Integration of C++ digital processing libraries and VTK through Tcl/Tk dynamic loadable extensions

Javier Suárez-Quirós, Daniel Gayo-Avello, Juán David González-Cobas, Rafael Pedro García-Díaz, Pedro Ignacio Álvarez-Peñín
Area of Graphic Expression in Engineering – GIworks (University of Oviedo)[1]

Abstract

After years of experience in computer graphics projects, GIworks workgroup at University of Oviedo came to the conclusion that some multipurpose graphics tools were an absolute necessity to perform rapid development on computer graphics. Such tools should provide basic facilities and should be easily adaptable to the kind of application on development to avoid starting from scratch every time a computer graphics project begins, allowing developers to concentrate on their project specific characteristics.

In an initial stage this tools would only consist on a multiplatform 2D and 3D visualization environment –which would provide methods to load and manage images and geometry– and an advanced digital image processing library –independent of but integrated into the environment like a plug-in–. This work methodology would be applied to whatever was developed in the future so this new code could be loaded in a transparent way and could modify the main tool so that the final look and feel for the user was a closed and coherent application.

1. Introduction

Nowadays the demand on computer graphics is huge in so different areas as engineering, biochemistry, medicine or geography, to talk only about a few. The reasons for such demand are multiple although we can focus in following ones:

- Almost immediate access to big data volumes with minimum effort.
- Easiness to extract initially hidden or deteriorated information.
- New ways on information management are offered.

Because of this many computer graphics projects are started; projects that, in great measure, repeat following circumstances:

1. Each project is different and requires a different and, many times, completely new conception.
2. All the projects share a very wide, and independent from the final goal, common base.
3. Most of the effort is invested in re-implement this common base, wasting previous work and affecting to the project specific development.

Keeping in mind all the above-mentioned, two suppositions can settle down:

- The only justifiable effort in a new computer graphics project is the necessary one for the development of that part that differentiates it from others.
- Loading of images and/or geometry, manipulation and later visualization are basic tasks, common to all the graphic applications and, therefore, widely reusable.

Starting on this suppositions, a toolkit providing all basic utilities for graphic treatment is needed; this toolkit should allow rapid applications development, allowing to the future user to be focused in his particular field without worrying about simple graphics management tasks. The multiplatform 2D and 3D visualization environment described in this paper is such a toolkit.

3. Goals

The visualization environment was planned as one only tool with two well differentiated work spaces: one for two-

[1] E.T.S.I.I.I.G., Campus de Viesques, s/n. Gijón – Asturias (SPAIN), quiros@etsiig.uniovi.es, uov02539@correo.uniovi.es, {cobas, pgarcia, pialvarez}@etsiig.uniovi.es

dimensional information manipulation (images) and another for three-dimensional information (geometry).

The first of this spaces, 2D manipulation, should provide methods to load, save and show images with different file formats; as well as the possibility to carry out conversions between formats and some minimum operations for image management.

The second one, 3D manipulation, would be similar to the previous one; offering ways to load, save and visualize geometry, tools to manipulate this geometry, as well as the lights and cameras in the scene, besides a basic material editor.

On the other hand, the tool had to be easily extensible and customizable; that is, external third party applications should be able to integrate in a transparent way, modifying the tool interface in such a way the user can use such utilities with the whole application.

2. Tool structure

To achieve such customization and extensibility goals it was clear the necessity to provide a simple enough structure so that the interface between the main tool and third party applications was minimum and, at the same time, flexible enough to be able to accept almost any kind of possibility.

This could only be possible by using loadable dynamic modules or plug-in's; each module would consist on a piece of code that could or not be loaded according to the purpose the final user wanted for the application. This way, in function of user's necessities one set or another would be used, constituting this sets plus the code that glue them together a coherent and closed environment.

Therefore, the whole tool would consist on a collection of modules dedicated to carry out different tasks communicating through a common channel, this channel is the application kernel. Therefore, the basic elements that would constitute the visualization environment would be the following ones:

- *Loader:* from the user's point of view, it would be the main program since it would be the only executable code; however, it is one of the less important parts since its only purpose is to load in memory the modules that will constitute the final tool.
- *Kernel:* composed by the 2D sub-kernel and the 3D sub-kernel; the first one would mainly provide methods to obtain memory for multi-layer images, to free this memory, to manage the status stack that constitutes each image and, also, to show the images. The 3D sub-kernel would offer mechanisms for geometry, lights and cameras management, as well as for the scenes rendering.
- *Plug-in's:* modules that will extend the tool functionality; although they would not be strictly necessary for the operation of the tool, the lack of some of them could cause an important limitation. Part of the functionality attributed at the beginning to the 2D sub-kernel, like different file formats management, would actually fall on extensions unknown by the kernel; this way it is easy to improve the tool's file format recognition in the future.

