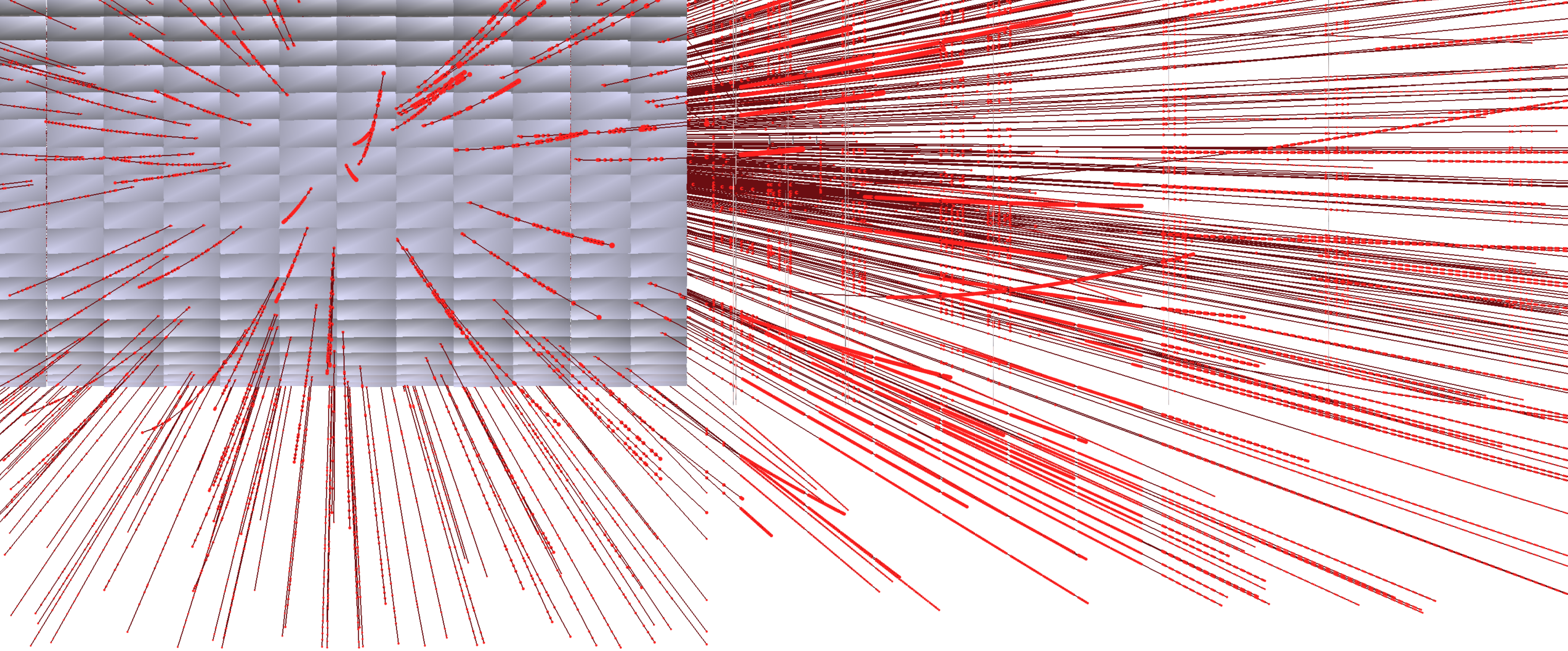
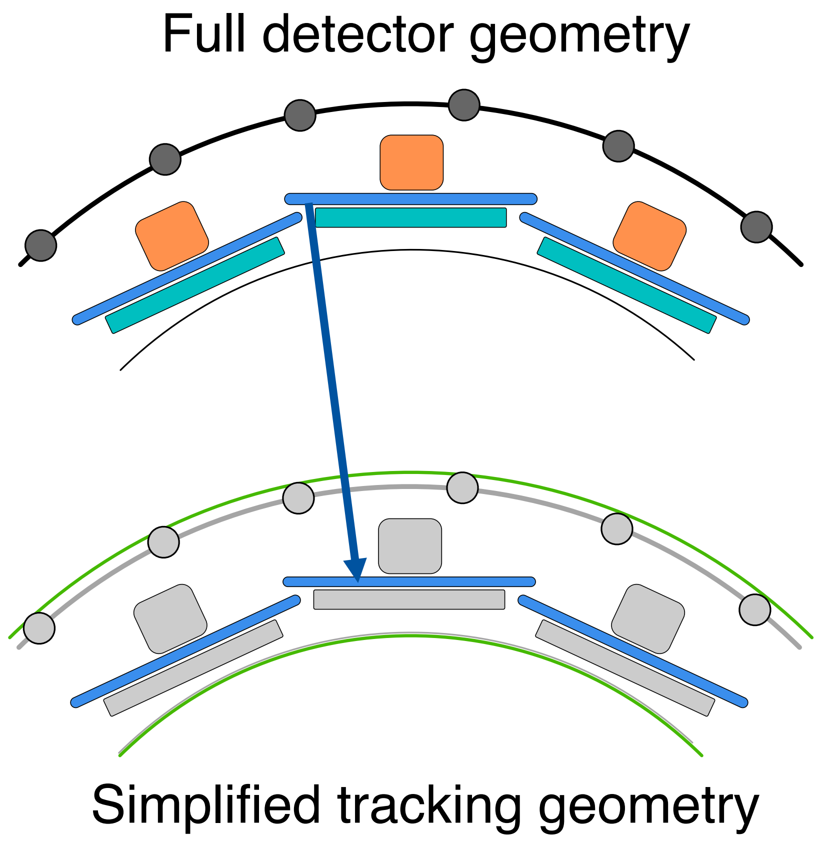


