SW performance of the IDEA Dual-Readout tubes-based calorimeter simulation

Lorenzo Pezzotti

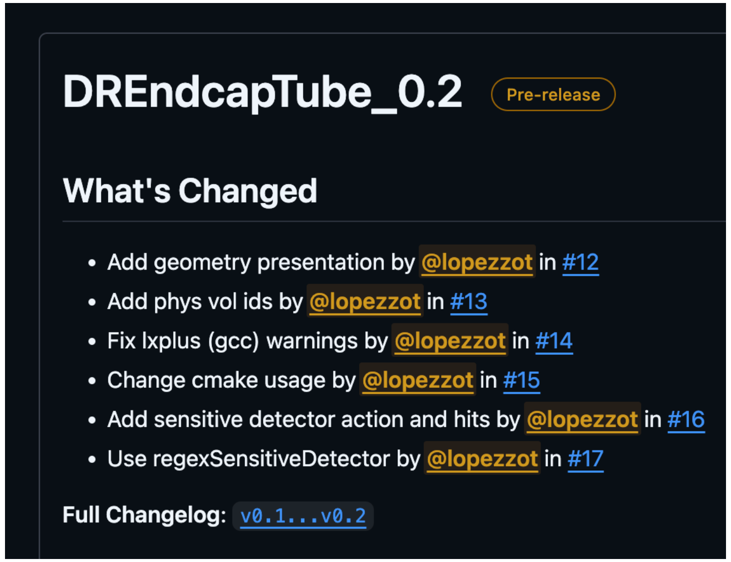
FCC Full Sim Working Meeting
18/9/2024

Objective

- In July a new DD4hep geometry for the IDEA capillary-tubes-based dual-readout endcap calorimeter was developed and presented at this meeting
  - A quick recap is given in the following
- The next goal was to develop a DD4hep sensitive (detector) action (`Geant4SensitiveAction<>`) for this detector which:
  - Allows to simulate events at a single-threaded rate of $O(1) - O(10)$ s/evt at all the FCCee energy poles
  - Reduces the (too) large memory footprint problem
  - Is validated against experimental test-beam data

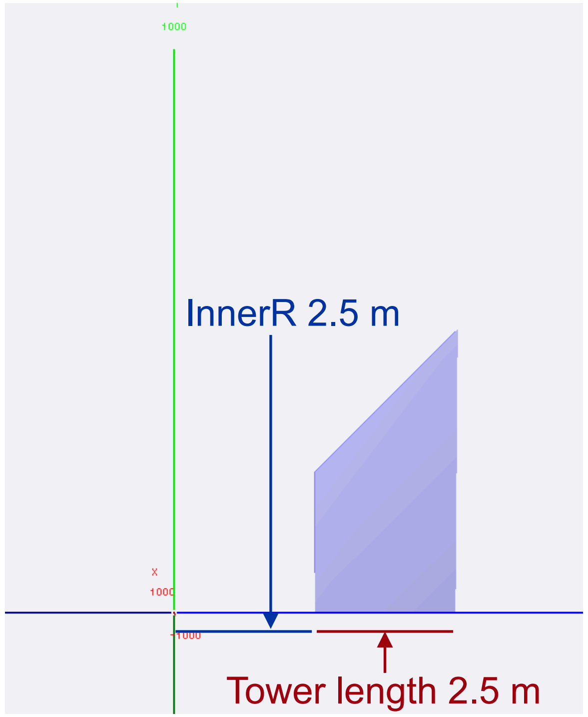
Objective

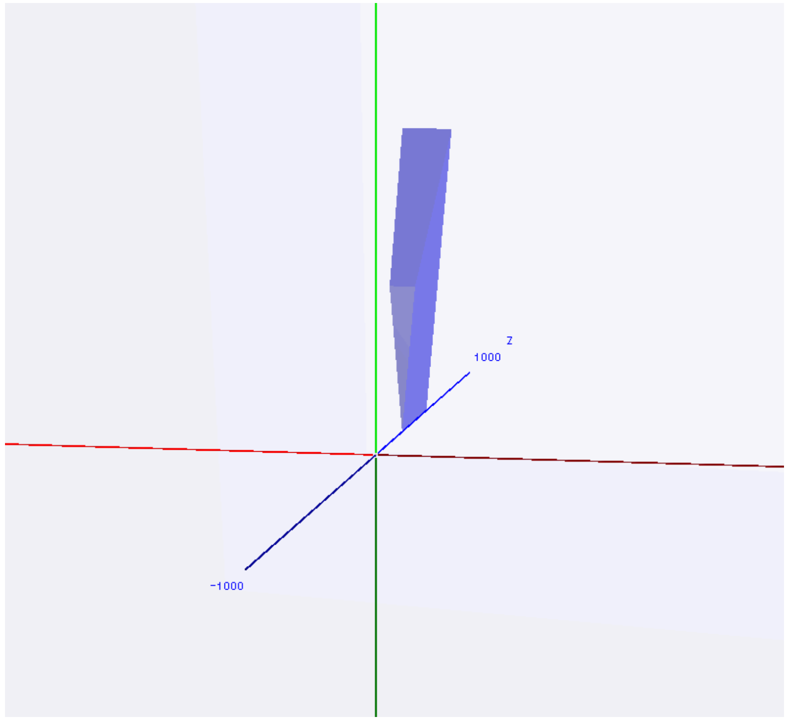
- In July a new DD4hep geometry for the IDEA capillary-tubes-based dual-readout endcap calorimeter was developed and presented at this meeting
  - A quick recap is given in the following
- The next goal was to develop a DD4hep sensitive (detector) action (`Geant4SensitiveAction<>`) for this detector which:
  - Allows to simulate events at a single-threaded rate of $O(1) - O(10)$ s/evt at all the FCCee energy poles
  - Reduces the (too) large memory footprint problem
  - Is validated against experimental test-beam data
- A solution is proposed for the endcap simulation
  - available as v0.2 of DREndcapTubes
  - the same solution is applicable to be the barrel simulation by Andreas

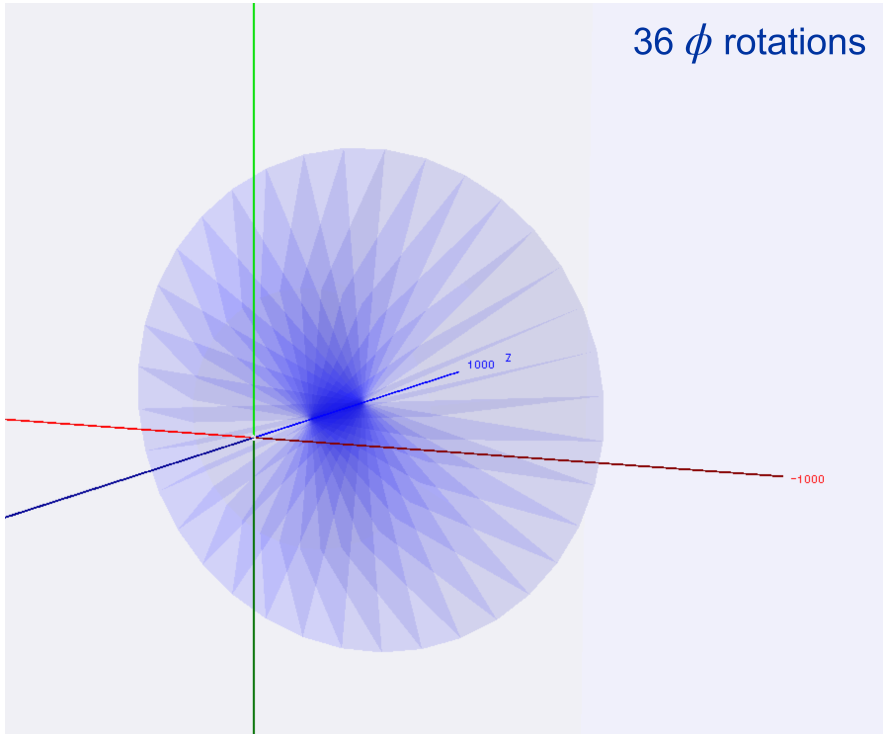


(Recap) ϕ-slice

- A single ϕ-slice is a DD4hep EightPointSolid (equivalent to root Arb8, or Geant4 G4GenericTrap)
  - It is build according to the calorimeter inner radius (2.5 m), tower length (2.5 m), and ϕ-unit ($2*\pi/36$)
  - Assumes that the barrel and endcap region will touch at $\pi/4$
  - Is replicated around the z-axis to full coverage





