(Geant4) Simulation and Concepts

Yulei Zhang

Shanghai Jiao Tong University

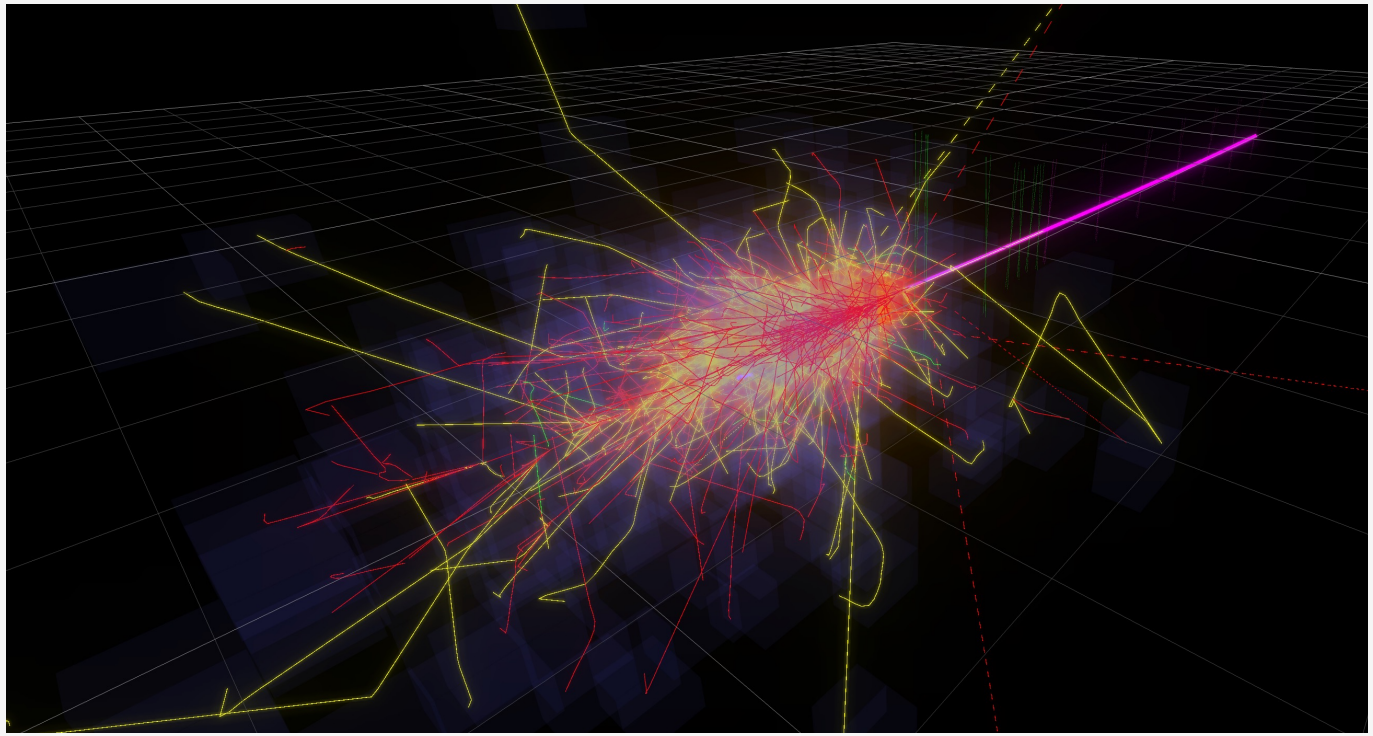
January 20th, 2023

What are Monte Carlo (MC) techniques for?

- Numerical solution of a (**complex**) macroscopic problem, by simulating the microscopic interactions among the components
- Applications not only in physics and science, but also finances, traffic flow, social studies, etc.
- In physics, elementary laws are (typically) **known** → MC is used to predict the outcome of a (complex) experiment
  - Exact calculation from the basic laws is unpractical
  - Optimize an experimental setup, support data analysis
- Monte Carlo for particle tracking (interaction of radiation with matter)

Most common application in HEP: particle tracking

- Problem: track an electron in a detector and determine the energy spectrum deposited.
- All physics is known from textbook (Compton scattering, photoelectric effect, etc.)
- However, the analytical calculation is a nightmare (😵) → Monte Carlo simulation clearly wins



Why Simulate Anything?

- We can (usually) only build one detector
  - What will we miss because of our detector design?
  - How would a slightly different detector affect things?
  - How will the detector stand up to radiation damage?
- Most detectors only measure voltages, currents, and times
  - It's an *interpretation* to say that such-and-such a particle caused such-and-such a signature in the detector
  - We can use simulation to correct our observables and understand our (in)efficiencies
- There is only one right answer in nature
  - What would new physics look like in our detector?
  - Could we find it under realistic conditions?
  - What are the biggest problems, and how do we ease them?
- **A good simulation is the way to demonstrate to the world that you understand your detector and the physics you are studying**

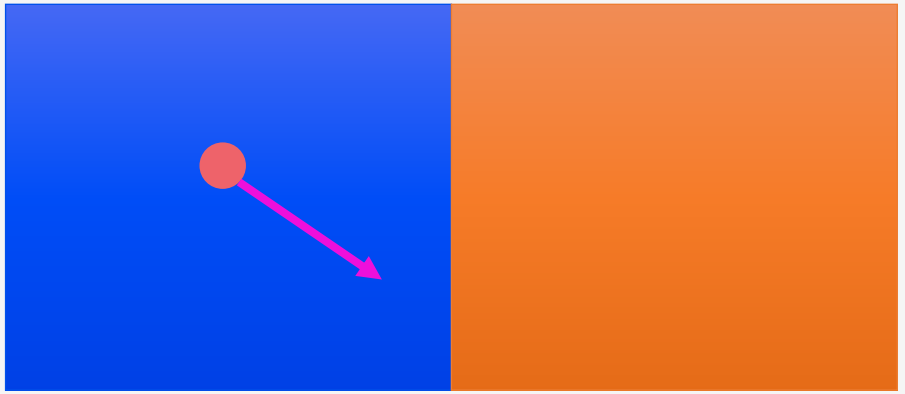
QUIZ: Write Me a Simulation for This

A Typical DarkSHINE Event

Simulation Basics

- Break the problem up as much as possible
  - Do you understand all the steps of the system?
- For each piece of the problem, write some code
  - Did you remember all the effects for each step?
- Spend enough time on each piece that you get the accuracy that you need out of them
- Cross your fingers and press the button 🙏
- There are two general approaches for a detector simulation
  - **Full Simulation:** We can simulate every little detail along the way → Geant4 (GEometry ANd Tracking)
  - **Fast Simulation:** We can go straight for the final state → "A pion will look like such-and-such" & Smear things directly

