softinex, inlib, exlib, ourex, ioda, g4view, g4exa, wall

Guy Barrand

LAL, Univ Paris-Sud, IN2P3/CNRS, Orsay, France

E-mail: `barrand@lal.in2p3.fr`

Abstract. Softinex names a software environment targeted to data analysis and visualization. It covers the C++ inlib and exlib "header only" libraries that permit, through GL-ES and a maximum of common code, to build applications deliverable on the AppleStore (iOS), GooglePlay (Android), traditional laptops/desktops under MacOSX, Linux and Windows, but also deliverable as a web service able to display in various web browsers compatible with WebGL. In this paper we explain the coarse graining ideas, choices, code organization of softinex along a short presentation of some applications done so fare (ioda, g4view, etc...). At end we present the "wall" programs that permit to visualize HEP data (plots, geometries, events) on a large display surface done with an assembly of screens driven by a set of computers. The web portal for softinex is `http://softinex.lal.in2p3.fr`.

1. 2010

The year 2010 had been the year of the tablet breakthrough but also of the availability of "stores" for them that permit to install an application with fingertips. At LAL, it had been also the year that Apple, in the context of an ARTS project, provided us with a set of four computers and eight screens in order to explore "wall of screens" techniques. Being involved for long in graphics for science we can't ignore these three facts. We took these technological revolutions as an opportunity to reconsider strongly the way we did graphics up so fare and took the occasion to build something new. Before 2010 we relied strongly on third party software as coin3d for the graphics and gtk, Qt, etc... for the user interface. On iOS, Android only GL-ES is here, and not the "full classical OpenGL", then we can't stay with coin3d anymore. We had decided to create our own scene graph scene manager based on GL-ES and on our strong knowledge of the internal of OpenInventor (and coin3d). For the GUI, we had decided to attempt something orginial : do it also with GL-ES and the same scene graph manager than the one used to represent data. Doing the GUI with GL is not so new, a lot of people, especially around gaming, do that. It simplifies a lot of things, espcially when porting on new devices. Moreover the fact to build the GUI with the same scene graph manager than the "graphics itself" permits to embarque much less code for a whole application. (GUI toolkit such as Qt, gtk duplicates the logic and code found also in high level graphics library as Inventor). The way to build applications promoted by Apple and Google for their devices is too different in order for one man alone to follow them easily. The languages are different (java for Google/Android, Objective-C for Apple/iOS), the build systems are different (Eclipse or "Android SDK make" for Google and Xcode for Apple) and obviously the GUI toolkits are different. But it appears that both environments permit to do C++ and GL-ES, and it is something that "one man with academic resources" can use to build applications for both kind of devices.

2. inlib/sg

The core of our approach is the "inlib/sg" which is a scene graph manager. It is used both for building representations of data but also to handle the GUI of our applications. In a GUI toolkit it is also needed to have something to say "I want a button here, a menu here, a list here, etc..." and this is nothing more than building a scene. And using a scene graph logic looks adapted for that. What singularizes a GUI is the fact to have to deposit callbacks on elements of the scene (typically push buttons), but this is not strange to a scene manager logic used to visualize data since we need that here too to create animations. Then someone can definitely arrange a scene graph manager to create push buttons beside visualizing scientific data. The inlib/sg handles nodes (inlib::sg::node) that can be gathered in container nodes as group or separator (same logic as Inventor). Some nodes permit to deposit points, lines, triangles (as inlib::sg::vertices) and some permits to specify transformations (as inlib::sg::matrix). An action (inlib::sg::action), similar to the SoAction of Inventor, can be defined to "traverse a scene graph". A typicall action is the render action that permits to do the GL-ES of a scene graph.

3. inlib, header only code

One difficulty is the way to build the applications for the various systems. Today there is no universal IDE, building system that permits to build straight a C++ application for Android, iOS and the various desktop/latop systems (UNIX, Windows based) and this up to the deposition on the stores. If relying to an environment based on numerous libs, we would have to pass our time in various IDEs (Xcode, Eclipse) to declare these libs and what must go in them; a pain. To avoid that we do a maximum of "header only" code. It permits to bypass this problem. A penalty is obviously the compilation time of the whole application, but experience shows that building for exemple, by using clang in debug mode on a "pretty good machine" is bearable. The app building in optimization mode is more lengthy but we do that in general once when delivering for the stores. (This "header only" approach is not new, good part of the STL and the boost library is done like that).

4. inlib, a simple example

In inlib, we have also some code to handle histograms. To illustrate, you can put the below code in a test.cpp file :

```
#include <inlib/histo/h1d>
#include <inlib/random>
#include <iostream>
int main(int,char**) {
  inlib::random::gauss rg(1,2);
  inlib::histo::h1d h("Gauss",100,-5,5);
  for(unsigned int count=0;count<1000000;count++) h.fill(rg.shoot());
  std::cout << " mean " << h.mean() << ", rms " << h.rms() << std::endl;
  return 0;
}
```

And this program can be compiled and run simply with :

```
    Linux> g++ -I<path to inlib> test.cpp
    Linux> ./a.out
```

The inlib starts to be rich now. It contains code for fitting (pure header version of MINUIT), but also pure header code to write histograms and ntuples at the ROOT format (yes,yes). There is a lot of code for plotting (inlib::sg::plotter class), and as the inlib contains also the render_zb action able to render a scene by using an inlib z-buffer, you can also do batch plotting in 3D without the need to tie to any GL library. Various examples can be found under the inlib/examples/cpp directory. (In particular the lego_zb.cpp example for 3D batch plotting).

