GEANT4
A SIMULATION TOOLKIT

Introduction

Mihaly Novak (CERN, EP-SFT)

Getting Started with **`Geant4`** at CERN, Geneva (Switzerland)

- What is **`Geant4`**?
- What is our goal during this course?
- Documentation and installation

- Key concepts of **`Geant4`** tracking
- The main: user initialisation and mandatory actions
- Our step-by-step plan

Introduction

WHAT IS GEANT4?

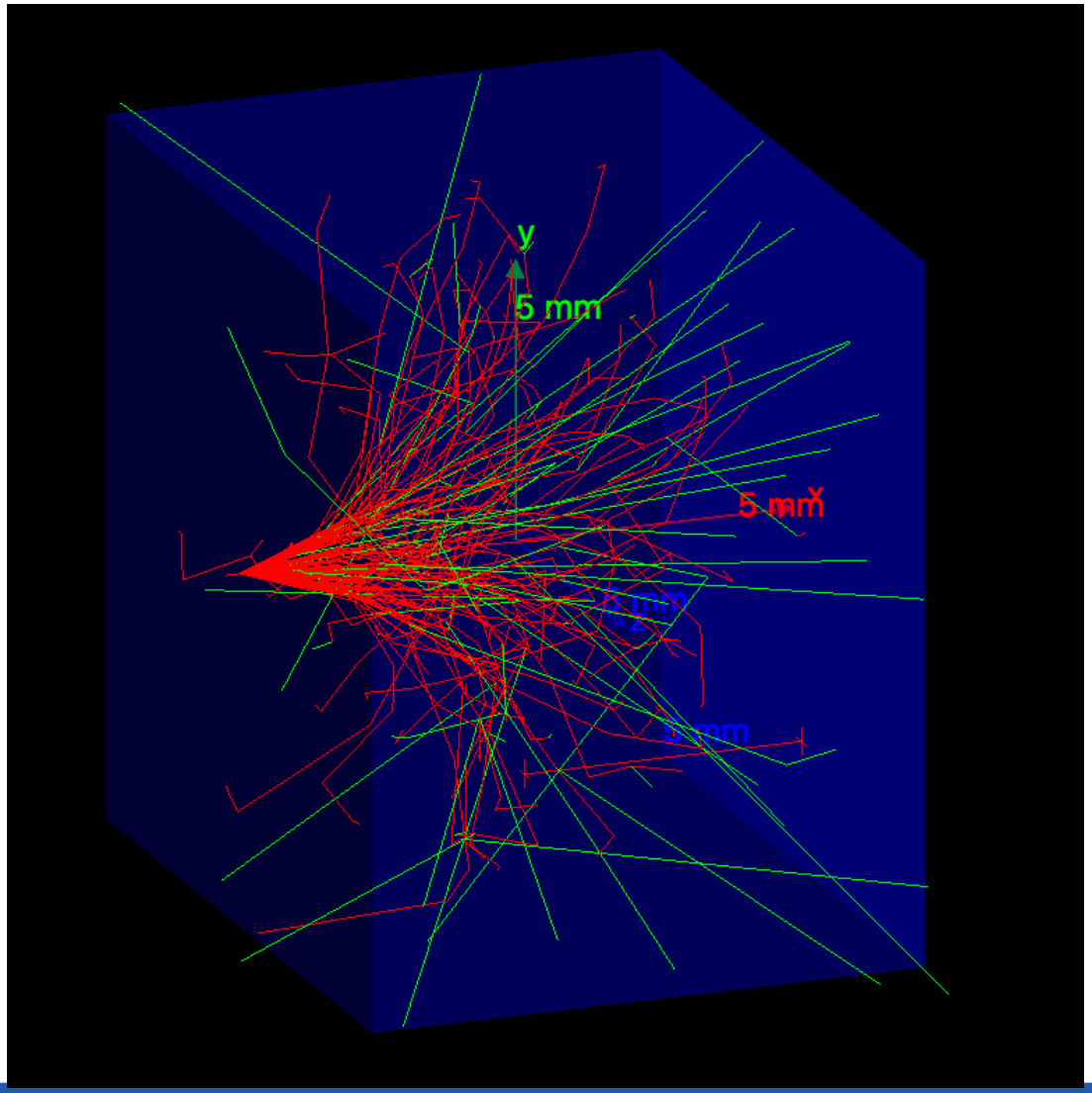
- **`Geant4` is a toolkit:**
  - for simulating the passage of particles through matter
  - toolkit i.e. there here is no main program
  - provides all the necessary **components** needed **to describe and to solve** particle transport **simulation** problems
  - **problem definitions/description:** geometry, particles, physics, etc.
  - **problem solution**: step-by-step particle transport computation
  - while providing **interaction point**s for the user
- **Toolkit vs program?**
  - as a toolkit, **`Geant4`** does **not** provide **a** main **program**
  - each simulation problem requires different configuration (geometry, scoring, particles, physics, etc..) that the user needs to define
  - **`Geant4`**, as a **toolkit**, provides all the necessary **components** in form of **interfaces** (called actions in **`Geant4`** terminology but see soon)

