Using C++ Class Libraries from an Interpreted Language

Wolfgang Heidrich, Philipp Slusallek, Hans-Peter Seidel
Computer Graphics Department, Universität Erlangen-Nürnberg
Am Weichselgarten 9, 91058 Erlangen, Germany.
EMail:{wgheidri,slusallek,seidel}@immd9.informatik.uni-erlangen.de

Abstract

The use of object-oriented programming, and C++ in particular, to build reusable class libraries has proven to be a very successful programming technique. However, the flexible composition of class libraries to create application programs has received little focus. In this paper we present a tool, that automatically maps a C++ class hierarchy to an equivalent hierarchy in an interpreted language. Using an interpreted language offers the programmer more flexibility when composing applications from existing class libraries.

1 Introduction

For quite some time object-oriented design and programming have received great attention for the development of large scale applications. One of the most important features is the ability to reuse software, once it has been implemented in well structured class libraries. C++ [Elis, 1990] has established itself as one of the major programming languages in this field.

Today, a wide range of class libraries is available for C++. Most of them come from a research background (for example InterViews [Linton, 1989], NIHCL [Gorlen, 1990], LEDA [Näher, 1990], Motif++, Fresco,...), but more and more commercial software is being developed in C++ and is available in form of class libraries (Rogue Wave [Keffer, 1992], Booch Components, Inventor [Strauss, 1992], C++/Views, USL, ImageVision [Neider, 1992], ...).

In this paper we present a tool called Itcl++, which allows the mapping of a C++ class hierarchy onto an equivalent class hierarchy in an object-oriented, interpreted language ([incr Tcl], pronounced and alternately written as Itcl). This mapping is possible without having access to the C++ source code of libraries, only the header files are required. The tool allows the creation of objects representing C++ classes, to invoke their member functions, and to derive other classes within the interpreted language. The syntax of [incr Tcl] permits a one-to-one mapping of C++ code to equivalent [incr Tcl] code.

This mapping gives the programmer the flexibility of an interpreted language when combining existing classes to form an application. It also facilitates a rapid-prototyping approach to application design, in that existing C++ classes can be extended and tested interactively in [incr Tcl], and then later possibly be recoded in C++.

1.1 Background

Building an application in an object-oriented language involves two steps. First, a set of classes must be designed, that implement the behavior of application specific objects. In this step predefined class libraries can be used to build application classes by specializing them with derived classes or instantiations of template classes. This usage results in a new set of specialized classes: a new specialized class library.

In the second step these classes must be assembled to form an application. This step normally includes the design of a user interface for the application. The use of an embedded interpreted language in the application makes this second step much easier. It allows the programmer to quickly change the structure of the class assembly, to rapidly exchange alternative class implementations with each other, or to flexibly react to changes in the user interface design. By embedding the object-oriented, interpreted language into the application, it can also be used as a tool for regression testing during the design process.

An embedded language may also have advantages for the final end product, as it can be used as a command language within the application, which is usually preferable to implementing an application-specific macro language from scratch.

There are several languages available, that can be embedded into an application (Perl, XLISP, Python [van Rossum, 1994], ...). Most of them, however, were not designed as an embedded language, and the majority lack a decent API from C or even better C++, or are not suitable for mapping C++ class hierarchies to it. We decided to use the Tool Command Language (Tcl) [Ousterhout, 1990] and its object-oriented extension [incr Tcl] [McLennan, 1993] as the base for our implementation, because it has found wide spread interest and was designed with a powerful C API. The [incr Tcl] extension is very similar to C++, which allows for a simple mapping of C++ to [incr Tcl]. We called our tool Itcl++, because it allows to use the flexibility of [incr Tcl] to access C++.

2 Tcl and [incr Tcl]

2.1 The Tool Command Language (Tcl)

Tcl has been especially designed for use as a command language for C applications. The Tcl implementation comes in form of a library, with the main data structure being the Tcl interpreter. C programs may allocate one or more interpreters, and use them to execute scripts contained in strings or files. Tcl scripts are lists of commands, each command taking a list of parameters.

Tcl supports all control structures known in most other structured programming languages. The Tcl syntax is a mixture of the syntax of C and that of the UNIX shell.

C functions taking an interpreter and an array of string arguments may be registered with an interpreter, and can then be accessed as normal Tcl commands. Parameters are passed to these functions in much the same way as they are passed to the C function `main()`.