# Geometry construction and navigation

Paul Gessinger - 01/15/2019 - Tracking workshop for HEP - LBNL

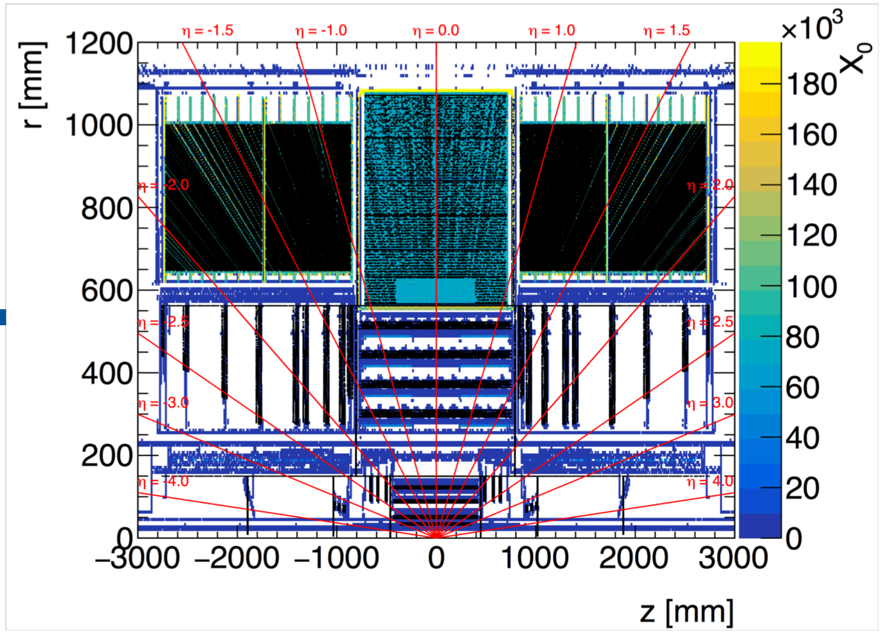
# Acts uses simplified detector geometry

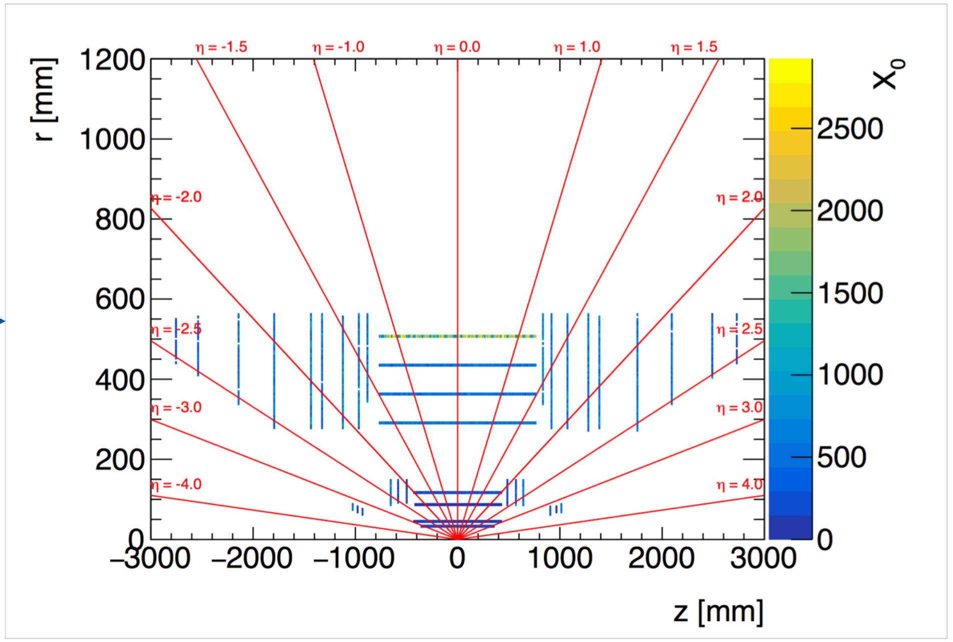
- Simulation uses full detailed geometry
- Volumes hierarchy, with material
- (Almost) everything is modelled
- Acts: **simplified tracking geometry**
- Only sensitive surfaces are translated (more-or-less) on—to-one