WHAT IS OUR GOAL DURING THIS COURSE?

- **Goal:**
  - introduce all the **mandatory** and some of the important **optional components** that needs to/can be utilised when writing a **simulation application** based on the **`Geant4`** simulation toolkit
  - show source of information that can be useful when writing such an application (documentation, **`Geant4`** source code, etc..)
  - become familiar with the *best practice* when developing your own application
- **How?**
  - we will write a **`Geant4`** simulation application together from scratch
  - the application:
    - **setup**: a simple box target with configurable thickness and material hit by a configurable particle source (see the next slide)
    - **goal**: collect information regarding the energy deposit in the target
  - we will do **mainly coding instead of lectures** with short explanations
  - more and more functionality will be added when time goes
  - both the agenda and the final state of our application is flexible

What is our goal during this course?

4 [MeV] electrons in Silicon (1 [cm])



Introduction

DOCUMENTATION

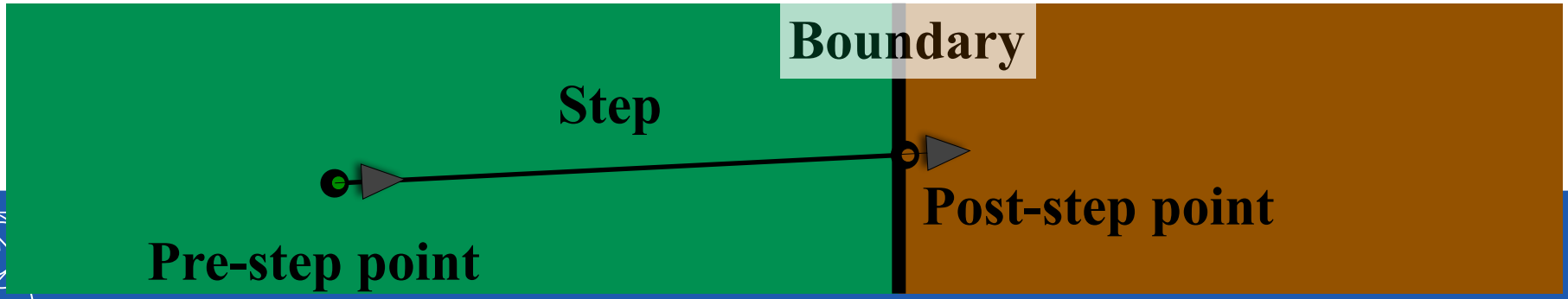
- **Documentation:**
  - all documentation can be found at the Geant4 webpage under the **User Support** menu
  - the **Book For Application Developers** will be our main source of documentation in the next 3 days
  - it is important that all of **you have a working version of** the **`Geant4`** toolkit available on your machine (the VM is the preferred one) before we go any further:
    - we will have a look to the **Installation Guide** now together
    - build/try one of the example applications that Geant4 provides (`/examples/basic/B1`)
    - inspect the installation directory structure (and understand the role of `-DGeant4_DIR` **`Geant4`** `cmake` variable with a simple example)
    - let's do it by ourself… :-)

