USENIX Association

Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference

Monterey, California, USA
June 10-15, 2002

Interactive 3D Graphics for Tcl

Oliver Kersting and Jürgen Döllner

kersting@hpi.uni-potsdam.de, doellner@hpi.uni-potsdam.de

Hasso Plattner Institute for Software Systems Engineering

University of Potsdam

Abstract

This paper presents an approach to integrate interactive real-time 3D graphics into the scripting language Tcl. 3D graphics libraries are typically implemented in system programming languages such as C or C++ in order to be type safe and fast. We have developed a technique that analyzes the C++ application programming interface of such a library and maps it to appropriate scripting commands and structures. As 3D graphics library, we apply the Virtual Rendering System, an object-oriented library that supports 3D modeling, interaction, and animation. The mapped API represents a complete and powerful development tool for interactive, animated 3D graphics applications. The mapping technique takes advantage of the weak typing and dynamic features of the scripting language, preserves all usability-critical features of the C++ API, and has no impact on performance so that even real-time 3D applications can be developed. The mapping technique can be applied in general to all kinds of C++ APIs and automated. It also gathers reflection information of the API classes and supports interactive management of API objects. Consequently, interactive development environments can be built easily based on this information. We illustrate the approach by several examples of 3D graphics applications.

1 Introduction

In 3D computer graphics and multi-media, a large number of software libraries have been developed in the past. Examples include the 3D graphics library OpenInventor [15], the visualization toolkit VTK [13], the 3D rendering system OpenGL [16], and an upcoming standard for media creation and playback called OpenML [7]. As a common characteristic, these libraries provide low-level C/C++ application programming interfaces (APIs) in terms of functions, data types, and classes. Developing applications based on these APIs is generally regarded difficult because developers are frequently confronted with a multitude of functionality and the implications of strong typing, and typically it becomes difficult to explore and use the library's capabilities.

Scripting languages have been successfully used to develop large, complex software systems [11]. The reasons for this include: 1) Scripting languages simplify gluing together existing software components to a single application. The simplification results mainly from weak typing because this allows developers to interface components that use incompatible data types at the programming-language level. 2) An API of a pre-built library can be integrated by adding new commands to the scripting language. Such an extension to a scripting language is called *binding* of an API. A binding further simplifies the application development because it gives developers easy access to the API and does not require detailed knowledge of the underlying system programming language. 3) Scripting enables developers to experiment with libraries and their functionality in a quite different way compared to studying a C++ API documentation. Interactively, developers can

- instantiate objects and see immediately whether their services fit to the problem to be solved;
- inquire available classes, methods, method arguments, and other public class elements;
- inquire current objects and manage these objects;
- modify and deploy objects; and
- easily integrate object management and object visualization into the graphical user interface.

In this paper, we discuss how to bind the C++ API of the Virtual Rendering System VRS [3], a complex and large library for real-time 3D computer graphics, to the scripting language Tcl [10]. The resulting Tcl/Tk package is called *interactive VRS* (iVRS). The *mapping*, which denotes the process of generating the binding, is based on an automated analysis of the class interfaces and generation of appropriate wrapper classes. The binding, however, must guarantee that a) all features of the API are transparently accessible in the scripting language and b) no major performance penalties are introduced. As proof-of-concept, we have developed an interactive real-time 3D-map system, called *LandExplorer*, which can be used to visualize and explore geo data (Figure 1).

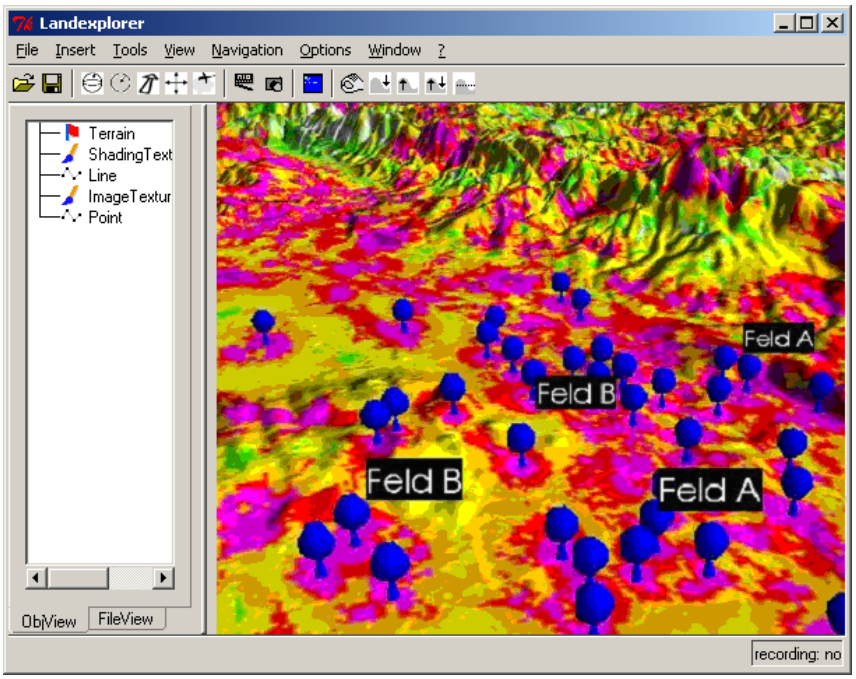

Figure 1: The iteractive 3D-map system *LandExplorer*, which has been built with iVRS. LandExplorer is used to visualize, explore, and analyze geo data.

Section 2 briefly discusses related work. Section 3 outlines main aspects of the Virtual Rendering System and its API. Section 4 discusses the requirements a binding should meet to simplify application development. Section 5 explains methods for mapping C++ APIs to Tcl bindings. Section 6 gives examples that illustrate how developers can interactively use iVRS. Finally, Section 7 draws some conclusions.