# Material: mapped

- Material effects are incorporated using mapped material
  - Use geantinos to probe complex geometry
  - Assign to closest material layer, aggregate
  - Average and write out material
- Next geometry construction can load material maps
- Propagation can query material and incorporate

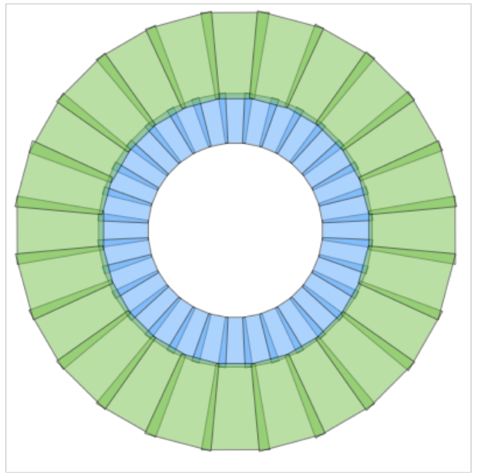
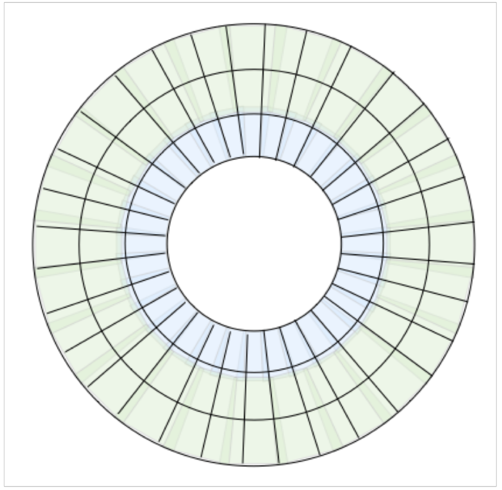


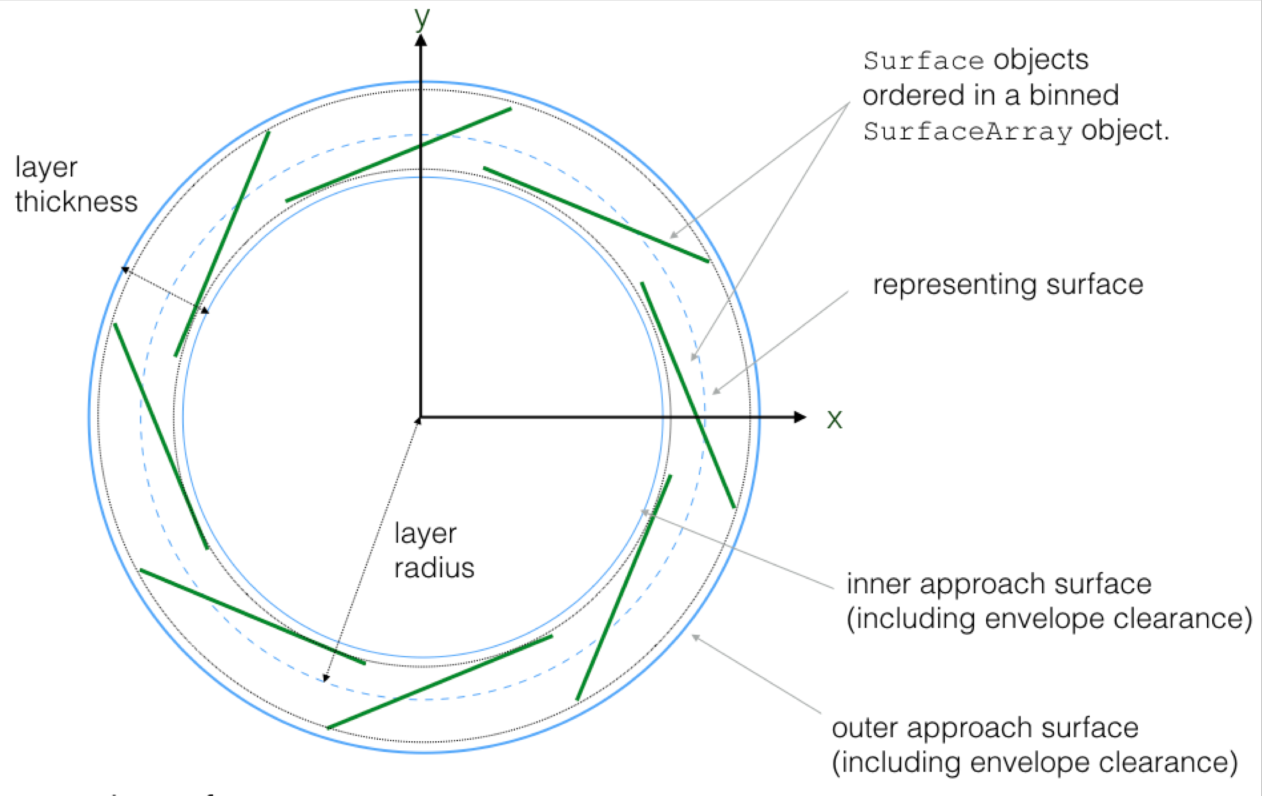


## Typical setup for silicon detectors

# Layers

- Sensitive surfaces are grouped in **Layers**
- Layers considered *thin* (as in, they have a thickness, but it's small)
- Layers are **binned** to allow fast access to contained sensitive surfaces

