Python binding for Geant4 toolkit using Reflex / PyROOT tool

Witek Pokorski

EuroPython 2006, CERN, Geneva
04.07.2006

Outline

- Motivation
  - why do we need Python in Geant4
- Short introduction to Geant4
  - basic Geant4 applications
- Creating Python binding
  - the power of Reflex/PyROOT
- Some examples
- Conclusion

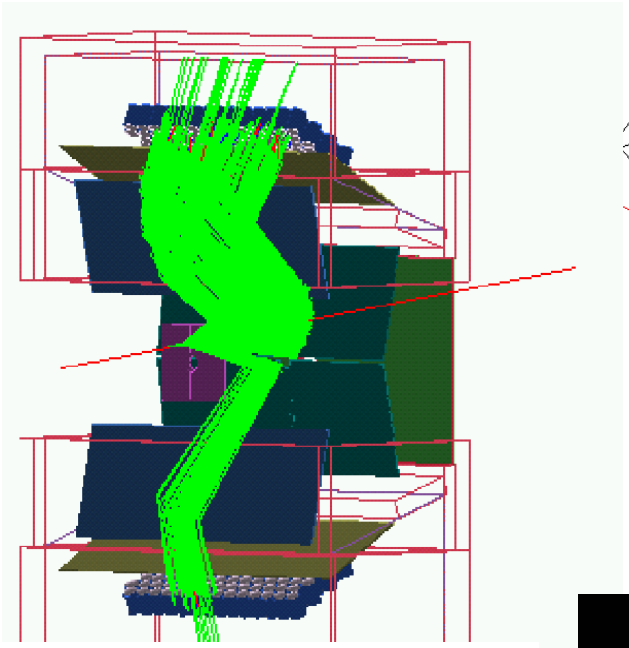
Motivation

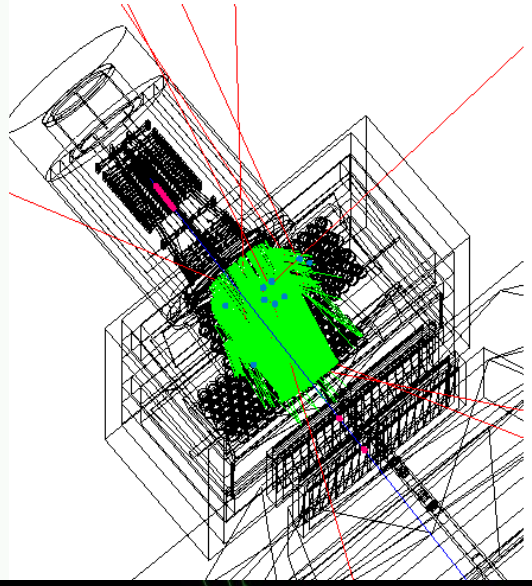
- to use Python for the construction of Geant4 simulation applications
  - to allow quick prototyping and testing
  - to add flexibility in the configuration
  - to make the application setup more homogenous
- to use Python shell as a powerful user interface for interactive running of Geant4
- to use Python as the 'glue' between simulation and analysis applications
  - to allow interactive simulation and analysis sessions
  - to facilitate debugging of the simulation setups

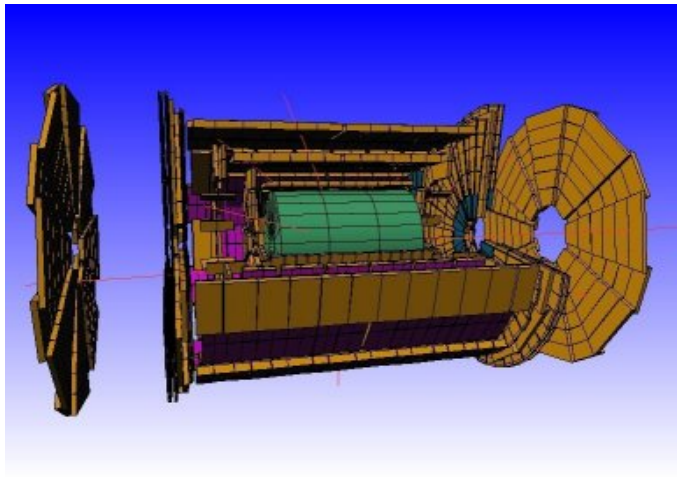
Introducing Geant4

- Geant4 - a toolkit for the simulation of the passage of particles through matter
  - implemented in C++
  - used in high energy, nuclear, accelerator physics, studies in medical and space science, etc
- complete range of functionality including tracking, geometry, physics models and hits
  - electromagnetic, hadronic and optical processes
  - energy range starting from (for some models) 250 eV and extending to the TeV energy range
- world-wide collaboration of scientists and software engineers
  - main users: LHC experiments, BaBar, Fermilab experiments, International Linear Collider, European Space Agency, Gamma Ray Large Area Space Telescope, medical physics, etc