Introduction

KEY CONCEPTS OF GEANT4 TRACKING

- **G4Track:**
  - a **G4Track** object represents/describes **the state of a particle** that is under simulation in a given instant of the time (i.e. **a** given **time point**)
  - a snapshot of a particle **without** keeping any **information regarding the past**
  - its **G4ParticleDefinition** stores **static** particle **properties** (charge, mass, etc.) as it describes a particle type (e.g. **G4Electron**)
  - its **G4DynamicParticle** stores **dynamic** particle **properties** (energy, momentum, etc.)
  - while all **G4Track**-s, describing the same particle type, share the same, unique **G4ParticleDefinition** object of the given type (e.g. **G4Electron**) while each individual track has its own **G4DynamicParticle** object
  - the **G4Track** object is propagated in a step-by-step way during the simulation and the dynamic properties are continuously updated to reflect the current state
  - manager: **G4TrackingManager**; optional user hook: **G4UserTrackingAction**
  - step-by-step? what about the difference between two such states within a step?

- **G4Step:**
  - a **G4Step** object can provide the information regarding the **change in** the **state of the particle (**that is under tracking) **within a** simulation **step** (i.e. **delta)**
  - has two **G4StepPoint**-s, pre- and post-step points, that stores information (position, direction, energy, material, volume, etc…) that belong to the corresponding point (space/time/step)
  - these are updated in a step-by-step way: the post-step point of the previous step becomes the pre-step point of the next step (when the next step starts)
  - **(important)** if a step is limited by the geometry (i.e. by a volume boundary), the post-step point:
    - **physically** stands **on the boundary** (the step status of the post step point i.e. `G4Step::GetPostStepPoint()->GetStepStatus()` is `fGeomBoundary`)
    - **logically** belongs **to the next volume**
    - since these "*boundary*" **G4Step**-s have information both regarding the previous and the next volumes/materials, boundary processes (e.g. reflection, refractions and transition radiation) can be simulated



- **G4Step:**
  - a **G4Step** object can provide the information regarding the **change in** the **state of the particle (**that is under tracking) **within a** simulation **step** (i.e. **delta)**
  - has two **G4StepPoint**-s, pre- and post-step points, that stores information (position, direction, energy, material, volume, etc…) that belong to the corresponding point (space/time/step)
  - these are updated in a step-by-step way: the post-step point of the previous step becomes the pre-step point of the next step (when the next step starts)
  - **(important)** if a step is limited by the geometry (i.e. by a volume boundary), the post-step point:
    - **physically** stands **on the boundary** (the step status of the post step point i.e. `G4Step::GetPostStepPoint()->GetStepStatus()` is `fGeomBoundary`)
    - **logically** belongs **to the next volume**
    - since these "*boundary*" **G4Step**-s have information both regarding the previous and the next volumes/materials, boundary processes (e.g. reflection, refractions and transition radiation) can be simulated
  - the **G4Track** object, that is under tracking i.e. generates information for the **G4Step** object, can be obtained from the step by the **G4Step::GetTrack()** method and the other way around **G4Track::GetStep()**
  - manager: **G4SteppingManager**; optional user hook: **G4UserSteppingAction**

- **G4Step – G4UserSteppingAction:**
  - optional user action class with the possibility to obtain information after each simulation steps
  - **virtual UserSteppingAction(const G4Step* theStep)** method:
    - is **called at the end of each step** by **G4SteppingManager**
    - providing access to the **G4Step** object representing the simulation step that has just done (see this class in Geant4 `/source/tracking/include/G4UserSteppingAction.hh`)
  - users can implement their own **YourSteppingAction** class by:
    - extending the **G4UserSteppingAction** class
    - providing their own implementation of the method mentioned above
    - creating and registering the corresponding object in the **ActionInitialisation::Build()** interface method (see later)

How to get information regarding the simulation when the **G4Step* theStep** is given?

