The following paper was originally published in the
Proceedings of the Fourth USENIX Tcl/Tk Workshop
Monterey, CA, July 10-13, 1996

Automated Wrapping of a C++ Class Library into Tcl

Ken Martin
General Electric Corporate Research and Development
Niskayuna, NY 12309

Automated Wrapping of a C++ Class Library into Tcl

Ken Martin
General Electric Corporate Research and Development
1 Research Circle, Niskayuna, NY 12309
martink@crd.ge.com

Abstract

In this paper we describe an approach to wrapping a preexisting C++ class library into the interpreted Tcl environment. Specifically, we look at our efforts over the past two years to add the Tcl interpreted environment on top of the Visualization Toolkit which consists of over three hundred C++ classes. We address how we overcame the fundamental issues involved in wrapping existing C++ code and what limitations we had to accept. We contrast our approach to other Tcl object oriented extensions such as Object Tcl and [incr Tcl] and explain why they were not suitable. We conclude by looking at future directions for our work.

1. Introduction

Over ten years ago researchers at General Electric started work on an interpreted, object-oriented visualization system [2]. At the time, C++ was still in its infancy and Smalltalk was considered one of the most promising languages. Taking some of the best features of Smalltalk, a new language was developed called LYMB. This provided an interpreter and script based language very similar to Tcl with the exception that it was built on object-oriented principles. While the system was very valuable, it had some weaknesses that made it monolithic and difficult to integrate into other systems. In late 1993 an effort was started to create a new system that would overcome many of the problems associated with its predecessor.

One thing we had learned was that object-oriented technology was a big productivity enhancer for our group. We decided that any replacement system should be object oriented. We also wanted the system to be portable, extensible and easily integrated into a wide variety of applications. To this end we selected C++ as our development language. About eighteen months into the development of The Visualization Toolkit [5] (VTK) as it later was named, we realized that using only C++ was proving to be a problem. In LYMB we had an interpreted environment that made interactive development very quick and easy. With VTK we had no interpreter or scripting language and its loss was profoundly felt. At this point we started looking into adding an interpreter into, or on top of our C++ class library.

Unlike LYMB which used its own interpreted language, we wanted to select a language that was already established. We would focus on data visualization which was our strength, and let others (such as the Tcl community) focus on language issues. The decision came down to Tcl or Python. Tcl's wide acceptance, maturity and easy extensibility won out in spite of the fact that it didn't have any direct object-oriented support.

After some quick research into the Tcl community, we found that there wasn't a fully automatic way to wrap C++ into Tcl without significantly modifying the C++ code. Since we already had over 200 classes written, a serious change in coding style was out of the question. At this point we decided to try writing our own automated C++ code wrapper for Tcl, the result of which is discussed in the remainder of this paper.

2. Related Work

When we started writing our C++ to Tcl wrapper generator there were no suitable alternatives. Now, over two years later, much progress has been made on this front and it is worth revisiting. [incr Tcl][3][4] has emerged as a popular object oriented extension to Tcl providing objects, mega-widgets namespaces and more. While this system does provide strong encapsulation and integration with Tcl, it does not provide a strong interface to C++ classes. [incr Tcl] allows you to bind C functions or static C++ functions into [incr Tcl] classes. It does not support wrapping of C++ classes, construction, destruction of C++ objects, or invocation of C++ member functions. Object Tcl[7] is similar to [incr Tcl] in that it is designed to support object oriented programming for Tcl, not the incorporation of existing C++ libraries.

The Simple Wrapper Interface Generator (SWIG) by Dave Beazley, is a tool for that can generate wrapper code for a number of different target languages including Tcl. SWIG requires an interface file that prototypes the functions that are to be wrapped. For many C libraries the ANSI C prototypes can be directly included into the interface file. For C++ code it isn't as easy. SWIG provides a good wrapper for prototype functions, but it doesn't provide much support for wrapping C++ classes. Creating, destroying, type-conversion and passing of classes as arguments is not supported in the current version. So while SWIG is an excellent package for wrapping C libraries, it currently isn't suitable for wrapping C++ class libraries.

Objective-Tcl[1] does provide much of the functionality we required for wrapping VTK. Since we had decided to use C++ as our compiled language and Objective-Tcl binds to Objective-C, we were not able to use it. In some ways Objective-Tcl is a better solution since it supports creating classes and methods within the Tcl domain. Our solution provides access to the C++ objects, but it does not provide any method for object oriented programming in the Tcl domain. The use of Objective-C's run time class information solves some of the tricky problems of C++ object integration that we had to address.