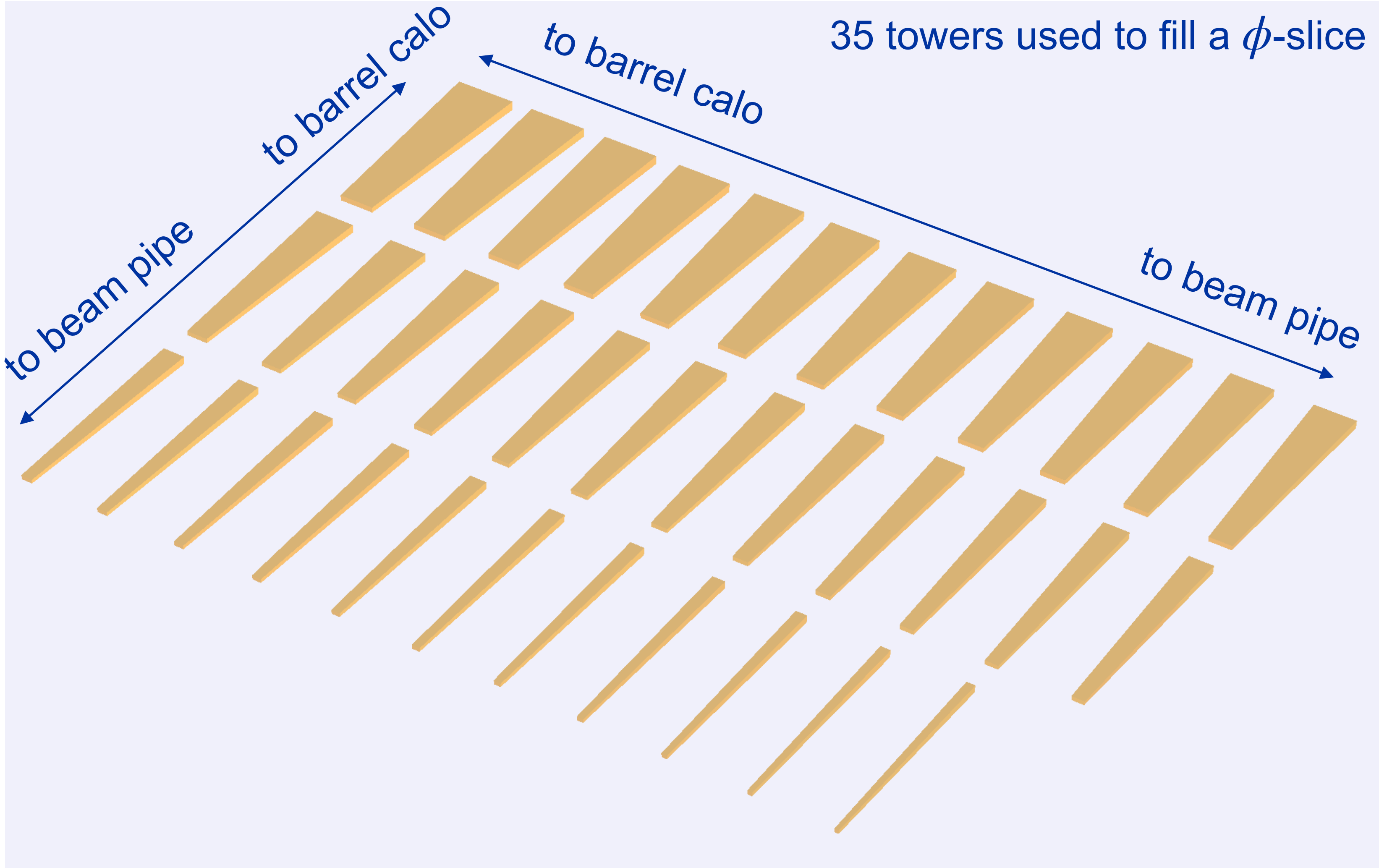


(Recap) **Towers**

- Towers are Trap (equivalent to Geant4 G4Trap) that define a θ-region inside the ϕ-slice. The center of each tower points to the IP.
- 40 towers are considered, each covering a region of

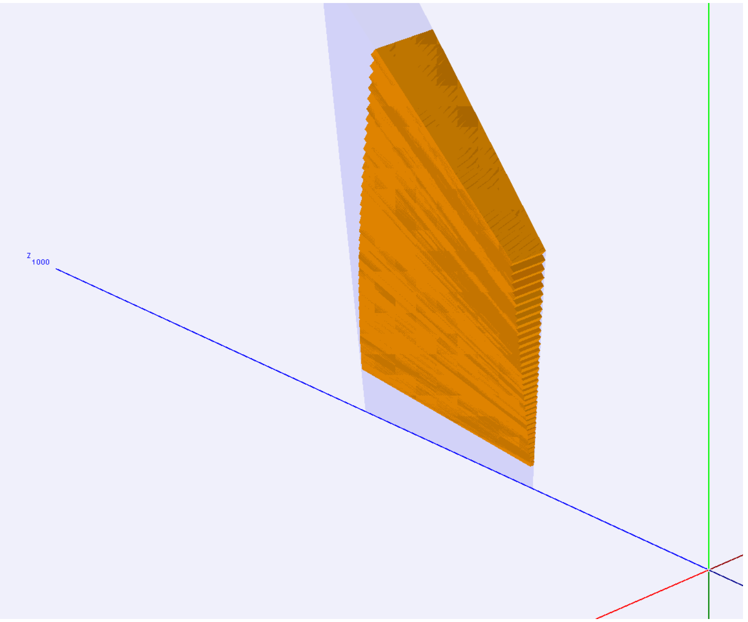
$$\Delta\theta = \frac{\pi/4}{40}$$

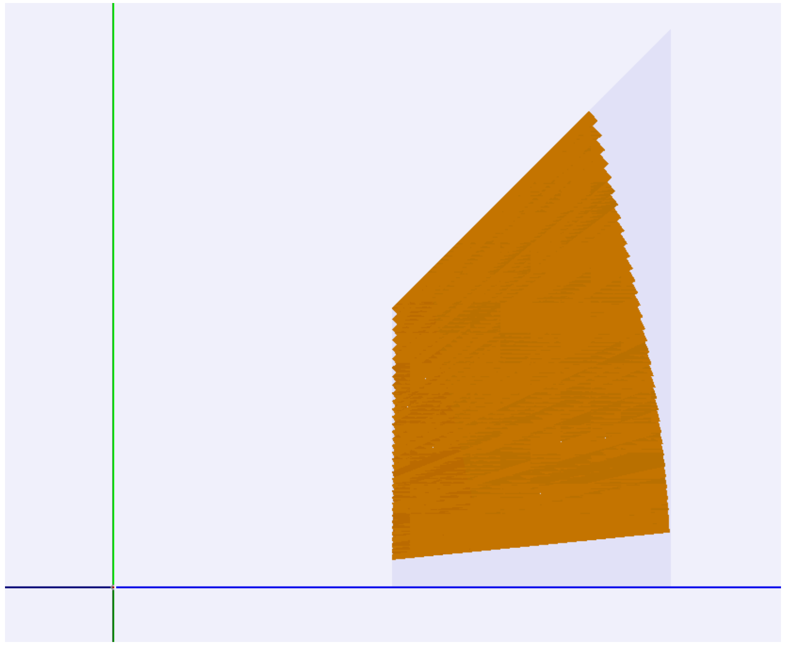
- As towers must fit inside the ϕ-slice their dimension changes with θ