How to get information regarding the simulation when the `G4Step* theStep` is given?

```
// get the pre-step point
G4StepPoint*         preStp = theStep->GetPreStepPoint();
// get the volume which the step was done
G4VPhysicalVolume* physVol = preStp->GetPhysicalVolume();
// get the energy deposit and length of the step
G4double            stpEdep = theStep->GetTotalEnergyDeposit();
G4double          stpLength = theStep->GetStepLength();
// get the track
G4Track*           theTrack = theStep->GetTrack();
// get the static and dynamic particle properties
const G4ParticleDefinition* partDef =  theTrack->GetParticleDefinition();
const G4DynamicParticle*    partDyn =  theTrack->GetDynamicParticle();
// get the charge of the particle that is under tracking in this step
G4double         partCharge = partDef->GetPDGCharge();
// get the post step point kinetic energy
G4double       postStpEkin = theStep->GetPostStepPoint()->GetKineticEnergy();
// or since the track is already updated to reflect the post-step point
// G4double       postStpEkin = partDyn->GetKineticEnergy();
// which is different in case of the pre-step point kinetic energy that can be
// obtained only from the pre-step point as
G4double         preStpEkin = preStp->GetKineticEnergy();
```

Key concepts of `Geant4` tracking: from an instant to a complete simulation time coverage

- **`G4Event`:**
  - a `G4Event` is the basic simulation unit that represents a **set of `G4Track` objects**
  - **at the beginning** of an event:
    - **primary G4Track object**(s) is(are) **generated** (with their static and initial dynamic properties) and pushed **to a track-stack**
    - **one G4Track object** is **popped from** this track-**stack** and transported/tracked/simulated:
      - ✳the track object **is propagated in a step-by-step way** and its dynamic properties as well as the corresponding `G4Step` object are updated at each step
      - ✳the step is limited either by physics interaction or geometry boundary
      - ✳**transportation** (to the next volume through the boundary) will take place in the later while **physics** interaction in the former case
      - ✳**secondary G4Track-s**, **generated by** these **physics interactions**, are **also pushed to** the track-**stack**
      - ✳a **G4Track object** is kept tracking till:
        - + leaves the outermost (World) volume i.e. goes out of the simulation universe
        - + participates in a destructive interaction (e.g. day or photoelectric absorption)
        - + its kinetic energy becomes zero and doesn't have interaction that can happen “at-rest”
        - + the user decided to (artificially) stop the tracking and kill
      - ✳when one track object reaches its termination point, **a new G4Track object** (either secondary or primary) is popped from the stack for tracking
    - processing an event will be terminated when there is **no any G4Track objects in the track-stack**
  - **at the end** of an event, the corresponding `G4Event` object will store its input i.e. the list of primaries (and possible some of its outputs like hits or trajectory collection)

- **G4Event – G4UserEventAction:**
  - optional user action class, with the possibility to get the control before/after an event processing
  - `virtual BeginOfEventAction(const G4Event* anEvent)`:
    - **called before** a new **event processing** starts by the **G4EventManager**
  - `virtual EndOfEventAction(const G4Event* anEvent)`:
    - **called after** an **event processing competed** by the **G4EventManager**
  - in both cases, the **G4EventManager** provides access to the corresponding **G4Event** object (see this class in Geant4 `/source/event/include/G4UserEventAction.hh`)
  - users can implement their own **YourEventAction** class by:
    - extending the **G4UserEventAction** class
    - providing their own implementation of the two methods mentioned above
    - creating and registering the corresponding object in the **ActionInitialisation::Build()** interface method (see later)
  - at the beginning of the event (`BeginOfEventAction`) we usually clear some data structures that we want to use to accumulate information during the processing of the current event (populated after each step in the **UserSteppingAction**) while at the end of the event (`EndOfEventAction`) we usually write the accumulated information to an upper (Run global) level

