Geant4 & NEXUS
Hands-on Tutorial

Justo Martín-Albo (IFIC)

IFIC • 24–25 February 2020

GOAL AND MEANS

Our main goal for this tutorial:

- Become familiar enough with NEXUS to be able to start contributing to its development.

How?

- Review the C++ concepts most frequently used in Geant4 and NEXUS.
- Introduce the basic components that form a Geant4 simulation.
- Learn how to build and run NEXUS, and review its structure and organization.
- Show what documentation resources can be useful when writing NEXUS code.
- Write a full simulation within NEXUS and analyze the generated data.

NEXUS (*NEXt Utility for Simulation*) is the Geant4-based detector simulation of the NEXT experiment. It handles the different *detector geometries* (DEMO, NEW, NEXT-100...) and *event generators* (DBDs, radioactive backgrounds, muons...) we need, and produces output files (using now HDF5) in a common format understood by the software downstream.

Geant4 is a C++ library for the simulation of the passage of particles through matter using Monte-Carlo methods.

A very brief introduction to C++

Fork and clone the following repository:

https://github.com/jmalbos/nexus-tutorial

Build the examples of the C++ introduction:

```
cd nexus-tutorial/C++Intro
mkdir build && cd $_;
cmake ..
cmake —build .
```

C++ INTRO

0. Function declaration and definition. Main function and control flow. Scope of variables and namespaces.

1. Passing arguments by value, reference and pointer.

2. Declaration and definition of classes. Constructors and destructors.

3. Class inheritance. Memory allocation.

4. Polymorphism. Vectors.

C++ INTRO: SCOPE


```
#include <iostream>

int var = 123; // Global variable with file scope

int main()
{
 int var = 456; // Local variable with scope limited to main()
 std::cout << “var = “ << var << std::endl;
 return 0;
}
```

Output:

```
var = 456
```

C++ INTRO: PASSING ARGUMENTS


```
#include <iostream>

int add_one_by_val(int  n); // Passing argument by value
int add_one_by_ref(int& n); // Passing argument by reference

int main()
{
 int var = 0;
 std::cout << add_one_by_val(var) << "!=" << var << std::endl;
 std::cout << add_one_by_ref(var) << "==" << var << std::endl;
 return 0;
}
```

Output

```
0 != 1
1 == 1
```

C++ INTRO: CLASSES


```
#include <iostream>

class MyFirstClass
{
public:
 MyFirstClass() {}
 ~MyFirstClass() {}
 void Greeting() { std::cout << greeting_ << std::endl; }
private:
 std::string greeting_ = “Hello World!”;
};

int main()
{
 MyFirstClass mfc;
 mfc.Greeting();
 return 0;
}
```

Output:

```
Hello World!
```

C++ INTRO: INHERITANCE


```
#include <iostream>

class BaseClass
{
public:
 virtual BaseClassFunction()
 { std::cout << “BaseClassFunction()” << std::endl; }
};

class DerivedClass: public BaseClass
{
public:
 virtual DerivedClassFunction()
 { std::cout << “DerivedClassFunction()” << std::endl; }
};

int main()
{
 DerivedClass dc_instance;
 dc_instance.BaseClassFunction();
 dc_instance.DerivedClassFunction();
 return 0;
}
```

Output:

```
BaseClassFunction()
DerivedClassFunction()
```

C++ INTRO: INHERITANCE

SPECIFIER	WITHIN SAME CLASS	WITHIN DERIVED CLASS	FROM OUTSIDE
`public`	✔	✔	✔
`protected`	✔	✔	✘
`private`	✔	✘	✘

C++ INTRO: POLYMORPHISM


```
#include <iostream>

class Square: public Shape
{ (…) };

class Circle: public Shape
{ (…) };

int main()
{
 Shape* square = new Square(1.0);
 Shape* circle = new Circle(1.0);
 std::vector<Shape*> vs = {circle, square};
 for (auto i: vs) std::cout << “Area: “ << i->Area() << std::endl;
 return 0;
}
```

Output:

```
Area: 3.14159
Area: 1.0
```

Geant4 Basics

Fork and clone the following repository (if you haven't done it already):

https://github.com/jmalbos/nexus-tutorial

Build the G4Basic application (N.B. Geant4 required for this):

```
cd nexus-tutorial/G4Basic
mkdir build && cd $_;
cmake ..
cmake —build .
```

BASIC COMPONENTS

To build a Geant4 application, one must provide at least the following components:

- **Detector geometry** (`G4VUserDetectorConstruction`): description of the detector setup in terms of shapes, their relative positions and the materials they are made of.
- **Primary generation** (`G4VUserPrimaryGeneratorAction`): initial conditions of the events; what primary particles should be produced, where and with what kinematics.
- **Physics list** (`G4VUserPhysicsList`): particle types and physical processes considered for the simulation.