The Tcl library also provides functions for manipulating Tcl variables, and implements hash tables which can be used to access C data from Tcl. We shall take a closer look on this in Section 3.1.

The main data type in Tcl is the string. Consequently, Tcl provides lots of functionality for string manipulations such as pattern matching and regular expressions. Specially formatted strings can act like higher level data types such as lists and associative arrays. One very powerful feature is the ability to trace actions (read, write and delete) on variables, both from Tcl and C. Tcl also implements an exception handling mechanism, which is a subset of the C++ mechanism.

A detailed description of Tcl and Tk (a Motif like widget library based on Tcl) and the interaction between Tcl/Tk and C/C++ can be found in [Ousterhout, 1994].

2.2 [incr Tcl]

[incr Tcl] extends Tcl in much the way C++ extends C. This is also indicated by the name, which means “increment Tcl” in Tcl notation. Many of the concepts of [incr Tcl], such as classes, (multiple) inheritance, public and protected member variables, constructor, destructor and static member functions have been adopted from C++.

However, [incr Tcl], like Tcl, is an untyped language, and thus operator overloading based on types can not be supported. As a consequence, there may exist at most one constructor per class.

A more restrictive property of [incr Tcl] is that a class may only occur once in the inheritance DAG of another class. This means that not every C++ class hierarchy can be mapped to [incr Tcl]. We will discuss this in detail in the next section.

An example [incr Tcl] class, implementing a simple counter class, is given below.

```
itcl_class counter {
  # Constructor:  no arguments, no code
  constructor {} {}

  # no arguments, return member variable
  method value {} {
    return $count
  }
  method += {val} {
    # increment count by val
    # and return the result
    return [incr count $val]
  }
  method -= {val} {
    # decrement count by val
    # and return the result
    return [incr count [expr -1*$val]]
  }
  protected count 0
}
```

Objects of this class can now be created using "`counter myCounter`", with `myCounter` being the name of the new object. Another way would be

```
set myCounter [counter #auto]
```

in which case an automatically generated object name would be stored in the variable `myCounter` ([...] is used for nested evaluation). For this paper we shall use the second notation. Once the object has been created, its methods may be called, for example

```
$myCounter += 10
```

By using the keyword `proc` instead of `method`, the C++ concept of static member functions is also available in [incr Tcl].

In [incr Tcl], classes and objects are implemented by special Tcl commands that instantiate the required scope, and then execute, for example, member functions in this scope. The names of the classes and objects act as names for these special functions. An introduction to [incr Tcl] can be found in [McLennan, 1993].

3 Mapping a C++ class hierarchy to [incr Tcl]

3.1 Basic Concepts

The major problem which arises when trying to attach existing C or C++ functions to Tcl, is that they do not normally receive their arguments by the `argc/argv` mechanism mentioned above. The developer has to write a C(++) wrapper function, which parses the parameter list, converts the string of each argument to the correct C type, and passes these arguments to the C++ member function. Return values and other output arguments must then be converted back into strings in order to be stored in a Tcl variable.

However, all wrapper functions are very similar to each other. Their main functionality, that is argument parsing and translation, could be created automatically if sufficient information about argument types is provided. The authors of *Wafe*, an Tcl interface to X Window widget sets [Neumann, 1993] used specification files with a special syntax for the description of C functions, widgets and special widget properties. These specification files were then parsed by a Perl script which created the appropriate C code.

Things get even more complicated if not only functions, but also C++ objects are to be accessed. Not only must a wrapper function be created for each public member function, but since C(++) data can not be addressed directly from Tcl, string handles need to be assigned, where each handle represents one C++ object on "the Tcl side" of the application. Tables that translate handles to and from C++ objects can be implemented using the hash tables provided by Tcl.

Moreover, our main goal was to transparently encapsulate C++ functionality in [incr Tcl] classes and objects. This means, that [incr Tcl] classes have to be built, with every member function calling the corresponding C++ function wrapper (see Figure 1 and Section 3.1.2). All necessary code should be created without human intervention, if at all possible.

3.1.1 Parsing C++

To meet these requirements, we decided to use a two step strategy. In the first step, C++ header files are parsed and specification files are created from C++ class definitions. The functionality of the specification files is a superset of those used in the *Wafe* project. In contrast to *Wafe*, which uses its own syntax, our specification files are just Tcl programs with additional functions defined. We shall discuss the file format in Section 3.2.