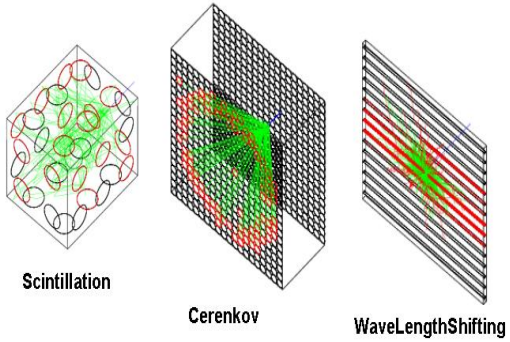
Geant4 applications

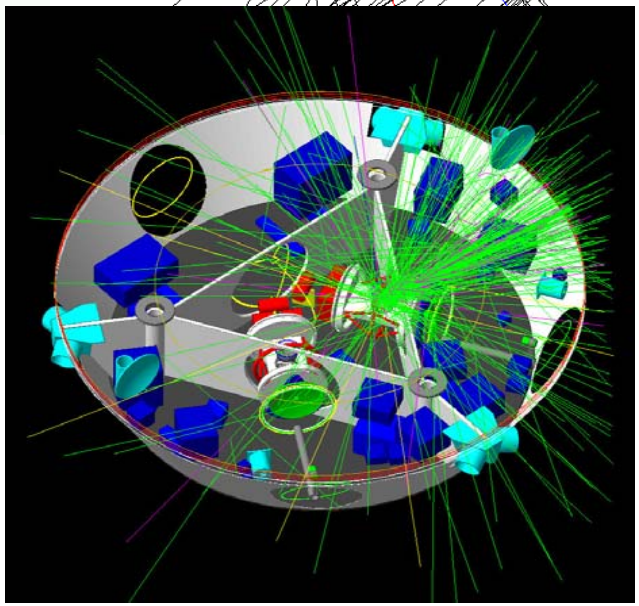


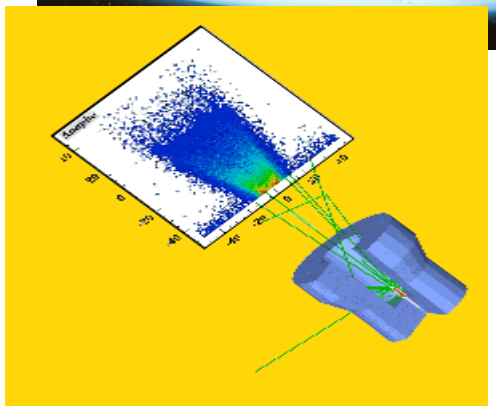




/examples/extended/optical/LXe

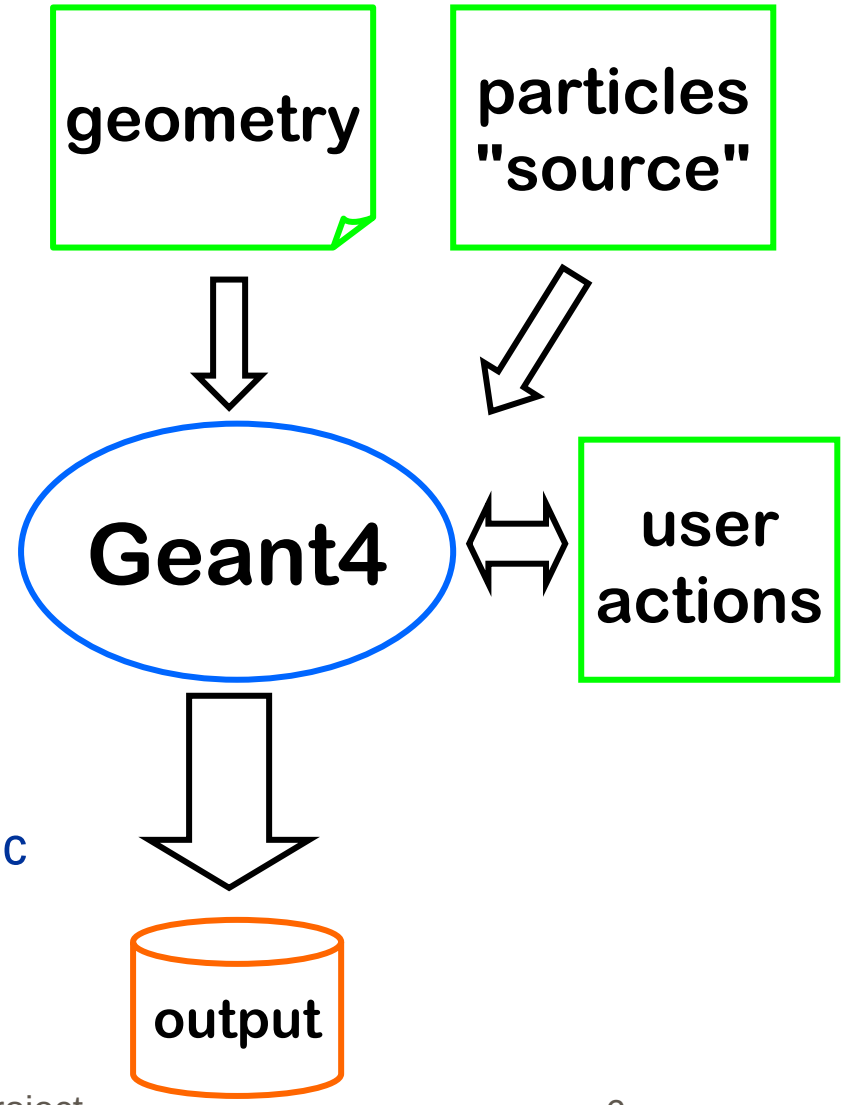






Running Geant4

- user's input:
  - experimental setup (geometry)
  - particle source
  - 'user actions'
  - 'sensitive detectors'
- simulation of interactions of particle according to physics processes modeled by the toolkit
- output generation
  - energy deposition (hits) in specific volumes



Geant4 simple example (main)


```
int main(int argc,char** argv)
{
  // run manager
  G4RunManager * runManager = new G4RunManager;

  // geometry and physics models
  runManager->SetUserInitialization(new ExN02DetectorConstruction);
  runManager->SetUserInitialization(new ExN02PhysicsList);

  // particle source and user actions
  runManager->SetUserAction(new ExN02PrimaryGeneratorAction);
  runManager->SetUserAction(new ExN02SteppingAction);

  //Initialize G4 kernel
  runManager->Initialize();

  //get the pointer to the User Interface manager
  G4UImanager * UI = G4UImanager::GetUIpointer();
