Computer Vision Scripting

Stephan Scholze and Christian Widmer
Swiss Federal Institute of Technology
Communication Technology Lab, Computer Vision Group
8092 Zürich, Switzerland
scholze@vision.ee.ethz.ch

ABSTRACT

Accessing a large C++ class library for computer vision research through a simple and intuitive command language, is the major goal of the presented Tcl extension. The paper investigates into the seamless integration of objects and algorithms of the library into Tcl. This is achieved by making extensive use of the Tcl object technology. A typical use of the extended scripting language is for teaching undergraduate students, who do not have relevant programming experience, in computer vision practice.

1. INTRODUCTION

Computer Vision is the branch of science which studies how information, given in form of images, is processed and analyzed by computers. This broad area can be divided into three major subfields: image enhancement, image compression and image interpretation. Due to the storage requirements and the computational complexity of even modest computer vision applications, one typically has to use compiled languages like C or C++. Excellent libraries are publicly available as source code and as compiled versions for several different platforms. We decided to base our work on `vxl` [1], which is the result of merging and redesigning two systems, namely `TargetJr` [2] and `IUE` [3], both well known in the computer vision community. `vxl` is a true cross-platform C++ library for computer vision research, containing a wealth of carefully tested and optimized code. The benefits of using existing library code can not be overestimated. The library offers support for various image formats, hierarchical data-structures for geometrical and topological entities in two and three dimensions, feature extraction algorithms, strong numerical support, integration of `STL` containers [4] and much more. These building blocks for computer vision systems are readily available and reusing them minimizes development time and sources of errors.

Although `vxl` is well designed and documented, the user has to be familiar with C++ language features like templates, operator overloading and inheritance. Moreover, he has to bear long compile and link cycles during program development and he has to face technical problems like incomplete template instantiation, unresolved symbols or broken makefiles. Especially for beginners this means a twofold burden: understanding the theory behind the algorithms and understanding their interface and invocation. The presented work originates in a project at our lab, where computer vision practice courses are given for undergraduate students. For teaching purposes we first of all want to explain state-of-the-art computer vision methodology without obfuscating the basic principles with advanced programming topics. Therefore a generic, clear and expressive scripting language for computer vision was developed on the basis of Tcl. The presented Tcl extension constitutes a framework for wrapping `vxl` algorithms and data-structures. The resulting application is an enhanced `wish`, which allows interactive computer vision programming.

Tcl is used as basis since it is easily extensible, it is mature, it supports various platforms, it has an apparent syntax, and it offers the graphical user interface toolkit Tk. Related work like `vtk` [5] already demonstrated the suitability of Tcl/Tk as high level interface to a large C++ library. Approaches for automated wrapping of C/C++ code, e.g. `SWIG` [6] do not fulfill our needs for different reasons. First of all, extensively used C++ language features like operator overloading or templates are not supported. Second, the syntax of the automatically generated Tcl commands is often cumbersome. Consequently, automated approaches had to be abandoned for the moment.

The organization of this paper is as follows. Section two introduces the system from a users point of view. Especially the structure of the new language patterns is introduced here. Section three discusses several technical aspects of the implementation and therefore reflects the developers view of the system. In section four, a realistic example of the system in use is given.

2. THE COMPUTER VISION SCRIPTING LANGUAGE

This section introduces the structure of the new Tcl commands by means of a few examples. One of the primary design goals of the computer vision scripting framework was the simplicity of the new language patterns. The language has to be as simple as possible, intuitive and consistent – for short: easy to use. This goal can be translated into requirements imposed onto the usability of the system. For example, every command has a simple name and behaves as a naive user might expect. Errors are handled gracefully. Reasonable default values for every parameter an algorithm might need are provided.

The command structure is illustrated in the next example. Consider the following piece of code, which loads an image from disk. The command returns a handle for the image; see section 3 for details about the handle concept.

```
vxlImg create -file <imageFile>
```

Here, `vxlImg` is the name of the *command*, `create` the *operation* and `-file <imageFile>` a *parameter*. Formally, the connection between commands, operations and parameters is depicted in Figure 1. This example already shows the general structure: each command corresponds to a data-structure of the computer vision library, hence the term 'command' can be seen as synonym to 'data-structure' in this context.

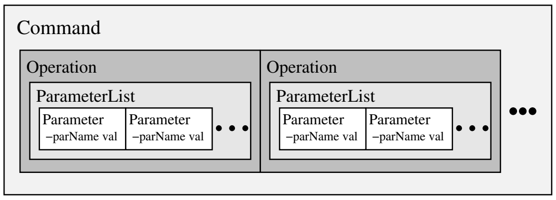

Figure 1: General structure of the new commands. Each command corresponds to a `vxl` data-structure. Associated with each command are several operations, each operation in turn can have parameters. The parameters are given in form of name/value pairs.