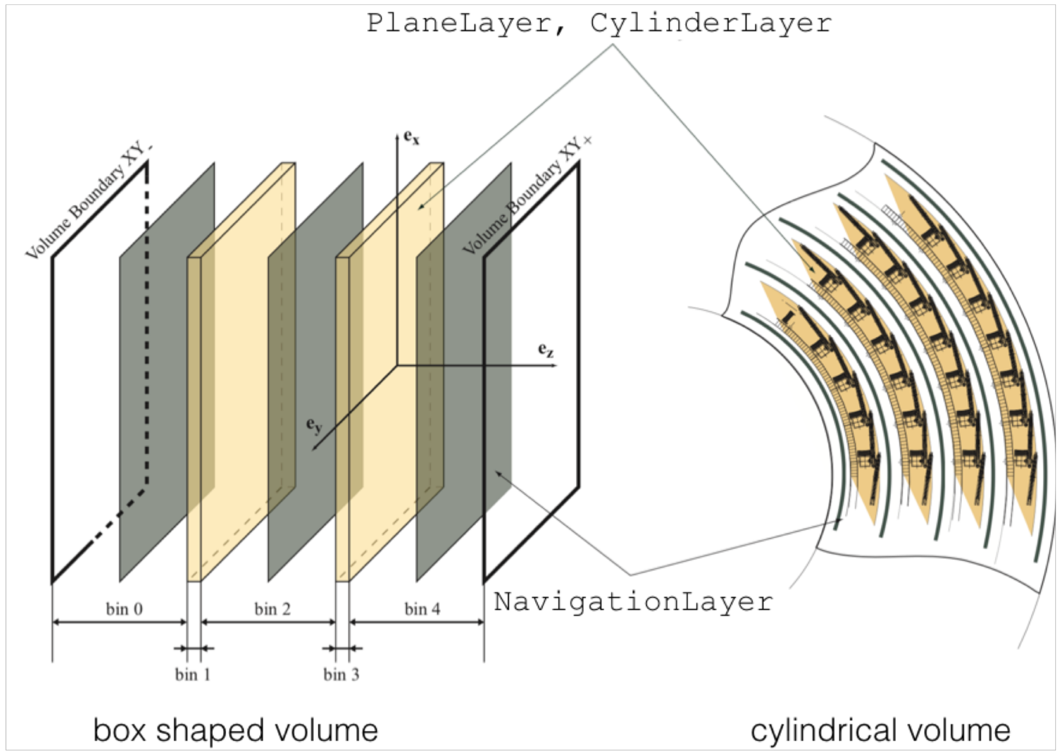



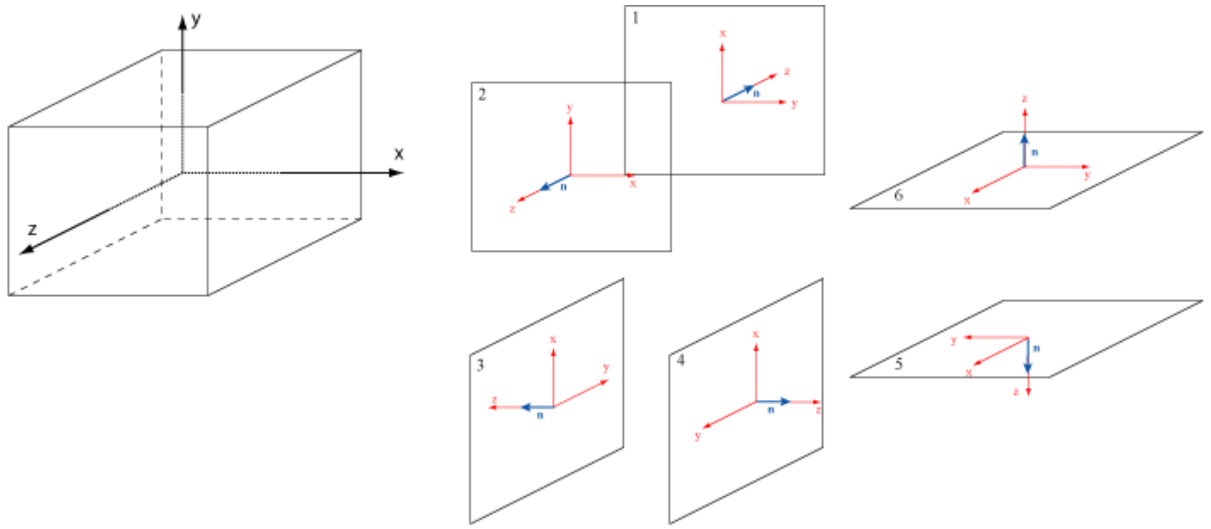

y
x
Surface objects ordered in a binned SurfaceArray object.
layer thickness
representing surface
layer radius
inner approach surface (including envelope clearance)
outer approach surface (including envelope clearance)

x-y view of a `CylinderLayer` (with planar detection elements)

