Wire Cell Software Overview (plus some data structure basics)

Brett Viren

Physics Department

BROOKHAVEN NATIONAL LABORATORY

BNLIF Wire Cell Team

2015 April 2

Outline

Repositories

All our Wire Cell source code is in the “BNLIF” GitHub organization.

The main repository is:

https://github.com/BNLIF/wire-cell

See full list or repositories at

https://github.com/BNLIF

- Feel free to commit, make new ones, add issues, etc.
- Not all repositories in BNLIF are for Wire Cell.

Package Names

A high-level naming convention is used:

source package `wire-cell-<name>` used to name source repositories.

source subdir `<name>` sub-directory location in a *build package* of a *source package* (more on this below).

binary package `WireCell<Name>` used in expressing **dependencies** between packages and to name the public API **header directory**.

The source package/subdir names are somewhat unimportant but the **binary package name** gets baked to a few places.

Namespaces

C++ `namespace`s:

`units` the **system of units** taken from CLHEP (more on this next).

`WireCell` all “core” C++ **library** code should be in this namespace. Don’t add redundant “WireCell” name to class/function names themselves. Two very symmetric sub-namespaces bring geometry and charge together:

- `Wire` wire geom/charge at a given time bin
- `Cell` cell geom/charge at a given time bin

`WireCellXxx` hold any code which **bridges** “WireCell” and some external code base.

Example of the last one: `WireCellSst` “glues” in the “simple simulation tree” (aka “celltree”) data access.

System of Units

```
#include "WireCellData/Units.h"
int distance1 = 2.5*units::meter;
int distance2 = 10.0*units::meter;
cout << "The area is "
     << distance1*distance2/units::centimeter2
     << " square centimeters" << endl;
```

Rules:

1. Do not “care” about a variable value’s unit as long as it is in the **system of units**.
2. Every **bare, literal** number should have a unit **multiplied**.
3. To **express** a value in an explicit unit **divide** by the unit.
4. If you **really really must** store an explicit unit in a variable the pick a variable name that implies the unit. (but, try to avoid this)

```
// avoid all these cases!
float energyCutMeV = 50; // bad, but at least name has unit
float angle_radians = some_angle / units::radian;
float pi_radians = 180.0*units::degree / units::radian;
```

Source Package Types

Wire Cell has two basic source **package types**:

code
: holds code for shared libraries, applications, tests, etc (most common)

build
: includes one ore more code packages via "`git submodule`" method (just one now: `wire-cell`)

You will mostly create and add to **code packages**.

Code Package

- A *code package* holds the code to produce various build products.
- The build system assumes intent based on **layout conventions**:

`src/` source code (and private headers) for shared library

`inc/WireCellName/` public headers for shared library API.

`dict/LinkDef.h` export API to ROOT dictionary.

`tests/` unit tests (more on these below)

`apps/` main programs, one `*.cxx` per app.

`python/WireCell/<Name>` python modules (not yet supported in build)

`wscript_build` simple file hooking into the build system.

Entire package build file is one line (`examples/wscript_build`)

```
bld.make_package("WireCellExamples",
    use="WireCellNav WireCellData WireCellTiling WireCellSst")
```

Current Packages

Roughly in order of increasing dependency:

- `data` common **data classes**.
- `nav` data **navigation** (geometry, frames ("events") and time slices)
- `sst` provides frame and geometry data source classes for **"simple simulation tree"** (aka "celltree") and accompanying wire geometry.
- `tiling` things that produce or modify a **cell tiling**, includes initial, reference implementation based on Michael's `CellMaker` (also available in that packages as an application.
- `examples` a growing set of **example** applications, python code, etc. Useful source of starting points.
- `matrix` Xin's nascent area.
- (top) top level directory, source code aggregation, doxygen, and build package. (the `wire-cell` source package)

(labels are source subdirectory names)

Tests Overview

Let's write well tested code!

- You **already** write little tests when you write code
  - (unless you are superman).
- So, write them in a **useful form** from the start and keep them around:
  - Gives lots of examples how to use the code.
  - Makes it safer to attempt needed changes to your code or others.
  - Running tests can be (and is) automated so you can write once and then forget about them (until they fail)
- One challenge: tests take **no arguments**
  - need to run same everywhere so no outside info
  - leads to needing to mock up some things which can take extra effort
  - if too hard, then write an **application** to hold your test and write instructions how to exercise it.
- You can write tests in C++ or Python or Shell.