The parsing step is partially based on heuristics, since the semantics of a parameter can sometimes not be determined by just evaluating its declaration. Consider, for example, the following function.

```
void foo( Foo *f );
```

No decision is possible about whether `f` is supposed to be a pointer to an object of type `Foo`, or an array of `Foo` objects: the two alternatives obviously require different conversion code. In cases where ambiguities occur, heuristics must be used, and warning messages should be generated. Decisions made in this step may be overridden by simply editing the specification file.

3.1.2 Code Generation

This specification file, perhaps with some changes made by hand, now contains all the information needed to create C++ and [incr Tcl] code in the second step. Every semantic ambiguity in the specification file should now have been resolved, so that code generation can take place without further intervention.

The generated C++ code consists mainly of the C++ function wrappers, while the [incr Tcl] code creates the corresponding [incr Tcl] class hierarchy.

Instead of having different wrappers for each public function of a class, we decided to group these functions into four categories (constructor, destructor, static and non-static member), for each of which we create one wrapper in order to prevent replication of code. The function call scheme is illustrated in Figure 1.

In order to keep track of the C++ objects referenced by [incr Tcl], we use an object server, which consists of two hash tables. One hash table maps C++ pointers to [incr Tcl] object names and is used for handling return values. It is important, that this table contains pointers to all C++ subobjects of every registered C++ object. That is, for each registered C++ object the table contains one entry for each class in the inheritance hierarchy of the object.

The second table maps [incr Tcl] objects to C++ object pointers. Again we need different pointers for every class in the inheritance graph of the object, so we can not just take the [incr Tcl] object name as a hash key, since there is no one-to-one correspondence between names and pointers. Instead, we have to assign a unique handle to each [incr Tcl] subobject. This handle is stored in a protected `self` variable within every subobject.

During construction of a [incr Tcl] object, the constructor wrapper recursively calls the wrappers of the superclasses, with each wrapper registering the corresponding subobject and initializing the `self` variable (see Figure 2).

When calling C++ functions from [incr Tcl], the object server is used to convert the arguments. If a member function takes an object as one of its parameters, the wrapper function simply queries the object handler for the address of this object. Since every [incr Tcl] object is assigned a C++ object at creation time, this pointer always exists.

If, however, a C++ function returns a pointer to an object, this object may or may not be registered, that is, there may or may not be a corresponding [incr Tcl] object. If such an object exists, its name should be returned to [incr Tcl], otherwise a new [incr Tcl] object has to be created and to be registered with the returned C++ object in the object server. In this case, if the object handler is not able to find a matching entry in its hash table, it creates a new [incr Tcl] object, which in turn registers itself with the object handler (see Figure 3).

One problem we already mentioned above is that in [incr Tcl] a class may only occur once in the inheritance graph of another class. Thus the inheritance graph for every class may only be a

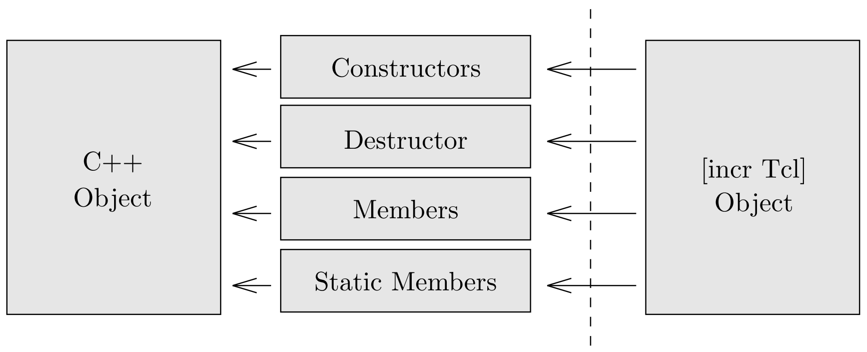

Figure 1: Access of C++ member functions through [incr Tcl]. [incr Tcl] members call C++ wrappers for constructors, destructors, methods and static methods, respectively. These functions convert function input parameters from Tcl to C++, execute the C++ object member and finally convert output parameters and the return value back to Tcl.

