Lelaps and GODL

Fast Detector Simulation Using Lelaps

Detector descriptions in GODL

Overview

- Lelaps reminder
  - For details, see http://lelaps.freehep.org
- Geometry descriptions using GODL
  - Features
  - Status
- Future
- Appendix: A brief tour of GODL

Lelaps

Lelaps Reminder

- Features:
  - Lelaps consists of a set of C++ class libraries and a main program, which itself is called lelaps.
  - The main class library is called CEPack, the actual simulation toolkit.
  - Built-in support for LDMar01, SDJan03 and SDMar04.
  - New! Also reads detector geometries in GODL format.
  - Reads StdHep (uses IStdHep class) generator files and produces SIO or LCIO output files.
- What it does:
  - Lelaps is a fast detector simulation that swims particles through detectors with magnetic fields, accounting for multiple scattering and energy loss.
  - It produces parameterized showers in EM and hadronic calorimeters.
  - It supports decays of certain short-lived particles ("V" decays).
  - It converts gammas.
  - Performance: ~1 typical (e.g. ZZ) event/second at 1 GHz, with everything turned on and writing LCIO output file.

Materials in CEPack

- All elements built in with default pressure/temperature/density.
- Any compound can be specified by chemical formula and density or (for gasses) temperature and pressure.
- Mixtures can be created by mixing elements and compounds (by volume or by weight).
- All needed quantities are calculated automatically
  - Constants needed for multiple scattering and energy loss
  - Radiation lengths (Tsai, PDG)
  - Interaction lengths (from a fit to element data)
  - Other constants needed for shower parameterization
- Lelaps distribution comes with a little program called **`matprop`**
  - Matprop is available online: http://www.slac.stanford.edu/comp/physics/matprop.html

Multiple Scattering and dE/dx

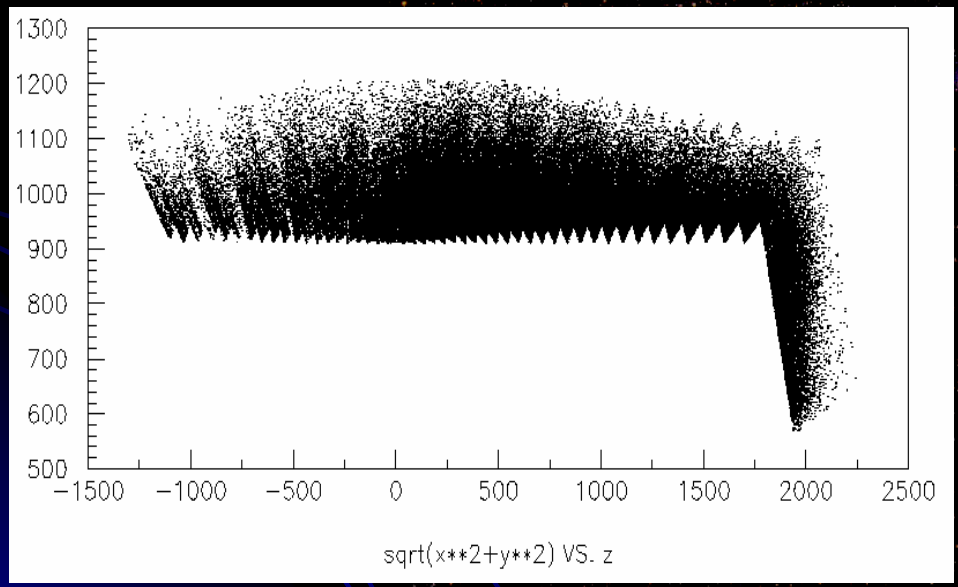
- Multiple scattering is performed using the algorithm of Lynch and Dahl.
  - Gerald R. Lynch and Orin I. Dahl, Nucl. Instr. And Meth. B58 (1991) 6.
- Material is “saved up” along the track until there is enough.

- dE/dx is calculated using the methods by Sternheimer and Peierls.
  - R.M. Sternheimer and R.F. Peierls, Phys. Rev. B3 (1971) 3681.

- All constants precalculated by the material classes.

Shower Parameterization

- Electromagnetic showers are parameterized using the algorithms of Grindhammer and Peters.
  - G. Grindhammer and S. Peters, arXiv:hep-ex/0001020v1 (2000) (1993 conference contribution, submitted to the archive in 2000).