# Volumes

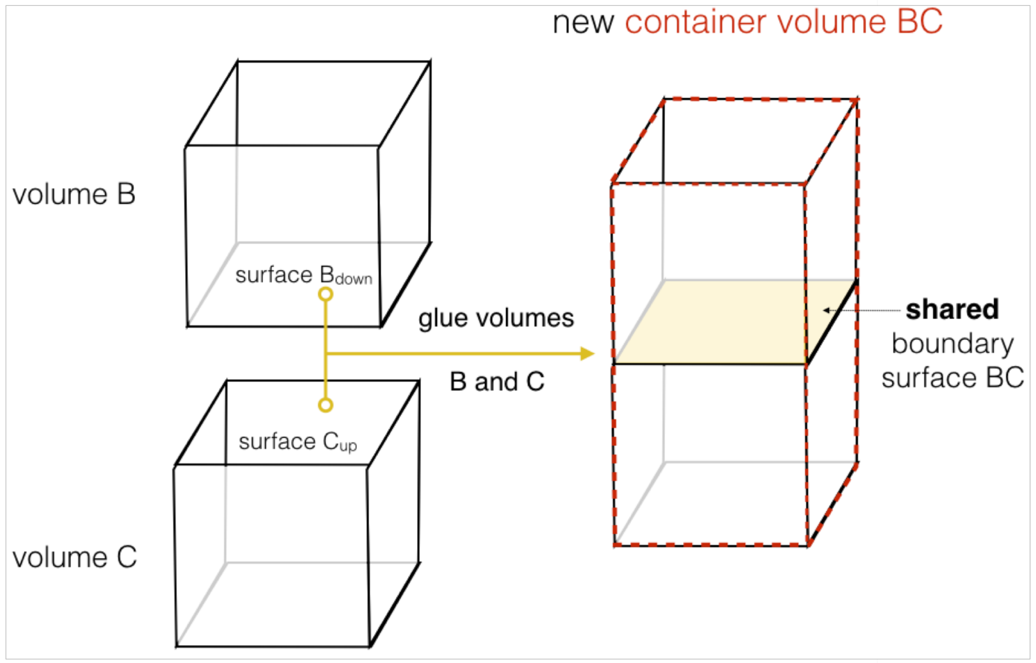
- **Volumes** can contain:
  - Layers
  - Floating volumes
  - Contained volumes
- They are ordered and stored in a *LayerArray*
- Volumes form a hierarchical tree
  - There is one **top volume**
- Volume has a set of *boundary surfaces* (decomposision)

