. Then it checks applicable configuration files, looks for available assemblies, and finally determines version that shall be loaded.

Configuration files, which influence the version determination, are stored in XML format in order to make possible for the user to manually edit them. There are three kinds of configuration files: machine configuration (it is applied to the whole machine), application configuration (it specifies information about application, policy for assembly binding, and among others it contains application settings), and security configuration (it provides information about security permissions). Even though it is possible to modify the configuration files by hand, it is not recommend approach due to corruption of such files may lead to application failure. For more detailed information see [MSDN].

2.8 Security

In non-managed environment, an application or library usually needs to be installed on local system in order to be used by the user. Opposite of that, managed environment allows dynamic download and remove execute of a code. However, this means a possibility that malicious code will be executed. Therefore, available security system shall prevent damages to the system. Every application (assembly) has to interact with it and handle possible security exceptions due to system may deny required permissions. It is important to know that security system setting may differ from computer to computer.

The application (assembly) may specify permission that it either requires to run or does not want. The syntax of permission declaration is either declarative or imperative. **Declarative** syntax means use of attributes (see section 2.10). Permissions are then stored in metadata and used while compiling. It is possible to express all security actions by declarative syntax. However, it is not possible to change declarative security at run-time.

Imperative syntax is based on creating an instance of permission class and invoking its methods in order to set the security permission. This approach allows constraints to be

made at runtime. However, it is not possible to express all security actions by this approach. Figure 2.6 shows simple example that is illustrative. It uses C# syntax.

```
[FileIOPermissionAttribute(
    SecurityAction.Demand, Read="C:\\Directory\\File.txt")]
FileIOPermission perm = new FileIOPermission(
    FileIOPermissionAccess.Read, "C:\\Dir\\File.txt");
perm.Demand();
```

Figure 2.6: Example of setting permission to read C:\Dir\File by declarative (upper) and imperative (lower) security syntax

There are two kinds of security: code access and role-based. **Role-based** security is based on knowledge of the user and the user's role. **Code access** security does specify permission for an assembly. The assembly may declare either required permissions or permission that it does not want. Permissions, which are granted to the assembly, are based on source of assembly code (e.g., local intranet, internet, etc.) and it may restrict access to local file system, registry, network, user-interface, or execution environment.

Permissions are provided in sets and managed environment contains predefined sets of permission, e.g., *FullTrush*, *SkipVerification* that allows assembly to skip verification process (see section 2.4), *Nothing* that grants no permission, Internet that represent non-trusted source with restricted access to local machine (e.g., files), etc.

Permission granting is controlled by **security manager**. It uses supplied assembly information (also called **evidence**) and it passes it through policy levels. There are four policy levels: enterprise, machine, user, and application. Each level may modify set of granted permission supplied by previous level, however, granting of a permission that was denied by a higher level is not allowed.

Each policy level contains a tree of code groups (see Figure 2.7). It is a tree of conditional expressions and permission sets. A permission set is granted if the condition is evaluated as true. Evaluation of branch is stooped whenever the condition fails.

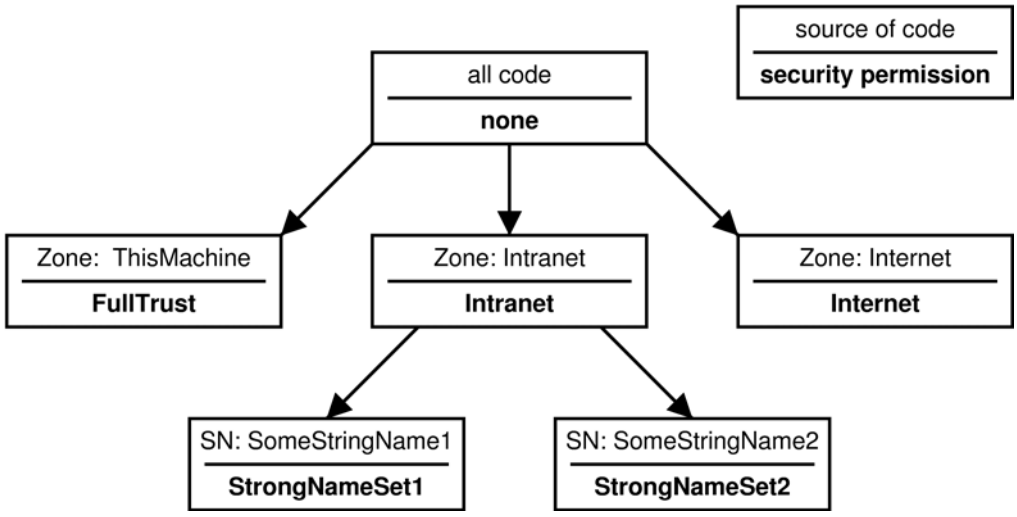


Figure 2.7: Example of tree of code groups

2.9 Multithreading

System uses process to separate applications. There exist multiple threads per process. Managed environment creates further division of processes into managed sub-processes called **application domains**. A system process may contain multiple application domains. Each domain is started in single thread. Domain has own security permissions and it is represented by `System.AppDomain` class.

Within single application domain, multiple managed threads may exist and such threads are represented by `System.Threading.Thread`. Managed environment also provides support for asynchronous programming. This means that a thread does not need to wait for results of an operation. It just initializes the operation and provides callback that handles the result. Support for synchronization primitives is also provided by managed environment. The primitives are:

- **Synchronized methods**: Methods containing a lock visible across all threads that control entry point of particular method.
- **Explicit locks and monitors**: Managed environment provides support for basic snychronization primitives, such as, monitors, mutexes, events (with either automatic or manual reset of a status), and locks that support single-writer and multiple-reader semantics.
- **Atomic operations**: Atomic operations are operations that cannot be interrupted by switching of threads. Managed environment supports atomic operation with variable, such as increment, decrement, exchange, and comporare-and-exachange (i.e., value is compared with supplied one and if it is equal the exchange of value is performed). Atomic operations are represented by `System.Threading.Interlocked`.

CLI itself shall guarantee that reading and writing of properly aligned memory location, which is not larger then native integer, is atomic. However, a developer shall not assume that values are properly aligned. Instead of that, the developer shall use class `System.Threading.Interlocked` for atomic operations.

2.9.1 Thread Pool

To improve performance of multithreaded application, support for thread pool is provided by managed environment. A thread pool is a group of threads that handle a request (job). Number of threads in thread pool is dynamic and depends on current CPU usage in order to gain maximum performance. Thread pools shall be used for relatively short tasks that do not block other threads. Even thought multiple threads may exists in a single process, only one thread pool is allowed per single process. This centralizes control over thread pool and makes possible to gain maximum performance (i.e., there is no third party thread pool that may decrease performance). All application domains in a process share the same thread pool.

Thread pool also offers an effective way how to create a code that is executed when the synchronization object is signaled. Only mutexes and events (both with manual and automatic reset) are supported and it is possible to set the method to be executed either every time the synchronization object becomes signaled or only once. The execution scheme is then optimized for minimum CPU time leaks.

2.9.2 Multithreading vs. Garbage Collection

To make garbage collection possible in multithreaded application, all threads are suspended while GC performs its task. It is also possible for GC to modify thread stack in order to make thread to initialize the garbage collection. This approach is called **hijacking**.

Another approach uses so-called **safe points**. It is based on a fact that GC can perform garbage collection undisturbed when a thread is executing unmanaged code. This concurrent run of GC and the thread is possible due to unmanaged code cannot access the most of managed objects with exception of pinned objects. Pinned objects are objects that cannot be moved or removed by GC. When then the thread returns back to manage code, it is suspended until the GC finishes its task.

2.10 Attributes

Attribute is a facility that allows an additional property to be specified for either an assembly or code elements, such as types, fields, methods, and properties. It may also provide further information and affect run-time behavior, e.g., it is possible to mark a particular code element to be obsolete, it is possible to allow an enum to be used as a field of bit flags, etc. Attributed are also used to specify declarative security permissions. Use of an attribute modifies metadata.

Metadata contain additional information to code elements or an assembly. This makes possible for a file to provide its description (i.e., self-describing file). It is used for assemblies and due to that, the assembly, opposite of COM, does not need registration in order to specify its functionality.

All attributes are derived from `System.Attribute` class and it is possible to specify to which code element can be such attribute applied. It is allowed for the user to create its own attributes that may provide user-specific information. Example of an attribute can be found at Figure 2.8. Example is illustrative and uses C# syntax.

```
[Obsolete("May not be supported.")]
public void FooMethod() ...
```

Figure 2.8: Example of attribute applied to method `FooMethod`. It causes a compiler warning with specified string to be generated at compilation time

2.11 Non-managed Code Interoperability

Due to the fact that there exist many non-managed libraries that are useful, managed environment (.NET Framework) provides support for accessing non-managed code and COM (Compound Object Module; see [MSDN]) that has direct support. When managed code accesses COM, a so-called Runtime Callable Wrapper (RCW; see Figure 2.9) is created. RCW handles COM interface querying, method calls, and data conversion, because of caller and called code may have different representation of data and data structures.

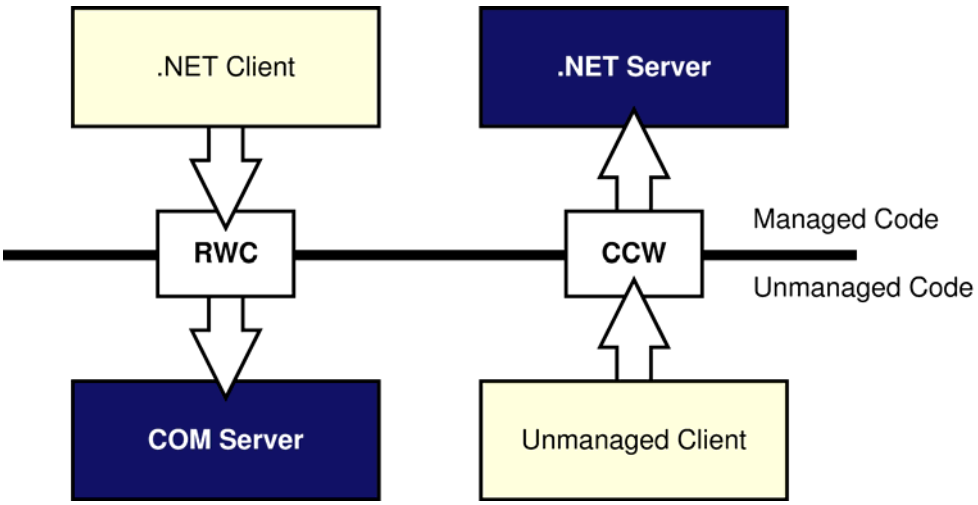


Figure 2.9: COM interoperability scheme

It is also possible for a non-managed code to call a COM that is implemented in managed code. In such case, managed environment creates wrapper called COM Callable Wrapper (CCW) that is similar to RCW, i.e., it handles method calls and data conversions. This feature provides a backward compatibility for older applications that are based on COM technology.

Even thought time costs of COM calls are considered low, it is not recommended to use COM interoperability for components that perform only short tasks. In this case, complete rewrite of this COM component to managed environment is recommended.

Sometimes are non-managed libraries available only in from of set of C-style functions stored in DLL (e.g., Win32API). Therefore, managed environment provides support for invoking of C-style functions. It is called as Platform Invocation (also **P/Invoke**) mechanism. This mechanism performs data conversions of function parameters and provides support for callbacks. However, it is not possible to directly call managed code from non-managed one. In such case, use of COM is recommended.

Code that uses P/Invoke mechanism cannot be verified and therefore it has higher security requirements, i.e., *FullTrust* set of permission is needed in order to run application or assembly that uses non-managed code. In order to use P/Invoke mechanism a function header has to be specified together with target DLL by `DllImportAttribute`.

The third approach that is available when interoperating with non-managed code is to use language that supports mixture of managed and non-managed code. Due to that, it is possible to link non-managed library similar to a usual approach for library linking and call non-managed code. This approach is usually called as "It Just Works" (IJW) and example of language that provides support for IJW is Managed Extension C++ (MC++).

3 Programming Languages for .NET

.NET is not only an environment that handles memory management and runs generated CIL code, but it is also specification of interface that can be used by programming languages. It specifies type system (CTS; see section 2.1) including allowed operations over specified types. It also describes recommended library interface construction including naming conventions and recommended usage of namespaces, classes, interfaces, etc.

This means, that .NET specifies interface and property of such interface and therefore it is possible to create compilers for various languages to aim .NET. This is also one of the purposes of .NET: to make cooperation of various languages as easy as possible.

Currently there exist compilers for Eiffel, Ada, Visual Basic, C++, C#, etc. This chapter contains brief introduction into languages that were used for implementation and testing, i.e., C++ and C#. Rather then complete description of both syntax and semantics; this chapter is an introduction into the most important language features and properties.

3.1 C#

C# (pronounced C Sharp) languages is a special object-oriented language. It is special language, which was created for .NET in order to make most of CLI capabilities available to user. Its design is aimed on simplicity (those of Visual Basic) and the shorted learning time as possible. Therefore is syntax very close to syntax of C/C++. C# is ECMA standard and for detailed syntax and semantics description see [CSharp]. This section describes major aspects of the language.

C# does have preprocessor similar to C/C++. However, it is not possible to use compile-time constants (`#define`) as macros, i.e., to replace portions of code before code is compiled. These compile-time constant are used for conditional compilation only, i.e., to exclude blocks of source code using preprocessor directives `#if` or `#elif`.

Next major difference in comparison to C/C++ is an absence of header files. It is situation similar to Java, i.e., single source code file contains both declaration and implementation. However, single source code file may contain more both accessible and visible classes.

Language fully supports Unicode even for identifiers. It also has support for multithreading in form of `lock` keyword that allows to put block of source code under protection of critical section.

C# also contains special keyword (`foreach`) that simplifies iteration through arrays or linked lists. Language includes implicit boxing of value type whenever it is needed, e.g., when performing type casting of value type (`Int16`, `Int32`, etc.) to `System.Object` type. Next interesting property of the language is that it is possible to add explicit overflow checking for arithmetic operations (`checked` keyword).

Language itself has support for unmanaged code interoperability (see section 2.11) in form of `unsafe` and `fixed` keyword. However, code that uses these keywords require high security permission to run and therefore it may cause security policy exception to be raised. For details see [CSharp].

3.1.1 Class and Class Members

Class is a fundamental language construction of C#. Class type and class type properties are described by CTS (see section 2.1). Class type does support **attributes** that may specify additional properties such as security permission, structure members memory layout, etc.

Class is declared similar to C++, i.e., by using `class` keyword. However, opposite of C++, only one parent is allowed for a single class and each class (excluding `System.Object`) is implicitly derived from `System.Object`. C# supports **abstract** classes (`abstract` keyword) together with classes and class member whose overriding is not allowed (`sealed` keyword).

Value types are declared using `struct` keyword. Both class types and value types (with exception of nested ones) shall have their **visibility and accessibility** specified. Only two modifiers are supported for them: **public** and **internal** that is similar to assembly accessibility permission (see subsection 2.1.10).

Similar to C++, current instance of class is reference by `this` keyword. However, unlike C++, base class is reference through `base` keyword. Next difference is that each class member including nested types shall have both its accessibility and its visibility specified. It is possible to choose from four accessibility modifiers: **private**, **protected**, **internal**, and **public**. Similar to C++, if accessibility modifier is not specified then class member is private.

Opposite of C++, C# does not allow default values for **method parameters** and each method parameter may be specified to be input, output (`out` keyword), or input/output (`ref` keyword). This together with compiler checking for use of local variable, which was not initialized, shall increase programming safety, e.g., during method call it is not possible to use uninitialized variable for input/output parameter while its use as output parameter is allowed.

C# support virtual methods, however, syntax of their overrides differs from C++. When virtual method is first declared keyword `virtual` is used. While overriding it, keyword `override` needs to be used instead of `virtual`. Operator overload is also supported, however, the syntax is slightly different that those of C++.

Language supports **destructors**, but they are similar rather to finalizers then to C++ destructors, i.e., they are called during finalization of object. It has also support for exception handling in form of **try-finally**, **try-catch**, and **try-catch-finally** blocks (see section 2.6).

3.1.2 Example

To illustrate some properties of language a short example is included (see Figure 3.1). In this example, first class `MyClass` is defined (line 1) that supports interface `IDisposable`. The class contains protected field (line 3) that is initialized to value 7, public method (line 4), public constructor (line 5), and property (line 6) with its getter (line 8) and setter (line 9) method defined.

The entry point of an application is specified in form of `Main` method (line 12) with an attribute (line 11). This method first creates instance of `MyClass` (line 14), writes value of `StoredValue` property to standard output (line 15), and calls method `Dispose` (line 16).

```
 1:      public class MyClass : IDisposable
 2:      {
 3:      protected int storedValue = 7;
 4:      public void Dipose() { ... }
 5:      public MyClass(int val) { storedValue = val; }
 6:      public int StoredValue
 7:      {
 8:          get { return storedValue; }
 9:          set { storedValue = value; }
10:      }
11:      [STAThread]
12:      public static void Main()
13:      {
14:          MyClass obj = new MyClass(10);
15:          Console.WriteLine(obj.StoredValue);
16:          obj.Dispose();
17:      }
18:      }
```

Figure 3.1: C# source code example

3.2 C++ Managed Extension

C++ is one of the languages that have compilers for .NET. However, language itself does not contain construction that would support .NET capabilities. Therefore, C++ is supported in form of **C++ Managed Extension** (MC++) that differs from plain C++ in few details.

MC++ supports attributes, properties, delegates, etc. It also widens accessibility permission of C++. It is possible to specify accessibility rights of class type members for both assembly and outside world by declaring two accessibility modifiers at once. The modifier with more restricting accessibility permission is valid for outside world, while the other is valid for the assembly, e.g., **public private** accessibility right means, that the member is **private** to the outside world and public for referents that are members of same assembly (i.e., it is similar to **internal**; see subsection 3.1.1).

Managed version of C++ also adds a few new keywords that are prefixed with two underscores. These new keywords allows usage of managed environment capabilities:

- **__nogc** is used when declaring non-managed type (class, structure). It is implicit for all types and it is inherited, i.e., it is not possible to derived managed class from non-managed one.
- **__gc** specifies managed (garbage collected) reference type (see subsection 2.1.1). Such class is allowed to have only one parent, but may implement multiple interfaces. This keyword is possible to use to define **managed pointers** and **managed arrays**. These arrays are similar to C# arrays, i.e., they have boundary control and automatic

initialization of array members. It is possible to have non-managed members of managed class. However, it is not permitted to have managed members of non-managed class. Similar to **__nogc**, it is not possible to derive non-managed class from managed one.

- **__value** denotes managed value type. It can be used with `struct`, `class`, or `enum` keyword. Its use with `struct` or `class` keyword is similar to `struct` keyword in C#.
- **__abstract** denotes abstract class.
- **__sealed** protects class or method from being overridden.
- **__interface** declares managed interface. It is similar to `interface` keyword in C#.
- **__delegate** declares delegate that is roughly comparable to C++ function pointers.
- **__property** denotes getter or setter method of property or indexer. Return value of getter method shall have same type as one of the setter method parameters. Usage of property is possible either by calling method or similar to C# (i.e., by specifying property name together with assignment operation).
- **__pin** specifies pinned pointer. This pointer is similar to **__gc pointer**, but it prevents garbage collector from moving of an object during garbage collection. Object is then unpinned by setting pinning pointer to 0 or by running out of scope where wan pinning pointer defined. It is allowed to use pinning pointer only for local variables.
- **__box** performs boxing operation of value type.
- **__identifier** allows use of C++ keyword as identifier.
- **__try_cast** tries to perform type conversion. If it fails then it raises an exception.
- **__type_of** gets type (`System.Type`) of specified type.

The important advantage of MC++ is that it allows mixing of managed and non-managed code. It uses IJW (see section 2.11) and therefore its cooperation with non-managed code is very easy.

The use of MC++ may also lead to higher performance then in the case of C#. It is also little bit more flexible (e.g., it allows C/C++ macros for preprocessor), however, the syntax is somehow cumbersome in comparison with C#. Also the fact that is allows to mix managed and non-managed code means that there is no verifier for MC++ (see subsection 2.1.12).

3.2.1 Example

To illustrate how does MC++ code actually look like, a short example is included at Figure 3.2. This example is similar to example at Figure 3.1 and shall provide an opportunity to compare C# and MC++ syntax. Example contains both declaration and implementation even thought they are usually placed in separated files.

```
 1:      public __gc class MyClass : public IDisposable
 2:      {
 3:      protected:
             int storedValue;
 4:      public:
             void Dipose() { ... };
 5:          MyClass(int val) { storedValue = val; };
 8:          __property int get_StoredValue() { return storedValue; };
 9:          __property int set_StoredValue(int value) { storedValue = value; };
11:          [STAThread]
12:          static void Main()
13:          {
14:              MyClass obj = __gc new MyClass(10);
15:              Console::WriteLine(obj->StoredValue);
16:              obj->Dispose();
17:          };
18:      };
```

Figure 3.2: MC++ source code example

4 Graphical Interfaces

The goal of this work was is to introduce common graphical interface to .NET environment. This chapter contains brief introduction to graphical interface (OpenGL). It also describes possibilities of graphical output provided by .NET Framework library. This chapter is a not a complete guide to interfaces that were used, rather it is an introduction.

4.1 .NET Framework Library

Managed environment (.NET) provides facilities to compile, manage, and run code. It includes .NET Framework library that contains huge set of classes. This set includes classes that allow user to use GUI of underlying operating system and because of GUI usually allows simple graphical output, support for such output is provided. Classes that provide graphical output are members of `System.Drawing` namespace and encapsulate services of GDI (graphics device interface) and GDI+ (for Windows® platform).

GDI itself is part of Win32API (see [MSDN]) and it is aimed on drawing operating system GUI. Therefore, it provides simple 2D output. It is possible to draw basic primitives (e.g., point, line, rectangle, etc.), to display images, and to write text using available fonts. It also includes possibility of simple adjustments of images.

GDI+ is improved version of GDI. It adds additional functionality that improves the output. This includes possibility to use of alpha channel in images in order to support translucency, support for more 2D graphical primitives (e.g., Bézier Splines), tools for anti-aliasing of drawn primitives (e.g., lines, curves, etc.), etc. GDI+ also supports transformation of drawn primitives that is similar to that of OpenGL (see subsection 4.2.2).

Even thought GDI+ improves functionality of GDI it does not support features that are implemented in current graphical hardware. It provides just enough functionality that meets needs of GUI. This means that only 2D output is supported and performance does not usually meet capability of used hardware.

4.2 OpenGL

OpenGL is a graphic library based on commercial graphical system by SGI and was introduces first in 1992. It is worldwide known interface and is used in both industry and games for graphical output. Opposite of GDI (GDI+) it is aimed on performance and it makes possible for the user to use available features of installed graphical hardware. It provides support for displaying of 3D and 2D object.

4.2.1 2D Object Support

The main aim of OpenGL is visualization of 3D objects. Support for 2D objects is limited and it is based on 3D objects, i.e., drawing of line as 2D graphical primitive is similar to drawing of line as 3D object on a surface. It is also possible to use OpenGL for pure 2D operations (image processing) because the interface contains functions for setting and retrieving values of pixels inside a specified area (rectangle) at the target frame buffer.

Unfortunately, these functions do not have good hardware support and their capabilities are not sufficient. This usually leads to lower quality of output (e.g., nearest-neighbor approach is used for scaling) or performance loss. Therefore it is better to use simple 3D objects with texture mapping instead of these pure 2D objects and because of features that are commonly supported by the hardware, the use of simple 3D object instead of 2D ones leads to improvement of performance.

4.2.2 Basic Features

OpenGL library provides complete rendering pipeline. This pipeline supports facilities for visualization of 3D objects, i.e., it handles clipping, lighting, texturing, transformations, and visibility solving. It may handle more tasks, however, these ones are fundamental and are supported by all versions of OpenGL. The result of 3D world projection is then rendered to the given frame buffer or window of current (underlying) GUI; i.e., projecting on **render target**.

Each object is described by its surface. **Geometry** of such surface can be specified using few basic render primitives, such as points, lines, triangles (including triangle strips and fans), and quads (including quad strips).

The library also provides **light** and **lighting** computation as a standard part of the rendering pipeline. It is possible to choose from common light types, such as point light, directional light and reflector and to set up their parameters. These light types are often supported by the hardware. If such support is not available on current graphic hardware, OpenGL implementations provide software emulation.

It is also possible to cover surface of every rendered primitive with user-defined **material**. The material describes interaction of surface and light, i.e., it describes its color. The description of surface may also include texture specification.

Texture is an image that is mapped on the surface (i.e., surface is covered with the texture). Each object may be covered with 2D or 1D texture. Use of multiple textures on single object (i.e., multi-texturing) is also supported as well as techniques that shall improve the result of texture mapping (e.g., mip-mapping, perspective correct mapping).

To allow object geometry manipulations, OpenGL provides support for **transformations** of rendered geometry (i.e., vertices) before its rasterization on target (screen, image) surface. Transformations are provided in form of matrices. Thanks to mathematical background (for detailed descriptions see [Wat00]) of such form, it is possible to combine simple transformations by multiplication of transformation matrices.

