Principles of Automatic Generation And Parsing of Device-Independent Metafiles by the Example of Object-Oriented VTK Library Adaptation

Dmitri Manakov, Ivan Komarovsky
Institute of Mathematics and Mechanics,
Ural Branch of the
Russian Academy of Sciences,
Ural State University, Yekaterinburg, Russia
manakov@imm.uran.ru

Abstract

This paper is about problems of implementing of parallel graphics using open source libraries. Automatic generation and parsing is discussed along with different approaches of its realization.

Keywords: *On-line Visualization, Metafile, Strategy.*

1. INTRODUCTION

One of the most important problems of modern graphics is on-line visualization of parallel computing. While working on the parallel machines, with restricted ability of data transfer via networks with low throughput, image rendering in-place is neither efficient nor profitable. But such kind of interactive graphics is needed when building visual debugger or at the beginning of scientific modeling, where user should interact with running program. Fairly often under these circumstances device-independent protocols, or metafiles are used [1].

Some kind of graphic library works on the parallel processors; all of the library function calls are redirected to the metafile, which is parsed by a powerful workstation, that makes real library calls with data taken from the file, thus rendering the resulting image. For an object-oriented library it may be said that an ordinary program running on one of the parallel processors uses proxy-objects instead of real ones.

When logging a call to a library method, we use the notion of named functions: each function has a number associated with it, which is passed to the metafile as well as its arguments. Since most of the graphic functions are procedures (i.e. they have no return value), we may use buffering which implies that we accumulate calls and send them to the file by big packets of data. Since we use the original function signatures, programs do not change at all (compared to their serial variant).

One of the evident drawbacks of such approach is that building this proxy-library is tedious and error-prone job to do, because typical graphical system has hundreds of routines. We offer to use an automatic generator that looks through the code of graphic library and builds all necessary proxy-objects as well as the program needed to perform metafile parsing.

2. AUTO-GENERATION GUIDELINES

2.1 Interaction scheme

Auto-generation assumes special “proxy-classes” (or meta-classes) creation based on input source code (either library header files or client source code) as well as building of a metafile parsing program. Every processor that uses library classes works with these proxy objects which substitute real method calls with metafile write operations (see Figure 1). The main CPU communicates with others and is responsible for metafile writing.

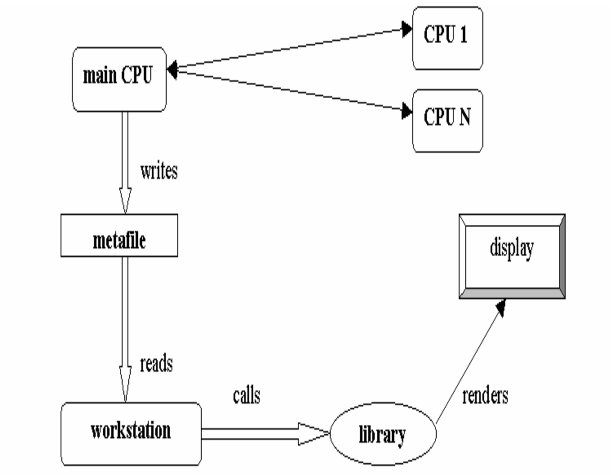

Figure 1: System in action.

2.2 General rules of auto-generation

1. Source code of a graphical task running on one of the parallel CPUs should be identical to that of serial variant.
2. Ability to choose appropriate data exchange protocol (via files or TCP/IP).
3. Wide use of buffering for minimum transfer overheads.
4. Separation of functions into different categories:
   - some functions imply coordination of concurrently running processes (like vtkRenderWindow::Render());
   - some need their return value immediately (so we can’t buffer them);
   - all kinds of functions are split into categories, which are implemented using different strategies.
5. Main graphical data is placed in workstation and parallel machine has only auxiliary data.
6. Usage of local variables duplicating principle: some of the most frequently data we cache on the parallel machine (for example, vtkPoints).

3. STRATEGIES

When implementing a suspended library function call, we should take into account its semantics, i.e. we can't just write it to the file (though in general it works). Every object method has a corresponding strategy of implementation, which reflects its functionality. Though generator may offer some kind of general strategy, it is the user who is responsible for making the right choice.