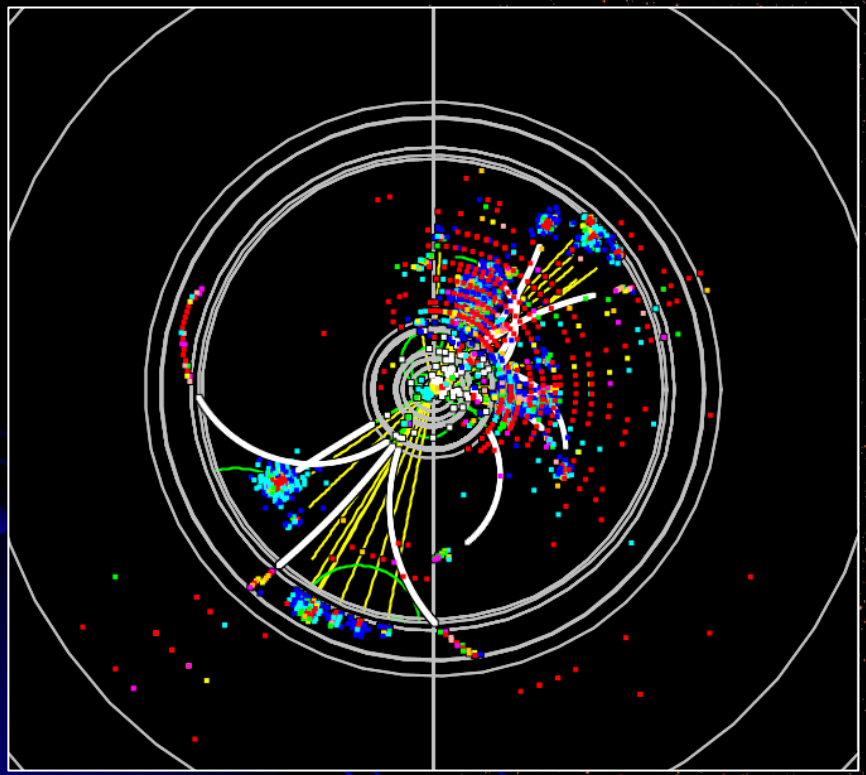


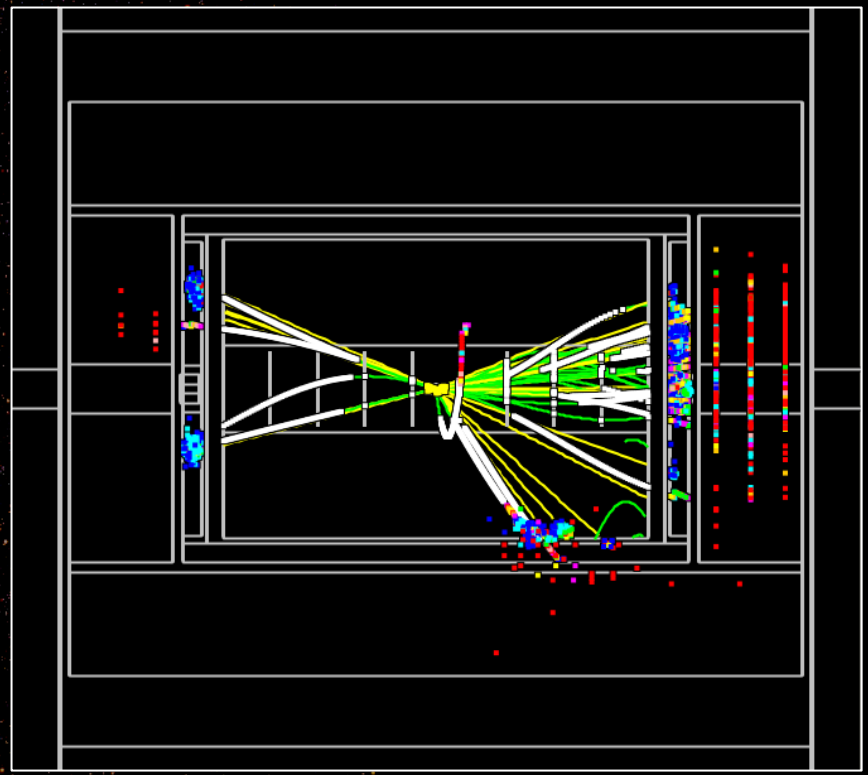
CEPack simulation of BaBar EM calorimeter in Moose (courtesy of Dominique Mangeol). See also the BaBar web site (computing – simulation – fast simulation).