- **G4Run:**
  - **G4Run** is a collection of **G4Event-s** (a **G4Event** is a collection of **G4Track**-s)
  - during a run, events are taken and processed one by one in an event-loop
  - **before the start of a run** i.e. at run initialisation (`G4RunManager::Initialize()`): the **geometry** is **constructed** and **physics** is **initialised**
  - **at the start of a run** (`G4RunManager::BeamOn()`): the **geometry is optimised** for tracking (voxelization), the event processing starts i.e. entering into the event-loop
  - as log as the event processing is running, i.e. during the run, the user cannot modify **neither the geometry** (i.e. the detector setup) **nor the physics** settings
  - they **can be changed** though **between run-s** but the **G4RunManager** needs to be informed (re-optimise or re-construct geometry, re-build physics tables):
    - if the **geometry** has been changed, depending on the modifications:
      - `GeometryHasBeenModified()` re-voxelization but no re-`Construct`
      - `ReinitializeGeometry()` complete re-`Construct`

      or with the UI commands **/run/geometryModified** or **/run/reinitializeGeometry**
    - same for the **physics**: `PhysicsHasBeenModifie()` or **/run/physicsModifie**
  - manager: **G4RunManager**; optional user hook: **G4UserRunAction**

- **G4Run – G4UserRunAction:**
  - optional user action class, with the possibility to get the control before/after a run and to provide custom run-object (see later)
  - `virtual BeginOfRunAction(const G4Run* aRun):`
    - **called before** the **run starts** i.e. **before the first event processing** starts by the **G4RunManager**
  - `virtual EndOfRunAction(const G4Run* aRun):`
    - **called after** a **run completed** i.e. **after the last event processing competed** by the **G4RunManager**
  - in both cases, the **G4RunManager** provides access to the corresponding **G4Run** object (see this class in Geant4 `/source/run/include/G4UserRunAction.hh`)
  - users can implement their own **YourRunAction** class by:
    - extending the **G4UserRunAction** class
    - providing their own implementation of the two methods mentioned above
    - creating and registering the corresponding object in the **ActionInitialisation::Build()** interface method (see later)
  - note, at the beginning of the run (`BeginOfRunAction`) we usually allocate/initialise some data structures/histograms that we want to use during the whole run to collect the final simulation results that we usually print out at the end of the run (`EndOfRunAction`)

- **G4Run – G4UserRunAction:**
  - an additional method, `virtual G4Run* GenerateRun()` is also available:
    - users can implement their own **YourRun** class by:
      - ✳extending the G4Run class and defining their own run-global data structure (i.e. quantities, objects to be collected during the complete run as the result of the simulation/run)
      - ✳instantiation of these custom **YourRun** object needs to be done in this `GenerateRun()` method of **YourRunAction**
  - this will be invoked by the **G4RunManager** at initialisation to generate **YourRun** instead of the (base) **G4Run** object
  - the generated **YourRun** object can be accessed during the simulation as the Run (in UserAction-s) and can be populated by information
  - then the final information, collated during the run into **YourRun** object, can be printed out e.g. by calling the summary method of **YourRun** from the `EndOfRunAction` method of **YourRunAction** class
  - in case of MT: we will cover this in the MT extension of our application !

- **G4Run – G4UserRunAction:**
  - an additional method, `virtual G4Run* GenerateRun()` is also available:
    - users can implement their own **YourRun** class by:
      - ✳extending the G4Run class and defining their own run-global data structure (i.e. quantities, objects to be collected during the complete run as the result of the simulation/run)
      - ✳instantiation of these custom **YourRun** object needs to be done in this `GenerateRun()` method of **YourRunAction**
  - this will be invoked by the **G4RunManager** at initialisation to generate **YourRun** instead of the (base) **G4Run** object
  - the generated **YourRun** object can be accessed during the simulation as the Run (in UserAction-s) and can be populated by information
  - then the final information, collated during the run into **YourRun** object, can be printed out e.g. by calling the summary method of **YourRun** from the `EndOfRunAction` method of **YourRunAction** class

Time to add our own user actions to the application!

THE MAIN: USER INITIALISATIONS AND MANDATORY ACTIONS

The main: user initialisations and mandatory actions

- As mentioned before, **`Geant4`** do not provide a main program:
  - there are components of a simulation, like the **geometry**, **physics** settings and the **primary** particle **generation** that are changing from problem to problem
  - therefore, the **user needs to provide these** settings **in the `main` method** of their **`Geant4`** application

The main: user initialisations and mandatory actions (`G4RunManager`)

- As mentioned before, **`Geant4`** do not provide a main program:
  - there are components of a simulation, like the **geometry**, **physics** settings and the **primary** particle **generation** that are changing from problem to problem
  - therefore, the **user needs to provide these** settings **in the `main` method** of their **`Geant4`** application