5. exlib
Another painfull problem in software is the handling of external librariries and the code that attach to them. We have choosen to put this kind of code under a separate namespace and packaging : the exlib. Here can be found pure header code that does something with expat, jpeg, png, cfitsio, freetype2, etc... The general rule is that the code in inlib is only on the STL and ANSI C libraries and that someone is expected to have to deal with some external software if having to use something from the exlib. Obviously the exlib depends on the inlib and then the STL and ANSI C library. Under exlib/examples/cpp, or exlib/apps there are various examples as the plotter_X11.cpp that demonstrates inlib histogram plotting by using X11. Contrary to an inlib example, a build script is rather mendatory since you have to pass all the "-I" and "-L" flags in order to use some external packages. Our build scripts are done in bash and we have tried to keep them readable. A build script to build an exlib example or an application uses a set of "use_[external]" scripts found under exlib/mgr, each dedicated to set flags to attach a given external package (for example use_X11 for X11). If having problem to compile one of the example because of "something not found" for a given external package, it is sufficient to check/change the flags found in the related use script. (For example, plotter_X11 needs exlib/mgr/use_[freetype,GL,GLX,X11] and the inlib/mgr/use_cpp to set the compiler).

6. ourex, master the externals, avoid "code inflation"
Beside the STL, it is hard to build a consequent application without some code not written at home. We call these "external packages". In general we are interested in an external package because we need a piece of code with "high added value" on a given problem, for example reading a jpeg file, parsing an XML file, decompressing a file at gzip format, etc... Any problem that would need us a lot of time to rewrite the algorithms because these algorithms embed a strong expertise on the problem at hand. In softinex we try to master our externals. Under the ourex directory, we keep a copy of the externals we need, and we give priority to the usage of these instead of using the ones coming with the system or installable by other way (apt-get on some Linuxes, Macport on a Mac, etc...). It permits first to have the same overall code on all platforms and then be sure to have the same behaviour of the applications on all platforms. Moreover, since we arrange to build the ourex externals with the same bash build system than exlib examples and applications, and without using any "config stuff", it permits to have in general a straightforward "build and install" of a given application.

7. Software Least Action Principle
The softinex icon represents the Maupertuis, Fermat or Least Action principle. The least action principle is what guided us in our choices for doing software to do physics: to build an application from nothing, we go straight at the essential by choosing simplest solutions and tools and then avoid all what is not needed or unecessarily complicated. The least action principle does not mean that we do... nothing! but that we do things by minimizing the number of lines of code involved, knowing that we have various constraints to fulfil, as the portabibilty and the efficiency (obviously). It does not mean also that we attempt to rewrite everything. We rely on a lot of "externals", but we choose them carefully by checking that they address properly one particular problem at a time in the same spirit.

Following our "software least action principle" (the... SLAP!) we have adopted C++ as a programming language. Operating systems being done in C, it is natural to "stay close" to C, it helps a lot. But experience shows that for big software we need object orientation. Encapsulation, namespace, class, inheritance and virtuality, if used properly, help a lot to organize. Due to its large availability we have adopted C++, but we are definitely not "C++ extremists" and avoid to jump to too "compact code" that poison the readabilty. As much as possible we try to code "header only"; experience shows that it simplifies a lot... everything. We then avoid languages as java that induces to have to handle a third party virtual machine between the application and the machine; it complicates. In the same spirit we target the native processors and native operating systems and don't have as a primary target virtual machines; also, it complicates (but we use anyway virtual machines to do some port and test the code). (In fact we are pretty close to think that virtual machines are for lazzy people that master nothing in their software). In the same spirit we use bash shell scripts to build applications: make, GNUmake, cmake, scons, ant, maeven, Eclipse, Xcode, VisualStudio, etc..., all these "tools" do not really permit to "pass at once" everywhere, and at some point finish to be more in the way if seeking the portability. In the same spirit we also get rid of, for the GUI, various things as Xt/Motif, gtk, win32, Qt, Cocoa, UIKit, etc... (depsite that we have a strong experience with all these), we prefer a "unified graphics approach" through GL-ES and a common scene graph manager ; it clearly simplifies a lot! (even if it is at the price of a much less sexy look and feel for the moment).

Someone must not see the upper point of view as "against everything" but more as "tired of a lot of things"! A lot of things that finish to be in the way when seeking to do physics. As said, we have adopted a lot of "externals" (jpeg, png, zip, zlib, expat, freetype, graphviz, cfitsio, hdf5, etc...). For them, we embarque their code and do not rely on the ones that may be found on a system. It simplifies a lot an installation "from scratch", and permits also to have some guarantee that the comportement of the application is the same on all machines.