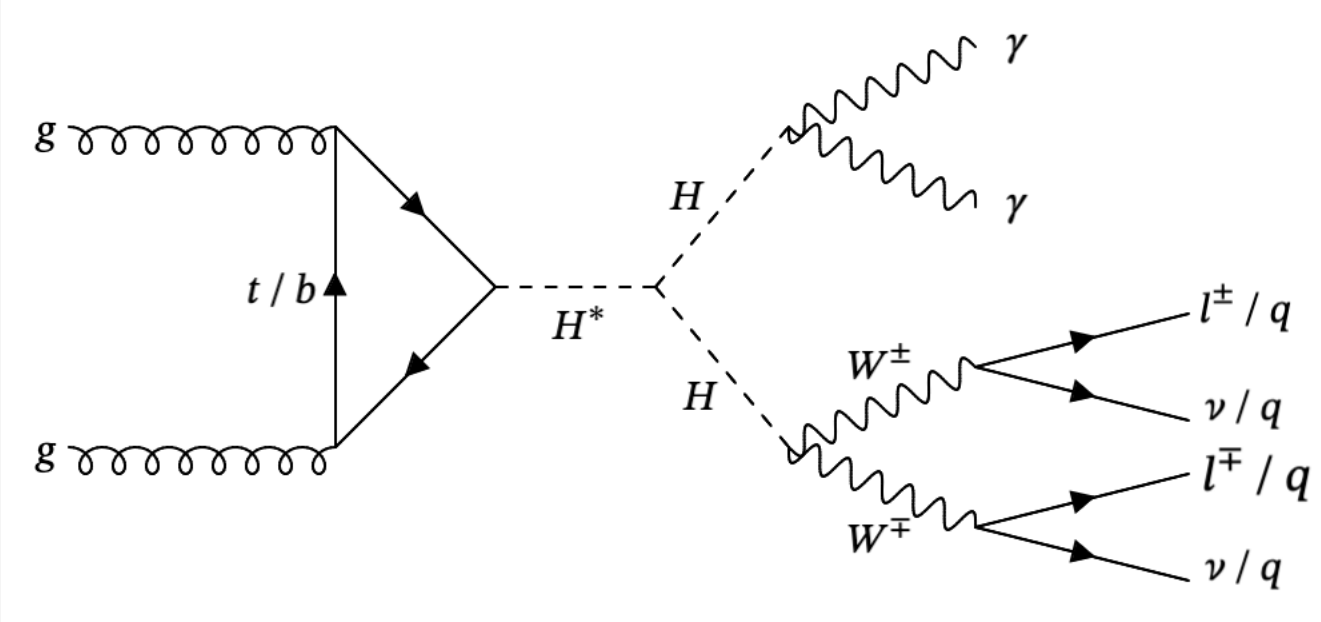
Primer on Simulation

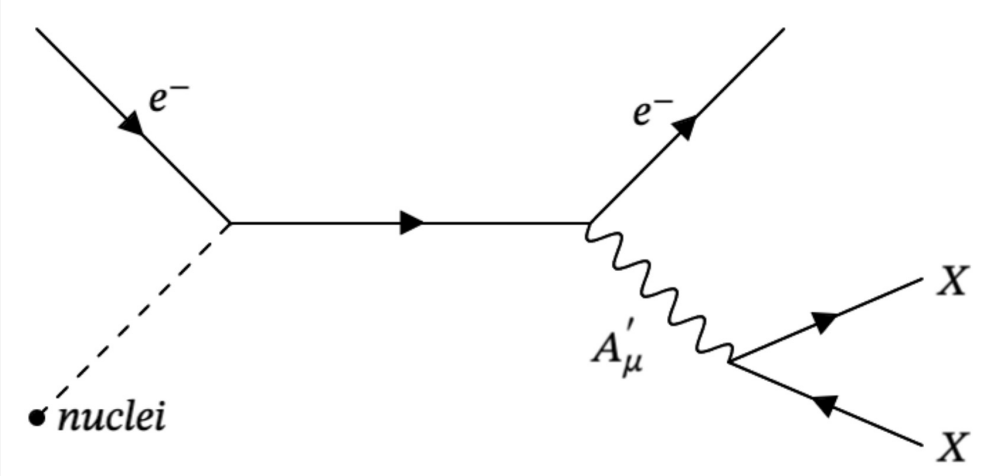


- In Geant4, every problem looks like this:
- It has the concepts of:
  - **Particle** (if it isn't standard model, G4 has no idea about it! → Define new particles 🤓)
  - **Material** (you define everything except the elements)
  - **Magnetic field** (you define it at every point)
  - **Physics process** (you get to pick from their list! → Write your own physics 😁)
- It is *only a toolbox* → it's up to you to put the pieces together
  - **Don't expect it to be any smarter than you are.**
  - Saying "we do the simulation with Geant4" is like saying "We do the analysis with ROOT." It's true, but it's not enough to explain anything.

Simulation Step 1

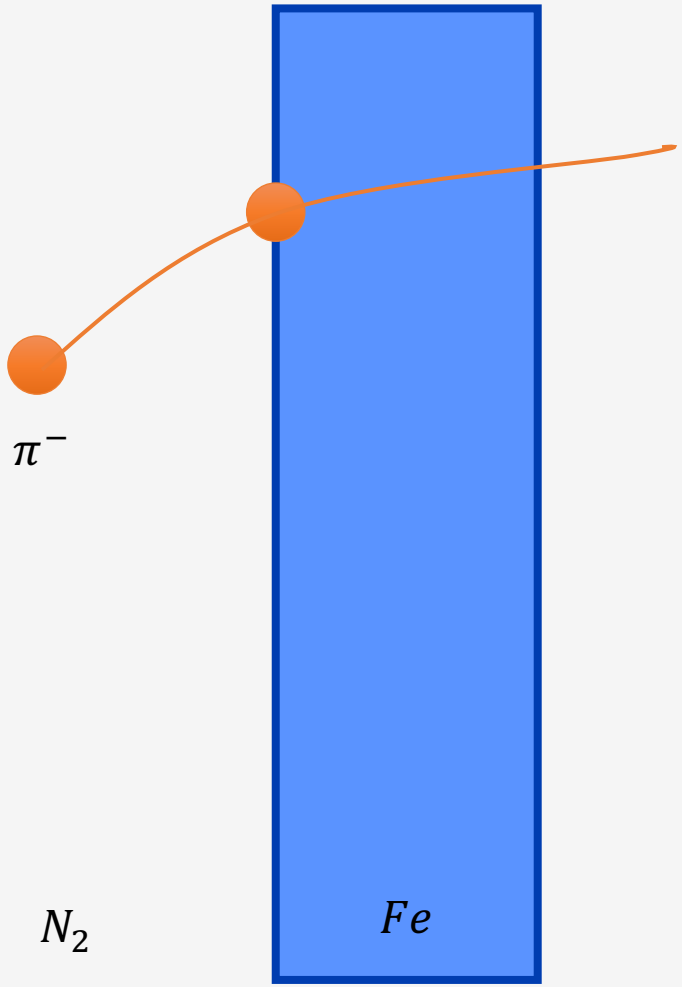
- Question 1: What am I looking at, and what can it do?
  - Simulation is agnostic about the generator; It *only* cares about final state (stable) particles
  - What is “final state” that *depends on the experiment*