No excuses, write tests! ☺

Guidelines for Writing Useful Tests

- Write **many, small** tests:
  - Make each test just one thing.
  - Limit the time any one test takes to run.
  - Strive for complete testing coverage of your package.
- Don't worry about test-code quality, **quick-and-dirty is better than non-existent**.
- Be conscious of **dependencies**
  - don't let test code determine package dependencies
  - make a new package just for tests if needed

It is better to write tests than to follow guidelines!

C++ Tests

- Mini application but no command line arguments allowed.
- Place code in `tests/test_*.cxx`
- Auto-(re)built and (re)run as needed, not installed.

```
// test_fail.cxx
int main(/*empty!*/) {
  exit(1); //
}

// test_succeed.cxx
int main(/*empty!*/) {
  return 0;
}
```

Python tests

- Form of one or more unit test **functions** per Python file.
- Place code in `tests/test_*.py`
- Follow naming and the no-argument calling convention
- Automated running not yet added to build system

```
# tests/test_fail_succed.py

def test_fail():      # name starts with test_
  "A test that always fails."
  raise RuntimeError

def test_succeed():  # function takes no args
  "A test that always succeeds."
  return
```

Shell tests

- Form of an open-ended **shell script**, no cmd line arguments
- Run package's application(s) or those from other packages on which the package depends.
- Place code in `tests/test_*.sh`
- Automated running not yet added to build system

```
#!/bin/bash
set -e          # fail early, fail often!
wd=$(mktemp -d)
cd $wd
wget https://raw.githubusercontent.com/BNLIF/wire-cell-event/master/geomet
sst-geom-dumper ChannelWireGeometry.txt
rm -rf $wd     # clean up
```

Note: we need to deal better with auxiliary data files and not rely on downloads from GitHub like this example. Maybe start depending on SQLite3 for simple database features.

Installation Overview

1. Provide **external packages** (mostly ROOT**6**)
   - Automated externals installation method provided or,
   - You are free to DIY
2. Set up your **run-time environment**.
   - Automated installation provides two strategies
   - DIY'ers must continue to DIY
3. Build **Wire Cell code** itself (details next)

Note: automated **externals installation** method details here:

`https://github.com/BNLIF/wire-cell-externals`

Building Wire Cell

Prepare the source area

```
git clone git@github.com:BNLIF/wire-cell.git
cd wire-cell
git submodule init
git submodule update
alias waf=`pwd`/waf-tools/waf
```

Configure, build and install:

```
waf --prefix=/path/to/install configure build install
```

Some developer dancing:

```
waf clean build  # force a full rebuild
waf              # rebuild after an edit
waf install      # (re)install
```

More details in the wire-cell source package README file.

Documentation

- Every package has a `README.org` file
  - Simple text markup (Org-mode) that GitHub renders.
  - (note: there is **much** more to org-mode)
- Code is peppered with Doxygen markup
  - Optional: install **GraphVis** to get "`dot`" for nice graphs.
  - TODO: I will add the running of Doxygen to the build
  - TODO: Put Doxygen output on the web somewhere
  - TODO: Got through source and add more doc strings.

For now, run doxygen yourself:

```
$ doxygen docs/Doxyfile
$ firefox doxy/html/index.html
```

(BNL internal link to doxygen)

Much Symmetry between Wire and Cell

`WireCell/WireCharge.h`:

GeomWire
: geometry and numbering of one wire **segment**

GeomWireSet
: an owning collection of wires

GeomWireSelection
: a non-owning sub-set

Wire::Charge
: geom+charge for wire

Wire::Group
: a grouping of charges on wires

Wire::GroupCollection
: collection of groups

`WireCell/CellCharge.h` has much the same for Cells.

For both there are operations on group via functions:

```
GroupCollection singlets(const Group& group);
double charge(const Group& group);
```

Symmetry partly broken. Eg, Cell's have:

```
double cross_section(const Group& group);
```

Charge and Time

Trace a starting **time** bin and a sequence of following **charges** from an electronics **channel**.

TraceCollection A sequence of Trace objects

Frame a `TraceCollection` with an **index** into the external "frame data source" (see nav below)

Slice `Wire::Group` (vector of **wire ID + charge pairs**) in same time bin across a `Frame`.

- One full channel readout (eg, MB's 9600 time bins) can be represented by multiple `Trace` objects to allow for zero-suppression / analysis thresholds.
- A `Frame` object effectively corresponds to an "event" (bad word!) with the `index` being the ROOT TTree entry.

WireCellNav(igation)

GDS `GeomDataSource` gives information about the wire (segments) in the detector used to produce (and own) `Wire` objects.

FDS `FrameDataSource` provides `Frame` objects, expect to write one FDS per data source type.