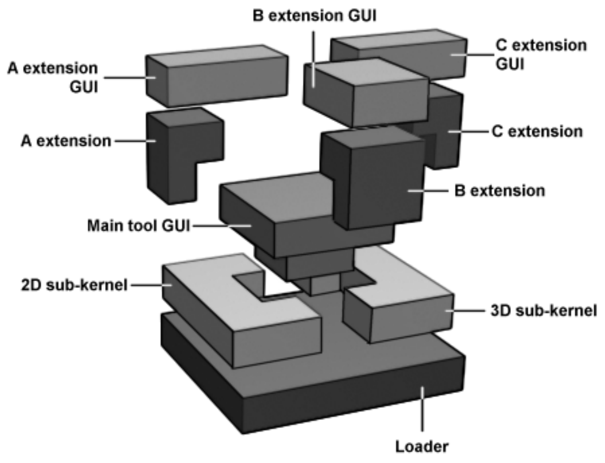


Figure 1: Tool structure.

In figure 1 the tool structure is shown. As you can see, the application interface allows to access part of the kernel functionality (constituted by the 2D and 3D sub-kernels). On the other hand, the extensions or plug-in's are composed from two different elements: the extension compiled code that interact and lean on the application kernel, and the GUI that interact with the application interfaces besides the extension code. At last, kernel, GUI and extensions lean on the loader.

4. Employed technologies

Previous exposition can be elegant but arises some questions about the way to carry out it: does exist any technology that...

- can be easily extended using an standard language?
- can allow the implementation of dynamic loadable modules?
- can provide simple, quick tools to develop GUI easily customizable by non-expert users?
- can be portable among different platforms?

The answer is, luckily, “yes”: modern script languages [Laird and Soraiz, 1999]. To explain in detail this languages nature is not the goal of this paper, although it is needed talk a little about some of their main characteristics, characteristic that make them specially suitable for a development as the one exposed.

Among the most attractive points in modern script languages we can stand out:

- They are open-source.
- They are interpreted languages so the software development process is speeded up.
- They are designed thinking of a wide portability among platforms.
- They are extensible by using C or C++ code.
- There are available extensions for GUIs development.

By using such a language the tool could be focused in the following way: the loader would be an interpreter for one of these languages while the kernel and the extensions would do it as dynamic modules written in C or C++; at last, the tool user interface would be built by using a toolkit for GUI development provide by the script language.

Among different available languages, Tcl and its extension Tk [Harrison and McLennan, 1998; Welch, 1997] for GUI development were chosen; mainly because of the language maturity and its well-known reliability on complex software projects. Following section will explain the way in which projects are developed with Tcl/Tk and the importance this approach would have to reach the goals outlined for this project.

4.1 Tcl/Tk

Tcl (Tool Command Language) comes from John Ousterhout’s work in the University of California at Berkeley; according to words Ousterhout:

The Tcl scripting language grew out of my work on design tools for integrated circuits [...] in the early 1980’s. My students and I had written several interactive tools for IC design [...] Each tool needed to have a command language [...] However, our primary interest was in the tools, not their command languages. Thus we didn’t invest much effort in the command languages and the languages ended up being weak and quirky. Furthermore, the language for one tool couldn’t be carried over to the next, so each tool ended up with a different bad command language. After a while this became rather embarrassing.

In the fall of 1987 [...] I got the idea of building an embeddable command language. The idea was to spend extra effort to create a good interpreted language, and furthermore to build it as a library package that could be reused in many different applications. The language interpreter would provide a set of relatively generic facilities, such as variables, control structures, and procedures. Each application that used the language would add its own features into the language as extensions, so that the language could be used to control the application. The name Tcl (Tool Command Language) derived from this intended usage

The idea was really bright: new applications would not be monolithic blocks of code but sets of a multipurpose language commands that could carry out simple (like generating a random number) or complex tasks (to obtain the edges for an image); the compiled code for that command set would be an extension that would be loaded by the language interpreter when it was needed and it could be repeatedly used in the future.

Mentioning again Ousterhout,

[...] Since most of the interesting functionality will come from the application, the primary purpose of the language is to integrate or "glue together" the extensions. Thus the language must have good facilities for integration.

This way of using Tcl code to integrate applications is fundamental; in fact, it is the main reason for the success in the previously described structure development, the reader will have already understood in which way the Tcl interpreter is the base for the loader while the kernel and plug-in’s are language extensions written in C/C++, glued and configured by using Tcl code; this extensions are immediately reusable thanks to its simple use as Tcl commands.

