UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science curriculum

Andres Traks

.NET and C++ Interoperation

Master's Thesis (30 ECTS)

Supervisor: Siim Karus, PhD

Tartu 2016

.NET and C++ Interoperation

Abstract: C# is a modern programming language aimed at code robustness and development productivity, but it cannot compete with C++ in performance. The best of both worlds can be had by interoperating between the two languages.

However, C# as a .NET language follows a different paradigm than C++ in many ways. For example, .NET cleans up memory using automatic garbage collection while C++ requires memory to be freed explicitly. Low-level memory access is natural in C++, but is strictly controlled in .NET. Not to mention differences in naming conventions and semantics.

This paper describes two approaches to creating an intermediate layer between .NET and C++ (Platform Invoke and C++/CLI) by making a wrapper interface around C++ code, explains how to overcome memory management and performance issues and introduces a framework for automatically generating the interface. By combining .NET and C++, developers can build their application in a safe and productive manner without sacrificing speed in performance-critical parts of the code.

Keywords: .NET, C++, wrapper, interface, intermediate layer

CERCS: P170 Computer science, numerical analysis, systems, control

Suhtlus .NET-raamistiku ja C++-i vahel

Lühikokkuvõte: Käesolev töö kirjeldab, kuidas realiseerida koostöö kahe erineva programmeerimiskeskkonna, .NET-raamistiku ja programmeerimiskeele C++ vahel.
.NET-raamistikku kasutades on arendaja produktiivsus suurem, kuid C++-is kirjutatud programmidel on parem jõudlus. Seega on eesmärk kasutada tarkvara arendamisel .NET-keeli (nt. C#), kuid jõudlus-kriitilistes kohtades kutsuda välja C++-koodi. Selleks tuleb luua vahekiht kasutades tehnoloogiaid Platform Invoke või C++/CLI.
Töös kirjeldatakse vahekihi ülesehitust, selle loomise etappe ning tutvustatakse projekti, mis loob vahekihi automaatselt. Vahekihi automaatne loomine aitab vähendab töökulu ja parandada veakindlust. Lisaks analüüsitakse viise, kuidas korraldada mäluhaldust ja parandada jõudlust.

Võtmesõnad: .NET, C++, ümbris, liides, vahekiht

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1 Introduction

This paper describes various aspects of interoperation between .NET languages and C++. Also, a software program is introduced to automatically generate the required intermediate layer that acts as a .NET interface wrapping the C++ code. The purpose is to provide a set of guidelines for how the interface should be constructed and to reduce manual work required to build the interface. This makes the interface more performant and reliable.

The guidelines are applied in the automatic interface generator. This generator is different from others in several ways. Firstly, it supports C++/CLI, which performs better on the Windows platform than the more traditional Platform Invoke. There are other projects that support C++/CLI, but they are proprietary. Secondly, it creates a C wrapper around C++ code, which makes the interface independent of the C++ runtime.

1.1 Comparison of .NET and C++

When developing software, it is often preferable to use two different programming languages. High-level languages such as C# simplify development and offer better productivity compared to low-level languages such as C or C++. On the other hand, low-level languages tend to have better performance thanks to having more control over implementation details.

.NET programming languages such as C# also benefit from having a bytecode representation of program code. The bytecode, also called common intermediate language (CIL), can be compiled into machine code on any computer that supports the .NET runtime, making programs written in .NET languages cross-platform. As compilers are improved, there is also a possibility to further optimize a program even after development was completed, because bytecode can be compiled again with an improved compiler. For example, a short method in one .NET program can be inlined inside a method in another program.

There are other benefits to C++ besides performance. For example, C++ has support for inline assembly, which can be used to hand-optimize code for either speed or size. Assembly language provides access to processor-specific instruction sets such as SSE, AVR and others. The *RDTSC* assembly instruction is used here to get precise performance measurements by reading the processor cycle count. C++ also provides direct access to operating system APIs and other existing C++ software libraries.

1.2 .NET and C++ Interoperation

A software developer may prefer to write their application in a .NET language that has high productivity and add a component written in C++ that implements performance-

critical operations. For example, a computer game written in C# may use a high-performance physics engine written in C++.

There are two main methods for calling C++ code from .NET: Platform Invoke (PInvoke) and C++ on Common Language Infrastructure (C++/CLI). C++/CLI is Microsoft's programming language that extends the C++ language with CLI types [1]. PInvoke is a feature of the .NET runtime that allows calling functions from dynamically linked libraries [**?**]. Both methods are described in more detail below in sections 2 and 3. Both methods have pros and cons, so the automatic interface generator project supports generating code for both.

The interface created using one of the above methods receives calls from a .NET consumer, marshals (translates) these calls from managed mode to native mode and calls the targeted C++ library. The interface should use semantics that are familiar to a .NET developer and hide irrelevant C++ implementation details. For example, this means providing non-deterministic memory management, using common .NET naming conventions and making sure that pointers to C++ objects are resolved into .NET wrapper objects.

1.3 Interface Generator

The project described in this thesis is a .NET application that parses C++ header files and generates PInvoke or C++/CLI code that allows the C++ code to be called from .NET. The project was originally created to generate a .NET wrapper for the C++ Bullet physics engine called BulletSharp, but the aim is to extend it for other wrapper projects. Both the generator and the .NET wrapper are hosted on GitHub [2].

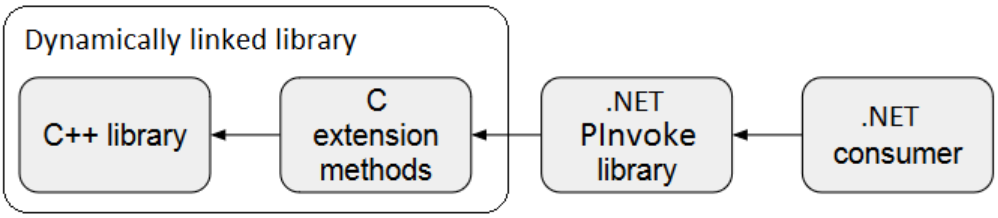
Not all C++ code constructs can be wrapped automatically, because C++ code can sometimes be ambiguous. For example, pointer types carry no information about how many objects they point to. In such cases, manual developer input is required. Ambiguous constructs are discussed in section 4.1. In most other cases, automatic code generation significantly reduces the time to create .NET wrapper libraries and helps to avoid bugs that arise from writing repetitive and yet complex marshalling code.

1.4 Organization of the paper

Section 2 describes the first type of interoperation: *Platform Invoke*. Section 3 describes the second type of interoperation: *C++/CLI*, also known as *implicit Platform Invoke*. Section 4 explains how the automatic code generator parses C++ code. Section 5 explains how C++ constructs are expressed in the .NET interface. Section 6 shows what kinds of transformations are done by the code generator to the C++ code model to create the .NET code model. Section 7 measures the overhead of making calls through the wrapper interface. Section 8 provides more details about what the code generator does.

2 Platform Invoke Project Layout

Figure 1: The Platform Invoke interface between a C++ library and a .NET consumer of that library.



A PInvoke project consists of at least three parts: the original C++ library, a C wrapper around the C++ library and a .NET library that marshals calls from .NET to the C wrapper. The C++ library and the C wrapper can typically be linked into the same dynamically linked library.

It is possible to skip the C wrapper layer and to use PInvoke to call functions from a C++ library directly. However, there is no common specification for the application binary interface (ABI) of C++ methods, making this kind of a solution platform- and compiler-dependent. C++ compilers use *name mangling* to encode the types of parameters and return values of methods into exported symbol names. Since multiple symbol names cannot be specified in a PInvoke call, this solution could only support one type of C++ compiler.

Table 1: Names of methods as exported by different compilers.

Method	Exported name (MSVC)	Exported name (GCC 4)

`// C++-style method`
`class Math`
`{`
`public:`
`  static`
`  int add(int a, int b);`
`}`
	?add@Math@@SAHHH@Z	_ZN4Math3addEii

`// C-style method` `int math_add(int a, int b);`	math_add	math_add

`// C extension method`
`// for C++ method`
`int math_add(Math* obj,`
`  int a, int b)`
`{`
`  obj->add(a, b);`
`}`
	math_add	math_add

A C wrapper solves this problem by assigning an unambiguous symbol name to each method (see table 1). This comes at the cost of type safety in the ABI, since the parameter and return types are no longer encoded in the name. This is not a concern though, because type safety is still provided by the .NET library. There is a performance hit associated with adding another layer, which is measured below.

In case of overloaded methods (when several methods have the same name, but different parameters), an index number can simply be appended to the name (see table 2). The same approach can be taken for C++ methods that have optional parameters (see table 3). For each optional parameter, there will be a C extension method that does not take the optional parameter in addition to one that does.

Table 2: Names of overloaded methods in the C wrapper.

Method	Exported names

`// C++-style method`
`class Math`
`{`
`public:`
`  static int add(int a, int b);`
`  static float add(float a, float b);`
`  static float add(double a, double b);`
`}`
	math_add
	math_add2
	math_add3

Table 3: Exported names of methods with optional parameters.

Method	Exported names

`class Math`
`{`
`public:`
`  static int log(int a, int b = 10);`
`}`
	math_log
	math_log2

2.1 Wrapper Class Layout

A .NET wrapper class contains an opaque pointer (*IntPtr*) to the underlying C++ class instance. The pointer can only be opaque, because .NET does not support type-safe pointers to native C++ classes.

When a method of a wrapper object is called, the call is *marshaled* from managed code to the unmanaged C wrapper and from there to the C++ code. Marshaling includes converting method return values and parameter values from .NET types to their corresponding C++ types. In the C wrapper, the first parameter of a method is a reference to

the C++ class instance (except for constructors and static methods, which do not require an object reference).

The wrapper class can have public constructors that initialize the native pointer. Additionally, if the unmanaged object is initialized by the C++ library, the wrapper class can have an internal constructor, which initializes the native pointer with a given pointer. Wrapper objects initialized directly via the internal constructor will not be responsible for freeing the underlying unmanaged object.