  G4UIsession * session = new G4UIterminal();
  session->SessionStart();

  delete session;
  delete runManager;
}
```

Geant4-specific user interface

Geant4 simple example (geometry)


```
//------------------------------ a calorimeter block

 G4Box* calorimeterBlock_box = new G4Box("calBlock_box", 1.0*m, 50.0*cm, 50.0*cm);

 calorimeterBlock_log = new G4LogicalVolume(calorimeterBlock_box,
                                            Pb,"caloBlock_log",0,0,0);

 calorimeterBlock_phys = new G4PVPlacement(0, G4ThreeVector(1.0*m, 0.0*m, 0.0*m),
           calorimeterBlock_log,"caloBlock",experimentalHall_log,false,0);


 //------------------------------ calorimeter layers

 G4Box* calorimeterLayer_box = new G4Box("caloLayer_box", 1.*cm, 40.*cm, 40.*cm);
 calorimeterLayer_log = new G4LogicalVolume(calorimeterLayer_box,
                                            Al,"caloLayer_log",0,0,0);

 for(G4int i=0;i<19;i++) // loop for 19 layers
 {
   G4double caloPos_x = (i-9)*10.*cm;
   calorimeterLayer_phys = new G4PVPlacement(0,
              G4ThreeVector(caloPos_x, 0.0*m, 0.0*m),
              calorimeterLayer_log,"caloLayer",calorimeterBlock_log,false,i);
 }
```