1. Create a **`G4RunManager`** object **(mandatory)**:
   - the **only** mandatory **manager object** that user the needs to create: all others (**`G4EventManager`**, **`G4SteppingManger`**, etc.) are created and deleted automatically
   - the **`G4RunManager`** is responsible to **control the** flow of a **run**, the top level simulation unit
   - this includes **initialisation of the run** i.e. building, **setting up the** simulation **environment**
   - all problem specific information need to be given to the **`G4RunManager`** by the user through the interfaces provided by the **`Geant4`** toolkit (we will see them one by one):
     - **`G4VUserDetectorConstruction`**(mandatory): how the geometry should be constructed, built
     - **`G4VUserPhyscsList`**(mandatory): all the particles and their physics interactions to be simulated
     - **`G4VUserActionInitialization`** (mandatory):
       - ✳ **`G4VUserPrimaryGeneratorAction`** (mandatory): how the primary particle(s) in an event should be produced
       - ✳ additional, **optional user actions** (**`G4UserRunAction`**, **`G4UserEventAction`**, **`G4UserSteppingAction`**, etc..)

- As mentioned before, **`Geant4`** do not provide a main program:
  - there are components of a simulation, like the **geometry**, **physics** settings and the **primary** particle **generation** that are changing from problem to problem
  - therefore, the **user needs to provide these** settings **in the `main` method** of their **`Geant4`** application

1. Create a **`G4RunManager`** object **(mandatory)**:
   - the **only** mandatory **manager object** that user needs to create: all others (**`G4EventManager`**,

See more when we write the `main` method of our own application!

should be produced

✳ additional, **optional user actions** (**`G4UserRunAction`**, **`G4UserEventAction`**, **`G4UserSteppingAction`**, etc..)

- **MT note:** **`G4MTRunManager`** object needs to be created in case of **`Geant4 MT`**

2. Create **`YourDetectorConstruction`** object and register it in your **`G4RunManager`** object **(mandatory)**:

- the **`G4VUserDetectorConstruction`** interface is provided by the **`Geant4`** toolkit to **describe** the **geometrical setup**, including all volumes with their shape, position and **material definition**
- its **`G4VUserDetectorConstruction::Construct()`** interface method (pure virtual) is invoked by the **`G4RunManager`** at initialisation
- **derive your own detector description**, e.g. **`YourDetectorConstruction`** class from this base class and implement the **`Construct()`** interface method:
  - create all materials will need to use in your geometry
  - describe your detector geometry by creating and positioning all volumes
  - return the pointer to the root of your geometry hierarchy i.e. the pointer to your "World" **`G4VPhysicalVolume`**
- create **`YourDetectorConstruction`** object and register it in your **`G4RunManager`** object by using the `G4RunManager:SetUserinitialization` method (see this in the source!)
- **MT note:** the **`Construct()`** interface method is **invoked only by the Master Thread** in case of **`Geant4 MT`** (i.e. only one detector object), while the other `ConstructSDandField()` interface method is **invoked by each Worker Threads** (i.e. thread local objects created)

2. Create `YourDetectorConstruction` object and register it in your `G4RunManager` object **(mandatory)**:
   - the `G4VUserDetectorConstruction` interface is provided by the **`Geant4`** toolkit to **describe** the **geometrical setup**, including all volumes with their shape, position and

See more at the Detector Construction lecture! We will write together the `DetectorConstruction`.

   - **MT note:** the `Construct()` interface method is **invoked only by the Master Thread** in case of **`Geant4 MT`** (i.e. only one detector object), while the other `ConstructSDandField()` interface method is **invoked by each Worker Threads** (i.e. thread local objects created)