C++ classes that are derived from other classes also have their own .NET wrapper class. The public constructor creates a native object and passes it to the internal constructor of the base class. This makes C++ base class methods also available to the derived class.

Figure 2: Class diagram showing native classes above and their respective .NET wrapper classes below.

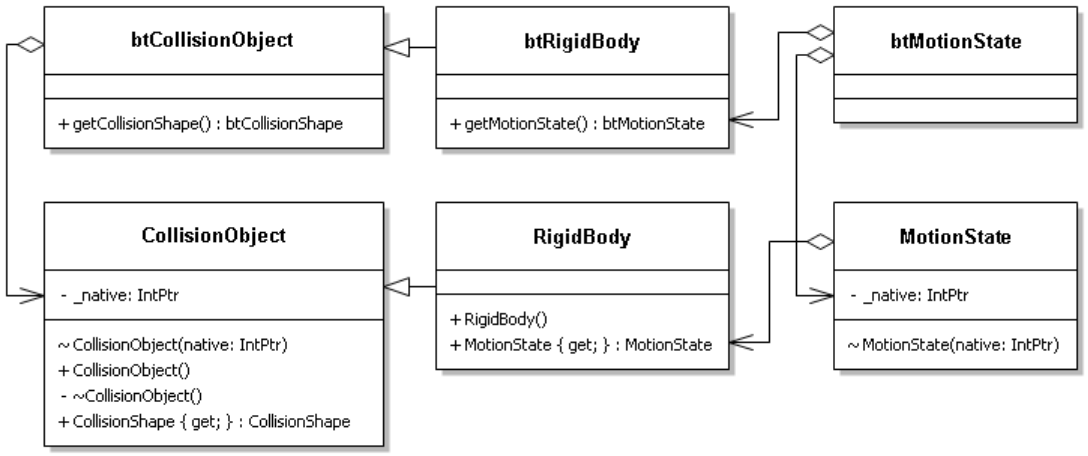


Figure 2 shows a typical class layout for unmanaged classes and their respective managed wrapper classes. *btCollisionObject* is an unmanaged class and *btRigidBody* is an unmanaged class derived from *btCollisionObjec*t. The managed wrapper class *CollisionObject* contains a pointer to the unmanaged *btCollisionObject* instance, which could also be a *btRigidBody* instance.

When a managed *RigidBody* is initialized, the *RigidBody* constructor creates a new native *btRigidBody* instance and passes the native pointer to the internal *CollisionObject* constructor. As an example of how native instance methods are called, when the *MotionState* property is accessed, the accessor method calls *_native->getMotionState()* and receives a *btMotionState* pointer. This pointer is used to initialize a managed instance of the *MotionState* wrapper class by calling the internal constructor of the *MotionState* class with the *btMotionState* pointer.

In general, C++ classes have a one-to-one mapping to their respective .NET wrapper classes. The same is true for namespaces. An exception to this is nested classes, because .NET discourages their use in general [3]. For example, if a *Raycast.RaycastResult* class is nested in a *Raycast* class and the result is definitely required to complete the raycast, then the nesting is discouraged. Instead, the *RaycastResult* class should be nested within the same namespace and on the same level as the *Raycast* class.

Also note that the names of managed classes are different from those of unmanaged classes. The "bt" prefix has been removed, because .NET discourages prefixes to show where the class is in the hierarchy of classes and namespaces. Instead, classes should be grouped into the appropriate namespaces. For example, physics-related classes in the Bullet physics engine should be placed into the *Bullet* namespace instead of having a prefix. So the native fully qualified name of *btCollisionObject* would be *btCollisionObject* while the managed fully qualified name would be *Bullet.CollisionObject*. See also section 6.2 on naming conventions in .NET.

2.2 Platform Invoke DllImport Attribute

In .NET, the *DllImport* attribute is used in conjunction with a function declaration to make a call from a managed application to an unmanaged dynamically linked library (DLL). The *DllImport* attribute specifies the name of the DLL, the name of the function to be called (if it doesn't match the name given in the function declaration), the calling convention and other marshaling options [4]. The Platform Invoke construct is used to marshal calls from .NET into the C wrapper layer. The following code shows how *DllImport* is used to make calls to constructors, static methods and instance methods.

```
1  public class CollisionObject // .NET wrapper class
2  {
3      private IntPtr _native;
4
5      // public constructor
6      public CollisionObject() {
7          _native = btCollisionObject_new();
8      }
9
10     // static method
11     public static float GetMargin() {
12         return btCollisionObject_getMargin();
13     }
14
15     // instance method
16     public void Simulate(float deltaTime) {
17         btCollisionObject_simulate(_native, deltaTime);
18     }
19
20     [DllImport("physics.dll", CallingConvention=...)]
```

```
21      static extern IntPtr btCollisionObject_new();
22      [DllImport("physics.dll", CallingConvention=...)]
23      static extern float btCollisionObject_getMargin();
24      [DllImport("physics.dll", CallingConvention=...)]
25      static extern void btCollisionObject_simulate(IntPtr instance,
            float deltaTime);
26  }
```

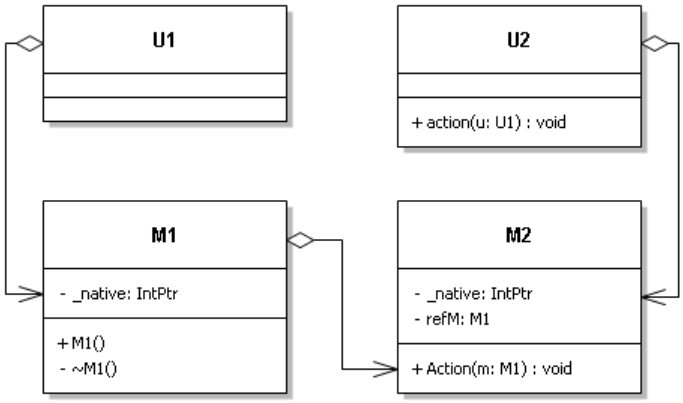
2.3 Object Lifecycle Management

An unmanaged object is allocated into unmanaged heap memory. The wrapper object is usually responsible for managing the lifetime of the unmanaged object and is allocated on the managed heap.

There are two ways in which a wrapper object can be initialised: 1) the wrapper object initializes an unmanaged object by calling its constructor and becomes the owner of that unmanaged object, 2) the wrapper class receives a pointer to an existing unmanaged object, but the unmanaged memory is still owned by the C++ library.

In the first case, the wrapper object is also responsible for calling the destructor of the unmanaged object. What makes this complicated and dangerous is that unmanaged memory is freed deterministically while managed memory is freed non-deterministically.

Figure 3: Scenario where *U1* may be destroyed while being used by *U2*.



Consider a scenario in which a managed wrapper object *M1* allocates an unmanaged object *U1* and a reference to *M1* is passed to another object *M2* in a method call (see figure 3). It is not known whether *U2* will use the reference to *U1* beyond the call to the *Action* method. If *M1* goes out of scope, the .NET garbage collector (*GC*) will call the destructor of *M1*, which in turn calls the destructor of *U1*. Now *U2* has an invalid

pointer to *U1* and may crash if it tries to use that pointer. The solution to this is for the wrapper object of *U2* (*M2*) to maintain a strong reference to *M1*. This prevents *M1* from being garbage collected until *M2* itself is freed.

In practice, it is difficult to know the exact dependencies between objects. If an object is passed as a method argument to another object, it is not clear whether the passed object will be used only for the duration of the method call or whether it will be stored and used later.

In .NET, developers are instructed to explicitly free objects containing unmanaged resources instead of relying on the garbage collector. When the developer forgets to do so or when certain error cases occur, the GC should still be able to release resources properly. The recommended way to release unmanaged resources is to use the *Dispose Pattern* [5].

2.3.1 The Dispose Pattern

The Dispose pattern is the recommended way to free unmanaged resources in .NET applications. A class that contains unmanaged resources should implement the *IDisposable* interface, which consists of a single method *Dispose()*. This method should call the protected *Dispose(bool disposing)* method, which can be overridden in inheriting classes. The *disposing* parameter tells whether *Dispose* was called deterministically (*disposing* = *true*) or whether it was called by the garbage collector non-deterministically (*disposing* = *false*). In the non-deterministic case, the *Dispose* method should not reference any other managed objects, because they may have already been destroyed. The following code shows a class implementing the *IDisposable* interface.

```
1  public class CollisionObject : IDisposable
2  {
3      private IntPtr _native;
4
5      public CollisionObject() {
6          _native = btCollisionObject_new();
7      }
8
9      // Implement IDisposable interface
10     public void Dispose() {
11         Dispose(true); // deterministic destruction
12         GC.SuppressFinalize(this);
13     }
14
15     protected virtual void Dispose(bool disposing) {
16         if (disposing) {
17             // Free any dependent IDisposable objects
18         }
19         btCollisionObject_delete(_native);
20     }
```

```
21
22      ~CollisionObject() { // Finalizer
23          // non-deterministic destruction (unexpected)
24          Dispose(false);
25      }
26  }
```

GC.SuppressFinalize instructs the GC that deterministic destruction has already been done, so the finalizer does not have to be called. Running the finalizer for no reason can hurt performance by putting extra stress on the GC [6]. In case the unmanaged object is owned by the C++ library, the wrapper class should be modified to not free the unmanaged object when *Dispose* is called. If the unmanaged object is always handled by the C++ library, any cleanup routines, including the Dispose pattern, can be omitted entirely.

```
1   public class CollisionObject : IDisposable
2   {
3       private IntPtr _native;
4       private bool _preventDelete;
5
6       public CollisionObject() {
7           _native = btCollisionObject_new();
8       }
9
10      internal CollisionObject(IntPtr native) {
11          _native = native;
12          _preventDelete = true;
13      }
14
15      public void Dispose() {
16          Dispose(true);
17          GC.SuppressFinalize(this);
18      }
19
20      protected virtual void Dispose(bool disposing) {
21          if (!_preventDelete) {
22              btCollisionObject_delete(_native);
23          }
24      }
25
26      ~CollisionObject() {
27          Dispose(false);
28      }
29  }
```

2.4 Object Hashing

In .NET, all objects implement a *GetHashCode* method. The hash returned by *GetHashCode* is used in keyed collections such as dictionaries to determine the location of the values stored in the collection. If two objects are equal, then their hash codes must also be equal. If two objects are not equal, then the hash codes are typically different, but do not necessarily have to be. The hash code should never change during the lifetime of the object, because a changed hash will cause a keyed collection to retrieve an object from another bucket (a slot in the hash table) [7].