Otcl[6] is the system that most resembles our solution. In many ways Otcl, like Objective-Tcl, provides a better solution to wrapping existing classes into Tcl. But there are some key differences between Otcl and our approach. Otcl requires the user to specify a Class Description Language (CDL) file for all the objects you want integrated into Tcl. In this file, you must specify the public methods, their arguments and whether they require static or dynamic binding. In our approach the required information is extracted from the C++ header files.

Otcl currently doesn't handle type conversions between superclasses and subclasses which can lead to problems, especially in libraries that make use of multiple inheritance. Early in our development we ran into this problem which prompted our efforts to perform true type conversion for C++ objects. To support VTK's hardware independence, instances are created on the C++ side which must be returned and used on the Tcl side. Otcl supports this through the use of special classes that the C++ side can instantiate. But this means that the original C++ code must be modified to create instances of these special classes instead of its normal behavior. This limitation isn't present in our approach. Otcl also lacks support for overloaded methods which we support.

While Otcl may not provide as tight a wrapping of C++ classes, it does provide much more flexibility in the Tcl domain. C++ classes can be subclassed in the Tcl domain, passed back and forth, and methods added or overridden. Our work does not support any object oriented extensions or subclassing from the Tcl domain.

3. Methods

There are a number of difficult issues to be resolved in automatically wrapping C++ into Tcl. We will examine them in the order that we dealt with them. The first problem that we ran into was parsing our C++ code. In order to wrap it into Tcl, we needed a method of parsing the code and breaking it into its lexical components. To do this we decided to use the traditional tools LEX and YACC.

As it turns out, writing LEX and YACC code for C++ is not an easy task since C++ has so many features. There were two ideas that enabled us to get by. The first was our philosophy regarding C++. We believed in using only the core features of C++ within a relatively rigid development environment. We did not use templates, run time type checking or exceptions. Our coding standards dictated that all class names start with a common prefix (vtk) a limit of one class per file, and a set of simple macros for performing common set/get methods. With these restrictions we were able to develop a LEX and YACC based program that could parse our header files and extract the pertinent information. We did not make an attempt to handle preprocessor directives, which means that our wrapping of a class is done based on that class's header file alone, without looking at its superclasses' header files.

The first step in wrapping a class library is to determine which classes are concrete and which are abstract. In object-oriented terminology, abstract classes define an API for their subclasses. They are not meant to be instantiated and under some circumstances it can be a compiler error. To resolve this we modified our LEX and YACC code to parse the header files and look for any pure virtual functions, which indicate that a class is abstract. We then create a list of abstract classes and concrete classes. This approach is not fool proof. There may be a class whose superclass is abstract, that hasn't made itself concrete by providing a concrete implementation of the pure virtual function of the superclass. Since we judge each class based solely on its header file, we cannot detect this condition. In this case, which has happened only twice in our now 360 classes, we must manually specify that the class is abstract.

```
int vtkNewInstanceCommand(ClientData cd, Tcl_Interp *interp,
                          int argc, char *argv[])
{
  Tcl_HashEntry *entry;
  int is_new;
  char temps[80];

  if (argc != 2)
    {
    interp->result = "vtk object creation requires one argument, a name.";
    return TCL_ERROR;
    }

  if ((argv[1][0] >= '0')&&(argv[1][0] <= '9'))
    {
    interp->result = "vtk instances names must start with a letter.";
    return TCL_ERROR;
    }

  if (Tcl_FindHashEntry(&vtkInstanceLookup,argv[1]))
    {
    interp->result = "a vtk instance with that name already exists.";
    return TCL_ERROR;
    }

  /* Create an instance of vtkActor */
  if (!strcmp("vtkActor",argv[0]))
    {
    ClientData temp = vtkActorNewCommand();
    entry = Tcl_CreateHashEntry(&vtkInstanceLookup,argv[1],&is_new);
    Tcl_SetHashValue(entry,temp);
    sprintf(temps,"%x",(void *)temp);
    entry = Tcl_CreateHashEntry(&vtkPointerLookup,temps,&is_new);
    Tcl_SetHashValue(entry,(ClientData)(strdup(argv[1])));
    Tcl_CreateCommand(interp, argv[1], vtkActorCommand, temp,
                      (Tcl_CmdDeleteProc *)vtkTclGenericDeleteObject);
    entry = Tcl_CreateHashEntry(&vtkCommandLookup,argv[1],&is_new);
    Tcl_SetHashValue(entry,(ClientData)(vtkActorCommand));
    }

  /* Create an instance of vtkAppendPolyData */
  if (!strcmp("vtkAppendPolyData",argv[0]))
    {
     ...
    }
  ...
}
```

Listing 1. C++ instance creation code for Tcl.