Pointers to these objects are passed to the `G4RunManager` — the class in charge of run control — in the main function of the simulation application.

BASIC COMPONENTS

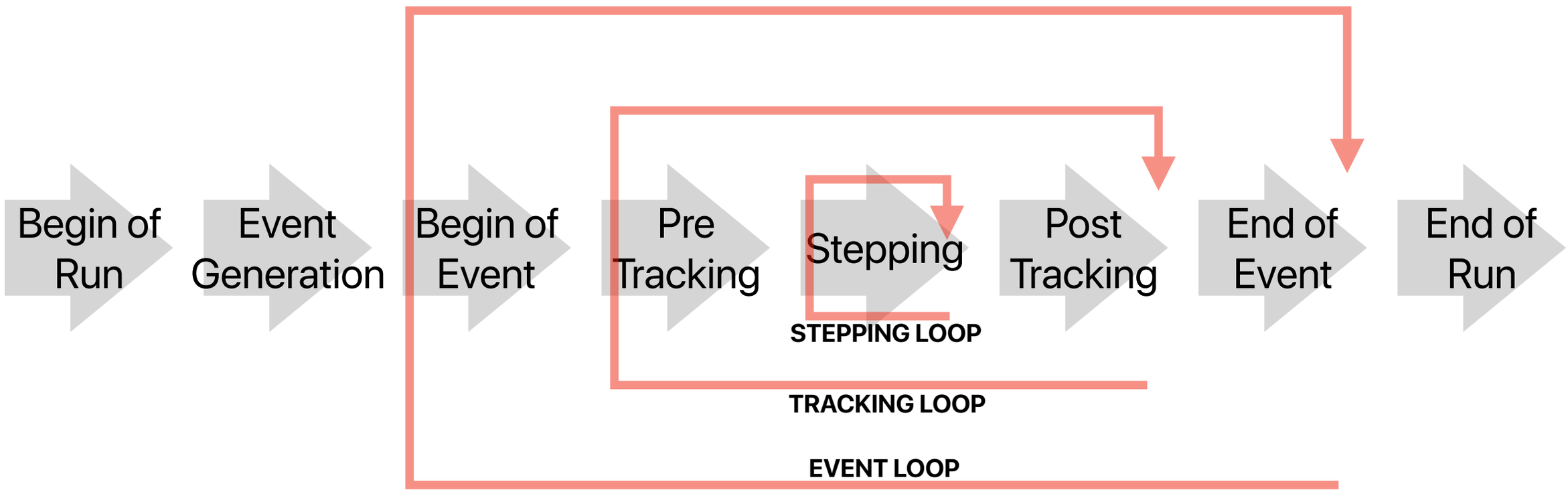

```
#include “PhysicsList.h”
#include “DetectorConstruction.h”
#include “PrimaryGeneration.h”

#include <G4RunManager.hh>

int main(int argc, char const *argv[])
{
 // Construct the run manager and set the initialization classes next
 G4RunManager* runmgr = new G4RunManager();
 runmgr->SetUserInitialization(new PhysicsList());
 runmgr->SetUserInitialization(new DetectorConstruction());
 runmgr->SetUserAction(new PrimaryGeneration());
 runmgr->Initialize();

 (…)

 delete runmgr;
 return 0;
}
```

BASIC COMPONENTS

In addition to the mandatory classes, several action classes can be provided to extract information from the simulation at several points in the processing:

- **Run action** (`G4UserRunAction`): actions before and after each run.
- **Event action** (`G4UserEventAction`): actions before and after each event.
- **Tracking action** (`G4UserTrackingAction`): actions before and after the processing of each track.
- **Stepping action** (`G4UserSteppingAction`): actions after each tracking step.