Shower Parameterization

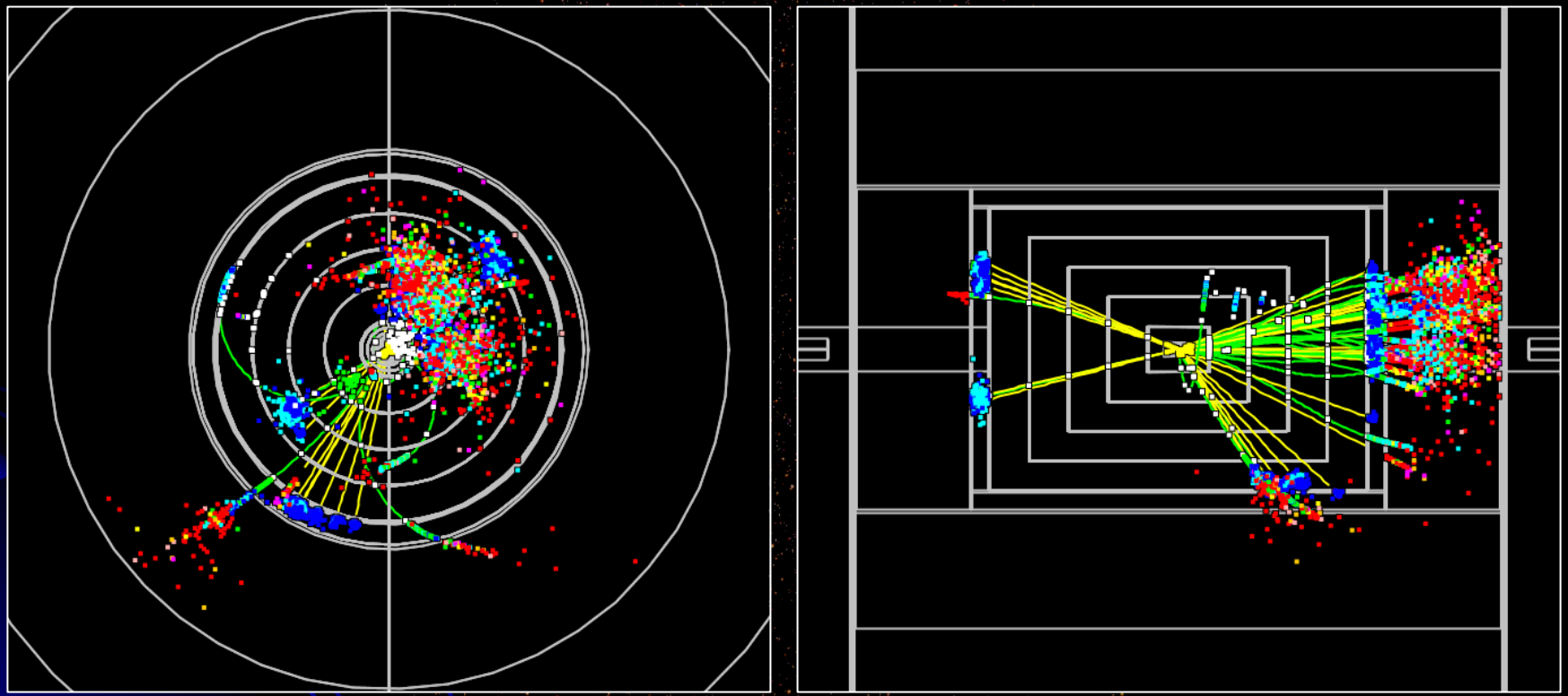
- Hadronic showers are parameterized using code that is similar to the code for electromagnetic showers, with some modifications.
  - R.K. Bock, T. Hansl-Kozanecka and T.P. Shah, Nucl. Instr. And Meth. 186 (1981) 533.
- Parameterized shower simulation was compared to Geant4.
  - "Parameterized Shower Simulation in Lelaps: A Comparison with GEANT4", Daniel Birt, Amy Nicholson, Willy Langeveld, Dennis Wright, SLAC-TN-03-005, Aug 2003.
- In general pretty good agreement for EM showers. Hadronic showers agree pretty well longitudinally, but not as well radially.
  - Hadronic shower parameterization has been tweaked since then.

Lelaps (LDMar01)