2 Related Work

Tcl is basically a procedural language bundled with the powerful user-interface toolkit Tk. A couple of object-oriented extensions exist that integrate many concepts of object-oriented programming languages in Tcl (e.g., classes, encapsulation, inheritance etc.), for example [incr Tcl] [8] and the extension of Sinnige [14], which mimics C++ class syntax. Using these extensions, developers can specify and implement classes in Tcl. However, it is rather difficult to implement real-time 3D graphics on top of these extensions because graphics data has to be efficiently processed and stored.

The Tcl interpreter can be extended by new commands, which interface external functions written in C or C++; object-oriented features of C++ are not directly supported by Tcl. Therefore, several techniques and tools, which are called *mappers*, have been developed to map the API of an existing C++ class hierarchy into adequate constructs of the Tcl language. A general C++-to-Tcl mapper is SWIG, the Simplified Wrapper and Interface Generator [1]. SWIG can map C, C++, and Objective-C classes into a variety of higher-level languages (e.g., Tcl, Python) by parsing the header files or special interface files and generating wrapper code for the scripting language. One limitation is that certain C++ features are not directly mapped, e.g., overloaded methods in C++ cannot be handled without a name affix, and smart pointers are not supported. In our approach, we focus on usability-critical features of C++ APIs such as overloaded methods, object management, memory management, and class reflection.

TkOGL [4] directly maps the OpenGL API to Tcl. For each OpenGL C function, there is a Tcl command. This way, simple scenes can be built based on OpenGL functions, but complex scene modeling can only be achieved using large numbers of calls to OpenGL C functions via Tcl, which decreases speed drastically.

The Itcl++ mapper [5] is based on [incr Tcl]. Itcl++ parses C++ headers and generates C++ wrapper code that is linked to generated [incr Tcl] classes. For each C++ class, C++ wrapper code and a corresponding [incr Tcl] class is generated. The inheritance relationships between classes are preserved under this mapping. This way, new [incr Tcl] classes can be derived from generated [incr Tcl] classes. As a proof of concept, the authors completely wrapped the C++ API of the OpenInventor 3D graphics library. The limitations of this approach include the missing support for abstract classes and overloaded methods and the costly mapping process. Our approach targets the same category of C++ APIs but does support important C++ API features while using significantly less resources because classes

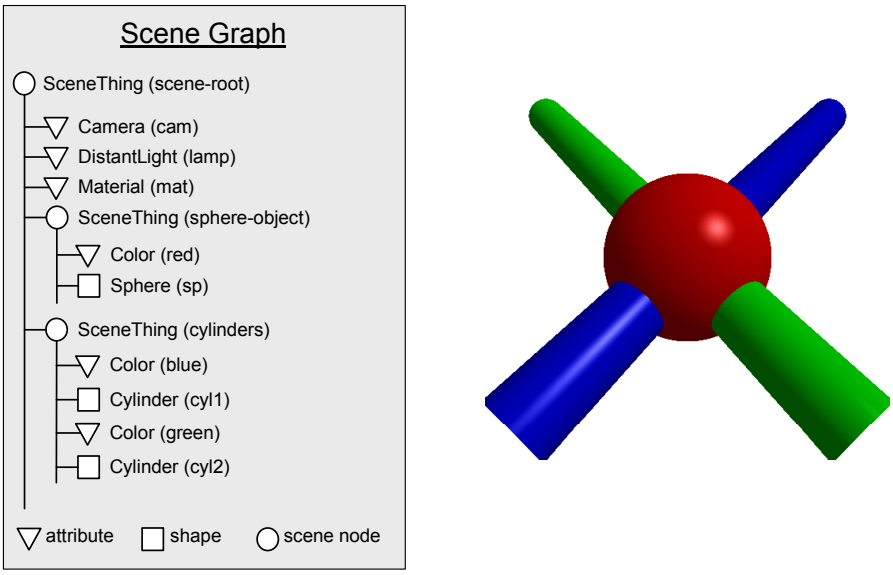

Figure 2: Example of a VRS scene graph (left) and a snapshot taken from the rendered scene (right).

are not mirrored into classes of an object-oriented extension of Tcl.

The TclObj mapper [12] works based on C++ macros and requires the original C++ source to be modified in order to access attributes and methods. As a major disadvantage of this approach, the mapping process is not automatic.

The 3D game engine Nebula [9] represents an example of a complex C++ library wrapped by Tcl focusing on game development. The Nebula library is based on the scene graph paradigm and can be extended by C++ class inheritance. Through the Tcl API, developers can construct and configure scene graphs. Nebula provides a good example of how efficiently real-time 3D games can be created by scripting languages. VRS, however, is a general-purpose 3D graphics library, intended for any kind of 3D application.

3 The 3D Graphics Library VRS

The *Virtual Rendering System* [3] is a 3D graphics library providing a large collection of 3D building blocks including different types of geometries, graphics attributes, and state-of-the-art real-time 3D rendering techniques such as shadowing, reflection, and multitexturing. Interactive, dynamic 3D applications are implemented by scene graphs and behavior graphs. A scene graph specifies geometry and appearance of a 3D scene, whereas a behavior graph specifies event handling, time-dependent actions, and constraint handling [2]. VRS supports advanced real-time rendering techniques such as texture-based shadows, reflection, or multi-texturing capabilities and encapsulates low-level OpenGL techniques like P-buffer rendering [17].

The scene graph is composed of scene nodes, called *scene things*. The nodes basically serve as containers for *node components*. Node components include shapes (3D geometries), graphics attributes (e.g., color, material specifications, light sources), geometric transformations (e.g., rotation, scaling, translation), and child nodes, which represent scene subgraphs (Figure 2). In analogy, the behavior graph is composed of behavior nodes that contain node components. For behavior nodes, node components include constraints, time requirements, time layouts, and event handlers. To display a scene, rendering engines traverse scene graphs and interpret the node components of the nodes. To animated and interact with scenes, events are sent through and processed by behavior graphs.