OpenGL uses it and provides support for combining of simple (particular) transformations into complex one. Particular transformations can be set either by the user (e.g. in the form of matrix) or by standard library functions. Such functions offer a possibility to parameterize basic transformations, such as rotation around given axis, translation by a given vector and scaling by given coefficients. The output of such functions is automatically combined with results of previous combinations.

Currently there are two matrices in OpenGL: projection matrix, which is used for final projection of transformed 3D objects on 2D surface, and modelview matrix, which is used to transform rendered 3D objects before their projection and rasterization (see [GL13]). User is allowed to retrieve, to paste, or to store current transformation (both projection and modelview) matrix on internal stack. The latest mentioned feature simplifies rendering of hierarchical object (e.g., robot arm, human body, etc.).

OpenGL also provides support for solving **visibility** of rendered objects in form of Z-buffer. This approach of visibility solving is based on pixel basis, it is supported by common graphical hardware, and therefore its use does not decrease performance. Similar to Z-buffer OpenGL also supports stencil-buffer that is used to mask parts of render target on per pixel basis. However, stencil-buffer is not supported by older graphical hardware and therefore its use may lead to significant performance degradation in some cases.

4.2.3 Interface

Interface is the most important part of a library. It is the only part visible to the user. In the case of OpenGL, this interface consists of a group of functions and constants. These functions are not grouped into classes and therefore it is possible to use the library in non-object-oriented languages (e.g., C). However, this construction may lead to disadvantage of not-well readable source code that uses OpenGL.

Interface structure (i.e., contained functions and constants) is defined by specifications (see [GL13, GL14]) that are open to the public. These specifications also describe behavior and prescribed reaction of the library to calls of its interface functions.

One of the important advantages of OpenGL interface is its stability. This means that each new version is full backward compatible and it neither adds a complete set of functions nor modifies existing ones. It just enlarges the existing set of functions (and/or constants) by new ones. OpenGL implementation, which fulfills particular specification, provides all functionality described. This means that even thought it may benefit from hardware support of features described by specification, it shall also provide software emulation if a feature is not available in hardware.

The additions and changes that are introduced by a new version usually follow common features that are implemented in the available graphical hardware. However, the version of OpenGL is not updated fast enough to reflect evolution of the hardware and therefore latest features are supported in the form of OpenGL Extensions (see subsection 4.2.5).

To show how an actual source code using OpenGL interface looks like, there is a short and simple **example** at Figure 4.1. Example uses C language and it does not contain platform specific (e.g., initialization) code.

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

Figure 4.1: OpenGL code example

Output of the example is a white triangle on black background. Fist function in the example sets color (black) for clearing the background. Then the background is cleared with such color. Afterwards the color of the triangle is set. The following block of the source code defines the triangle (i.e., sets coordinates of its vertices). The last function

ensures that all functions above are actually performed (i.e., they do not stay waiting in the queue or buffer to be performed later).

4.2.4 Inside of OpenGL

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

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

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

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

4.2.5 Extensions and Other Libraries

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

One of these is the **GLU** (OpenGL Graphic System Utility Library; see [GLU13]) library. It provides hi-level functions and functionality for OpenGL. It simplifies setting of projection transformation and provides facilities for rendering and tessellation of parametric surfaces (e.g., NURBS) and quadrics (e.g., sphere).

Another case of such library is the **GLUT** library. It aims at simplification and unification of OpenGL initialization and its cooperation with currently available GUI. This is because OpenGL interface itself is standardized by specifications while its initialization and cooperation with current GUI is not. In addition, OpenGL does not contain any facilities for input because it handles output only.

Due to that, the initialization may not be simple enough and it is as well as user input handling platform specific. Therefore, it may complicate porting of an application to another platform. The GLUT library provides environment, which unifies these tasks and makes source code portable to different platforms.

Add-ons such as **GL Extensions** were mentioned last. These are part of the OpenGL library and provide a possibility to use the latest hardware features, although they are not available in specifications yet. However, the main drawback of GL Extensions is that, they are not part of the specification and that each graphical hardware vendor usually creates its own set. Due to that drawback, OpenGL implementations do not support their software emulation in the case they are not supported by graphical hardware.

5 The Goal

This chapter describes in detail goal of this work and it explains the reason for particular goals. The chapter does not contain description of difficulties and description of current state of the art.

In the beginning of year 2002, the final release version of .NET Framework was released. The .NET Framework is collection of libraries and runtime environment (for more details see chapter 1). This environment is quite comfortable and together with libraries that are shipped as a part of .NET Framework it offers quite powerful environment.

However, none of these libraries offers hi-speed graphical output that would be comparable to OpenGL/GLU or DirectX. Therefore, the goal of this work is to connect the comfort of .NET and performance of OpenGL together. The result shall benefit from both of them, i.e., from managed memory (.NET) and from worldwide known interface of OpenGL/GLU.

The result shall completely **avoid** use of **unsafe** blocks of code in a code that will use the result. This shall increase a possibility of code verification for languages that have verifiers and can lead to better code optimization during compilation.

Next goal of the work is that the result (library) shall be **CLS-compliant**, i.e., it shall follow CLS rules mentioned in [CLI-I]. This shall make an interface of the library usable in languages whose compilers aim .NET (for more details see [CLI-I]).

Due to .NET is aimed on **object oriented** languages, the result shall be object-oriented even thought the OpenGL/GLU is not. The result shall not be just encapsulation of all OpenGL functions into single object as static methods but it shall split it into a few **classes** based on the meaning of functions.

This shall make possible to **separate** various OpenGL/GLU versions in order to make possible for the user to select and use just the version that it is needed. It shall increase flexibility. However, the split shall not create an interface, which is completely new. The result shall maintain the highest possible **compatibility** with original OpenGL specification. This shall simplify learning of this interface.

Next important goal of the work is to **increase programming safety**, i.e., simplify debugging. This will achieved by additional method parameter checking and by replacing constants with enumeration data types.

Parameter checking shall be aimed on use of arrays as the source of many application crashes, e.g., the application rewrites wrong part of a memory due to user passed too short array to retrieve data.

Replacing of constants with **enumeration** data types is similar to parameter checking, i.e., it shall prevent user from passing invalid data to a method (e.g., parameter value that is not valid for particular interface version).

It is clear that this additions increase library port overhead, i.e., it may cause such code to be somewhat slower that its non-managed counterpart. Therefore, the result will be available in two versions:

- **debug** version, that will be used for debugging purposes and will contain full parameter checking mechanism. This version will be aimed on programming safety rather then on performance.
- **release** version, that will be stripped of parameter checking and other additional code. This version will be aimed on **performance** rather then safety. The recommended use of this version is for a code that was developed with debug version.

The next important goal of the work is to **simplify** porting of a new version or GL Extension. This simplification shall **minimize** need of manual input in the form of a tool. The tool shall create a template for particular OpenGL version/GL Extension that can be then adjusted manually.

This work includes creation of set of **examples** that shall **test selected functions** for correct behavior. These examples may be used for testing of correct implementation of an interface. They may be used in the case when the inner implementation of port changes and there is a need to check whether the new implementation behaves same as the previous (correct) one.

The result will be **measured** in order to compute slowdown due to overhead of porting. The measurement shall be done for **selected functions** and few **scenes**. The values then will be compared to other existing solutions (see chapter 1) and non-managed version. It shall prove whether is the slowdown of the result significant for single functions and whether is the slowdown significant for whole scene as mixture of previously measured functions.

The result shall **not** be complete port of all OpenGL/GLU versions and GL Extensions. It shall rather prove if the selected solution is useful and it shall create mechanism for porting of newer OpenGL/GLU versions and GL Extensions.

6 Introduction to Porting and Difficulties

This chapter contains introduction to porting of library. It shall describe possibilities, advantages, and disadvantages of this process. It shall also describe difficulties that occur when creating non-managed library port. This chapter is not aimed strictly on OpenGL. It is general introduction, which shall explain majority of terms used in next chapters.

6.1 Porting × Wrapping

The goal of this work was described in previous chapter (see chapter 1). It was mentioned there, that the result is a port of OpenGL library. This section describes possibilities of porting **without** aim at any particular library. However, it does not contain description of COM-based libraries that have direct support from .NET Framework.

For non-managed libraries that are not based on COM technology, there are two ways how to achieve the needed port:

- porting of a source code,
- wrapping of an interface.

6.1.1 Porting of Source Code

Porting of source code means to rewrite original library from non-managed code to the managed one. Unfortunately, this means that there must be changes done at source code level: changes, which are not trivial, and changes that may lead to significant modification of internal library structure.

Syntax of source code needs to be changed as well as the structure of the library including an implementation of algorithms. This is because managed environment (.NET) has different requirements. Compared to non-managed environment that was used for implementation and development of the library, managed environment has several restrictions, e.g., memory access, memory management, etc.

It means that it is not possible to use pointer arithmetic similar to non-managed environment in order to create fully verifiable (i.e., without non-managed blocks) library. It also means that it is possible neither to make port just by compiling the source code with compiler that aims managed environment nor to convert it to language that aims .NET (e.g., C#) and then compile it. That is why the automation of this process is not trivial.

These libraries are usually huge. This means that it would take too long to rewrite the code and therefore it could happen that the next version of library would be released before the old one is ported. In addition, this approach is possible only for libraries whose source code is available to a developer.

Next problem of this approach is the fact that some libraries are heavily system-dependent or low-level (e.g., OpenGL). Therefore porting of such libraries on the source code level is either not possible (i.e., original library uses block of machine code) or leads to libraries that may not execute on another platform (i.e., library depends on operating system calls). Both of these cases lead to library that may contain non-managed code and therefore it cannot be considered verifiable.

The major advantage of this approach is that the result may be fully platform independent and verifiable. This means that it would be possible to execute such code with lower security permissions (see section 2.8) and thanks to executional system, it may be optimized for particular platform more effectively.

6.1.2 Wrapping of an Interface

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

Thanks to that, the only thing that needs to be ported is the interface. This approach is more flexible in comparison with the previous one. It has one major advantage: it is possible to automate the task of porting in order to minimize manual input during porting. It means that ports of new version of the library can be created fast enough and therefore this shall avoid the situation, where new version is released before the old one is ported.

Next important advantage is that low-level and heavily system-dependent libraries can be ported with this approach. This means that it is possible to port wide range of system and/or low-level libraries that usually provide highly useful services for the user (e.g., OpenGL).

However, this approach leads to non-verifiable result due to it is a **wrapper**. A wrapper is a set of functions (or methods) that are usually similar to functions of the original library. These functions allow cooperation of managed and non-managed code, i.e., they handle data sharing, needed conversions, and type casting. Each of these wrapper functions usually contains call of particular original library function that is non-managed.

This means that binary form of original library needs to be available and therefore such port of the library is able to run only on specific CPU and/or operating system platform. This disadvantage is quite significant because every new platform (operating system and/or CPU) on which managed environment runs needs new port of the library.

It is somehow against the idea of managed environment, where the compiled application created on one platform can run without any changes on another one. However, it is the only possibility for porting of low-lever or system libraries. The example of such low-level library is OpenGL.

Another example is VTK library (refer to [Fra03]). This library is not low-level or system dependent, however, it is quite huge library and therefore it is not possible to port it as it was described in the first approach (see subsection 6.1.1). Use of the first approach together with the fact that the library is quite huge may lead to the problem that was

described before, i.e., new version of the library can be released before the old one is ported.

6.2 Difficulties

This section contains description of common problems when porting (i.e., creating a wrapper) non-managed library similar to OpenGL (i.e., group of static functions and constants) to managed environment. It briefly describes facilities of .NET that solve (at least partially) the problem.

However, solution that was used for the work is not part of this section. This section shall prepare reader for description of existing solutions (see chapter 1) and solution used by this work (see chapter 1).

6.2.1 Data Sharing

Data sharing is the fundamental role of an interface. It is the only way the library communicates with the code that uses it therefore it is very important to achieve simple and effective data sharing between the code and the library. In the case of this work, data are shared between managed and non-managed environment. The approach to the data sharing depends on used data type, i.e., value data types are handled differently then referenced data types or callbacks.

Sharing of **built-in value data types** (see section 2.1.1) is simple and it is fully handled by P/Invoke mechanism. It is simple because of the memory layout of such types is usually similar to a memory layout of appropriate non-managed data types (for ix86 compatible platforms).

Similar situation governs other **value types** and **structures** that are passed same way as built-in value types, e.g., they are copied onto the stack when particular function is invoked. However, the major difficulty of data sharing is use of **arrays** whose members are **value types**. This difficulty is caused by managed memory of .NET environment.

In the case of .NET environment, the memory is managed, i.e., there is a facility called **Garbage Collector** (see subsection 2.5.1) that handles memory deallocation and optimization of free memory layout. This means it can release allocated memory blocks and move allocated blocks of memory in order to compact allocated ones.

The automatic deallocation happens whenever there is no reference (e.g., managed pointer) to such block of the memory. The deallocation may happen even thought there is active reference to such block of the memory in the form of non-managed pointer because of non-managed pointers are not registered by managed environment. From viewpoint of managed environment, non-managed pointers are considered integers no matter the meaning of the value.

For data sharing between managed and non-manage environment the next important disadvantage of automatic memory deallocation and layout optimization (garbage collection) is the fact that it can happen anytime. Therefore, this can be source of unpredictable access violation exceptions and application crashes.

This means that passed array needs to be protected from garbage collection before it is passed inside of the library. This is a task that is handled either by P/Invoke mechanism (see section 2.11), pinning pointers (MC++ `__pin` keyword; refer to section 3.2), or by `GCHandle` structure (i.e., `System.Runtime.InteropServices.GCHandle`). Each of these approaches has its advantages and disadvantages that are described in subsection 8.2.6.

Similar problem is when **array** members are **reference types** or **callbacks**. This leads to the need of handling every array member separately and therefore it may cause significant slowdown. However, this is not case of OpenGL/GLU and therefore it is not handled by this work or by existing solutions.

To illustrate the problem a short **example** is included. Let us consider that an array is passed to the wrapped library and is stored inside of the library for later use. In order to allow non-managed code to access data stored inside of managed array, a pointer (non-managed one) has to be retrieved. Such pointer is then passed to the library.

However, let us consider this wrapped function call was made in a method, the passed array was allocated in the method locally, and the only reference to the array was stored into the local variable. After the method exits, the reference became invalid and therefore the array is a next candidate for garbage collection even thought there is reference in the form of non-managed pointer that is stored inside the library.

After an application invokes a function that uses stored pointer, three situations may occur:

- Function exits properly without any harm to the system, i.e., garbage collection has not occurred yet.
- Function exits properly but may cause memory corruption, i.e., garbage collection was performed and referenced memory block may contain different data structures of same application. This may cause an application crash (or raise an exception) in completely different part of the code due to some other data were corrupted.
- Function causes access violation, i.e., garbage collection occurred and referenced memory block is invalid (e.g., it is not longed owned by the application). This usually crashes the application.

6.2.2 Callbacks

Callback (or callback function) is construction used by interfaces to provide feedback to the user. In the case of this work, callbacks are used by some GLU functions. Callback itself may be implemented either by a **delegate** or by an **interface**. Its call from non-managed environment is handled by **P/Invoke** mechanism. For description of callback implementation, see section 8.2.8.

P/Invoke mechanism uses delegates and handles data marshaling, e.g., it is possible to convert non-managed pointer to managed array of given (static) size. The difficulty of callback use is that some callback functions allow user to pass so-called **user data**. These data are then passed as one of the callback parameters and may be used to reference user data structures. In such case there is a question whether to allow any **reference data types** (i.e., classes inherited from `System.Object`) or to allow just simple **integer data types** for user data.

First approach (i.e., allowing generic reference data type) is more comfortable for the user and does not need any additional structures create by the user because. All structures are contained within a library port and therefore they are transparent for the user. However, it causes slowdown due to maintaining of structures that are used for storing user data.

Second approach (i.e., allowing integers only) is less comfortable for the user. It causes need of additional data structures that are managed by the user. However, this approach minimizes slowdown due to the structures are handled by the user and therefore the user may choose the optimal implementation for particular case.

6.2.3 Void Pointers

Last major difficulty of the interface implementation is a **void pointer**. In non-managed environment, it is very useful language construction. A function that uses void pointer as a parameter can easily get various data structures via void pointer without need of explicit data conversion. The rest of function parameters provide description of passed data structure.

However, in non-managed environment there is no direct equivalent of void pointer. The closest construction to void pointer is use of `IntPtr` structure. This structure contains integer number that can be interpreted either as integer (32 or 64 bits) or as void pointer.

The major disadvantage of use of this structure is that its size is platform dependent, i.e., its size may differ on 32-bit and 64-bit machines. Therefore it may be source of difficulties, e.g., when working on 64-bit machine with a code that assumes address has length of 32 bits this may cause access violation due to narrowing of 64-bit value to 32-bit.

Important thing to note is that the use of `IntPtr` structure usually causes need of unsafe (non-verifiable) blocks of code. Such application then needs higher security permission in order to be executed (refer to section 2.8).

7 Existing Solutions

OpenGL is worldwide spread interface for graphical output and it is standardized. However, .NET Framework library lacks interface for high performance graphical output and therefore there were efforts to create port of OpenGL library to .NET environment. This section describes existing ports together with their major advantages and disadvantages. For timing comparison of this work and selected existing solution (CsGL) see chapter 1.

7.1 CsGL

The most spread port of OpenGL to .NET environment is called CsGL (for further information and downloads see [CsGL]). It has been developed for about two years; first version was released at August 14, 2001. It implements both OpenGL and GLU interfaces.

Interface of CsGL consists of static OpenGL/GLU functions that are grouped into few classes. However, due to inheritance between these classes, final class contains OpenGL, GLU, and GL Extensions functions.

OpenGL/GLU constants are implemented as static members of particular class. They are not grouped into enumeration data types due to static member of the class is the closest equivalent to C/C++ symbolic constants, i.e., `#defines`.

The major advantage of CsGL interface is that it is very close to original OpenGL/GLU interface, i.e., all functions are static and constants are not members of any enumeration data type. The fact, that all OpenGL/GLU functions and constants are encapsulated into single class shall simplify porting of already existing applications.

The recommended use of CsGL is via **inheritance**. This means that user has to derive class, which uses OpenGL/GLU, from CsGL class. Thanks to the inheritance, all functions becomes members of user's class and therefore it is possible to call them from such class without need of specifying their **qualified name** (see subsection 2.1.9). Resulting code is then similar to original OpenGL/GLU code.

Interface construction allows its use via **composition**. However, this means that **qualified name** of each function and constant needs to be specified. Therefore, this way of use is not recommended. For example of such code can be found at Figure 7.1. Example uses C# and it does the same as example at Figure 4.1, i.e., it draws white triangle on black background.

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

Figure 7.1: Example of CsGL use via composition

Disadvantage of such interface construction is that, the user cannot **separate** different OpenGL and GLU versions. However, because of interface's resemblance to original OpenGL/GLU interface is this disadvantage not significant.

CsGL itself is interface wrapper, i.e., it provides just connectivity between non-managed OpenGL/GLU library implementation and managed environment. It uses **P/Invoke** mechanism (see section 2.11) that allows cooperation of managed and non-managed code.

The major advantage of P/Invoke use is that simple automation is possible, i.e., it is possible to port new GL Extension or OpenGL/GLU version just by executing prepared script and/or application that is easy to create. Next advantage of P/Invoke mechanism is that it is possible to use it any programming language including C# or MC++.

However, its use may lead to somewhat slower code than in the case of IJW (see section 3.2) and may cause need of **unsafe** (i.e., not verifiable) blocks in application's code. This is also case of CsGL: code that uses CsGL needs unsafe blocks. The use of unsafe blocks may lead to faster code due to user's optimization of data manipulation, i.e., the user is the one who can decide whether and when protect data structures from garbage collector. However, it may allow unpleasant bugs due to use of pointers to appear in the same time and user needs knowledge about managed and non-managed code cooperation in order to work with the interface.

CsGL does not implement any parameter checking and it also does not contain any additional code in wrapper functions that may increase programming safety, e.g., it does not checks if there is available current OpenGL Render Context, it does not perform parameter checking, etc.

This may lead to unpleasant bugs similar to those of non-managed code that uses OpenGL. For example, user may call OpenGL function with no current OpenGL Render Context. The result of such function call depends on current OpenGL implementation of hardware drivers (see section 4.2). Therefore, it may vary from machine to machine.

CsGL is implemented using C# and C programming language. System dependent parts are written in C due to simplicity and possibility to create non-managed DLL. Such DLL is then used via P/Invoke mechanism. OpenGL/GLU functions and constants are implemented in C# via P/Invoke mechanism. As it is mentioned in CsGL documentation (see [CsGL]), this shall simplify mechanism of porting CsGL to another platform.

CsGL interface is full functional and is open source. It implements OpenGL version 1.1, 1.2, 1.3 and 1.4, GLU, and at least 90 OpenGL Extensions. This includes additional helper classes that allows operations with fonts, mouse cursors, etc.

It also includes tool to port new extension. User just needs C language header files for GL Extensions that can be downloaded from OpenGL site [OpenGL]. From this view CsGL is ready to use.

7.2 GLSharp

GLSharp is the next project that aims creation of OpenGL/GLU port for managed environment. Even thought the interface construction differs from CsGL, inner mechanism is the same. This means that GLSharp is an interface wrapper and uses P/Invoke. All OpenGL/GLU functions (constants) are static members of class but opposite of CsGL OpenGL functions (constants) are separated from GLU functions.

Currently this project is under development and therefore it is not complete. It lacks documentation and it does not follows naming conventions described in [CLI-V]. For more details on development and source code snapshots see [GLSharp]. Because the project (GLSharp) was inspired by CsGL and it uses same mechanism as CsGL, it is not used for timing comparison with this work.

8 Solution

This chapter describes interface implementation and design of this work. It contains examples how were specific parts of the solution designed and implemented. Even thought this chapter describes the solution it does not contain complete list of all structures and classes that were implemented (see Appendix A).

8.1 Interface Structure

This section describes interface of OpenGL library port. It explains reasons for using of particular approaches and specifies naming conventions used by this work. Even thought it contains implementation notes, it does not contain details of actual implementation and lists of all class members. Only important class and/or structure members are mentioned.

For details on actual implementation and solutions of difficulties refer to section 6.2 and section 8.2. For detailed description of all namespaces, classes, structures, and members see Appendix A.

Interface was designed to benefit from comfort of managed environment. Design was influenced by the fact that it shall maintain the highest possible compatibility with original OpenGL/GLU specifications ([GL13, GLU]). Interface is object oriented, however, due to specification compatibility issue it was not possible to further divide OpenGL/GLU functions into classes.

It is implemented as a **wrapper** for OpenGL/GLU functions, i.e., it needs binary form of the library in order to work. It is an approach similar to those of CsGL (refer to section 7.1), but it is the only one possible due to OpenGL is low-level and system dependent library.

Interface benefits from use of **namespaces**. All classes with a few exceptions of additional structures (classes) are member of single namespace. Additional structures as well as enumeration data types and callbacks are grouped into nested namespaces that will be described later.

Naming convention of nested namespaces containing items (classes, structures, enumeration types, and callbacks) that are owned by the particular class is based on a name of the owner class. The scheme for such name is then `XMembers` where `X` is name of the owner class, e.g., `GL11Members` contains items owned by `GL11` class.

Use of such namespace system leads to ambiguous item's identifier due to it allows modifying of existing items for newer OpenGL/GLU version without modification of

item's identifier. However, it suppresses need of names that contain version information (e.g., `RenderPrimitive11`).

Classes that are members of the interface implementation are **not thread-safe**, i.e., it is not possible to use them in multithreaded application without further synchronization primitives. The reason for this constraint is the fact, that synchronization of threads (i.e., critical sections and monitors) is quite CPU time consuming task and generic implementation of it is less effective then particular mechanism implemented by the user. It is good to note that even OpenGL is generally not thread-safe and therefore additional thread-safety would be addition to already fulfilled specification.

Interface implements OpenGL, GLU, and GL Extensions. It does **not** contain functions of **GLUT** [GLUT] due to its purpose. Purpose of GLUT is to isolate window system dependencies from OpenGL, i.e., it make possible for OpenGL to be independent on current windowing system. It also makes OpenGL applications to be portable on source code level, i.e., such application just needs recompilation on destination platform in order to run.

However, application that uses managed environment does not need unification of window systems because it already contains unified window system with support for window event handling that is simple enough. Eventual port of GLUT to managed environment will lead to something similar to classes that handles GUI (`System.Windows.Forms` namespace). Another issue is the question of efficiency, stability (newer version of GLUT is not stable enough on Windows platform), and difficulty of such interface use. Due to that, the port of GLUT library is not part of this work.

Interface consists from five major groups of classes:

- **System classes** that handle underlying GUI (window system) cooperation.
- **OpenGL classes** that handle OpenGL calls.
- **GLU classes** that handle GLU calls.
- **GL Extension classes** that handle GL Extensions calls.
- **Additional classes and structures** that are used to either store internal data or provide additional functionality and/or comfort.

8.1.1 System Classes

System classes are a fundamental part of the interface implementation. These classes handle underlying GUI cooperation and OpenGL initialization. The base class (i.e., remaining classes are derived from this class) is named `BaseRenderContext` (in the next text references as `RC`). This class wraps API (i.e., Win32API and GDI) of underlying windowing system and is independent on forms that are provided by .NET (i.e., `System.Windows.Forms` namespace). This allows derived classes to use either .NET GUI or native GUI of underlying windowing system whenever there is no .NET like GUI available (e.g., SSCLI; see [SSCLI]).