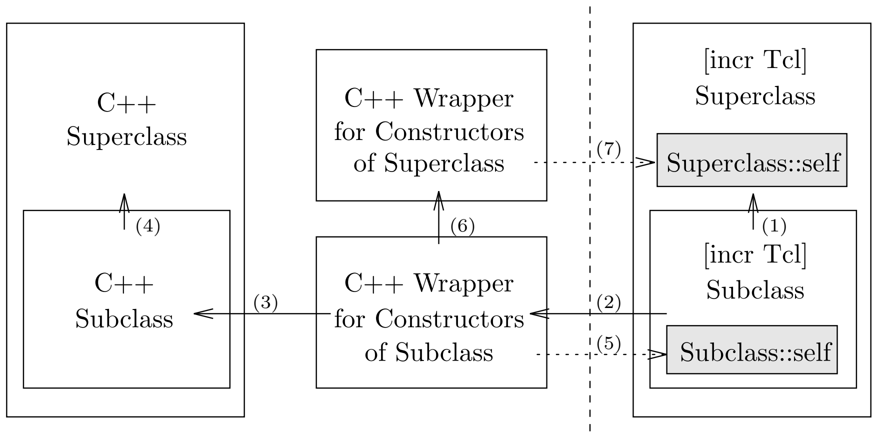

Figure 2: Whenever a new object is created, [incr Tcl] first executes the constructor of each [incr Tcl] base class (1). Afterwards, the C++ wrapper for the constructor is called (2), and a new C++ object is being created (3,4). Then the wrappers for the constructors of each baseclass are called recursively (6). They register the new object with the object handler, and initialize the `self` variables of the [incr Tcl] object (5,7).

tree instead of an arbitrary DAG as in C++, for example, when using virtual base classes. This means that C++ class hierarchies that use this feature cannot be completely mapped to [incr Tcl], but rather the [incr Tcl] hierarchy has to be cut below the point where this problem would occur.

While this is clearly a restriction, in practice the consequences seem not to be too striking, since in C++ this feature is most often used to provide groups of classes with low level functionality, for example being writable to some sort of stream. In cases where [incr Tcl] is used as an high level interface on top of a C++ hierarchy, which seems to be the most appropriate range of application, one could as well do without such low level functionality. Nonetheless, we think that [incr Tcl] should be changed to support the full C++ semantics of inheritance in the future.

3.2 Specification Files

Both the C++ parser, which generates the specification file, and the set of code generators for C++ and [incr Tcl] are themselves implemented in Tcl. Each code generator implements a set of proce-

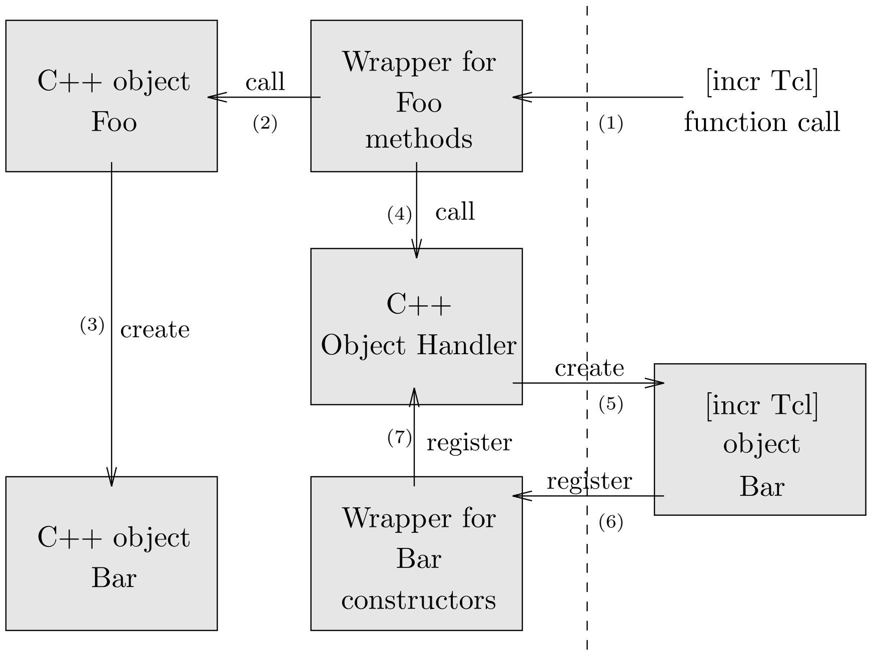

Figure 3: A member function of object `Foo` has been called (1,2), which returns a newly created object `Bar` (3). The `Foo` wrapper queries the object handler for the name of the corresponding [incr Tcl] object (4). Since the object handler is not able to find the name in the hash table, it creates a new [incr Tcl] object (5), which in turn registers itself through the constructor wrapper (6,7).

dures for generating C++ and [incr Tcl] code, respectively. These procedures are called from the specification scripts when they are executed by the code generators.