All objects in .NET are derived from the *Object* class, which provides a default implementation of *GetHashCod*e. The default implementation may have poor performance, because it uses reflection to read members of the object before computing the hash. For wrapper objects, it is appropriate to use the native pointer as the source of its hash code. In that case, the native pointer should not be cleared to a null pointer in the finalizer. To prevent the native pointer from accidentally being changed, the pointer field can have the *readonly* modifier, which allows the field to only be modified in its declaration or within the object's constructor.

A keyed collection needs to distinguish between objects that have the same hash, because such objects are retrieved from the same bucket. For this purpose, the wrapper class must also override the Equals method in addition to the *GetHashCode* method.

```
1  public class CollisionObject : IDisposable
2  {
3      private readonly IntPtr _native;
4
5      public CollisionObjectWrapper() {
6          _native = collisionObject_new();
7      }
8
9      // override Object.GetHashCode
10     public override GetHashCode() {
11         return _native.GetHashCode();
12     }
13
14     // override Object.Equals
15     public override Equals(object obj) {
16         var colObj = obj as CollisionObjectWrapper;
17         if (obj == null)
18           return false;
19         return this == colObj;
20
21         // incorrect!
22         //return _native == colObj._native;
23     }
24 }
```

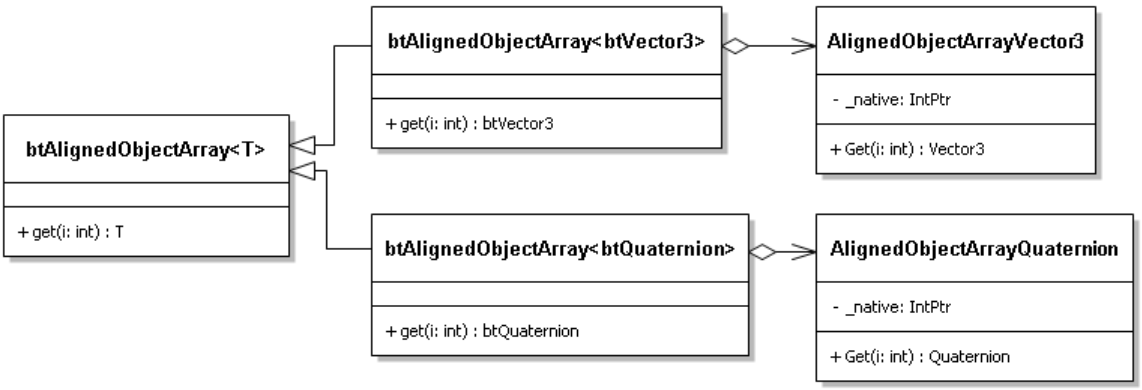
While using the native pointer is appropriate for calculating the hash value, there is a subtle bug that will be introduced by using native pointers to determine object equality. If an unmanaged object is created and a wrapper class is placed into a dictionary, then the object may be explicitly disposed while it is still in the dictionary. In that case, the C++ runtime is free to allocate another unmanaged object into the same memory location. If the new object is also placed into the dictionary, then the two objects will be equal since they have the same native pointer value and retrieval from the dictionary can fail. Therefore, reference equality (==) should be used instead.

2.5 Class Templates

C++ templates are a flexible way to define multiple classes that only differ by few parameters. For example, the Bullet physics library has a template called *btAlignedObjectArray<T>* that can be specialized into an array-like container for any kind of type.

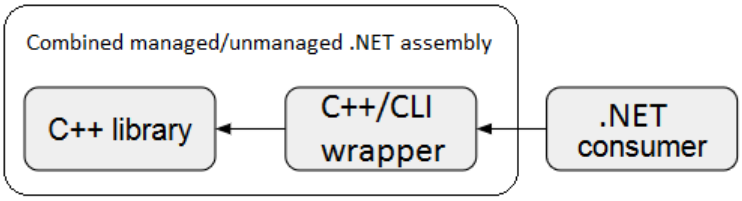
Templates cannot be directly exposed in .NET, because there is no exact equivalent to them in .NET. .NET generics are similar to templates with type parameters, but the types in templates are resolved at compile time whereas the types in generics are resolved at runtime [8]. Because of this, generic classes cannot be used to wrap C++ templates. Instead, whenever a class template specialization is used in a C++ header, this specialization should be wrapped by a concrete class in .NET. For example, instead of wrapping the *btAlignedObjectArray<T>* class with a generic *AlignedObjectArray<T>* class, each specialization has a separate wrapper class like *AlignedObjectArrayVector3* and *AlignedObjectArrayCollisionObjec*t.

Figure 4: Wrapper classes for C++ template specializations.



.NET generics do not support non-type parameters (e.g. numbers) either, so a C++ template specialization such as *btVector<3>* must be wrapped by a concrete class like *Vector3*.

3 C++/CLI Project Layout



In the C++/CLI programming language, both unmanaged C++ and managed .NET wrapper code can be included within the same assembly (.dll or .exe file). This is called *implicit PInvoke* [9] and it allows for some optimizations that are not available with *explicit PInvoke*, since the original C++ code is also available to the compiler. C++/CLI therefore has slightly smaller overhead of marshalling (see performance benchmarks in section 7).

A disadvantage of the implicit PInvoke approach is that C++/CLI is only implemented for the Windows platform, while explicit PInvoke is also available for Linux and BSD platforms.

C++/CLI can be more flexible than C# as it supports macros. In BulletSharp C++/CLI, C++ macros are used to integrate with different graphics libraries. A differently targeted version of the library can be compiled by changing the macro definitions.

```
#if GRAPHICS_MOGRE
#define Matrix Mogre::Matrix4^
#elif GRAPHICS_OPENTK
#ifdef BT_USE_DOUBLE_PRECISION
#define Matrix OpenTK::Matrix4d
#else
#define Matrix OpenTK::Matrix4
#endif
#elif GRAPHICS_SHARPDX
#define Matrix SharpDX::Matrix
#endif
```

In C++/CLI, types can also be aliased. For example, a managed library can switch between using floats and doubles by changing a single preprocessor definition.

```
#define BT_USE_DOUBLE_PRECISION 1
#if BT_USE_DOUBLE_PRECISION
typedef double btScalar;
#else
typedef float btScalar;
#endif
```

C# does not support aliasing of types. It is possible to define an implicit conversion operator to convert a type into the required type in the targeted graphics framework,

but this conversion has some overhead compared to the C++/CLI macro and typedef approaches.

In the case below, when getting a vector value from a wrapper library, an intermediate *Vector3* struct from the wrapper library is first allocated and then implicitly converted to a *Vector3* struct from XNA, which is the targeted graphics library.

```
public struct Vector3 // defined in the wrapper library
{
  public float X,Y,Z;

  public Vector3(float x, float y, float z) {
    X = x; Y = y; Z = z;
  };

  // implicit conversion operator
  public static implicit operator Microsoft.Xna.Framework.Vector3(
      Vector3 value)
  {
    return new Microsoft.Xna.Framework.Vector3(value.X, value.Y,
        value.Z);
  }
}
```

In C++/CLI, a macro can be used to initialize the required vector type directly without overhead.

```
#if GRAPHICS_XNA
  #define VECTOR3(x,y,z) Microsoft::Xna::Framework::Vector3(x,y,z)
#elif GRAPHICS_GENERIC
  // no target library, use own vector struct
  #define VECTOR3(x,y,z) Vector3(x,y,z)
#endif

Vector3 CollisionObject::AngularVelocity::get() {
  return VECTOR3(
    _native->m_angularVelocity.x,
    _native->m_angularVelocity.y,
    _native->m_angularVelocity.z);
}
```

3.1 Wrapper Class Structure

The .NET wrapper class structure in C++/CLI is the same as in PInvoke. However, the code layout is different. As is typical in C++, there is a header file and a source file. It is useful to review the class structure here, because it is important that the automatic code generator produces code that looks consistent for each wrapper class.

```
#include <btCollisionObject.h>

// class declaration in the header file
public ref class CollisionObject
{
internal:
  btCollisionObject* _native;

  CollisionObject(btCollisionObject* native);
  ~CollisionObject(); // destructor
  !CollisionObject(); // finalizer

public:
  CollisionObject();

  static float GetMargin(); // static method
  void Simulate(float deltaTime); // instance method
}
```

Note that in C++/CLI, access specifiers of destructors and finalizers of managed classes (*~CollisionObject* and *!CollisionObject*) are ignored, so they do not have to be written out [10, 11].

