First discussion on repackaging Geant and ROOT libraries

Pere Mato

SFT group meeting, 16th June 2013

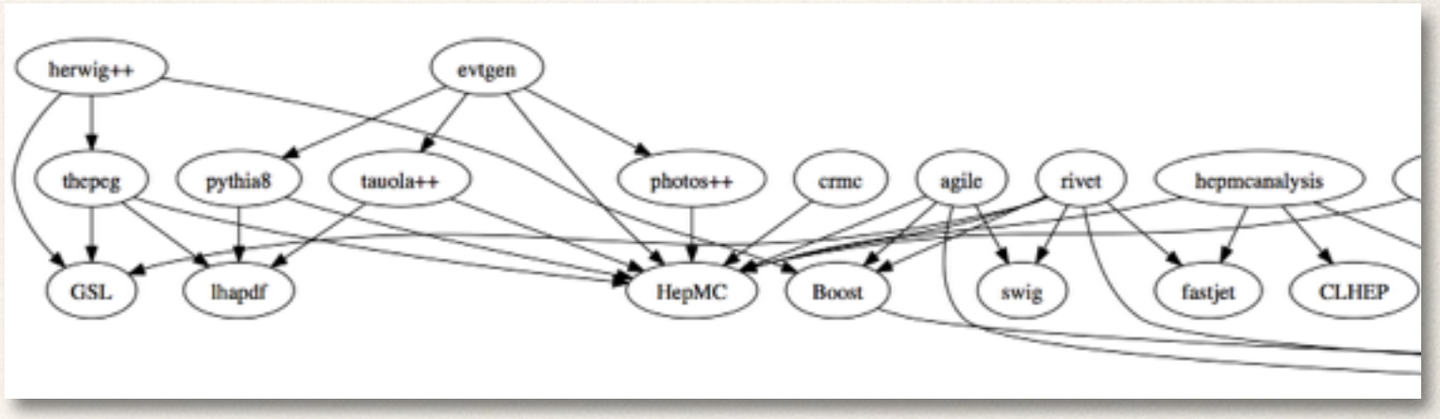
Foreword

- Will be using ROOT as example but is also applicable to Geant4
- Personal ideas with the simple purpose to trigger a discussion

ROOT
Data Analysis Framework

Motivation: Managing Dependencies

- Very few packages are truly standalone
  - Very often packages depend of other packages
- Package dependencies are difficult to manage
  - Complicates the configuration, the build process, the distribution and the deployment
- Avoiding dependencies is not a good solution in general
  - Adds code duplication
  - Reduces code re-use
- **Managing dependencies is essential**



Fragment of the dependency graph of some MC generator packages

Motivation: Modular ROOT^2

- ROOT core team should define a **model** for defining **modules** with all sort of functionalities that be easily integrated
  - Provides the 'platform' with a number of core modules with the basic functionality
  - C++ introspection, modular build system, plugin system, flexible testing system are some of the needed ingredients

- Essential for making ROOT more open and encourage external people to contribute
  - Authors may keep ownership of their modules and provides user support (e.g. bug fixes, documentation)
  - Core team ensures consistency and provides support for the full life-cycle
- Support addition and build of a package (with one or more modules) in a later stage
  - Like is currently done in R

ROOT plans: SFT meeting in January

ROOT current situation

package	internal	external
X11/cocoa/win32gdk		x
Gif/Tiff/Png/Jpeg		x
zlib	x	x
freetype	x	x
afterimage	x	x
pcre	x	x
lzma	x	x
FTgl	x	x
GLEW	x	x
GSL	(x)	x
Python		x
OpenGL		x
Graphviz		x
xrootd	(x)	x
llvm/clang	x	
vc	x	
vdt	x	

package	external
Qt3/Qt4	x
Kerberos5	x
LibXml2	x
OpenSSL	x
ldap	x
rfio/Castor	x
MySQL/Oracle/Odbc	x
PostgreSQL/SQLite	x
Pythia6/Pythia8	x
FFTW3	x
cFITSIO	x
gfal/dcache/globus/davix	x
monalisa/alien/….	x

