SPONSORED BY THE

Federal Ministry of Education and Research

EP-R&D

Programme on Technologies for Future Experiments

CERN

# ACTS Workshop 2023

## Geometry Model: `detray`

Joana Niermann on behalf of the `detray` developers

09.11.2023

# The `detray` Project

detray

powered by acts

## Project Outline

- Realistic tracking geometry description and progagation, without compromises in accuracy.
- Geometry classes without run-time polymorphism (in particular, no virtual function calls).
- Flat container structure with index based data linking.
- Implementation of core package equally usable in host and device code.

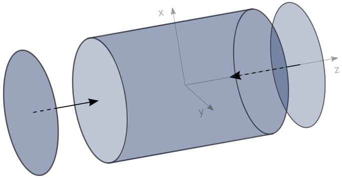
## Heterogeneous Computing Model

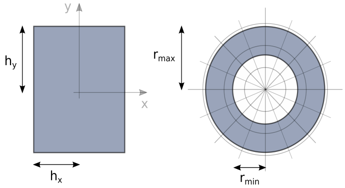
- Core classes templated on `STL` vs. `vecmem` containers.
- Memory allocation strategy is determined by *`vecmem` memory resources*.
- The data structures are built host-side and then passed to the kernel via *data views*
- The copy to device happens into buffers, which are allocated on the device.

Source: https://github.com/acts-project/detray

# Geometry Description

- **Volumes**: defined by their boundary surfaces
- **Surfaces**: Placed by transformations and defined by boundary masks
- **Masks**: Defined by a shape type.
  Specify local coordinates and extent of surfaces.
- **Portals**: Special surfaces that tie volumes together through index links.
- **Material:** Homogeneous *slabs* or *rods* of parametrized material. Many predefined materials available.



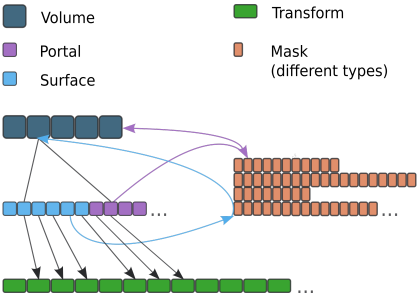




**No abstract classes:** Every type needs its own container. Solved by compile-time *unrolling* of tuple containers.

# Geometry Container Structure

## Linking by Index

- Volumes keep a multi-index to the acceleration data structures.
- Accelerator store: tuple of e.g grids.
- Transform store holds transformation matrices (contextual).
- Mask/material store: tuple of mask/material vectors.
- Surfaces/Portals keep indices into the transform, mask and material containers.
- "Barcode" identifies surfaces uniquely in flat surface vector.

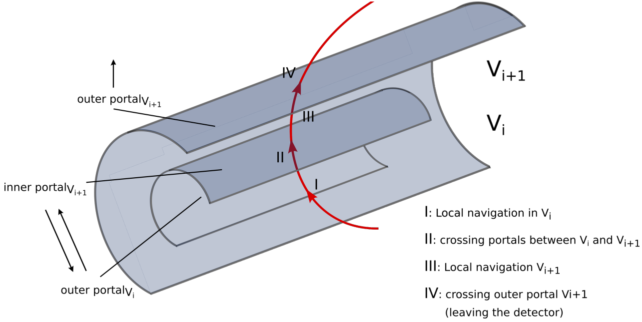



```
geometry::barcode bcd;     // 64 bits
bcd.volume();              // Index of the volume the surface belongs to
bcd.id();                  // Id of the surfaces type (sensitive, passive, portal)
bcd.index();               // Index of the surface in the surface vector.
```

# Track State Propagation

## Participants

- **Propagator:** runs the propagation loop: Calls stepper, navigator and the actors.
- **Navigator:** Moves between detector volumes and finds distance to next candidate surface.
- **Stepper:** Transports the track parameters and corresponding covariance matrix through magnetic field.
- **Actors/Aborters:** Extend propagation with various functionality (e.g. watch termination criteria).