For all of the concrete classes we run a second LEX/YACC program to generate the required Tcl code to create a new instance. We use the same model as the Tk widgets in that the class name serves as the command to create an instance of that class. In our VTK_Init function, we simply create a command using Tcl_CreateCommand for every concrete class in the library. All of these commands invoke the same function vtkNewInstanceCommand, which checks to make sure that the instance name was specified and is unique. It then creates a new command with the same name as the instance, and attaches it to a function that is responsible for handling methods on that class. It also creates a C++ instance of the desired class and associates the Tcl instance name with the Tcl method function for that class and the C++ object pointer. We use three Tcl hash tables to store these associations.

The code sample in Listing 1, shows a small portion of the vtkNewInstanceCommand function. In this example it shows the logic for creating an instance of the class vtkActor. First it does some quick checks to make sure that the desired instance name is specified, starts with a letter, and isn't already in use. Then it does a string compare for argv[0] against “vtkActor”. If it matches, it calls a function called vtkActorNewCommand which will create an instance of vtkActor and then return it as type ClientData. The next two lines create an association between the Tcl name for that instance and the actual C++ pointer address. This is stored in the vtkInstanceLookup hash table. The next three lines perform a similar task, associating the pointer address with the Tcl instance name. The next line creates a Tcl command with the same name as this instance. It also associates the vtkActorCommand function with this instance to handle any method invocations. The final two lines create an association between the Tcl instance name and the function that should be used for invoking methods.

The next step in the process creates the method functions for all of the classes. Where we only allow the user to create instances of concrete classes, we must provide support for invoking methods from both concrete classes or their abstract superclasses. This is because some of the methods of a concrete class may be implemented in an abstract superclass. So we must wrap methods of the abstract classes even though we do not allow the user to create a direct instance of one. The method function serves to connect the Tcl commands with their string arguments, to the C++ method invocations. There are four main operations that occur in these functions: typecasting which we will discuss later, chaining up the class hierarchy to search for unresolved methods, additional methods that are specific to the Tcl interpreted environment, and the bulk of the code serves to handle the invocation of the C++ methods.

Chaining up the hierarchy is handled in a simple manner. If a method wasn't found in the current function, we then invoke the same Tcl command, but this time using the superclass's method function. This continues until either the method is found or the top of the hierarchy is reached. For classes with multiple inheritance we perform a depth first search of the inheritance tree. If we cannot find a method that matches then we return TCL_ERROR. Since Tcl is an interactive language, we use the method function to add two additional commands to each class. We provide a hook for the Tcl user to invoke the PrintSelf method on a class and have it be returned in interp->result. We also added a command to list all the possible methods you can send to a class and its superclasses.

As we mentioned above, the bulk of the method function handles wrapping the C++ methods. From the C++ header file we obtain the names of the methods and the number and type of arguments that they take and return. For each method we first compare the Tcl method name, stored as argv[1], with the target method. We then check to see if the number of arguments matches. After this we start taking apart the Tcl string arguments and converting them into the appropriate C++ arguments. For the traditional C data types we use the standard Tcl functions such as Tcl_GetDouble. For passing C++ objects we use our own function. If any of the argument conversions fail, we assume that this wasn't the correct method and we continue searching for a match. This is how we handle overloaded functions. Two functions may have the same name and the same argument count, but the string arguments passed in from Tcl may convert correctly for one method and not the other, effectively disambiguating them. The return value of a method is converted into a string and then returned in interp->result. The example in Listing 2, from vtkActor's method function will help clarify this process.