(x) sources downloaded at build time

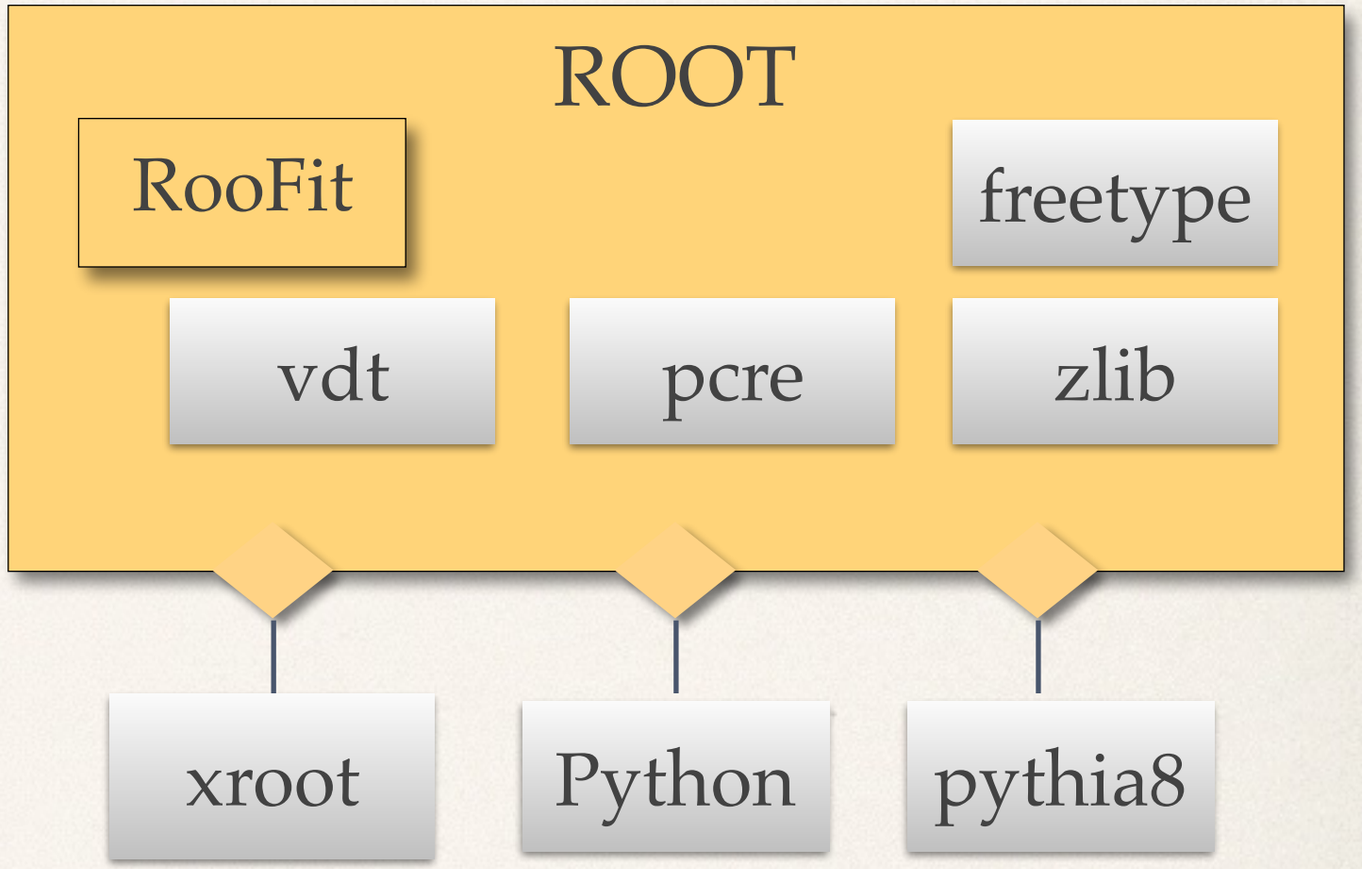
ROOT
Data Analysis Framework

Main Use Cases

- 1 - ROOT standalone with a number of ‘options’ enabled
  - Typically end-users doing analysis with ROOT
  - Building ROOT from sources should be simple (single command)
    - Automatically include all the ‘core’ dependent packages on the same build
    - External package versions selected by ROOT team
    - For the ‘options’ may need to provide external builds
- 2 - ROOT integrated as part of a larger software stack
  - Experiment’s applications
  - Typically all dependent **packages are externally provided**
    - Only way to ensure consistency
    - Versions selected by experiment librarians (within constrains)