As C++/CLI supports native C++, the *DllImport* construct is not necessary (although it can be used). Instead, methods can be called directly on the instance pointer.

```
#include <CollisionObject.h>

// class definition in the source file
CollisionObject::CollisionObject(btCollisionObject* obj) {
  _native = obj;
}

CollisionObject::CollisionObject() {
  _native = new btCollisionObject();
}

float CollisionObject::GetMargin() {
  btCollisionObject::getMargin();
}

void CollisionObject::Simulate(float deltaTime) {
  _native->simulate(deltaTime);
}
```

3.2 Dispose Pattern in C++/CLI

The Dispose Pattern is implemented differently in C++/CLI than in PInvoke, but the resulting .NET assembly will be equivalent to the PInvoke version. Whenever a managed class in C++/CLI has a destructor, the compiler will implement the *IDisposable* interface automatically [12, 13] This means that the *Dispose* method cannot be explicitly written and the " : IDisposable" base interface does not have to be explicitly specified by the programmer. The compiler itself inserts a *Dispose()* method, which first executes the body of the destructor and then calls *GC.SuppressFinalize* at the end to prevent the finalizer from being called. The following code shows a C++/CLI class implementing the *IDisposable* interface.

```
1  public ref class CollisionObject
2  {
3      btCollisionObject* _native;
4
5  public:
6      CollisionObject() {
7          _native = new btCollisionObject();
8      }
9
10     ~CollisionObject() {
11         // free any managed resources here
12         // call finalizer to free unmanaged resources
13         this->!CollisionObject();
14     }
15
16     // Finalizer
17     !CollisionObject() {
18         delete _native;
19     }
20 }
```

Unmanaged resources are freed in the finalizer and the destructor can call the finalizer to do that. Managed resources can only be freed in the destructor and not the finalizer. The reason is that in case of non-deterministic destruction, the GC calls the finalizer, but dependent objects may then already have been destroyed, so they cannot be referenced.

The destructor is always merged into the *public Dispose()* method and the finalizer is always *protected* regardless of the access specifier written in the code.

4 C++ Parsing

The first step in automatically generating wrapper code is to parse the C++ code of the target library and create a model of the library interface. In C++, header files (.h) specify the interface of the library and the source files (.cpp) contain the implementation. In order to generate wrappers for a C++ library, only the header files need to be parsed, because the implementation does not contain any information that is directly relevant.

A C++ project can be parsed using a C++ compiler with a frontend that exposes the program structure before final compilation into an executable. For instance, Clang is a frontend for the LLVM compiler that has an API allowing access to the AST [14]. Clang is more suitable than GCC for parsing code, because it was designed from the start to have a frontend API and it does not optimize the AST before exposing it [15].

Parts of the code that need parsing are class members such as methods, field, aliases and nested classes [16]. The application recurses over the header files in the directories and subdirectories of the specified C++ library. Clang is then given the C++ compiler options that were used when compiling the target C++ library. The compiler options include preprocessor directives (e.g. "-DNDEBUG") and include directories (e.g "-Isrc/Extras/Serialize"). Then the header files are preprocessed one by one.

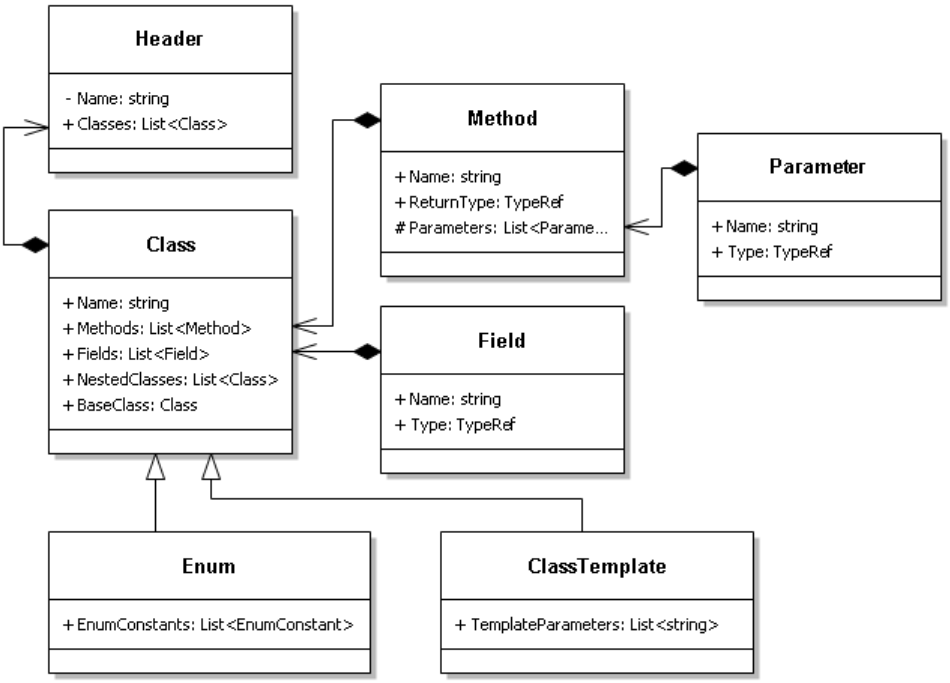
Once preprocessing is complete, a special type of callback called a *visitor method* is passed to Clang. Clang will call this method with an argument that is a cursor to the current code segment. For each header file, there is a top-level code segment which is called the *translation unit*. If the visitor method returns the value *ChildVisitResult.Recurse*, then the visitor method is called again with cursors to child elements of the code segment. For example, if the cursor points to a class, then returning *Recurse* will cause the method to be called again with a code segment cursor pointing to a field, method or some other member of that class. Returning *ChildVisitResult.Continue* instead causes the cursor to be moved past the end of the class, skipping the child segments. This can be used to speed up parsing of segments that are known to not be useful later (e.g. classes or methods that have been explicitly excluded).

The class structure is stored in-memory in a C++ code model (see figure 5). In the next pass, this model is transformed into a .NET code model.

4.1 Ambiguous C++ Methods

In C++, there are some constructs that do not translate well to .NET because of ambiguity and require manual input from the developer to be resolved. These include pointers to arrays, which do not have a known size, and method parameters that have unknown direction.

Figure 5: C++ code model.



4.1.1 Arrays

In C++, arrays of fixed size can be referenced by a pointer.

```
1 void init(char (*array)[10]);
```

In many cases though, the size of an array is known only at run-time, so arrays are referenced by a pointer, with the length of the array stored separately.

```
1 void init(char* array, int length);
```

This can lead to ambiguity in method calls. It is possible that the *array* pointer points to an array of predetermined, but unspecified size and the length parameter may be completely unrelated. The pointer might also point to a single value only and not an array at all.

In .NET, arrays are objects and always have a *Length* property associated with them. To pass a native C++ array to .NET, manual input is required for the automatic code generator to know how to determine the length of the array.

4.1.2 Parameter Marshaling Direction

In C++, a *const*-qualified method parameter can only point to an input structure, because it cannot be written to. On the other hand, a non-*const* pointer parameter could either point to an input structure, an output structure or both at the same time.

The following example demonstrates why it is not possible to distinguish between input-output and output-only parameters based on C++ method declarations.

```
void normalizeVector(
  const float* vector, // input-only
  float* normalized);  // output-only

void normalizeVectorInPlace(
  float* vector); // input-otput
```

The *normalized* parameter holds the output of the normalizeVector method and is never read from, but this cannot be automatically determined from the method signature. Interpreting all non-*const* parameters as input-output by default ensures correct operation, but unnecessary parameter marshaling will decrease performance. If the parameter is output-only, then there is no need to marshal it to the callee. Likewise, if the parameter is input-only, then there is no need to marshal it back to the caller.

To marshal such parameters without a performance hit, the marshaling direction has to be manually specified for the interface generator. It must be kept in mind that an overridden method may reinterpret a non-*const* output-only parameter as input-output [17].

4.2 Inferring Information from Doxygen Documentation

Doxygen is a C++ documentation tool that can provide hints about ambiguous method declarations. For example, a method can document the direction of attributes with *[in]*, *[out]* or *[in,out]*.

```
/**
 *\param[in]  vector      input vector
 *\param[out] normalized  output vector
 */
void normalizeVector(const float* vector, float* normalized);
```

```
/**
 *\param[in, out]  vector vector to normalize
 */
void normalizeVectorInPlace(float* vector);
```

The *deprecated* keyword in Doxygen hints to the library user that the method should not be used in the future and could be removed [18].

```
/**
 *\deprecated Method will be removed soon.
 *\param[in, out]  vector vector to normalize
 */
void normalizeVectorInPlace(float* vector);
```

The *deprecated* keyword can be translated into a .NET *ObsoleteAttribute*. This marks the method in the wrapper class also obsolete.

```
/// <param name="vector">vector to normalize</param>
[ObsoleteAttribute("Method will be removed soon.")]
void NormalizeVectorInPlace(float[] vector);
```

5 Data Marshaling

This section describes how C++ code constructs can be expressed in .NET and how data is passed from .NET to C++.

5.1 Basic Types

Basic types such as integers, floats and pointers (IntPtr) can generally be passed from .NET to C++ without any special marshaling. The only thing that needs to be done is to rename certain types from their C++ names to C# names, e.g. *void m(unsigned int a)* → *void M(uint a)* (see table 4).

In any .NET language, there are two sets of basic type names that are aliases of each other. One set comes from the particular .NET language used, which is familiar to developers that may have experience with the non-.NET version of the language, and another set of type names from the *System* framework, which is recognizable in whatever language the .NET library is used [19] (see table 4).

Table 4: Basic C++ types and the corresponding type names in C# and .NET.

C++ type	Corresponding C# type	.NET framework type
`unsigned long`	`ulong`	ULong
`unsigned int`	`uint`	UInt32
`unsigned short`	`ushort`	UInt16
`unsigned char`	`byte`	Char
`long`	`long`	Int64
`int`	`int`	Int32
`short`	`short`	Int16
`char`	`sbyte`	SByte
`bool`	`bool`	Boolean
`float`	`float`	Single
`double`	`double`	Double

When writing type names in C#, *short* is the preferred alias of the *Int16* type in the *System* namespace in the .NET framework. But in case of method names in public .NET libraries, it is expected that the .NET framework names are used, which are common in all .NET languages. For example, *readShort()* → *ReadInt16()* (see table 5).

Table 5: Examples of C++ method signatures and corresponding signatures in .NET.