Different operations are associated with each command (`vxl` data-structure). The operations query, manipulate or process the data-structure represented by the command. Here, the `vxlImg` command corresponds to a `vxl` data-structure representing the generic image class. Algorithms which operate on this kind of images appear as operations associated to the `vxlImg` command. This emphasizes the connection between the algorithms and the data they operate upon. In case of the `vxlImg` command, some of the possible operations are: creation and deletion of images, cropping and warping, thresholding, conversion to a grey-level image, feature extraction, etc.

Operations are accompanied by parameters, these appear as name/ value pairs. Some parameters are mandatory, however, default values are provided for all optional ones.

The second example is slightly more complex. It will be used to illustrate two points. Firstly, the presented computer vision scripting language is generic enough to allow different solutions to a given problem. Secondly, the scripting language has a clear and consistent syntax. In the following, a threshold operation is applied to an image. This operation sets all pixels with values below a specified threshold to zero and the ones above the threshold to a fixed value.

Since thresholding is available as operation for the `vxlImg` command, this task can be performed in two lines of code.

```
set img [vxlImg create -file <imageFile>]
set bin [vxlImg thresh -img $img]
```

In the first line, an image handle is assigned to the variable `img`. This handle is used in the second line, to refer to the specific image on which the threshold operation should be performed. The default value is used as actual threshold value, thus this parameter is omitted. The same operation can be implemented explicitly in the scripting language by means of the following procedure.

```
# read image, determine dimensions
set i [vxlImg create -file <imageFile>]
set h [vxlImg height -img $i]
set w [vxlImg width  -img $i]

# create new image to store thresholded version
set ti [vxlImg copy -img $i]

# set the threshold value
set th 128

# scan image
for {set x 0} {$x < $w} {incr x} {
  for {set y 0} {$y < $h} {incr y} {
    if {[vxlImg getpix -img $i -x $x -y $y] > $th} {
      vxlImg setpix -img $ti -x $x -y $y -grey 255
    } else {
      vxlImg setpix -img $ti -x $x -y $y -grey 0
    }
  }
}
```

Besides the already mentioned `vxlImg` command, the presented extension also supports commands for handling geometric primitives. For the representation of 2D points, which can also act as 2D vectors, there is a `vxlPnt` command. From two `vxlPnt` objects, a `vxlLne` can be constructed, which represents the geometry of a 2D line. Several operations for setting and querying data-members are provided, see Table 2 for an overview of available commands and operations.

command	operation	description
`vxlImg`	`create`	load image from disk
	`delete`	free storage and unregister name
	`width`	get image width
	`height`	get image height
	`setpix`	set value of a pixel
	`getpix`	get value of a pixel
	`copy`	create a deep copy of the image
	`rgb2gray`	convert to grey-level image
	`band`	select on color band from RGB
	`thresh`	perform threshold operation
	`mask`	perform mask operation
	`crop`	crop a rectangular image region
	`warp`	perform translation, rotation and scaling of image
	`corr`	compute cross-correlation with a provided template
	`canny`	perform edge detection and line fitting
	`to_tk`	create valid TkPhotoImage representation
	`save`	save image to disk
`vxlPnt`	`create`	create point with given coordinates
	`add`	add two points by adding their components
	`sub`	subtract two points by subtracting their components
	`equ`	check for equal components
`vxlLne`	`create`	create a line from two points
	`length`	query Euclidean length of line
	`ab0`	query slope and intersection point with y axis
	`scalar`	compute inner product
	`dist`	compute perpendicular distance from point to line

Figure 2: List of the new commands with associated operations.

Additionally, a library with auxiliary Tcl procedures is available via the Tcl package mechanism. This library mainly contains procedures for displaying objects of type `vxlImg`, `vxlPnt` and `vxlLne` in a standard canvas widget. Displaying the result of the previous example is straightforward:

```
vxlImgShow $th
```

3. GENERIC WRAPPING

In this section the implementation details of the presented framework are described. Here we concentrate on the Tcl/Tk part; the computer vision related part of the problem (concerning the use of the `vxl` library) will be only discussed as far as needed. For more information about the `vxl` library, the interested reader might want to consult the web page [1]. A schematic representation of the components of our system is given in Figure 3.