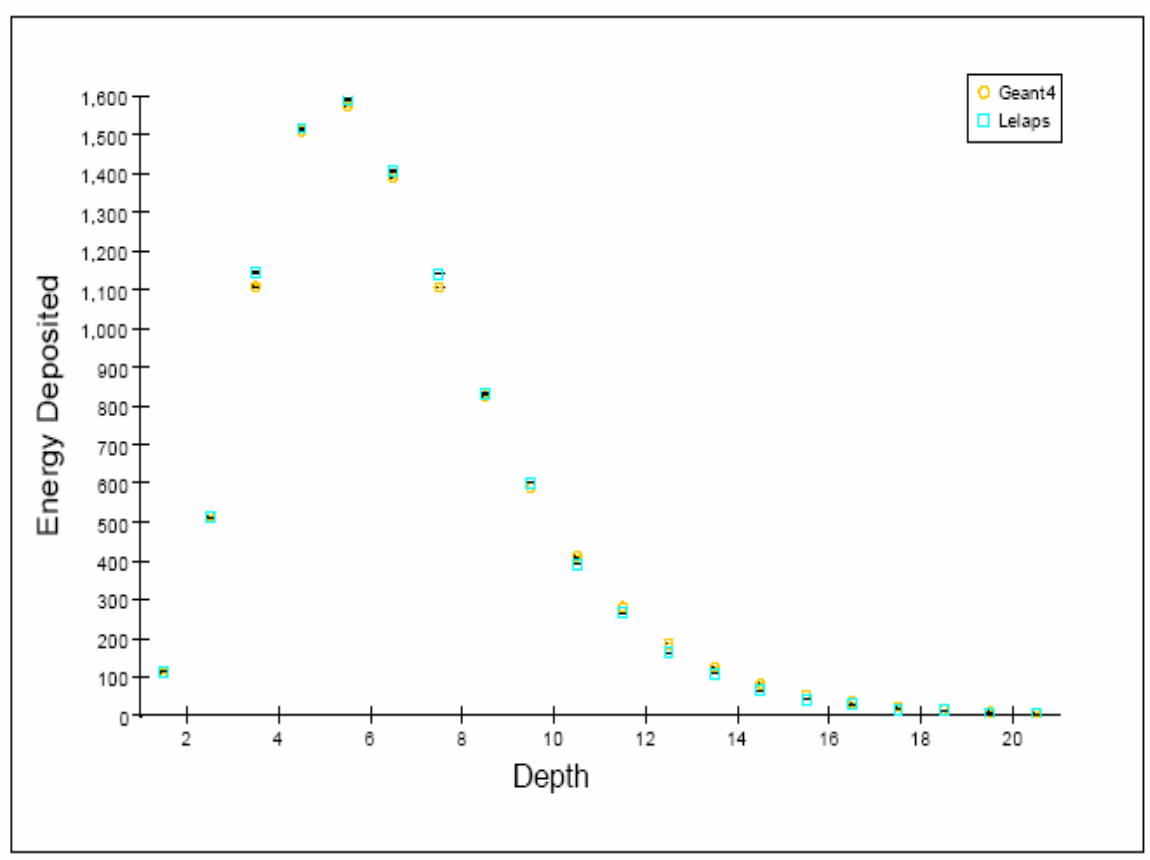




Lelaps (SDJan03)



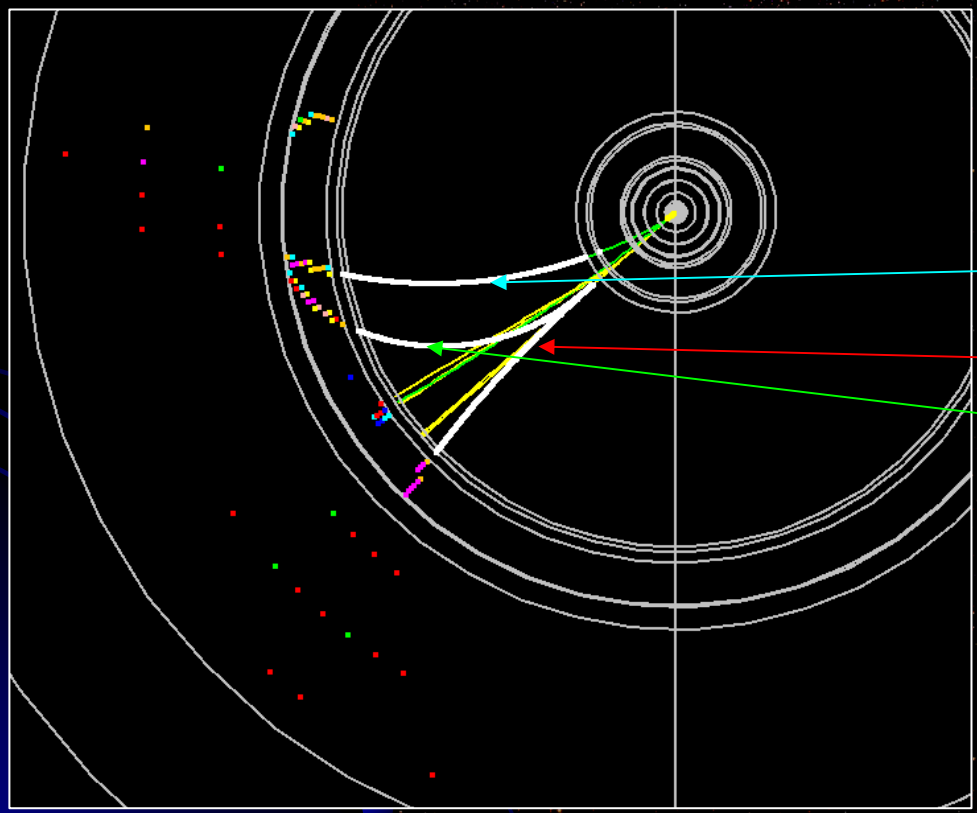
EM Shower Parameterization and Geant4



Comparison of CEPack longitudinal profile (green) of a 10 GeV electron in an EM calorimeter with Geant4 (orange).