C++ method signature	Corresponding .NET signature
`void m(unsigned long a);`	`void M(ulong a);`
`unsigned int read_uint();`	`uint ReadUInt32();`
`float readFloat();`	`float ReadSingle();`

5.1.1 Booleans

Booleans have a different binary representation in .NET than they do in Microsoft's implementation of C++. The size of C++ booleans is 1 byte while .NET booleans are 4 bytes. To convert the return value of an unmanaged function from C++ to .NET, the PInvoke signature must have the *MarshalAs* attribute as shown below [20].

```
1 [DllImport(...)]
2 [return: MarshalAs(UnmanagedType.I1)]
3 static extern bool btCollisionObject_isActive(IntPtr obj);
```

5.2 Array Types

There are two ways to marshal arrays between .NET and C++. One is to convert a native array into a .NET array when reading it (and vice versa when modifying the array). Whenever the array is accessed, the native array is copied into a managed .NET array. This is quite inefficient if only part of the array needs to be accessed. It may also be confusing to the library consumers, because any changes made to the .NET array are not immediately reflected in the native array and changes to the native array are not reflected in the .NET array. The reason is that a basic .NET array cannot be made to point directly to a native C++ array. A .NET array is an independent object, so copying the array data back and forth is the only way to access it.

The following is a C# property that wraps an unmanaged integer array using a .NET array. Any changes made to the .NET array are only applied when the array is written back in the *set* method.

```
1  private int _length;
2
3  public int[] Weights
4  {
5    get
6    {
7      int[] value = new int[_length];
8      Marshal.Copy(Node_getWeights(_native), value, 0, _length);
9      return value;
10   }
11
```

```
    set
    {
      Marshal.Copy(value, 0, Node_getWeights(_native), _length);
    }
}
```

Another, more complicated way to marshal arrays is to define a wrapper object for the array type. For example, the following is a wrapper class for signed integers, which can access unmanaged memory directly and behaves semantically similarly to a .NET array.

```
public class IntArray {
  private IntPtr _ptr;
  private int _length;

  internal IntArray(IntPtr ptr, int length) {
    _ptr = ptr;
    _length = length;
  }

  public int this[int index]
  {
    get
    {
      return Marshal.ReadInt32(_native, index * sizeof(int));
    }

    set
    {
      Marshal.WriteInt32(_native, index * sizeof(int), value);
    }
  }
}
```

This wrapper class can then be returned from the get method of the required property and the changes made to it are applied immediately. There is no need for a setter method unless the entire array actually needs to be set. The IntArray instance can also be cached to prevent it from being recreated on every access as shown below.

```
public class WeightedObject {
  private IntPtr _native;

  private int _length;
  private IntArray _weights;

  /* Initialization, etc. */

  public IntArray Weights
  {
```

```
    get
    {
      if (_weights == null) {
        _weights = new IntArray(Node_getWeights(_native), _length);
      }
      return _weights;
    }
  }
}
```

The IntArray class can be made even more similar to a regular *int[]* array by implementing the *IList< T>* interface. This interface exposes the *Count* of elements, methods like *Contains*, *CopyTo*, *IndexOf* and also *GetEnumerator* for enumerating elements in the array. Since the array has a fixed length, the *IList* methods *Add*, *Remove*, etc. should throw a *NotSupportedException*.

5.3 Fields

Fields can be marshaled in two ways: by value and by reference. When marshaling by value, accessor methods are created in the C wrapper for getting and setting the field. Consider the following C++ class that has the field *number* of a basic integer type and the field *vec* of struct type.

```
// C++ code
struct Vector3 { float x, y, z; };

class CppClass {
public:
    int number;
    Vector3 vec;
};
```

```
// Marshal field number by value (C wrapper)
int CppClass_getNumber(CppClass* obj) {
    return obj.number;
}

void CppClass_setNumber(CppClass* obj, int value) {
    obj.number = value;
}

// Marshal field vec by reference
Vector3* CppClass_getVec(CppClass* obj) {
    return &obj->vec;
}
```

Since an integer type fits into a method parameter, which is typically 32 or 64 bits, it makes sense to marshal it by value. The struct on other hand is larger and must be passed by reference.

When marshaling by reference, it is not strictly necessary to provide a setter, because the reference to the field provided by the getter can be used to do the assignment. However, a setter can make use of the copy assignment operator [21] in C++ to assign a value to the entire field.

```
// Marshal field vec by reference (C wrapper)
void CppClass_setVec(CppClass* obj, Vector3* value) {
  obj->vec = *value;
}
```

```
// .NET wrapper methods calling the C wrapper
public int GetNumber() {
  return CppClass_getNumber(_native);
}

public void CppClass_SetNumber(int value) {
  CppClass_setNumber(_native, value);
}

public Vector3 GetVec() {
  return new Vector3(CppClass_getVec(_native));
}

public void CppClass_SetVec(Vector3 value) {
  CppClass_setVec(_native, value._native);
}

[DllImport(...)]
static extern int CppClass_getNumber(IntPtr obj);
[DllImport(...)]
static extern void CppClass_setNumber(IntPtr obj, int value);
[DllImport(...)]
static extern IntPtr CppClass_getVec(IntPtr obj);
[DllImport(...)]
static extern void CppClass_setVec(IntPtr obj, IntPtr value);
```

5.4 Marshaling Using .NET Struct Types

For C++ structures with value semantics, a .NET wrapper class can be created that also has value semantics.

```
// C++ structure
struct Vector3 { float x, y, z; }
```

```
// .NET wrapper struct
[StructLayout(LayoutKind.Sequential)]
struct Vector3 { float X, Y, Z; }
```

Marshaling can be done using the *Marshal.PtrToStructure* method in the .NET framework.

```
// .NET wrapper
public Vector3 Vec
{
    get
    {
        return (Vector3)Marshal.PtrToStructure(CppClass_getVec(
            _native), typeof(Vector3));
    }

    set
    {
        Marshal.StructureToPtr(value, CppClass_getVec(_native), false
            );
    }
}
```

Marshal.PtrToStructure and *Marshal.StructureToPtr* can be used if the unmanaged and managed structures match exactly. These methods do a plain memory copy between the managed and unmanaged heap. If the structure includes references to other objects, then more complex marshaling is required, e.g. the *ICustomMarshaler* interface (see below).

The .NET runtime is free to move around fields in structures for optimization purposes. To make structs suitable for marshaling, the *StructLayout* attribute must be used to order fields sequentially to match the layout of the unmanaged struct [22]. The *Pack* field of the *StructLayout* attribute controls the alignment of fields in the struct.

Sometimes it is necessary to specify the exact location of fields in unmanaged memory. For example, to emulate C++ unions, two fields must have the same location. To do that, the struct must have the *LayoutKind.Explicit* attribute and each field must have a *FieldOffset* attribute.

```
[StructLayout(LayoutKind.Explicit)]
struct StructWithUnion {
    [FieldOffset(0)]
    float RegularField;

    // UnsignedField and SignedField form a union
    [FieldOffset(4)]
    uint UnsignedField;
    [FieldOffset(4)]
    int SignedField;
}
```

5.4.1 Passing Struct Parameters

Struct arguments can be passed either by value or by reference. Passing large structs by reference is faster, because passing by value requires the entire struct to be copied. However, it is not always possible to copy structs by reference. For example, while properties act like fields, they are in fact methods, but it is not possible to obtain a direct reference to the return value of the accesor method of a property.

In C# (PInvoke), it is possible to provide an overload to a method with struct parameters. This way the method can be passed either by value or by reference, if possible.

```
void SetWorldTransform(Matrix value) {
  //btCollisionObject_setWorldTransform(ref value); // faster
  SetWorldTransform(ref value);// simpler
}

// overload with ref parameter
void SetWorldTransform(ref Matrix value) {
  btCollisionObject_setWorldTransform(ref value);
}

SetWorldTransform(obj.WorldTransform);
SetWorldTransform(ref _worldTransform);
```

The problem with this approach is that some .NET languages such as C++/CLI do not support overloading methods that differ only by the ref specifier of a parameter. Also, if a C++/CLI program tries to call such an overloaded method in a library written in C#, the compiler wil give an error that the method call is ambiguous. There is no way to fix the ambiguity by specifying which overload was meant. CLS-complicance, which is recommended for public .NET libraries, also dictates that such overloads should not be used, even in C# libraries. See section 6.1 for more details about CLS-compliance.

Instead of overloading, another way to provide ref/non-ref methods is simply to append "Ref" to the method name. For example, the signature *void SetWorldTransform(ref*

Matrix value) becomes *void SetWorldTransform**Ref**(ref Matrix value)*. This does not work for constructors, which cannot be renamed. Also, accessor methods in properties must be named *get*/*set* and they inherently pass parameters by value, so neither of the solutions work there.

5.5 ICustomMarshaler Interface

For classes, arrays and boxed value types, the *ICustomMarshaler* interface can be used to provide custom marshaling. The interface consists of methods to marshal objects in both directions, cleanup methods and an unused (legacy) *GetNativeDataSize* method. A class that implements the *ICustomMarshaler* interface must also implement a static *GetInstance* method, which initializes the marshaler.

```
1  public class DispatcherMarshaler : ICustomMarshaler
2  {
3      public object MarshalNativeToManaged(IntPtr ptr) {
4          return new Dispatcher(ptr);
5      }
6
7      public IntPtr MarshalManagedToNative(object obj) {
8          return (obj as Dispatcher)._native;
9      }
10
11     public void CleanUpNativeData(IntPtr ptr) {
12         Dispatcher_delete(ptr);
13     }
14
15     public void CleanUpManagedData(object obj) {
16         (obj as Dispatcher).Dispose();
17     }
18
19     public int GetNativeDataSize() { return -1; }
20
21     public static ICustomMarshaler GetInstance(
22         string cookie) {
23         return new DispatcherMarshaler();
24     }
25 }
```

The marshaler type is specified in the PInvoke signature for either return values or parameter values.

```
[DllImport(...)]
[return: MarshalAs(UnmanagedType.CustomMarshaler, MarshalTypeRef =
    typeof(DispatcherMarshaler))]
static extern Dispatcher World_getDispatcher(IntPtr ptr);

[DllImport(...)]
static extern void btCollisionWorld_getDispatchInfo(IntPtr obj,
    [MarshalAs(UnmanagedType.CustomMarshaler, MarshalTypeRef = typeof
        (DispatcherMarshaler))]
    DispatcherInfo info);
```

5.6 Over-aligned Data Structures for SSE

Streaming SIMD Extensions (SSE) is an instruction set for high performance computing on the x86 architecture. Code that makes use of SSE instructions must ensure that parameters to SSE calls are aligned on 128-bit (16-byte) memory boundaries. This is done to optimize data access inside the CPU. If data is not aligned to 16 bytes, then SSE instructions will trigger an *AccessViolationException* in .NET. An *AccessViolationException* indicates corrupted process state and so it cannot be handled in a try-catch block by the user unless the *HandleProcessCorruptedStateExceptions* attribute is set for the calling method [23, 24].