# Build a Detector

**Build your detector, i.e.:**

- Build volumes/gap volumes from boundary surfaces
- Set the linking between the portals
- Define module surface factories to fill the volumes (using containers that mirror the underlying detector containers to keep everything sorted correctly).
- Insert the per-volume containers to an empty the detector.
- Example: `tutorials/src/cpu/do_it_yourself_detector.hpp`

**. . . or set up a predefined detector:**

- **Toy Detector:** Models the ACTS generic detector's pixel detector.
- **Telescope Detector:** Construct a number of surfaces at predefined positions or along a pilot track.
- **Wire Chamber:** Construct a number of layers that contain line surfaces.

```
> detray_generate_toy_detector --write_material --write_grids

> detray_generate_telescope_detector --write_material --length 550 --modules 10

> detray_generate_wire_chamber --write_material --write_grids --layers 11
```

# Detector IO

## [json] IO

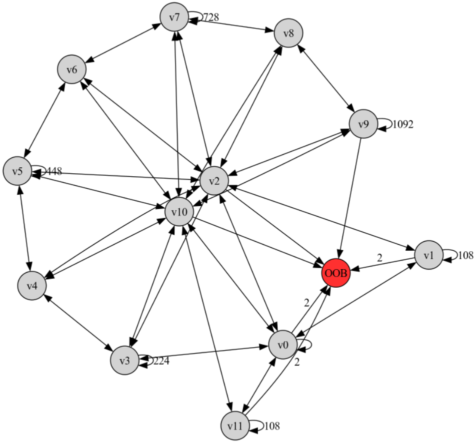
- Readers and writers to and from an intermediate "payload" description
- Readers and writers are held and registered in detector reader/writer on demand
- Get json files from ACTS to read in tracking geometries like ITk.
- Input json files can be checked for correct layout with python tool (`tests/validation/python/file_checker.py`)

```cpp
// Detector writer
auto writer_cfg = io::detector_writer_config{}.format(io::format::json);
io::write_detector(toy_det, toy_names, writer_cfg);

// Detector Reader
io::detector_reader_config reader_cfg{};
reader_cfg.add_file("toy_detector_geometry.json")
          .add_file("toy_detector_homogeneous_material.json")
          .add_file("toy_detector_surface_grids.json");

const auto [det, names] =
    io::read_detector<detector_t>(host_mr, reader_cfg);
```

# Display the Portal Linking as a Graph

Volumes are nodes, linked by their portal boundary surfaces (edges).Sensitive and passive surfaces are loops.

```
> detray_detector_display --geometry_file toy_detector_geometry.json
                          --write_volume_graph
> dot -Tpng plots/toy_detector_volume_graph.dot > toy_detector_volume_graph.png
```

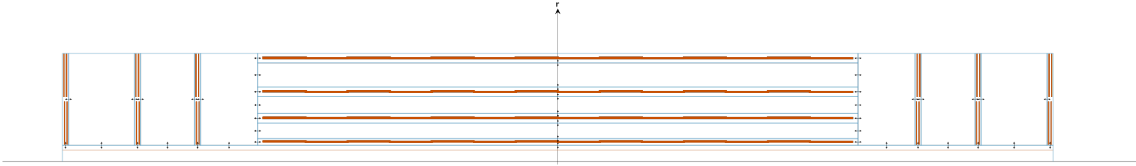


# Display the Geometry as an SVG

## Geometry SVG Visualization

- Different views (xy, rz)
- Can display the entire detector or single volumes/surfaces by index
- Can toggle portals and passive surfaces.

```
> detray_detector_display --geometry_file toy_detector_geometry.json
[--volume 7]
```

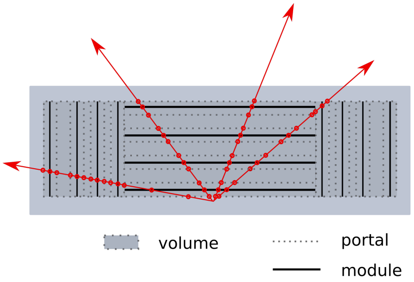


# Geometry Validation

```
> detray_detector_validation --gtest_filter=detray_validation.detector_consistency
          --geometry_file toy_detector_geometry.json
          --material_file toy_detector_homogeneous_material.json
          --grid_file toy_detector_surface_grids.json
```

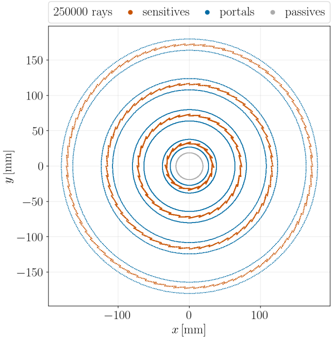
## Ray Scan

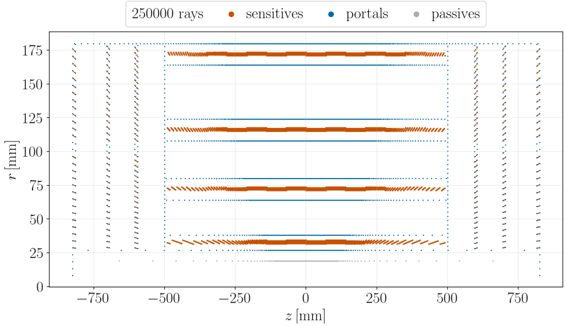
- Shoot straight line ray/helix through detector setup
- Record every intersection, together with associated volume index.
- Sort by distance and check for consistent crossing of adjacent portals.




```
> detray_detector_validation --gtest_filter=detray_validation.ray_scan_toy_detector
    --geometry_file toy_detector_geometry.json --write_scan_data

>  python3 /tests/validation/python/ray_scan_validation.py
--input ray_scan_toy_detector.csv [--hide_portals --hide_pssives]
```