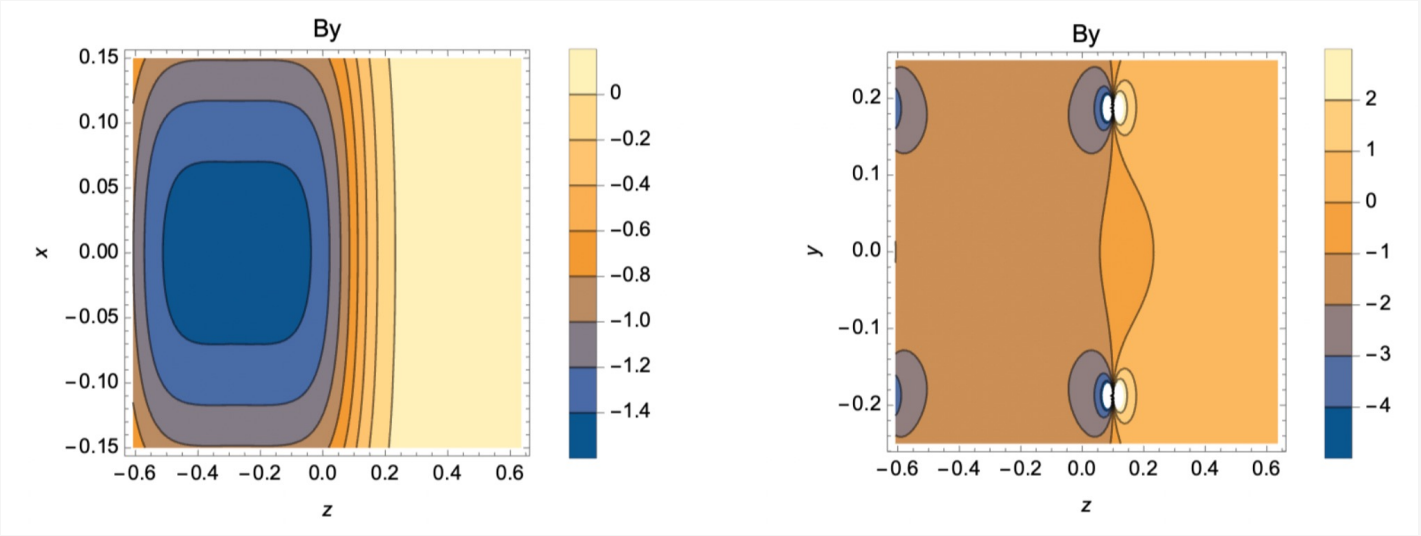
Simulation Step 2

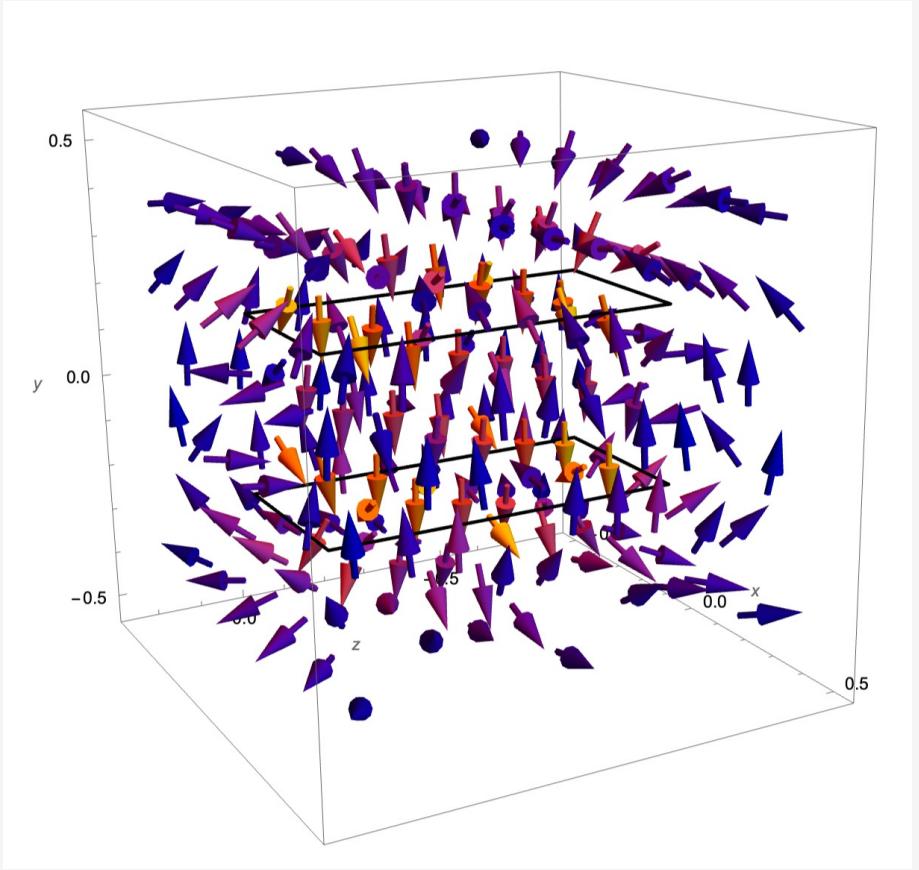
- Question 2: How far may I go?
- Never move farther than a volume boundary
  → the physics could change!!
- Check on all physics models. For a pion, this means multiple scattering, bremsstrahlung, nuclear interactions, decay, ionization…
- New secondary particles?
- Adjust energy and momentum accordingly



The Magnetic Field

- No hope for simulating on the fly - must have a **map**
  - Do they match reality?
  - Do they match the geometry?





Numerical Models

- In reality, particle moves continuously, but numerical models are not!
  - All processes become discrete, including “transportation”
- Use phenomenological models tuned to experimental data
  - Never solve a Lagrangian 😎
- Some interactions are easy
  - Photon conversion ($\gamma \rightarrow e^+e^-$), Bremsstrahlung
- Some are hard to model
  - Photo-nuclear, electro-nuclear
- Some have a variety of models
  - Good in an energy range; transitions can be problematic.

■ **Some Hadronic options:**
- “QGS” Quark Gluon String model (> ~15 GeV)
- “FTF” FRITIOF String model (> ~5 GeV)
- “BIC” Binary Cascade model (< ~10 GeV)
- “BERT” Bertini Cascade model (< ~10 GeV)
- “P” `G4Precompound` model used for de-excitation
- “HP” High Precision neutron model (< 20 MeV)

■ **Some EM options:**
- No suffix: standard EM i.e. the default `G4EmStandardPhysics` constructor
- “EMV” `G4EmStandardPhysics_option1` CTR: HEP, fast but less precise
- “EMY” `G4EmStandardPhysics_option3` CTR: medical, space sci., precise
- “EMZ” `G4EmStandardPhysics_option4` CTR: most precise EM physics

New Secondaries

- Each particle is called "**Track**" (*in Geant4*)
- If there is an interaction, generate all the secondaries
  - EM processes have a "range cut" – $e^+/e^-/\gamma$ that have an average range below that distance are never created.
  - Incorrect setting of this range cut caused a major production problem. (*tracking reconstruction problem in DarkSHINE*)
  - Secondaries will be pushed back into "**Stack**" for later simulation
- In general, the models of what secondaries are produced in an interaction are well-educated guesses

What Really Happens

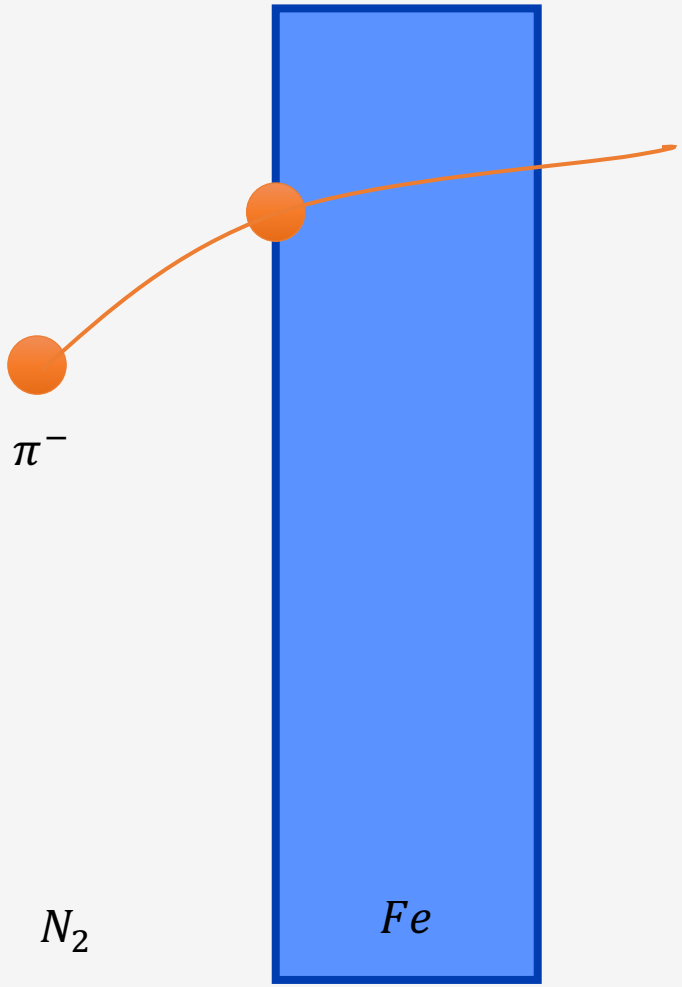
- Each iteration is called a "**step**"
- The number of steps dictates the speed of the simulation
  - The best code changes get us $\sim 3 - 10\%$ speed ups; the best physics changes get us $20 - 50\%$ speed ups
  - Simulation time goes with energy, so it is very important how far forward you simulate particles – but you need to be careful to not harm detector response!
- Most time just moving stuff around in the calorimeter
  - 50% of your simulation time is spent moving $e^{\pm}$ and γ below 10 MeV

Simulation Step 3

- Question 3: Anything else to do?

- The user is allowed a hook at the end of this "step" to perform any necessary action, for example to **make a record of energy deposition**

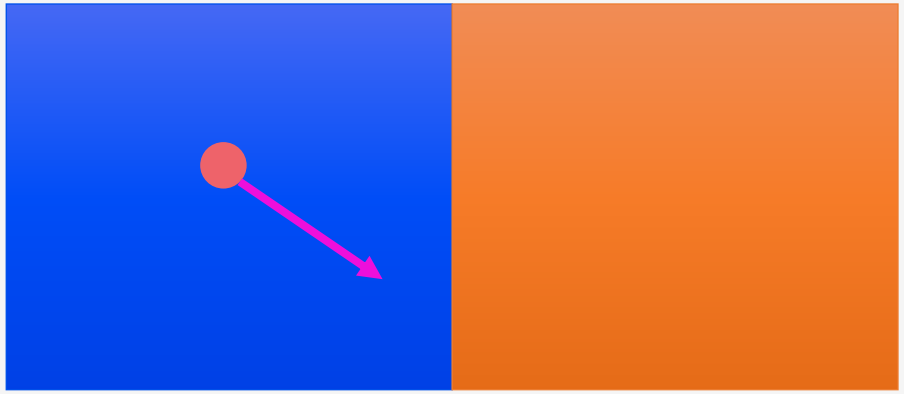
- All particles are tracked to zero energy or their exit of your world



Truth in DarkSHINE

- Of course, the most interesting interactions in the detector need to be saved to understand how the reconstruction does
- DarkSHINE combines several approaches to ensure everything that needs to be saved, is
  - Initial Particle Steps
  - MC particles (*all* the secondary particles produced in one event)
  - Simulated Hits (Energy deposition in detectors, without digitization)
  - More truth information…
- At the end (in reconstruction) these different records are reconstructed back together into what ever is required

Short Summary on Simulation



Things to be determined through simulation

1. Generating **initial particle(s)** from particle gun or external generator
2. Particle will move ahead through the following criteria:
   - **Transportation + Physics processes**: EM process, hadronic process…
   - Roll a dice to determine which physics process happens → $\min(\ell_{p_1}, \ell_{p_2}, \ell_{p_3}, T)$
   - If there are **new secondaries** generated, **push back into stack**
   - Stop simulating this track until $E_k = 0$ or exit the world
3. Choose the next particle in stack, and repeat step 2
4. When the stack is empty, DONE 😍

- ℓ: Interaction length
- σ: raw cross section
- T: geometry boundary
- $p(\ell) = \sigma e^{-\sigma\ell}$

Things won't change

Magnetic Fields

Detector Construction

- Materials
- Geometry
- Electronics (SiPM, PMT…)

Truth? Real Data?