Apart from these language characteristics it is necessary to add the ones its extension Tk has; this is a toolkit for GUIs development; Tk allows to obtain great complexity interfaces in record time by using widgets. Tk, just as Tcl, is extensible so new components can be implemented for later use. One of them is Togl[2], a widget that allows to carry out OpenGL rendering; in an initial stage of project development it was thought use this widget although, in the end, it was discarded in favor of the Visualization Toolkit (VTK).

Therefore, Tcl applications can be structured in two ways: as a Tcl interpreter with some few specific application commands, as shown in figure 2, or by adding commands provided by Tk, besides by other extensions, as in figure 3.

[2] http://togl.sourceforge.net

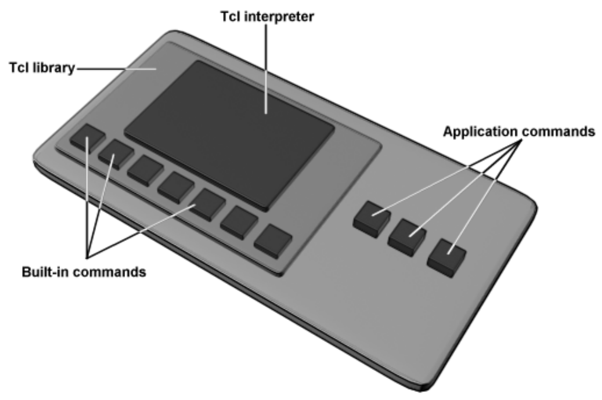


Figure 2: Tcl application.

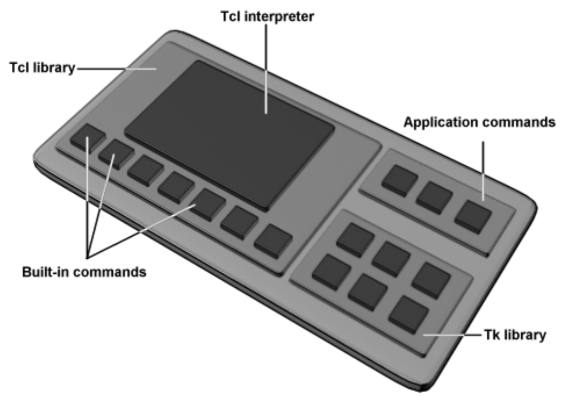


Figure 3: Tcl/Tk application.

The developed tool is a second type application: a program based on *wish*[3] loads and executes a number of scripts that, in turn, proceed to load all the available extensions as well as the Tk code Tk for the GUI and tool customization; this code uses commands defined by the Tcl library, by the tool kernel (that is, by the 2D sub-kernel or the 3D sub-kernel) or by the different extensions, commands whose code will have been written in C or in C++.

Once the way to develop the tool by mixing Tcl/Tk code (GUI development an integration) and C/C++ code (tool functionality implementation) had been established the project changed to a new phase to study the 3D sub-kernel[4] implementation.

4.2 VTK (Visualization ToolKit)

In previous section a widget is mentioned which is able to provide OpenGL rendering in Tcl/Tk, it is also affirmed that such a component wasn't used because the developers preferred a visualization library. Which were the differences between both approaches to discard one in favor of another? The reasons were mainly the same ones to carry out an implementation based on a combination of script and traditional languages instead of coding fully in C++: speed development and code reusability.

To develop the 3D sub-kernel with pure OpenGL would imply the implementation of an whole class hierarchy for geometry, lights, cameras, materials, interaction among objects and so on. However, that work becomes unnecessary if a framework that already provides this classes is used; nowadays there are a big number of visualization environments that carry out most of the performance needed for a computer graphics project, they are also able to be extended if needed by adding own classes and methods.

The best known visualization tools are IDL, IRIS Explorer, OpenDX and VTK[5], among others. What characteristics does VTK show to have been developed the 3D sub-kernel upon this library? We could stand out the following ones:

- Library source code is provided for free.
- It is portable among different Unix flavours and Windows.
- The user can extend it by adding his own classes.
- It provides wrapping programming interfaces for C++ (since it is written in this language), Java, Python... and Tcl!

As for the other tools, OpenDX is the only one open-source, IDL and IRIS Explorer require an user license; on the other hand, VTK is the only one which is provide as a “simple” linking library, the others are frameworks that allow application development by using an own language or in a visual way. Of course, only VTK provides a wrapping layer for other languages, mainly, Tcl.

[3] *wish* is a generic application that interprets Tcl/Tk code, it constitutes the base for GUI applications development; in fact, all the applications based on Tcl/Tk use the *wish* code.