Decays

- Supported unstable particles are pi0, K0-short (K0-long treated as stable), Lambda, Sigma+/-/0, Xi-,0 and Omega- . Only decay modes > 2% supported ("V" decays)



Wired picture of the decay chain:

$\Omega^- \rightarrow \Xi^0 \pi^-$

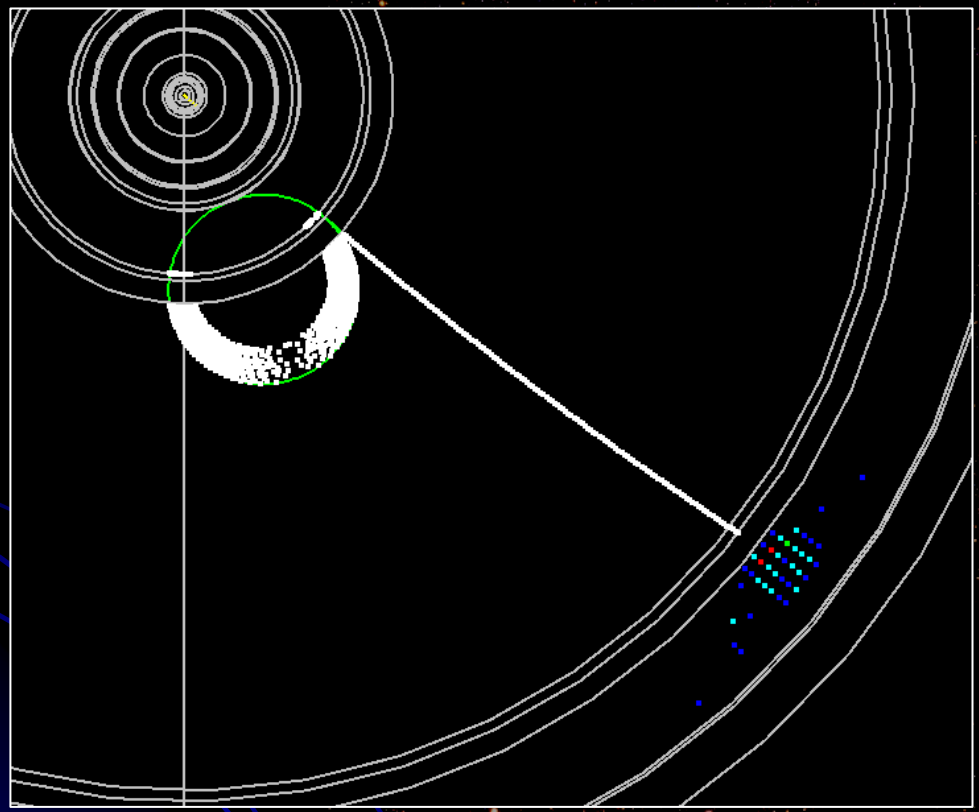
$\Xi^0 \rightarrow \Lambda \pi^0$

$\Lambda \rightarrow p \pi^-$

$\pi^0 \rightarrow \gamma \gamma$

as simulated by Lelaps for the LCD LD model.

Conversions



Wired picture of a gamma conversion as simulated by Lelaps for the LCD LD model.

GODL

Generalized Object Description Language Features

- GODL is a language with variables and operations.
  - Don’t have to hard-code dimensions and locations
  - Can compute dimensions and locations from previously defined ones
  - Don’t have to chase down all dependencies when changing one number
  - Human readable and editable (like XML but more so), portable
  - Interpreter catches mistakes (like XML but better)
- GODL knows about units and enforces them.
  - Can e.g. mix microns and meters
  - Can define new units based on built-in ones and use them
  - interpreter enforces consistent usage in operations and function calls
- GODL has control constructs (loops, if).
  - Makes repetitive operations easier