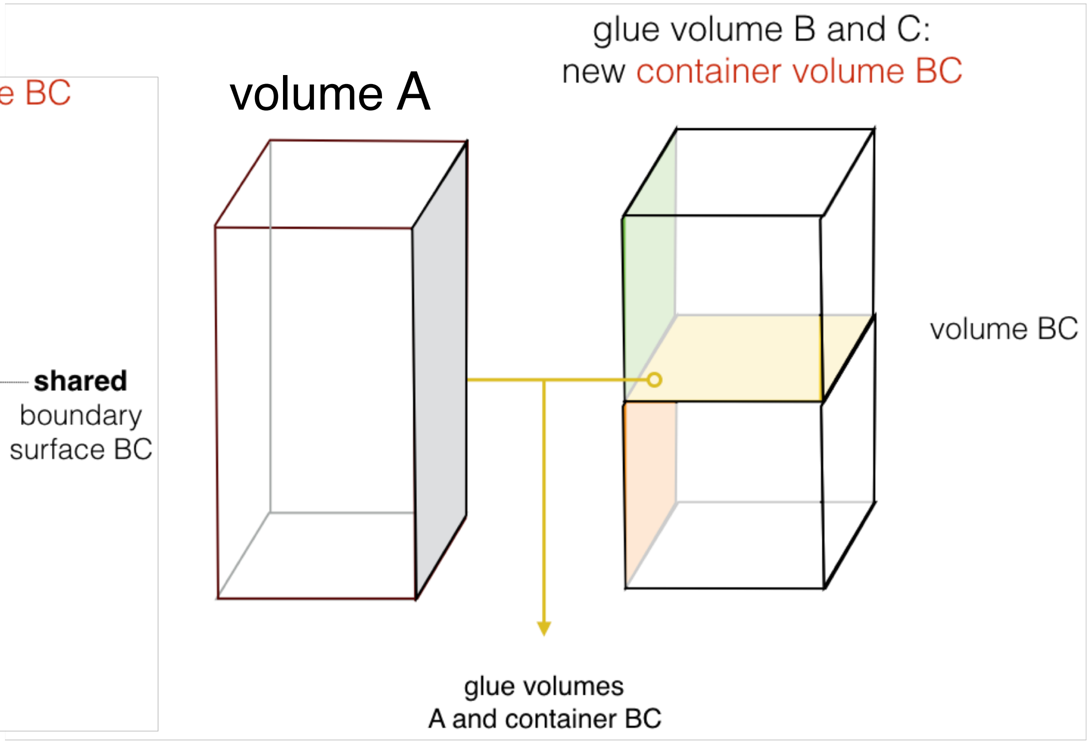


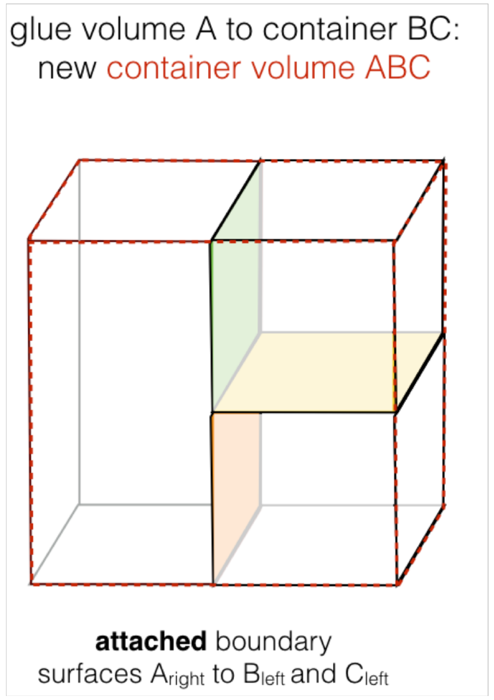


# Navigation through volumes

- Navigation always starts within one volume (innermost, for example)
- Every boundary surface is ***glued*** to the next volume
- During geometry construction, volumes have to be glued together

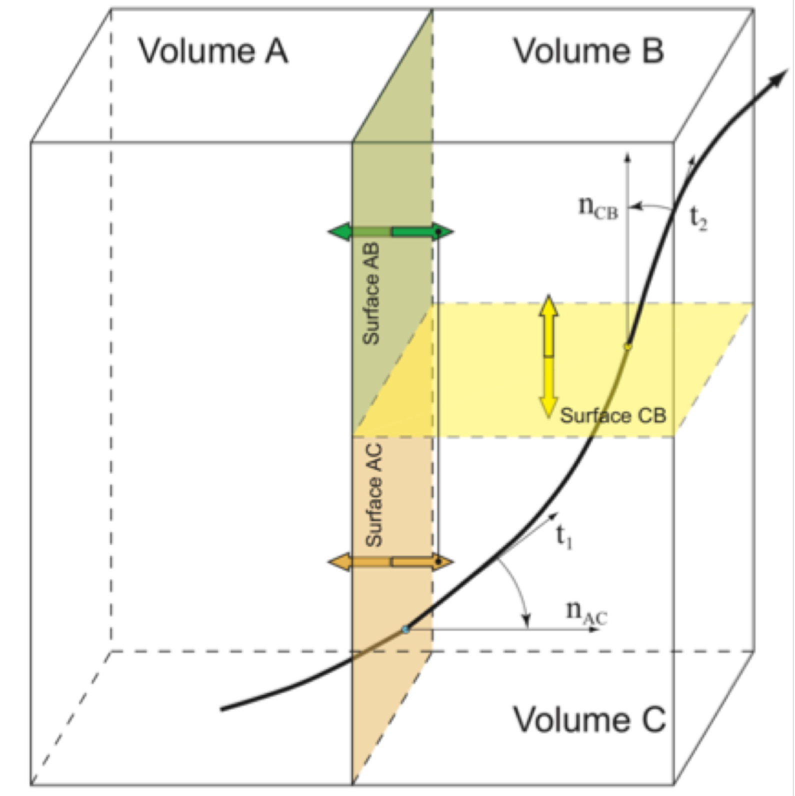





glue volume A to container BC:
new container volume ABC
**attached** boundary surfaces $A_{right}$ to $B_{left}$ and $C_{left}$