Where can Python help

- top level configuration (geometry, etc) hardwired in the 'main' if implemented in C++
  - Python could make the configuration more flexible
  - Python would allow dynamic loading of geometry, etc
- Geant4 user interface requires dedicated implementation ('messenger' classes)
  - Python would provide a natural interface to Geant4 classes
  - powerful scripting language like Python would replace limited Geant4 UI
- using Python for configuration and interactivity adds homogeneity

Introducing Reflex

- Reflex - package developed at CERN to provide reflection capabilities for C++
  - part of ROOT framework
  - talk by P. Mato "Interfacing Python and C++ frameworks"
- reflection = ability of a programming language to introspect its data structure and interact with them at runtime without prior knowledge
- reflection mechanism can be used for dynamic Python binding of C++ classes
- Reflex features
  - non-intrusive, no code instrumentation required
  - reflection information is generated at the build time in form of dynamically loaded libraries (dictionaries)
    - 'selection' file used to specify the contents of the dictionary

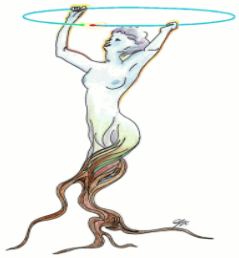
One word about ROOT

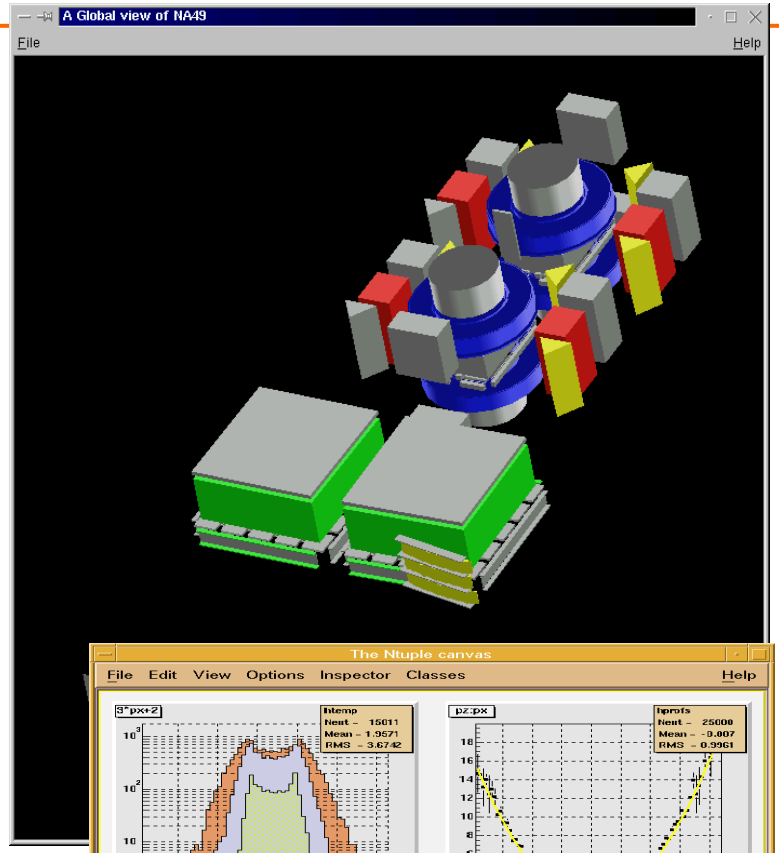
- ROOT - an object-oriented data analysis framework
- started in the context of NA49 CERN experiment by R.Brun and F.Rademakers
- some of the areas of application:
  - C interpreter
  - histograms and fitting
  - input/output (persistency for C++ objects)
  - 2D and 3D graphics
- recently added
  - Reflex
  - Python binding