In C++, data can be aligned using the *align* keyword [25] and class instances can be aligned using the *__declspec(align(m))* specifier in the MSVC compiler. However, these specifiers do not hold for memory allocated dynamically using the *new* construct, because *new* cannot allocate aligned memory. In fact, the MSVC compiler will give a warning if classes requiring alignment are allocated using *new* [26]. The solution is to override the *new* and *delete* operators to use the *_aligned_malloc* and *_aligned_free* functions [27].

Alignment is required when passing data to an unmanaged method that works with SSE instructions. However, there is no explicit way to align data in .NET. A common workaround for allocating *n* bytes of data aligned on an *m*-byte boundary is to allocate *n+m-1* bytes of memory (e.g. using the *Marshal.AllocateHGlobal* method in the .NET framework) and then take the start of the next aligned 16-byte block from the returned address [28].

```
const int n = 64; // reqested data size
const int m = 16; // alignment, power of 2
IntPtr allocAddr = Marshal.AllocHGlobal(n + m - 1);
IntPtr alignedAddr =
  new IntPtr((allocAddr.ToInt64() + m - 1) & ~(m - 1));
```

The allocated memory must be freed later using the original start address of the block.

```
Marshal.FreeHGlobal(allocAddr);
```

If a method has an output parameter pointing to the result of an SSE instruction, the wrapper class needs to allocate a temporary variable into an aligned location and copy the final result using non-SSE instructions. In the following C wrapper code, a macro is used to create a temporary variable that is aligned to 16 bytes. The temporary variable is later written to the final variable using another macro.

```
void btMotionState_getWorldTransform(btMotionState* obj, btScalar*
    worldTrans)
{
  TRANSFORM_DEF(worldTrans);
  obj->getWorldTransform(TRANSFORM_USE(worldTrans));
  TRANSFORM_DEF_OUT(worldTrans);
}
```

If a class contains members requiring alignment for SSE, then instances of that class must be allocated to aligned memory. In wrapper class constructors, the unmanaged class must be initialised with an aligned allocator. A wrapper class should not try to instantiate an unmanaged C++ class if it has the *__declspec(align(m))* specifier, but no overloaded *new* and *delete* operators.

5.7 Callback Methods

C++ code may use callbacks, which can be marshaled from C++ to C#. In the code below, *cb* is a native callback with the *callback* signature. A consumer can provide a callback using the *nativeSetCallback* method. The callback is then called with a parameter.

```
typedef bool (*callback) (M* obj);
callback cb;

void nativeSetCallback(callback fnPtr) {
  cb = fnPtr;
}

cb(new M());
```

The callback can be set up in .NET by converting a delegate into a function pointer and passing the pointer to the *nativeSetCallback* method. If the method signature includes types that cannot marshalled directly, then another method is required that accepts the native signature and marshals the parameters to the final managed method. For example, the opaque IntPtr *m* needs to be converted into a managed object, so the *CallbackUnmanaged* method is provided that marshals the call to *Callback*.

```
[UnmanagedFunctionPointer(CallingConvention.Cdecl)]
delegate bool CallbackUnmanagedDelegate(IntPtr m);

bool CallbackUnmanaged(IntPtr m) {
  return Callback(new MObject(m));
}

bool Callback(MObject m) {
  Console.WriteLine("Called with " + m);
  return true;
}

void SetCallback() {
  var _delegate = new CallbackUnmanagedDelegate(Callback);
  IntPtr fnPtr = Marshal.GetFunctionPointerForDelegate(_delegate);
  nativeSetCallback(fnPtr);
}

[DllImport(...)]
static extern void nativeSetCallback(IntPtr fnPtr);
```

5.8 Overriding C++ Classes in Managed Code

To override virtual methods in a C++ class with managed methods, a wrapper class must be created over the C++ class that marshals the methods calls. In the code below, the managed *SpecialMotionState* class overrides methods in the native *MotionState* class. Whenever the abstract *MotionState::getState* method is called in unmanaged mode, the call is handled by the *MotionStateWrapper::getState* override, which marshals the call to the *SpecialMotionState.GetState* method.

```
// C++ class
class MotionState {
public:
  virtual int getState(int index) = 0;
};

// Signature of the virtual method
typedef int (*p_getState) (int index);

// C++ wrapper class
class MotionStateWrapper : MotionState {
  p_getState _getState;

public:
  MotionStateWrapper(p_getState getState)
    : _getState(getState) {}
```

```
17
18    virtual void getState(int index) {
19      _getState(index);
20    }
21  };
```

In the .NET wrapper class, an instance of the *GetState* delegate is created. The *GetFunctionPointerForDelegate* method creates a function pointer for the delegate that can be called from native mode. This pointer is passed to the constructor of the C++ wrapper class. The reference to the delegate must be stored in a field so that it wouldn't get garbage-collected. Note that it is not required to have a separate *GetStateUnmanaged* method, because there is no special marshaling required when going from native to managed mode.

```
1  // GetState function delegate
2  [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
3  delegate int GetStateDelegate(int index);
4
5  // Managed class overriding the native class
6  public class SpecialMotionState
7  {
8    GetStateDelegate _delegate;
9
10   public SpecialMotionState() {
11     _delegate = new GetStateDelegate(GetState);
12     IntPtr getStatePtr = Marshal.GetFunctionPointerForDelegate(
           _delegate);
13     _native = MotionStateWrapper_new(getStatePtr);
14   }
15
16   public int GetState(int index) {
17     Console.WriteLine("Called with " + index);
18
19     // Do work here
20     return 0;
21   }
22 }
23
24 [DllImport(...)]
25 static extern void nativeSetCallback(IntPtr obj, IntPtr getStatePtr);
```

GetState can also be a virtual or even an abstract method, in which case *SpecialMotionState* can be extended by other managed classes.

Another way to extend a C++ class is to directly modify its virtual method table (VMT). This is the idea behind the CXXI project [29]. However, since the VMT is compiler-specific, we would like to avoid this approach.

6 Class Structure Transformations

This section describes how the C++ code model created by the C++ parser is transformed into a .NET code model.

6.1 Common Language Specification Compliance

In .NET, assemblies, classes and methods can be marked as Common Language Specification compliant (CLS-compliant). CLS-compliant APIs are those that are specified using a subset of language features that are common to all .NET languages [30].

In public interfaces of .NET libraries, it is advisable to avoid non-compliant types and features such as *SByte*, *TypedReference*, unsigned integer types (*UInt16*, *UInt32*, *UInt64*, *UIntPtr*), boxed value types, *Nullable* types, unmanaged pointers and function pointers. If unsigned integers need to be wrapped, then they can be changed to one size larger (*UInt16* $\rightarrow$ *Int32*) to accommodate the most significant bit.

Method names differing only by case are not CLS-compliant:

```
1 void Init();
2 void init();
```

Structs can be passed by reference, but Visual Basic.NET and other CLS-compliant languages do not distinguish between parameters passed by value and by reference, i.e. the following method declarations are ambiguous:

```
1 void Init(Vector4 v);
2 void Init(ref Vector4 v);
```

When such a class is exported to VB.NET, trying to call any of the methods results in an ambiguous call compiler error, which cannot be resolved with any additional specifiers. C# does support such overloads.

6.2 Naming Conventions

Public interfaces of .NET libraries have a number of requirements when it comes to naming and organizing interface elements [31]. The automatic code generator should convert from C++ conventions to .NET conventions. Some relevant .NET conventions:

1. Avoid abbreviations that are not commonly used.
2. Avoid underscores and Hungarian notation (type information in names).
3. Classes, methods and properties should use PascalCasing.
4. Method parameters should use camelCasing.

5. Use namespaces instead of common prefixes.

6. Method names should be verbs, property names should be nouns or adjectives.

Table 6 shows differences between Google's C++ capitalization conventions [32] and those in .NET.

Table 6: Differences in capitalization between C++ and .NET.

Google C++ Style Guide	Corresponding C# naming convention
`// constant` `const int kDaysInAWeek = 7;`	`// constant` `const int DaysInAWeek = 7;`

`// variables`
`int price_count_reader;`
`int tablename;`
	`// variables`
	`int priceCountReader;`
	`int tableName;`

`// private variable`
`private:`
`  int num_entries_;`
	`// private variable`
	`private int _numEntries;`

`// namespace`
`namespace bullet`
`{`
`  ...`
	`// namespace`
	`namespace Bullet`
	`{`
	`  ...`

Some C++ libraries (such as the Bullet physics library) use prefixes instead of C++ namespaces to group classes. Since .NET does not allow prefixes by convention and requires classes to be placed into namespaces, the prefixes should be removed and converted into namespaces instead (see table 7).