Generalized Object Description Language Features

- GODL has built-in math functions
  - Allows calculating derived quantities (e.g. tangent of an angle)
- GODL has list objects
  - Built-in objects to describe materials, objects, placements etc.
- GODL allows specification of arbitrary calorimeter segmentation and encoding of tracker and calorimeter ID's
  - Built-in parser of RPN (PostScript-like) ID code specification
  - Compiles to byte-code for fast execution
  - Allows changing encoding and/or segmentation without modifying simulator source code
- GODL supports levels of detail
  - Allows using low level of detail for fast simulation, high level of detail for full simulation

Generalized Object Description Language Features

- GODL comes with simple API (currently 11 virtual methods)
  - Implemented fully in Lelaps
  - GODL-to-HepRep converter exists (requires new Wired4)
  - Geant4 implementation planned
- GODL supports volume hierarchies
  - Save time by embedding sub-detector elements into mother volume
- GODL supports (in principle) all Geant4 solid types (with the possible exception of BREPS)
  - Not all of them implemented at this time, but easy to do
- GODL supports (in principle) combinatorial (“boolean”) geometry (“CSG” in Geant4).
  - Not yet implemented, but straightforward.

GODL - Status

- Parser/evaluator is essentially complete.
  - API layer to access the volume list exists.
  - Completely implemented in Lelaps V03-23-26.
  - Includes levels of detail and ID calculation.
  - SDMar04.godl file exists.
  - SDJan03.godl with two different levels of detail exists.
  - GODL-to-HepRep converter exists. The HepReps it produces need recently updated Wired 4.

Future

Future

- Lelaps and CEPack interfaces are not yet frozen!
- New features planned for CEPack
  - Support for combinatorial geometry
  - Shower continuation into next volume
  - More tuning of hadronic showers
- New features planned for GODL:
  - Add the remaining standard geometrical shapes
  - Support for combinatorial geometry

About the name Lelaps

Lelaps (“storm wind”) was a dog with such speed that, once set upon a chase, he could not fail to catch his prey. Having forged him from bronze, Hephaestus gave him to Zeus, who in turn gave him to Athena, the goddess of the hunt. Athena gave Lelaps as a wedding present to Procris, daughter of Thespius, and the new bride of famous hunter Cephalus.

A time came when a fox created havoc for the shepherds in Thebes. The fox had the divine property that its speed was so great that it could not be caught. Procris sent Lelaps to catch the fox. But because both were divine creatures, a stalemate ensued, upon which Zeus turned both into stone. Feeling remorse, Zeus elevated Lelaps to the skies, where he now shines as the constellation Canis Major, with Sirius as the main star.

Introduction: Lelaps

…but clearly, Lelaps (the program) is not a dog!

Appendix: GODL - A Brief Tour

GODL – Language Features
Variables and Arrays

- GODL is an extensible typeless programming language.
  - Type determined by assignment:

```
a = 2.4;        # real
b = 2;          # integer
c = “text”;     # string
d = true;       # boolean
```

- It has variables and operations that can be performed on them:

```
a += 1;
b = a * 12;
d = c + “ more text”;
e = false;
b = e != true;
```

- Array-like constructs:

```
i = 5; foo.i = 12;    # Same as foo.5 = 12;
```

GODL – Language Features
Operators

- Set of operators (some cannot be used in some contexts):
  - `+ - * / = += -= *= /= == < > <= >= != ! && ||`
- Reference operator @

```
a = 12;
b = @a; print(b, "\n");
```

```
@a->(12)
```

Useful for referencing objects multiple times without recreating them

GODL – Language Features
Built-in Functions

- It knows about the usual set of math functions:
  - **`exp, log, sqrt, pow, cos, sin, tan, acos, asin, atan, atan2, cosh, sinh, tanh, acosh, asinh, atanh, log10, abs, fabs, ceil, floor, mod`**
- In addition:
  - **`list`**

```
a = list(a, b, c, d);    # Creates unnamed list
```

  - **`print`**

```
print(a, "\n");
print(a, "\n", b, "\n");
```

  - **`argc, argv`**
    - When arguments are provided
  - **`unit`**
    - See later

GODL – Language Features
Control Constructs

- It has a limited set of control constructs:
  - C-style for and while loops:

```
for (i = 0; i < 25; i += 1) {
    …
}
while (true) {
    …
    if (something) break;
}
```

  - C-style if statements (no “else” yet)

```
if (a < b) {
    …
}
```

GODL – Language Features
List Objects

- Variables can be list objects:

```
a = foo(a, b, c, d);
```

- Lists can contain objects of any type, including other lists.
- To add objects to a list:

```
a += e;
a += f;
```

- Note that this is not necessarily the same as:

```
a += e + f;
```

which would first add f to e and then the result to a. If e and f are list objects, this adds to “a” a single list “e” which in turn contains “f”.