The API of VRS uses common C++ language features such as classes, encapsulation, method overloading, inheritance, and templates. The API is strictly object-based and maintains encapsulation, that is, objects are exclusively accessed by method calls. VRS makes a clear distinction between API classes and internal classes (e.g., interface classes and private classes).

The VRS API can be considered as a typical API of an object-oriented graphics or multimedia library, whose usage can be characterized by the following main tasks:

- *Constructing models.* To construct scene graphs and behavior graphs, developers search for appropriate classes, look through the services they provide, and use instances of these classes to construct 3D scenes.
- *Modifying models.* To modify models, their components have to be accessed. To do so, developers want to inquire objects of a certain kind currently

instantiated, inspect their state, inquire provided services, and call methods.

- *Evaluating models.* To evaluate models, they are interpreted with respect to an underlying rendering system or multimedia system. In VRS, for example, the OpenGL engine traverses scene graphs to render them, whereas the ray-request engine traverses scene graphs to find picking results.

Completeness and simplicity are the most important quality criteria for any API mapping technique. In addition, no source code modifications should be necessary to generate an API mapping.

The ability to extend a graphics library at Tcl level (e.g., deriving new Tcl classes from mapped Tcl classes) is generally not required because to implement extensions, access to low-level functionality of the underlying system libraries (e.g., OpenGL) is necessary. When the C++ library has been extended, the mapping process can be invoked again. Developers able to extend the library will also be able to control the mapping process.

4 Requirements of API Bindings

A binding of an API should allow developers to easily access all interface elements of the C++ API within an interactive environment such as the Tcl interpreter. This way, developers can experimentally explore the API and incrementally develop applications at runtime. For this, the API binding should support creating, inspecting, modifying, and destroying objects by Tcl commands.

The API binding needs an efficient and powerful mechanism to identify and call methods of the C++ API. In particular, string arguments of Tcl commands must be automatically interpreted and converted to typed C++ arguments (and vice versa), and a corresponding method must be chosen based on the method signature. To illustrate that API mapping is subtle undertaking, let us consider a few details of the VRS API:

- *Objects with identity versus objects as values*. Most VRS classes are derived from the root class `SharedObj`, which represents shareable, dynamically allocated objects. Shareable objects are known by their identity. Objects that are handled like values are not derived from `SharedObj`. These classes include, for example, mathematical classes such as `Vector` or `Matrix`. The mapper must be able to treat both categories of objects.
- *Reference counting.* The class `SharedObj` implements a semi-automatic memory management. If an object has a link to another object, it references that object. If the link is no longer needed, the object un-references the other object. If an object's reference counter becomes zero, it is not referenced by any other object and, therefore, gets deleted. The API binding can take advantage of the reference counting mechanism to avoid memory leaks and to provide convenient memory management from a developer's point of view.
- *Method overloading.* Many VRS classes specify overloaded methods (e.g., two or more constructors). If overloaded methods differ in the number of arguments, they can clearly be distinguished. If not, the API binding must be able to distinguish the argument types although Tcl does not support argument typing.
- *Default argument values.* Methods can define default values for their arguments. The API binding should allow us to call these methods with and without default arguments.
- *Abstract classes.* The API binding must detect abstract classes and prohibit instantiating objects of them.
- *Template classes.* All VRS data container classes are implemented as template classes. The API binding must generate frequently used template classes in advance.

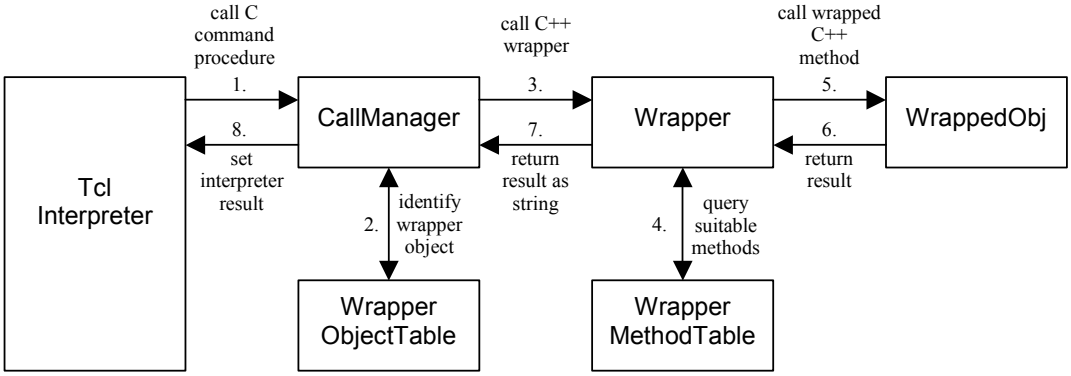

Figure 3: The handling of method calls by the Tcl binding.

- *Enumerations.* Symbolic names specified as enumeration constants are frequent elements of VRS classes. The API binding must preserve these names in its communication with the Tcl interpreter.
- *Static class elements.* Static methods are frequent elements in VRS classes. The API binding must integrate static methods similar to the syntax used in C++, that is, static methods should be accessible via the class name.

VRS consists of approximately 530 C++ classes; about 280 are relevant for being mapped to Tcl. These 280 classes include 6 numerical classes and about 45 template classes. All classes implement about 2500 methods. The 70 base classes define 54 abstract methods. Method overloading occurs 88 times.

5 Mapping Techniques

Our mapping technique concentrates on object-oriented language features of C++. It maps, as basic constructs, classes and template classes. In the case of templates, concrete instantiations must be specified. For each class, it can handle virtual and non-virtual methods, static methods, overloaded methods, and arithmetic operators. In addition, single inheritance is supported.

The mapping technique does currently not support direct access to member variables, non-constant references, namespaces, nested classes, and multiple inheritance. These language features rarely occur in the VRS API. They could be added in a straightforward way to the mapping technique.