Table 7: Converting C++ prefixes to .NET namespaces

Bullet physics library	.NET (C#) wrapper for the library

`class btCollisionObject`
`{`
`  ...`
`};`
 	
`class btRigidBody`
`{`
`  ...`
`};`
	`namespace Bullet`
	`{`
	`  class CollisionObject`
	`  {`
	`  ...`
	`  }`
	
	`  class RigidBody`
	`  {`
	`  ...`
	`  }`
	`}`

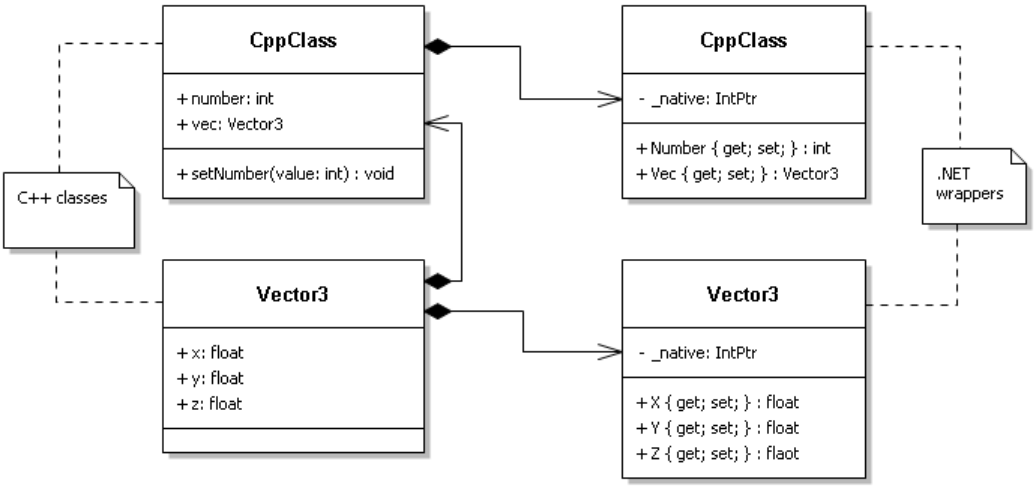
6.3 Wrapping Accessor Methods Using .NET Properties

Once acessor methods for fields have been created, they can be further transformed into properties. Properties in .NET are like *smart fields* [33]. A property has a getter method and optionally a setter method that are used to get and set the value of the property similarly to a field. The two methods can include additional logic that is applied whenever the property is accessed.

The *number* and *vec* accessor methods described in section 5.3 can be transformed into a property.

```
1 public int Number
2 {
3     get { return CppClass_getNumber(_native); }
4     set { CppClass_setNumber(_native, value); }
5 }
```

Figure 6: Wrapping C++ fields using .NET properties.



If the C++ code includes a field and also its setter with additional logic (e.g. *setNumber*), then a property can combine the .NET getter of the field with the C++ setter.

Some getter methods should not be turned into properties [34]. Getters that have side-effects can influence the behavior of the application when it is being debugged, because a debugger will attempt to access properties and show their values. Also, getters that take a long time to execute can make the debugger slower.

6.4 Inline documentation

Both C++ and C# support inline documentation for methods. Doxygen is a common tool and specification for generating documentation based on annotations in C++ comments. Doxygen also supports C#, but Microsoft's own feature for this is called *XML Documentation Comments*. The two approaches share similar annotations, but have different syntax. Automatic conversion is straightforward.

C++ Doxygen documentation	C# XML Documentation Comments

`/*!`
` * \brief Addition.`
` * \details Adds two numbers.`
` * \param a first term`
` * \param b second term`
` * \returns a+b`
` */`
`int add(int a, int b)`
`{`
`    return a + b;`
`};`
	`/// <summary>Addition.</summary>`
	`/// <remarks>Adds two numbers.</remarks>`
	`/// <param name="a">First term</param>`
	`/// <param name="b">Second term</param>`
	`/// <returns>a+b</returns>`
	`int Add(int a, int b)`
	`{`
	`    return a + b;`
	`};`

7 Performance benchmarks

Here, the overhead of calling methods with various signatures is measured in Platform Invoke and C++/CLI. Minimizing overhead is especially important in scenarios where unmanaged code is called very frequently. For example, games that query the physics engine for object locations several times per second.

The most accurate timer on the x86 platform is the timestamp counter, which is an integer register that is incremented at every CPU clock cycle. It provides resolution in the order of nanoseconds as opposed to microseconds for the managed Stopwatch timer and the unmanaged *QueryPerformanceCounter* function [35]. This counter can be accessed using the *RDTSCP* (Read Time-Stamp Counter) machine instruction [36]. *RDTSCP* is a serializing variant of the earlier *RDTSC* instruction, meaning that other instructions near *RDTSCP* are not executed out-of-order. Out-of-order execution would interfere with the measurement, because instructions that are being measured could be executed in parallel with instructions that do the measurement.

The counter may also be affected by CPU frequency scaling technologies such as Cool'n'Quiet on AMD platforms [37]. Therefore such technologies must be turned off before making measurements. To prevent context switching from affecting measurements, the code under test should be executed several times and the results statistically aggregated to exclude any outliers. Some sources suggest the process should be run at high priority and the processor affinity mask should be set such that only a single core can run the code [38]. This prevents the processor from having to reload the cache in case the process is moved to another core [39]. However, this proved to give inconsistent results as there may be other processes that are set to run on that particular core. Intel also recommends running tests in kernel mode, but there is no straight-forward way to do this in .NET.

Since *RDTSCP* is an unmanaged instruction with no direct equivalent provided by the .NET framework, the instruction must be called from native code. There are three ways to do it, by using PInvoke or C++/CLI, or generating platform-specific bytecode that returns the counter value. The bytecode solution could be used, because it is only a small amount of machine code that needs to be created, but this is not appropriate as a general purpose marshalling method.

Table 8: The most frequently occuring CPU cycle count among 1 million empty measurements.

Method	CPU cycles (AMD)	CPU cycles (Intel)
Baseline (PInvoke)	176	76
Baseline (C++/CLI)	141	50
Baseline (bytecode)	161	60

These three methods are compared by looking at the difference between two con-

secutive measurements, which is equivalent to measuring no code at all. This will be the baseline value that can be subtracted from the measurements that will be made afterwards. Table 8 shows the most frequently occuring CPU cycle count for 1 million measurements.

The code for doing these measurements is available on GitHub [40]. The measurements were made on 1) an AMD FX-6350 processor at 3.90 GHz with throttling disabled and 2) an Intel Core i7-6700 processor at 3.40 GHz. Intel CPUs are designed to have a higher IPC (instructions per cycle) than AMD, which explains the faster execution. Note however, that the results are relative to clock speed, so as the AMD processor runs at a faster clock rate, the actual time taken to execute the calls is somewhat evened out. The measurement method with the lowest overhead was C++/CLI, so it was used to do the next measurements.

Tables 9 and 10 show the performance of making various types of method calls. The delta value is the CPU cycle count with the baseline subtracted from it. The overhead of a call is roughly 185 clock cycles for C++/CLI and 210 clock cycles for PInvoke on an AMD processor, 75 clock cycles for C++/CLI and 90 clock cycles for PInvoke on an Intel processor.

Table 9: Various types of method calls and their cost in CPU cycles (AMD).

Method	C++/CLI CPU cycles	C++/CLI cycles delta	Platform Invoke CPU cycles	Platform Invoke cycles delta
Baseline (C++/CLI)	141	0	141	0
static empty method	182	41	211	70
static constant method	183	42	209	68
static identity method	183	42	210	69
class constructor	507	366	536	395
class construct and Dispose	744	609	804	663
empty method	182	41	212	71
constant method	188	47	212	71
identity method	186	45	212	71

Table 10: Various types of method calls and their cost in CPU cycles (Intel).

Method	C++/CLI CPU cycles	C++/CLI cycles delta	Platform Invoke CPU cycles	Platform Invoke cycles delta
Baseline (C++/CLI)	50	0	50	0
static empty method	74	24	90	40
static constant method	74	24	88	38
static identity method	76	26	90	40
class constructor	246	196	270	220
class construct and Dispose	354	304	398	348
empty method	76	24	90	40
constant method	76	26	90	40
identity method	76	26	90	40

8 Automatic Interface Generator Project

The project is a .NET application that parses C++ header files and outputs a PInvoke or C++/CLI wrapper for the C++ project [2].

8.1 Existing solutions

There are existing projects to generate interface code, but they all have some deficiencies.

8.1.1 Simplified Wrapper Interface Generator (SWIG)

SWIG is a program that can generate interface code not only for .NET, but also for other platforms like Python and Java [41]. SWIG requires all method signatures to be specified manually, which requires a lot of developer input. This is not convenient in the case of large projects that can have thousands of classes. SWIG uses the GPL license, which makes it difficult to write proprietary extensions to it, if this became necessary.

8.1.2 xInterop C++ .NET Bridge

xInterop C++ .NET Bridge is an advanced proprietary project for wrapping C++ code [42]. This project does not create a C wrapper over the C++ library, so the interface will be C++ runtime specific. The project also cannot be extended, since it is proprietary. While C++/CLI is supported, native C++ code can only dynamically linked, as opposed to the project described in this thesis, which does support static linking.

8.1.3 CXXI

CXXI is an open-source framework for creating .NET wrappers [29], which seems to be abandoned. It is part of the Mono project, which is a cross-platform .NET implementation. CXXI does not create a C wrapper over the C++ library, so it is not runtime-agnostic, similarly to xInterop.