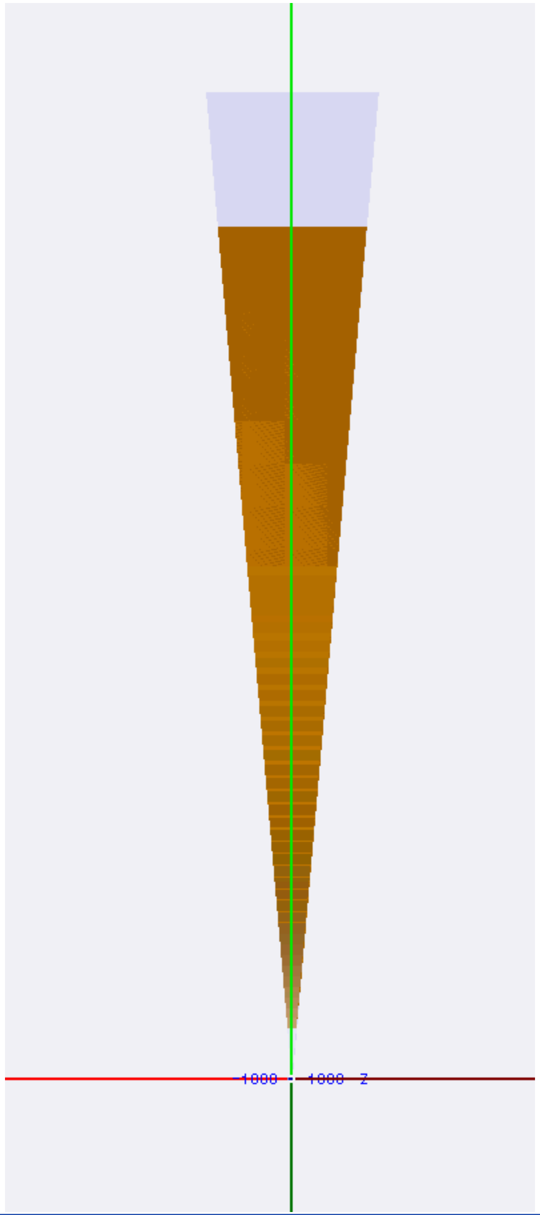


(Recap) **Towers**

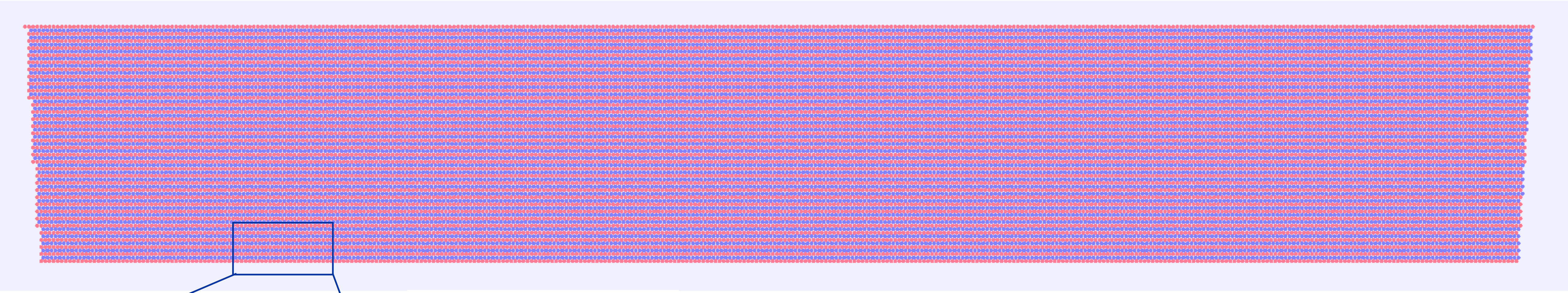
- Towers are Trap (equivalent to Geant4 G4Trap) that define a θ-region inside the ϕ-slice. The center of each tower points to the IP.
- To leave room to the beam pipe 35 towers are actually placed in a projective manner inside the ϕ-slice
- Each of the 35 is 2.5 m long, i.e. the calo containment is independent of θ

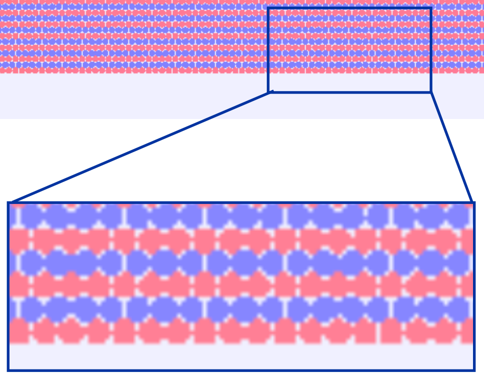


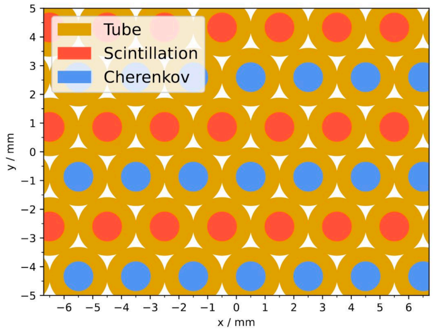




(Recap) **Filling the towers**





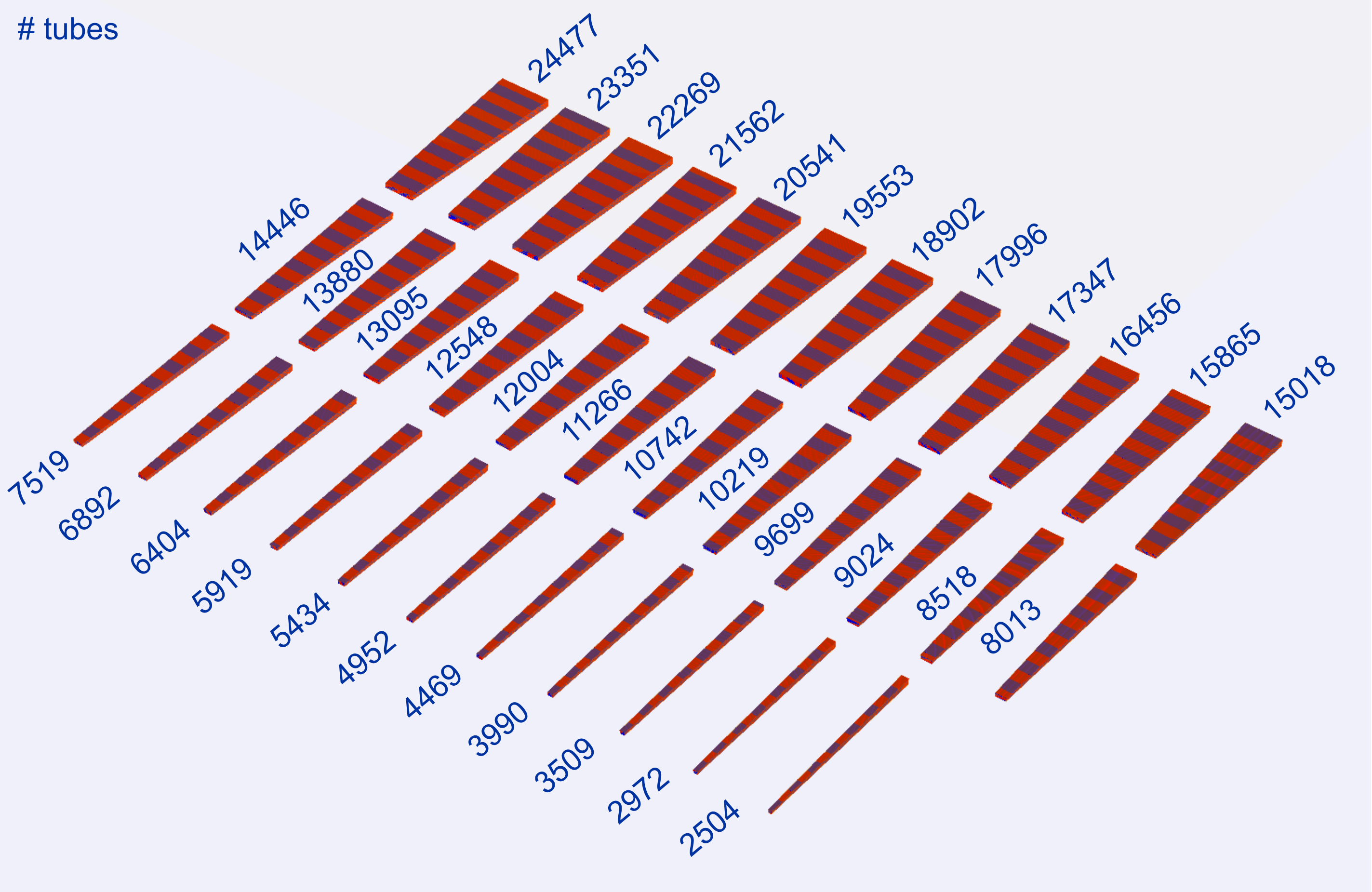


- Tubes are 1-mm-thick radius Tubs (equivalent to G4Tubs) and house Scintillating or Cherenkov (clear) optical fibers. In the following optical fibers inside tubes are not displayed to aid visibility.
- Tubes z-axis is always parallel to the tower axis, i.e. they are projective pointing to the IP
- Tubes are placed starting from the back face of the tower (the biggest one)

(Recap) Some numbers

- Considering the default parameters (36 rotations around ϕ, 35 towers along θ, inner radius 2.5 m, 2.5 m long towers)
  - tubes per ϕ-slice: 421325
  - tubes per both endcaps: 30335400*
  - total tube length: 47128.5 km

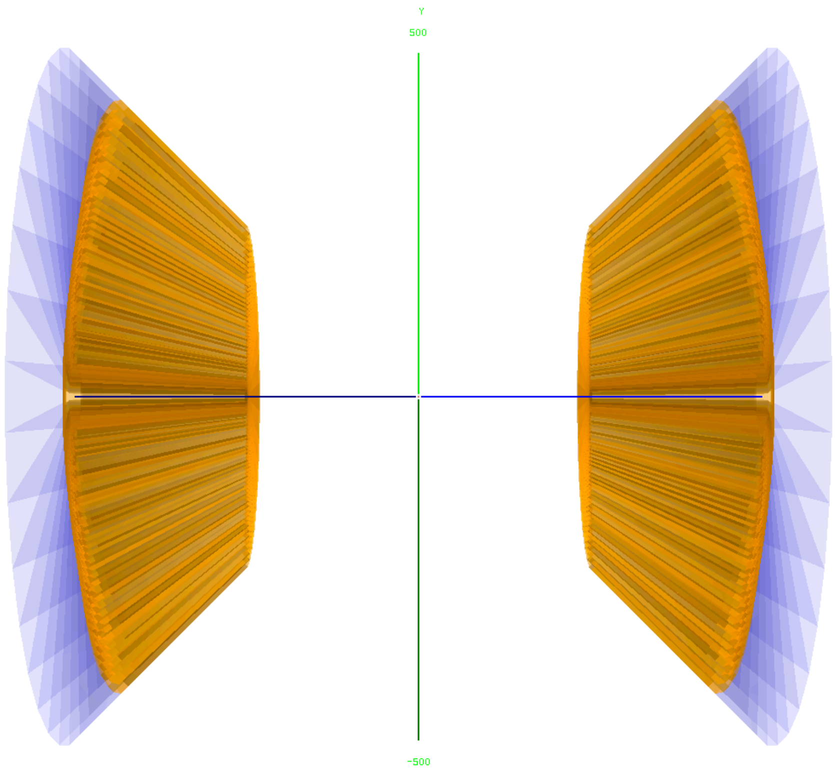
- If we reduce the tower length to 2 m
  - tubes per ϕ-slice: 346877
  - tubes per both endcaps: 24975144
  - total tube length: 33072.5 km