This approach ensures that our code generators have enough flexibility to handle even very complex situations, as arbitrary Tcl commands may be executed within the specification file.

A specification file for a counter class similar to the one shown in Section 2.2 is shown in the next column.

The empty list behind the class name indicates that class `counter` does not inherit from any other class. Static member functions may be specified by using the keyword `staticMember` instead of `member`. Since destructors do not take parameters, neither in C++ nor in [incr Tcl], only the existence of the destructor needs to be stated. No destructor must be created if the C++ class contains a private or protected destructor, because such destructors can not be accessed from within the C++ wrapper function.

```
class Counter {} {
  constructor Constructor {
    cmdCode{ returnVar= new Counter();}
  }

  destructor

  member int {} getValue {
    cmdCode \
         {returnVar= self->getValue();}
  }

  member void {} += {
    in int {cname value}
    cmdCode {self->operator+=( value );}
  }

  member void {} -= {
    in int {}
    cmdCode \
         {self->operator-=( localVar1 );}
  }
}
```

The lines starting with keyword `in` declare an input parameter for the member function, with the second entry being the C++ type, and the third entry being a list of option/value pairs. For example the option `cname` is used to assign a name to a function parameter in function `+=`. If no name is assigned to a variable, a default name starting with "`localVar`" is chosen as is the case in function `-=`. The same mechanisms apply for output values and their return types.

The lines starting with `cmdCode` contain the C++ code which is to be executed after parameters have been parsed, converted and stored in C++ variables. For more complex types clean up code may also be specified. The clean up code frees dynamically allocated memory and is executed as the last command in the wrapper function, after all output and return values have been written back to [incr Tcl].

In order to support types other than the standard ones built into C++, a type declaration file may be used to declare compound types like structures, enumerations and other complex types for which a conversion function from/to Tcl must be specified. C++ structures are mapped to associative arrays in Tcl, while enumerations are implemented by mapping a set of Tcl string constants to the corresponding C++ constants.

C++ arrays are mapped to Tcl lists. An example specification of an array of integers would be

```
in array {ctype int cname fooArg}
```

In addition to classes, specification files also support normal C/C++ functions. C++ template files are not directly supported by the specification files, because heuristic rules are necessary to create appropriate template instantiations. Thus the instantiation of C++ class templates has to take place in the parsing step, *before* specification files are created. These instances are then treated as ordinary class definitions, and are inserted in the specification files (see below).

4 Implementation

4.1 Creating Specification Files from C++ Headers

The C++ parser, which is itself implemented in Tcl, is provided with a list of header files and a list of classes, that are to be mapped to [incr Tcl]. The parser also has access to the type definition file mentioned above.

The public members of each class or class template contained in the list of classes to be converted, are extracted from the header files, and their parameter types are processed. The code for argument conversion between C++ and [incr Tcl] is fetched by walking through a table of rules and matching each type against a regular expression. This approach was chosen in order to be able to easily add and modify rules, and to be able to easily introduce new features. The table also contains information about whether a rule is hard-and-fast, or a heuristic.

Every time a heuristic rule is used, or a type can not be interpreted, a warning message is generated. In this cases the decision made in the specification file needs to be checked by the programmer. Functions having unknown classes as arguments (i.e. classes which are not in the list of classes to be converted) may automatically be ignored by placing the name of the unknown class in a special list.

A problem arises when handling template classes: It can not easily be determined which instances of a given template have to be generated. As an convention, the parser interprets typedefs of the form

```
typedef templateName<instance variables>
  instance;
```

as template instantiation. Instances, for which no `typedef` exists in the C++ header files, have to be inserted into the specification file by hand.

4.2 Generating C++ Code

Code generation is similar for the different types of function wrappers. Each wrapper function receives the name of the [incr Tcl] member function that has been called. The wrappers for the destructor and non-static members additionally get the [incr Tcl] `self` string of the calling object.

The wrapper does type checking and conversion of the function parameters from Tcl values to C++ values according to function type. Then the code contained in the `cmdCode` line of the member's specification is executed, and finally the output and return values are converted back to [incr Tcl].