The key idea for mapping a C++ class to the Tcl binding is to create a *wrapper class*, which collects interface information and reflects C++ class methods with their signatures consisting of string arguments only. The mapping technique obtains interface information by parsing the C++ header files. The wrapper class uses *converter functions* to transform between strings and C++ data types, and it links wrapper methods and wrapped methods (Figure 3). Both, wrapper classes and converter functions are generated automatically.

The C++ object instantiated by the library is called *wrapped object*. It is accessed by Tcl through a corresponding *wrapper object*. The *call manager* handles VRS-related scripting commands. First, it identifies the wrapper object by the *wrapper-object table*. Next, it is looking through the *wrapper-method table* of the wrapper object for an appropriate method, called the *wrapper method*. If found, it calls that method, which converts its string arguments into C++ data types and subsequently calls the corresponding method of the wrapped object.

5.1 Wrapper Classes

Wrapper classes are responsible for type-sensitive method selection and delegation of their execution to C++ method calls. The class **`Wrapper`** is the common base class of all concrete wrapper classes. It defines methods for identifying the class of an object and all its base classes. The "class" of the wrapper base class is **`void*`**. Derived wrapper classes specialize the identification methods. For example, the wrapper class for the **`Sphere`** class defines that its wrapped objects belong to the **`Sphere`** class and its base classes **`Shape`** and **`SharedObj`**.

A wrapper method expects an array of strings as arguments. The following example (Figure 4) shows the

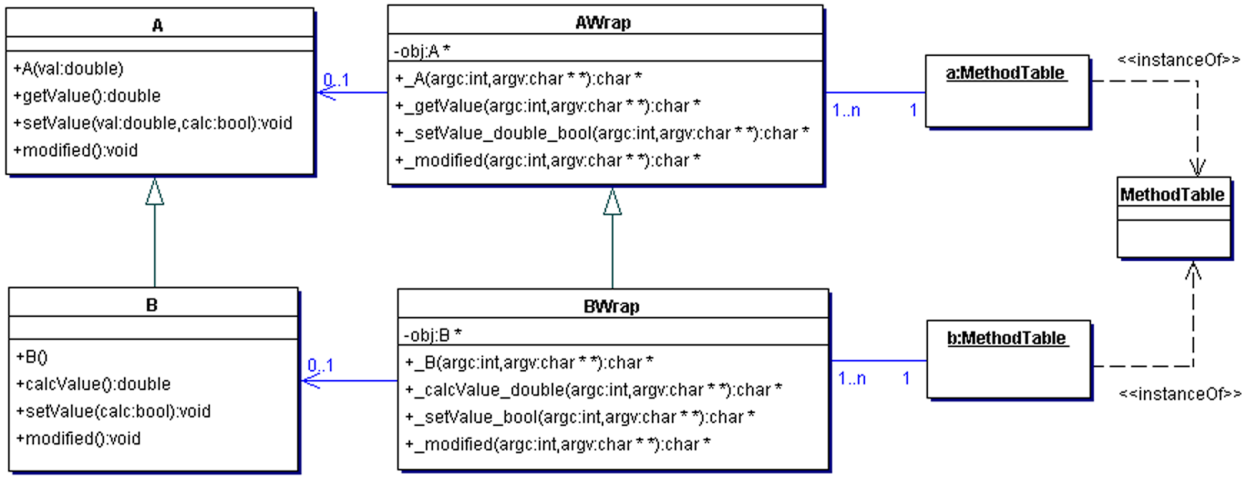

Figure 4: A sample class hierarchy (A and B), its corresponding wrapper classes (`AWrap` and `BWrap`), and their associated wrapper-method tables.

data relations among the class **A** and its wrapper class **AWrap**, implemented as follows:

```
// wrapped C++ class
class A {
    double value;
public:
    // constructor
    A(double val = 1.0);

    // set/get value
    double getValue();
    void setValue(double val,
                  bool calc = true);

    // polymorph method modified
    virtual void modified();
};

// C++ wrapper class
class AWrap : public Wrapper {

    // pointer to wrapped object
    A* obj;

public:

    // constructor wrapper method
    char* _A(int argc, char** argv);

    // set/get value wrapper method
    char* _getValue(int argc, char** argv);
    char* _setValue_double_bool(int argc,
                                char** argv);

    // modified wrapper method
    char* _modified(int argc, char** argv);

};
```

The implementation of two typical wrapper methods is outlined in the following example (converter functions are named **type2str** respectively **str2type**):

```
char* Awrap::_getValue(int argc, char** argv){

    // call wrapped method
    double res = obj->getValue();

    // convert return type to char*
    return type2str(res);
}

char* Awrap::_setValue_double_bool
                (int argc, char** argv) {

    // convert argument 0
    double arg0 =
            str2type<double>(argv[0]);

    // convert argument 1
    // handle default argument
    bool arg1 = (argc <= 1) ? true :
            str2type<bool>(argv[1]);

    // call wrapped method
    obj->setValue(arg0,arg1);

    // no return value (void)
    return NULL;

}
```

5.2 Wrapper-Method Table

The wrapper-method table stores signature information for each mapped method. The signature information includes method name, arguments, minimum and maximum number of method parameters, flags, and the pointer to the wrapper method (Table 1 and 2). The flags indicate special-purpose methods such as constructors (**CTOR**) or static methods (**STATIC**). The signature information is needed for correctly resolving overloaded methods and argument default values.

The mapper creates a wrapper-method table by copying the wrapper-method table of the base class (if any), except methods marked with **CTOR** or **STATIC**. Next, it inserts entries for methods declared in the class under consideration. An entry replaces a stored entry if method name and arguments are the same. This way, overridden methods are handled. For example, see the **modified** entry in the wrapper-method table of class **B** (Figure 4).