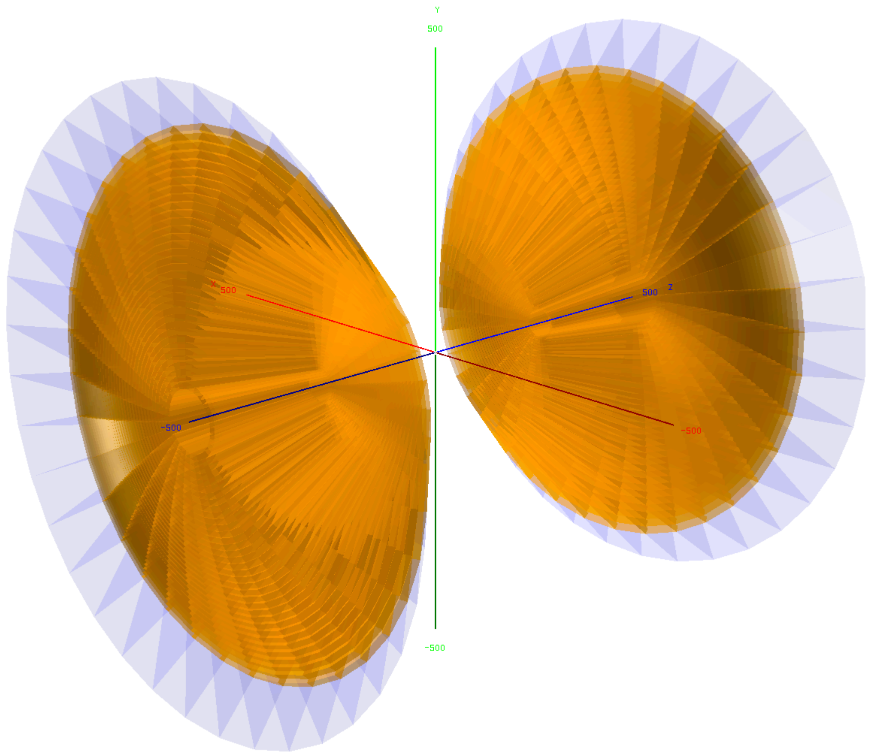


*the number of Tubs objects in memory is ~295648 (only counting for the absorber tubes)

(Recap) Final geometry

- The final geometry is obtained with a simple repetition of the ϕ-slice around the z-axis (right endcap, $z > 0$), and a reflection + translation for the left endcap ($z < 0$)



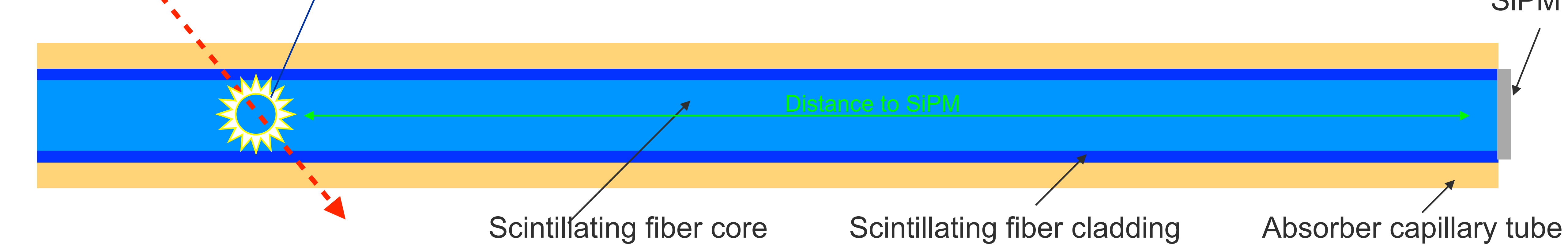


Signal treatment

- This calorimeter simulation must include two signals:
  Scintillation light in scintillator-doped optical fibers and *Cherenkov light* in clear optical fibers
- Simulating light in calorimeters has always been a problem:
  as of today no LHC Experiment simulation include light propagation in calorimeters, instead it is parametrized based on experimental inputs
- I believe the same should be adopted for the IDEA calorimeter simulation as a baseline to allow large MC productions
  - More advanced implementation are always possible but should be addressed in terms of physics improvements and CPU cost
- The following is an implementation as a DD4hep Geant4SensitiveAction<>

Scintillation signal simulation

* optical photons are produced *isotropically* around the step position → no need to take into consideration their direction in the Monte Carlo
* the amount of photons produced is related to the energy deposited via the Birks Law
* on a step-by-step basis it fluctuates according to Poissonian statistics
* light collected at the SiPM is attenuated according to the fiber attenuation length



- ✦ This approach was validated against test-beam data from 2021 finding a good MC-to-data agreement [Article]
  - ✤ It will be refined using new results from 2023 and 2024 test beams which are being analyzed

Creation of Scintillating fibers signal

```
if(!IsCher){ // it is a scintillating fiber

  m_userData.fEvtAction->AddEdepScin(Edep);

  if ( aStep->GetTrack()->GetDefinition()->GetPDGCharge() == 0 || steplength == 0. )
    return true; // not ionizing particle
  }

  G4double distance_to_sipm = DREndcapTubesSglHpr::GetDistanceToSiPM(aStep);

  signalhit = DREndcapTubesSglHpr::SmearSSignal( DREndcapTubesSglHpr::ApplyBirks( Edep,
steplength ) );

  signalhit = DREndcapTubesSglHpr::AttenuateSSignal(signalhit, distance_to_sipm);

  if(signalhit == 0) return true;

  m_userData.fEvtAction->AddSglScin(signalhit);

} // end of scintillating fibre signal calculation
```

Skip step for non ionizing particles

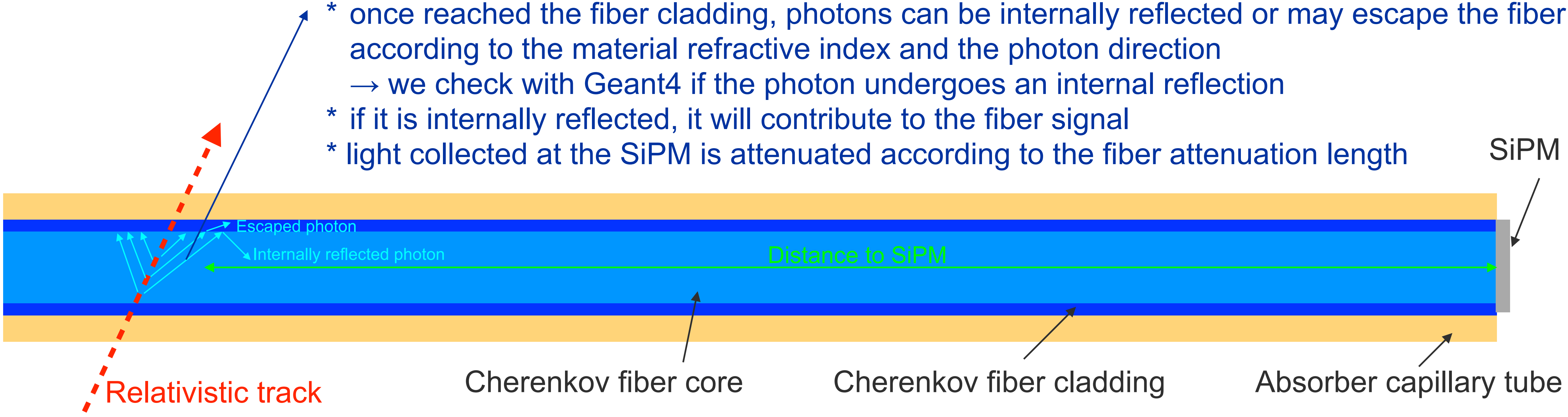
Calculate hit-SiPM distance

Apply Birks Law and smear according to Poissonian fluctuations

Apply light attenuation in fibers

Cherenkov signal simulation

* optical photons direction is calculated by the Geant4 Cherenkov process
  → the emission of the optical photon must be included in the Monte Carlo
* once reached the fiber cladding, photons can be internally reflected or may escape the fiber according to the material refractive index and the photon direction
  → we check with Geant4 if the photon undergoes an internal reflection
* if it is internally reflected, it will contribute to the fiber signal
* light collected at the SiPM is attenuated according to the fiber attenuation length



- This approach was validated against test-beam data from 2021 finding a good MC-to-data agreement [Article]
  - It will be refined using new results from 2023 and 2024 test beams which are being analyzed