# Geometry Validation

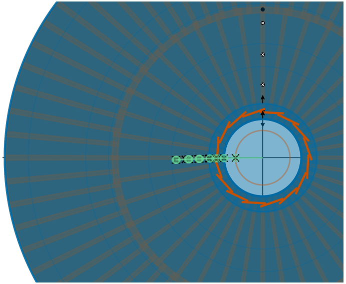


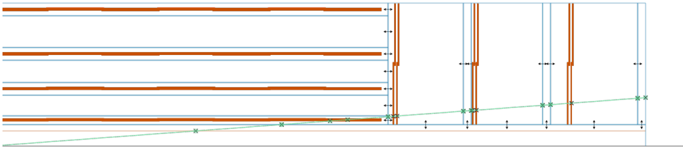



# Geometry Validation

## Navigation Validation

- Shoot ray/helix, but this time follow with navigator.
- Compare the entire intersection trace with the objects encountered by navigator.





# Material Validation

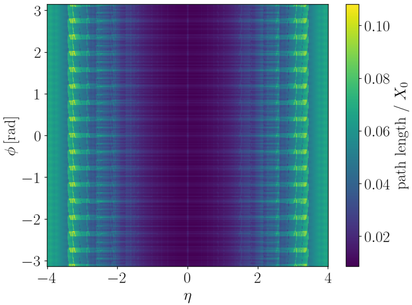
## Material Ray scan

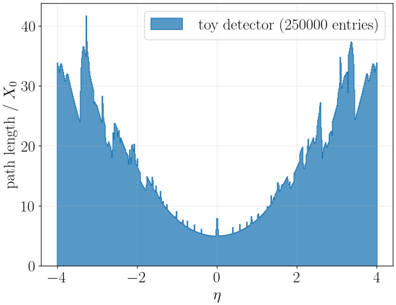
- Shoot rays through detector and record the material ($X_0$, $L_0$).
- Compare the collected material to ACTS.

---

```
> detray_material_validation --geometry_file toy_detector_geometry.json --material_file toy_dete
--phi_steps 500 --eta_steps 500

>  python3 /tests/validation/python/material_validation.py --input material_scan_toy_detector.cs
```

---



# Status and Outlook

**Status**

- Multiple testbed geometries available, inlcuding navigation
- Adaptive Runge-Kutta-Nyström algorithm for field integration
- Transport of track parametrization and covariance through (in-)homogeneous B-field
- Simple material description with material interactions

**Outlook**

- IO optimizations: Deduplication, sorting
- ACTS geometry import on-going (ODD is currently under validation)
- Material maps implementation using detray grids
- Benchmarking and profiling

# Backup

# Heterogeneous Computing Model

## Heterogeneous Computing Model

- Core classes templated on STL vs. `vecmem` containers.
- Memory allocation strategy is determined by *`vecmem` memory resources*.
- The data structures are built host-side and then passed to the kernel via *data views*

```
// Transform store using managed memory
vecmem::cuda::managed_memory_resource mng_mr;
// Build with host vector type
transform_store<vecmem::vector> store(mng_mr);
// Get store view object
auto sv = detray::get_data(store);

// Run the kernel
test_kernel<<<block_dim, thread_dim>>>(sv);
```

```
// Kernel-side construction
__global__ void test_kernel(store_view sv) {
   // Build with device vector type
   transform_store<vecmem::device_vector> store(sv);
   // Do something
}
```

# Define a Detector

**A detector type is defined by *metadata*:**

```
struct example_metadata {
    // Define links to types
    enum class mask_ids {
        e_square2 = 0,
        e_portal_rectangle2 = 1
    };
    enum class material_ids {
        e_slab = 0,
    };
    // Define data store types (including shapes)
    using bfield_backend_t = ...
    using transform_store = ...
    using mask_store = ...
    using material_store = ...
    ...
};

struct detector_registry {
    using default_detector = full_metadata<volume_stats, 1>;
    using toy_detector = toy_metadata<>;
    template <typename mask_shape_t>
    using telescope_detector = telescope_metadata<mask_shape_t>;
};
```