# Navigation through volumes

- Boundary surfaces act as *portals* to the next volume



resulting navigation through the boundary portals

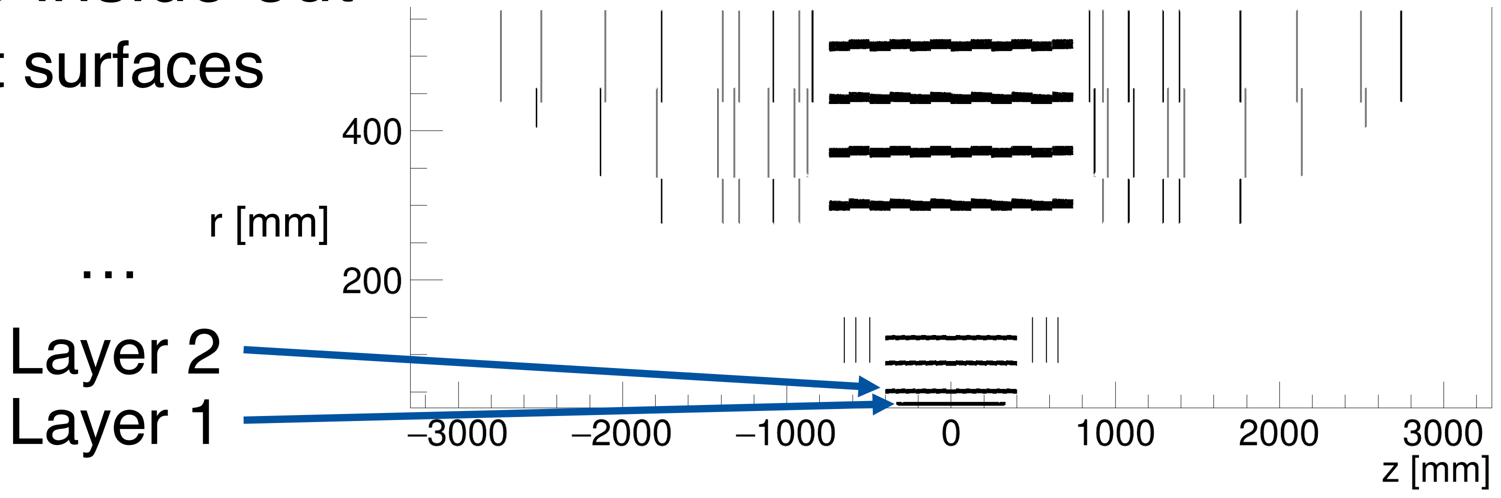
# Navigation through the TrackingGeometry

- Is handled in the Acts propagator by the *Navigator*
- m_navigator.status() figures out where we are at this step
  - Handles volume to volume, layer to layer, surface to surface loop
- m_navigator.target() sets up the next target surfaces
  - Resets current surface
- Navigator considers all types of surfaces (sensitive, approach, boundary, passive)

```
template <typename result_t, typename propagator_state_t>
Status
propagate_impl(result_t& result, propagator_state_t& state) const
{
  // Navigator initialize state call
  m_navigator.status(state);
  // Pre-Stepping call to the action list
  state.options.actionList(state, result);
  if (!state.options.abortList(result, state)) {
    // Pre-Stepping: target setting
    m_navigator.target(state);
    // Stepping loop
    for (; result.steps < state.options.maxSteps; ++result.steps) {
      // Perform a propagation step - it takes the propagation state
      double s = m_stepper.step(state);
      // ...
      // figure out where we are
      m_navigator.status(state);
      // call action list
      state.options.actionList(state, result);
      // check if abort
      if (state.options.abortList(result, state)) {
        break;
      }
      // set next target after abort
      m_navigator.target(state);
    }
  }
  // ...
  state.options.actionList(state, result);
}
```

# Geometry construction: barrel-endcap design

- Barrel layers: concentric hollow cylinders which contain each other successively
- Endcap layers: disks along +-z, which contain each other in r and do not overlap
- Assumption: you have geometry description that logically groups sensitive surfaces into detector groups / volumes / layers
- Basically: you want to go inside out
    - Start in the center, collect surfaces



## Example: ATLAS geometry construction

- … of the Pixel and SCT
- Typical, barrel-endcap design
- What do we need?
- ATLAS-Acts implementation: TrackingGeometry service which initializes geometry building and owns the tracking geometry

# Exampl cont.

We need:

- `LayerArrayCreator`
- `TrackingVolumeArrayCreator`
- `CylinderVolumeHelper`
- `VolumeBuilders` for every subdetector
- `TrackingGeometryBuilder`

And we get a `TrackingGeometry`

```
std::list<std::shared_ptr<const Acts::ITrackingVolumeBuilder>> volumeBuilders;

auto layerArrayCreator = std::make_shared<const Acts::LayerArrayCreator>(
      makeActsAthenaLogger(this, "LayArrCrtr", "ActsTGSvc"));

auto trackingVolumeArrayCreator
      = std::make_shared<const Acts::TrackingVolumeArrayCreator>(
          makeActsAthenaLogger(this, "TrkVolArrCrtr", "ActsTGSvc"));

Acts::CylinderVolumeHelper::Config cvhConfig;
cvhConfig.layerArrayCreator          = layerArrayCreator;
cvhConfig.trackingVolumeArrayCreator = trackingVolumeArrayCreator;

auto cylinderVolumeHelper
    = std::make_shared<const Acts::CylinderVolumeHelper>(
        cvhConfig, makeActsAthenaLogger(this, "CylVolHlpr", "ActsTGSvc"));

volumeBuilders.push_back(makeVolumeBuilder(p_pixelManager, cylinderVolumeHelper, true));

// SCT
volumeBuilders.push_back(makeVolumeBuilder(p_SCTManager, cylinderVolumeHelper));

// TRT (this is a bit different)
volumeBuilders.push_back(makeVolumeBuilder(p_TRTManager, cylinderVolumeHelper));

Acts::TrackingGeometryBuilder::Config tgbConfig;
tgbConfig.trackingVolumeHelper   = cylinderVolumeHelper;
tgbConfig.trackingVolumeBuilders = volumeBuilders;

auto trackingGeometryBuilder
      = std::make_shared<const Acts::TrackingGeometryBuilder>(tgbConfig,
          makeActsAthenaLogger(this, "TrkGeomBldr", "ActsTGSvc"));

m_trackingGeometry = trackingGeometryBuilder->trackingGeometry();
```