Constructor wrappers have to register the new object and initialize the `self` variable of the [incr Tcl] object, while destructor wrappers must delete an old object from the object handler. Both have to call the corresponding wrappers of any superclass in order to recursively continue with the registration/deletion process. The code that would be generated for the `+=` member of the `Counter` class from Section 3.2 would look like this:

```
if( !strcmp( "+=", command ) ) {
  int value;

  if( argc!= 1 ) {
    argcError( "+=", "", 1, argc );
    DBUG_RETURN( TCL_ERROR );
  }

  if( (sscanf(argv[1],"%d",&value)!=1 ) {
    convError("+=", "1", argv[1], "int");
    DBUG_RETURN( TCL_ERROR );
  }

  self->operator+=( value );
}
```

The `command` variable is initialized to the function name, and the `self` variable contains a pointer to the calling object. This pointer has previously been returned by an object handler query.

The conversion function (`sscanf` in this example) is hardwired for every standard C++ type. For compound types it may be specified in the types definition file.

4.3 Generating [incr Tcl] Code

[incr Tcl] code generation is straightforward. Destructors, static and non-static members just pass their parameters along to the wrapper functions, with destructors and non-static members additionally providing the `self` variable.

The constructor (recall that [incr Tcl] supports only one constructor per class!) first checks, whether it is called directly or indirectly, which might happen as a result of inheritance. In the latter case the C++ wrapper must not be called, because the C++ object has already been constructed.

In the former case, the constructor uses the constructor name to pass the correct arguments and the [incr Tcl] object name to the C++ wrapper. An additional constructor `__registerObject`, which is used for registering already existing C++ objects with [incr Tcl] is added to every class.

The [incr Tcl] code for the constructor of our example `Counter` would be

```
constructor {args} {
  if { [$this info class]== "Counter" } {
    if { [llength $args] < 1 } {
      error "Wrong number of arguments"
    }

    case [lindex $args 0] in {
      {Constructor} {
         __CounterConstructor $this\
             Constructor
      }
      {__registerObject} {
         __CounterConstructor $this\
             __registerObject\
             [lindex $args 1]
      }
      default {
         error "Unknown constructor\
             [lindex $args 0]"
      }
    }
  }
}
```

To simplify creation of [incr Tcl] objects, an [incr Tcl] procedure is created for each C++ constructor. In our example, this would be

```
proc Constructor {} {
  return [Counter #auto Constructor]
}
```

so that an instance of class `Counter` could be generated with the command

```
set myCounter [Counter :: Constructor]
```

5 Results

5.1 Use of Itcl++ with our own Classes

Itcl++ has originally been developed for the use in a object oriented rendering system called Vision, which is currently under development at the graphics lab at the University of Erlangen.

The heuristics used for C++ parsing have been developed using the classes of the Vision hierarchy as a reference. However the C++ classes have not been changed to accommodate Itcl++.

The Vision system currently consists of about 140 classes, making extensive use of advanced C++ features such as templates. About 50 of the high level classes with a total of over 350 member functions have been mapped to [incr Tcl].

Heuristics have proven to work very well for this project: After inserting some type declarations in the types file, correct decisions have been made for *all* parameters of the 350 functions.

As a result, we are able to start Itcl++ from a "makefile", so that code is now generated completely automatically, without the need for human intervention.

We now use the [incr Tcl] interface to do initialization and configuration of our application, to describe scenes for our rendering system, and to test and debug new classes.

5.2 Use of Itcl++ with the OpenInventor Class Library

We tested Itcl++ with the commercial OpenInventor class library [Strauss, 1992] from Silicon Graphics. OpenInventor is an object oriented 3-D toolkit, which provides means to display and interactively manipulate complex scenes, using the Graphics Library OpenGL.

For our testing purposes we chose 32 classes with 190 member functions, mainly geometric objects and manipulators. The C++ parser detected 13 ambiguities, all relating to parameters of type `char *`. Based on the heuristic rules, these parameters were interpreted as strings, not as pointers to `char`. In all cases this interpretation turned out to be the right one, so that no further human intervention has been necessary.

The specification file of 839 lines has been used to create 8204 lines (about 18 KB) of C++ code. This makes an average of about 43 lines of code per member function.