What is sure is that with our SLAP, right now, we have compact applications that run natively, and then effectively, on a broader choice of machines that would have been permitted with other "fundamental guiding choices", and be sure that we are very very happy with that.

(We do not hide that our choices came also by having been a little bit traumatized by the "inflation of code and complexity" observed around software for the LHC experiments. Now these software can be built only for one given platform : clone of Linux CERN lxplus. Even a so common UNIX as MacOSX is out of reach, then iOS and Android... Right now to install the binary of the LHCb Panoramix event display, someone have to download more than 30 Gigabytes of "various things" to visualize a bunch of volumes and tracks. This includes a full g++ compiler, some C++ interpreter, a full version of python, etc, etc, etc.... Put all together it is three times more than any operating system! It would be interesting to know what engineers at Apple, Google, Microsoft would think of that...)

8. Applications done with softinex

8.1. ioda

ioda should be read "IO-DA", for Input/Output and for Data Analysis. It is an application that permits to read files at various formats as FITS used in astronomy, DICOM used in medical, AIDA and ROOT used in HEP, JPG and PNG format to store images and FOG developed at CEA/Saclay (France) to describe the LHC/ATLAS geometry. ioda permits to browse these files and visualize some of their data. For AIDA and ROOT files, the histograms 1D, 2D, profiles 1D, 2D can be plotted. For JPG, PNG files, the image is visualized. For FITS files, the "HDUs" can be listed and their keys can be seen. If the HDU is an IMAGE_HDU type, ioda attempts to visualize it. If the HDU is a BINARY_TBL, ioda shows a description of the columns (name, type) and proposes to histogram and plot a selected column. Files at the FOG CEA/Saclay format permit to visualize LHC/ATLAS sub detectors. ioda is available on the AppStore since the 5th January 2011. It is an historical date for us. It is probably the first HEP application that can be delivered so easily in this way (at least the first one that can read CERN/ROOT files !). Some kind of giant step for us. ioda is available on GooglePlay and the ioda-release.apk is also available from the download area of the web pages.

8.2. g4view, g4exa

g4view is an application that permits to read geometry files at the Geant4 [1] GDML format describing particle physics detectors. It permits to visualize the geometry of a loaded detector. It can read also "scenarios" files that permit to choose the volumes seen from GDML, strongly customize their graphical attributes as the coloring or the wire frame/solid rendering. A scenario permits also to customize the "particle gun" by choosing, for exemple, the primary particle type and its momentum. A scenario permits also to setup the coloring of trajectories when shooting events. From the main menu, you can then start a "particle through matter" simulation of the detector by using the Geant4 toolkit. g4view can also read and execute g4mac files containing Geant4 scripting commands. Under the examples menu, there are predefined typical setups as a piece of an electromagnetic calorimeter intended to be used for particle physics detector teaching. The code is based over the Geant4 core and the inlib/exlib code. g4exa is a more simple Geant4 application done with the code of the Geant4 A01 example which is intented to be a template application for people wanting to use the softinex tools to write their own Geant4 application to run then on Android and iOS devices.

8.3. pmx,agora

These are "concept apps" of smartphone/tablet like event displays for the LHCb and ATLAS experiments. They are more a proof of concept than applications ready to do physics. Anyway it can give to someone novice in high energy physics (HEP) a glance at what a HEP detector looks like.

8.4. wall

The "wall" package contains code to steer a set (a "wall") of screens with multiple computers. See the web pages for photos and videos. Our today (2013) setup is made of four Macs driving each three screens (then twelve screens) and a iMac used to pilot the system, all these machines being connected on a fast private network. There is one process per screen (then an overall of twelve and then three per machine). There is a master process on the front-end machine (the iMac). Each screen-process receives from the master-process a same scene graph containing a camera node, some shape nodes, image nodes and matrix nodes to position objects in the scene. But the camera node (an inlib::sg::lrbt) of each screen-process is tuned to see only a twelfth of the projection of the scene. It is the "tiled rendering" way of doing. The master-process can send camera displacement and zoom orders to the screen-processes in a consistent way, which leads to nice animations. For some scenes, we can upload at once triangles, segments, points in the GPUs of the screen-machines, and this leads to very fluent animations, even on big geometries. This setup is commonly used at LAL to do outreach : HEP diaporama, showing interaction of common particles (electron, proton, muon) in a simple calorimeter (in fact done with the Geant4 novice/N03 example code), showing ATLAS and LHCb geometries, showing astronomical pictures (in fits files), showing DICOM medical files. In May 2013 we made a connection with the LHCb/Panoramix event display, which permits us to show now "true recent physics". We made the connection with apps running on tablets. Some applications as ioda, g4view running on a tablet can send scene graphs to the wall. This kind of setup is still a research and development platform, but what had been done up so fare convinced us that it has clearly a huge potential for helping doing physics.

Reference

[1] S.Agostinelli et al.,GEANT4-a simulation toolkit, Nucl. Instr. Meth. A, vol. 506, no. 3, pp. 250-303, 2003.