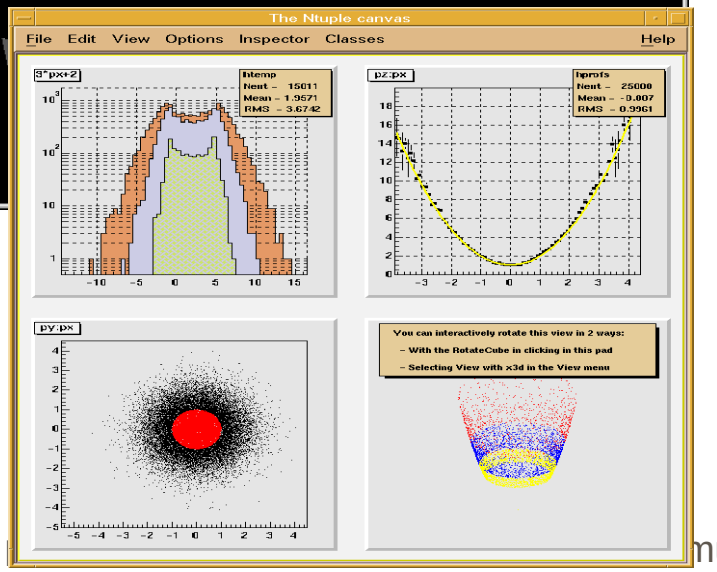
ROOT applications

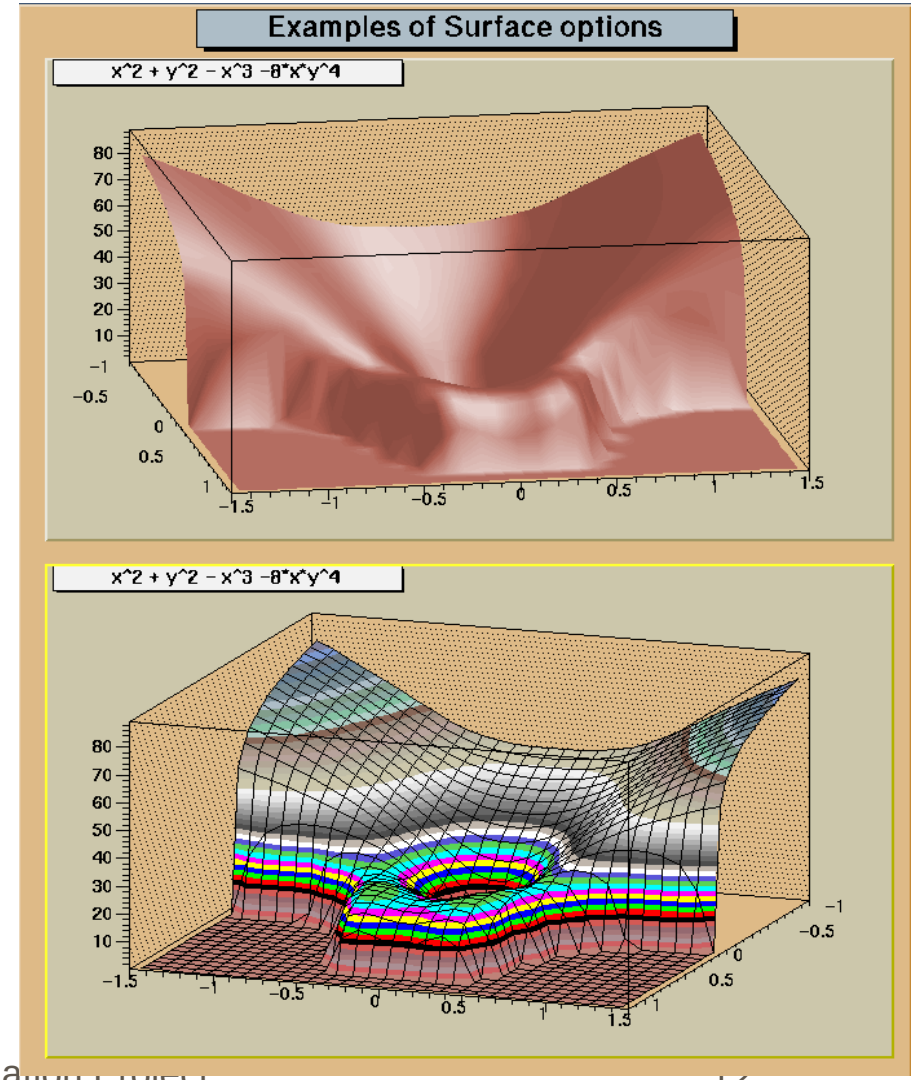
ROOT

An Object-Oriented
Data Analysis Framework









Using Reflex dictionary

- PyROOT - Python module originally developed as the Python binding for ROOT classes
- extended to provide Python binding for any C++ classes with the Reflex dictionary information
  - Python binding using Reflex/PyROOT does not require any instrumentation of the C++ code
  - no additional code required (like for BOOST-Python)
  - dictionary can be generated for any C++ class
- PyROOT provides very useful emulation (pythonization) of several C++ constructs
  - STL iterators, null pointers, templates