[4] The 2D sub-kernel didn't need such a phase since its functionality is, in comparison with the 3D sub-kernel, reduced, focusing in multi-status and multi-layer image management, as well as visualization.

[5] Interested reader can find more information about these utilities in following links: http://www.rsinc.com/idl/index.cfm (IDL), http://www.nag.com/Welcome_IEC.html (IRIS Explorer), http://www.opendx.org (OpenDX) and http://www.kitware.com/vtk.html (VTK).

One of the VTK's strongest points is that programming interface between the library and interpreted languages as Tcl [Schroeder and Martin, 1999]:

The Visualization Toolkit consists of two basic components: a compiled C++ class library, and an "interpreted" wrapper layer that lets you manipulate the compile classes using the languages Java, Tcl, and Python.

The advantage of this architecture is that you can build efficient (in both CPU and memory) algorithms in the compiled language, C++, and retain the rapid code development features of interpreted languages (avoidance of compile/link cycle, simple but powerful tools, and access to GUI tools).

This is certainly interesting since it follows a very similar approach as Ousterhout's: to provide very efficient compiled code that is invoked by simple commands on a very high level script language. In this sense VTK could carry out the work required by the 3D sub-kernel, the question was if the library could be really used in huge graphic projects. The answer was highly positive, VTK doesn't only allow to develop fast but rather it provides really advanced characteristics that make of it a solid base for any kind of scientific visualization:

- The working way follow the classic rendering pipeline, so any person who knows the basic 3D graphics theoretical concepts can understand the way the library classes are used and how to obtain results from just the first moment.

- The objects provided by library kernel are easy to understand[6]; this way, it provides vtkActor, vtkLight, vtkCamera, vtkProperty, vtkTransform, vtkRenderer and vtkRenderWindow among others. Without knowing anything about VTK, the reader can already imagine what they make and how they are linked together to obtain the results (remember the pipeline).

- It provides high performance characteristics such as level of detail actors, implicit modelling, hierarchical assemblies, real-time interaction, stereography, etc.

- It is extensible so a big number of classes are contributed by different people[7], so the package is always on development, earning more and more functionality.

- There are image, geometry and scene importers and/or exporters available for the most extended formats.

[6] This is absolutely true, what is also true is that to get high proficiency in VTK some practice is required, although the learning curve is not too pending.

[7] Or companies, in fact General Electric has contributed code to VTK, mainly medical data treatment algorithms; the only con is that this code is patented and it is necessary a license to employ it.

6. The digital processing plug-in

Once the tool structure and employed technologies have been introduced it's easy to understand the way any kind of external application can be integrated as a plug-in. The first of such plug-in's was a digital image processing library (developed at the same time the described environment was built).

This library should provide a series of image processing methods, from noise reduction to segmentation; all this functionality has to be based on a very simple image structure (each "image" would consist in a state stack where each state would be composed of a series of layers) and a reduced number of methods for its handling (to alloc/free memory, put/get data into this memory space).

An essential requirement for the library was its independence, although its first mission would be to be embodied as a plug-in for another tool this didn't should in anyway condition its characteristics; the code that would implement all the functionality shouldn't know other tools existence.

This way, the main part of the library was implemented as an aseptic C++ class hierarchy, requiring two later steps to "wrap" it in such a way that it was accepted by the main tool structure and it was easy to use.

First of this steps was the generation of a full series of C functions to invoke C++ library methods, this functions would constitute the binary code for the commands in a Tcl extension. Thank to this process, the digital image processing library was already able to be loaded by the tool.

Second step forced to write a GUI in Tk to allow the use of the previous Tcl commands that invoke the C++ library methods.

At this point we yet have a plug-in for digital image processing; the described procedure, employee for the library conversion into an external module can be equally followed to obtain extensions from legacy code.

In figure 4 it is shown the final structure for the set of tools: an opern-source visualization library like VTK integrated inside an application kernel and a self-made library that works as a plug-in that communicates with this kernel.