`RC` class provides initialization of OpenGL including possibility of automatic detection of available **bit depths** of **color** buffer, **depth** buffer, and **stencil** buffer. This automatic detection is used in the case the user does to set these bit depths manually (i.e., as parameters of `RC` class constructor). It also handles use of multiple `RC`s and switching between them. However, it does not handle thread synchronization tasks.

This class provides a reference to the **current** `RC` that is used for parameter checking. This reference is exposed in a form of static field (`CurrentRC`) and it is intended only for internal use (i.e., it has **Assembly** accessibility permission; see subsection 2.1.10).

Similar to current `RC` reference, it also contains reference to **internal data structures** (classes) used by OpenGL/GLU implementation. Naming convention of these references is **`X`**`Data` where **`X`** is name of owner class, i.e., class that owns these data. Even thought reference to them is stored in `RC` class, their allocation (i.e., instance creation) is not handled by the `RC` class.

The allocation of internal data structures is handled by classes that own these structures. The advantage of this is that it increases performance of `RC` class initialization and saves memory due to internal data structures of classes that are not used by the application are not allocated. Internal data structures and their references are intended for internal use only and therefore they are not exposed to the user, i.e., they have **Assembly** accessibility permission.

8.1.2 OpenGL Classes

OpenGL classes are classes that handle OpenGL functions calls, i.e., they encapsulates functions and constants (see subsection 8.1.6) of particular OpenGL version. Opposite of original OpenGL or CsGL, single OpenGL class instance belongs to exactly one instance of `RC` (in following text this instance will be reference as an **owner**), i.e., it is not possible to use similar OpenGL class instance with multiple instances of `RC` (owners).

The reason for that is that some OpenGL functions may needs internal data that are valid only for specific instance of `RC`. Reference to owner of OpenGL class instance is stored in protected field named `rc` and is used for parameter checking purposes.

Each **version** of OpenGL is encapsulated into separate class that is derived from class of previous OpenGL version. Exception to this is `GL11` class that contains OpenGL version 1.1 and that is a root of inheritance relations between OpenGL classes.

Naming convention of class that contains new version of OpenGL is then `GL`**`Mm`** where **`M`** is major version number and **`m`** is minor version number, e.g., class `GL11` implements OpenGL version 1.1. The reason for this is to allow separation of different OpenGL version in order to make possible for the user to use exactly the version that suits user's needs.

Every class may need **internal data structures**. These are used to store references to arrays or objects that are passed inside of OpenGL library and are stored there for later use, e.g., function `glVertexPointer`. Storing references (in managed meaning) shall prevent the array or the object from being garbage collected because the only reference passed inside OpenGL library is non-managed pointer (see section 6.2.1). These internal data structures also allow retrieving of managed references to arrays or objects that were passed inside the library.

Internal data structures are implemented in form of managed classes because non-managed class cannot have class members of managed types. Their name shall have form of **`X`**`InternalData` where **`X`** is a name of class that uses them, e.g., `GL11InternalData` contains internal data structures of `GL11` class. They are intended for internal use only. Therefore, they have **Assembly** access permission and are not visible for the user. There may exist only **one** instance per each `RC` instance.

Even thought these classes contain data for classes that inherit from a newer version, internal data classes are derived only from `System.Object` class similar to any other

managed class. This means that they are **not derived** from older version of internal data class. This approach shall simplify both construction and initialization of newer internal data class versions due to lack of copy constructors and complicated data copying. Therefore, this fact may lead to lower slowdown.

Reference to internal data structure is stored in protected field that is member of particular class. However, the approach described in previous text needs separated storage of these parameters for different versions. For `GL11` class (i.e., base class for all OpenGL classes) is the reference stored in field named `data`. Higher versions of OpenGL need to add additional fields for their own internal data. Such storage shall be named as `data` with suffix of version number, e.g., for `GL12` class (OpenGL version 1.2) this field shall be named as `data12`.

In a few cases, there may be need of **common internal data structure**, i.e., data structure that is shared across OpenGL versions. This shall solve differences of stored data handling between newer versions. Illustrative example can be problem of storage of reference to multiple texture coordinates used by multi-texturing.

Setting of such reference may be provided by the same function, which is used in OpenGL version that does not allow multi-texturing. Such function then needs to access common internal data structure in order to set reference properly because it is allowed to use instances of different OpenGL versions (i.e., classes that implements OpenGL versions) over single `RC` class instance.

Common internal data structure is managed class and is accessible only from inside of the assembly, i.e., it has **Assembly** accessibility permission. This class is named as `GLInternalData`.

To simplify use of OpenGL classes, there is added one special class. This class is derived from the latest implemented OpenGL version and it is named as `GL`. It shall simplify use of the interface, i.e., user does not need to know which is the latest implemented version.

8.1.3 GLU Classes

GLU class design is **similar to OpenGL**. This means that GLU class (in next text referenced as GLU) may have internal data structures (including common internal data structures) whose reference is stored in class members similar to OpenGL. Also a single instance of GLU may be used with the instance of `RC` that was used to create instance of GLU, i.e., similar to OpenGL, it is not allowed to use single instance of GLU with multiple instances of `RC`.

Similar to OpenGL, class that implements newer version of GLU inherits from previous version of GLU. Naming convention is very close to those of OpenGL: each GLU class is named as `GLU`**`Mm`** where **`M`** is major version number and `m` is minor version number (e.g., `GLU11` is name of class that implements GLU version 1.1).

Opposite of OpenGL, there exist special structures in GLU called **GLU objects** (e.g., `GLUquadric`). These objects are close to system handles (e.g., to open files). Only GLU functions may manipulate with these objects and user is not allowed to directly access their members. Currently (GLU version 1.3) there exist three GLU objects: `GLUquadric`, `GLUtesselator`, and `GLUnurbs`.

All these three objects are implemented by the same way. Each of these objects is encapsulated into separated object. These objects are managed and are derived from `GLUobject` due to all of these objects have similar properties. This approach then allows sharing of routines thanks to inheritance and simplifies future improvements.

Objects themselves are used to store **internal data** that cohere with these objects, i.e., they are similar to internal data structures that were mentioned above. Currently they are use to store references to callbacks.

Opposite of OpenGL, GLU needs **callbacks**. Callbacks are implemented in a form of delegates due to delegates are closest possible equivalent of function pointers that is available in managed environment. Names of GLU callbacks are similar to original.

Delegates are stored in GLU object in order to protect them from Garbage Collector. The next important reason is due to setting of a callback needs valid GLU object reference. This means that the callback belongs only to particular GLU object. Current implementation of GLU callbacks including user data handling is described in detail in subsection 8.2.8.

8.1.4 GL Extension Classes

GL Extensions classes are implemented similar to OpenGL classes. Each GL Extension is encapsulated into a single managed class and is owned only by single `RC`, i.e., it is not possible to use GL Extension class with different instance of `RC` then those that was current during initialization of GL Extension class instance.

The reason for this limitation is that GL Extensions are both hardware and current OpenGL instance dependent. Addresses of all GL Extension functions need to be retrieved in order to used them and this address is `RC` instance (OpenGL instance) dependent, i.e., it may differ for different OpenGL (`RC`) instances. GL Extensions functions implementation is described in subsection 8.2.3.

Even thought GL Extensions are implemented similar to OpenGL, there exists no inheritance similar to OpenGL versions. All GL Extension classes are derived from `Extension` class in order to allow sharing of common functionality such as routines for checking whether GL Extension is available.

Some GL Extensions may add or enlarge existing enumeration types in order to be used for OpenGL functions (e.g. `GL_ARB_imaging`) and they do not specify new functions. In such case, GL Extensions class also contains OpenGL function that uses new and/or modified enums as parameter types.

Naming convention of GL Extension class is based on original GL Extension name (see [Kil02]) and it uses “PascalCasing” (see [CLI-V]). This original name has form of `GL_`**`X`**`_`**`NAME`** where **`X`** indicates developer of this extension and is uppercase. **`NAME`** is actual name of extension, e.g., `imaging` for `GL_ARB_imaging`.

Class name is then created by removing `GL_` prefix and underscores. All characters become lowercase. Exception of this is `X` and each character that follows underscore in original name, such characters them become uppercase (e.g., `GL_ARB_imaging` transforms to `ARBImaging`). The result then follows naming convention recommendations (see [CLI-V]).

GL Extension classes are stored in nested namespace that is called `GLExtensions`. It is replacement for `GL` prefix, which was removed from original GL Extensions names. All modified and/or new enumeration types are grouped into namespace that is nested in `Extensions` namespace. Naming convention of such namespace is then equal to naming convention of similar namespace for OpenGL classes, i.e., **`X`**`Members` where **`X`** is name of a class that caused creation of grouped types (e.g., `ARBImagingMembers`).

8.1.5 Additional Classes and Structures

In order to improve comfort of OpenGL and to solve a few cases of void pointers, additional classes and structures were added. These classes/structures are not part of OpenGL standard specification even though several of them are based on OpenGL functions parameters. Currently there are three groups of these add-ons:

- exceptions,
- additional classes,
- additional structures.

Exceptions are used for exception handling. Currently there exist only one class, which is used for this purpose. This class is called `GLException` and it is derived from `System.ApplicationException`. The reason for its creation was an effort to make possible to distinguish exceptions that are raised by OpenGL implementation from exception raised by other utilities including CLR.

Additional classes are added in order to simplify initialization and handling of common events (e.g., window resizing, window redrawing). These classes are grouped into nested namespace `Forms` and shall be partial replacement for GLUT functionality. Example of these classes is `GLForm` class that simplifies creation of standalone window including improved event handling (see section C.2).

Additional structures increase programming comfort and shall replace use of arrays. The basic idea came out from the fact that `glInterleavedArrays` function parameters accept array of structures in a form of void pointer. Structure is then described by function parameters. Therefore, a logical step is to replace the void pointer with an actual array of predefined structures.

These structures are replacement for void pointers and simplify passing of data, such as color, position, transformation matrix, etc. They also avoid need of additional user structures that have to be created in order to improve source code readability.

In order to improve their practicability each structure have implementation of constructor that allows to fill all structure members with value that is either set by the user or copied from similar .NET Framework Library structure. E.g., it is possible to set up `Color3f` structure either by specifying value for all color components (red, green, blue) by hand or use value of `System.Drawing.Color` structure, which already offers many predefined colors (e.g., gold).

An illustrative example of additional structure use with comparison to original OpenGL interface can be found at Figure 8.1. Code in example sets diffuse color of a light to blue. Example uses C# syntax for managed version and C for original OpenGL function call. It does not contain specification of all parameters because they are not important in this case.

```
gl.Lightfv(..., Color.Blue);
```

```
GLfloat color[4];
...
color[0] = 0; color[1] = 0; color[2] = 1; color[3] = 1;
glLightfv(..., color);
```

Figure 8.1: OpenGL function call in managed environment that uses additional structures (top) in comparison with original C-style code (bottom)

Additional structures are grouped into nested namespace `Structures` in order to organize interface structure. Complete list of implemented structures can be found in section C.3.

8.1.6 OpenGL/GLU/GL Extension Methods, Constants, and Enums

Opposite of CsGL, all OpenGL/GLU/GL Extension **methods** (referenced as 'methods' or 'functions' in this subsection) are neither static nor virtual members of particular class. Methods are not static due to they may to access internal data structures and this design allows to add checking if an object is called with current `RC` similar to the `RC` that was current during creation of instance (e.g., `GL11`).

Even thought OpenGL specification may modify implementation of functions for newer versions in order to improve functionality (e.g., support for multi-texturing), methods that wrap functions are not marked as virtual. It is due to virtual methods are slower and it is possible to replace use of virtual methods by modifying of common internal data structures.

Naming convention of functions differs from original function name. This difference is caused by removing of 'gl' ('glu') prefix for OpenGL/GLU functions and removing of suffix that indicates GL Extension for GL Extension functions (e.g., `glWeightbvARB` is transformed to `Weightbv`).

This modification is because these prefixes (suffixes) shall prevent name collision due to lack of object structure in original OpenGL/GLU interface. In object-oriented environment are functions grouped into classes that prevent name collision of class members with members of another class.

Next reason for this modification is that it improves source code readability. For comparison of code both with and without removed prefixes ('gl') see Figure 8.2. Example is illustrative and its functionality is similar to example at Figure 4.1.

```
gl.glClearColor(0, 0, 0, 0);          gl.ClearColor(0, 0, 0, 0);
gl.glClear(clearMask);                gl.Clear(clearMask);
gl.glColor3d(1, 1, 1);                gl.Color3d(1, 1, 1);
gl.glVertex3d(-1.0, -1.0, 0.0);       gl.Vertex3d(-1.0, -1.0, 0.0);
gl.glVertex3d(0.0, 1.0, 0.0);         gl.Vertex3d(0.0, 1.0, 0.0);
gl.glVertex3d(1.0, -1.0, 0.0);        gl.Vertex3d(1.0, -1.0, 0.0);
gl.glEnd();                           gl.End();
gl.glFlush();                         gl.Flush();
```

Figure 8.2: Example of code with (left) and without (right) 'gl' prefix

Constants are implemented as static read-only fields. However, their use is only for exceptional cases because all constants are grouped into enumeration types. Names of constants are similar to original OpenGL in order to be close to OpenGL specification. This approach also solves problem of names that would not be valid if `GL_` prefix would be removed, e.g., `GL_2_BYTES`.

Enumeration types (enums) are members of nested namespaces of particular class (e.g., `GL11Members` namespace for `GL11` class). They replace constants in order to increase programming safety (i.e. it shall prevent user from passing invalid value) and programming comfort due to enumeration type describes values that are available for particular parameter.

Names of enumeration types shall follow recommendations described in [CLI-V]. However, in a few cases, there is need to use possible unusual abbreviations. These abbreviations shall minimize name collision with identifies of .NET Framework Library.

When minimizing name collision, there shall be emphasis on `System.Drawing` namespace due to this namespace is used at the most in windowing application, e.g., `PxlFormat` is used instead of `PixelFormat` due to it collides with an enumeration type of the same name that is member of `System.Drawing` namespace.

Naming convention of enumeration type members is similar to naming convention of constants with exception of constants that belong to particular GL Extension. In such case extension developer suffix is stripped, e.g., `GL_TEXTURE0_ARB` become `GL_TEXTURE0`.

8.2 Implementation Details and Difficulty Solution

This section describes in detail implementation issues and solutions of difficulties described in section 6.2. Descriptions contain examples of source code. These examples use MC++ syntax and are similar to templates used to solve particular implementation issue. It also discusses possible solutions including description of their advantages and disadvantages. In order to make these examples simple and short, the source code may not be complete, i.e., parts that are not important to particular issue are eliminated.

8.2.1 Programming Language

CLI (.NET Framework) allows sharing of compiled code (CIL) between languages, i.e., it is possible to implement managed library in whatever programming language whose compiler aims .NET. The language that was selected for this work is MC++. Its brief description can be found in section 3.2.

The advantage of this language (opposite of C#, Visual Basic .NET) is that it allows mixing managed and non-managed code very easily. It supports IJW (It Just Works; see section 2.11) mechanism that is quite powerful. It allows wrapping non-managed libraries very easily and it shall be aimed on performance at the same time, i.e., it shall minimize managed and non-managed code cooperation overhead.

It allows to mix managed and non-managed code very easily, therefore there was a high probability to unify language used for whole solution, i.e., not to mix various programming languages even thought .NET itself allows that and it is its important feature. This unification simplifies developing of implementation and its compilation.

8.2.2 Function Wrapper

Function wrapper is a method of particular class. It wraps OpenGL/GLU functions that are available in non-managed OpenGL/GLU library (e.g., `opengl32.lib` for OpenGL on Windows platform) without need of retrieving their address. The wrapper contains checking for current instance of `RC` and checking of selected parameters.

For illustration, there is an example at Figure 8.3. It wraps `glBegin` function. It is an example of function that has single parameter that is neither array nor structure and does not return any value. All other functions, including those with array and/or structure as parameter type, have similar structure even thought they contain additional parts (e.g., data sharing mechanism, temporary storing of return value, etc.)

```
void Begin(GL11Members::RenderPrimitive mode)
{
    ...
    /* parameter checking */
    ...
    ::glBegin((unsigned int)mode);
};
```

Figure 8.3: Simple function wrapper example

8.2.3 GL Extension Function Wrapper

GL Extension function wrapper is very close to function wrapper described in previous section, i.e. it contains similar parts. It wraps functions that are not contained within non-managed library, i.e., functions whose starting address needs to be retrieved. Therefore, an essential part of GL Extension function wrapper is **pointer** to original function.

Both pointer type and pointer itself are non-managed and are **protected** members of particular class. Naming convention for a pointer type is similar to those defined in GL Extensions header file. This means that it is based on original function name including 'gl' prefix and developer suffix with all characters uppercase. Such name is then completed with 'PFN' prefix and 'PROC' suffix (e.g. pointer type for GL Extension function `glSampleCoverageARB` is named as `PFNGLSAMPLECOVERAGEARBPROC`). Naming convention for pointer (to particular function) consists of 'pfn_' prefix and string that similar to pointer type identifier, e.g., `pfn_PFNGLSAMPLECOVERAGEARBPROC` is identifier of pointer to `glSampleCoverageARB` function.

Example of this wrapper can be found at Figure 8.4 and it contains a wrapper for `glSampleCoverageARB` function. This function is member of `ARB_multisample` GL Extension.

```
protected:
typedef void (APIENTRY * PFNGLSAMPLECOVERAGEARBPROC) (...);
PFNGLSAMPLECOVERAGEARBPROC pfn_PFNGLSAMPLECOVERAGEARBPROC;
public:
void SampleCoverage (Single value, GL11Members::Bool invert)
{
    ...
    /* parameter checking */
    ...
    pfn_PFNGLSAMPLECOVERAGEARBPROC(...);
};
```

Figure 8.4: Simple GL Extension function wrapper example

8.2.4 Additional Structures

Additional structures are added to the interface in order to improve programming comfort. They are managed value types and therefore their use may lead to better performance. These structures need special memory layout due to they are accessed from non-managed code.

Their members shall have **sequential** layout and shall be aligned on exactly **one byte boundary**, i.e., in the memory they shall be laid in the same order that was used for their declaration with no gaps between each other. This layout can be set up using `StructLayout` attribute (see Figure 8.5 line 1).

Structure members (fields) have **public** accessibility and they are accessible directly (i.e., there are not properties). This direct access is violation of recommendations [CLI-V] but it may lead to better performance due to missing accessor (getter and setter) functions.

Each structure contains **constructor** that allows filling of all members at once. Simple additional structures that have already existing equivalent in .NET Framework library also contains **additional constructor** (e.g., see Figure 8.5 line 4) and **implicit type conversion** operator overload (e.g., see Figure 8.5 line 6). These additional members shall improve comfort and allow construction similar to a function call at Figure 8.1 (top).

Example of such additional structure definition can be found at Figure 8.5. This structure (`Vertex2f`) is similar to `PointF` structure of .NET Framework Library and is used in `glVertex2fv` function and structures for `glInterleavedArrays`.

```
1:      [StructLayout(LayoutKind::Sequential, Pack=1)]
2:      public __value struct Vertex2f {
3:          Single x, y;
4:          Vertex2f(PointF pnt) { x = pnt.X; y = pnt.Y; };
5:          static Vertex2f op_Implicit(PointF pnt) { return Vertex2f(pnt); };
6:      };
```

Figure 8.5: Example of additional structure definition

8.2.5 Enumeration Types

Enumeration types are managed value types. They are similar to non-managed C++ enumeration type. The underlying type of enumeration type is set to `Int32` similar to recommendations [CLI-V]. Example of enumeration type definition can be found at Figure 8.6. Preprocessor directive (`#undef`) used in example prevents from name collision of enum members and predefined symbolic constants.

```
public __value enum NewListMode : Int32 {
#undef GL_COMPILE
    GL_COMPILE = 0x1300,
#undef GL_COMPILE_AND_EXECUTE
    GL_COMPILE_AND_EXECUTE = 0x1301,
};
```

Figure 8.6: Example of enumeration definition

8.2.6 Data Sharing

Data sharing between managed and non-managed code is essential part of the interface. Approach that solves data sharing depends on purpose of data. There are five kinds of data purposes:

- **value types (struct)**: These are data that are passed as input value. Their value is copied and used similar to local variable. There is no need for garbage collection protection and it is possible to retrieve non-managed pointer without any additional code. Structures used by this approach shall be value types and have sequential memory layout with all members aligned to one byte boundary, i.e., these structures shall be additional structures described in subsection 8.1.5.

  It is faster the fastest approach possible and illustrative example can be found at Figure 8.7. Example is function wrapper of `glVertex2fv` and provides an example of additional structure use.

```
void Vertex2fv(Struct::Vertex2f v)
{
    ...
    /* parameter checking */
    ...
    ::glVertex2fv((float *)&v);
};
```

Figure 8.7: Value type (structure) use example

- **output value type**: These data are value types that are used as output parameters, i.e., function sets its value that is then passed outside the function. Its handling is similar to value type. Data sharing is handled by additional local variable that is non-managed type and that is used as a temporary storage.

  This approach is faster then use of arrays and comfortable for user. An illustrative example is at Figure 8.8. Example is a function wrapper for `glGetDoublev`.

```
void GetDoublev(GL11Members::StatusPname pname, [Out] Double &params)
{
    GLdouble d_params;
    ::glGetDoublev((unsigned int)pname, &d_params);
    params = d_params;
};
```

Figure 8.8: Example of value type used as output parameter

- **return value**: These data are passed out of the function as return value. Usually is this return value a value type. In such case is return value either returned directly from a function (e.g., `return ::glGenLists(...);`) or if there is additional code for data sharing, the return value is stored to local variable and then returned. Figure 8.9 shows example of the last mentioned approach.

```
Byte AreTexturesResident(...)
{
    ...
    /* parameter checking */
    /* data sharing preparation */
    ...
    Byte ret = ::glAreTexturesResident(...);
    ...
    /* data sharing finalization (e.g., releasing of handles) */
    ...
    return ret;
};
```

Figure 8.9: Example of return value handling

  In a few cases (e.g., `glGetString`) is not the return value a value type that is usually a pointer to zero ended string. Each of these cases needs to be solved individually depending on meaning of return value.

- **arrays not stored inside**: These are data passed in the form of array of value types (including structures) that are used only once, i.e., they are not stored inside OpenGL library (non-managed code) for later use. This fact allows use of faster approach (e.g., pinning pointers) in order to protect these data from memory layout optimization done during garbage collection.

- **arrays stored inside**: These are data in a form of value type arrays that are passed and stored inside OpenGL library (non-managed code) for later use. They need to be protected from both memory layout optimization and garbage collection due to they

may loose all their references from application. Implementation requires use of either internal data structures or common internal data structures.

Arrays need special handling for data sharing between managed and non-managed code. There is a need to protect these arrays from Garbage Collector. It is possible to solve sharing of arrays in three ways:

- **P/Invoke mechanism**: This approach is supported directly by managed environment and it allows creation of ”link” to non-managed (static) function in external DLL library. This ”link” includes routines for data sharing that are transparent for the user. Due to it is supported directly by managed environment and therefore it is possible to use it in majority of .NET language.

  Use of this mechanism is declared by `DllImport` attribute and example can be found at Figure 8.10. This approach is slower than pinning pointers and allows use of type arrays only, i.e., it is not possible to use instance of `System.Array` with this approach even thought it is an array of value types (e.g., `Double`).

```
[DllImport("OpenGL32.dll", EntryPoint = "glTexImage1D")]
static void TexImage1D(..., [In, Out] Int32 pixels __gc []);
```

Figure 8.10: P/Invoke mechanism use example

  The only use of P/Invoke mechanism is for functions that set up GLU callbacks. It is used because of it solves calling of callback functions and it is quite simple.

- **pinning pointers**: This approach uses pinning pointers that prevents memory block from being moved by garbage collector and from their garbage collection. Primary purpose of pinning pointer is to retrieve non-managed (`__nogc`) pointer to an array.

  Use of pinning pointers together with IJW mechanism provides better performance than codes that use P/Invoke mechanism. However, pinning pointers have certain limitations: it is possible to use them only as local variables and it is not possible to create pinning pointer for general array (i.e., instance of `System.Array`). Therefore are pinning pointers used only for **arrays** that are **not stored inside** non-managed OpenGL library.

  The advantage of pinning pointers is that they provide automatic releasing of pinned objects during exiting of a scope (e.g., function) in which they were declared. This leads to simple and short source code. Example of pinning pointer use can be found at Figure 8.11.

```
void TexImage1D(..., Single pixels __gc [])
{
    ...
    /* parameter checking */
    ...
    float __pin *ptr_pixels = &pixel[0];
    ::glTexImage1D(..., (GLfloat *)ptr_pixels);
};
```

Figure 8.11: Pinning pointer use example

- **GCHandle**: This approach provides more functionality then pinning pointers, e.g., it has support for weak pointers (see subsection 2.5.3), etc. It is applicable only to **blittable types**, i.e. types that have static and well-defined memory layout (e.g., built-in value types, value structures with sequential layout, arrays of those value types).