3. Create `YourPhysicsList` object and register it in your `G4RunManager` object **(mandatory)**:

- the `G4VUserPhysicsList` interface is provided by the **`Geant4`** toolkit to **describe** the **physics setup**, including **definition of** all **particles** and their physics interactions, **processes**
- its `G4VUserPhysicsList::ConstructParticle()` and `::ConstructProcess()` interface methods (pure virtual) are invoked by the **`G4RunManager`** (actually by the **`G4RunManagerKernel`** and process construction is invoked indirectly) at initialisation
- **derive your own physics list**, e.g. `YourPhysicsList` class from this base class and implement the `ConstructParticle()` and `ConstructProcess()` interface methods:
  - create all particles in the `ConstructParticle()` method
  - create all processes and sign them to particles in the `ConstructProcess()` method
- create `YourPhysicsList` object and register it in your `G4RunManager` object by using the `G4RunManager:SetUserinitialization` method (see this in the source!)
- constructing physics list as described above is **recommended only for advanced users**!
- **`Geant4`** provides possibilities with different level of granularity to build up or obtain even complete pre-defined physics list

3. Create `YourPhysicsList` object and register it in your `G4RunManager` object **(mandatory)**:
   - the `G4VUserPhysicsList` interface is provided by the `Geant4` toolkit to **describe** the **physics setup**, including **definition of** all **particles** and their physics interactions, **processes**

See more at the Physics List lecture!

We will use one of the pre-defined physics list.

- `Geant4` provides possibilities with different level of granularity to build up or obtain even complete pre-defined physics list

4. Create `YourPrimaryGeneratorAction` object **(mandatory**, see next slide how to register**)**:

- the `G4VUserPrimaryGeneratorAction` interface is provided by the **`Geant4`** toolkit to describe how the primary particle(s) in an event should be produced
- its `G4VUserPrimaryGeneratorAction::GeneratePrimaries()` interface method (pure virtual) is invoked by the **`G4RunManager`** during the event-loop (in its `G4RunManager::GenerateEvent()` method)
- **derive your own primary generator action,** e.g. `YourPrimaryGeneratorAction` class from this base class and implement the `GeneratePrimaries()` interface method:
  - describe how the primary particle(s) in an event should be produced
  - we will use a **`G4ParticleGun`** object, provided by the **`Geant4`** toolkit, to generate primary particles: one particle per event with defined kinematics
- note:
  - the **Detector-Construction** and the **Physics-List** need to be **created directly in the main** program and **registered directly in the `G4RunManager`** object
  - all **User-Actions** needs to be **created** and **registered** in the **User-Action-Initialisation** (**including the** only **mandatory Primary-Generator-Action** as well as **all** other, **optional User-Actions**)
  - see more on this and on the `G4VUserActionInitialization` in the next slide

4. Create `YourPrimaryGeneratorAction` object **(mandatory**, see next slide how to register**)**:
   - the `G4VUserPrimaryGeneratorAction` interface is provided by the **`Geant4`** toolkit to describe how the primary particle(s) in an event should be produced
   - its `G4VUserPrimaryGeneratorAction::GeneratePrimaries()` interface method (pure virtual) is invoked by the **`G4RunManager`** during the event-loop (in its `G4RunManager::GenerateEvent()` method)

We will write together the `PrimaryGeneratorAction`.

   - note:
     - the **Detector-Construction** and the **Physics-List** need to be **created directly in the main** program and **registered directly in the `G4RunManager`** object
     - all **User-Actions** needs to be **created** and **registered** in the **User-Action-Initialisation** (**including the** only **mandatory Primary-Generator-Action** as well as **all** other, **optional User-Actions**)
     - see more on this and on the `G4VUserActionInitialization` in the next slide

The main: user initialisations and mandatory actions (`G4VUserActionInitialization`)