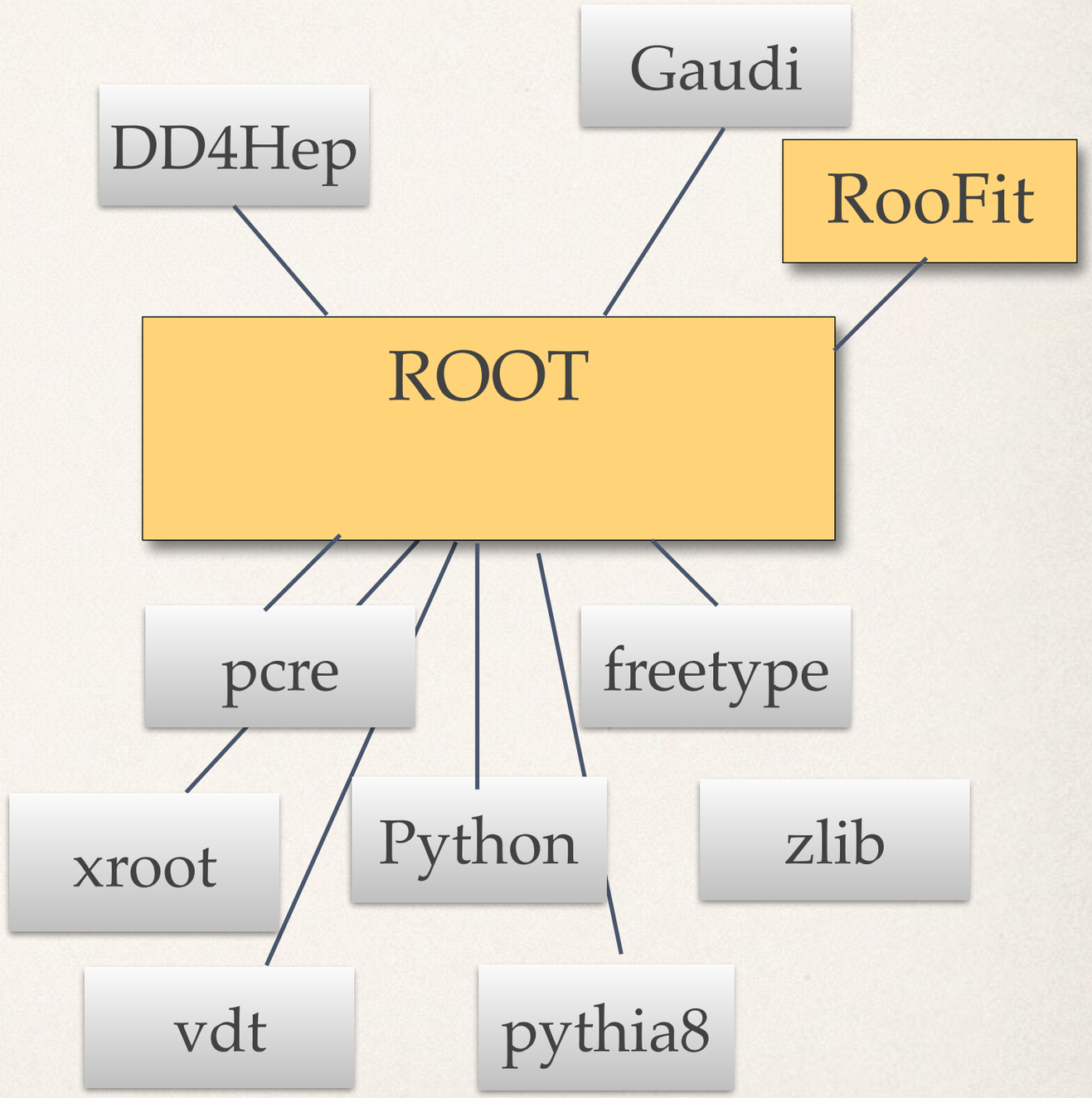
ROOT Standalone

- Typically the internal libraries will be statically linked to avoid symbol leakage
- The configure step will try locate external libraries present in the system



ROOT Integrated

- Every dependency should be configurable
  - explicit versions and locations
  - version compatibility rules
- Build ROOT on top a managed and pre-built stack
- Chunks of ROOT can be made available as layered components
  - version compatibility rules



ROOT
Data Analysis Framework

CMake at Rescue

ExternalProject: Create custom targets to build projects in external trees

The 'ExternalProject_Add' function creates a custom target to drive download, update/patch, external project:

```
ExternalProject_Add(<name>     # Name for custom target
  [DEPENDS projects...]        # Targets on which the project depends
  [PREFIX dir]                 # Root dir for entire project
  [LIST_SEPARATOR sep]         # Sep to be replaced by ; in cmd lines
  [TMP_DIR dir]                # Directory to store temporary files
  [STAMP_DIR dir]
#--Download step----------
  [DOWNLOAD_NAME fname]
  [DOWNLOAD_DIR dir]
  [DOWNLOAD_COMMAND cmd...]
```

find_package: Load settings for an external project.

```
find_package(<package> [version] [EXACT] [QUIET] [MODULE]
             [REQUIRED] [[COMPONENTS] [components...]]
             [OPTIONAL_COMPONENTS components...]
             [NO_POLICY_SCOPE])
```