A GEANT4 RUN



A **run** consists of a number of events.

An **event** consists of a number of tracks.

A **track** consists of a number of steps.

A GEANT4 EVENT

An **event** is the is the basic unit of simulation in a Geant4 application.

At the beginning of an event, **primary particles** are generated according to the conditions specified in the primary generation action and pushed into a stack.

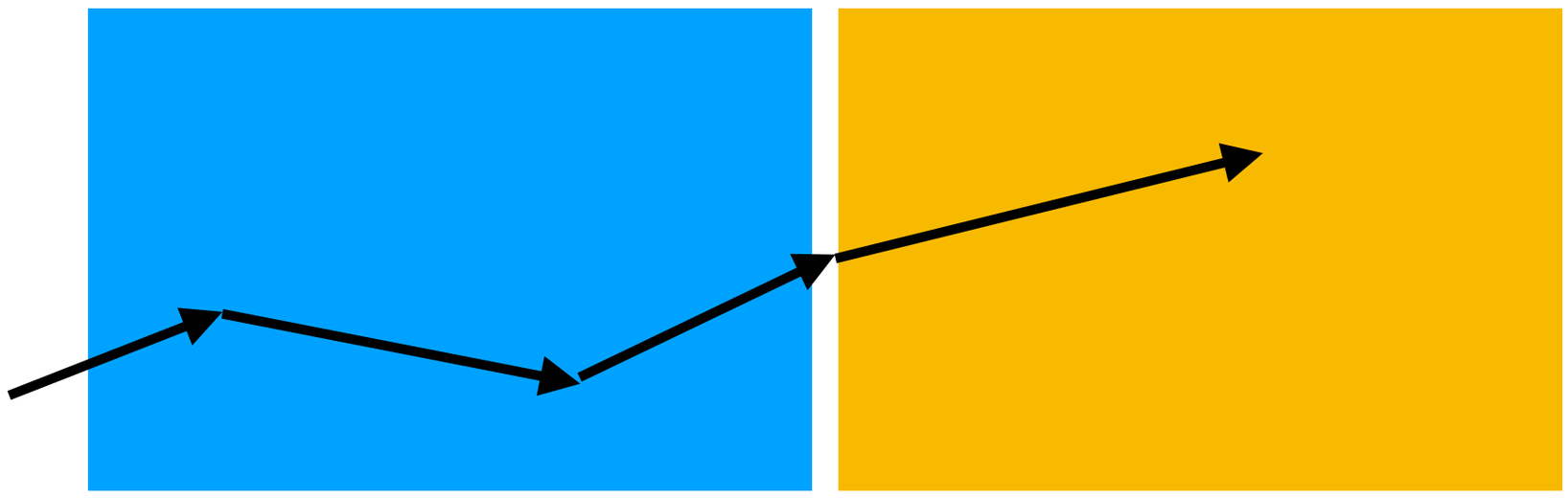
Particles are popped up from the stack one by one and tracked through the geometry in a series of steps down to zero energy. Any resulting secondary particle is pushed into the stack.

A GEANT4 STEP

At the beginning of a step, Geant4 calculates for every possible physical process a posible step length via random sampling of their probability distributions.

The process which requires the shortest (in space-time) interaction length limits the step.

The step is limited as well by geometrical boundaries.



GEOMETRIES

The detector definition requires the representation of its geometrical elements and the materials they are made of. The geometrical representation of detector elements focuses on the definition of solid models and their spatial position, as well as their logical relations to one another (the volume hierarchy), such as in the case of containment.

Geant4 uses the concept of **solid volume** to manage the representation of the shape and dimensions of a detector element.