Truth

- Particle Gun/Generator
- Particle trajectories (track + step)
- Truth interaction information
- Truth energy deposition in detector

Real Data

- Optical photons in calorimeter
- Electrons in Silicon

Need hardware test to validate/calibrate

What is Geant4?

- Toolkit for the **Monte Carlo simulation** of the interaction of **particles** with **matter**
  - **Physics processes** (EM, hadronic, optical) cover a comprehensive set of particles, materials and over a wide energy range
  - offers a complete set of support functionalities (tracking, geometry)
  - **Distributed** software production and management: developed by an **international Collaboration**
    - Established in 1998
    - Approximately **100 members**, from Europe, America and Asia

- Written in **C++** language
  - Takes advantage from the **Object-Oriented** software technology

Get started

Everything you need to get started with Geant4.

I'm ready to start!

Download

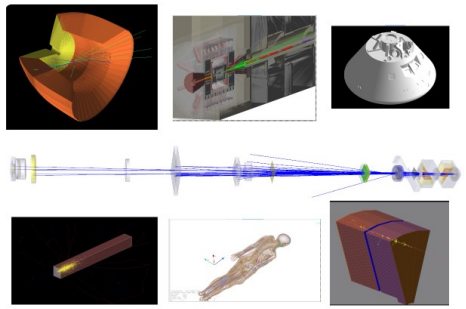
Geant4 source code and installers are available for download, with source code under an open source license.

Latest: 11.2.0

Docs

Documentation for Geant4, along with tutorials and guides, are available online.

Read documentation



About us

What is Geant4, where it's used, details on Collaboration.

Learn More

Collaboration

Geant4 team and documents

Learn More

Contribute

How external users can contribute to Geant4.

Learn More

Events

» More

1/14/2024	-	1/19/2024	11th International Geant4 School, University of Pavia, Pavia (Italy)
3/25/2024	-	3/29/2024	Geant4 tutorial at JLAB, Jefferson Lab, Newport News (Virginia, USA)
3/25/2024	-	3/26/2024	Geant4-DNA International tutorial, Osaka University, Osaka (Japan)
3/27/2024	-	3/29/2024	The 5th Geant4 International User Conference at the Physics - Medicine - Biology frontier, Osaka University Nakanoshima Center, Osaka (Japan)
10/7/2024	-	10/11/2024	29th Geant4 Collaboration Meeting, INFN LNS Laboratories, Catania (Italy)

- Website [link]
- **Code** and **documentation** available in the main web page
- Regular tutorial courses held worldwide

Geant4 Versions and Releases

- First release (Geant4 1.0) in December 1998
  - ~ Two releases per year since then
  - Major releases(**x**.y) or minor releases(x.**y**) or beta releases
  - Patches regularly issued
- Last version: **Geant4 11.2**
  - Released: December 8th, 2023
- Version used in DarkSHINE: **Geant4 10.6.3**
  - This is also the version installed in the docker used for this tutorial
  - Require C++17 standard

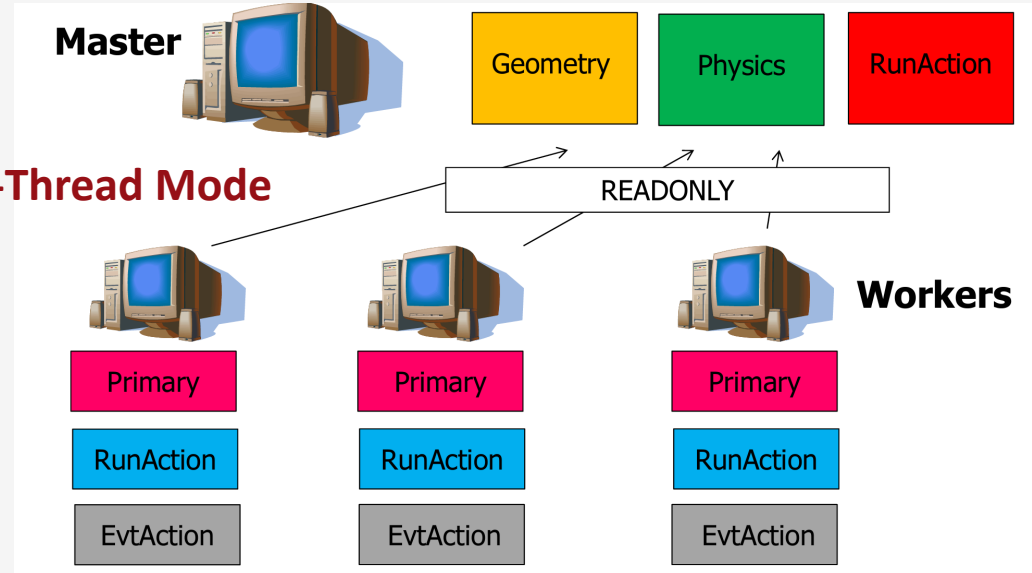
Toolkit and User Application

- Geant4 is a **toolkit** (= a collection of tools)
  - you cannot "run" it out of the box
  - You must write an application, which uses Geant4 tools
  - There are no such concepts as "Geant4 defaults"
  - You must provide the necessary information to configure your simulation
  - You must deliberately choose which Geant4 tools to use
- What you **MUST** do:
  - Describe your **experimental setup** (Detectors, magnetic fields…)
  - Provide the **primary particles** input to your simulation
  - Decide which **particles** and **physics models** you want to use out of those available in Geant4 and the precision of your simulation (cuts to produce and track secondary particles)
- You may also want
  - To interact with Geant4 kernel to **control** your simulation
  - To visualize your simulation configuration or results
  - To record the simulation results to disk for further analysis

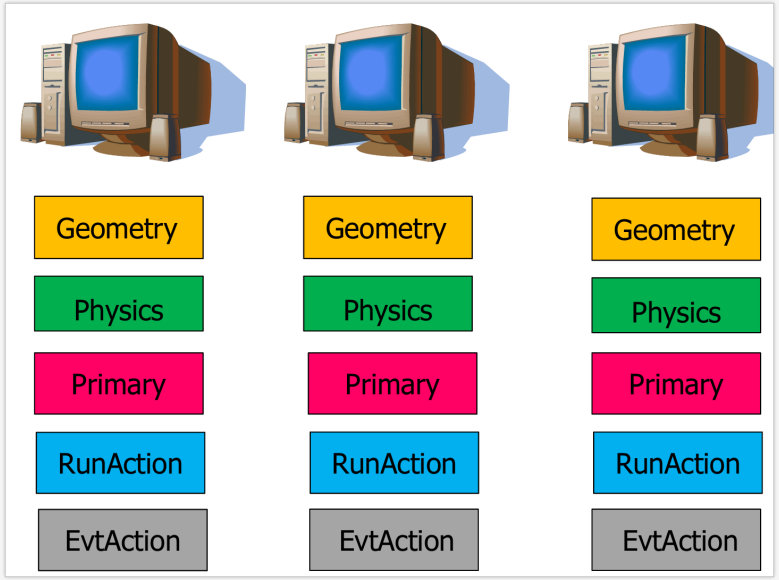
Geant4: Multi-Thread Modes

- Geant4 supports multi-thread approach for multi-core machines
  - Simulation is automatically split on an event-by-event basis
  - Different events are processed by different cores
  - Can fully profit of **all cores** available on modern machines → substantial **speed-up** of simulations
  - **Unique** copy (master) of geometry and physics: all cores have them as read-only (saves memory)
- Backwards compatible with the sequential mode

Multi-Thread Mode



Parallelization with Sequential Mode



Interaction with Geant4 kernel

- Geant4 design provides **tools** for a user **application**
  - To tell the **kernel** about your simulation configuration
  - To interact with Geant4 kernel itself
- Geant4 tools for user interaction are **base classes**
  - You create **your own concrete class** derived from the base classes → **interface** to the Geant4 kernel
  - Geant4 kernel handles your own derived classes transparently through their base class interface (polymorphism)