The first line checks to make sure the method name matches and that the argument count is correct. Remember that the first Tcl argument is the instance name, the second is the method name and the third Tcl argument is the first C++ argument. So if we need three C++ arguments then we check to see if argc is five. The next series of statements try to convert the Tcl arguments into the desired type for C++, in this case floats. If all the arguments convert correctly, we invoke the function and return TCL_OK. In this example the C++ method does not have a return value. The C++ instance, stored in the variable op, is passed in as the ClientData from Tcl.

```
int vtkActorCppCommand(vtkActor *op, Tcl_Interp *interp,
                       int argc, char *argv[])
{
  int    tempi;
  double tempd;
  static char temps[80];
  int    error;

  // some type conversion routines etc
  ...

  // check for an invocation of the SetPosition method
  if ((!strcmp("SetPosition",argv[1]))&&(argc == 5))
     {
     float    temp0;
     float    temp1;
     float    temp2;
     error = 0;

     if (Tcl_GetDouble(interp,argv[2],&tempd) == TCL_OK)
       {  temp0 = tempd; }
     else
       {  error = 1;  }
     if (Tcl_GetDouble(interp,argv[3],&tempd) == TCL_OK)
       {  temp1 = tempd;  }
     else
       {  error = 1;  }
     if (Tcl_GetDouble(interp,argv[4],&tempd) == TCL_OK)
       {  temp2 = tempd;  }
     else
       {  error = 1;  }
     if (!error)
       {
       op->SetPosition(temp0,temp1,temp2);
       interp->result[0] = '\0';
       return TCL_OK;
       }
     }

  // check other methods
  ...

  // If we haven't found a match, try chain up the superclasses
  if (vtkObjectCppCommand((vtkObject *)op,interp,argc,argv) == TCL_OK)
    {
    return TCL_OK;
    }

  ...
}
```

Listing 2. Excerpt from vtkActor's method function.

The first concern with this approach is that C++ header files do not provide enough information to correctly convert to and from Tcl strings. The classic example of this is a method that returns a pointer to a float. The C++ header file provides no information about how many elements are in that array. In some cases the correct size may not be known until runtime, but for many cases the return size is fixed and should be used in the wrapping process.

To support this we create a hint file that contains a series of lines of the form: class name, method name, return type and expected size. When the wrapping code encounters a method that it normally couldn't wrap, it checks the hint file to see if the required information is in there. As it turns out, our hint file is only one or two pages long. This is because many of our methods that return fixed size arrays are done with macros which include the size of the result in the macro call. These macros are typically Set/Get methods for instance variables such as the (X, Y, Z) position of an actor. These same macros allow us to pass arrays into C++ methods since the required size of the array is known at the time of wrapping.

Passing C++ objects to and from methods is where a great portion of the complexity comes from. A string passed to a method that is expecting an object can be looked up in the hash tables defined earlier. If it is found, we then have to perform a proper C++ typecasting to the desired argument type. For example, say that you have a class B that is a subclass of class A. You have created an instance of B named foo from the Tcl interpreter, and you want to pass it into a method that takes an argument of type A. To do this we first look up the instance foo in our hash tables to get the C++ object pointer. Then we invoke the method function for that class (B in this example) asking it to perform a typecasting to a result of type A. We obtain the method function by looking it up in a hash table. If the typecasting is possible, e.g. if B is really a subclass of A, then the correct pointer is returned and the method will be invoked. Otherwise an error occurs and method resolution continues as with any other argument mismatch.

An example illustrates why this typecasting is so important. Given the following piece of C++ code where B is a subclass of A, you might expect that bar and foo would point to the same location in memory, but in many cases they will not. Doing a blind typecasting of a C++ object obtained from the hashtable into the desired argument type will create all sorts of problems, especially since no checking will be done to ensure that the two C++ types are even compatible.

```
class A ...;
class B: A ...;
B *foo;
A *bar = foo;
```

It also happens that some C++ methods create instances of C++ classes and return them. In this case, there will not be a Tcl string name for the instance since it was created on the C++ side. When a C++ method returns a C++ class pointer, we look up that pointer in the hash tables. If it has an associated name, then we return that string. If it doesn't then we generate a name of the form vtkTemp1 or vtkTemp2 etc. We then enter that name into the hash tables as if the user had created the instance. This is commonly used in situations like the following Tcl script:

set activeCamera [aRenderer GetActiveCamera];

We set the value of the variable activeCamera to the generated name returned from the GetActiveCamera method. When it comes to freeing instances, we use the rule that if the instance was created from the Tcl side, then we will free the C++ side when the Tcl side is freed. If the instance was created on the C++ side, then we require that whatever class created the instance, also free it.