5.3 Memory Management

The Tcl commands **new** and **delete** create and destroy, respectively, wrapped objects. The constructor command has the following format:

```
% new ClassName arg1 arg2 …
objClassName1
```

The **CallManager** instantiates a new wrapper object depending on the class name written after **new** and iterates over all methods marked with the flag **CTOR** in the wrapper-method table. The first constructor method that can convert all incoming arguments is called and the wrapped object is constructed. If a shared object is created, its reference counter is incremented by one. The **CallManager** generates a unique name for the new object and returns it as result. To destroy an object we write:

```
% delete objClassName1
```

The **CallManager** is searching for the given wrapper object, removes the entry in the wrapper-object table, and deletes the wrapper object. If the wrapped object is a shared object, its reference counter is decremented by one. Otherwise the wrapped object is immediately destroyed.

5.4 Converting Arguments

Converter functions transform data values (respectively objects) between Tcl and C++: **str2type** converts a string to the C++ data value and **type2str** converts a C++ data value to a string. If a conversion fails, a **ConvertException** is thrown. The **CallManager** catches this exception. This way, overloaded methods are distinguished.

For standard data types (e.g., **char**, **int**, **double)** converter functions are built-in. Derived standard data types, like **double***, are interpreted as non-typed data, that means they are treated as **void***. Converter functions for object pointers can be handled type-safe because of the stored type information in each wrapper object. Converter functions for objects of classes derived from **SharedObj** can additionally use the C++ runtime type information and convert precisely to the underlying type. In addition, inline conversion provides a complementary handling of arguments that are handled "by-value".

5.5 Inline Conversion

Numerical classes such as points, vectors, matrices, and rays are frequently used in graphics and multimedia applications. Numerical objects are mostly transient, i.e., applications use them by value. The example below shows how to add vectors:

```
% set v [new Vector 1 2 3]

% $v + "4 5 6"
5 7 9

% delete $v
```

To support numerical objects, we could use the same mechanism as for non-transient objects. In the example, we could initialize a new **VectorWrapper** object with the arguments "**1 2 3**", register and return its name, call the method "**+**" with the argument "**4 5 6**", deregister its name, and delete the wrapper object. This approach, however, is neither syntactically elegant nor computationally efficient.

Alternatively, we could implement all API relevant methods of numerical classes as Tcl procedures. This would duplicate the implementation, lead to less efficient implementations, and involve manual work.

To cope with transient objects handled by-value, our mapping technique supports inline conversion of numerical objects using directly transient C++ objects. The vector example above is written as follows:

```
% VECTOR "1 2 3" + "4 5 6"
5 7 9
```

The **VECTOR** function initializes a static **VectorWrapper** object with the first argument "**1 2 3**", calls the method "**+**" with the argument "**4 5 6**", and returns the result. This way, we save time for creation, registration, deregistration, and destruction of the wrapper object. Inline conversion functions are generated automatically for numerical classes such as **Vector**, **Color**, **Matrix**, **Ray**, and **Area**.

5.6 Enumerations

Enumerations typically represent integer constants by symbolic names and, this way, facilitate the usage of these constants. An enumeration in C++ is defined as a pair of integer value and name.

Our mapping technique supports C++ enumerations similar to the C++ syntax by **classname::enumname**. An enumeration specified this way in Tcl can be directly mapped to its C++ counterpart. To map a C++ enumeration to its Tcl name, the enumeration type (i.e., the class) must be known, otherwise only the integer value of the enumeration value can be returned.

```
% set polygon [new PolygonSet
                      PolygonSet::Quads]

% $polygon getType
PolygonSet::Quads
```

5.7 Overloaded Methods

The C++ compiler can differentiate between overloaded methods at compile time based on argument number and/or argument type. The Tcl interpreter cannot differentiate based on argument types because Tcl is typeless.

To solve this problem, our mapping technique uses a try-and-error strategy. In the case of overloaded meth-

Method Name	Arguments	Min.	Max.	Flags	Method Pointers
"A"	"double"	1	1	CTOR	AWrap::_A
"setValue"	"double bool"	1	2		AWrap::_setValue_double_bool
"getValue"	""	0	0		AWrap::_getValue
"modified"	""	0	0		AWrap::_modified

Table 1: Wrapper-method table for class A.

Method Name	Arguments	Min.	Max.	Flags	Method Pointers
"B"	""	0	0	CTOR	BWrap::_B
"setValue"	"double bool"	1	2		AWrap::_setValue_double_bool
"setValue"	"bool"	1	1		BWrap::_setValue_bool
"getValue"	""	0	0		AWrap::_getValue
"modified"	""	0	0		BWrap::_modified
"calcValue"	""	0	0		BWrap::_calcValue

Table 2: Wrapper-method table for class B.