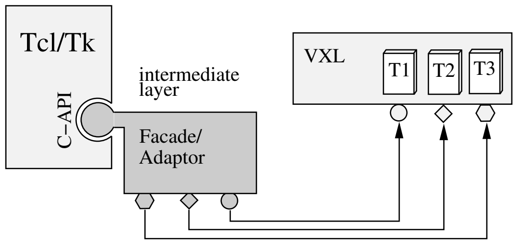

Figure 3: Schematic representation of the relations between the components which contribute to the system. The `vxl` library contains computer vision algorithms and data-structures (denoted T1, T2, T3). These are wrapped by an intermediate software layer and connected to the Tcl C-API.

The building blocks of the system are the `vxl` library which contains the data-structures and computer vision algorithms needed and a Tcl interpreter which glues the new components together and provides all constructs required to write programs, e.g. control structures, mathematical evaluations, subroutines and so forth. An *intermediate layer* connects the library with the Tcl C-API. This layer acts as both, as facade and as adapter [7]. As adapter, it converts the interface of the library into an interface conforming to the Tcl C-API. As facade, the intermediate layer is used to create an unified interface to the set of used classes and to bundle functionality which is spread over several classes. In the following, the intermediate layer is discussed in more detail. First, an overview of the underlying Tcl object system is given. Then, the structure of the auxiliary classes, used to interface the library with Tcl, is presented.

3.1 Tcl Object System

With the release of Tcl version eight, Tcl has a new system for representing data internally which substitutes the former string representation. This system is based on *dual ported objects*. Here, *object* does not refer to any object oriented concepts. Instead, Tcl objects can be thought of as containers, holding a specific data type and providing functions for the management of this datum. The term *dual ported* relates to the fact, that an object has two representations. One is the internal representation by which the object is manipulated from `C` code. The other one is the string representation of the object. The string representation is used when the object is accessed via Tcl commands. Since either representations are cached and the conversions between the two are preformed only when needed, this concept prevents a lot of time consuming transitions between the internal and the string representation. The Tcl object system allows `C` programmers to introduce new types of objects into Tcl, which behave like the built-in ones. In order to do so, every new object type has to be registered at the interpreter, together with pointers to a set of functions, needed by the interpreter to manage the object. More precisely, the following `C` functions have to be provided by the programmer of a new Tcl object:

- a function to create a valid string representation from the internal representation of the object,
- a second function, doing the inverse conversion from the string representation of the object to its internal representation,
- a function to copy an internal representation from one object to another,
- and finally a function which deletes the object by freeing the memory used by the object.

In our system, Tcl objects are used to represent data-structures of the `vxl` library, hence the name *object* is appropriate. Nevertheless, the new Tcl objects are created and manipulated through commands, hence the new Tcl objects appear as commands inside the Tcl shell. We distinguish two different kinds of Tcl objects: *named objects* and *anonymous objects*. The proper distinction of the two types is given below.

Named Objects

Named objects are Tcl objects which provide no string representation of their content but an unique name (called *handle*) instead. This handle is used to access the object. A well known representative for this technique are the file handles for Tcl file I/O. In our system, named objects are used to represent images. Named objects behave similar to Tk widgets in the sense that they are globally defined and therefore do not go out of scope. Named objects are managed by the system inside an object pool. The memory management is also done by the framework. Although the named objects are well integrated, a particular drawback exists: since named objects reside in the global name-space, they have to be deleted manually when no longer needed.

Anonymous Objects

Anonymous objects behave exactly like Tcl built-in objects. They are solely managed by the Tcl interpreter. The name was chosen to emphasize the fact, that no handle is associated with this kind of objects. Anonymous objects have all the advantages, built-in objects have: automatic management of the object in an object pool, automatic memory management, support of type-checking and type-conversion. Tcl build in commands can interact naturally with new anonymous objects since these can provide a valid string representation. Finally, the context of an anonymous object (global versus local scope) is treated correctly. In the presented system all new commands for those a string representation can be defined, are implemented as anonymous objects. For example, the string representation of a two-dimensional point (`vxlPnt`) is defined as a list of two floats, containing the x and y coordinates of the point.

As mentioned above, we resort to named objects whenever no useful string representation of an object can be defined. This is the case for the `vxlImg` object used to represent images. This particular object is treated even more special, since it is desirable to display images within Tk. Tk supports displaying `TkPhotoImage` types in a canvas widget. Every time a `vxlImg` object is created, an empty Tcl object of type `TkPhotoImage` with the same name is created and registered in the interpreter. This name establishes a one to one correspondence between the Tcl object and the named object. Therefore a `vxlImg` object can be used inside Tk, whenever a `TkPhotoImage` is expected. The actual representation of the `TkPhotoImage` is created on demand, whenever the user wants to display a `vxlImg` inside a canvas widget.