5. Create `YourActionInitialization` object and register it in your `G4RunManager` object **(mandatory)**:

- the `G4VUserActionInitialization` interface is provided by the **`Geant4`** toolkit to **create** and **register:**
  - **the only one mandatory `G4VUserPrimaryGeneratorAction` user action**
  - all other **optional user actions** (**`G4UserRunAction`**, **`G4UserEventAction`**, etc..)
- its `G4VUserActionInitialization::Build()` interface method (pure virtual) is invoked by the **`G4RunManager`** at initialisation
- **derive your own action initialisation**, e.g. `YourActionInitialization` class from this base class and implement the `Build()` interface methods:
  - **create** an object from `YourPrimaryGeneratorAction` (see next slide) and **register** by calling the corresponding `G4VUserActionInitialization::SetUserAction()` base class method
  - **create all** additional, **optional user action** objects and **register them** similarly
- **MT note:**
  - the above `Build()` method is invoked **by all Worker Thread** while the additional `BuildForMaster()` method is invoked **only by the Master Thread** in **`Geant4 MT`**
  - the **only user action,** that is supposed **to be created in** this `BuildForMaster()` method, is the **`G4UserRunAction` for the Master Thread**
  - this **`G4UserRunAction`** is the same or different compared to that used in case of the **Worker Threads**
  - but contains a **`G4Run`** generation that will generate the **run-global, thread-global** i.e. **Master `G4Run` object** to **which all** other **Worker `G4Run` objects will be Merged at the end** by calling the **`G4Run::Merge`** method
  - the `BuildForMaster()` method is invoked only in case of **`Geant4 MT`**

The main: user initialisations and mandatory actions (`G4VUserActionInitialization`)

5. Create `YourActionInitialization` object and register it in your `G4RunManager` object **(mandatory)**:

- the `G4VUserActionInitialization` interface is provided by the `Geant4` toolkit to **create** and **register:**
  - **the only one mandatory** `G4VUserPrimaryGeneratorAction` **user action**
  - all other **optional user actions** (`G4UserRunAction`, `G4UserEventAction`, etc..)
- its `G4VUserActionInitialization::Build()` interface method (pure virtual) is invoked by the `G4RunManager` at initialisation
- **derive your own action initialisation**, e.g. `YourActionInitialization` class from this base class and implement the `Build()` interface methods:
  - **create** an object from `YourPrimaryGeneratorAction` (see next slide) and **register** by calling `SetUserAction()` base class method

We will write together the `ActionInitializaton.`

- but contains a `G4Run` generation that will generate the **run-global, thread-global** i.e. **Master** `G4Run` **object** to **which all** other **Worker** `G4Run` **objects will be Merged at the end** by calling the `G4Run::Merge` method
- the `BuildForMaster()` method is invoked only in case of `Geant4 MT`

OUR STEP BY STEP PLAN

I. **Intermediate application:** the bare minimum that is needed to run the simulation:

- the bare minimum that is needed to run the simulation (execute, without collecting data!)
- implement all the mandatory components mentioned before:
  - **`YourDetectorConstruction`**:
    - ✳a simple **box** (shape) as the detector/**target** filled with with **silicon** as material
    - ✳placed in a **box** (shape) "**world**" volume filled with **low density hydrogen gas**
  - **`YourPhysicsList`: we will use** one of the **pre-defined**, ready-to-use **physics list provided by** the **`Geant4`** toolkit (therefore no need to write any user physics list class like in our case)
  - **`YourPrimaryGeneratorAction`**:
    - ✳a simple **particle gun (`G4ParticleGun`):** generates a **single primary** particle **per event** with pre-**defined** particle **type** and **kinematics** pointing toward to our target
  - **`YourActionInitialization`:**
    - ✳implement the construction and registration of our **`YourPrimaryGeneratorAction`** object
- develop the `main` method of the application and execute the simulation
- inspect the results

II. **Intermediate application:** more control over the execution

- more control over the execution, but still no simulation data collected
- add **functionality to** the `main` method of our application to be able to **run the application**:
  - in **interactive** or **batch mode**
  - with or without **visualisation**
  - write the corresponding macro files
- become familiar with all possible modes of executions, understand their advantage and find your most comfortable one

III. **Final application:** extension with optional User-Actions

- extension needed to collect some data during the simulation
- add **optional User Actions** (run-, event-, stepping-actions):
  - simulate mean value and sigma of the energy deposited in the detector/target per event
- become conformable with all of these optional user actions that provide access to the simulation workflow, data, states to the application developer (and eventually to the user)

IV. **Final application:** add more flexibility regarding the configuration

- define and add **User Interface (UI) commands** to the detector construction to be able to **configure the target material** and **thickness**
- become familiar with developing your own UI commands to your **`Geant4`** application that can increase significantly the flexibility of your application