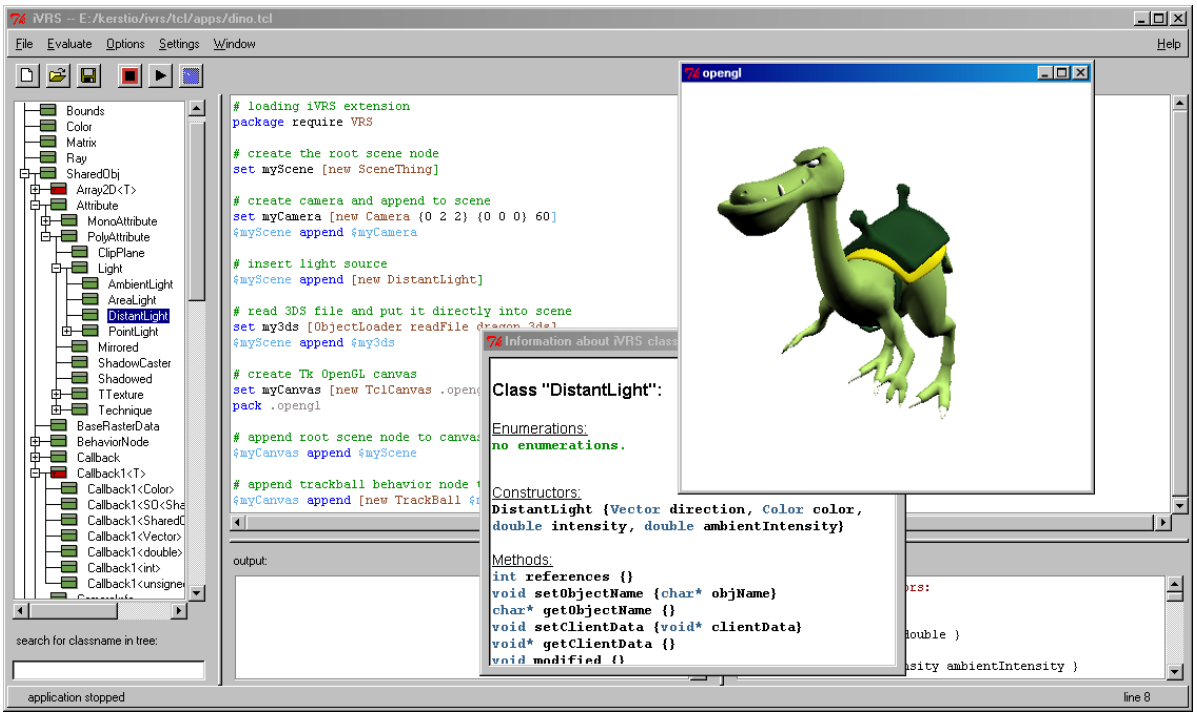

Figure 5: Integrated development environment for iVRS.

ods, we iterate through the method table searching for methods with the specified name and the correct number of arguments. For each suitable method, we try to convert argument strings into C++ data objects. If a conversion exception occurs, the next suitable method will be searched. If no exception occurs, the method call was successful. The method-table entry of a successful method is moved to the beginning of the table to reduce the number of method tests for the next call of the same method.

Some data types, however, cannot be distinguished by their value. For example, the characters `17` can be interpreted as `int`, `float`, `char`, or `string`. In such a case, the `CallManager` calls the *first* method that can convert all arguments of a method. To avoid this non-deterministic behavior, the name of the method can explicitly force a specific data conversion. Example:

```
$obj setValue:char 17
```

or force conversion as string

```
$obj setValue:string 17.
```

If no method with the explicit type is present, no conversion will be done and an error occurs. The explicit conversion is the fastest method because no method searching is required but the readability of the scripting code is reduced.

5.8 Class Reflection

The information gathered during the parsing process can be inquired at runtime. Each class or object can be analyzed regarding:

- parent and child classes,
- constructors with complete signature,
- methods including complete signature,
- enumerations,
- objects currently instantiated, and
- object relationships.

This information can be used to build sophisticated integrated development environments, including syntax highlighting for classes, methods and enumerations, class hierarchy browser, and run-time object browser (Figure 5).

6 Examples

iVRS represents a complete 3D graphics package for Tcl/Tk. In the following, we illustrate this along several examples.

6.1 3D Object Viewer

As a first example, let us develop a 3D object viewer. It can be used to view and inspect 3D objects constructed with AutoDesk's 3ds max™.

The script below completely implements the 3D object viewer. A snapshot of the application is shown in Figure 5.

```
# loading iVRS extension
package require VRS

# create the root scene node
set myScene [new SceneThing]

# create camera and append to scene
set myCamera [new Camera {0 -2 -2} {0 0 0} 60]
$myScene append $myCamera

# insert light source
set distantlight [new DistantLight]
$myScene append $distantlight

# read 3DS file and put it directly into scene
set my3ds [ObjectLoader readFile dragon.3ds]
$myScene append $my3ds

# create Tk OpenGL canvas
set myCanvas [new TclCanvas .view 400 400]
pack .view

# append root scene node to canvas
$myCanvas append $myScene

# append trackball behavior node to canvas
$myCanvas append [new TrackBall $my3ds]
```

The first command loads the VRS package and initializes VRS classes and wrapper tables. The application is implemented by a scene graph, a behavior graph, and a 3D canvas.

First, we create the root node of the scene graph and store its name in the Tcl variable **myScene**. The scene graph contains a virtual camera, a light source, and the subgraph that represent the 3DS object components. The **Camera** object defines camera position, camera focus and field of view angle. We activate the camera by appending it to **myScene**. To illuminate the scene, we insert a distant light into the scene graph.

VRS provides an object loading mechanism that supports several file formats (e.g., JPEG, TIF, 3D StudioMAX). A call to the static **ObjectLoader** method **readFile**, tries to find a reader for the given file and, if found, returns the object the reader creates. The name of the 3DS data object read from **dragon.3ds** is stored in the variable **my3ds** and inserted into the scene graph. The 3DS data object consists of a node whose node content objects represents geometry and graphics attributes of the 3DS object.

Next, we create an OpenGL canvas to display the scene graph. The **TclCanvas** can be treated as a usual Tk GUI component; it can be integrated in any Tk toplevel or container widget. The constructor requires a well-defined Tk pathname (**.view**). Then, we pack the widget to make it visible.

We link the canvas to the scene graph and to a behavior graph that consists just of a track ball node, which allows users to interactively rotate the 3DS object by mouse motion.