Creation of Cherenkov fibers signal

```
else{ // it is a Cherenkov fiber
  m_userData.fEvtAction->AddEdepCher(Edep);
  // calculate the signal in terms of Cherenkov photo-electrons
  if ( aStep->GetTrack()->GetParticleDefinition() == G4OpticalPhoton::Definition() ) {

  // skipped code to check optical boundary process

    switch ( theStatus ){
      case TotalInternalReflection: {
        G4double distance_to_sipm = DREndcapTubesSglHpr::GetDistanceToSiPM(aStep);
        G4int c_signal = DREndcapTubesSglHpr::SmearCSignal( );
        signalhit = DREndcapTubesSglHpr::AttenuateCSignal(c_signal, distance_to_sipm);
        if(signalhit == 0) return true;
        m_userData.fEvtAction->AddSglCher(signalhit);
        aStep->GetTrack()->SetTrackStatus( fStopAndKill );
        break;
    }
    default:
      aStep->GetTrack()->SetTrackStatus( fStopAndKill );
      return true;
    } //end of switch cases

  } //end of optical photon
  else {return true;}

} //end of Cherenkov fiber
```

Some code skipped here

It is a photon reflected in fiber

Calculate distance to SiPM

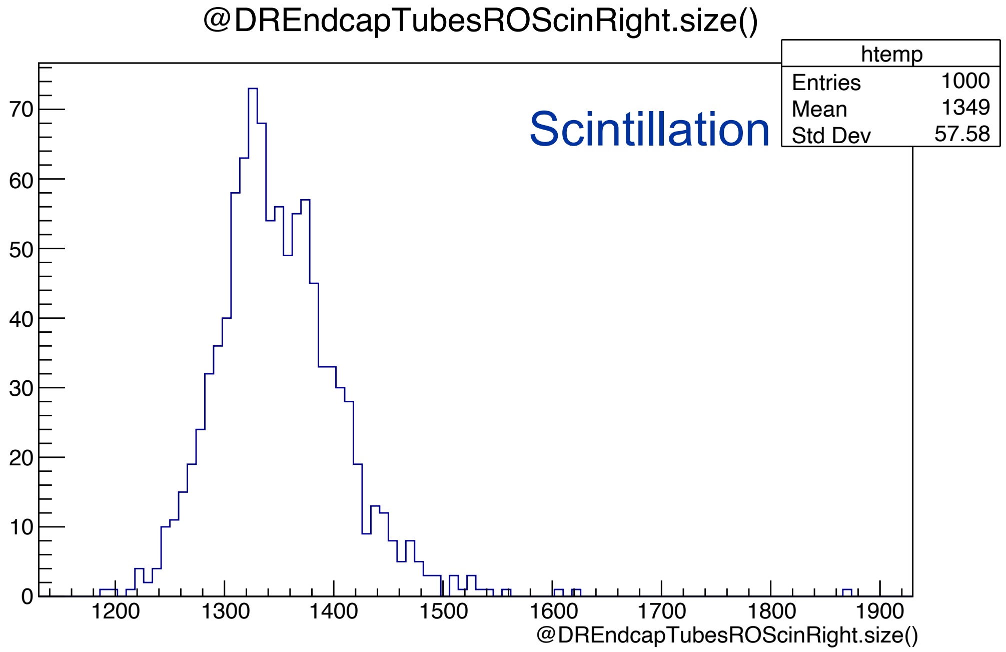
Attenuate light in fiber

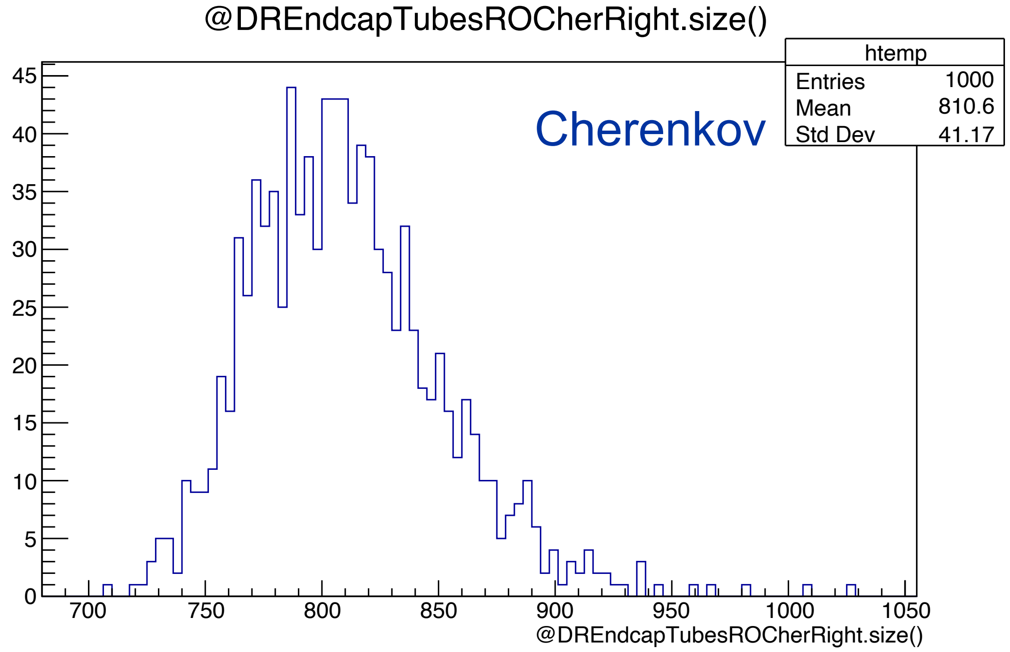
Then do no track the photon

Health checks

- An edm4hep hit collection is created per each endcap (right and left) and per each fiber type (Scintillation and Cherenkov)
- A hit is created per each fiber with a signal (photo-electrons) above 0, i.e. applying a zero-suppression → the hit collection size represents the number of fibers with a signal in the event

Number of fibers readout in each event @90 GeV e^-
~1300 Scintillating fibers and ~800 Cherenkov fibers

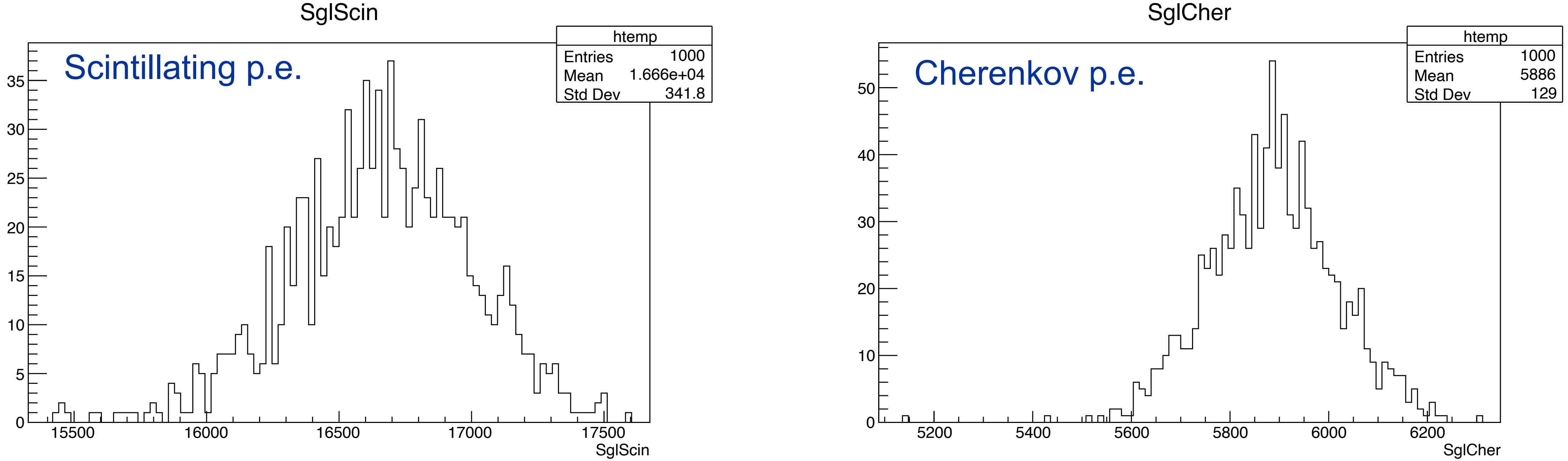




Health checks

- The simulated signal in terms of photo-electrons is set to ~180 Scintillating p.e./GeV and ~66 Cherenkov p.e./GeV
- It will be re-tuned according to new experimental findings as they come from test-beams

Total signal in photo-electrons @90 GeV e^-
~180 Scintillating p.e./GeV and ~66 Cherenkov p.e./GeV



Event rate

- Results from key4hep-nightlies on an lxplus9 machine with 2.9 GHz (single-threaded)*
- Event time is taken using `G4Timer` between `BeginOfRunAction()` and `EndOfRunAction()`

```
ParticleHandler   INFO  +++ Event 9 Begin event action. Access event related information.
Geant4Output2EDM4hep INFO  +++ Saving EDM4hep event 9 run 0.
GenerationInit    WARN  +++ Finished run 0 after 10 events (10 events in total)
=====================================================================
 --> DREndcapTubes: run terminated, 10 events transported
 --> DREndcapTubes: time: User=6.820000s Real=7.429838s Sys=0.090000s [Cpu=93.0%]
 --> DREndcapTubes: time per event: 0.682
=====================================================================
```

- Initialization time is ~6 min:
  - 1 min to construct DD4hep/TGeo geometry, 1.5 min to setup Geant4 and convert geometry, 3 min for regex sensitive detector search, 0.5 min to start event 0

electron	s/evt	s/evt/GeV
10 GeV	0,64-0,70	~0,065
90 GeV	5,7-6,0	~0,065
160 GeV	10,4-10,7	~0,065
240 GeV	16,2-16,7	~0,067
365 GeV	24,8-25,1	~0,068