Two types of Geant4 base classes:

- **Abstract base classes** for user interaction (classes starting with G4V)
  - User derived concrete classes are **mandatory**
  - User to implement the purely virtual methods
- **Concrete base classes** (with virtual dummy default methods) for user interaction
  - User derived classes are **optional**

How Geant4 Works – Glossary

- **Run:** a simulation session. It begins with the initialization of the simulation environment and ends with the accumulation of data. During a run, a specified number of events are simulated.
- **Event:** a single instance of the simulation process in one Run. Each event starts with primary particles (defined by the user) and tracks their interactions and decays.
- **Track:** the representation of a particle. It includes the particle's type, energy, position, momentum, and other physical properties. Tracks are created for **primary particles** and for **secondary particles generated through interactions**.
- **Step:** a segment of a track. It represents **the particle's movement from one point to another** and includes any physical interactions that occur during this movement. Each step is characterized by the particle's state at the beginning and end of the step, as well as the energy loss and other changes that occur.
- **Stack:** the data structure used to manage the tracks of particles (**a to-do list**). When a new particle is created (either a primary particle or a secondary particle from an interaction), it is placed on the stack.

User Classes

Initialization classes

Invoked at the initialization

- G4VUserDetectorConstruction [How]
- G4VUserPhysicsList [How]
- G4VUserActionInitialization

Global: only one instance of them exists in memory, shared by all threads (**read-only**). Managed only by the **master** thread.

Action classes

Invoked during the execution loop

- G4VUserPrimaryGeneratorAction
- G4UserRunAction
- G4UserEventAction
- G4UserStackingAction
- G4UserTrackingAction
- G4UserSteppingAction

Type 1: The Mandatory User Classes

- **G4VUserDetectorConstruction**: describe the experimental set-up, including the geometry, material, sensitive detector, magnetic fields.
- **G4VUserPhysicsList**: select the physics you want to activate (normally from the pre-defined physics list [link])
- **G4VUserActionInitialization**: takes care of the user action initializations
- **G4VUserPrimaryGeneratorAction**: define the initial particles, from particle gun, **general particle source (GPS)**, or external file

- **Abstract base classes** for user interaction (classes starting with G4V)
  - User derived concrete classes are **mandatory**
  - User to implement the purely virtual methods

Type 2: Optional User Classes

- Five **concrete base classes** whose virtual member functions the user may override to gain control of the simulation at various stages
  - G4UserRunAction: beginning of the **run** & end of the **run**
  - G4UserEventAction: beginning of the **event** & end of the **event**
  - G4UserTrackingAction: beginning of the **track** & end of the **track**
  - G4UserSteppingAction: end of each **step** of a particle's track
  - G4UserStackingAction: when a new track has been generated (*not covered in this course*)
- Each member function of the base classes has a dummy implementation (not purely virtual)
  - Empty implementation: **does nothing**
  - Override only the methods that you need
- User action classes must be **registered** to the Run Manager via the *G4VUserActionInizialization*

How to Build a Detector?

Geant4 Official User Book [link]

Units in Geant4

- Don’t use default units! ~~[mm, ns, MeV]~~
- When specifying dimensions, always multiply by an appropriate unit:

```
1 G4double width = 12.5 * m;
2 G4double density = 2.7 * g/cm3;
```

- Most common units are defined in CLHEP library (included in Geant4):

```
1 #include "G4SystemOfUnits.hh"
2 // or
3 #inlcude "CLHEP/SystemOfUnits.h"
```

- Output data in terms of a specific unit (divide a value by the unit):

```
1 G4cout << dE / MeV << " (MeV)" << G4endl;
```

- **Useful feature**: Geant4 can select the most appropriate unit to use:

```
1 G4cout << G4BestUnit(StepSize, "Length");
```

```
//
// Time [T]
//
static const double nanosecond  = 1.;
static const double second      = 1.e+9 *nanosecond;
static const double millisecond = 1.e-3 *second;
static const double microsecond = 1.e-6 *second;
static const double  picosecond = 1.e-12*second;

static const double hertz = 1./second;
static const double kilohertz = 1.e+3*hertz;
static const double megahertz = 1.e+6*hertz;

// symbols
static const double ns = nanosecond;
static const double  s = second;
static const double ms = millisecond;

//
// Electric charge [Q]
//
static const double eplus = 1. ;// positron charge
static const double e_SI  = 1.602176487e-19;// positron charge in coulomb
static const double coulomb = eplus/e_SI;// coulomb = 6.24150 e+18 * eplus

//
// Energy [E]
//
static const double megaelectronvolt = 1. ;
static const double     electronvolt = 1.e-6*megaelectronvolt;
static const double kiloelectronvolt = 1.e-3*megaelectronvolt;
static const double gigaelectronvolt = 1.e+3*megaelectronvolt;
static const double teraelectronvolt = 1.e+6*megaelectronvolt;
static const double petaelectronvolt = 1.e+9*megaelectronvolt;

static const double joule = electronvolt/e_SI;// joule = 6.24150 e+12 * MeV

// symbols
static const double MeV = megaelectronvolt;
static const double  eV = electronvolt;
static const double keV = kiloelectronvolt;
static const double GeV = gigaelectronvolt;
static const double TeV = teraelectronvolt;
static const double PeV = petaelectronvolt;
```

Materials in Geant4

- Different levels of material description:
  - Isotopes → G4Isotope
  - Elements → G4Element

  } Using G4Isotope and G4Element to build **G4Material**

  - Molecules, compounds and mixtures → **G4Material** (G4Material is all you need 🙃)
- Attributes associated:
  - G4Isotope and G4Element describe properties of the atoms: Atomic number, number of nucleons, mass of a mole, shell energies, cross-sections per atoms, etc.
  - **G4Material** describes the macroscopic properties of the matter: Temperature, pressure, state, density, radiation length, absorption length, etc.
- G4Material is used by **tracking**, **geometry** and **physics**
- Materials in Users Guide [link]

Elements and compounds

```
G4Material (const G4String &name, G4double z, G4double a, G4double density, G4State state=kStateUndefined, G4double temp=CLHEP::STP_Temperature, G4double pressure=CLHEP::STP_Pressure)
G4Material (const G4String &name, G4double density, G4int nComponents, G4State state=kStateUndefined, G4double temp=CLHEP::STP_Temperature, G4double pressure=CLHEP::STP_Pressure)
G4Material (const G4String &name, G4double density, const G4Material *baseMaterial, G4State state=kStateUndefined, G4double temp=CLHEP::STP_Temperature, G4double pressure=CLHEP::STP_Pressure)
```

- Single-element material:

```
1 G4double density = 1.390*g/cm3;
2 G4double a = 39.95*g/mole;
3 G4double z = 18;
4 G4Material* lAr = new G4Material("liquidAr", z, a, density);
```

- Molecule material (composition by number of atoms):

```
 1 a = 1.01*g/mole;
 2 G4Element* elH = new G4Element("Hydrogen", symbol="H", z=1., a);
 3
 4 a = 16.00*g/mole;
 5 G4Element* elO = new G4Element("Oxygen", symbol="O", z=8., a);
 6
 7 density = 1.000*g/cm3;
 8
 9 G4Material* H2O = new G4Material("Water", density, ncomponents=2);
10
11 H2O->AddElement(elH, natoms=2);
12 H2O->AddElement(elO, natoms=1);
```

Mixtures

- Composition by fraction of mass:

```
a = 14.01*g/mole;
G4Element* elN  = new G4Element(name="Nitrogen",symbol="N", z= 7., a);

a = 16.00*g/mole;
G4Element* elO  =  new G4Element(name="Oxygen",symbol="O", z= 8., a);

density = 1.290*mg/cm3;
G4Material* Air = new G4Material(name="Air", density, ncomponents=2);

Air->AddElement(elN, 70.0*perCent);
Air->AddElement(elO, 30.0*perCent);
```

- Composition of mixtures:

```
G4Element* elC  = …;    // define “carbon” element
G4Material* SiO2 = …;   // define “quartz” material
G4Material* H2O = …;    // define “water” material

density = 0.200*g/cm3;
G4Material* aerogel = new G4Material("Aerogel", density, ncomponents=3);

aerogel->AddMaterial(SiO2,fractionmass=62.5*perCent);
aerogel->AddMaterial(H2O, fractionmass=37.4*perCent);
aerogel->AddElement (elC, fractionmass= 0.1*perCent);
```

NIST material database

- No need to predefine elements and materials
- Retrieve elements and materials from NIST manager:

```
1 #include “G4NistManager.hh”    Include header file for G4NistManager
2
3 G4NistManager* manager = G4NistManager::Instance();  Singleton
4
5 G4Material* H2O =    manager->FindOrBuildMaterial("G4_WATER");
6 G4Material* air =    manager->FindOrBuildMaterial("G4_AIR");
7 G4Material* vacuum = manager->FindOrBuildMaterial("G4_Galactic");
8 G4Element* Si = manager->FindOrBuildElement("Si");
```

How to Build a Detector Geometry – Glossary

- **Solid:** a basic shape. It defines the **size** and **shape** of an object but doesn't have any physical properties like material or color.
- **Logical Volume:** Once you have a solid (your shape), the next step is to define **what it is made of** and how it behaves. It's giving **material**, **properties, hierarchy of volumes, and magnetic field** to the shape, but it's still *not a real, physical* one.
- **Physical Volume:** Finally, the 'physical volume' is like the actual object placed somewhere. It takes the **shape** (**solid**) and the **material** (**logical volume**), and places it in the simulation world. This is where you decide the **position** and how many **copies of the physical volume** you want.