A sample Inventor program using the mapped classes is given in Figure 4, with the left column containing the [incr Tcl] code and the right column containing the same program as it would be implemented in C++.

After initialization of the library, a root object is created and initialized with a camera, light etc. Then a simple scene with a cone and a "trackball manipulator" is created and inserted in the root object.

The trackball manipulator allows the user to rotate the scene on the screen. Finally a window is created and the main event loop is entered. An image taken from the sample program is shown in Figure 5.

6 Future Extensions

6.1 Accessing C++ Variables

The concepts described above allow transparent access of C++ functions and class member functions from [incr Tcl]. So far, however, it is not possible to directly manipulate C++ public member variables or global variables. The problem is that there would have to be two instances of every variable, one instance in C++ and one in [incr Tcl]. While C++ code manipulates the C++ instance, [incr Tcl] code changes the corresponding [incr Tcl] instance, so we face the problem of keeping the two instances consistent.

In order to ensure consistence, one could use the Tcl trace facility mentioned in Section 2.1 to

```
set window\
  [SoXt :: init "Gray Cone"]

set root [SoSeparator :: Constructor]
$root ref

set camera\
  [SoPerspectiveCamera :: Constructor]
$root addChild $camera

$root addChild\
  [SoDirectionalLight :: Constructor]
$root addChild\
  [SoTrackballManip :: Constructor]
$root addChild\
  [SoMaterial :: Constructor]
$root addChild\
  [SoCone :: Constructor]

set ra\
  [SoXtRenderArea :: Constructor $window]
$ra setTitle "Gray Cone"

$camera viewAll $root\
  [$ra getViewportRegion] 1

$ra setSceneGraph $root
$ra show

SoXt :: show $window
SoXt :: mainLoop
```

```
main()
{
  Widget window=
    SoXt::init( "Gray Cone" );

  SoSeparator *root= new SoSeparator;
  root->ref();

  SoPerspectiveCamera *camera=
    new SoPerspectiveCamera;
  root->addChild( camera );

  root->addChild(
    new SoDirectionalLight );
  root->addChild(
    new SoTrackballManip );
  root->addChild(
    new SoMaterial );
  root->addChild(
    new SoCone );

  SoXtRenderArea *ra=
    new SoXtRenderArea( window );
  ra->setTitle( "Gray Cone" );

  camera->viewAll( root,
    ra->getViewportRegion(), 1 );

  ra->setSceneGraph( root );
  ra->show();

  SoXt::show( window );
  SoXt::mainLoop();
}
```

Figure 4: Simple Inventor program, written in [incr Tcl](left) and C++ (right).

implement a read through/write through mechanism. Tcl variable traces allow the attachment of a Tcl function to an arbitrary variable. This function is then called on every read or write access on the variable.

On read access, the trace function is called *before* the contents of the variable are read. This means that the trace function could call a C++ wrapper which updates the [incr Tcl] variable with the corresponding C++ value.

On a write access, the trace function is called *after* the [incr Tcl] variable has been set to the new value. Thus the trace function could update the C++ variable to this value.

This approach would require one wrapper for the public member variables of each class, and one additional wrapper for the global variables.

6.2 Accessing [incr Tcl] Objects From C++

We described a way of mapping classes implemented in C++ to [incr Tcl]. At times it may also prove useful to have direct access to [incr Tcl] classes from within C++ code. This could be used for rapid prototyping of new applications by using the high level features of [incr Tcl], for example regular expressions and string manipulation. Once the algorithms are implemented and tested, each class may be moved to C++.

In the current implementation, it is possible to access [incr Tcl] objects from C++, but this access is not transparent. That is, instead of just