*I found it difficult to test the event rate on lxplus machines due to the different load each machine experiences, for an apple-to-apple comparison we should use a dedicated machine

Memory footprint

- A DD4hep simulation using this sub detector has a memory footprint of ~15.4 Gb
  - 2.1 Gb for the DD4hep/TGeo geometry representation
  - 2.2 Gb for the Geant4 geometry representation
  - 11.1 Gb of information cached during the geometry conversion

Default DD4hep

```
 Main   I/O
    PID△USER       PRI  NI  VIRT   RES   SHR S  CPU% MEM%   TIME+  Command
1557572 lopezzot    20   0 20096  7808  3840 S   0.0  0.0  0:00.15 -bash
1727002 lopezzot    20   0 19208  7040  3840 S   0.0  0.0  0:00.04 -bash
1733818 lopezzot    20   0 15.7G 15.4G  298M R  99.5 54.3  7:25.95 python3 /cvmfs/sw-nightlies.hsf.org/key4hep/releases/2024-09-11/x86_64-almalinux9-gcc11.4.1-opt/dd4hep/a516ab9c37575905c3f
```

- DD4hep developers (many thanks to Markus and Andre) introduced `regexSensitiveDetector`, it allows to
  - Associate a sensitive (detector) action to every volume with name matching a substring pattern
  - No volume is marked as *sensitive* in geometry construction → no cache is created
  - Memory footprint is reduced to 4.3 Gb (de facto only geometry)

DD4hep + regexSensitiveDetector

```
 Main   I/O
    PID△USER       PRI  NI  VIRT   RES   SHR S  CPU% MEM%   TIME+  Command
1557572 lopezzot    20   0 20096  7808  3840 S   0.0  0.0  0:00.15 -bash
1727002 lopezzot    20   0 19208  7040  3840 S   0.0  0.0  0:00.04 -bash
1758347 lopezzot    20   0 4581M 4279M  304M R  93.8 14.7  7:39.38 python3 /cvmfs/sw-nightlies.hsf.org/key4hep/releases/2024-09-11/x86_64-almalinux9-gcc11.4.1-opt/dd4hep/a516ab9c37575905c3f
```

About `regexSensitiveDetector`

- `regexSensitiveDetector` unfortunately is not a free lunch. Some drawbacks:
  - Initialization time is increased by 3 minutes (we can live with it)
  - Some DD4hep methods are not working any longer, for instance getting TGeo volIDs:

Retrieving DD4hep/TGep volIDs from the G4Step

```
dd4hep::BitFieldCoder decoder("endcap:1,stave:10,tower:8,air:1,col:10,row:7,clad:1,core:1,cherenkov:1");
auto VolID = volumeID(aStep);
CherenkovID = decoder.get(VolID,"cherenkov");
```

→ `volumeID()` does not work without cached information

About `regexSensitiveDetector`

- `regexSensitiveDetector` unfortunately is not a free lunch. Some drawbacks:
  - Initialization time is increased by 3 minutes (we can live with it)
  - Some DD4hep methods are not working any longer, for instance getting TGeo volIDs:

Retrieving DD4hep/TGep volIDs from the G4Step

```
dd4hep::BitFieldCoder decoder("endcap:1,stave:10,tower:8,air:1,col:10,row:7,clad:1,core:1,cherenkov:1");
auto VolID = volumeID(aStep);
CherenkovID = decoder.get(VolID,"cherenkov");
```

→ `volumeID()` does not work without cached information

  - A possible brute force workaround

Mapping G4Volumes to TGeoVolumes

```
G4VPhysicalVolume* PhysVol = aStep->GetPreStepPoint()->GetTouchableHandle()->GetVolume();
Geant4Mapping& Mapping = Geant4Mapping::instance();
PlacedVolume PlacedVol = Mapping.placement(PhysVol);
const PlacedVolumeExtension::VolIDs& TestIDs = PlacedVol.volIDs();
auto it = TestIDs.find("name");
std::cout<<it->first<<" "<<it->second<<std::endl;
```

→ `works, but slows down the simulation a factor ~350`

About regexSensitiveDetector

- Accessing TGeo volIDs from the G4Step becomes unfeasible
- Luckily, DD4hep still offers the possibility to set copynumbers in TGeo volumes (together with volIDs) and propagate them till the corresponding Geant4 volume construction. For instance

DD4hep

```
PlacedVolume Volume::placeVolume(const Volume& volume, int copy_no) const {
  return _addNode(m_element, volume, copy_no, detail::matrix::_identity());}
```

DD4hep

```
PlacedVolume _addNode(TGeoVolume* par, TGeoVolume* daughter, int id, TGeoMatrix* transform) {
  TGeoVolume* parent = par;
  // a lot of things
  /* n = */ parent->AddNode(daughter, id, transform);}
```

TGeo

```
void TGeoVolume::AddNode(TGeoVolume *vol, Int_t copy_no, TGeoMatrix *mat, Option_t * /*option*/){
  TGeoNodeMatrix *node = 0;
  node = new TGeoNodeMatrix(vol, matrix);
  node->SetNumber(copy_no);}
```

About `regexSensitiveDetector`

- Accessing TGeo volIDs from the G4Step becomes unfeasible
- Luckily, DD4hep still offers the possibility to set copynumbers in TGeo volumes (together with volIDs) and propagate them till the corresponding Geant4 volume construction
- The obvious solution for us is to use copynumbers (instead of volIDs) as in Geant4 (`aStep->GetPreStepPoint()->GetTouchable()->GetCopyNumber();`)

About `regexSensitiveDetector`

- Accessing TGeo volIDs from the G4Step becomes unfeasible
- Luckily, DD4hep still offers the possibility to set copynumbers in TGeo volumes (together with volIDs) and propagate them till the corresponding Geant4 volume construction
- The obvious solution for us is to use copynumbers (instead of volIDs) as in Geant4 (`aStep->GetPreStepPoint()->GetTouchable()->GetCopyNumber();`)
- However, it would be good to recreate the volID of each sensitive volume starting from the copynumber of the volume history → it would allow `DD4hep::BitFieldCoder` usage also in the analysis stage
  - Is it possible to reconstruct the TGeo volIDs from Geant4 copynumbers (see also issue#1319)?
  - Not sure…
    - as volIDs are 64 bits, while copy numbers are int (32 bits)
    - One TGeo volume can have multiple volIDs (`volume.addvolID("name1",1).addvolID("name2",2)`) while Geant4 volumes can only have one copynumber

Conclusion

- A DD4hep description of the IDEA dual-readout tubes-based endcap calorimeter was developed over the summer
  - Together with the barrel geometry proposed by Andreas completes this subdetector
- I proposed a DD4hep sensitive action that allows simulating 10 GeV e^- events at a rate of $\simeq 0.6$ s/evt
  - This approach was validated against test-beam data with good results and might be a common baseline for these simulations
- The large memory footprint problem ($\simeq 15$ Gb) can be avoided using `regexSensitiveDetector` with some caveat when accessing TGeo volumes information from Geant4 volumes
- If you agree I can prepare a PR to merge this within key4geo