  In order to protect itself from pinning of invalid type, it performs runtime checks. This allows using `GCHandle` for general arrays (instance of `System.Array`). However, these runtime checks lead to slower code. Such code can be then significantly slower

then code that uses P/Invoke or pinning pointers, but it is the only way for handling of **arrays** that are **stored inside** OpenGL library for later use.

Disadvantage of this approach is that it needs manual releasing and it does not allow operating with references that are not initialized, i.e., `null` reference. Therefore, use of `GCHandle` leads to additional code in a wrapper function. Example of `GCHandle` use can be found at Figure 8.12. Example also illustrates possible solution of void pointer problem.

```
void  TexImage1D(...,System::Array __gc * pixels)
{
    GCHandle gcHandle_pixels;
    if (pixels != NULL)
        gcHandle_pixels = GCHandle::Alloc(pixels, GCHandleType::Pinned);
    ::glTexImage1D(..., (void *)((pixels == NULL) ?
        NULL : gcHandle_pixels.AddrOfPinnedObject().ToPointer()));
    if (pixels != NULL) gcHandle_pixels.Free();
};
```

Figure 8.12: GCHandle usage example

8.2.7 Void Pointer

Void pointer is a difficulty that was described in subsection 6.2.3. It is useful for specifying a parameter of general array or structure type. There is no direct equivalent in managed environment, but it is possible to replace void pointer in three ways by using:

- **IntPtr**: This approach is very close to void pointer in non-managed environment. However, the disadvantage of this approach is that it requires the user to handle protection of data structures from Garbage Collector. Next disadvantage is that it needs unsafe blocks in an application or code that uses it.

  Advantage of this approach is that it may lead to faster code. However, due to disadvantages mentioned in a previous paragraph, this work does not use `IntPtr` structure as replacement for void pointers.

- **general array**: This approach is based on a fact, that void pointer is often used for arrays with unknown array member type. Therefore a possible replacement for void pointer is general array in a form of `System.Array` class instance due to all arrays are derived from this class.

  Disadvantage of this approach is that it requires use of `GCHandle` in order to protect the array from Garbage Collector. This and the fact that there is implicit type casting to `System.Array` may lead to slowdown. However, it is possible to use this approach for array of any value type (i.e., built-in value type and structures) and of any rank (i.e., it is possible to use it for 1D arrays as well as for *n*D arrays where *n* is rank of array).

- **overloading**: This approach means to create overloads for all possible combinations of array member types and structures that can be passed for void pointer. It is useful only for **arrays** whose references are **not stored inside** of OpenGL library (non-managed) for later use due to it allows to use of pinning pointers in order to gain performance. It is possible to automate creation of these overloads. It also causes limitation of possible array member types and therefore it disables invalid types from being used.

  However, this limitation is major disadvantage of this approach is that it is not possible to create all combination of function wrapper parameters that are replacement of void pointer due to an array may have larger number of ranks.

8.2.8 Callbacks

Callbacks (or callback functions) are special case functions. Similar to void pointer it is a difficulty described in subsection 6.2.2. They are used in GLU to provide user feedback for situation that may occur.

Callbacks are implemented in a form of **delegate** because delegate is the closest equivalent to non-managed function pointer. The implementation uses P/Invoke in order to set up callback in non-managed GLU library and callback itself is stored to GLU object. The reason for use of P/Invoke is that it provides simple and effective way of handling callbacks, it does not need any additional structures and non-managed classes, and it simplifies creation of callbacks that have no user data parameters.

The major difficulty of callbacks is that it allows the user to pass own data inside the system. Such data are then used as input parameter of a callback in order to identify the caller of the callback. Therefore, the user can reuse single function for multiple callbacks. In non-managed environment, void pointer type is used for user data parameter type. This allows the user to pass various data types.

However, in managed environment there is no such possibility. The available solution is either to provide mechanism that is transparent for the user and that allows the user to pass generic object (i.e., `System.Object`) as instance or to use integers (i.e., `Int32`) instead as an index to storage structures (e.g., array, collection).

Transparent mechanism approach (i.e., first approach) is comfortable for the user because together with the fact that all objects are inherited from `System.Object` it creates a situation similar to non-managed code. However, this approach needs additional and complicated data structures including additional code in order to handle them.

It also leads to two layered callbacks, where the first layer prepares user data and then call user supplied callback (second layer). It may lead to significant both memory and CPU time overhead and therefore it is not used by this approach.

Opposite of that, approach that uses **integers** instead of `System.Object` provides better performance. It is based on a fact, that these callbacks are used for tessellation operations in GLU. There is a high probability that the user would create own data structures in order to store value. Therefore, it is reasonable to allow the user to pass index to these data structures instead of content. This approach leads to simple implementation and much better performance then the first one.

8.2.9 Parameter Checking

Parameter checking is included in interface in order to improve programming safety. It is aimed on debugging purposes and it uses C/C++ preprocessor directives (e.g., `#ifdef`, `#endif`) in order to simplify their disabling during compilation. If a parameter check fails, it raises an exception.

Parameter checking is implemented in a form of C/C++ macros (i.e., it uses `#define` directive). The reason for this approach is that a code, which performs the checking, is quite short and therefore placing it into will lead to slowdown. In addition, the implementation benefits from features of MC++ macros, i.e., it is possible to use parameter name for both referencing to a parameter and creating of exception message string that contains such name.

Example of code with parameter checking can be found at Figure 8.13. Parameter checking is capable to perform these six kinds of tests:

- **Checking of current RC**: It prevents a method to be executed if current active instance of `RC` class differs for owner (i.e., instance of `RC` that was used to create particular OpenGL/GLU/GL Extension class instance). This macro is named `CheckCurrentRC`.
- **Checking for `null` param**: It prevents from passing reference that is not initialized (i.e., it has `null` value). It is used for GLU objects and the macro is called `CheckNull`.
- **Checking for validity**: It prevents from passing of invalid (i.e., not initialized or already released) GLU object. Macro is called `CheckValid`.
- **Checking for value types**: It prevents the user from passing a general array with array members that are not value types. Macro is called `CheckValue`.
- **Checking for required size**: It prevents the user from passing array of invalid (e.g., to small) size. Macros are called `CheckSize` and `CheckByteSize`.
- **Checking for given value**: It prevents the user from passing either invalid value (macro `CheckParamValue`) or invalid enumeration value (macro `CheckEnumValue`). In a case of invalid enumeration value, it is used to prevent the user from passing an invalid description for parameter value.

```
    ...
#ifndef NO_CORRECT_RC_CHECK
    CheckCurrentRC(params);
#endif /*NO_CORRECT_RC_CHECK*/
#ifndef NO_PARAM_CHECK
    CheckEnumValue(format, GL11Members::InterleavedArrayFormat::GL_V2F);
#endif /*NO_PARAM_CHECK*/
    ...
```

Figure 8.13: Example of parameter checking

9 Automatic Conversion

This chapter briefly describes design and implementation for a tool that was created during implementation of OpenGL function wrappers. Even thought this tool is part of this work its creation was not requested. Therefore, only a short description is included.

Purpose of this tool is described in section 9.1. All other sections contain just brief overview. Due to the tool is not essential part of the work detailed lists, descriptions, and user manual, are part of appendices (see Appendix A).

9.1 The Goal

Creating of function wrappers is quite monotonous task and therefore it would be good to have a tool that would create either complete wrappers or templates for wrappers. This is also goal of this tool. Original motivation is based on the fact that there exist too many GL Exceptions (100+) and without this tool each extension have to be wrapped completely by hand. The tool shall simplify creation of wrapper functions and enumeration data types that are based on existing constants.

The tool shall be able to read ANSI C header files provided for OpenGL (i.e., `gl.h`, `glu.h`, and `glext.h`) and extract stored items (i.e., functions, constants, etc.). Design and implementation shall be as generic as possible. However, this tool is not intended to consume neither generic ANSI C code nor generic ANSI C header files because it is helper tool created just for purposes of this work. It also shall be able to store data into human readable files in order to allow manual modifications and simplify reuse of extracted information.

9.2 Design

This section contains overview of basic ideas of tool's design. It mentions only major ideas and properties of design, for complete list of classes including their relations see section F.1. Tool can **read** information from either data file or **ANSI C header files**, i.e., `gl.h`, `glu.h`, and `glext.h`. It is the most important part of the whole tool. Flow of data and important components of reader can be found at Figure 9.1.

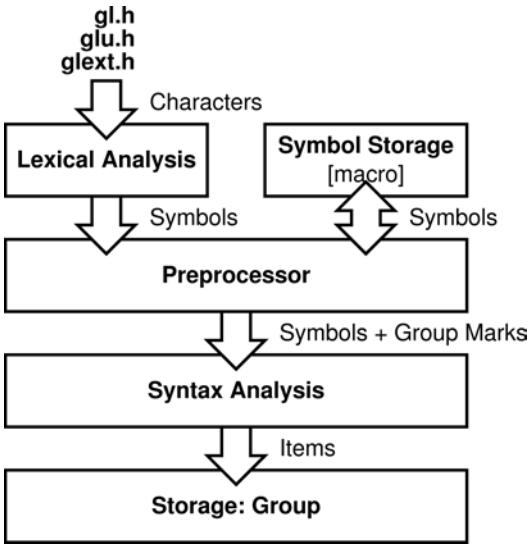


Figure 9.1: C Header file reader data flow scheme

Due to simplification of implementation `glu.h` header file have to be modified. This modification simplifies extraction of functions. It consists of definitions of particular symbolic constant and modification of function's headers: symbolic constant `WINGDIAPI` **has to be added** in **front** of every function header. Thank to the nature of GLU header files, it is possible to perform this by simple Cut/Paste All operation. The result shall be similar to function headers of `gl.h` header file.

Tool provides simplified version of C-style preprocessor in order to read ANSI C files. It also allows macro expansion even thought none of OpenGL header files uses it. Preprocessor does not handle evaluation of `#if` conditions and assumes them to be `false`. It also assumes symbolic constants prefixed with '__' (double underscore) to be not defined in order to unify the approach, i.e., there is no need for special handling of construction at the beginning of the header file and it allows to mask out all C++ stuff.

Preprocessor also assumes a few symbolic constants to have **default values**. These constants are `WINGDIAPI` (=`extern`), `APIENTRY` (= empty), and `GLAPI` (=`extern`). They have to be defined in every read header files. Their presence simplifies item extraction from header files because every function header has to be prefixed with either `WINGDIAPI` (for `gl.h`) or `GLAPI` (for `glext.h`).

All items are grouped into **groups**. A group consists of items that are enclosed into `#ifndef-#endif` blocks. This construction is based on format of file that contains GL Extensions: all GL Extensions are contained within single header file (i.e., `glext.h`) and all items of single GL Extension are grouped into a block that is enclosed by `#ifndef-#endif` preprocessor directives. Each group then contains constants, headers for functions, and function pointer definitions. Whole file is enclosed by a **root group**. While groups are assumed GL Extensions, root group is not.

Due to the fact, that GL Extensions are contained within single file and are enclosed by preprocessor directives it is not possible to filter out these directives similar to common C/C++ preprocessor. Therefore, preprocessor provides special set of symbols that represents the beginning and the end of such block (i.e., group marks).

Extracted data are stored into structures and classes. There is a single class for each language item (e.g., enums, functions, etc.) and such class is not derived from any .NET Framework Library class but `System.Object` class. Inheritance of these classes is based on their common functionality, e.g., function and pointer to function shares similar set of routines. For complete list of such classes see section F.1.

If such language item can be used as a type (e.g., structure, enums, etc.) then it has to implement `IType` interface. This interface contains set of methods that provides information about kind of a type (e.g., array, structure, etc.) and performs conversion between non-managed (C) and managed (MC++) syntax.

Exporting and storing is quite simple. It is similar to copying of existing structure to defined format (i.e., data file or MC++ source). MC++ source format is based on examples provided in chapter 1.

Data can be stored into a data file that contains human readable structure in form of valid XML, which is exact copy of existing relations between instances of objects (references are replaced by names). XML tags used in this file are described in section F.3. Description includes list of their attributes and values. It is good to note that backward references are **not** allowed, e.g., it is not possible to use type that was not defined before. This limitation is based on properties of OpenGL header files and it simplifies implementation of file's reader.

9.3 Implementation Notes

Event thought the design allows this tool being generic, implementation does not. It is because this tool is created as a helper tool aimed on simplification of function wrapper and enumeration types creation. Implementation follows ideas described in previous section. Therefore, this section contains just brief overview of interesting parts.

The tool is implemented in C# due to comfort of both the language and managed environment. It uses finite machine in order to perform **lexical analysis** of ANSI C header files. Finite machine was create by application **flex** [Flex] using modified ANSI C grammar. Modifications enlarge set of symbols by preprocessor directives in order to allow their processing. For modifications see section F.2.

Implementation provides simple **GUI** that allows the user to operate the application by mouse clicking. Descriptions of control's meaning including usage notes can be found in section F.4.

9.4 Output

Tool allows exporting of data in form of MC++ source file. It generates multiple header files in order to improve source code organization. Naming of these particular header files is based on a name of a generated class (e.g., `GL11`) with suffix that identifies content of such header file. Tool then generates files:

- **main source file**: This file contains constructor implementation (e.g., `GL11.cpp`).
- **main header file**: This file (e.g., `GL11.h`) contains class definition and includes majority of other generated header files. It is the only file, which needs to be modified whenever user modifies generated templates.
- **enum**, **constants**: These files contain definition of enums (e.g., `GL11Enums.h`) and constants (e.g., `GL11Constants.h`). Constants that are grouped into enums are **excluded** from compilation. In order to include them back a symbolic constant `INC_CONST_IN_ENUMS` needs to be defined in main header file.
- **functions**: This file (e.g., `GLU11Functions.h`) contains wrapped functions. Each function is enclosed into `#ifndef-#endif` block in order to allow for the user to mask out functions whose implementation was done manually by simply defining of

particular symbolic constant. Such symbolic constant then have form of `FN_X_Y`, where **`X`** is name of class and **`Y`** is name of method (e.g., `FN_GLU11_GetString`).

- **delegates**: This file (e.g., `GLU11Delegates.h`) contains definition of delegates, i.e., managed version of function pointers. If there are not delegates present, this file is not generates. This file is not included by generated main header file due to use of delegate needs additional code that has to be generated manually.
- **structures**: This file (e.g., `GLU11Structures.h`) contains definition of generated structures. Similar to file containing delegates is this file generated only if there are some structures present and is not included by generated main header file due to similar reason.

10 Verification and Validation

This chapter describes verification and validation of interface implementation. It explains its designs and reasons for particular steps. However, this chapter does not contain complete class reference. It also does not contain user manual and class reference for tool that was designed in order to provide at least particular automation of this task. These topics are covered in Appendix A.

Verification and validation of this work is quite important task because it could prove whether is the result usable and whether it supports features described by OpenGL specification. From this viewpoint it is possible to divide this task into two parts: validation of interface design and verification of interface functionality.

10.1 Design Validation

Interface structure that is used by this work differs slightly from original OpenGL specification because it attempts to use object-oriented approach. Due to that it is not possible to validate structure design by simple comparing it to OpenGL specification. Possible approach for this task is to prove whether is this object-oriented design useful in praxis and easily extendable.

The question of simple **extendibility** is one of the most important ones because this implementation is a **wrapper**. This means that primary development is done in non-managed environment using different language and therefore every new version needs additional work in order to create wrappers.

It seems that the best way, how to gain answer for extendibility question, is to implement all available versions of OpenGL and all available GL Extensions. However, task of port creation is not easy to automate due to design decisions consequent upon goals of this work (see chapter 1). Therefore creating of such large number of OpenGL/GL Extension classed would take too long even thought this work provides a tool (see chapter 1) that allows automation of at least parts of this task. Due to these reasons, the extendibility question is solved only upon theoretical basis.

Second part of design validity question has importance similar to extendibility. It is question of **usability**. The answer shall prove whether expecting improvement of programming comfort and safety does not mean too complicated source code.

The answer for this question is matter of examples implementation. Then, these examples shall be compared to either their original (non-managed) versions or CsGL versions

whenever possible. This shall prove whether does the usage of this work lead to code that is more complicated and whether is its usage comfortable as it is expected.

Disadvantage of that is the fact that it cannot be done by author of this interface himself due to his knowledge about intended usage of the interface. It is matter of getting feedback from the users that are familiar with original OpenGL interface.

10.2 Functionality Verification

Second important thing that needs to be proved is **functionality**. It is a question whether the functionality (i.e., behavior) follows specifications and therefore whether the implementation can be marked as **certified**. However, it is good to note that this implementation is a wrapper. Therefore, the output depends on underlying binary OpenGL library, device drivers, and hardware capabilities. It could then happen that output (behavior) is different even thought the code inside wrappers follows specification.

It is true, that it is possible to distinguish between difference caused by incorrect implementation and difference caused by invalid combination of drivers and hardware. However, it needs additional knowledge, i.e., it has to be sure that either combination of drivers and hardware is valid or implementation is correct. Therefore, it is possible to create set of tests that could be used whenever either new driver/hardware was installed or implementation was modified, e.g., a new version required modification of existing code or port to new platform was created.

Such **test** is then based on comparing of generated image and reference image. Reference image is an image that was created by the same test on trusted hardware and that was then checked visually whether it is equal to expected image. For testing purposes, only **still images** are used. Use of animations would lead to complicated framework and the results would be the same because animation is set of still images. They would be useful only for performance testing purposes, i.e., to test how much frames per second can the system generate.

Both generated and reference image have to be generated with **same resolution** and **color bit depth**. Images that uses palette are not allowed due to possible difference in optimized palette generation. Reference images are not stored using loss compression algorithms (e.g., JPEG) in order to minimize differences based on different storage approaches.

Comparison is then performed on per pixel basis by comparing of color components, i.e., red, green, and blue. The result of this comparison is then value of maximum of all founded differences, mean of these differences, and standard deviation of these values. For computation purposes a size of difference is used (i.e., absolute value) and the difference is scaled to range of $\langle 0.0; 1.0 \rangle$. Results are then compared with user's selected thresholds. Test fails whenever one of values (i.e., maximum, mean, or standard deviation) is greater then given threshold.

Design and **implementation** of single test is aimed on isolation test itself from common tasks that needs to be handled for every application (e.g., OpenGL initialization, window event handling). Each test is standalone (console) application or assembly that contain **exactly one** class that is derived from `TestBase` class. This class is an abstract class that handles common tasks and prepares an environment for automatic processing of tests and test results. For class member list and test application framework see Appendix A.

Thanks to that, developing (debugging) of the test is simple (i.e., it can be executed as standalone application) and **automation** is possible. It also simplifies enlarging of existing set of tests: it is just matter of copying test application executable/assembly to specified directory and modifying text configuration file (see section D.3). After that, the new test

can be processed by simple tool (see section D.4) that uses reflection mechanism for creating instances of dynamically loaded types.

Currently there are two tests available: texture object test and pixel read test. These tests are base on personal experience of author with various combinations of hardware and drivers. However these tests are rather examples due to there are too many combinations of hardware and versions of drivers. Also the fact that many problems depends on particular situation and such problems usually appear in complex applications, make nearly impossible to reproduce the behavior without knowledge of application's source code and situation in which did it occurred.

11 Results

This chapter contains results in a form of performance comparison between selected implementation, non-managed version, and implementation provided by this work. Sections of this chapter contain only results and comments, for source code of particular test and outputs refer to Appendix A.

These tests shall provide a possibility to compare different approaches to data sharing because data sharing is crucial part of interface implementation. First, different approaches to data sharing are compared. This comparison is done between CsGL, this work without parameter checking, and this work with parameter checking.

It is assumed that the version with parameter checking would produce the worst results (i.e., highest slowdown). All measured time intervals are referenced to results of non-managed version of this test. In this case, results are referenced to Win32 version of particular test.

In order to provide similar environment for time measurement, all three versions of the tests uses Win32API function `QueryPerformanceCounter`. It is used to measure the time interval and in managed environment is this function accessible via P/Invoke mechanism. This function offers high resolution that usually depends only on speed of actual CPU.

It is good to note that all results **depend on status** of both hardware and underlying operating system because most of OpenGL's facilities are implemented in hardware. Therefore, the results may vary even for high number of test repeats. None of tests provides measuring of initialization and startup routines performance because these routines depend heavily on status of operating system and they are usually called only during application startup.

All tests are repeated for various numbers of calls (or objects) in order to check whether is the implementation dependent on number of calls. It is assumed that for ideal situation the slowdown shall be the same no matter the number of calls or objects. In order to allow the user to check this behavior, results are displayed in a form of **graphs**.

Horizontal axis of single graph contains number of calls or objects. This number is considered **parameter** for particular test function. It is good to note that horizontal axis has **logarithmic** scale because it allows comfortable display of values that differ quite significantly.

Vertical axis then contains measured results (time intervals). However, only relative values are displayed because of they can be compared between each more easily. These

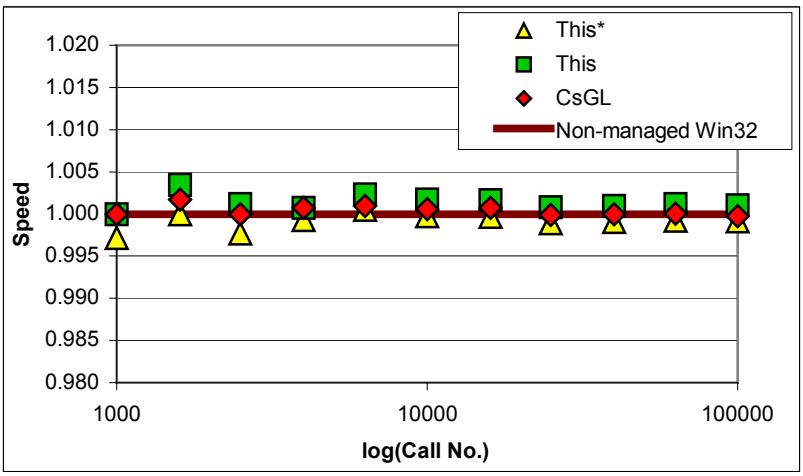
relative values (*speed*) are computed from results of non-managed (Win32) version (T_{Win32}) and managed version (T_{result}) of particular test:

$$speed = \frac{T_{Win32}}{T_{result}} \qquad \text{(eq. 11.1)}$$

Relative value then means percentage non-managed version speed, i.e. value equal to *1.000* means no slowdown and no speedup at all, value that is less then *1.000* means slowdown.

11.1 Test 1: Built-in Value Types

First test is based on that fact that majority of OpenGL functions uses built-in value types as their parameters. This means that built-in value types are used quite often and therefore significant slowdown of sharing them between managed and non-managed code would cause significant decrease of whole OpenGL code performance. Test uses `glVertex2d` function that has two parameters that both are `double`. Test results can be found at Figure 11.1.



Test 1 Results												
Call No.	1000	1600	2500	4000	6300	10000	16000	25000	40000	63000	100000	**average**
CsGL	1.000	1.002	1.000	1.001	1.001	1.001	1.001	1.000	1.000	1.000	1.000	**1.000**
This	1.000	1.003	1.001	1.001	1.002	1.002	1.002	1.001	1.001	1.001	1.001	**1.001**
This*	0.997	1.000	0.998	0.999	1.000	1.000	1.000	0.999	0.999	0.999	0.999	**0.999**

* version with parameter checking

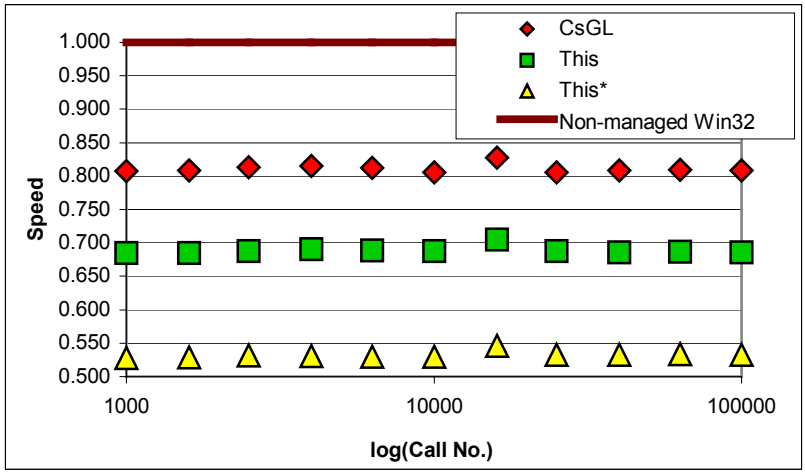
Figure 11.1: Test 1 results

Results of this test leads to a conclusion that there is **no significant slowdown** due to data sharing of built-in value types. Average results are very close to speed of non-managed version for both CsGL and this work even for version with parameter checking.

11.2 Test 2: General Arrays

Second test deals with the next most used type for parameter of OpenGL functions, i.e., arrays. It is aimed on general array as replacement of **void pointer**, i.e., it tests how influences usage of `System.Array` (as parent of all managed arrays) performance of the code. In fact, it compares approaches that use **P/Invoke** (i.e., CsGL) and **GCHandle** structure (i.e., this work).