3.2 Auxiliary Class Structure

The integration of vxl inside Tcl/Tk is done in a very generic way. Common tasks like parameter parsing or setting of default values is done by auxiliary classes of the framework. Generic tasks for integrating the library into Tcl are:

- mapping of data-structures of the library to new Tcl objects,
- managing this objects via new Tcl commands,
- integrating algorithms as operations of the new commands,
- and finally handling of parameters for the operations.

New Tcl objects and commands

There are slight differences between the wrapping of data-structures and algorithms. We start by describing, how data-structures are integrated into the system. Wrapping a vxl data-structure involves three layers of abstraction and two auxiliary classes of the framework. The lowest level of abstraction is the C++ data-structure itself (*library level* in Figure 4).

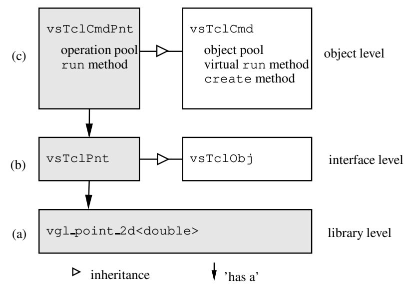

Figure 4: Three levels of abstraction are used to wrap a vxl data-structure. (a): Implementation of the data-structure as provided by the library. (b): The implementation is wrapped into a class derived from vsTclObj to enforce a common interface. (c): Highest level, contains the new Tcl object and and the corresponding command.

For illustrative purposes we choose a concrete example for this class. Let vgl_point_2d<double> be a class representing the coordinates of a two-dimensional point with double coordinate values (given by the template argument).

The next higher level (*interface level* in Figure 4) adds the functionality which will be needed to use the vxl data-structure as Tcl object, however this level of wrapping is not specific to Tcl. By deriving the class vsObjPnt from the abstract base-class vsTclObj, a common interface for all C++ data-structures integrated in the system can be enforced. The minimal interface consists of: a constructor, a copy constructor, a destructor and access functions to the built-in and – if possible – to a string representation of the data-structure.

The highest level of abstraction (*object level* in Figure 4) creates the new Tcl command representing the new Tcl object. Again, an abstract base class (vsTclCmd) ensures a consistent interface and functionality. The minimal interface consists of a create method called on system startup to register the Tcl object, the management functions for the object introduced in section 3.1, and a run method which will be invoked during runtime, whenever the new command is issued in the Tcl shell. The base-class also manages an object pool to keep track of all instances of the new Tcl object. Each concrete instance of the vsTclCmd class (e.g. vsTclCmdPnt) is implemented as a singleton to ensure only one instance of this class exists. This instance manages all known operations and provides methods to register and execute this operations and to handle their parameters. The connection to the Tcl C-API is only present here, consequently porting the system to another scripting language would mainly affect this level.

New algorithms

The difference in wrapping data-structures and algorithms is mainly a conceptual one. Algorithms operate on data-structures, therefore algorithms enter the framework as operations associated to a specific command (which in turn corresponds to a data-structure). Algorithms are integrated in the interface level (see Figure 4). This opens the possibility to bundle different functionalities of the library in one place to more compact operations.

During system startup, operations are registered at the vsTclCmd object in the command level. The vsTclCmd object has a hashtable to keep track of the available operations and parameters. When a command is invoked, the following steps are executed:

- read operation name from Tcl parameter list, and search for it in operation pool
- request a parameter list for the operation with default values,
- ask the operation to read available parameters into the parameter list,
- call the run method with the parameter list,
- and finally, request the operation to release the parameter list.

Due to the generic design, a specific operation can be added to a command by simply adding the operation to the run method of the command.

Parameter handling

One important helper class is for generic parameter handling. This includes parsing the parameters, type-checking and – if needed and possible – type conversion. A parameter is a pair consisting of *name* and *value*. The name of a parameter corresponds to its name used inside the Tcl shell. During the declaration of a parameter, the programmer has the possibility to provide a default value. Parameters passed to an operation during runtime from within the Tcl shell overwrite the default. This mechanism works analogous to the X window system configuration on UNIX systems, where system wide files contain all default values. These can be overwritten by user-specific values. Currently the following parameter types are supported: integers, floats, strings, anonymous objects, named objects and operations.

3.3 Extensibility and flexibility

The system was designed with respect to be generic and thus flexible enough to allow different approaches and solutions to a given computer vision problem. Therefore the proposed framework is open to extensions of different types and for different purposes. Three flavors of extensibility, together with their main purpose are discussed below.