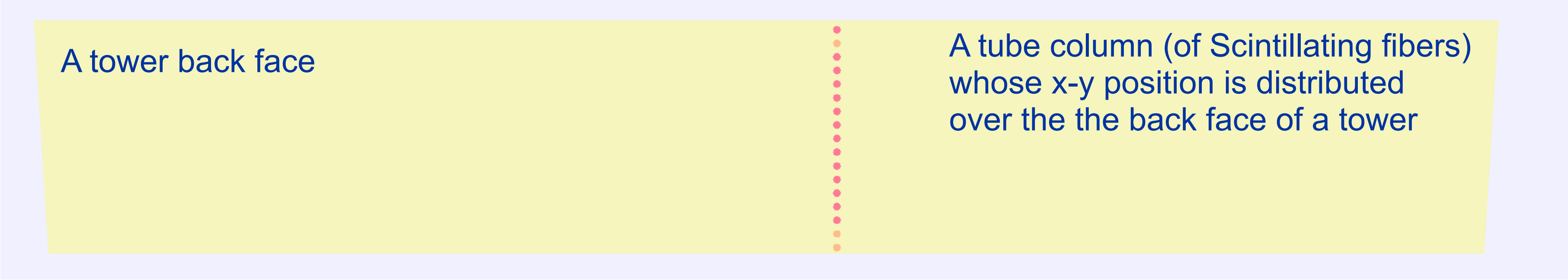
Backup material

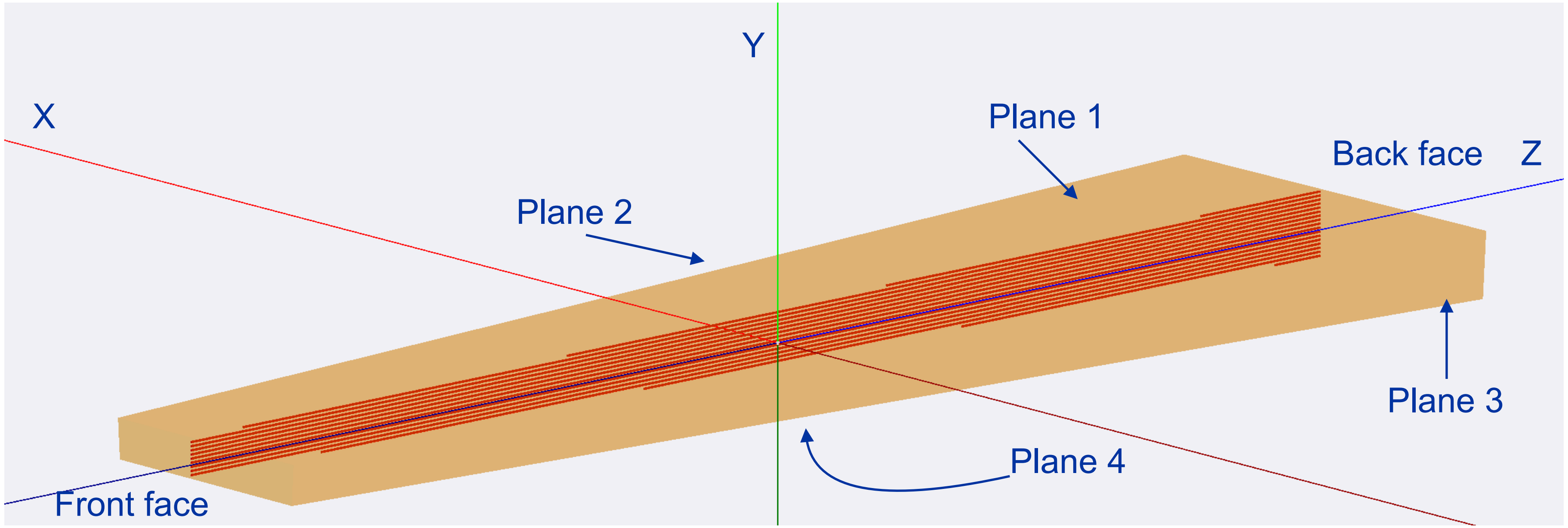
Geometry nesting

- The capillary tubes technology has no “tower” object, instead it glues together (many!) tubes to create a tower (trapezoid). However,
  - Placing in the simulation individual tubes directly in one endcap (or a ϕ-slice of it) is not the ideal solution as
  - navigating through many volumes is always a bad idea (even if we have voxelization),
  - also, having set of tubes housed in one “tower” will help the reconstruction, for instance for the calibration when each tower will be calibrated based on its signal and deposited energy from the MC truth
- Better to have “towers” of Air material and placing tubes inside each tower
  - Overall four levels of nesting are used

ϕ-slice (or stave)
Air → θ-towers
Air → Tubes
Metal (e.g. brass) → Optical fibers
Plastic (sensitive elements)

Tubes placement (y-direction)

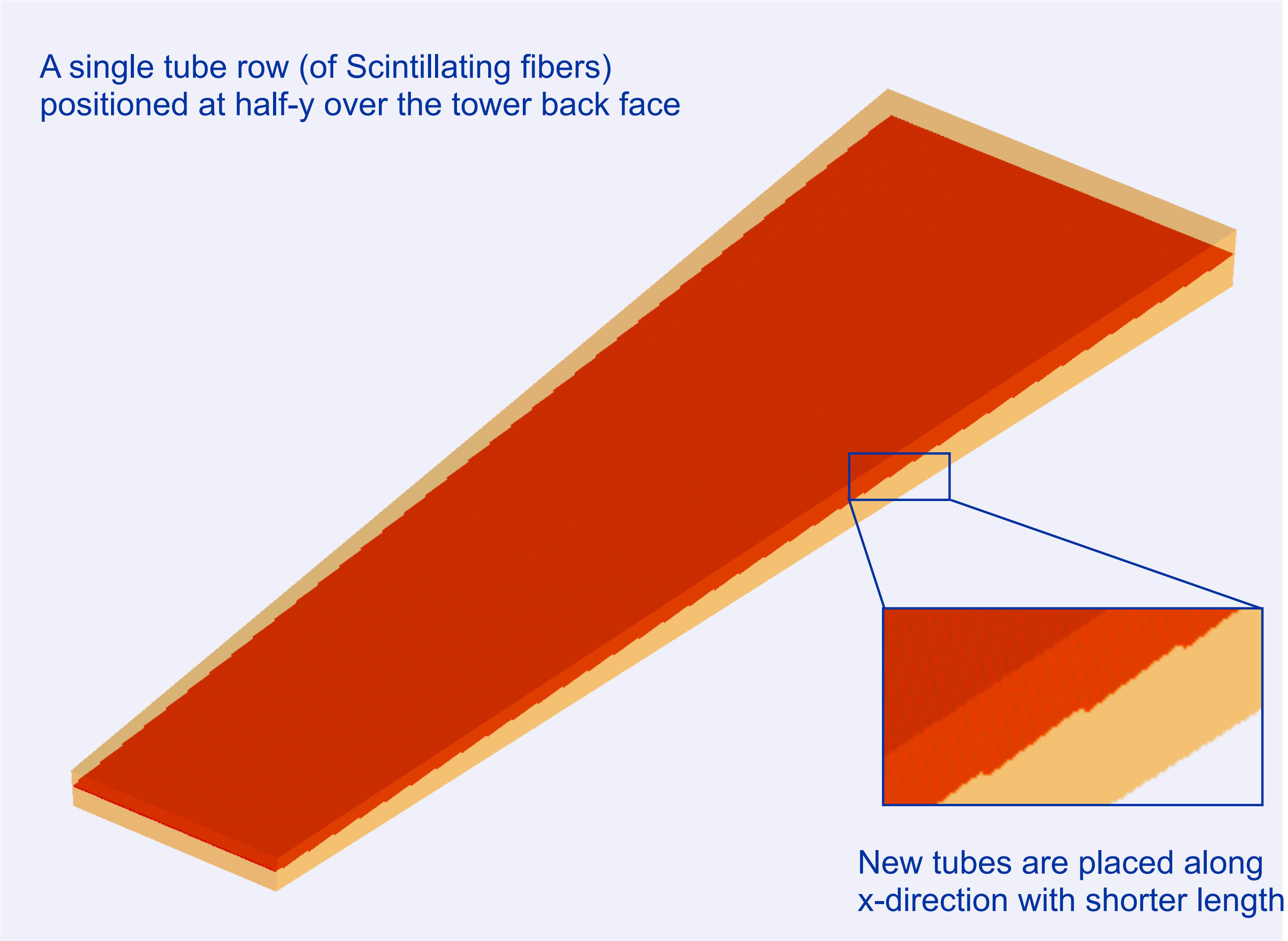




Tube radius was increased to 2 mm to help visualization

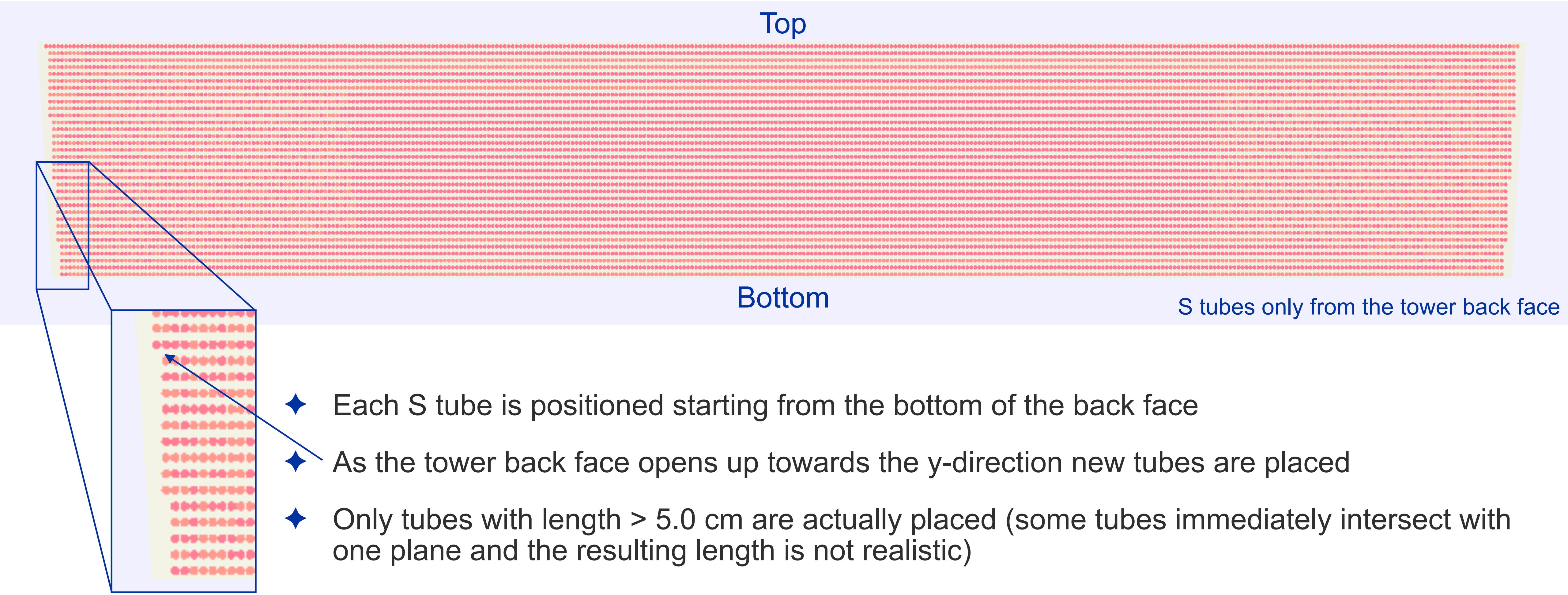
- For each tube starting at x-y from the back face, I calculate the intersection of a line parallel to the z-axis passing through x-y with each of the 4 planes in figure + the front face
- The shortest intersection defines the tube length
  - In reality some corrections are needed because the intersection happens between a plane and a cylinder (the tube)
- Tubes intersecting with the front face have the same length and are represented by the same Solid object in memory
- Tubes intersecting with any other plane are of specific length and are created on the fly