We offer the following strategies [1]:

- general (default choice, no other actions needed after writing to the metafile);
- for returning value functions;
- for coordinating functions;
- duplicating of local variables.

4. IMPLEMENTATION AND ISSUES

The most important part of generator is its database of proxy objects. In order to build it, we need some information about library classes and their methods. This information may come in two ways: from header files of the graphical library or from client's source code. In the former case we analyze the library entirely, building proxy-classes for all classes at once. The other way is to build our database gradually, i.e. update it when encounter new methods (not implemented yet).

Header files parsing has one important advantage: in this case we have access to methods' signatures. On the contrary, client's code contains only methods calls with actual arguments. In C++ that means that we can't figure out types of function's arguments even if we build symbol tables (due to hidden type conversions). We can only make assumptions about them. However, since typical library contains hundreds of functions but only small part of them is used most frequently, we may do extra work that we'll probably never need. If we build the database step by step (parsing user's source code) we generate proxies only for really useful methods. But in this case we have to ask user for signatures (due to problems mentioned above).

What is clear now is that we can't parse only a part of C++ grammar, because this will lead to frequent parsing mistakes (though client's code is okay and compiles well). Tools such as lex/yacc will ease a lot building of such a parser.

5. METHOD CALLS THROUGH A METAFILE

The structure of file records is rather simple: it contains the identification number of the given object's class, the ordinal number of the method and its arguments. If call is non-static, we put the ordinal number of object of the given class. It may also have the rank of calling processor (i.e. its id number). When actual parameter of the method is an object (read: polymorphic pointer to it), we substitute it with the object index. Every library class on the workstation has a list of indices that help the parsing program to recognize different objects.

The proxy-class database may maintain global class index which means that class' index doesn't change once assigned; when encounter new class, we give next free index. Another way is to assign indices for every input program, which allows to create array of them and gain access to the given class instantly (by the index read from metafile). The same is true for methods.

The parsing program is nothing else but a big loop (till the end of file). Following the structure of the file record, we should:

- read classIndex;
- read methodIndex;
- having both, we know what to do since these two numbers identify needed library call;
- now we read parameters of this particular method and call real library function providing actual arguments we've just read.

The obvious way to do it is to have a pair of nested switch-case operators, the outer one for classIndex and the inner for methodIndex. Remind that such operator is translated into a sequence of serial if/else checks, this approach may be rather ineffective (as time complexity of recognizing the class and the method is O(N*M), where N is the number of used classes and M is the maximum of used methods inside all classes). Especially when done in big loop. This is where local indexation is useful: before entering the parsing loop, we build two nested arrays. The inner is array of pointers for reading procedures (that know everything about corresponding library functions and read parameters from the file), it is indexed by methods ordinal numbers. The outer is indexed by class ordinal numbers and through given index gives access to the appropriate methods array described above. This structure eats memory but makes call time constant (more exactly, the time for figuring out what to call).

6. CONCLUSION

Auto-generation is well worth implementing; those who tried to adapt graphical programs to work in concurrent environment manually (even the simplest ones) have no doubts about it. Ideally, such code-generation program should provide convenient GUI and minimize the need for user's interaction with its parsing/generating part. Acting as a front-end, it should also ease the process of starting the program on parallel machine (by creating, for example, make-files and batch startup files).

The most important (and the hardest) part of such generator is strategy scheme implementation. In order to have code of the generator unchanged, we need flexible mechanism for adding new strategies, because none is able to foresee all the needed ones and hardcode them.

7. REFERENCES

[1] D. Manakov, M. Shagubakov. Adaptive Builder for Interactive Tasks in Mass-Parallel Machines // Proceedings of Graphicon 2002, Nijniy Novgorod, 2002, pp. 405-408. (In Russian)

About the author

Dmitri Manakov is a lead programmer at Institute of Mathematics and Mechanics,
Yekaterinburg, Russia.
His e-mail is manakov@imm.uran.ru

Ivan Komarovsky is a student at Ural State University,
Yekaterinburg, Russia