Extensibility of the core
Due to the generic design of the framework, extending the core is a straight forward task. Both, adding new data-structures and algorithms have already been described in section 3.2. The purpose of extending the system this way is clear: providing more functionality of the library via an easy to use scripting language.

The Tcl library
As already mentioned, a Tcl library with some procedures to facilitate displaying tasks is available. This procedures encapsulate the code for displaying a `vxlImg`, including conversion to a `TkPhotoImage` and creation of a top-level window with a canvas adjusted to fit the image size. Once an image is displayed, procedures to draw points, lines and Tcl lists of these on the image canvas are available. Secondly, this library is also used to provide procedures which encapsulate several more generic steps into one high-level command. An example is the geometric manipulation of an image. The steps of rotating, scaling and translating an image are bundled in one procedure which can be called to perform this tasks at once. Here the purpose is to be able to present different levels of complexity to the users of the system.

The `C/C++`-plugin
For several reasons we have provided the possibility to integrate additional `C/C++` dynamically into the system. Firstly, a successful solution written in Tcl might be too slow, hence it could be advantageous to implement some inner loops in a compiled language. Secondly, more advanced users of the system, who already have experiences in `C/C++` should not be forced to use Tcl. Instead, they can use their preferred programming language and still use the system as a convenient tool for loading, accessing and displaying images and geometric primitives.

Runtime integration of `C/C++` code is realized with the dynamic load mechanism. The external code is compiled into a shared library which can be loaded using the `load` command. After loading, the user-provided `C/C++` functions are available as new Tcl commands. All Tcl specific aspects of command writing are handled by a plugin helper class, therefore the user can concentrate on the computer vision related part of the problem. The plugin class provides a set of helper functions which realize the actual interfacing with the core of the system. The most important helper functions are used for:

- declaration of new Tcl commands,
- reading arguments,
- accessing image data,
- creating new images,
- and returning values.

4. AN EXAMPLE

The presented system is currently used to teach undergraduate students basic methods of computer vision research. The students are given the task to write a program which is able to play Black-Jack. This mainly involves two steps: detection and recognition of a card in the field of view of a video camera, and the decision if a further card should be requested. A reasonably simple though far from optimal solution is to detect edges (this are strong intensity gradients) in the input image. The longest edges will describe the outline of the card. This information can be used to transform the image into a standard position in which the region of interest, containing the corner with the value of the card, can easily be extracted. By searching through a database containing images of all cards, the most similar one can be determined. Figure 5 shows screen-shots of the different processing steps.

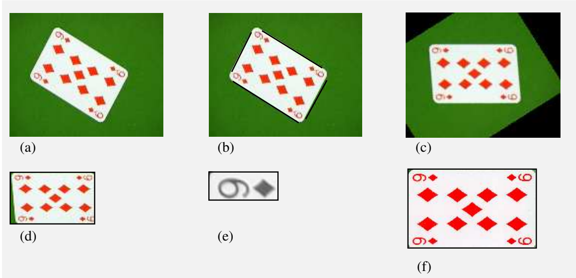

Figure 5: Steps of a successful card recognition. (a): Input image as provided by the video camera. (b): Detected edges in this image. (c): Rotated image. (d): Cropped image. (e): Region of interest. (f): Most similar template card.

5. CONCLUSION

In conclusion, our extension provides a generic and flexible high level scripting interface to a large computer vision research library. Significant effort was taken in integrating the support of the library for many different image formats, e.g. jpeg, png, tiff, pnm, bmp. Standard functions of the library like cropping, morphing, color decomposition, binarization and feature extraction are made accessible through Tcl commands. For all algorithms, reasonable default parameters were chosen, therefore the algorithms can be used immediately. The new Tcl language patterns were realized by making use of the Tcl object technology. A reasonable string representation is provided, such that the new objects can interact naturally with Tcl/Tk. Additionally a C-API for the data-structures shared by Tcl and vxl is included. This makes it easy to hook additional code into the framework. The first example application built with the presented Tcl extension is a computer vision system, able to play Black-Jack.

Acknowledgments

We are very grateful to the head of our institute, Prof. L. Van Gool for supporting this work.

6. REFERENCES

[1] The `vxl` book,
http://www.robots.ox.ac.uk/ awf/vxl/book/

[2] The TargetJr homepage,
http://www.targetjr.org/

[3] The IUE site at Amerinex,
http://www.aai.com/AAI/IUE/

[4] Standard Template Library Programmer's Guide,
http://www.sgi.com/tech/stl/

[5] The vtk homepage at Kitware,
http://www.kitware.com/vtk.html

[6] The SWIG homepage,
http://www.swig.org/

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides.
Design Patterns Addison Wesley, 1995.