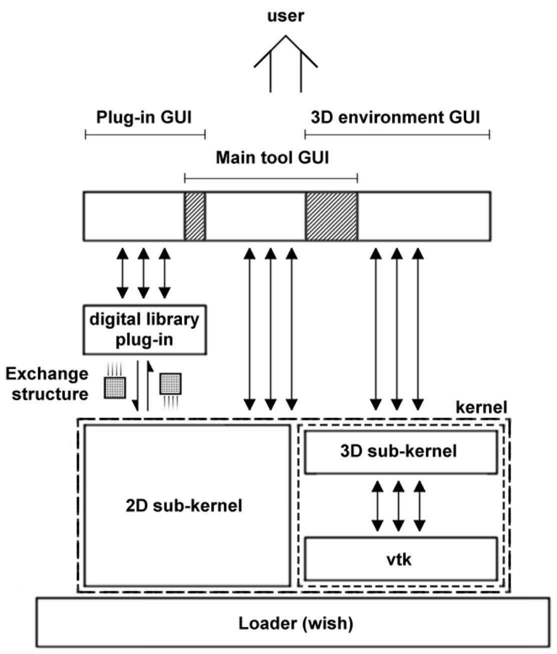
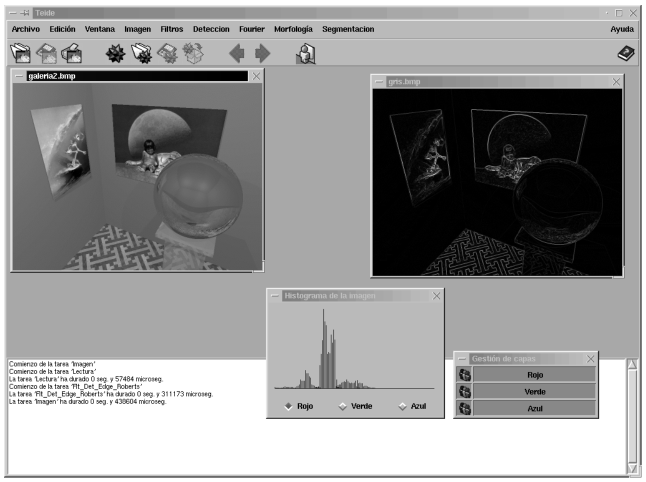


Figure 4: Digital library an VTK integration.

5. Final results and conclusions

While writting this paper the described tool development has already finished; the goals outlined for it have been fully reached and we can say that this approach has not only been a success but rather it is one of the few that can guarantee same results in a reasonably short lapse of time.

As a sample of the finished project, in figures 5 and 6 some tool snapshots are shown. First one shows the tool's image treatment capacities thank to the digital image processing plug-in. In figure 6 it is shown the 3D environment, with a sample of basic primitives.



```
Comienzo de la tarea 'Imagen'
Comienzo de la tarea 'Lectura'
La tarea 'Lectura' ha durado 0 seg. y 57484 microseg.
Comienzo de la tarea 'Flt_Det_Edge_Roberts'
La tarea 'Flt_Det_Edge_Roberts' ha durado 0 seg. y 311173 microseg.
La tarea 'Imagen' ha durado 0 seg. y 438604 microseg.
```

Figure 5: Tool snapshot showing 2D environment.

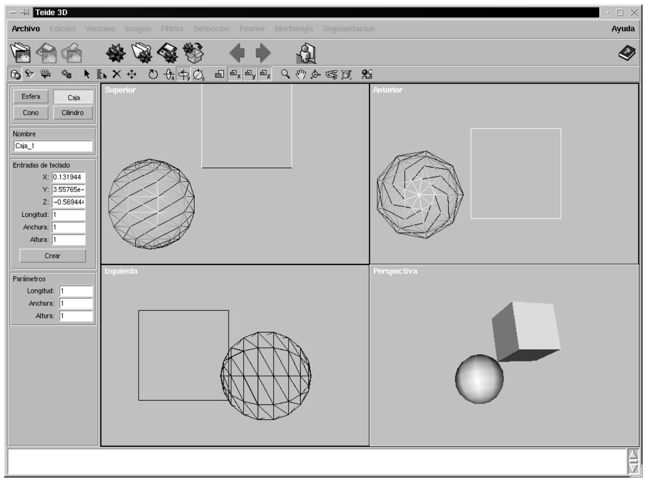


Figure 6: Tool snapshot showing 3D environment.

Just to finish, the described tool cannot be considered in any way as a final milestone but as a first step for future development. Perhaps such development will be based on this tool or not, but what is a true fact is that the technology which allow the building of multipurpose, high-performance, easily customizable and extensible, graphic tools is available, allowing such tools quick graphic applications development that, as feed-back, will contribute to the improvement of such utilities.

References

Laird, C.; Soraiz, K.; "Choosing a scripting language". http://www.sunworld.com/swol–10–1997/swol–10–scripting.html, 1999.

Harrison, M.; McLennan, M.; "Effective Tcl/Tk Programming." Addison-Wesley, 1998.

Welch, B.B.; "Practical Programming in Tcl & Tk." Prentice Hall PTR, 1997.

Schroeder, W.J.; Martin, K.M.; "The vtk User's Guide." Kitware, Inc., 1999.