Tubes placement (x-direction)



Tube radius was increased to 2 mm to help visualization

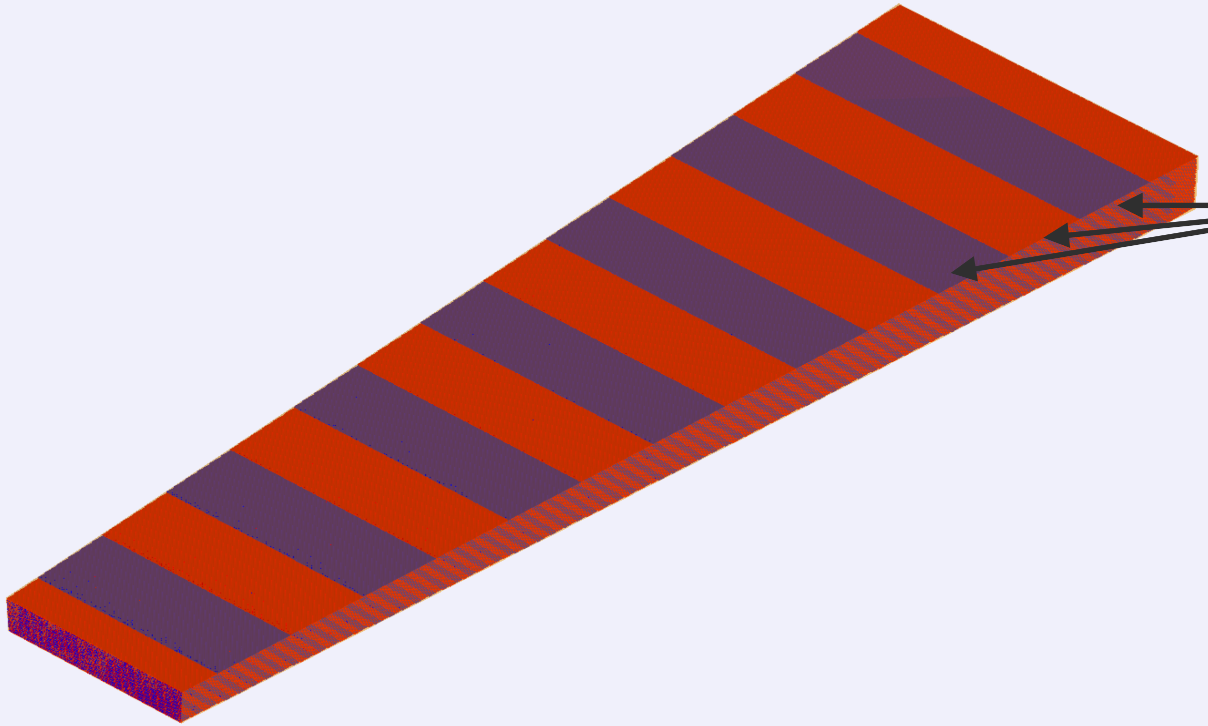
Filling the towers (Scintillating tubes only)



- Each S tube is positioned starting from the bottom of the back face
- As the tower back face opens up towards the y-direction new tubes are placed
- Only tubes with length > 5.0 cm are actually placed (some tubes immediately intersect with one plane and the resulting length is not realistic)

Filling the towers (all tubes)

A single tower filled with 24477 tubes

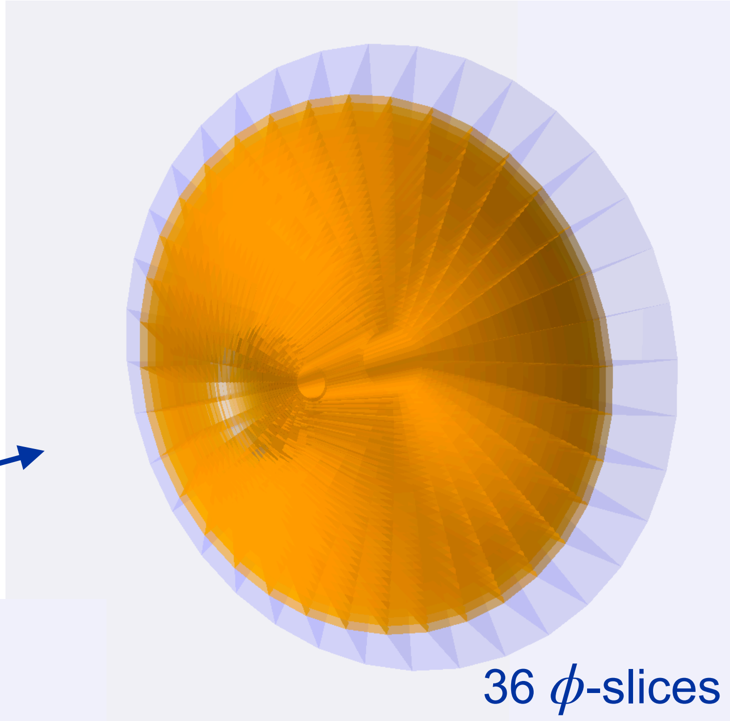


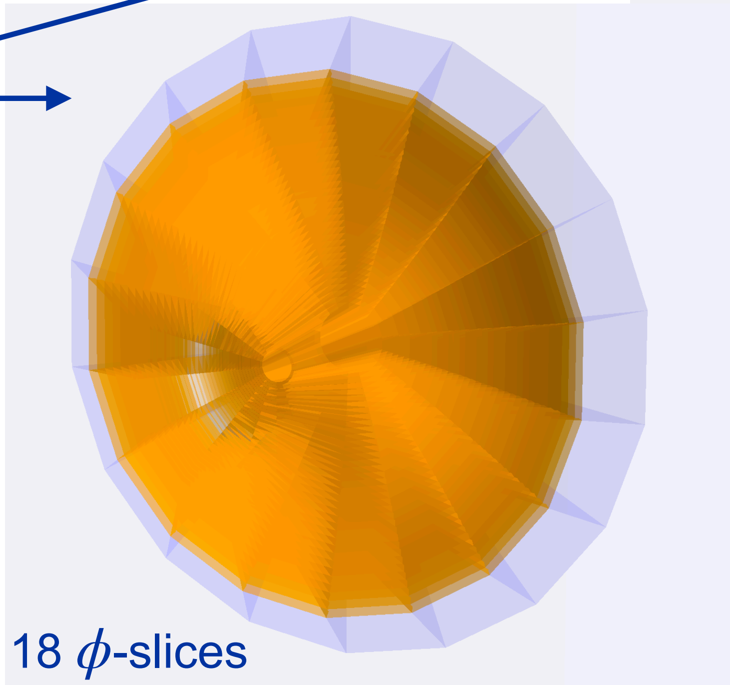
- A tower fully filled with tubes (S and C)
- S and C tubes alternates in y direction and the y-position determines the tube length
  - hence the colored bands visible from the top view

Modularity

- All the geometry custom parameters are encapsulated in the XML description file

```
<define>
  <constant name="world_side"  value="6*m"/>
  <constant name="world_x"     value="world_side/2"/>
  <constant name="world_y"     value="world_side/2"/>
  <constant name="world_z"     value="world_side/2"/>
  <constant name="innerRadius"value="2.5*m"/>
  <constant name="towerHeight"value="2.5*m"/>
  <constant name="NbOfZRot"    value="36"/>
  <constant name="TubeRadius" value="1.0*mm"/>
  <constant name="CladRadius" value="0.5*mm"/>
  <constant name="CoreRadius" value="0.45*mm"/>
</define>
```





Modularity

- All the geometry custom parameters are encapsulated in the XML description file

```
<define>
  <constant name="world_side"  value="6*m"/>
  <constant name="world_x"     value="world_side/2"/>
  <constant name="world_y"     value="world_side/2"/>
  <constant name="world_z"     value="world_side/2"/>
  <constant name="innerRadius"value="2.5*m"/>
  <constant name="towerHeight"value="2.5*m"/>
  <constant name="NbOfZRot"   value="36"/>
  <constant name="TubeRadius" value="1.0*mm"/>
  <constant name="CladRadius" value="0.5*mm"/>
  <constant name="CoreRadius" value="0.45*mm"/>
</define>
```

- The only geometry constrain: the barrel inner radius and the endcap inner radius are identical or, equivalently, the endcap starts at $\theta = \pi/4$