Now that we have discussed how we pass primitive data types, arrays and C++ objects, there is one more type of argument that is worth mentioning: the user defined function. User defined functions are essentially callbacks that certain classes support. The user specifies a function to be called and an argument to pass to that function whenever a certain event occurs. With a Tcl wrapping we don't want to pass a function pointer, but rather a string of Tcl commands to be executed. So we package up the string and the interpreter that sent the command into a structure, and set that as the argument to be sent to the function. We then specify that the user defined function call the generic function below. This function takes apart the structure and invokes Tcl_GlobalEval to execute the string of Tcl script.

```
void vtkTclVoidFunc(void *arg)
  {
  vtkTclVoidFuncArg *arg2;
  arg2 = (vtkTclVoidFuncArg *)arg;
  Tcl_GlobalEval(arg2->interp,
                 arg2->command);
  }
```

From Tcl, a user defined function can be setup in the following straightforward manner.

```
# tcl code to draw a cube
# create a few instances of vtk classes
vtkRenderMaster renMaster;
vtkCubeSource cubeSrc;
vtkPolyMapper cubeMap;
vtkActor cube1;

# create the rendering window and renderer
set renWin [renMaster MakeRenderWindow];
set ren1   [renWin MakeRenderer];

# connect the pieces and draw the result
cubeMap SetInput [cubeSrc GetOutput];
 cube1 SetMapper cubeMap;
$ren1 AddActor cube1;
$renWin Render;
```

Listing 3. A simple Tcl script using the wrapped Visualization Toolkit.

```
vtkContourFilter c;
c SetEndExecuteFunction {puts "Done"}
```

4. Results

Using the combination of techniques described above we have developed an automated technique for wrapping a C++ library into Tcl. We have used this for over a year in developing The Visualization Toolkit which now contains over three hundred classes and thousands of methods. The process has required almost no human interaction. The most common issue is adding a few methods to the hint file now and then. We managed to do this with making only a couple changes to our code. There were a few classes that had helper classes defined in the same header file. To handle this we added a special comment as below:

```
// BTX - short for begin Tcl exclude
   code that breaks our parser in here
// ETX - short for end Tcl exclude
```

Taking advantage of the dynamic loading features of Tcl7.5 we have wrapped additional C++ libraries into Tcl so that they can be loaded at run time by the scripts that require them. At this point our automated technique wraps about 98% of the possible methods within VTK and generates about 100,000 lines of source code. We have had success applying the same approach to wrap VTK into Java, although some of the issues are different. This ability to generate wrapper code for Tcl or Java is a nice feature.

Unfortunately this technique does not leverage any of the recent object oriented extensions to Tcl (such as Otcl) and it is not a general purpose technique that can wrap any C++ code. As nice as that would be, it was through our moderated use of C++ that we were we able to successfully wrap the Visualization Toolkit and a few additional C++ libraries into Tcl. In the future it might be worth considering using our approach to parse the C++ header files and then generate CDL files for Otcl. This would remove one of the primary obstacles to our use of Otcl.

Listing 3, shows a simple example of using the C++ Visualization Toolkit from within Tcl. More information on The Visualization Toolkit can be found at http://www.cs.rpi.edu/~martink. While our wrapper generator isn't suitable for many C++ class libraries, you can obtain a copy of it by emailing a request to martink@crd.ge.com.

I would like to acknowledge Bill Lorensen and Will Schroeder for their suggestions, encouragement and help in this work.

References

[1] P. Bogdanovich. "Objective-Tcl: An Object Oriented Tcl Environment" *Proceedings of the Tcl/Tk Workshop*, Toronto, Ontario, Canada, July 6-8, 1995.

[2] W. E. Lorensen, B. Yamrom. "Object Oriented Computer Animation." *Proceedings of IEEE NAECON*, 2:588-595, Dayton Ohio, May 1989.

[3] M. J. McLennan., "[incr Tcl]: Object-Oriented Programming in Tcl" *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11, 1993.

[4] M. J. McLennan, “The New [incr Tcl]: Objects, Mega-Widgets, Namespaces and More” *Proceedings of the Tcl/Tk Workshop*, Toronto, Ontario, Canada, July 6-8, 1995.

[5] W. Schroeder, K. Martin, B. Lorensen. *The Visualization Toolkit: An Object Oriented Approach to 3D Graphics*. Prentice-Hall, Englewood Cliffs, NJ, 1996.

[6] D. Sheehan. “Interpreted C++, Object Oriented Tcl, What next?” *Proceedings of the Tcl/Tk Workshop*, Toronto, Ontario, Canada, July 6-8, 1995.

[7] D. Wetherall, C. J. Lindblad. “Extending Tcl for Dynamic Object-Oriented Programming” *Proceedings of the Tcl/Tk Workshop*, Toronto, Ontario, Canada, July 6-8, 1995.