- WireCellSst/FrameDataSource is so far only one

SDS `SliceDataSource` takes any `FrameDataSource` and produces slices

- One SliceDataSource covers all needs.
- Requires a GDS to resolve channel $\rightarrow$ wire IDs

Top-level user code will interact with all three of these objects. See `wire-cell-example-loop` for working example.

Tiling

A tiling is a class which **creates and owns cells** and provides access to them and answers **queries** about their associations with wires.

Tilings inherit from `TilingBase` and must provide:

```
/// Must return all wires associated with the given cell
GeomWireSelection wires(const GeomCell& cell) const;

/// Must return all cells associated with the given wire
GeomCellSelection cells(const GeomWire& wire) const;

/// Must the one cell associated with the collection of wires or 0.
GeomCell* cell(const GeomWireSelection& wires) const;
```

- The interface exposes a purely geometrical collection.
- Can write tilings that take into account charge and use other tilings as input.

Types of tilings

Tilings we have or will soon have*:

TileMaker the cell making code from Michael's CellMaker

TriangleTiling* special purpose exploiting MB's symmetry

Filter tilings* tilings that take other tilings as input, mostly to reduce available cells based on:

- removing chargeless wires
- ranking of cells
- info from a cell in neighboring slices
- kinematic fitters

Tilings are nodes to construct high-level process flows:

- chain individual tilings to produce final result
- supports branches, iterations, recombinations

Vector

Used best for collections where order is determined at fill time.

- "Array" type behavior
- Cheap random access, expensive insertion

```
#include <vector>
int main() {
    int dat[] = {5, 10, 15};
    std::vector<int> vec(dat, dat+3);
    vec.push_back(42);  // append
    for (std::size_t ind=0; ind < vec.size(); ++ind) {
        vec[ind] *= ind+1;
        // or
        vec.at(ind) *= ind+1;
    }
    return 0;
}
```

List

Used best for collections where order may be determined later

- "Doubly linked list" behavior
- Cheap insertion, no random access ($\mathcal{O}(N)$)
- Reverse iteration, insertion, erasure

```
#include <list>
#include <iostream>
int main() {
    typedef std::list<int> MyList;
    int dat[] = { 1,2,3 };
    MyList lst(dat,dat+3);
    lst.push_front(24);  // prepend, also pop_front()
    lst.push_back(42);  // append, also pop_back()
    MyList::iterator it, done = lst.end();
    for (it = lst.begin(); it != done; ++it) {
        *it *= 100;
        std::cout << *it << std::endl;
    }
    return 0;
}
```

Set

An collection keyed by its value (unique entries) and kept in order.

- Also comes in an unordered, faster version

```
#include <set>
#include <iostream>
int main()
{
    typedef std::set<int> IntBag;
    IntBag ib;
    ib.insert(5);
    ib.insert(3);
    ib.insert(7);
    IntBag::iterator it, done = ib.end();
    for (it = ib.begin(); it != done; ++it) {
        std::cout << *it << std::endl;
    }
}
```

Prints: `3 5 7`

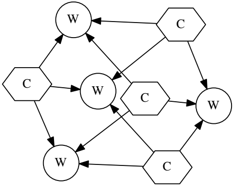
Map

Used to associate one type to another.

- `std::map` ordered by key ($\mathcal{O}$(N) insert/access)
- `std::unordered_map` ($\mathcal{O}$(1) insert/access)
- Each element is a `std::pair<type1,type2>`

```
#include <map>
#include <iostream>
int main()
{
    typedef std::map<int,float> SparseHist; // save typing
    SparseHist sh;
    sh[2] = 20;
    sh[4] += 2; // default value springs into life
    SparseHist::iterator it, done = sh.end();
    for (it = sh.begin(); it != done; ++it) {
        std::cout << "bin #" << it->first
                  << " content:" << it->second << std::endl;
    }
}
```

Graph



- Nodes connected by Edges
- Directed Acyclic Graphs (DAG)
  - Edge directs from tail to head node, no loops
  - If node has zero or one "input edge" $\Rightarrow$ "tree"
- Undirected Cyclic Graphs (meshes)
  - Express connectivity with no direction, cycles allowed.
  - Wires and Cells can form a duel-node type mesh
  - Wire-Cell/Cell-Wire but not Wire-Wire/Cell-Cell
  - Walk mesh to find any wire in a cell and vice versa, cell given three wires, connecting wires given a set of cells

```
// WireCellMap.h
typedef std::map<const Cell*, WireSelection> CellMap;
typedef std::map<const Wire*, CellSelection> WireMap;
```