ROOT
Data Analysis Framework

Example: GSL in ROOT

```
#---Check for GSL library-------------------------------------------------------
if(mathmore)
  message(STATUS "Looking for GSL")
  if(NOT builtin_gsl)
    find_package(GSL 1.10)
    if(NOT GSL_FOUND)
      message(STATUS "GSL not found. Set variable GSL_DIR to point to your GSL installation")
      message(STATUS "               Alternatively, enable the option ‘builtin_gsl’")
      message(STATUS “               to build the GSL libraries internally")
      message(STATUS "               For the time being switching OFF 'mathmore' option")
      set(mathmore OFF CACHE BOOL "" FORCE)
    endif()
  else()
    set(gsl_version 1.15)
    message(STATUS "Downloading and building GSL version ${gsl_version}")
    ExternalProject_Add(
      GSL
      URL http://mirror.switch.ch/ftp/mirror/gnu/gsl/gsl-${gsl_version}.tar.gz
      INSTALL_DIR ${CMAKE_BINARY_DIR}
      CONFIGURE_COMMAND <SOURCE_DIR>/configure --prefix <INSTALL_DIR>
    )
    set(GSL_INCLUDE_DIR ${CMAKE_BINARY_DIR}/include)
    set(GSL_LIBRARIES -L${CMAKE_BINARY_DIR}/lib -lgsl -lgslcblas -lm)
  endif()
endif()
```

Building Complete Stacks

- A lot of experience with LCG releases
- A simple file describe all the versions of the packages
- For each package a simple set of instructions is sufficient

```
# Externals
LCG_external_package(4suite      1.0.2p1)
LCG_external_package(AIDA        3.2.1)
LCG_external_package(blas        20110419)
LCG_external_package(Boost       1.55.0)
LCG_external_package(CLHEP       2.1.4.1)
LCG_external_package(CLHEP       1.9.4.7)
LCG_external_package(cmake       2.8.9)
LCG_external_package(cmaketools 1.1)
LCG_external_package(cmt         v1r20p20090520)
LCG_external_package(coin3d      3.1.3p2)
LCG_external_package(coverage    3.5.2)
LCG_external_package(CppUnit     1.12.1_p1)
LCG_external_package(cx_oracle   5.1.1)
LCG_external_package(doxygen     1.8.2)
LCG_external_package(expat       2.0.1)
LCG_external_package(fastjet     3.0.6)
LCG_external_package(fftw        3.1.2)
LCG_external_package(Frontier_Client 2.8.10)
LCG_external_package(GCCXML      0.9.0_20131026)
LCG_external_package(genshi      0.6)
LCG_external_package(graphviz    2.28.0)
...
```

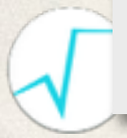
ROOT
Data Analysis Framework

Example

- Few lines are sufficient to describe the steps required for a given package
  - Dependencies to other packages are explicit
  - Variables such as ${XXX_home} point to the installation of package XXX