It is assumed that functions that use general arrays as parameter are not called as often as in the case of built-in value types. Test uses `glTexImage2D` function because this function is called quite often and it allows comparison of results with situation where **.NET provided data sharing** is used instead of general arrays (see section 11.3). It is good to note that even this function uses built-in value type as parameter. Test results can be found at Figure 11.2.



Test 2 Results

Call No.	1000	1600	2500	4000	6300	10000	16000	25000	40000	63000	100000	**average**
CsGL	0.807	0.809	0.813	0.816	0.813	0.806	0.828	0.806	0.809	0.809	0.808	**0.811**
This	0.685	0.685	0.688	0.690	0.689	0.687	0.705	0.687	0.686	0.686	0.686	**0.689**
This*	0.528	0.529	0.532	0.531	0.530	0.530	0.546	0.533	0.533	0.533	0.533	**0.532**

* version with parameter checking

Figure 11.2: Test 2 results

Both CsGL and this work are slower then non-managed code. It is caused by the fact that managed array needs to be protected from garbage collector (i.e., pinned). In the case of this work the slowdown of wrapper is even greater then those of CsGL.

This slowdown is caused by `GCHandle` structure because it performs runtime checks of handled data type in order to prevent user from operating with inappropriate data type. However, these runtime checks consumes quite significant amount of time and the overall influence depends on ration of times consumed by wrapper and function itself.

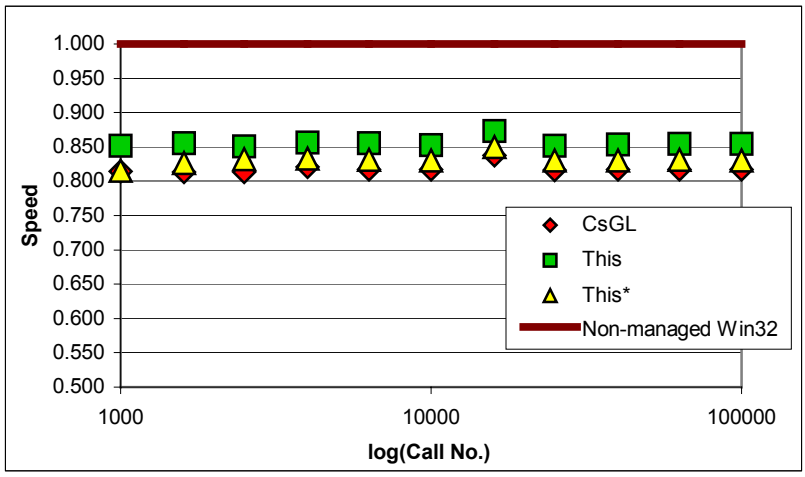
It the case of this function is the consumed time greater then time consumed by the wrapper and therefore is slowdown not significant. For opposite situation (i.e., time consumed by the wrapper is greater then time consumed by a function itself) refer to section 11.4.

Results of this test are comparable to results of third test (see section 11.3) because it uses same OpenGL function with different approach to data sharing. For comparison of various data sharing approaches used, both by this work and by CsGL refer to section 11.5.

11.3 Test 3: .NET provided Data Sharing

This test is similar to previous test (see section 11.2), i.e., it uses same function (`glTexImage2D`). However, it differs in approach to data sharing. In this case both versions (i.e., this work and CsGL) use functionality provided by .NET itself.

This means they use a method that allows locking contents of the image (instance of `System.Drawing.Bitmap`) without need of additional language constructions. Results can be found at Figure 11.3 and graph is formatted similar to those of previous test in order to simplify their comparison.



Test 3 Results												
Call No.	1000	1600	2500	4000	6300	10000	16000	25000	40000	63000	100000	**average**
CsGL	0.815	0.814	0.814	0.821	0.818	0.817	0.837	0.817	0.818	0.817	0.817	**0.818**
This	0.851	0.855	0.851	0.856	0.855	0.852	0.873	0.851	0.854	0.855	0.854	**0.855**
This*	0.816	0.826	0.832	0.833	0.831	0.830	0.850	0.830	0.831	0.831	0.830	**0.831**

* version with parameter checking

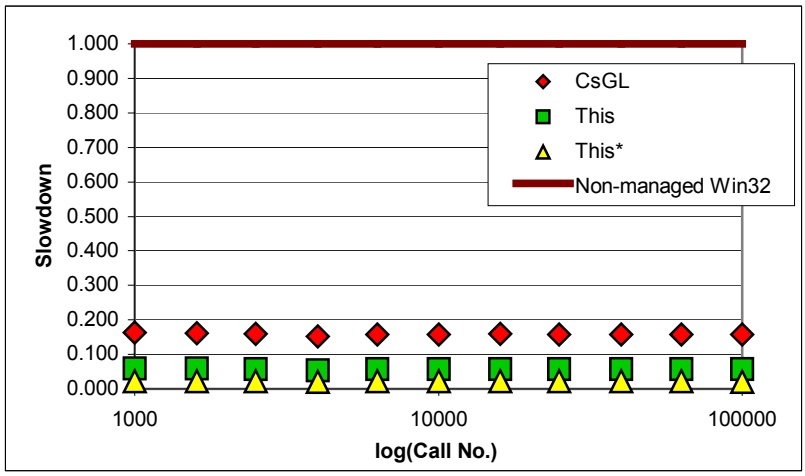
Figure 11.3: Test 3 results

In comparison with previous test, results are better for both CsGL and this work. In the case of this work are results not even significantly better then for previous test but also slightly better then CsGL (for this test). It is caused by the fact that wrapper does not use `GCHandle` structure in order to perform data sharing and uses capabilities of MC++ (i.e., IJW mechanism).

11.4 Test 4: General Arrays Stored

Fourth test deals with data sharing similar to second test (see section 11.2), i.e., it uses general arrays. The only difference is that these data (i.e., pointers) are stored inside OpenGL for later use. This means that in the case of this work these data (their references) need to be stored inside wrapper's internal data structures in order to protect them from garbage collection.

Similar to the second test, this test compares two approaches on data sharing: **P/Invoke** used by CsGL and **GCHandle** structure used by this work. Test uses `glVertexPointer` function that itself consumes only a very short time. Therefore, it is assumed that result of this test will be example of the **worst case**, i.e., situation when the wrapper overhead is **significant** in comparison to function itself. Test result can be found at Figure 11.4.



Test 4 Results												
Call No.	1000	1600	2500	4000	6300	10000	16000	25000	40000	63000	100000	**average**
CsGL	0.163	0.161	0.160	0.153	0.159	0.158	0.159	0.158	0.158	0.158	0.158	**0.159**
This	0.059	0.059	0.058	0.054	0.058	0.057	0.057	0.058	0.057	0.057	0.057	**0.058**
This*	0.023	0.022	0.022	0.021	0.022	0.022	0.022	0.021	0.021	0.021	0.021	**0.022**

* version with parameter checking

Figure 11.4: Test 4 results

Results are significantly slower then non-managed version even thought they should be close to results of the second test (see section 11.2). The reason for that is an assumption from previous paragraph, i.e., time consumed by OpenGL function itself is much shorter then time consumed by wrapper.

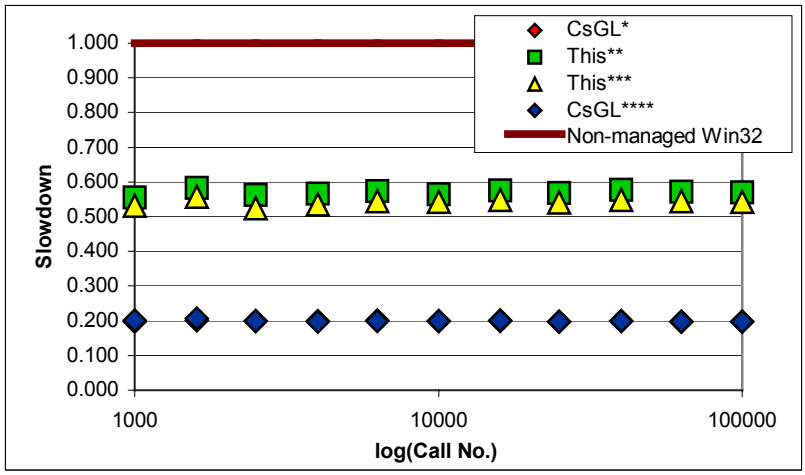
However, it is good to note that these functions (i.e., functions that stores reference to data for later use) are not called too often and time interval consumed by single function is quite short. Therefore, when mixed with another OpenGL function calls, the overall impact on application performance shall not be significant (see section 11.6).

11.5 Test 5: Different Data Sharing Approaches

This test compares different approaches of data sharing used either by this work or by CsGL. It aims on a large group of OpenGL functions that are used to set up specific parts of rendering pipeline such as particular light parameters, material, etc. It is assumed that the time consumed by wrapper is greater or equal to time consumed by function itself. Test is uses `glColor4fv` function. Results of the test can be found at Figure 11.5.

Test compares **P/Invoke** mechanism (CsGL) and **MC++** capabilities (this work) to mix both managed and non-managed code. In the case of P/Invoke mechanism, version that uses unsafe block (i.e., `unsafe` and `fixed` keyword in C#) is compared to a version without unsafe blocks. It is assumed that version with unsafe block shall be faster then version that lacks them.

In the case of this work, the test compares version that uses array of `doubles` and **pinning pointers** with version that uses simple **structure** (i.e., value type) instead of the array. It shall prove whether the use of simple structures leads to performance loss. It is good to note that use of these structures is more comfortable for developer then use of arrays. It is assumed that there is performance trade off for comfort. However, it shall not be significant. All time intervals were measured for version **without** parameter checking.



Test 5 Results

Call No.	1000	1600	2500	4000	6300	10000	16000	25000	40000	63000	100000	**average**
CsGL*	0.198	0.202	0.199	0.200	0.201	0.199	0.200	0.197	0.200	0.198	0.198	**0.199**
This**	0.556	0.583	0.563	0.567	0.574	0.564	0.576	0.568	0.577	0.572	0.570	**0.570**
This***	0.531	0.557	0.523	0.534	0.545	0.543	0.548	0.540	0.548	0.544	0.542	**0.541**
CsGL****	0.202	0.207	0.200	0.198	0.200	0.199	0.200	0.197	0.200	0.198	0.197	**0.200**

* version that uses P/Invoke
** version that uses pinning pointers
*** version that uses structures
**** version that uses unsafe + fixed keyword

Figure 11.5: Test 5 results

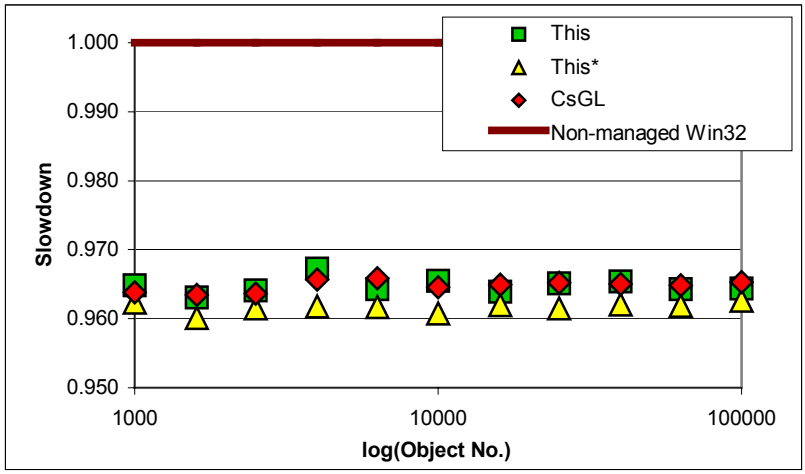
Results show that both CsGL and this work are significantly slower then non-managed version. The cause of that is similar to the assumption that was made above, i.e., time consumed by the function itself is either less or equal to time consumed by the wrapper.

Both pinning pointer and structures are faster then CsGL even using unsafe blocks. It is good to note that structures are only slightly slower then pinning pointer. Therefore, application that uses structures instead of arrays shall experience only slight performance loss as trade off for more comfortable source code developing.

11.6 Test 6: Scene

Last test provides a comparison of both CsGL and this work used to render a simple scene that shall be close to real-world situation. Scene that is used for this test consists of three-dimensional array of textured spheres (see section E.6). Number of spheres is parameter of the test. Each sphere is considered independent object, i.e., there is setting of material, geometry (it uses vertex array; refer to section 11.4), texture (it uses texture objects), and transformation (i.e., sphere position and rotation) for each sphere.

Test shall prove whether is the performance loss of this work significant in comparison with both CsGL and non-managed version. Results of the rest can be found at Figure 11.6.



Test 6 Results												
Call No.	1000	1600	2500	4000	6300	10000	16000	25000	40000	63000	100000	**average**
CsGL	0.964	0.963	0.964	0.966	0.966	0.965	0.965	0.965	0.965	0.965	0.965	**0.965**
This	0.965	0.963	0.964	0.967	0.964	0.965	0.964	0.965	0.965	0.964	0.964	**0.965**
This*	0.962	0.960	0.962	0.962	0.962	0.961	0.962	0.962	0.962	0.962	0.963	**0.962**

* version with parameter checking

Figure 11.6: Test 6 results

Results of both CsGL and this work are comparable and very close to performance of non-managed version. Even version with parameter checking provides reasonable results and is only slightly slower then both version without parameter checking and CsGL version. Therefore, there is an assumption that real-world application that uses OpenGL in managed environment will suffer from only minimal performance loss.

12 Conclusion

The goal of this work was to **verify** the approach rather then create complete port of OpenGL. The reason for that is that OpenGL including all of GL Extensions is quite large library and porting of GL Extensions and OpenGL versions is matter of applying approaches that were verified by this work.

Interface that was designed and implemented differs from original OpenGL specification only slightly. It is more object oriented and uses enumeration types instead of constants. It is true that is approach makes porting little bit difficult, however, it allows to handle non-managed and managed code cooperation easily.

It is good to note that **comfort** of use is one of the things that is paid attention to: enumeration types, improved debugging capabilities, and additional structures. **Enumeration types** are more comfortable in comparison with constants because it is easier to select the value from limited set of value (i.e., enumeration type members) then to select the value from a huge list (i.e., all constants).

Improved debugging capabilities shall **simplify debugging** of OpenGL code by checking parameters of OpenGL functions. Current implementation does not check all parameters due to number of functions together with the fact that the developer has to manually edit the code in order to provide parameter checking for particular function.

Additional structures solve problem of pointers including void pointers. They are basic structures (e.g., vertex, color, etc.) that are usually implemented by user himself during developing of OpenGL code.

Implementation verifies solution introduced in this work. Therefore, it does not provide complete set of all available OpenGL versions and GL Extensions. This work contains implementation of OpenGL version 1.1, GLU version 1.1, and few GL Extensions such as `ARB_multitexture`, `ARB_texture_cube_map`, `EXT_vertex_weighting`.

It is good to note that one of the parts of this work is a tool that allows **semi-automatic conversion** of OpenGL function to managed code based on OpenGL C/C++ header files. The tool suits needs of this work and therefore is quite simple because more general tool would have complicated design and operation of it would be complicated too.

This work also provides a framework for **verification and validation** of implementation. This framework consists of a simple template for test and tool that allows automation of testing. Currently this work contains two sample tests. They are both based on problems of various applications. However, they are examples rather then real tests. Construction of

next tests (i.e., real tests) will depend on particular experience of users with different device drivers and/or future implementations of OpenGL interface.

A result of this work is interface that allows using OpenGL in managed environment. It tries to be more comfortable and easier to debug. It is an alternative to existing approaches and performance of this work is close to CsGL that is example of existing solution and comparable to non-managed version (i.e., Win32). Implementation was tested on various OpenGL tutorials and examples.

Bibliography

[MONO] *Mono project*. [WWW] http://www.go-mono.com (March 1 2003).

[SSCLI] Stutz, D., Neward, T., and Shilling, G. (2003). *Shared Source CLI Essentials*. Sebastopol: O'Reilly.

[IEC] IEC 60599:1989. *Binary Floating-point Arithmetic for Microprocessor Systems*. Previously designated IEC 599:1989.

[CLI-I] *Common Language Infrastructure Partition I: Concepts and Architecture*. (2002). [WWW] http://msdn.microsoft.com/net/ecma/ (March 28 2003).

[CLI-II] *Common Language Infrastructure Partition II: Metadata Definitions and Semantics*. (2002). [WWW] http://msdn.microsoft.com/net/ecma/ (March 28 2003).

[CLI-III] *Common Language Infrastructure Partition III: CIL Instruction Set*. (2002). [WWW] http://msdn.microsoft.com/net/ecma/ (March 28 2003).

[Dav99] Davis, M. and Dürst, M. (1999). *Unicode Technical Report #15*. Revision 18.0. [WWW] http://www.unicode.org/unicode/reports/tr15/tr15-18.html (March 28 2003).

[Ric00] Richter, J. (2000). *Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework*. MSDN Magazine, **11/2000**. [WWW] http://msdn.microsoft.com/msdnmag/issues/1100/GCI/default.aspx (March 28 2003).

[Car02] Carmona, D. (2002). *Programming the Thread Pool in the .NET Framework*. [WWW] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/progthrepool.asp (March 28 2003).

[Ric03] Richter, J. (2003). *Safe Thread Synchronization*. MSDN Magazine, **1/2003**. [WWW] http://msdn.microsoft.com/msdnmag/issues/03/01/net/default.aspx (March 28 2003).

[WL02] Watkins, D. and Lange, S. (2002). *An Overview of Security in the .NET Framework*. [WWW] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/netframesecover.asp?frame=true (March 1 2003).

[RFC1766] IETF RFC1766:1995. *Tags for the Identification of Languages*. [WWW] http://www.ietf.org/rfc/rfc1766.txt (April 20 2003).

[MSDN] *MSDN Library*. (2001). [WWW] http://msdn.microsoft.com (April 10 2003).

[CLI-V] *Common Language Infrastructure Partition V: Annexes*. (2002). [WWW] http://msdn.microsoft.com/net/ecma/ (March 28 2003).

[CSharp] *C# Language Specification*. (2002). [WWW] http://msdn.microsoft.com/net/ecma/ (March 28 2003).

[Watt00] Watt, A. (2000). *3D Computer Graphics*. Harlow: Addison-Wesley.

[GL13] Segal, M. and Akeley, K. (2001). *The OpenGL Graphics System: A Specification Version 1.3*. [WWW] http://www.opengl.org/developers/documentation/version1_3/glspec13.pdf (March 28 2003).

[GL14] Segal, M. and Akeley, K. (2002). *The OpenGL Graphics System: A Specification Version 1.4*. [WWW] http://www.opengl.org/developers/documentation/version1_4/glspec14.pdf (March 28 2003).

[GLU13] Chin, N., Frazier, C., Ho, P., Liu, Z., Smith, K. P. (1998). *The OpenGL Graphics System Utility Library (Version 1.3)*. [WWW] ftp://ftp.sgi.com/opengl/doc/opengl1.2/glu1.3.pdf (March 28 2003).

[SGI] *SGI web-site*. [WWW] http://www.sgi.com/ (March 10 2003).

[Fra03] Frank, M. (2003). *VTK for .NET*. M. A. thesis, University of West Bohemia in Pilsen, Pilsen.

[OpenGL] *OpenGL web-site*. [WWW] http://www.opengl.org/ (February 18 2003)

[CsGL] *CsGL project web-site*. [WWW] http://csgl.sourceforge.net/ (February 20 2003).

[GLSharp] *GLSharp project web-site*. [WWW] http://www.headbits.com/ (April 4 2003).

[GLUT] Kilgard, J. M. (1996). *The OpenGL Utility Toolkit (GLUT) Programming Interface (API Version 3)*. [WWW] http://www.opengl.org/developers/documentation/glut/spec3/spec3.html (May 4 2003).

[Kil02] Kilgard, J. M. *All About OpenGL Extensions*. [WWW] http://www.opengl.org/developers/code/features/OGLextensions/OGLextensions.html (May 4 2003).

[Flex] *Flex project web-site*. [WWW] http://www.gnu.org/software/flex/ (April 16 2003).

[ANSI-C] *ANSI C Grammar for Lex*. 1985. [WWW] http://www.lysator.liu.se/c/ANSI-C-grammar-l.html (April 3 2003).

Abbreviations and Terminology

Assembly Enclosing construction over types. It is the smallest distributable unit of code in .NET Framework.

Delegate Managed (.NET) equivalent of function pointer.

CIL Common Immediate Language. Low-level language used by .NET.

CLI Common Language Infrastructure. Specification of .NET Framework.

CLR Common Language Runtime. See CLI.

CLS Common Language Specification. Specifies rules for outer interface of assembly, which shall simplify language interoperability.

CTS Common Type System. Specification of data types and rules for their use. Part of CLI.

GDI Graphics device interface. Simple 2d graphic output provided by Windows operating system.

GDI+ Advanced version of GDI. See GDI.

GL Extensions Additions to OpenGL library function that might not be available on all hardware that supports OpenGL.

GLU OpenGL Graphic System Utility Library. Additional library that enhances capability of OpenGL library.

GLUT The OpenGL Utility Toolkit. Additional library that simplifies initialization of OpenGL and makes OpenGL code completely platform dependent. This dependency includes OpenGL initialization and system events handling.

GUI Graphical User Interface such as windowing system.

IJW “It Just Works”. Name for capabilities of MC++ to mix managed and non-managed code. Use of IJW shall lead to higher performance then use of P/Invoke.

MC++ C++ Managed Extension. Extended version of C++ language that aims .NET.

Managed code Code (CIL) that is fully under control of .NET.

Manifest Additional informational data for assembly. Contains description of assembly and all types enclosed inside.

P/Invoke Platform Invoke. Mechanism that allows for managed code to execute non-managed code such as function in libraries.

Render Context Represents instance of OpenGL. It needs to be created in order to use OpenGL functionality.

Unmanaged (non-managed) code Code that is not under control of .NET such as native binary code of particular machine.

Win32API Windows Application Programming Interface. Basic interface that is provided by Windows operating system for applications, which uses services of this operating system.

Appendices

Appendix A Usage

This appendix contains **templates** for applications that use this work. All templates use C# syntax. Templates may not a handle all common tasks, e.g., window resizing. Application that uses such template have to reference to `ZCU.Graph.OpenGL.dll` assembly in order to perform **compilation**. In the case of **Visual Studio .NET (2002)** this assembly shall be added to list of references.

When compiling in **command line** the `/reference` option of C# compiler (`csc.exe`) shall be used. In such case, assembly file (i.e., `ZCU.Graph.OpenGL.dll`) has to be situated on one of three possible locations (for detailed information see [MSDN]):

- directory in which the compilation is performed,
- directory specified by `/lib` option of the compiler,
- directory specified by the `LIB` environment variable.

Example of command line compilation can be found at Figure A.1. It compiles single file that (i.e., `myfile.cs`) that contains C# source code. In example it is assumed that assembly file is situated at one of locations specified above. The result of compilation is stored to `myfile.exe` file. Example produces **console application**. However, it is possible to create pure window application. In such case add `/t:winexe` option before source file name.

```
csc myfile.cs /r:ZCU.Graph.OpenGL.dll
```

Figure A.1: Command line compilation example

In order to **run** application that uses this assembly, the assembly has to be situated in the **same** location (i.e., path) as application executable. However, there is another option that uses **Global Assembly Cache** (see [MSDN]). Due to assembly itself is strong-named (i.e., it is signed) it is possible to insert it to Global Assembly Cache so applications can be executed without need to have local copy of assembly file. For details, refer to Appendix A.

Executing of application that does not have access to assembly file produces unhandled exception (e.g., `System.IO.FileNotFoundException`). Due to security policy reasons it is allowed to execute assemblies that contain non-managed code (such as this assembly) **only** from **local storage**, i.e., it is not possible to execute it from network-mapped disk. Attempt to execute assembly from invalid source leads to occurrence of unhandled security exception.

Due to simplification reasons, only templates for **window application** are contained within this appendix. All examples uses .NET classed for windowed output.

```
1:        using System;
2:        using System.Windows.Forms;
3:        using ZCU.Graph.OpenGL;
4:        namespace MyNamespace
5:        {
6:            public class MyForm : Form
7:            {
8:                protected RenderContext rc = null;  /* OpenGL render context */
9:                protected GL gl = null;             /* OpenGL functions */
10:               public MyForm() : base()
11:               {
12:                   /* initialization */
13:                   rc = new RenderContext(this);
14:                   rc.MakeCurrent();
15:                   gl = new GL();
16:                   rc.DoneCurrent();
17:                   /* ... window setup ... */
18:               }
19:               /* painting */
20:               protected override void OnPaint(PaintEventArgs e)
21:               {
22:                   /* ... painting operations ... */
23:               }
24:               /* dispose render context */
25:               protected override void Dispose( bool disposing )
26:               {
27:                   if( disposing )
28:                   {
29:                       if (rc != null) rc.Dispose();
30:                       rc = null; gl = null;
31:                   }
32:                   base.Dispose( disposing );
33:               }
34:               /* main function */
35:               [STAThread]
36:               static void Main() { Application.Run(new MyForm()); }
37:           }
38:       }
```

Figure A.2: Windowed application template

```
 1:        using ZCU.Graph.OpenGL;
 2:        using ZCU.Graph.OpenGL.Forms;
 3:        namespace MyNamespace
 4:        {
               /* GLForm derived class */
 5:            class MyGLForm : GLForm
 6:            {
 7:                MyGLForm() : base() {}
                   /* OpenGL initialization */
 8:                public override void OnGLInitialize()
 9:                {
10:                    /* ... OpenGL initialization ... */
11:                }
                   /* form resizing */
12:                public override bool OnGLResize(System.Drawing.Size size)
13:                {
14:                    /* ... resizing ...
                          ... return true to repaint, false otherwise ... */
15:                }
                   /* form painting */
16:                public override void OnGLPaint()
17:                {
18:                    /* ... painting ... */
19:                }
                   /* main function */
20:                [STAThread]
21:                static void Main(string[] args)
22:                {
23:                    System.Windows.Forms.Application.Run(new MyGLForm());
24:                }
25:            }
26:        }
```

Figure A.3: Windowed application template that benefits from prepared `GLForm` helper class

Appendix B Installation

Installation of assembly (i.e., ZCU.Graph.OpenGL.dll) is very simple and thanks two the fact that this assembly is strong-named there are two options. The user has to provide local copy and/or install the assembly into **Global Assembly Cache**. In both cases .NET Framework has to be **installed** prior to this assembly. Installation and management of .NET Framework is beyond the range this documentation and this work; for such information refer to [MSDN].