6.2 3D Object Viewer with Shadows

Let us extend the example of the previous section by adding shadows (Figure 6). Shadow rendering is based on shadow maps, a texture-based approach that is now supported by graphics hardware.

```
# loading iVRS extension
package require VRS

# create the root scene node
set myScene [new SceneThing]

# create camera and append to scene
set myCamera [new Camera {0 -2 -2} {0 0 0} 60]
$myScene append $myCamera

# insert light source
set distantlight [new DistantLight]
$myScene append $distantlight

# specify light source that casts shadow
set cast [new ShadowCaster $distantlight]
$myScene append $cast
$myScene append \
    [new ShadowCasterSwitch $cast true]

# read 3DS file and put it directly into scene
set my3ds [ObjectLoader readFile dragon.3ds]
$myScene append $my3ds

# specify objects that receive shadows
set shadowed [new Shadowed $distantlight]
$myScene append $shadowed
$myScene append \
    [new ShadowedSwitch $shadowed true]

# add box to make shadow visible
$myScene append \
    [new Box {-2 -1.1 -2} {2 -1 2}]

# create Tk OpenGL canvas
set myCanvas [new TclCanvas .view 400 400]
pack .view

# append root scene node to canvas
$myCanvas append $myScene

# append trackball behavior node to canvas
$myCanvas append [new TrackBall $my3ds]
```

In VRS, several attribute classes control the shadow rendering technique. In the example, we specify in which scene subgraphs shapes and light sources cast shadows (**ShadowCaster**) and in which scene graphs shapes receive shadows (**Shadowed**). Because shadowing is a global illumination phenomenon, we can control locally shadowing by switch objects (**ShadowCasterSwitch** and **ShadowedSwitch**)

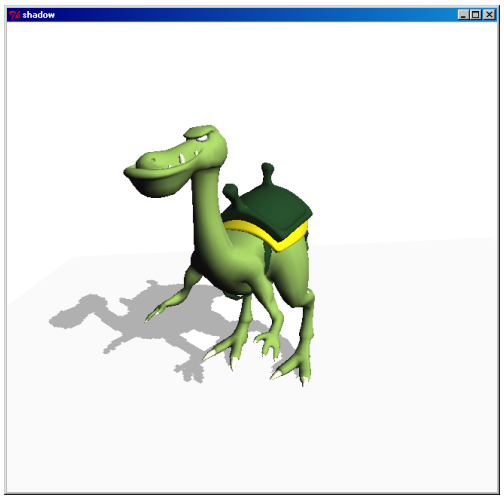

Figure 6: Viewer for 3D objects with shadows.

6.3 Integrating Tcl Scripts as Callbacks

In VRS, scene graphs and behavior graphs can also include nodes that define callbacks for certain types of events (e.g., time events, redraw events). Callbacks are able to call, for example, C functions or object methods. We extended VRS by a specialized callback class that encapsulates a Tcl script and invokes that script if the callback is activated. As a consequence, we can insert Tcl scripts at any position in scene graphs and behavior graphs. This way, the traversal of a graph is not entirely under control of the C++ engines but can be partially defined by scripts.

The following example extends example 6.1 by creating a Tcl callback that saves the current contents of the 3D canvas and writes the contents to an image file. The resulting image files could be compressed to an AVI or MPEG stream.

```
# snapshot proc
proc writeSnapshot {} {

    # make canvas available
    global myCanvas

    # save canvas content to Image object
    set myImage [$myCanvas snapshot]

    # make unique filename
    set filename snap[clock ticks].ppm

    # call static method to write ppm image
    PPMWriter writeFile $myImage $filename
}

# create TclCallback for writeContent
set myCall [new TclCallback writeSnapshot]

# create callback node for scene graph
set myRedraw [new SceneCallback $myCall]

# add SceneCallback to scene graph
$myCanvas append $myRedraw
```

The Tcl procedure **`writeSnapshot`** is responsible for capturing the current canvas contents into an **`Image`** object. The snapshot is saved as PPM file under a unique filename. To write the canvas contents after each redraw, a **`TclCallback`** object is used; it is inserted into the scene graph node **`myRedraw`** that invokes the callback in the case of a redraw event. Finally, we have to append that new node into the scene graph.

6.4 Development Environments

The API information, which is gathered during API analysis and stored as part of the API mapping, facilitates the construction of development environments. As core parts, we can take advantage of the collected information to build automatically control widgets for objects. The following example shows widgets that query constructor arguments and types, and instantiate GUI components based on this information, which are used to manipulate the initial values of the constructor arguments (Figure 7).

```
# specify the desired class
set what Sphere

# iterate over all constructors
foreach {name types args defs} \
                [VRS info ctors $what] {

    # iterate over all constructor arguments
    foreach t $types a $args d $defs {

        # build GUI components for arguments
        label .l$a -text "$a ($t)" -width 10

        # switch depending on type
        if {[string equal $t double]} {
            scale .e$a -variable $a
        } else {
            entry .e$a -textvariable $a
        }
        pack .l$a .e$a -side top

        # append to argument string
        append ctor_args "\$$a "
    }

    # button to create and insert object
    button .b -text Create -command " \
        set obj \[new $what $ctor_args\];\
        \$myScene append \$obj"
    pack .b -side top
}
```

The **`VRS info ctors`** command returns a list of all constructors of the specified class containing all argument types, argument names, and argument default values. The widget snapshots (Figure 7) show widgets for torus objects (defined by outer radius, inner radius, center, and three aperture angles), sphere objects (defined by radius, cutting planes in y, and aperture angle), box objects (defined by two corner points), and cone objects (defined by height, radius, and aperture angle).

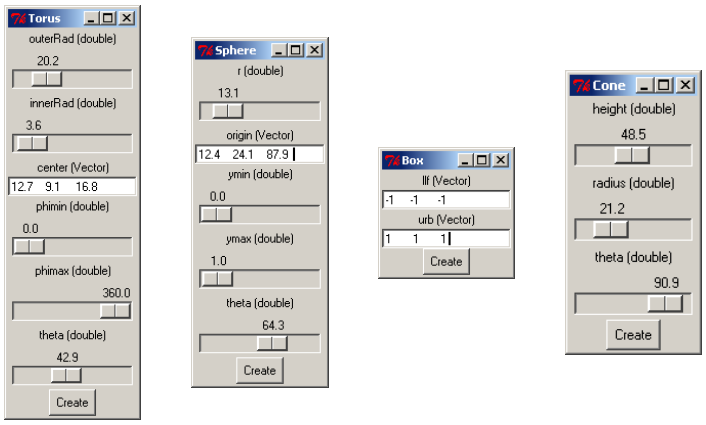

Figure 7: Automatically generated construction widgets for the classes `Torus`, `Sphere`, `Box`, and `Cone`.