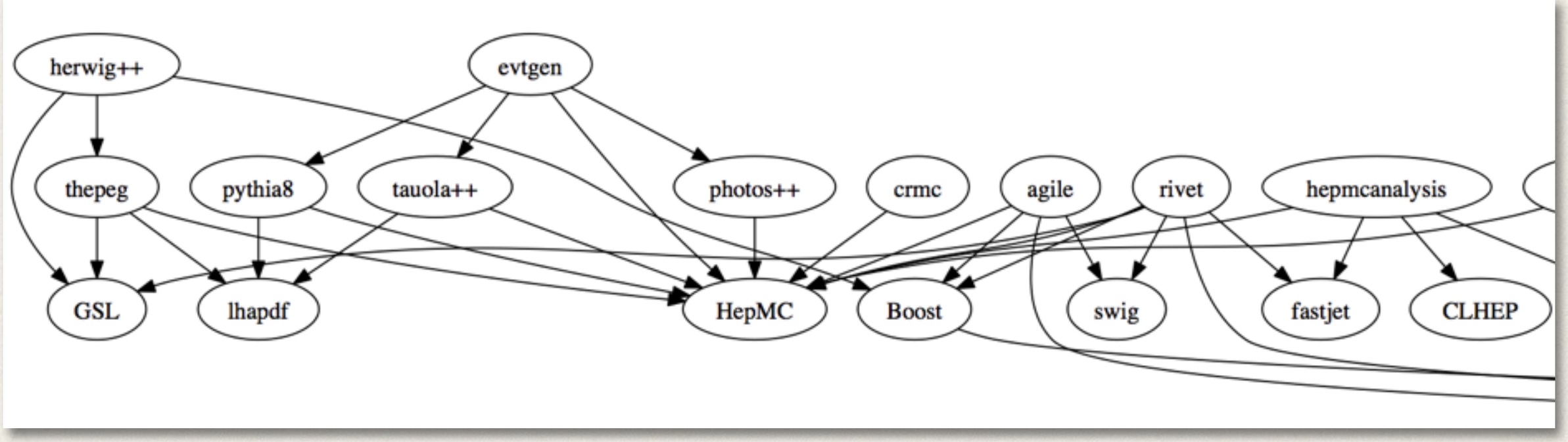
```
#---agile-----------------------------------------------------------------

LCGPackage_Add(
  agile
  URL http://www.hepforge.org/archive/agile/AGILe-${agile_native_version}.tar.bz2
  CONFIGURE_COMMAND ./configure --prefix=<INSTALL_DIR>
     --with-hepmc=${HepMC_home}
     --with-boost-incpath=${Boost_home_include}
     --with-lcgtag=${LCG_platform}
     PYTHON=${Python_home}/bin/python
     LD_LIBRARY_PATH=${Python_home}/lib:$ENV{LD_LIBRARY_PATH}
     SWIG=${swig_home}/bin/swig
  BUILD_COMMAND make all LD_LIBRARY_PATH=${Python_home}/lib:$ENV{LD_LIBRARY_PATH}
  INSTALL_COMMAND make install
     LD_LIBRARY_PATH=${Python_home}/lib:$ENV{LD_LIBRARY_PATH}
  BUILD_IN_SOURCE 1
  DEPENDS HepMC Boost Python swig
)
```



Package Dependencies

- From the dependencies we can generate dependency graphs
  - Useful for documentation
  - Full package dependency versions for binary compatibility



Build instructions are fairly simple

- get or setup **cmake**
- checkout **lcgcmake** package from SVN
- setup C/C++/Fortran compilers
- create workspace area
- configure with **cmake**
- build with **make**

1. On **lxplus** set PATH to use one of latest CMake versions (default is 2.6)
   `export PATH=/afs/cern.ch/sw/lcg/external/CMake/2.8.9/Linux-i386/bin:${PATH}`
2. Checkout the lcgcmake package from lcgsoft SVN repository
   `svn co svn+ssh://svn.cern.ch/reps/lcgsoft/trunk/lcgcmake`
3. Create a workspace area in which to perform the builds
   `mkdir lcgcmake-build`
   `cd lcgcmake-build`
4. You may need at this moment to define the compiler to use if different from the native compiler
   `source /afs/cern.ch/sw/lcg/external/gcc/version/platform/setup.(c)sh`
5. Configure the build of all externals with cmake
   `cmake -DCMAKE_INSTALL_PREFIX=../lcgcmake-install ../lcgcmake`
6. In order to build against the existing external repository use the option
   `-DLCG_INSTALL_PREFIX=/afs/cern.ch/sw/lcg/external`
   to tell the system to look for packages in the LCG area.
7. Build and install all external packages
   `make -j`
8. Or to build a single external package
   `make -j <package>` (use `make help` to see the list of all available packages)
9. You may need to restart de build of a package from beginning in case of obscure errors. The best is to clean a specific package
   `make clean-<package>`

ROOT Data Analysis Framework

http://sftweb.cern.ch/spi/HowtoBuildWithCMake

Conclusions

- What we are doing is basically correct
  - We will need to add additional packages (e.g. USolids, …)
- Can be improved and streamlined
  - The double role is good
  - Embedding inside is bad

package	internal	external
X11/cocoa/win32gdk		x
Gif/Tiff/Png/Jpeg		x
zlib	x	x
freetype	x	x
afterimage	x	x
pcre	x	x
lzma	x	x
FTgl	x	x
GLEW	x	x
GSL	(x)	x
Python		x
OpenGL		x
Graphviz		x
xrootd	(x)	x
llvm/clang	x	
vc	x	
vdt	x	

What Next

- Develop ‘standard’ patterns for:
  - Building the external together with the project
    - static linking if possible to make it invisible outside
    - better than ExternalProject (e.g. add_subdirectory)
    - copying or downloading the external?
  - Building the full stack of externals and configuring the project
    - standard variables/options for configuration
    - runtime environment setting (e.g. ROOTSYS would not be enough)
- Try the patterns for few libraries
  - E.g. vdt, USolids, etc.
  - correct the problems

ROOT
Data Analysis Framework