Dictionary generation for G4 classes

- dictionary generated at the build time
  - usually by additional target in the makefile
  - 'genreflex' tool part of ROOT distribution (uses gccxml)

```
genreflex Classes.h -s selection.xml -I${INCLUDES}
```

tool for dictionary generation

header files for classes to be processed

selection file

standard include directories flag

Selection file structure


```
<lcgdict>

<class name="G4RunManager"/>
<class name="G4EventManager"/>
<class name="G4TrackingManager"/>
<class name="G4VUserDetectorConstruction"/>
<class name="ExN02DetectorConstruction"/>

<exclusion>
  <class name="G4TrackingManager">
    <method name="SetNavigator"/>
  </class>
</exclusion>

</lcgdict>
```

mandatory starting tag

classes to create dictionary for

methods to be excluded from the dictionary

mandatory end tag

Geant4 simple example in Python

binding module

dictionary loading

Geant4 'main'

```
import ROOT
ROOT.Cintex.Enable()

ROOT.gSystem.Load('N02ExampleDict')

g4rm = ROOT.G4RunManager()

det = ROOT.ExN02DetectorConstruction()
g4rm.SetUserInitialization(det)

phy = ROOT.ExN02PhysicsList()
g4rm.SetUserInitialization(phy)

pgen = ROOT.ExN02PrimaryGenerator(det)
g4rm.SetUserAction(pgen)

evact = ROOT.ExN02EventAction()
g4rm.SetUserAction(evact)

g4rm.Initialize()
g4rm.SetVerboseLevel(1)

g4rm.BeamOn(1)
```

will not be needed in the future

can be automatic

- only the 'main' is 'pythonized'
- dictionary created for all the 'user' C++ classes (detector construction, physics list, etc)
- Python used to put different 'blocks' together
  - no performance penalty!
- no need to use Geant4 user interface
  - configuration done in Python

Further 'pythonization'

- having the dictionary for all the Geant4 geometry classes generated, one can define the detector geometry in Python
  - geometry tree and materials definitions can be done interactively or dynamically loaded from a Python script
  - Python used only for putting the volumes together, the geometry tree remains all C++
  - no performance penalty
    - C++ object interact directly with each other, no interaction via Python
- converters can be easily implemented to import geometries from other formats like GDML (XML)
- Python for the physics configuration (physics list)
  - specific physics models can be switched on/off and configured from Python
  - physics parameters (production cuts, etc) can be interactively adjusted

PyROOT specific features (1/2)

- null pointers

C++	PyROOT
Class* ptr = 0;	ptr = None

- lifetime of objects and memory management

C++	PyROOT
A(new B)	~~A(B( ))~~
A(new B)	b = B( ) A(b)

B would got deleted just after returning from A(...)