Approach that uses **local copy** of the assembly (i.e., copy of assembly file situated in current working directory) is the most simple and straightforward. The major advantage of this approach is the fact that it is suitable for both compilation and execution of application which references to this assembly.

However, previous approach (i.e., local copy) requires an assembly file to be copied together with every application executable. This leads to redundant copies of the same and may cause difficulties while providing updated (bug fixed) version of the assembly file. In such case it is better to use **Global Assembly Cache**.

It is a special storage for assemblies that are shared by multiple applications. These assemblies must have strong name and any user with sufficient security permissions can install (see Figure B.1) and uninstall (see Figure B.2) the assembly with help of **gacutil** tool that is part of .NET Framework. It is good to note that installed assembly is then accessible **only for execution** of application not for compiling of the source code.

```
gacutil /i ZCU.Graph.OpenGL.dll
```

Figure B.1: Assembly installation to Global Assembly Cache

```
gacutil /u ZCU.Graph.OpenGL
```

Figure B.2: Uninstallation of assembly previously inserted into Global Assembly Cache

Appendix C Reference

C.1 Namespaces

ZCU.Graph.OpenGL
Base namespace. Contains system class (e.g., `Render Context`) and enumeration types. All namespaces and types created for this work resides inside this namespace or nested namespaces. Due to that all namespace identifiers are striped of the base namespace name in this section.

Examples
Contains examples of OpenGL applications that were converted from non-managed version to managed version. Examples are grouped via their source.

Forms
Contains classes that simplifies OpenGL instance creation and window management, e.g., `GLForm`. All classes that have similar functionality to GLUT library shall be nested inside this namespace.

GL11Members
Contains types (e.g., enumeration types) that are used for implementation of OpenGL version 1.1 in a form of `GL11` class.

GLExtensions
Contains OpenGL extensions. All extensions shall be derived from `Extension` class that resides in this namespace.

GLExtensions.ARBMultitextureMembers
Contains enums of `ARB_multitexture` GL Extension.

GLExtensions.ARBTextureCubeMapMembers
Contains enums of `ARB_texture_cube_map` GL Extension.

GLExtensions.EXTVertexWeightingMembers
Contains enums of `EXT_vertex_weighting` GL Extension.

GLU11Members
Contains types (e.g., GLU objects) that are used for implementation of GLU version 1.1 in a form of `GLU11` class.

Structures
Contains additional structures that improve comfort of programming, e.g., `Matrix`, `Vertex2d`, etc.

Tests
Contains types used for testing and verification purposes.

Tools

Contains tools that help with either port creation or programming under OpenGL. Currently it contains tool that creates source code templates for OpenGL port.

C.2 Classes

C.2.1 ARBMultitexture

Public. `ARB_multitexture` GL Extension implementation. Class contains OpenGL functions and constants., however, they are not listed here due to their number and the fact that they are described by particular OpenGL specification.

C.2.1.1 Constructors

ARBMultitexture()

Public. Creates instance of GL Extension. Requires current instance of OpenGL (`RenderContext` class instance) to be valid. It is then considered owner of this instance.

C.2.2 ARBTextureCubeMap

Public. `ARB_texture_cube_map` GL Extension implementation. Class contains OpenGL functions and constants., however, they are not listed here due to their number and the fact that they are described by particular OpenGL specification.

C.2.2.1 Constructors

ARBTextureCubeMap()

Public. Creates instance of GL Extension. Requires current instance of OpenGL (`RenderContext` class instance) to be valid. It is then considered owner of this instance.

C.2.3 BaseRenderContext

Public. OpenGL render context class. Represents OpenGL instance. Instance of this class or derived class needs to be created in order to use OpenGL/GLU/GL Extension. Used for windowed rendering, i.e., it has no support for fullscreen rendering.

This class is base class for all classes representing OpenGL instance.

C.2.3.1 Constructors

BaseRenderContext()

Protected. Initializes internal data structures.

C.2.3.2 Methods

Dispose()

Public. Destroys render context, releases resources, and disposes object. Users are required to call this method.

DoneCurrent()

Public. Virtual. Sets this instance of OpenGL to be not current. Used in pairs with `MakeCurrent`. All OpenGL/GLU/GL Extension method calls and class instance constructions shall be performed in a block encapsulated by this pair.

MakeCurrent()

Public. Virtual. Makes this instance of OpenGL to be current one. Used in pairs with `DoneCurrent`. All OpenGL/GLU/GL Extension method calls and class instance constructions shall be performed in a block encapsulated by this pair.

SwapBuffers()

Public. Virtual. Swaps current buffer with buffer that is displayed on screen. Shall be called outside block enclosed by `MakeCurrent` and `DoneCurrent` pair.

Dispose(bool)

Protected. Virtual. Destroys render context, releases resources, and disposes object. Users are required to call this method.

`disposing`	True to dispose both managed and non-managed resources. False to dispose only non-managed resources (e.g., HWND, DC, etc.).

InitializeRC(HWND, ColorBitDepth, BitDepth, BitDepth, RCProperties)

Protected. Creates instance of OpenGL with specified properties.

`hWnd`	Win32 window handle.
`color`	Color bit depth.
`depth`	Depth buffer bit depth.
`stencil`	Stencil buffer bit depth.
`properties`	Properties that are used for this instance of OpenGL (e.g., double buffering, etc.).

SetPixelFormat(ColorBitDepth, BitDepth, BitDepth, RCProperties)

Private. Sets pixel format of target window device context. It also performs automatic selection of color, depth, and stencil buffer bit depth if required. Analogous to Win32API function `SetPixelFormat`.

`color`	Color bit depth.
`depth`	Depth buffer bit depth.
`stencil`	Stencil buffer bit depth.
`properties`	Properties that are used for this instance of OpenGL (e.g., double buffering, etc.).

C.2.3.3 Properties

IsInitialized

Public. Read-only. True if this instance of OpenGL is valid (i.e., it was initialized and it was not disposed).

ColorDepth

Public. Read-only. Color buffer bit depth of this OpenGL instance.

DepthDepth

Public. Read-only. Depth buffer bit depth of this OpenGL instance.

StencilDepth

Public. Read-only. Stencil buffer bit depth of this OpenGL instance.

IsDoubleBuffered

Public. Read-only. True if this instance uses double buffering.

IsStereo

Public. Read-only. True if this instance support stereo output, e.g., for stereo glasses.

C.2.3.4 Fields

hWnd

Protected. Win32 handle for output window.

hDC

Protected. Win32GDI device context of output window.

hRC

Protected. OpenGL instance render context handle.

color
Protected. Color buffer bit depth.

depth
Protected. Depth buffer bit depth.

stencil
Protected. Stencil buffer bit depth.

properties
Protected. Properties of OpenGL instance output.

CurrentRC
Internal. Static. Current OpenGL instance.

stackRC
Private. Static. Stack for storing of current OpenGL instances. It allows the user changing of current OpenGL instance without need of explicit storing of previous current one.

possibleColors
Private. Static. All possible color buffer bit depths. Used for automatic resolution of the best available bit depth.

possibleDepths
Private. Static. All possible depth buffer bit depths. Used for automatic resolution of the best available bit depth.

possibleStencils
Private. Static. All possible stencil buffer bit depths. Used for automatic resolution of the best available bit depth.

C.2.4 DisplayModeInfo

Public. Contains specification of current and/or available display mode. It is used for setting of fullscreen mode properties.

C.2.4.1 Constructors

DisplayModeInfo(System.Int32, System.Int32, ColorBitDepth, System.Int32)
Public. Creates instance of this class.

`width`	Width of screen resolution.
`height`	Height of screen resolution.
`colorDepth`	Color buffer bit depth, i.e., bits per pixel.
`displayFrequency`	Frequency of refresh for fullscreen mode.

DisplayModeInfo(System.Drawing.Size, ColorBitDepth, System.Int32)
Public. Creates instance of this class.

`size`	Screen resolution.
`colorDepth`	Color buffer bit depth, i.e., bits per pixel.
`displayFrequency`	Frequency of refresh for fullscreen mode.

DisplayModeInfo(DEVMODE)
Internal. Creates instance of this class. Used for retrieving of current and/or available display modes.

`devMode` WIN32 device mode information.

C.2.4.2 Methods

`ToString()`
Public. Overridden. Returns information about contents of instance, e.g., 640x480x24bppx75Hz.

returns Contents of instance.

`EnumDisplayModes()`
Public. Static. Enumeration of all available modes.

returns Available display mode.

`ToDEVMODE()`
Internal. Converts contents of instance into DEVMODE structure. Used for switching to display mode.

returns Contents of instance in a form of DEVMODE structure.

C.2.4.3 Properties

`Size`
Public. Read-only. Screen resolution.

`ColorDepth`
Public. Read-only. Color buffer bit depth.

`DisplayFrequency`
Public. Read-only. Fullscreen mode screen refresh.

`Current`
Public. Static. Read-only. Current display mode.

C.2.4.4 Fields

`size`
Private. Screen resolution.

`colorDepth`
Private. Color buffer bit depth.

`displayFrequency`
Private. Fullscreen mode display frequency.

C.2.5 EXTVertexWeighting

Public. `EXT_vertex_weighting` GL Extension implementation. Class contains OpenGL functions and constants., however, they are not listed here due to their number and the fact that they are described by particular OpenGL specification.

C.2.5.1 Constructors

`EXTVertexWeighting()`
Public. Creates instance of GL Extension. Requires current instance of OpenGL (`RenderContext` class instance) to be valid. It is then considered owner of this instance.

C.2.6 FullscreenRenderContext

Public. OpenGL render context class with possibility to render on fullscreen. Derived from `RenderContext`.

C.2.6.1 Constructors

`FullscreenRenderContext(System.Windows.Forms.Form)`

Public. Creates instance of OpenGL. Uses current color bit depth and screen resolution for fullscreen mode. Chooses the best possible bit depth for depth buffer and stencil buffer. Created instance uses double buffering and is not in fullscreen mode implicitly.

`form`	Form that is used for output. It have to be application top-most form. Class may modify properties of such window in order to perform switching between fullscreen and windowed mode.

`FullscreenRenderContext(System.Windows.Forms.Form, DisplayModeInfo)`

Public. Creates instance of OpenGL. Uses user supplied color bit depth and screen resolution for fullscreen mode. Chooses the best possible bit depth for depth buffer and stencil buffer. Created instance uses double buffering and is not in fullscreen mode implicitly.

`form`	Form that is used for output. It have to be application top-most form. Class may modify properties of such window in order to perform switching between fullscreen and windowed mode.
`mode`	Display mode (i.e., resolution, color buffer bit depth, and refresh rate) used for fullscreen mode.

`FullscreenRenderContext(System.Windows.Forms.Form, DisplayModeInfo, BitDepth, BitDepth, RCProperties)`

Public. Creates instance of OpenGL. Uses user supplied display mode for fullscreen mode, depth buffer bit depth, stencil buffer bit depth, and render output properties.

`form`	Form that is used for output. It have to be application top-most form. Class may modify properties of such window in order to perform switching between fullscreen and windowed mode.
`mode`	Display mode (i.e., resolution, color buffer bit depth, and refresh rate) used for fullscreen mode.
`depth`	Depth buffer bit depth.
`stencil`	Stencil buffer bit depth.
`properties`	Properties that are used for this instance of OpenGL (e.g., double buffering, etc.).

`FullscreenRenderContext(System.Windows.Forms.Form, System.Drawing.Size, ColorBitDepth, BitDepth, BitDepth, System.Int32, RCProperties)`

Public. Creates instance of OpenGL. Uses user supplied display mode for fullscreen mode, depth buffer bit depth, stencil buffer bit depth, and render output properties.

`form`	Form that is used for output. It have to be application top-most form. Class may modify properties of such window in order to perform switching between fullscreen and windowed mode.
`size`	Resolution for fullscreen mode.
`color`	Color bit depth.
`depth`	Depth buffer bit depth.
`stencil`	Stencil buffer bit depth.
`refresh`	Refresh rate for fullscreen mode.
`properties`	Properties that are used for this instance of OpenGL (e.g., double buffering, etc.).

C.2.6.2 Methods

`EnterFullscreen()`

Public. Enters fullscreen mode.

`LeaveFullscreen()`
Public. Leaves fullscreen mode.

`Dispose(bool)`
Protected. Overriden. Destroys render context, releases resources, and disposes object. Users are required to call this method.

`disposing`	True to dispose both managed and non-managed resources. False to dispose only non-managed resources (e.g., HWND, DC, etc.).

`Init(DisplayModeInfo)`
Private. Performs initialization of data structures.

`mode`	Display mode (i.e., resolution, color buffer bit depth, and refresh rate) used for fullscreen mode.

C.2.6.3 Properties

`IsFullscreen`
Public. Read-only. True if it is in fullscreen mode currently.

`DisplayMode`
Public. Read-only. Retrieves fullscreen display mode parameters.

C.2.6.4 Fields

`form`
Protected. Contains reference to output form. Its contents is similar to those of `RenderContext.control`.

`origDesktopBounds`
Protected. Output form's original desktop bounds. Used for fullscreen mode switching, i.e., it allows to restore original window.

`origFormBorderStyle`
Protected. Output form's original border style. Used for fullscreen mode switching, i.e., it allows to restore original window.

`origWindowState`
Protected. Output form's original state. Used for fullscreen mode switching, i.e., it allows to restore original window.

`displayMode`
Protected. Fullscreen display mode.

C.2.7 GL

Public. Latest implemented version of OpenGL (currently 1.1). Inherited from `GL11`.

C.2.7.1 Constructors

`GL()`
Public. Creates instance of OpenGL functions. Requires current instance of OpenGL (`RenderContext` class instance) to be valid. It is then considered to be owner of this instance.

C.2.8 GL11

Public. OpenGL version 1.1 interface implementation. Class contains OpenGL functions and constants., however, they are not listed here due to their number and the fact that they are described by particular OpenGL specification.

C.2.8.1 Constructors

`GL11()`
Public. Creates instance of OpenGL functions. Requires current instance of OpenGL (`RenderContext` class instance) to be valid. It is then considered to be owner of this instance.

C.2.8.2 Properties

`RenderContext`
Public. Read-only. Owner instance of `RenderContext`.

C.2.8.3 Fields

`rc`
Protected. Owner instance of `RenderContext`.

`data`
Protected. Internal data.

C.2.9 GLException

Public. Exception for OpenGL exception. Derived from `System.ApplicationException`.

C.2.9.1 Constructors

`GLException()`
Public. Constructor.

`GLException(System.String)`
Public. Constructor.

`strErr`	Error message.

C.2.10 GLU

Public. Latest implemented version of GLU (currently 1.1). Inherited from `GLU11`.

C.2.10.1 Constructors

`GLU()`
Public. Creates instance of GLU functions. Requires current instance of OpenGL (`RenderContext` class instance) to be valid. It is then considered to be owner of this instance.

C.2.11 GLU11

Public. GLU version 1.1 interface implementation. Class contains GLU functions and constants., however, they are not listed here due to their number and the fact that they are described by particular GLU specification.

C.2.11.1 Constructors

`GLU11()`
Public. Creates instance of GLU functions. Requires current instance of OpenGL (`RenderContext` class instance) to be valid. It is then considered to be owner of this instance.

C.2.11.2 Properties

`RenderContext`
Public. Read-only. Owner instance of `RenderContext`.

C.2.11.3 Fields

`rc`
Protected. Owner instance of `RenderContext`.

C.2.12 RenderContext

Public. OpenGL render context class. Represents OpenGL instance. Instance of this class or derived class needs to be create in order to use OpenGL/GLU/GL Extension. Used for windowed rendering, i.e., it has no support for fullscreen rendering.

This class is base class for classes representing OpenGL instance for **.NET**. Derived from `BaseRenderContext`.

C.2.12.1 Constructors

`RenderContext(System.Windows.Forms.Control)`
Public. Creates instance of OpenGL. Uses current color bit depth and the best possible bit depth for depth buffer and stencil buffer. Created instance uses double buffering.

`control`	Control that is used for output.

`RenderContext(System.Windows.Forms.Control, RCProperties)`
Public. Creates instance of OpenGL. Uses current color bit depth and the best possible bit depth for depth buffer and stencil buffer.

`control`	Control that is used for output.
`properties`	Properties that are used for this instance of OpenGL (e.g., double buffering, etc.).

`RenderContext(System.Windows.Forms.Control, ColorBitDepth, RCProperties)`
Public. Creates instance of OpenGL. Uses specified color bit depth and the best possible bit depth for depth buffer and stencil buffer.

`control`	Control that is used for output.
`color`	Color bit depth of output.
`properties`	Properties that are used for this instance of OpenGL (e.g., double buffering, etc.).

`RenderContext(System.Windows.Forms.Control, ColorBitDepth, BitDepth, RCProperties)`
Public. Creates instance of OpenGL. Uses specified color and depth buffer bit depth. Select the best possible stencil buffer bit depth.

`control`	Control that is used for output.
`color`	Color bit depth.
`depth`	Depth buffer bit depth.
`properties`	Properties that are used for this instance of OpenGL (e.g., double buffering, etc.).

`RenderContext(System.Windows.Forms.Control, ColorBitDepth, BitDepth, BitDepth, RCProperties)`
Public. Creates instance of OpenGL. Uses specified color, depth, and stencil buffer bit depth.

`control`	Control that is used for output.
`color`	Color bit depth.
`depth`	Depth buffer bit depth.
`stencil`	Stencil buffer bit depth.
`properties`	Properties that are used for this instance of OpenGL (e.g., double buffering, etc.).

C.2.12.2 Methods

`Dispose(bool)`

Protected. Virtual. Destroys render context, releases resources, and disposes object. Users are required to call this method.

`disposing`	True to dispose both managed and non-managed resources. False to dispose only non-managed resources (e.g., HWND, DC, etc.).

`Initialize(System.Windows.Forms.Control, ColorBitDepth, BitDepth, BitDepth, RCProperties)`

Protected. Creates instance of OpenGL with specified properties.

`control`	Control that is used for output.
`color`	Color bit depth.
`depth`	Depth buffer bit depth.
`stencil`	Stencil buffer bit depth.
`properties`	Properties that are used for this instance of OpenGL (e.g., double buffering, etc.).

C.2.12.3 Fields

`control`

Protected. Control used for displaying of OpenGL output.

C.2.13 Forms.GLForm

Public. Abstract. Simple standalone OpenGL output window for .NET. Replacement for some of GLUT capabilities. Derived from `System.Windows.Forms.Form`. Class protects all user's OpenGL overrides calls with critical section.

C.2.13.1 Constructors

`GLForm()`

Public. Creates instance of form. Uses current color buffer bit depth, the best available depth and stencil buffer bit depth, and double buffering.

`GLForm(ColorBitDepth, BitDepth, BitDepth)`

Public. Creates instance of form with given parameters.

`color`	Color bit depth.
`depth`	Depth buffer bit depth.
`stencil`	Stencil buffer bit depth.

C.2.13.2 Methods

`OnGLDestroy()`

Public. Virtual. Called at window closing with OpenGL instance of this form set as current.

`OnGLInitialize()`

Public. Virtual. Called at window initialization with OpenGL instance of this form set as current.

`OnGLPaint()`

Public. Abstract. Called at window painting with OpenGL instance of this form set as current. Rendered image is then automatically displayed on screen, i.e., user does not need to call `rc.SwapBuffers()`.

`OnGLResize(System.Drawing.Size)`

Public. Virtual. Called at window resizing with OpenGL instance of this form set as current. Default implementation does no modifications to projection matrix.

`size`	Size of form's client area.

`Dispose(bool)`

Protected. Override. Destroys render context.

`disposing`	True to dispose both managed and non-managed resources. False to dispose only non-managed resources (e.g., HWND, DC, etc.).

`OnPaintBackground(System.Windows.Forms.PaintEventArgs)`

Protected. Override. Default implementation disables redrawing of form's background.

`pevent`	Painting event information.

`OnPaint(System.Windows.Forms.PaintEventArgs)`

Protected. Override. Default implementation locks critical section and calls user's drawing method (`OnGLPaint`).

`pevent`	Painting event information.

`OnSizeChanged(System.Windows.Forms.EventArgs)`

Protected. Override. Default implementation locks critical section and calls user's resizing method (`OnGLResize`).

`pevent`	Event information.

`Init(ColorBitDepth, BitDepth, BitDepth)`

Private. Initializes internal data and OpenGL instance.

`color`	Color bit depth.
`depth`	Depth buffer bit depth.
`stencil`	Stencil buffer bit depth.

C.2.13.3 Properties

`rc`

Public. Read-only. OpenGL instance.

C.2.13.4 Fields

`gl`

Protected. OpenGL functions.

`glu`

Protected. GLU functions.

`mutexGL`

Protected. Critical section that encloses user's override calls.

`renderContext`

Private. OpenGL instance.

C.2.14 GLExtensions.Extension

Public. GL/GLU extension base class. All GL Extensions shall be derived from this class. All classes derived from this render class have to be created only when there is current OpenGL instance available.

C.2.14.1 Constructors

`Extension()`
Protected. Creates instance of GL Extensions. Used in derived classes.

`control`	Control that is used for output.

C.2.14.2 Methods

`IsExtAvail(System.String)`
Protected. Checks if given extension exists.

`extName`	Extension name.
`returns`	True, if extension is available.

C.2.14.3 Properties

`RenderContext`
Public. Read-only. Owner OpenGL instance.

C.2.14.4 Fields

`rc`
Protected. Owner OpenGL instance.

C.3 Structures

C.3.1 Vertex

`Vertex2f`
Public. Contains 2D vertex coordinate. Its components stored in `System.Single`.

`Vertex3f`
Public. Contains 3D vertex coordinate. Its components stored in `System.Single`.

`Vertex4f`
Public. Contains 4D vertex coordinate. Its components stored in `System.Single`.

`Vertex2d`
Public. Contains 2D vertex coordinate. Its components stored in `System.Double`.

`Vertex3d`
Public. Contains 3D vertex coordinate. Its components stored in `System.Double`.

`Vertex4d`
Public. Contains 4D vertex coordinate. Its components stored in `System.Double`.

C.3.2 Vector and Normal

`Vector3f`
Public. Contains 3D vector. Its components stored in `System.Single`.

`Vector3d`
Public. Contains 3D vector. Its components stored in `System.Double`.

C.3.3 Color

`Color3b`
Public. Contains RGB color. Single component stored in `System.SByte`.

Color3ub
Public. Contains RGB color. Single component stored in `System.Byte`.

Color3s
Public. Contains RGB color. Single component stored in `System.Int16`.

Color3us
Public. Contains RGB color. Single component stored in `System.UInt16`.

Color3i
Public. Contains RGB color. Single component stored in `System.Int32`.

Color3ui
Public. Contains RGB color. Single component stored in `System.UInt32`.

Color3f
Public. Contains RGB color. Single component stored in `System.Single`.

Color3d
Public. Contains RGB color. Single component stored in `System.Double`.

Color4b
Public. Contains RGBA color. Single component stored in `System.SByte`.

Color4ub
Public. Contains RGBA color. Single component stored in `System.Byte`.

Color4s
Public. Contains RGBA color. Single component stored in `System.Int16`.

Color4us
Public. Contains RGBA color. Single component stored in `System.UInt16`.

Color4i
Public. Contains RGBA color. Single component stored in `System.Int42`.

Color4ui
Public. Contains RGBA color. Single component stored in `System.UInt42`.

Color4f
Public. Contains RGBA color. Single component stored in `System.Single`.

Color4d
Public. Contains RGBA color. Single component stored in `System.Double`.

C.3.4 Texture Coordinate

TexCoord2f
Public. Contains 2D texture coordinate. Its components stored in `System.Single`.

TexCoord4f
Public. Contains 4D texture coordinate. Its components stored in `System.Single`.

C.3.5 InterleavedArrays Structure

C4fN3fV3f
Public. Structure for `InterleavedArrays` function. Similar to structure specified by original OpenGL constant `GL_C4F_N3F_V3F`.

C3fV3f
Public. Structure for `InterleavedArrays` function. Similar to structure specified by original OpenGL constant `GL_C3F_V3F`.