Solid: size & shape

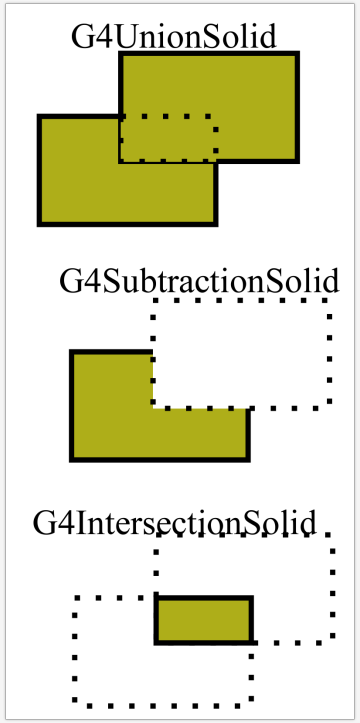
Logical Volume: material & properties

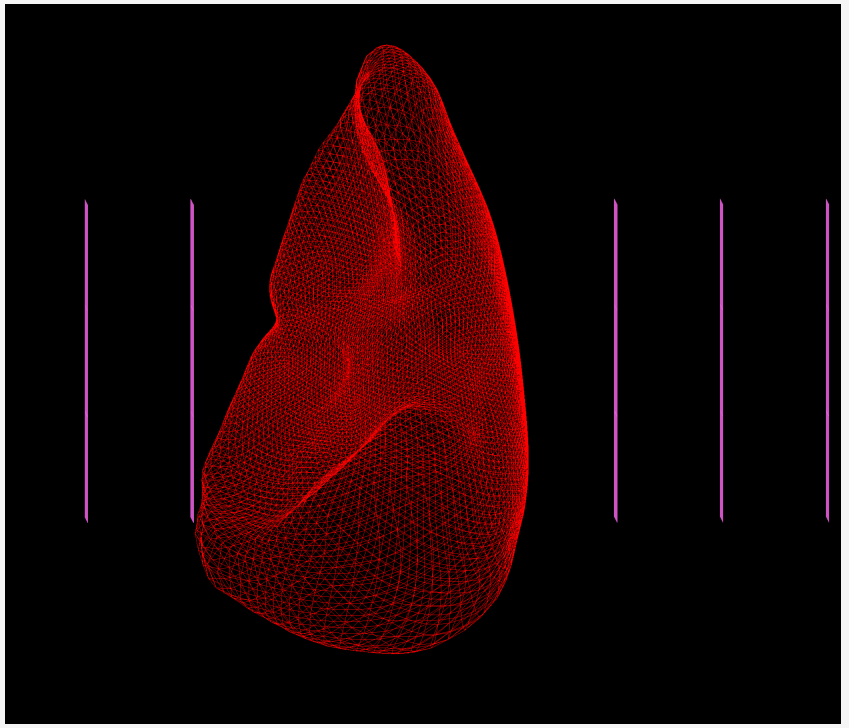
Physical Volume:
actual placement & copies

Define Solid

- CSG (Constructed Solid Geometry) solids
  - G4Box, G4Tubs, G4Cons, G4Trd, …
- Specific solids (CSG like)
  - G4Polycone, G4Polyhedra, G4Hype, …
  - G4TwistedTubs, G4TwistedTrap, …
- BREP (Boundary REPresented) solids
  - G4BREPSolidPolycone, G4BSplineSurface, …
  - Any order surface
- Boolean solids
  - G4UnionSolid, G4SubtractionSolid, …

```
1   G4double world_sizeXY = 1.2*env_sizeXY;
2   G4double world_sizeZ  = 1.2*env_sizeZ;
3
4   G4Box* solidWorld =
5     new G4Box(
6         "World",
7         0.5*world_sizeXY,  // x half size
8         0.5*world_sizeXY,  // y half size
9         0.5*world_sizeZ    // z half size
10    );
```





Define Logical Volume

- Contains all information of volume except position:
  - **Shape** and **dimension** (G4VSolid)
  - **Material**, Sensitive Detector, visualization attributes
  - Position of **daughter** volumes
  - **Magnetic field**, User limits
- Physical volumes of same type **can share** a logical volume.
- The pointers to solid and material must be **not nullptr**

```
1 G4LogicalVolume(G4VSolid* pSolid,
2                 G4Material* pMaterial,
3                 const G4String& name,
4                 G4FieldManager* pFieldMgr=0,
5                 G4VSensitiveDetector* pSDetector=0,
6                 G4UserLimits* pULimits=0,
7                 G4bool optimise=true);
```

Optional

Define Physical Volume

- A physical volume is a positioned instance of a logical volume inside another logical volume (**the mother volume**)
- **Never Overlap with other logical volumes**
- Repeated: a logical volume placed many times can represent any number of volumes reduces use of memory
- G4PVReplica (= simple repetition), G4PVParameterised (= more complex pattern), etc.

```
1  new G4PVPlacement(0,                    //no rotation
2                    G4ThreeVector(),      //at (0,0,0)
3                    logicEnv,             //its logical volume
4                    "Envelope",           //its name
5                    logicWorld,           //its mother  volume
6                    false,                //no boolean operation
7  Unique Number     0,                    //copy number
8                    checkOverlaps);       //overlaps checking
```

Geometry Hierarchy

- A volume is placed in its mother volume
  - Position and rotation of the daughter volume is described with respect to the **local coordinate system** of the mother volume
  - The origin of the mother's local coordinate system is at the **center** of the mother volume
  - Daughter volumes **cannot protrude** from the mother volume
  - Daughter volumes **cannot overlap**
  - The logical volume of mother knows the daughter volumes it contains
- One logical volume can be placed **more than once**. One or more volumes can be placed in a mother volume.
- The mother-daughter relationship is an information of ***G4LogicalVolume***. If the mother volume is placed **more than once**, all daughters by definition appear in each placed physical volume.
- The **world volume** must be a unique physical volume which fully contains all other volumes (root volume of the hierarchy)
- The world volume defines the **global coordinate system**. The origin of the global coordinate system is at the center of the world volume.

How to Access Simulation Data?

Geant4 Official User Book [link]

What is Sensitive Detector?

- We know now how to create a detector, but how to record data from detector?
  - energy deposition, number of particles, particle trajectories, etc.
- Different methods to extract useful information from simulation:
  - **Sensitive Detector** (usually record information in the detector, energy deposition)
  - User Actions (usually record truth MC particles, *TrackingAction*, *SteppingAction*)
  - Scoring (not covered in this course)
- A SD can be used to simulate the "**read-out**" of your detector:
  - It is a way to declare a geometric element "**sensitive**" to the passage of particles
  - It gives the user a handle to collect quantities from these elements. For example: energy deposited, position, time information
- **Using SD to record hit is better than using User Stepping Action:**
  - Generally more efficient
  - Geant4 **only** records hits when a particle **interacts with the sensitive material**, reducing the computational load

Sensitive Detector: Tracker & Calorimeter

- A **tracker** detector typically generates **a hit for every single step of every single (charged) track**.
  - A tracker hit typically contains:
    - Position and time
    - Energy deposition of the step
    - Track ID
- A **calorimeter** detector typically generates a hit for every cell and **accumulates energy deposition in each cell for all steps of all tracks**.
  - A calorimeter hit typically contains
    - Sum of deposited energy
    - Cell ID

More information on Sensitive Detector

- Each Logical Volume can have a pointer to a sensitive detector.
  - Then this volume becomes sensitive.
- **A sensitive detector creates hit(s)** using the information given in G4Step object. The user has to provide his/her own implementation of the detector response.
  - *UserSteppingAction* class **should NOT** do this.
- Hit objects, which are still the user's class objects, are collected in a G4Event object at the end of an event.