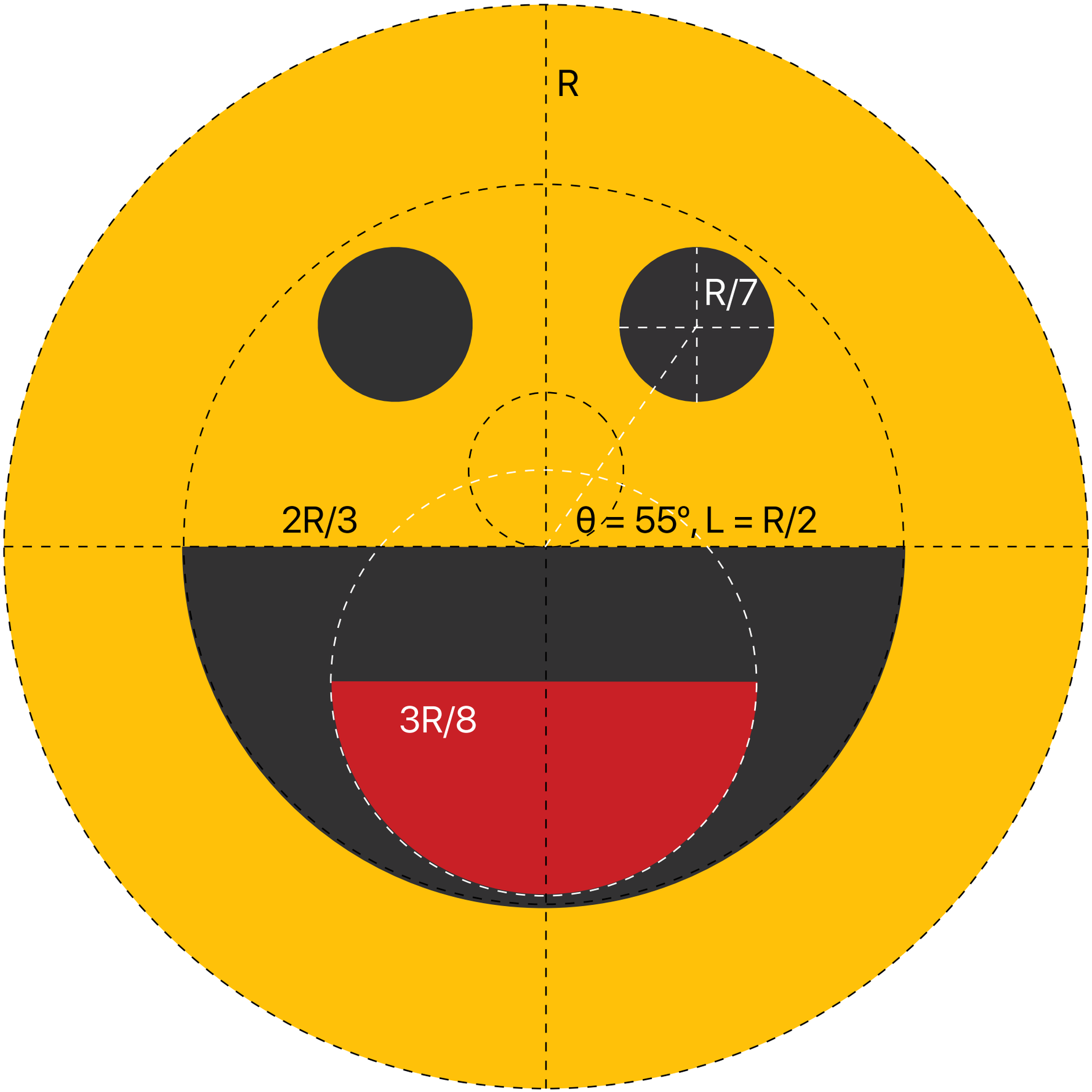
A **logical volume** represent a solid volume made of a certain material.

A **physical volume** manages the representation of the spatial positioning of detector elements and their hierarchical relations.

All these volumes must be dynamically allocated in the user program, and are deleted automatically at the end of the job.

GEOMETRIES


```
 G4String world_name = "WORLD";
 G4double world_size = 25.*m;

 G4Sphere* world_solid_vol =
     new G4Sphere(world_name, 0., world_size/2., 0., 360.*deg, 0.,
180.*deg);

 G4Material* world_mat =
     G4NistManager::Instance()->FindOrBuildMaterial("G4_Galactic");

G4LogicalVolume* world_logic_vol =
     new G4LogicalVolume(world_solid_vol, world_mat, world_name);

G4VPhysicalVolume* world_phys_vol =
     new G4PVPlacement(nullptr, G4ThreeVector(0.,0.,0.),
                       world_logic_vol, world_name, nullptr,
                       false, 0, true);
```


R
R/7
2R/3
θ = 55°, L = R/2
3R/8