T2fC4ubV3f
Public. Structure for `InterleavedArrays` function. Similar to structure specified by original OpenGL constant `GL_T2F_C4UB_V3F`.

T2fC3fV3f
Public. Structure for `InterleavedArrays` function. Similar to structure specified by original OpenGL constant `GL_T2F_C3F_V3F`.

T2fC4fN3fV3f
Public. Structure for `InterleavedArrays` function. Similar to structure specified by original OpenGL constant `GL_T2F_C4F_N3F_V3F`.

C4ubV3f
Public. Structure for `InterleavedArrays` function. Similar to structure specified by original OpenGL constant `GL_C4UB_V3F`.

T2fN3fV3f
Public. Structure for `InterleavedArrays` function. Similar to structure specified by original OpenGL constant `GL_T2F_N3F_V3F`.

T2fV3f
Public. Structure for `InterleavedArrays` function. Similar to structure specified by original OpenGL constant `GL_T2F_V3F`.

T4fV4f
Public. Structure for `InterleavedArrays` function. Similar to structure specified by original OpenGL constant `GL_T4F_V4F`.

T4fC4fN3fV4f
Public. Structure for `InterleavedArrays` function. Similar to structure specified by original OpenGL constant `GL_T4F_C4F_N3F_V4F`.

N3fV3f
Public. Structure for `InterleavedArrays` function. Similar to structure specified by original OpenGL constant `GL_N3F_V3F`.

C4ubV2f
Public. Structure for `InterleavedArrays` function. Similar to structure specified by original OpenGL constant `GL_C4UB_V2F`.

C.3.6 Other

Matrixf
Public. Contains 4x4 matrix used by OpenGL for transformations. Simplifies loading and storing of OpenGL transformation matrix. Provides indexer that allows accessing similar to either an array with 16 members or an array with rank equal to 2. Components are stored in `System.Single`.

Matrixd
Public. Contains 4x4 matrix used by OpenGL for transformations. Simplifies loading and storing of OpenGL transformation matrix. Provides indexer that allows accessing similar to either an array with 16 members or an array with rank equal to 2. Components are stored in `System.Double`.

C.4 Enumeration Types

This section contains list of enumeration types **excluding** types that are used by OpenGL/GLU function instead of constants. Such enumeration types are then described by

their members (i.e., OpenGL/GLU constants) whose description can be found in particular OpenGL/GLU specification.

`ColorBitDepth`

Public. Enum. Color buffer bit depths.

Member	Description
`Current`	Forces system to use current color bit
`Auto`	Forces system to choose bit depth.
`Palette`	256 colors / 8 bits per pixel.
`HighColor`	64K colors / 16 bits per pixel.
`TrueColor`	16M colors / 24 bits per pixel.
`TrueColor32`	16M colors / 32 bits per pixel.

`BitDepth`

Public. Enum. Depth and stencil buffer bit depths.

Member	Description
`Auto`	Forces system to choose bit depth.
`None`	Zero. Declares that particular buffer is not supported.
`Depth1bit`	1 bit depth.
`Depth2bit`	2 bits depth.
`Depth4bit`	4 bits depth.
`Depth8bit`	8 bits depth.
`Depth16bit`	16 bits depth.
`Depth32bit`	32 bits depth.

`RCProperties`

Public. Enum. Render output property flags. Flags can be combined via bit-based operations.

Member	Description
`DoubleBuffer`	Double buffering support for render target.
`Stereo`	Stereo support for render target, e.g., support for 3D glasses.
`Fullscreen`	Forces fullscreen mode immediately after creation of OpenGL instance. Valid only for render context with fullscreen support.

C.5 Helper Macros

Helper macros a C/C++ macros that are used for parameter checking at most. Macros contains only a short code and can be easily replace by methods. Opposite of functions they offer slightly higher performance due to lack of function call mechanism.

C.5.1 Function Parameter Checking

`CheckCurrentRC(__func_name)`

Tests whether is current `RenderContext` instance equal to owner instance. If test fails then an exception is raised and `__func_name` is used for creating of exception message in order to identify source of an exception.

`CheckGetCurrentRC(__var)`

Tests whether is current `RenderContext` valid (i.e., equal to `null`). If it is valid then it is assigned to `__var`. Otherwise an exception is raised.

CheckNull(__ar)
Tests whether is __ar equal to null. If test succeeds then an exception is raised.

CheckValid(__obj)
Tests whether is given GLU object (__obj) valid. If test fails then an exception is raised.

CheckValue(__ar)
Tests whether are members of given array (__ar) either built-in value types or structures with sequential layout. If test fails then an exception is raised.

CheckSize(__ar, __min_sz)
Tests whether given array (__ar) has at least __min_sz members. It is used only for **non-general array**s, i.e., arrays that are not type-casted to System.Array. Test always succeeds if the __min_sz is less then zero. If test fails then an exception is raised.

CheckByteSize(__ar, __min_sz)
Tests whether given array (__ar) has such members the sum of their sizes if greater or equal to __min_sz. It is used only for **general array**s, i.e., arrays that are type-casted to System.Array. Test always succeeds if the __min_sz is less then zero. If test fails then an exception is raised.

CheckEnumValue(__par, __val)
Tests whether __par has value of enumeration type member __val. Used for enumeration types. If test fails then an exception is raised.

CheckEnumValue*n*(__par, __val1, __val2, ..., __val*n*)
Tests whether __par has value equal to one of listed enumeration type members __val*n*. Used for enumeration types. If test fails then an exception is raised.

CheckParamValue(__par, __val)
Tests whether __par has value equal to __val. Used of types that are not enumeration types. If test fails then an exception is raised.

C.5.2 Value Type and Structure Definition

STRUCTURE_LAYOUT
Declares that following structure has **sequential** layout and its members are aligned on **byte boundary**. Uses StructLayout attribute.

DEFAULT_ITEM
Declares that following class or structure may have default indexer. Such indexer is called Item.

VALUE
Used for definition of value types (enumeration types, structures). Shall be used instead of __value keyword. Simplifies moving of nested value types out of enclosing class type.

C.5.3 Other

GetGLExtFncAddress(__fnc_name)
Returns either pointer to GL Extension function specified by __fnc_name or NULL if there is no such function available.

PtrOrNull(__par)
Returns either void pointer value of IntPtr data type for giver GCHandle (__par) or null if __par is not allocated or valid.

RetrieveInternalData(__to, __from, __class)
Retrieves or creates instance of internal data class (__class). Reference is copied from __from field of rc variable (field) to __to field. Field rc is assumed to contain valid instance of RC class.

C.6 Preprocessor Directives

Preprocessor directives are used to perform **conditional compilation**, i.e., they are used to mask out parts of code. All of such directives are contained in `CompilationSetup.h` header file.

`INC_CONST_IN_ENUMS`
If it is **not defined** then all OpenGL/GLU constants that are members of enumeration types are not included into compilation, e.g., constant `GL_QUADS` does not exists due to it is member of `RenderPrimitive` enumeration type.

`NO_CORRECT_RC_CHECK`
If it is **not defined** then a test is performed in order to check whether a current `RenderContext` is equal to an owner (i.e., macro `CheckCurrentRC` is used) for every OpenGL/GLU function call.

`NO_PARAM_CHECK`
If it is **not defined** then runtime parameter checking is performed.

Appendix D Interface Verification

D.1 Test

Test is implemented in a form of standalone console application or assembly. Such assembly (application) may contain exactly **one** class derived from `TestBase` class and multiple other classes. Class derived from `TestBase` then contains base runtime code for test. Skeleton (framework) of simple test can be found at Figure D.1.

```
 1:        class MyTestClass : TestBase
 2:        {
 3:            public MyTestClass(string[] args) : base(args) {}
 4:
 5:            protected override void OpenGLInit()
 6:            {
 7:                /* ... OpenGL environment settings ... */
 8:            }
 9:
10:            protected override void OpenGLPaint()
11:            {
12:                /* ... scene rendering ... */
13:                gl.Flush();
14:            }
15:
16:            protected override void OpenGLDone()
17:            {
18:                /* ... releasing of allocated resources ... */
19:            }
20:
21:            [STAThread]
22:            static void Main(string[] args)
23:            {
24:                Run(typeof(MyTestClass), args);
25:            }
26:        }
```

Figure D.1: Test application skeleton.

D.2 TestBase Class Reference

Public. Abstract. Interface verification test base class. Provides common functionality. Derived from `System.Windows.Forms.Form`.

D.2.1 Constructors

`TestBase(System.String[])`

Protected. Creates instance of test form. Creates form and instance of OpenGL.

`args` Command line parameters. Cannot be null.

Parameter	Description
`-rX`	Specifies path to resource, e.g., `-r/res`. Default path is `resources`.
`-oX`	Specifies rendered image filename, e.g., `-oMyImage.png`.
`-q`	Quiet mode. Suppress console output and disables possibility to close form with mouse.
`-h`	Displays help information.

D.2.2 Methods

`Run(System.Type, System.String[])`

Public. Static. Runs console application, i.e., creates output window, parses command line parameters, if any.

`type` Type of class derived from this class that is used for testing.

`args` Command line parameters.

`Dispose(System.Boolean)`

Protected. Overridden. Destroys render context, releases resources, and close form.

`disposing` True to dispose both managed and non-managed resources. False to dispose only non-managed resources (e.g., HWND, DC, etc.).

`OnPaintBackground(System.Windows.Forms.PaintEventArgs)`

Protected. Overridden. Prevents from painting background.

`pevent` Painting event information.

`OnPaint(System.Windows.Forms.PaintEventArgs)`

Protected. Overridden. Handles painting of form contents.

`e` Painting event information.

`OnResize(System.Windows.Forms.EventArgs)`

Protected. Overridden. Handles form resizing. Sets orthogonal projection with range for both horizontal and vertical axis of at least <-1.0; 1.0>.

`e` Event information.

`OpenGLInit()`

Protected. Abstract. Called immediately after creation of OpenGL instance.

`OpenGLPaint()`

Protected. Abstract. Called during window painting and capturing rendered image.

`OpenGLDone()`

Protected. Abstract. Called before is OpenGL instance disposed.

D.2.3 Properties

`ScreenSnapShot`

Public. Read-only. Rendered image. Buffered, i.e., it is read only when user reads this property for first time.

`IsInitialized`
Protected. Read-only. True, if form and OpenGL were initialized.

D.2.4 Fields

`gl`
Protected. OpenGL functions.

`glu`
Protected. GLU functions.

`dirResource`
Protected. Directory that contains resources.

`OutputSize`
Protected. Static. Read-only. Client area size for output form. It also determines dimensions of rendered image.

`rc`
Private. OpenGL instance (`RenderContext` class).

`screenSnapShot`
Private. Buffered rendered image.

D.3 Configuration File

List of all tests including directory in which they are stored, directory that contains reference image, and thresholds are stored in a configuration file. This files is text and uses formatted XML in order to allow its modifications. This section contains list of all used tags.

`<config></config>`
Root tag. Contains all tags but `<config>` tag.

`<directories></directories>`
Contains configuration for directories. Contains `<references>`, `<tests>`, and `<resources>` tag. These tags contains paths in a form of strings. If there are multiple appearance of single tag, the last one is used. If **relative** paths are used then they are relative to **location of** automatic processing **tool**.

`<thresholds></thresholds>`
Contains threshold. Contains `<maximum>` (maximum threshold), `<mean>` (mean threshold), and `<sigma>` (standard deviation threshold). Values for thresholds are floating-point numbers. If there are multiple appearance of single tag, the last one is used.

`<tests></tests>`
Contains tests. Contains multiple `<test>` tags.

```
<test title="title" assembly="assembly.exe"
    reference="image.png"></test>
```

Test description. Contains **single** `<description>` tag whose value is string: test description.

`title`	Title of test.
`assembly`	Standalone console application that contains **one** test, i.e., single class derived from `TestBase` class.
`reference`	Name of reference image file.

D.4 Quick User Manual

Test Manager application is an application with simple GUI. The GUI consist of **main window**, **image dialog**, and **threshold dialog**. Both image and threshold dialogs are quite simple and self-documenting, therefore only main window (see Figure D.2) is described here in order to provide guideline for user.

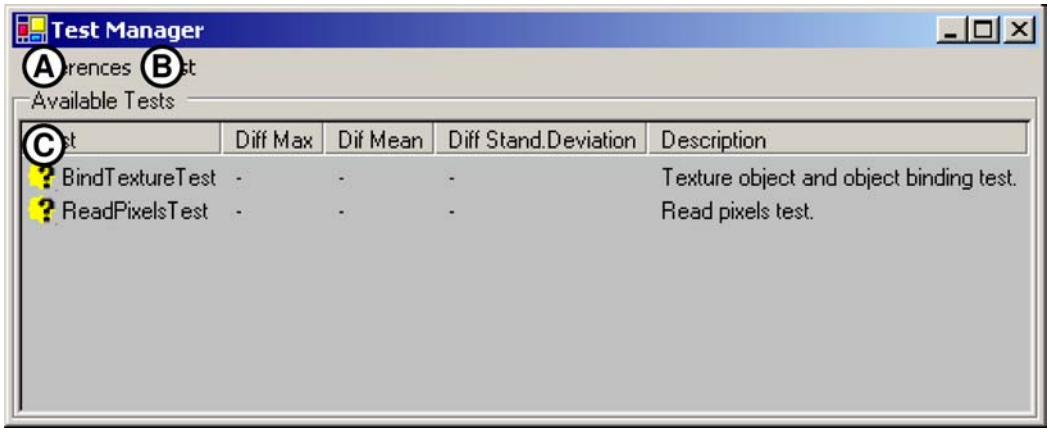


Figure D.2: Main window. It consists of three parts: reference submenu (A), test submenu (B), and test list (C)

References submenu provides a possibility to generate reference images for all tests that does not have generated images yet (item Build). It also allows generation of reference images for either selected tests (item Build selected) or all tests (item Rebuild all) no matter if they have reference images available.

Test submenu provides functions for performing test on either selected test (item Test selected) or all tests (item Test all). It also contains item (item Setup Thresholds) that displays threshold dialog.

Test list contains list of all tests, their names (first column), value of maximum difference (second column), mean value of differences (third column), value of standard deviation (fourth column), and test description (fifth column). Status of the test is visualized in the form of icon in first column. Test status can be one of the five cases:

- Test was performed and result meets specified threshold.
- Test was performed, however, result does not meet specified threshold.
- Test was **not** performed and there is no reference image available.
- Test was **not** performed event thought there are reference images available.
- Test in progress.

Double clicking on single test invokes image dialog. This dialog contains reference image, generated image, and difference image. If there is not reference image available then new one is generated.

Difference image visualizes locations where there is difference between generated and reference image. Image is created as combination of difference images for each color component (i.e., red, green, blue). Image visualizes **location** only, i.e., all pixel color components where generated image differs from original a displayed with maximum available intensity no matter how large the difference actually is.

Appendix E Performance Tests

This appendix contains listing of all functions used for testing. Due to large amount of source code only bodies of testing functions are listed, i.e., listings does not contain function headers and/or used fields (variables for non-managed version) definitions.

E.1 Test 1

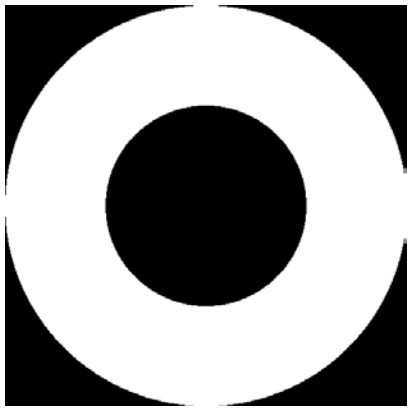


Figure E.1: Test 1 – Output

```
int i = 0, cnt = coords_lenght - 4;
double *p_coord = coords;
glBegin(GL_QUADS);
while (i < cnt)
{
    glVertex2d(coords[i], coords[i + 1]);
    glVertex2d(coords[i + 2], coords[i + 3]);
    glVertex2d(coords[i + 6], coords[i + 7]);
    glVertex2d(coords[i + 4], coords[i + 5]);
    i += 4;
}
glEnd();
glFlush();
```

Figure E.2: Test 1 – non-managed version

```
int i = 0, cnt = coords.Length - 4;
GL.glBegin(GL.GL_QUADS);
while (i < cnt)
{
    GL.glVertex2d(coords[i], coords[i + 1]);
    GL.glVertex2d(coords[i + 2], coords[i + 3]);
    GL.glVertex2d(coords[i + 6], coords[i + 7]);
    GL.glVertex2d(coords[i + 4], coords[i + 5]);
    i += 4;
}
GL.glEnd();
GL.glFlush();
```

Figure E.3: Test 1 – CsGL version

```
int i = 0, cnt = coords.Length - 4;
gl.Begin(RenderPrimitive.GL_QUADS);
while (i < cnt)
{
    gl.Vertex2d(coords[i], coords[i + 1]);
    gl.Vertex2d(coords[i + 2], coords[i + 3]);
    gl.Vertex2d(coords[i + 6], coords[i + 7]);
    gl.Vertex2d(coords[i + 4], coords[i + 5]);
    i += 4;
}
gl.End();
gl.Flush();
```

Figure E.4: Test 1 – This work version

E.2 Test 2

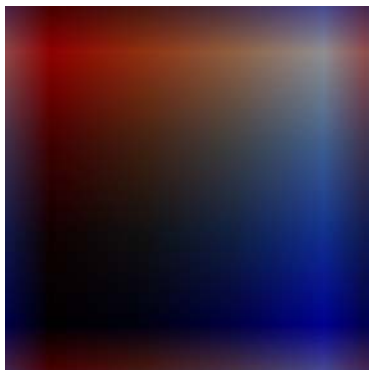


Figure E.5: Test 2 – Output

```
int i = 0;
for(i = 0; i < repeat_count; i++)
{
    glTexImage2D(
        GL_TEXTURE_2D,
        0,
        GL_RGBA,
        TEX_W, TEX_H,
        0,
        GL_RGBA,
        GL_UNSIGNED_BYTE,
        (void*)texture);
    glFlush();
}
```

Figure E.6: Test 2 – non-managed version

```
for(int i = 0; i < count; i++)
{
    GL.glTexImage2D(
        GL.GL_TEXTURE_2D,
        0,
        (int)GL.GL_RGBA,
        TextureSize.Width, TextureSize.Height,
        0,
        GL.GL_RGBA,
        GL.GL_UNSIGNED_BYTE,
        texture);
    GL.glFlush();
}
```

Figure E.7: Test 2 – CsGL version

```
for(int i = 0; i < count; i++)
{
    gl.TexImage2D(
    TexTarget.GL_TEXTURE_2D,
    0,
    TexInternalFormat.GL_RGBA,
    TextureSize.Width, TextureSize.Height,
    0,
    TexFormat.GL_RGBA,
    PixelValueType.GL_UNSIGNED_BYTE,
    texture);
    gl.Flush();
}
```

Figure E.8: Test 2 – This work version

E.3 Test 3

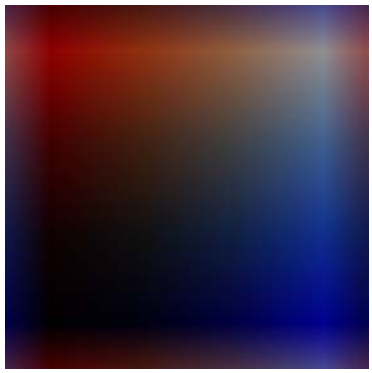


Figure E.9: Test 3 – Output

```
int i = 0;
for(i = 0; i < repeat_count; i++)
{
    glTexImage2D(
        GL_TEXTURE_2D,
        0,
        GL_RGBA,
        TEX_W, TEX_H,
        0,
        GL_RGBA,
        GL_UNSIGNED_BYTE,
        (void*)texture);
    glFlush();
}
```

Figure E.10: Test 3 – non-managed version

```
for(int i = 0; i < count; i++)
{
    GL.glTexImage2D(
        GL.GL_TEXTURE_2D,
        0,
        (int)GL.GL_RGBA,
        TextureSize.Width, TextureSize.Height,
        0,
        GL.GL_RGBA,
        GL.GL_UNSIGNED_BYTE,
        bmpData.Scan0);
    GL.glFlush();
}
```

Figure E.11: Test 3 – CsGL version

```
for(int i = 0; i < count; i++)
{
    gl.TexImage2D(
        TexTarget.GL_TEXTURE_2D,
        0,
        TexInternalFormat.GL_RGBA,
        TextureSize.Width, TextureSize.Height,
        0,
        TexFormat.GL_RGBA,
        PixelValueType.GL_UNSIGNED_BYTE,
        bmpData);
    gl.Flush();
}
```

Figure E.12: Test 3 – This work version

E.4 Test 4

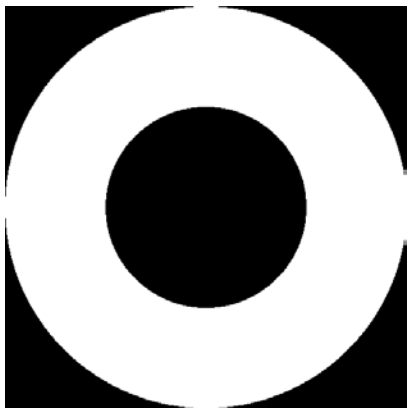


Figure E.13: Test 4 – Output

```
int i = 0;
for(i = 0; i < repeat_count; i++)
    glVertexPointer(2, GL_DOUBLE, 0, (void*)coords);
glFlush();
```

Figure E.14: Test 4 – non-managed version

```
for(int i = 0; i < cntRepeat; i++)
{
    unsafe
    {
        fixed (void *ptr = coords)
        {
            GL.glVertexPointer(2, GL.GL_DOUBLE, 0, ptr);
        }
    }
}
GL.glFlush();
```

Figure E.15: Test 4 – CsGL version

```
for(int i = 0; i < cntRepeat; i++)
    gl.VertexPointer(2, CoordValueType.GL_DOUBLE, 0, coords);
gl.Flush();
```

Figure E.16: Test 4 – This work version

E.5 Test 5

Figure E.17: Test 5 – Output

```
int i;
for(i = 0; i < count; i++)
    glColor4fv((float*)&color);
glFlush();
```

Figure E.18: Test 5 – non-managed version

```
int i;
for(i = 0; i < count; i++)
    GL.glColor4fv(color);
GL.glFlush();
```

Figure E.19: Test 5 – CsGL version without unsafe block

```
unsafe
{
    fixed (float *pColor = &color[0])
    {
        int i;
        for(i = 0; i < count; i++)
            GL.glColor4fv(pColor);
    }
}
GL.glFlush();
```

Figure E.20: Test 5 – CsGL version with unsafe and fixed keyword

```
int i;
for(i = 0; i < count; i++)
    gl.Color4fv(color);
gl.Flush();
```

Figure E.21: Test 5 – This work version using both pinning pointer and structure. Differs only by type of color field

E.6 Test 6

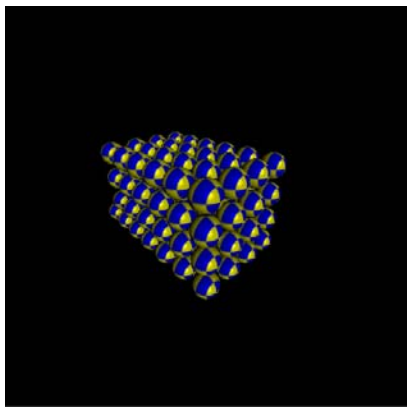


Figure E.22: Test 5 – Output

```
COLOR_4F mat_color;
int count = 0, k, j;
double z, step, y, x;
/* setup view */
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslated(0, 0, -1.8);
glRotated(30, 1, 0, 0);
glRotated(45, 0, 1, 0);
/* setup light position */
glLightfv(GL_LIGHT1, GL_POSITION, (float *)&light_pos);
/* draw spheres */
step = (double)BLOCK_SIDE / sphere_side;
z = (BLOCK_SIDE - step) / 2.0;
for(; count < sphere_cnt; z -= step)
{
    y = (step - BLOCK_SIDE) / 2.0;
    for(k = 0; (k < sphere_side && count < sphere_cnt); k++, y += step)
    {
        x = (step - BLOCK_SIDE) / 2.0;
        for(j = 0; (j < sphere_side && count < sphere_cnt); j++,x+=step,count++)
        {
            glPushMatrix();
            glTranslated(x, y, z);
            /* material */
            mat_color.a = 1.0f;
            mat_color.r = mat_color.g = mat_color.b = 0.7f;
            glMaterialfv(GL_FRONT, GL_DIFFUSE, (float*)&mat_color);
            mat_color.r = mat_color.g = mat_color.b = 1.0f;
            glMaterialfv(GL_FRONT, GL_SPECULAR, (float*)&mat_color);
            glMaterialf(GL_FRONT, GL_SHININESS, 100);
            /* texture */
            glBindTexture(GL_TEXTURE_2D, TEXTURE_ID);
            /* coordinates */
            glInterleavedArrays(GL_T2F_N3F_V3F, 0, tris);
            /* paint */
            glDrawArrays(GL_TRIANGLES, 0, tris_len * 3);
            glPopMatrix();
        }
    }
}
glFlush();
```