GODL – Language Features
Units

- Variables can have units, and units are enforced across operations and in arguments to functions and list objects:

```
m = _meter;       # _meter is a built-in unit
unit("m");        # Declare as unit
a = 2 m;
b = 12 * a;
area = a * b;
area += 5;        # Error: incorrect units
d = cos(area);    # Error: cos() only takes angles
```

GODL – Language Features
Units

- Available units (like CLHEP):
  - Basic units: _meter, _second, _joule, _coulomb, _kelvin, _mole, _candela, _radian, _steradian
  - Derived units: _angstrom, _parsec, _barn, _degree, _hertz, _becquerel, _curie, _electronvolt, _gram, _watt, _newton, _pascal, _bar, _atmosphere, _ampere, _volt, _ohm, _farad, _weber, _tesla, _gauss, _henry, _gray, _lumen, _lux
- Create new units:

```
m = _meter; g = _gram;  # For convenience
unit("m", "g")          # Declare as units
gcc = g/cm3;            # New unit of density
unit("gcc");            # Declare
```

- Automatically converts SI prefixes and powers:

```
a = 1 cm2;              # = 0.0001 _meter squared
```

GODL – Language Features
Miscellaneous

- Built-in constants:
  - _pi (3.14…) has units of rad
  - _e_SI (electron charge 1.6...10^{-19} C), _e (2.71…) dimensionless
- Debugging functions:
  - **`verbose:`** prints a lot of debugging information to stdout
  - **`__printvars:`** prints a list of all variables to stdout
- Control variables for **`print()`** function:
  - **`printlevel_:`** (default 1) controls how much information to print (mostly for for object lists).
  - **`precision_:`** controls how many digits are displayed for floating point numbers.
  - **`fieldwidth_:`** controls how much space a printed number takes up.

GODL – Built-in List Objects
Materials

- Materials are declared using the element, material or mixture list objects (use the @ operator to pass by reference):

```
Si      = element("Si");
vacuum  = material("vacuum");
O2      = material(formula("O2"),
                   pressure(1.0 atm),
                   temperature(293.15 K));
Tyvek   = material(name("Tyvek"),
                   formula("CH2CH2"),
                   density(0.935 g/cm3));
Air     = mixture(part(@O2, 20.946),
                  part(@N2, 78.084),
                  part(@Ar, 0.934),
                  by("volume"));
```

GODL – Built-in List Objects
Volumes and Placements

- First define a World Volume:

```
World = cylinder(radius(700.0 cm), length(14.0 m), @vacuum);
```

- Define another volume:

```
em_ec_irad  = 21.0 cm;                  em_ec_orad   = 125.0 cm;
em_b_irad   = em_ec_orad + 2.0 cm;      em_thickness = 15 cm;
em_b_orad   = em_b_irad + em_thickness; em_nlayers   = 30;
em_sampfrac = 0.02664;                  em_b_length  = 368.0 cm;

EM_Barrel = cylinder(name("EM Barrel"),
                     innerRadius(em_b_irad),
                     outerRadius(em_b_orad),
                     length(em_b_length), @SiW,
                     type("emcal"), nLayers(em_nlayers),
                     samplingFraction(em_sampfrac));
```

- Add to World using placement:

```
World += placement(@EM_Barrel);
```

GODL – Built-in List Objects
Volumes and Placements

- Use loops to do repetitive tasks and if statements for conditionals:

```
Vertex_Barrel = cylinder(name("Vertex Barrel"),
                         innerRadius(v_irad), outerRadius(v_orad),
                         length(v_lenmax));

for (i = 1; i <= v_nlayers; i += 1) {
   vlen = v_leninner;
   if (i > 1) vlen = v_lenmax;
   Vertex_Barrel.i = cylinder(name("Vertex Barrel " + i),
                              innerRadius(v_spacing * i),
                              outerRadius(v_spacing * i + v_thickness),
                              length(vlen), @Si,
                              type("tracker"));
   Vertex_Barrel += placement(@Vertex_Barrel.i); # Notice hierarchy
}

World += placement(@Vertex_Barrel);
```

GODL – Built-in List Objects
Levels of Detail

- Specify levels of detail with “level” tag:

```
Had_Endcap = cylinder(name("Had Endcap"), level(1),
                      innerRadius(had_ec_irad),
                      outerRadius(had_ec_orad),
                      length(had_thickness), @StainlessPoly,
                      type("hadcal"), nslices(had_nlayers),
                      samplingFraction(had_sampfrac));

Had_Endcap += placement(@something, …, level(max(0)), …);
Had_Endcap += placement(@something_else, …, level(min(1)), …);

World += placement(@Had_Endcap, translate(0, 0, 0.5 * (had_b_length –
                   had_thickness)));
World += placement(@Had_Endcap, rotate(axis(“y”), angle(180 degrees)),
                 translate(0, 0, -0.5 * (had_b_length – had_thickness)));
```