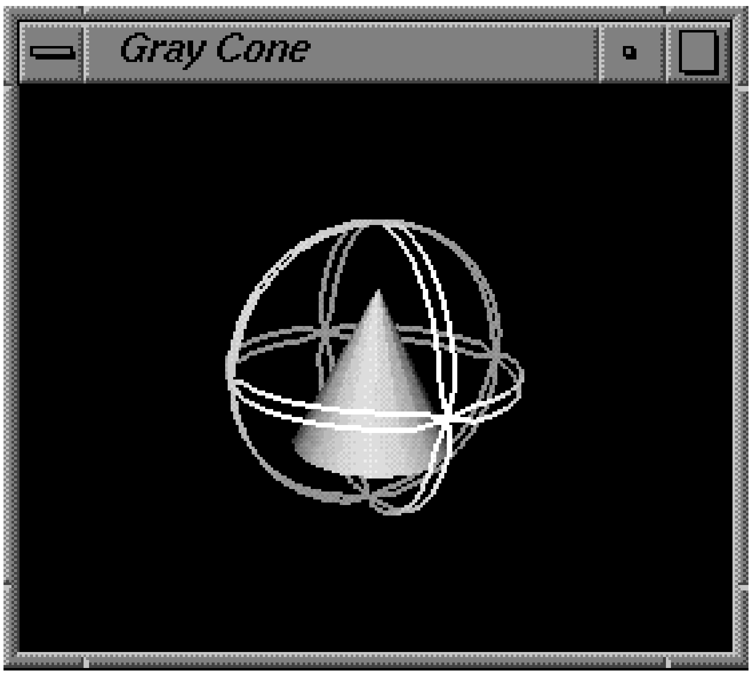

Figure 5: Window of the sample Inventor program given in Section 5.2. The cone may be rotated using the trackball manipulator.

calling a C++ member function, one has to create and evaluate a string containing [incr Tcl] code.

In order to encapsulate these calls within C++ objects, concepts similar to those described in this paper might be used to map [incr Tcl] classes to C++. The existing specification file format could also be used for this conversion.

A problem arises when trying to automatically generate the specification files from [incr Tcl] code. Since [incr Tcl] is an untyped language, the generator of a specification file must be provided with additional type information for each function that is to be mapped to C++. This could be achieved by introducing a convention for [incr Tcl] comments.

7 Conclusion

We have shown that it is possible to map C++ class hierarchies into equivalent hierarchies in an interpreted command language ([incr Tcl]). This mapping allows flexible access to C++ class libraries from [incr Tcl], with all the advantages of an interpreted language. Although [incr Tcl] lacks many features of C++ (overloading, virtual base classes, templates), most C++ class libraries can be mapped to [incr Tcl] with no loss in functionality. The approach has been demonstrated using examples from a rendering class library and a commercial graphics library.

8 Acknowledgments

We would like to thank Gustav Neumann and Stefan Nusser, who implemented a specification syntax and a related code generator for the *Wafe* program [Neumann, 1993]. We used their Perl implementation as a starting point for Itcl++. Gustav Neumann also made some suggestions concerning the syntax of our specification files. We would also like to thank the members of the Vision project, for which the tool has originally been written since they tested early versions of Itcl++ and provided valuable feedback.

References

[Elis, 1990] Elis, M. A. and Stroustrup, B. (1990). *The Annotated C++ Reference Manual*. Addison Wesley.

[Gorlen, 1990] Gorlen, K. E. (1990). *Data Abstaraction and Object-Oriented Programming in C++*. Teubner.

[Keffer, 1992] Keffer, T. (1992). *Tools.h++*. Rogue Wave Software.

[Linton, 1989] Linton, M. A., Vlissides, J. M., and R.Calder, P. (1989). Composing user interfaces with interviews. *IEEE Computer*, pages 8–22.

[McLennan, 1993] McLennan, M. J. (1993). [incr Tcl]: Object – Oriented Programming in Tcl. In *Proc: Tcl/Tk Workshop, University of California at Berkeley, 1993*.

[Näher, 1990] Näher, S. (1990). *LEDA 2.0 User Manual*. Universität des Saarlandes, Saarbrücken.

[Neider, 1992] Neider, J. and Tillmann, C. (1992). *ImageVision Library C Programming Guide*. Silicon Graphics Computers.

[Neumann, 1993] Neumann, G. and Nusser, S. (1993). Wafe — an X Toolkit Based Frontend for Application Programs in Various Programming Languages. In *Proc: Usenix Winter Conference, 1993*.

[Ousterhout, 1990] Ousterhout, J. K. (1990). Tcl: an Embedded Command Language. In *Proc: Usenix Winter Conference, 1990*.

[Ousterhout, 1994] Ousterhout, J. K. (1994). An Introduction to Tcl and Tk. To be published. Addison Wesley.

[Strauss, 1992] Strauss, P. S. and Carey, R. (1992). An Object–Oriented 3D Graphics Toolkit. In *ACM Computer Graphics*. SIGGRAPH '92 Conference Proceedings.

[van Rossum, 1994] van Rossum, G. (1994). *Python Reference Manual*.