8.2 Stages of the code generator

1. Parse C++ headers.
   (a) Use Clang to extract all of the code elements (classes, methods, etc.) into an in-memory representation.
   (b) Create an initial wrapper project configuration.
2. Apply default code transformations.
   (a) Resolve all references to types, including forward references.
   (b) Find all include files that are required to reference any used types.
   (c) Look for abstract methods and mark entire classes as abstract if needed.
   (d) Generate a concrete wrapper class for each class template specialization.
   (e) Remove redundant methods such as those that only differ by *const*-ness. Also remove constructors of abstract classes, since abstract classes are not allowed to be extended by default.
   (f) Generate default constructor if no explicit C++ constructor exists.
   (g) Generate accessor methods (get/set) for fields.
   (h) Detect property values that should be cached.
3. Apply target language specific transformations.
   (a) Detect enums that are flags (each value is a power of two).
   (b) Give each code element a name that conforms to .NET standards.
   (c) Convert get/set methods into .NET properties where appropriate.
   (d) Insert marshaling code for special data types.
4. Manual steps (changing the project configuration files)
   (a) Review ambiguous methods.
   (b) Exclude irrelevant classes and methods.
   (c) Insert custom transformations and code.
5. Output files.
   (a) Output C wrapper, PInvoke wrapper and C++/CLI wrapper.
   (b) Output Visual Studio solution files.
   (c) Output CMake build scripts.
   (d) Overwrite the wrapper project configuration with user customizations.

8.3 Wrapper Generator Unit Testing

To protect against regressions, the wrapper generator has unit tests to check that the interface code is generated correctly. The tests are written using the NUnit unit-testing framework. The test suites are organized as follows:

1. Basic project structure tests (namespaces, headers, classes).
2. Symbol mapping tests (header, class, method names).
3. Generating field accessor methods and .NET properties.
4. Generating class template specializations.
5. Default constructor and method override tests.

8.4 Future

To make the code generator more widely useable, it should be tested with libraries other than Bullet. Other libraries may contain language constructs not supported in the current version of the generator.

It is feasible to generate a minimal subset of the wrapper library by looking at the types used in the consuming application using .NET reflection.

9 Conclusion

This paper provided a set of guidelines for how to create a wrapper interface around C++ code. More common C++ constructs were discussed, but some were not, such as bit fields. The automatic interface generator can generate most of the code for the BulletSharp project, but as expected, some manual input is needed from the developer and other projects have not been tested. There are benefits over existing wrapper generators. We also determined the overhead of making calls from .NET into C++.

There is potential for improvements like adding support for more C++ constructs and testing with more C++ libraries.

References

[1] H. Stutter, “A Design Rationale for C++/CLI.” "http://www.gotw.ca/publications/C++CLIRationale.pdf”. February 2006 [Online].

[2] Andres Traks, “BulletSharpGen, a code generator for BulletSharpPInvoke.” "https://github.com/AndresTraks/BulletSharpPInvoke/tree/master/BulletSharpGen". GitHub [Online].

[3] Microsoft, “Recommendations on Nested Classes in Components.” "https://msdn.microsoft.com/en-us/library/s9f3ty7f(v=vs.71).aspx". Microsoft Developer Network [Online].

[4] Microsoft, “Using the DllImport Attribute.” "https://msdn.microsoft.com/en-us/library/aa984739%28v=vs.71%29.aspx". Microsoft Developer Network [Online].

[5] Microsoft, “Cleaning Up Unmanaged Resources.” "https://msdn.microsoft.com/en-us/library/498928w2%28v=vs.110%29.aspx". Microsoft Developer Network [Online].

[6] Microsoft, “Dispose Pattern.” "https://msdn.microsoft.com/en-us/library/b1yfkh5e%28v=vs.110%29.aspx". Microsoft Developer Network [Online].

[7] Microsoft, “Object.GetHashCode Method ().” "https://msdn.microsoft.com/en-us/library/system.object.gethashcode%28v=vs.110%29.aspx". Microsoft Developer Network [Online].

[8] Microsoft, “Differences Between C++ Templates and C# Generics (C# Programming Guide).” "https://msdn.microsoft.com/en-us/library/c6cyy67b.aspx". Microsoft Developer Network [Online].

[9] Microsoft, “Using C++ Interop (Implicit PInvoke).” "https://msdn.microsoft.com/en-us/library/2x8kf7zx.aspx". Microsoft Developer Network [Online].

[10] Ecma International Standard ECMA-372, “C++/CLI Language Specification, 19.13.1 Destructors.” "http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-372.pdf". December 2005].

[11] Ecma International Standard ECMA-372, “C++/CLI Language Specification, 19.13.2 Finalizers.” "http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-372.pdf". December 2005.

[12] Microsoft, “How to: Define and Consume Classes and Structs (C++/CLI) - Destructors and finalizers.” "https://msdn.microsoft.com/en-us/library/ke3a209d(v=vs.110).aspx#Anchor_9". Microsoft Developer Network [Online].

[13] CodeProject, Nish Nishant, “Deterministic Destruction in C++/CLI.” "http://www.codeproject.com/Articles/7965/Deterministic-Destruction-in-C-CLI". CodeProject [Online].

[14] The Clang Team, “Clang documentation, Introduction to the Clang AST.” "http://clang.llvm.org/docs/IntroductionToTheClangAST.html". Clang 3.9 documentation [Online].

[15] The Clang Team, “Comparing Clang to other open source compilers.” "http://clang.llvm.org/comparison.html". [Online].

[16] Microsoft, “Class Member overview.” "https://msdn.microsoft.com/en-us/library/10cwk72y.aspx". Microsoft Developer Network [Online].

[17] Bart Demeyere, “C++ in out parameters.” "http://users.telenet.be/bart.demeyere/C++InOutParameters.html". [Online].

[18] Doxygen, “Doxygen Manual: Special Commands.” "https://www.stack.nl/~dimitri/doxygen/manual/commands.html". Doxygen Manual, Dec 30 2015 [Online].

[19] Microsoft, “Built-In Types Table (C# Reference).” "https://msdn.microsoft.com/en-us/library/ya5y69ds.aspx". Microsoft Developer Network [Online].

[20] Microsoft, “CA1414: Mark boolean P/Invoke arguments with MarshalAs.” "https://msdn.microsoft.com/en-us/library/ms182206.aspx". Microsoft Developer Network [Online].

[21] cppreference.com, “Copy assignment operator.” "http://en.cppreference.com/w/cpp/language/copy_assignment". C++ reference [Online].

[22] Microsoft, “StructLayoutAttribute.Pack Field.” "https://msdn.microsoft.com/en-us/library/system.runtime.interopservices.structlayoutattribute.pack%28v=vs.110%29.aspx". Microsoft Developer Network [Online].

[23] Andrew Pardoe, “CLR Inside Out - Handling Corrupted State Exceptions.” "https://msdn.microsoft.com/en-us/magazine/dd419661.aspx#id0070035”. Microsoft Developer Network [Online].

[24] Microsoft, “HandleProcessCorruptedStateExceptionsAttribute Class.” "https://msdn.microsoft.com/en-us/library/system.runtime.exceptionservices.handleprocesscorruptedstateexceptionsattribute%28v=vs.110%29.aspx". Microsoft Developer Network [Online].

[25] Microsoft, “Microsoft developer network align (C++).” "https://msdn.microsoft.com/en-us/library/83ythb65.aspx". [Online].

[26] Microsoft, “Compiler Warning (level 3) C4316.” "https://msdn.microsoft.com/en-us/library/dn448573.aspx". Microsoft Developer Network [Online].

[27] Clark Nelson, “Dynamic memory allocation for over-aligned data.” "http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0035r1.html". C++ Standards Committee Papers, 20 Dec 2015[Online].

[28] Microsoft, “WindowsProtocolTestSuites on GitHub - WindowsProtocolTestSuites/IntPtrUtility.cs.” "https://github.com/Microsoft/WindowsProtocolTestSuites/blob/master/ProtoSDK/Common/IntPtrUtility.cs". Microsoft Developer Network [Online].

[29] Novell, Inc, “mono/cxxi: C++ interop framework.” "https://github.com/mono/cxxi". GitHub [Online].

[30] Microsoft, “Language Independence and Language-Independent Components.” "https://msdn.microsoft.com/en-us/library/12a7a7h3%28v=vs.110%29.aspx". Microsoft Developer Network [Online].

[31] Microsoft, “Naming Guidelines.” "https://msdn.microsoft.com/en-us/library/ms229002(v=vs.110).aspx ". Microsoft Developer Network [Online].

[32] Google, “Google C++ Style Guide.” "https://google.github.io/styleguide/cppguide.html". [Online].

[33] Microsoft, “Properties Overview.” "https://msdn.microsoft.com/en-us/library/65zdfbdt.aspx". Microsoft Developer Network [Online].

[34] Microsoft, “Choosing Between Properties and Methods.” "https://msdn.microsoft.com/en-us/library/ms229054(v=vs.100).aspx". Microsoft Developer Network [Online].

[35] Microsoft, “Acquiring high-resolution time stamps.” "https://msdn.microsoft.com/en-us/library/windows/desktop/dn553408%28v=vs.85%29.aspx". Microsoft Developer Network [Online].

[36] Intel, “How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures.” "http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf". September 2010.

[37] AMD, “AMD64 Architecture Programmer’s Manual Volume 3: General-Purpose and System Instructions,” Tech. Rep. 24594, AMD, May 2013. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2008/10/24594_APM_v3.pdf.

[38] Thomas Maierhofer, CodeProject, “Performance Tests: Precise Run Time Measurements with System.Diagnostics.Stopwatch.” "http://www.codeproject.com/Articles/61964/Performance-Tests-Precise-Run-Time-Measurements-wi". CodeProject 2010 [Online].

[39] Microsoft, “Process.ProcessorAffinity Property.” "https://msdn.microsoft.com/en-us/library/system.diagnostics.process.processoraffinity%28v=vs.110%29.aspx". Microsoft Developer Network [Online].

[40] Andres Traks, “BulletSharpPerfTest, Performance testing of BulletSharp.” "https://github.com/AndresTraks/BulletSharpPerfTest". GitHub [Online].

[41] The SWIG Developers, “Simplified Wrapper and Interface Generator.” "http://swig.org/". SWIG [Online].

[42] xInterop Software, LLC, “xInterop C++ .NET Bridge.” "http://www.xinterop.com/". [Online].

Non-exclusive licence to reproduce thesis and make thesis public

I, Andres Traks,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

.NET and C++ interoperation

supervised by Siim Karus

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 19.05.2016