How to Define Sensitive Detector?

- Sensitive detector is a user-defined class derived from ***G4VSensitiveDetector***

```
 1 #include "G4VSensitiveDetector.hh"
 2 #include "Hit.hh"
 3
 4 class SensitiveDetector : public G4VSensitiveDetector {
 5   public:
 6   SensitiveDetector(G4String SDname);
 7   ~SensitiveDetector();
 8
 9   G4bool ProcessHits(G4Step *step, G4TouchableHistory *ROhist);
10
11   void Initialize(G4HCofThisEvent* HCE);
12
13   void EndOfEvent(G4HCofThisEvent* HCE);
14
15   private:
16   SDHitCollection* fHitCollection;
17 };
```

Call for every particle step in the sensitive volume

Call at the beginning of each event

Call at the end of each event

Think of your own way to record Hits

How to build an application

Geant4 Official Example: B1 [link]

Code on GitHub [link], GitLab [link]

What is Example B1 doing?

- This example demonstrates a simple (**medical**) application within which users will familiarize themselves with simple placement, use the NIST material database, and can utilize electromagnetic and/or hadronic physics processes. Two items of information are collected in this example: the energy deposited and the total dose for a selected volume.

Main Program

```
1  int main(int argc,char** argv)
2  {
3    G4UIExecutive* ui = 0;
4    if ( argc == 1 ) {
5      ui = new G4UIExecutive(argc, argv);
6    }
7
8    G4RunManager* runManager = new G4RunManager;
9
10   runManager->SetUserInitialization(new B1DetectorConstruction());
11
12   G4VModularPhysicsList* physicsList = new QBBC;
13   physicsList->SetVerboseLevel(1);
14   runManager->SetUserInitialization(physicsList);
15
16   runManager->SetUserInitialization(new B1ActionInitialization());
17
18   G4VisManager* visManager = new G4VisExecutive;
19   visManager->Initialize();
20
21   G4UImanager* UImanager = G4UImanager::GetUIpointer();
22
23   if ( ! ui ) {
24     // batch mode
25     G4String command = "/control/execute ";
26     G4String fileName = argv[1];
27     UImanager->ApplyCommand(command+fileName);
28   }
29   else {
30     // interactive mode
31     UImanager->ApplyCommand("/control/execute init_vis.mac");
32     ui->SessionStart();
33     delete ui;
34   }
35
36   delete visManager;
37   delete runManager;
38 }
```

Detect interactive mode (if no arguments) and define UI session

G4RunManager: a manager class to control all actions

Register Detector Construction

Register Physics List, using QBBC (pre-defined)

Register all optional user action classes here

Initialize visualization

not mandatory, but it's better to visualize your program and easy to debug

Just copy 😝

But remember to copy the *init_vis.mac* file to your run directory

Job Termination

Free the store: user actions, physics_list and detector_description are owned and deleted by the run manager, so they should not be deleted in the main() program !

Detector Construction (header file)

```
1  #ifndef B1DetectorConstruction_h
2  #define B1DetectorConstruction_h 1
3
4  #include "G4VUserDetectorConstruction.hh"
5  #include "globals.hh"
6
7  class G4VPhysicalVolume;
8  class G4LogicalVolume;
9
10 class B1DetectorConstruction : public G4VUserDetectorConstruction
11 {
12   public:
13     B1DetectorConstruction();
14     virtual ~B1DetectorConstruction();
15
16     virtual G4VPhysicalVolume* Construct();
17
18     G4LogicalVolume* GetScoringVolume() const { return fScoringVolume; }
19
20   protected:
21     G4LogicalVolume*  fScoringVolume;
22 };
23
24 #endif
```

Derive your own concrete class from ***G4VUserDetectorConstruction*** abstract base class

Implementing the pure virtual method ***Construct()***:

- Define shapes/solids required to describe the geometry
- Construct all necessary materials
- Construct and place volumes of your detector geometry
- (Define "sensitivity" properties associated to volumes)
- (Associate magnetic field to detector regions)
- (Define visualization attributes for the detector elements)

Detector Construction (source file)

```cpp
B1DetectorConstruction::B1DetectorConstruction()
: G4VUserDetectorConstruction(), fScoringVolume(0) { }

B1DetectorConstruction::~B1DetectorConstruction() { }

G4VPhysicalVolume* B1DetectorConstruction::Construct()
{
  G4NistManager* nist = G4NistManager::Instance();

  G4double env_sizeXY = 20*cm, env_sizeZ = 30*cm;
  G4Material* env_mat = nist->FindOrBuildMaterial("G4_WATER");

  // Option to switch on/off checking of volumes overlaps
  G4bool checkOverlaps = true;
```

Define World

```cpp
  // World
  G4double world_sizeXY = 1.2*env_sizeXY;
  G4double world_sizeZ  = 1.2*env_sizeZ;
  G4Material* world_mat = nist->FindOrBuildMaterial("G4_AIR");

  G4Box* solidWorld = new G4Box("World", 0.5*world_sizeXY, 0.5*world_sizeXY, 0.5*world_sizeZ);

  G4LogicalVolume* logicWorld = new G4LogicalVolume(solidWorld, world_mat, "World");

  G4VPhysicalVolume* physWorld =
    new G4PVPlacement(0,                    //no rotation
                      G4ThreeVector(),      //at (0,0,0)
                      logicWorld,           //its logical volume
                      "World",              //its name
                      0,                    //its mother  volume
                      false,                //no boolean operation
                      0,                    //copy number
                      checkOverlaps);       //overlaps checking
```

```cpp
  // Envelope
  G4Box* solidEnv = new G4Box("Envelope", 0.5*env_sizeXY, 0.5*env_sizeXY, 0.5*env_sizeZ);

  G4LogicalVolume* logicEnv = new G4LogicalVolume(solidEnv, env_mat, "Envelope");

  new G4PVPlacement(0,                    //no rotation
                    G4ThreeVector(),      //at (0,0,0)
                    logicEnv,             //its logical volume
                    "Envelope",           //its name
                    logicWorld,           //its mother  volume
                    false,                //no boolean operation
                    0,                    //copy number
                    checkOverlaps);       //overlaps checking
```

Define Envelope

```cpp
  //...

  return physWorld;
}
```

Always return the physical volume of the world

Define 2 shapes

```cpp
// Shape 1
G4Material* shape1_mat = nist->FindOrBuildMaterial("G4_A-150_TISSUE");
G4ThreeVector pos1 = G4ThreeVector(0, 2*cm, -7*cm);

// Conical section shape
G4double shape1_rmina =  0.*cm, shape1_rmaxa = 2.*cm;
G4double shape1_rminb =  0.*cm, shape1_rmaxb = 4.*cm;
G4double shape1_hz = 3.*cm;
G4double shape1_phimin = 0.*deg, shape1_phimax = 360.*deg;
G4Cons* solidShape1 =
  new G4Cons("Shape1",
  shape1_rmina, shape1_rmaxa, shape1_rminb, shape1_rmaxb, shape1_hz,
  shape1_phimin, shape1_phimax);

G4LogicalVolume* logicShape1 =
  new G4LogicalVolume(solidShape1,         //its solid
                      shape1_mat,          //its material
                      "Shape1");           //its name

new G4PVPlacement(0,                       //no rotation
                  pos1,                    //at position
                  logicShape1,             //its logical volume
                  "Shape1",                //its name
                  logicEnv,                //its mother  volume
                  false,                   //no boolean operation
                  0,                       //copy number
                  checkOverlaps);          //overlaps checking

// Shape 2
G4Material* shape2_mat = nist->FindOrBuildMaterial("G4_BONE_COMPACT_ICRU");
G4ThreeVector pos2 = G4ThreeVector(0, -1*cm, 7*cm);

// Trapezoid shape
G4double shape2_dxa = 12*cm, shape2_dxb = 12*cm;
G4double shape2_dya = 10*cm, shape2_dyb = 16*cm;
G4double shape2_dz  = 6*cm;
G4Trd* solidShape2 =
  new G4Trd("Shape2",                      //its name
            0.5*shape2_dxa, 0.5*shape2_dxb,
            0.5*shape2_dya, 0.5*shape2_dyb, 0.5*shape2_dz); //its size

G4LogicalVolume* logicShape2 =
  new G4LogicalVolume(solidShape2,         //its solid
                      shape2_mat,          //its material
                      "Shape2");           //its name

new G4PVPlacement(0,                       //no rotation
                  pos2,                    //at position
                  logicShape2,             //its logical volume
                  "Shape2",                //its name
                  logicEnv,                //its mother  volume
                  false,                   //no boolean operation
                  0,                       //copy number
                  checkOverlaps);          //overlaps checking

// Set Shape2 as scoring volume
fScoringVolume = logicShape2;
```