Our mapping technique preserves not only the argument types of methods but also the argument names and their default values. This information is important for documenting the API and provides valuable information when developers want to interactively explore classes, objects and their methods. API information and run-time type information thus allow us to design integrated development environments in a straightforward way. Figure 5 illustrates such an environment built for VRS: Developers can browse classes, edit scripts, explore classes, inspect objects, and run applications.

6.5 An interactive 3D Map System

The mapping technique has been used to bind the APIs of VRS and LandExplorer, a 3D-map library built on top of VRS, to Tcl. Using iVRS and Tcl, we have implemented a complete interactive 3D-map system.

The 3D-map system supports real-time, multi-resolution terrain rendering, multi-texturing of the terrain surface, and integration of 3D objects into the terrain model. In addition, various exploration functions have been added such as information lenses, fly-throughs, and meta-views.

Although real-time terrain rendering is time critical, the API mapping has no noticeable impact on performance because scene graph traversal, the most critical part of the display, is performed within C++. Even if some callbacks of a scene graph or behavior graph use Tcl scripts, the overall performance is not being affected.

As main advantages for building complex 3D graphics applications we observed:

- Easier class, object, and method selection.
- Easier construction of scene graphs and behavior graphs.
- Easier implementation of variants.
- Easier execution of experiments.
- Rapid prototyping.

7 Conclusions and Future Work

The described API mapping technique copes with important C++ language features such as classes, overloaded methods, operator methods, enumerations, and inheritance relations. It also analyzes the API with respect to argument names and default values. A semi-automatic memory management facilitates the usage of objects through the scripting language. Objects used "by value" are treated differently by inline conversion. Furthermore, the mapping technique uses standard Tcl without the need for any object-oriented Tcl extension.

The mapping technique gathers API information (e.g., classes, methods, and arguments) and provides run-time type information (instantiated objects, list of available classes, etc.). Both are the prerequisites for interactive development environments.

As a proof-of-concept, the complete C++ API of the Virtual Rendering System has been mapped to Tcl successfully. Since scripts mainly construct and modify models (e.g., scene graphs, behavior graphs, and node components) but are not directly involved in evaluating these models (e.g., scene graph traversal), applications written with Tcl are almost as efficient as applications written in C++. Even real-time 3D computer graphics such as 3D terrain viewers can be implemented based on the mapped API.

Finally, we observed that developers can use the mapped API more easily than the C++ API because no detailed knowledge of C++ is required and the library's functionality can be interactively explored and immediately applied.

In our future work, we are going to support additional C++ language features such as namespaces and nested classes. Furthermore, C++ comments of public methods

should be recognized (e.g., doxygen style [6]) and integrated into the run-time class reflection of iVRS. The implementation of mappings to other scripting languages such as Perl or Python is currently under development.

VRS and iVRS are free software under the GNU General Public License and can be downloaded at `www.vrs3d.org`.

Acknowledgements

We would like to thank Konstantin Baumann, Stephan Brumme, Christian Günther, Florian Kirsch, Stephan Kirsch, Haik Lorenz, Marc Nienhaus, and Anne Rozinat for their contributions to VRS.

References

[1] D. Beazley (1997): SWIG Reference Manual. Department of Computer Science, University of Utah, www.swig.org/Doc1.1/PDF/Reference.pdf

[2] J. Döllner, K. Hinrichs (1997): Object-Oriented 3D Modeling, Animation and Interaction. *The Journal of Visualization and Computer Animation*, 8(1):33-64

[3] J. Döllner, K. Hinrichs (2001): A Generic 3D Rendering System. *IEEE Transactions on Visualization and Computer Graphics*, 8(2)

[4] C. Esperanca, S. Fels, J. Joyner (2000): TkOGL 3.0. www.ece.ubc.ca/~hct/projects/tkogl.html

[5] W. Heidrich, P. Slusallek, H.-P. Seidel (1994): Using C++ class libraries from an interpreted language. *Proceedings of TOOLS USA '94*, 397-408

[6] D. van Heesch (2002): www.doxygen.org

[7] Khronos Group (2001): OpenML Specification. www.khronos.org/OpenML_1-0_Final_Spec.pdf

[8] M. McLennan (1993): [incr Tcl] - Object Oriented Programming in TCL. *Proceedings of Tcl/Tk Workshop 1993*, Berkeley

[9] Nebula Game Engine (1998): www.radonlabs.de

[10] J. Ousterhout (1994): Tcl/Tk, Addison-Wesley

[11] J. Ousterhout (1998): Scripting: Higher Level Programming for the 21st Century. IEEE Computer, 31(3), 23-30

[12] F. Pilhofer (1997): Using C++ objects with Tcl. www.fpx.de/fp/Software/tclobj/tclobj-1.2.tar.gz

[13] W. Schroeder, K. Martin, B. Lorensen (1997): The Visualization Toolkit - An Object-Oriented Approach To 3D Graphics, Prentice Hall

[14] S. Sinnige (2000): Tclpp: An Object-Oriented Extension to Tcl www.geocities.com/SiliconValley/Network/2836/projects/tclpp/

[15] P. Strauss, R. Carey (1992): An Object-Oriented 3D Graphics Toolkit. *Proceedings SIGGRAPH '92* ,26(2):341-349

[16] M. Woo, J. Neider, T. Davis, D. Shreiner (1999): *OpenGL Programming Guide - 3rd edition*, Addison-Wesley

[17] C. Wynn (2002): Using P-Buffers for Off-Screen Rendering in OpenGL. *Nvidia Technical Paper*