GODL – Built-in List Objects
Levels of Detail

Level syntax:	Create object when:	Used for:
<not specified>	always	Fundamental objects that are always present
level(min(2))	level >= 2	Detailed objects that should not be simulated at lower levels
level(max(4))	level <= 4	Fundamental objects that are replaced with other objects at higher levels
level(min(2), max(4)) level(range(2, 4)) level(mask(0x1C))	2 <= level <= 4	Combinations: objects relevant only in a certain level range

GODL – Built-in List Objects
ID Calculation

- ID calculation (such as CalorimeterID and TrackerID) is generally used for two purposes:
  - To specify segmentation of a detector: hits with the same ID are combined to a single energy deposition.
  - To specify an abbreviated version of the location of an energy deposition.
- Problem: how can we change the amount or method of segmentation without changing the C++ source code of the simulator?
- One solution: Specify the segmentation method in the geometry file and “interpret” it inside the simulator.
- GODL API provides a simple, fast, interpreter to do that.
  - In fact, it “compiles” the segmentation specification into byte-code, and runs the byte code for each hit.

GODL – Built-in List Objects
ID Calculation

- Example: Tracker ID. For the Vertex detector barrel we would use:

```
Vertex_Barrel.i = cylinder(name("VXD"),
                           innerRadius(1.2 cm * i),
                           outerRadius(1.2 cm * i +
                                       0.01 cm),
                           length(vlen), @Si,
                           type("tracker"),
                           idCode(code(tb_code),
                                  data("system", 1),
                                  data("id", i - 1)));
```

- The ID calculation in idCode is specified as a string in a “code” list object. The algorithm for the vertex and barrel trackers is:

```
tb_code = "x: fp0 y: fp1 z: fp2 layer: d3 id z H
0x40000000 mul or system 28 bitshift or stop"
```

GODL – Built-in List Objects
ID Calculation

- For the tracker end cap it is:

```
"x: fp0 y: fp1 z: fp2 layer: d3 id 0x80000000 or z H
0x40000000 mul or system 28 bitshift or stop";
```

- The "**`x: fp0`**" part means that the API routine that evaluates the byte code associated with the above expects x to be given in the first floating point "register". Similarly, "layer" is provided as an integer in the fourth register.
- Reverse polish PostScript-like language with some limitations and some extras:
  - Some named variables must be provided by the simulator as standard arguments (**`x, y, z, layer`**)
  - Some named variables are provided using "data" object lists in the specification.
  - **`H`** is the Heaviside step function: 1 if argument positive, 0 otherwise.

GODL – Built-in List Objects
ID Calculation

- Slightly more work for the calorimeters. For the end caps we have:

```
cal_code_ec = "x x mul y y mul add sqrt z atan2
theta_seg mul pi div “ + standard_code;
```

- For the barrel we have:

```
cal_code_b = "x x mul y y mul add sqrt z atan2 cos
1.0 add theta_seg mul 2.0 div “ + standard_code;
```

- where standard_code is:

```
standardl_code = “truncate 11 bitshift y x atan2m
phi_seg mul 0.5 mul _pi div truncate or layer 21
bitshift or system 28 bitshift or stop";
```

- We have to add standard argument specifications to this:

```
cal_code_ec = "x: fp0 y: fp1 z: fp2 layer: d3 "+
                cal_code_ec;
```

GODL API

- The GODL API consists of four classes: GODLParser, MCode, MStack and MVar.
- There are (currently) 11 virtual functions that the API implementer must write. Example:

```
virtual int constructCylinder(
        const char   *nameForFutureReference,
        const char   *objectName,
        double        innerRadius,// length units: meter
        double        outerRadius,
        double        length,
        const char   *materialRefName,
        const char   *type,
        int           nLayers,
        int           nSlices,
        double        samplingFraction,
        const MStack &IDCode)
```

GODL API

- Other functions:
  - `constructCone(...);`
  - `addField(...);`
  - `addPlacement(...);`
  - `constructPlacement(...);`
  - `rotate(...);`
  - `translate(...);`
  - `constructElement(...);`
  - `constructCompound(...);`
  - `constructMixture(...);`
  - `addMixture(...);`
- API reads .godl file and calls “construct” routines to construct objects and placements. It then calls rotate and translate on the placements and addMixtures to add materials to the mixtures. Finally it calls addPlacement to instantiate an actual placement of an object.

The End