# Exampl cont.

- `LayerBuilder` (`ActsLayerBuilder` here is fully ATLAS specific)
  - `SurfaceArrayCreator`
- `LayerCreator`
- `CylinderVolumeBuilder`

```cpp
std::shared_ptr<const Acts::ILayerBuilder> gmLayerBuilder;
auto matcher = [](Acts::BinningValue bValue, const Acts::Surface* aS,
                  const Acts::Surface* bS) -> bool { /* ... */ };
Acts::SurfaceArrayCreator::Config sacCfg;
sacCfg.surfaceMatcher = matcher;

auto surfaceArrayCreator = std::make_shared<Acts::SurfaceArrayCreator>(
        sacCfg, makeActsAthenaLogger(this, "SrfArrCrtr", "ActsTGSvc"));
Acts::LayerCreator::Config lcCfg;
lcCfg.surfaceArrayCreator = surfaceArrayCreator;

auto layerCreator = std::make_shared<Acts::LayerCreator>(
        lcCfg, makeActsAthenaLogger(this, "LayCrtr", "ActsTGSvc"));

ActsLayerBuilder::Config cfg;
cfg.mng = static_cast<const InDetDD::SiDetectorManager*>(manager);
cfg.elementStore = m_elementStore;
cfg.layerCreator = layerCreator;

gmLayerBuilder = std::make_shared<const ActsLayerBuilder>(cfg,
                 makeActsAthenaLogger(this, "GMLayBldr", "ActsTGSvc"));

Acts::CylinderVolumeBuilder::Config cvbConfig;
cvbConfig.layerEnvelopeR = {0, 0};
cvbConfig.layerEnvelopeZ       = 2;
cvbConfig.trackingVolumeHelper = cvh;
cvbConfig.volumeName           = managerName;
cvbConfig.layerBuilder         = gmLayerBuilder;
cvbConfig.buildToRadiusZero = toBeamline;

auto cylinderVolumeBuilder = std::make_shared<const Acts::CylinderVolumeBuilder>(
                    cvbConfig, makeActsAthenaLogger(this, "CylVolBldr", "ActsTGSvc"));

return cylinderVolumeBuilder;
```

# The components

- LayerCreator: factory for disk, cylinder and plane layers
- LayerArrayCreator: factory for ordered and binned layer array, also creates navigation layers
- LayerBuilder: constructs (translates) sensitive surfaces into layers
  - Uses: LayerCreator, LayerArrayCreator
- CylinderVolumeHelper: functions to create and wrap cylinder volumes
- CylinderVolumeBuilder: factory method for cylinder volume from layers, calls LayerBuilder, uses VolumeHelper
- TrackingGeometryBuilder: calls volume builders in order

# The LayerBuilder

- Exposes (mainly) three methods:
```
virtual const LayerVector negativeLayers() const = 0;
virtual const LayerVector centralLayers() const = 0;
virtual const LayerVector positiveLayers() const = 0;
```
- Can usually be implemented in one, called from these

# The LayerBuilder

- Convert all detector elements
- Figure out how many layers, and which element belongs to which
- Collect layer surfaces, and build layers
- (left out some logic for binning and material)

```
void
ActsLayerBuilder::buildLayers(Acts::LayerVector& layersOutput, int type) {
  std::vector<std::shared_ptr<const ActsDetectorElement>> elements =
                                                    getDetectorElements();
  std::map<std::pair<int, int>, std::vector<const Surface*>> layers;

  for (const auto &element : elements) {
    IdentityHelper id = element->identityHelper();
    if (type == 0 && id.bec() != 0) continue;
    if (type != 0 && id.bec() == 0) continue;
    if (type != 0 && type * id.bec() < 0) continue;

    m_cfg.elementStore->push_back(element);
    std::pair<int, int> layerKey(id.layer_disk(), id.bec());
    if (layers.count(layerKey) == 0) {
      layers.insert(std::make_pair(layerKey, std::vector<const Surface*>()));
    }
    layers.at(layerKey).push_back(&element->surface());
  }
  for (const auto& layerPair : layers) {
    // ...
    std::vector<const Surface*> layerSurfaces = layerPair.second;

    if (type == 0) {  // BARREL
      Acts::ProtoLayer pl(layerSurfaces);
      auto layer = m_cfg.layerCreator->cylinderLayer(layerSurfaces, /* ... */ );
      layersOutput.push_back(layer);
    } else {  // ENDCAP
      Acts::ProtoLayer pl(layerSurfaces);
      auto layer = m_cfg.layerCreator->discLayer(layerSurfaces, /* ... */);
      layersOutput.push_back(layer);
    }
  }
}
```

## Example using CuboidVolumeBuilder

- … and basic setup of the propagation:

https://gitlab.cern.ch/berkeleylab/acts/examples