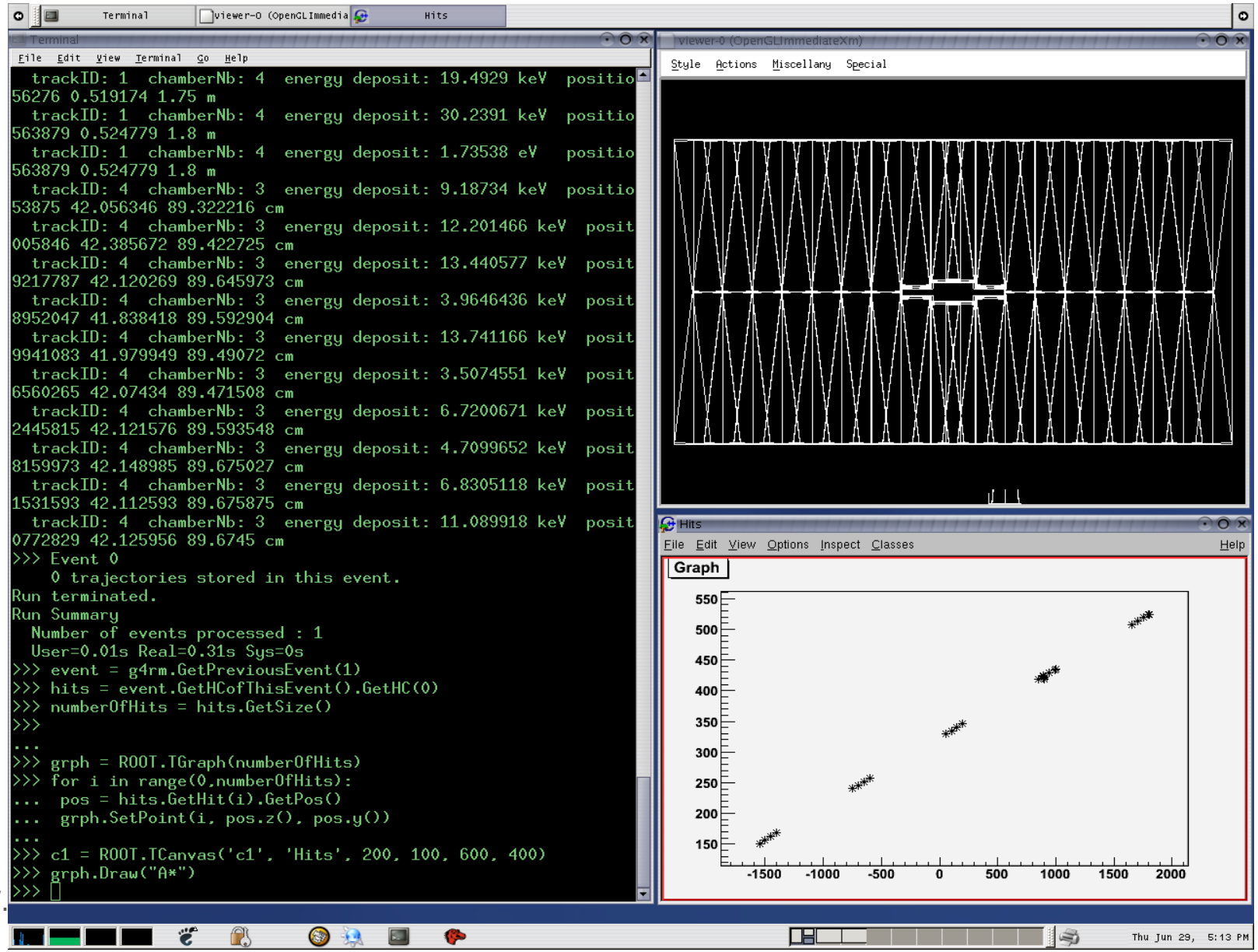
PyROOT specific features (2/2)

- objects ownership
  - possibility of changing the ownership policy
    - objects can be own by PyROOT or by Geant4
- possibility of using Python iterators on STL containers
  - 'for x in X' can be used for X being std::vector
- primitive types automatically converted
- mapping of Python arrays to C++ arrays

Interactive running with ROOT

- PyROOT module contains Python binding for all ROOT classes
  - comes for free when loading binding for Geant4
- ROOT can be run interactively together with the simulation application
  - Python naturally 'glues' different applications together
- examples use cases
  - histograms
  - events persistency
  - geometry persistency
  - 3D graphics (detector visulalization)

Running Geant4 and ROOT interactively



Some remarks

- since the generation of the dictionary is not intrusive, it could be added to the standard Geant4 procedure
  - users would immediately have the Python binding
  - persistency of Geant4 objects would be possible
    - ROOT I/O could be used for detector geometry persistency

Conclusions

- Python bindings are starting to be common practice for scientific software
  - playing the role of a 'software bus' to connect different applications together
  - used as very powerful scripting language for configuration
- with Reflex/PyROOT, Python binding for Geant4 comes for free
  - Reflex dictionary used also for Geant4 objects persistency
- no performance penalty
  - Python only used for putting 'blocks' together and for setting options
- improves modularization of Geant4 applications
- provides natural support for dynamic loading of components
- improves interconnectivity with other applications