Figure E.23: Test 6 – non-managed version

```
COLOR_4F mat_color;
float[] matColor = new float[4];
int count = 0, k, j;
/* setup view */
GL.glMatrixMode(GL.GL_MODELVIEW);
GL.glLoadIdentity();
GL.glTranslated(0, 0, -1.8);
GL.glRotated(30, 1, 0, 0);
GL.glRotated(45, 0, 1, 0);
/* setup light position */
GL.glLightfv(GL.GL_LIGHT1, GL.GL_POSITION, lightPos);
/* draw spheres */
double step = (double)BLOCK_SIDE / sphereSide;
double z = (BLOCK_SIDE - step) / 2.0;
for(; count < sphereCnt; z -= step)
{
    double y = (step - BLOCK_SIDE) / 2.0;
    for(k = 0; (k < sphereSide && count < sphereCnt); k++, y += step)
    {
        double x = (step - BLOCK_SIDE) / 2.0;
        for(j = 0; (j < sphereSide && count < sphereCnt); j++,x+=step,count++)
        {
            GL.glPushMatrix();
            GL.glTranslated(x, y, z);
            /* material */
            matColor[3] = 1.0f;
            matColor[0] = matColor[1] = matColor[2] = 0.7f;
            GL.glMaterialfv(GL.GL_FRONT, GL.GL_DIFFUSE, matColor);
            matColor[0] = matColor[1] = matColor[2] = 1.0f;
            GL.glMaterialfv(GL.GL_FRONT, GL.GL_SPECULAR, matColor);
            GL.glMaterialf(GL.GL_FRONT, GL.GL_SHININESS, 100);
            /* texture */
            GL.glBindTexture(GL.GL_TEXTURE_1D, TEXTURE_ID);
            /* painting */
            unsafe
            {
                fixed(void *pTris = &tris[0])
                {
                    /* coordinates */
                    GL.glInterleavedArrays(GL.GL_T2F_N3F_V3F, 0, pTris);
                    /* paint */
                    GL.glDrawArrays(GL.GL_TRIANGLES, 0, tris.Length * 3);
                    /* pointer to managed data; may become invalid *
                     * due to garbage collection                   */
                    GL.glInterleavedArrays(GL.GL_T2F_N3F_V3F, 0, null);
                }
            }
            GL.glPopMatrix();
        }
    }
}
GL.glFlush();
```

Figure E.24: Test 6 – CsGL version

```
int count = 0, k, j;
/* setup view */
gl.MatrixMode(MatrixMode.GL_MODELVIEW);
gl.LoadIdentity();
gl.Translated(0, 0, -1.8);
gl.Rotated(30, 1, 0, 0);
gl.Rotated(45, 0, 1, 0);
/* setup light position */
gl.Lightfv(Light.GL_LIGHT1, LightPname.GL_POSITION, lightPos);
/* draw spheres */
double step = (double)BLOCK_SIDE / sphereSide;
double z = (BLOCK_SIDE - step) / 2.0;
for(; count < sphereCnt; z -= step)
{
    double y = (step - BLOCK_SIDE) / 2.0;
    for(k = 0; (k < sphereSide && count < sphereCnt); k++, y += step)
    {
        double x = (step - BLOCK_SIDE) / 2.0;
        for(j = 0; (j < sphereSide && count < sphereCnt); j++,x+=step,count++)
        {
            gl.PushMatrix();
            gl.Translated(x, y, z);
            /* material */
            gl.Materialfv(Face.GL_FRONT, MatPname.GL_DIFFUSE,
                new Color4f(0.7f, 0.7f, 0.7f, 1.0f));
            gl.Materialfv(Face.GL_FRONT, MatPname.GL_SPECULAR,
                new Color4f(1, 1, 1, 1));
            gl.Materialf(Face.GL_FRONT, MatSinglePname.GL_SHININESS, 100);
            /* texture */
            gl.BindTexture(TexTarget.GL_TEXTURE_2D, TEXTURE_ID);
            /* coordinates */
            gl.InterleavedArrays(InterleavedArrayFormat.GL_T2F_N3F_V3F,
                0, trisVert);
            /* paint */
            gl.DrawArrays(RenderPrimitive.GL_TRIANGLES, 0, trisVert.Length);
            gl.PopMatrix();
        }
    }
}
gl.Flush();
```

Figure E.25: Test 6 – This work version

Appendix F Generator Tool

Generator tool is an application that was designed in order to simplify and automate process of OpenGL port creation. This tool is not essential part of work and it was not required by assignment. Therefore, this appendix contains just brief reference of data classes including their relations, modifications that were made to ANSI C grammar, description of data file, and quick user manual. This appendix is complementary to chapter 1 that contains description of design and generated files.

F.1 Data Classes and Interfaces

F.1.1 Interfaces

`ILanguageElement`
Public. Construction that forms a complete part of a code that can be defined without need of additional constructions, e.g., function, enumeration type, constant.

`IType`
Public. Construction that can be used as type, such as structure, function pointer, generic type.

`IParTypeVariant`
Public. Construction that can be used for function parameter type. Provides interface for generating data sharing routines and non-managed code cooperation.

F.1.2 Data Classes

`BasicType`
Public. Represents basic data type, i.e., built-in value type or OpenGL type in the case there are no OpenGL types defined in parsed header file. Contains both C and .NET type equivalent. Class implements `IType` and `ITypeVariant` interfaces.

`Contant`
Public. Represents numeric OpenGL constant. Allow comparison by its name. Implements `ILanguageElement` and `System.IComparable` interfaces.

`Enumeration`
Public. Represents enumeration data type whose members are OpenGL constants. Implements `ILanguageElement` and `IType` interface.

`Function`
Public. Function. Implements `ILangugageElement`. Is derived from `FunctionBase`.

FunctionBase
Public. Provides common functionality (e.g., parameter parsing) and data storage (e.g., parameter list) for function and pointer to function.

FunctionParameter
Public. Parameter for function and function pointer. Derived from `TypeInstance`.

Group
Public. Group of language elements. Aimed on GL Extensions where group (extension) is enclosed in `#ifdef`…`#endif` preprocessor directives. Derived from `GroupBase`.

GroupBase
Public. Provides common functionality for groups, such as source file export, data storing, and framework for input parsing.

ImportedGroup
Public. Generic group that is read from data file. Derived from `GroupBase`.

RootGroup
Public. Group of root language elements, i.e., language elements that are not enclosed in any `#ifdef`…`#endif` preprocessor directives. Used for OpenGL function stored in `gl.h` header file. Derived from `GroupBase`.

Type
Public. Generic type. Handles pointers and arrays. Used for function return values, function parameters, structure members, etc. Implements `IType` interface.

Typedef
Public. Solves `typedef` keyword parsing. Used only for input parsing.

TypedefFunction
Public. Function pointer. Implements `IType` interface. Derived from `FunctionBase`.

TypedefStructure
Public. C/C++ Structure. Implement `IType`.

TypeInstance
Public. Instance of type, i.e., type with name. Used for function parameters and structures. Derived from `Type`.

F.2 Modification to ANSI C Grammar

Tool uses finite state machine generated by **flex** [Flex]. State machine is based on ANSI C grammar [ANSI-C]. However, do to simplification reasons this grammar has to be modified. These modifications then allows for a state machine to consume particular preprocessor directives.

Grammar itself was modified by adding of few new regular expressions that allows to recognize selected preprocessor directives: `defined` keyword, new line, `#define`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`, `#include`, `#if`, and general directive, i.e., identifier that starts with hash-mark (‘`#`’). New line is used for searching of the end of C/C++ macros (`#define`). For rules in flex (lex) notation see Figure F.1.

```
"#define"        { count(); return(PREP_DEFINE); }
"#ifdef"         { count(); return(PREP_IFDEF); }
"#ifndef"        { count(); return(PREP_IFNDEF); }
"#else"          { count(); return(PREP_ELSE); }
"#elif"          { count(); return(PREP_ELIF); }
"#endif"         { count(); return(PREP_ENDIF); }
"#include"       { count(); return(PREP_INCLUDE); }
"#if"            { count(); return(PREP_IFDEF); }
\#{L}({L}|{D})* { count(); return(PREP_DIRECTIVE); }
"defined"        { count(); return(PREP_IF_DEFINED); }
\n               { count(); return(NEW_LINE); }
```

Figure F.1: ANSI C grammar modification

F.3 Data File

Tool allow to store read data to data file. This data file uses formatted XML and UTF-8 encoding in order to allow manual modification of file. This section contains list of all tag that are used in the data file. List includes description of tag's attributes and contents.

Tags with exception of `group` tag can appear anywhere inside `group` tag. However, forward reference is **not** allowed, i.e., it is not possible to have an item that refers to another item that was not defined yet. This limitation is based on structure of OpenGL and GL Extensions header file that are the only ones used as input.

`<group name="name" class="class" extension="false"></group>`

Root tag of the file. Contains group items.

`name`	Name of the group.
`class`	Name of a class that shall contain code generated for this group.
`extension`	True, if this group is GL Extension.

`<const name="name"></const>`

Numeric OpenGL constant. Contains numeric value in decimal format.

`name`	Original constant identifier.

`<enum name="name" flags="false"></enum>`

Enumeration type. Contains names of OpenGL constants reference. Each constant is referenced via its name. Name is stored as value of `const` tag, e.g., `<const>GL_AMBIENT</const>`.

`name`	Original name of enumeration.
`flags`	True, if this enumeration type contains flags, i.e., particular enumeration members can be combined via logical operations.

`<basictype name="name" c="c" managed="managed" />`

Basic type. Empty.

`name`	Name. Usually similar to value of `c` attribute.
`c`	Non-managed version of type, e.g., `short`.
`managed`	Managed (MC++) version of type, e.g., `System::Int16`.

`<type base="void" const="false" refs="*" />`

Type. Used for return value of function and for type instance. Empty.

`base`	Reference to defined base type.
`const`	True, if `const` keyword was used in source header files.
`refs`	List of references. Used for creating of pointers to specific type. Asterisk means pointer. Number enclosed in bracket means array with specific length, e.g., '`*[10]`' is array of pointers to particular type that has 10 members.

`<typeinstance name="name"></typeinstance>`

Instance of a type. Contains **single** tag: `type`.

`name`	Name of type instance, e.g., structure member name.

`<fparam name="name" stored="false" allowNULL="true" check="-1"></fparam>`

Function parameter. Contains **single** tag: `typeinstance`.

`name`	Name of function parameter. Must be equal to `name` attribute value of nested `typeinstance` tag.
`allowNULL`	True, if this parameter allows `null` value to be passed. Used only for arrays, pointers, and function pointers.
`check`	String that is used for size checking. Used only for arrays and pointers.

`<typedeffunc name="name"></typedeffunc>`

Function pointer. Contains **single** tag `type` for return type and **multiple** `fparam` tags for function parameters. Order of appearance of `fparam` tags is states order of parameters. Must contain return value tag even if it is `void`, i.e., no return value.

`name`	Original name of function pointer.

`<function name="name"></function>`

Function. Contains **single** tag `type` for return type and **multiple** `fparam` tags for function parameters. Order of appearance of `fparam` tags is states order of parameters. Must contain return value tag even if it is `void`, i.e., no return value.

`name`	Original name of function, e.g., `glBegin`.

F.4 Quick User Manual

Generator tool provides simple GUI that allows to operate with the tool quite easily. Whole GUI consist of two major parts: **main window** and **enum dialog**. It is good to note, that this tool **suits need of its use**, i.e., it is not possible to use it for general header files or OpenGL/GLU/GL Extension header files that contains strange but valid combinations such as function pointer as return type. Tool generates template, i.e., in a few cases (e.g., pointer (string) as return type) the user has to modify implementation of particular function wrapper.

F.4.1 Main Window

Main window (see Figure F.2) is the base component of whole application GUI. It contains complete list of all available groups and information of selected group.

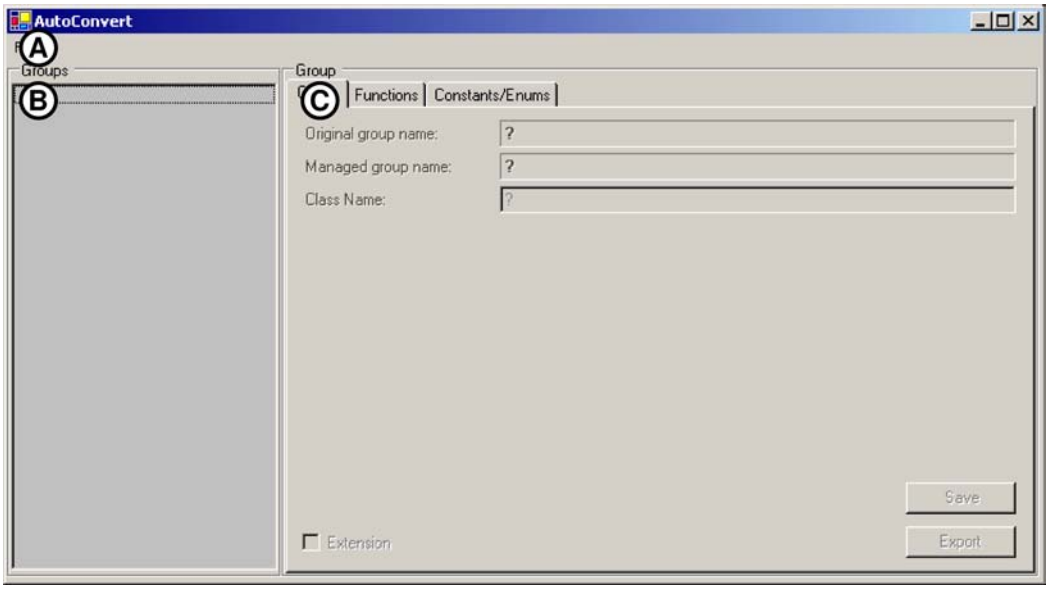


Figure F.2: Main window. It contains file submenu (A), group list (B), and group tabs (C)

File submenu allows reading either group data file (item Open Group) or OpenGL C/C++ header files. When importing data from OpenGL header file, the user has to distinguish between header file with defined OpenGL types such as `GLdouble`, `GLint`, etc. (item Open Header File) and header file without defined OpenGL types (item Open Header File (no GL types)).

Application is capable to read **only C/C++ header files**: `gl.h`, modified `glu.h`, and `glext.h` due to simplification reasons and the fact that only these files contains OpenGL/GLU/GL Extensions functions and contents. Application benefits from structure of mentioned header files and therefore reading of inappropriate C/C++ header files may lead to read failure.

Group list contains complete list of all available groups. Root group is named as --root--. Identifier inside parenthesis is a name of the class that will be used for exporting of group contents, e.g., --root--(GL11) is root group that will be exported as `GL11` class. It is good to not that it is **not** possible to export two groups into single class even thought they both have same class name.

Group tabs contains information of selected group. Currently there are three group tabs available: group tab (Figure F.3) that contains general information of selected group, functions tab (Figure F.4) that contains list of functions, and constants/enums tab (Figure F.5) that contains list of constants including enumeration types of the group.

F.4.1.1 Group Tab

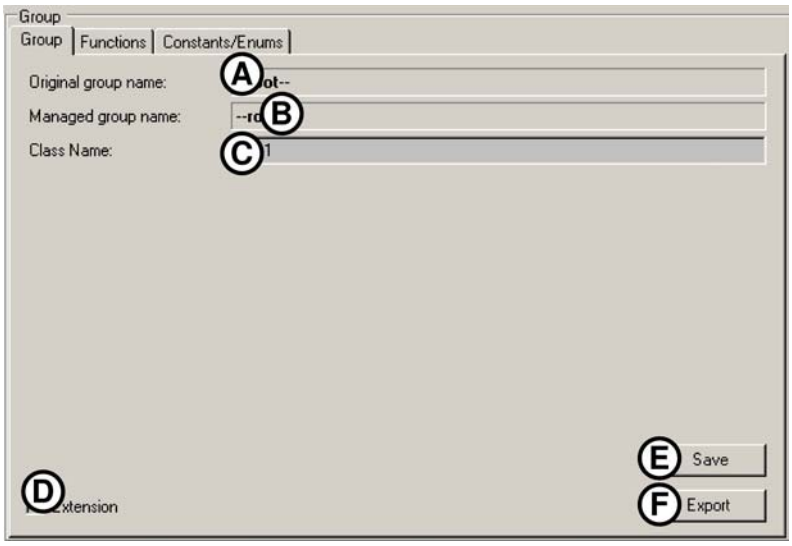


Figure F.3: Group tab. It contains original group name (A), fancy group name (B), class name (C), extension flag (D), save button (E), and export button (F)

Extension flag is checked whenever is current group considered to be GL Extension.

Save button invokes data storing operation. It stores content of current (selected) group into a data file.

Export button invokes data exportation. User selects directory (main file) that will contain exported files. If particular file already exists then it is overwritten without any prompt and user notification. This allows fast and simple updated of already generated files. For details on generated files such as their structure see chapter 1 and chapter 1.

F.4.1.2 Functions Tab

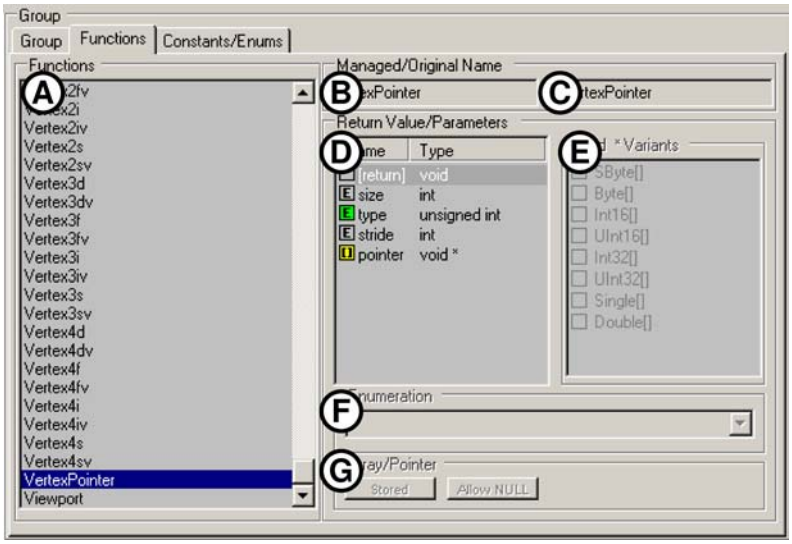


Figure F.4: Functions tab. It contains function list (A), selected function managed name (B), selected function original name (C), parameter list that includes return type (D), possible variants for void pointer (E), enumeration list (F), and parameter options (G)

Parameter list contains list of all parameters of selected function. List contains information about parameter's name (first column) and type (second column). List also contains information about return value type. This information is stored as first item in the list and is marked as [return]. It also describes parameter's status. Status is visualized in a form of small icon and can be one of following six values:

- ☐ General type. This type cannot be an array, enumeration type, or function pointer.
- E Type that **can** be replaced by enumeration type.
- E Type that **is** replaced by enumeration type that is indicated by selected item in **enumeration list**.
- [] Type that is probably an array (i.e., C/C++ style pointer including void pointer) and it is **not stored** in internal structures for later use.
- [] Type that is probably an array and it **is stored** for later use.
- f Type that is a function pointer. This type is not allowed for return value.

Double click on an item that can be replaced by enumeration type and is not already replaced invoked enumeration creation dialog (**enum dialog**) where user can create new enumeration type from available constants. Type replacement (enumeration type) can be selected and/or modified via **enumeration list**.

Void pointer variant list contains possible replacement for void pointer type, i.e., there will be overloads of parameter generated as replacement for parameter's type (void pointer).

Enumeration list contains all defined enumeration types. Currently selected type is used as replacement for type of selected parameter from **parameter list**. User can change current type replacement by choosing difference item from the list. Replacement can be removed by choosing -- none -- item from the list.

Whenever the user changes/removes type replacement the application checks whether is deselected (i.e., previously used) enumeration type used anywhere else. If it is not used then user is asked whether this enumeration type shall be removed from the group permanently.

Parameter options contains modification flags for parameter. These modification influences source code that is exported:

- Pointer that is marked as **stored** (option Stored) is a pointer that is stored inside internal data structures for later use. Storing it inside internal data structures provides garbage collection protection.
- Pointer that is marked as **allow null** (option Allow NULL) allows null value. It influences parameter checking code generation.

F.4.1.3 Constants/Enums Tab

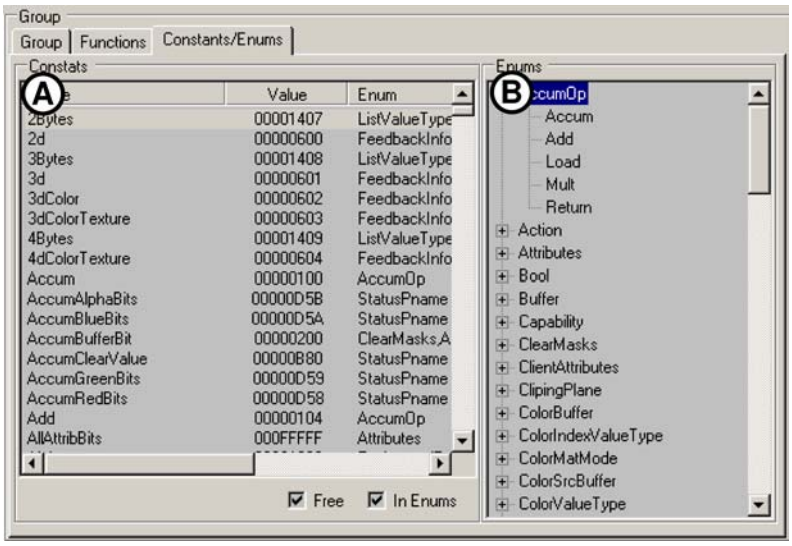


Figure F.5: Constants/Enumeration tab. It contains list of constants (A) and tree of enumeration types (B)

List of constants contains complete list of constants available in the group. Each item of the list consist of constant name (first column), value (second column), and list of enumeration types that uses this constant (third column). A content of the list is modified by options that allow displaying constants that are not used in any enumeration type (checkbox Free) and constants that are used in single or multiple enumeration types (checkbox In Enums).

Tree of enumeration types contains all created enumeration types for current group. Constants that are used in particular enumeration type are listed as children of node (enum). Right click on particular enumeration type displays a simple context menu that allows to rename (item Rename), to modify (item Modify), or to delete (item Delete) it. Modification of enum is performed via **enum dialog**.

If user tries to delete (remove) an enum that is still in use, a message box appear and user is asked to confirm the operation. Otherwise, the enum is removed without any confirmation.

F.4.2 Enum Dialog

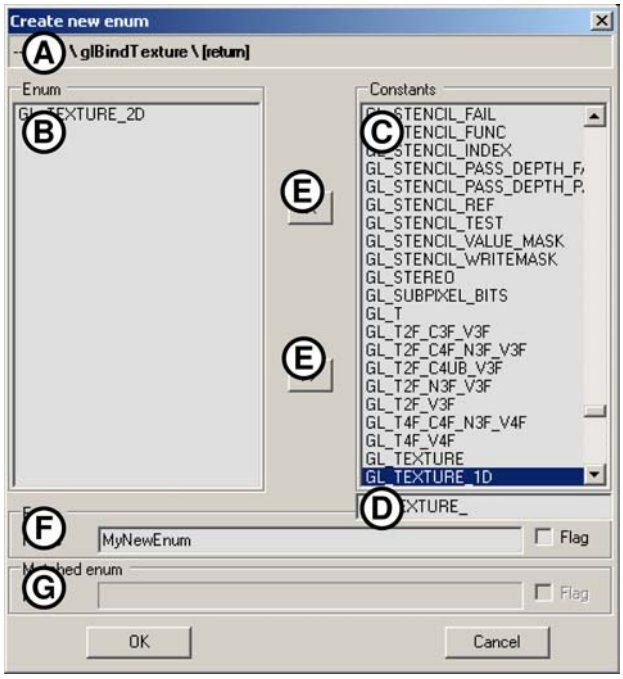


Figure F.6: Enumeration dialog. It consist of enum usage info (A), list of constants that are enum members (B), list of all available constants (C), quick search (D), move buttons (E), enum identifier (F), and matched enum (G)

Enumeration dialog is used for both creation and modification of enumeration types. Dialog performs checks in order to prevent the user from creation of the enum whose contents is similar to already existing one.

Enum usage info contains information either about parameter for which is the enum created (when creating) or about enumeration type that is currently being modified (when modifying). In the case of new enum creation this control contains group name, function identifier, and parameter identifier. Otherwise it contains group name and identifier of enum that is being modified. In both cases is this field read-only.

List of all available constants contains constants that are members of current group and are not listed in **list of constants that are enum members**. User can move constants between list by selecting them in source list and by pressing of either particular move button or <Ins> (<Del> for list of constants) key.

This list also supports quick searching of particular constant simply by typing its identifier in **quick search** edit box. Edit box is usually hidden and becomes visible whenever user provides keyboard input when list of all available constants is active control. It disappears whenever user presses either <Enter> or <Esc> key. Quick search box becomes invisible if the list looses its activity status, e.g., user clicks somewhere else.

Enum identifier contains name of the enum. If this dialog is used for modifying of existing enum, the control is read-only. It also becomes read-only whenever list of enum's constants matches already existing enum. Such "matched" enum is then displayed in **matched enum** control. Option Flag is used to set up enum to be collection of bit values rather then collection of values.