Define Scoring (?) Volume

Action Initialization

```
#ifndef B1ActionInitialization_h
#define B1ActionInitialization_h 1

#include "G4VUserActionInitialization.hh"

/// Action initialization class
class B1ActionInitialization : public G4VUserActionInitialization
{
  public:
    B1ActionInitialization();
    virtual ~B1ActionInitialization();

    virtual void BuildForMaster() const;
    virtual void Build() const;
};

#endif
```

dummy implementation, defined by user if needed

```
B1ActionInitialization::B1ActionInitialization()
 : G4VUserActionInitialization()
{}


B1ActionInitialization::~B1ActionInitialization()
{}

void B1ActionInitialization::BuildForMaster() const
{
  B1RunAction* runAction = new B1RunAction;
  SetUserAction(runAction);
}

void B1ActionInitialization::Build() const
{
  SetUserAction(new B1PrimaryGeneratorAction);

  B1RunAction* runAction = new B1RunAction;
  SetUserAction(runAction);

  B1EventAction* eventAction = new B1EventAction(runAction);
  SetUserAction(eventAction);

  SetUserAction(new B1SteppingAction(eventAction));
}
```

Define Primary Generator, Run, Event, Stepping Action

Event Action

```
#ifndef B1EventAction_h
#define B1EventAction_h 1

#include "G4UserEventAction.hh"
#include "globals.hh"

class B1RunAction;

class B1EventAction : public G4UserEventAction
{
  public:
    B1EventAction(B1RunAction* runAction);
    virtual ~B1EventAction();

    virtual void BeginOfEventAction(const G4Event* event);
    virtual void EndOfEventAction(const G4Event* event);

    void AddEdep(G4double edep) { fEdep += edep; }

  private:
    B1RunAction* fRunAction;
    G4double     fEdep;
};

#endif
```

What to do in the beginning/end of the event

```
#include "B1EventAction.hh"
#include "B1RunAction.hh"

#include "G4Event.hh"
#include "G4RunManager.hh"

B1EventAction::B1EventAction(B1RunAction* runAction)
: G4UserEventAction(),
  fRunAction(runAction),
  fEdep(0.)
{}

B1EventAction::~B1EventAction()
{}

void B1EventAction::BeginOfEventAction(const G4Event*)
{
  fEdep = 0.;
}

void B1EventAction::EndOfEventAction(const G4Event*)
{
  // accumulate statistics in run action
  fRunAction->AddEdep(fEdep);
}
```

Stepping Action

```
#ifndef B1SteppingAction_h
#define B1SteppingAction_h 1

#include "G4UserSteppingAction.hh"
#include "globals.hh"

class B1EventAction;

class G4LogicalVolume;

/// Stepping action class
class B1SteppingAction : public G4UserSteppingAction
{
  public:
    B1SteppingAction(B1EventAction* eventAction);
    virtual ~B1SteppingAction();

    // method from the base class
    virtual void UserSteppingAction(const G4Step*);

  private:
    B1EventAction*  fEventAction;
    G4LogicalVolume* fScoringVolume;
};

#endif
```

Make sure we are in the region of interest (Scoring Volume)

Record the energy deposition in this step for this event

```
B1SteppingAction::B1SteppingAction(B1EventAction* eventAction)
: G4UserSteppingAction(),
  fEventAction(eventAction),
  fScoringVolume(0)
{}


B1SteppingAction::~B1SteppingAction()
{}


void B1SteppingAction::UserSteppingAction(const G4Step* step)
{
  if (!fScoringVolume) {
    const B1DetectorConstruction* detectorConstruction
      = static_cast<const B1DetectorConstruction*>
        (G4RunManager::GetRunManager()->GetUserDetectorConstruction());
    fScoringVolume = detectorConstruction->GetScoringVolume();
  }

  // get volume of the current step
  G4LogicalVolume* volume
    = step->GetPreStepPoint()->GetTouchableHandle()
      ->GetVolume()->GetLogicalVolume();

  // check if we are in scoring volume
  if (volume != fScoringVolume) return;

  // collect energy deposited in this step
  G4double edepStep = step->GetTotalEnergyDeposit();
  fEventAction->AddEdep(edepStep);
}
```

Primary Generator Action

```
#ifndef B1PrimaryGeneratorAction_h
#define B1PrimaryGeneratorAction_h 1

#include "G4VUserPrimaryGeneratorAction.hh"
#include "G4ParticleGun.hh"
#include "globals.hh"

class G4ParticleGun;
class G4Event;
class G4Box;

class B1PrimaryGeneratorAction : public G4VUserPrimaryGeneratorAction
{
  public:
    B1PrimaryGeneratorAction();
    virtual ~B1PrimaryGeneratorAction();

    // method from the base class
    virtual void GeneratePrimaries(G4Event*);

    // method to access particle gun
    const G4ParticleGun* GetParticleGun() const { return fParticleGun; }

  private:
    G4ParticleGun*  fParticleGun; // pointer a to G4 gun class
    G4Box* fEnvelopeBox;
};

#endif
```

Define default initial particle:

- 1 photon with energy 6 MeV along +z direction

Need users to implement

```
B1PrimaryGeneratorAction::B1PrimaryGeneratorAction()
: G4VUserPrimaryGeneratorAction(),
  fParticleGun(0),
  fEnvelopeBox(0)
{
  G4int n_particle = 1;
  fParticleGun  = new G4ParticleGun(n_particle);

  // default particle kinematic
  G4ParticleTable* particleTable = G4ParticleTable::GetParticleTable();
  G4String particleName;
  G4ParticleDefinition* particle
    = particleTable->FindParticle(particleName="gamma");
  fParticleGun->SetParticleDefinition(particle);
  fParticleGun->SetParticleMomentumDirection(G4ThreeVector(0.,0.,1.));
  fParticleGun->SetParticleEnergy(6.*MeV);
}

B1PrimaryGeneratorAction::~B1PrimaryGeneratorAction()
{
  delete fParticleGun;
}

void B1PrimaryGeneratorAction::GeneratePrimaries(G4Event* anEvent)
{
  G4double envSizeXY = 0;
  G4double envSizeZ = 0;

  if (!fEnvelopeBox)
  {
    G4LogicalVolume* envLV
      = G4LogicalVolumeStore::GetInstance()->GetVolume("Envelope");
    if ( envLV ) fEnvelopeBox = dynamic_cast<G4Box*>(envLV->GetSolid());
  }

  if ( fEnvelopeBox ) {
    envSizeXY = fEnvelopeBox->GetXHalfLength()*2.;
    envSizeZ = fEnvelopeBox->GetZHalfLength()*2.;
  }
  else  {
    G4ExceptionDescription msg;
    msg << "Envelope volume of box shape not found.\n";
    msg << "Perhaps you have changed geometry.\n";
    msg << "The gun will be place at the center.";
    G4Exception("B1PrimaryGeneratorAction::GeneratePrimaries()",
     "MyCode0002",JustWarning,msg);
  }

  G4double size = 0.8;
  G4double x0 = size * envSizeXY * (G4UniformRand()-0.5);
  G4double y0 = size * envSizeXY * (G4UniformRand()-0.5);
  G4double z0 = -0.5 * envSizeZ;

  fParticleGun->SetParticlePosition(G4ThreeVector(x0,y0,z0));

  fParticleGun->GeneratePrimaryVertex(anEvent);
}
```

Make sure the envelope is defined, and get its boundary

Randomly generate particles according to the size of envelope

Run Action

```
#ifndef B1RunAction_h
#define B1RunAction_h 1

#include "G4UserRunAction.hh"
#include "G4Accumulable.hh"
#include "globals.hh"

class G4Run;

/// Run action class
///
/// In EndOfRunAction(), it calculates the dose in the selected volume
/// from the energy deposit accumulated via stepping and event actions.
/// The computed dose is then printed on the screen.

class B1RunAction : public G4UserRunAction
{
  public:
    B1RunAction();
    virtual ~B1RunAction();

    // virtual G4Run* GenerateRun();
    virtual void BeginOfRunAction(const G4Run*);
    virtual void   EndOfRunAction(const G4Run*);

    void AddEdep (G4double edep);

  private:
    G4Accumulable<G4double> fEdep;
    G4Accumulable<G4double> fEdep2;
};

#endif
```

What to do in the beginning/end of the run

How about the source file?