See: `detray/detectors/detector_metadata.hpp`

# Geometry Building

## json IO

- Readers and writers to and from an intermediate "payload" description
- Readers and writers are held and registered in detector reader/writer on demand
- In the writer this can be done from the detector type
- In the reader this has to be done by parsing the headers of the given files (yet to be added)

## Volume builders

- Volume builder class mimics detector containers to be able to build the linking correctly
- Data is added to the volume builders by surface factories, which can either be filled during IO or be surface "generators"
- After the volume builder is filled, the data is appended to the detector and the links are updated accordingly
- The basic volume builder can then be decorated with other volume builder dynamically (e.g. material builder, grid builder) files (yet to be added)

# Display the Portal Linking as a Graph

**Build and display the volume graph:**

```
// Build graph from detector
volume_graph graph(det);

std::cout << graph.to_string() << std::endl;

const auto &adj_mat = graph.adjacency_matrix();
// auto geo_checker = hash_tree(adj_mat);  Still WIP...
```

```
[...]
[>>] Node with index 1
 -> edges:
    -> 0
    -> 1                                        (108x)
    -> 2
    -> leaving world                            (2x)
[>>] Node with index 2
 -> edges:
    -> 0
    -> 1
    -> 3
    -> leaving world
[...]
```

# The `detray` Actor Model

## What is an actor in `detray`?

- Callable that performs a task after every step.
- Has a per track state, where results can be passed.
- Can be plugged in at compile time.
- In `detray`: Aborters are actors

```
// initialize the navigation
navigator.init(propagation);

// Run while there is a heartbeat
while (propagation.heartbeat) {

  // Take the step
  stepper.step(propagation);

  // And check the status
  navigator.update(propagation);

  // Run all registered actors
  run_actors(propagation.actor_states, propagation);

  // And check the status
  navigator.update(propagation);
}
```

## Implementation

- Actors can 'observe' other actors, i.e. additionally act on their subject's state.
- Observing actors can be observed by other actors and so forth (resolved at compile time!).
- Observer is being handed subject's state by actor chain
  $\Rightarrow$ no need to know subject's state type and fetch it.

# Define your own Actor

## What is an actor in detray?

- Inherit from detray::actor
- Implement an actor state, if needed
- Implement the call operator (overloads)

```
struct actor {
  /// Tag whether this is a composite
  struct is_comp_actor : public std::false_type {};
  /// Defines the actors state
  struct state {};
};
```

```
struct print_actor : detray::actor {
    struct state {
        ...
    };
    /// Actor implementation
    template <typename propagator_state_t>
    void operator()(state &printer_state, const propagator_state_t & /*p_state*/) const {
        // print something
    }

    /// Observing actor implementation
    template <typename subj_state_t, typename propagator_state_t>
    void operator()(state &printer_state, const subj_state_t &subject_state,
                    const propagator_state_t & /*p_state*/) const {
        // print something from the subject's state
    }
};
```

# Actor Chain Implementation

**Overview of actor implementation:**

```
/// Base class actor implementation
struct actor {

  /// Tag whether this is a composite
  struct is_comp_actor :
           public std::false_type {};

  /// Defines the actors state
  struct state {};
};
```

```
// Actor with observers
template <class actor_impl_t = actor,
          typename... observers>
class composite_actor final :
                           public actor_impl_t {
 struct is_comp_actor : public std::true_type{};
 // Implement this actor
 using actor_type = actor_impl_t;
 // Actor implementation + notify call
 void operator()(...) const { [...] notify(...);}

 private:
 // Call all observers
 void notify(...) const {...}
};
```

**Building a chain:**

```
// Define types
using observer_lvl1 = composite_actor<dtuple, print_actor, example_actor_t, observer_lvl2>;
using chain = composite_actor<dtuple, example_actor_t, observer_lvl1>;

// Aggregate actor states to be able to pass them through the chain
auto actor_states = std::tie(example_actor_t::state, print_actor::state);

// Run the chain
actor_chain<dtuple, chain> run_chain{};
run_chain(actor_states, prop_state);
```

# Full Chain

## Assemble a Propagation Flow

- Define B-Field (currently only homogeneous)
- Step-size constraints
- Navigation Policies: `stepper_default_policy`, `always_init`
- Additional inspectors run in actor chain

## Propagation type definitions

```
// Define navigator, stepper, actor chain and propagator
using navigator_t = navigator<detector_t>;
using b_field_t = detector_t::bfield_type;
using track_t = free_track_parameters<transform3>;
using constraints_t = constrained_step<>; // different step-size constr.
using policy_t = stepper_default_policy;  // how to update the navigation
using stepper_t = rk_stepper<b_field_t::view_t, transform3, constraints_t, policy_t>;
using actor_chain_t =
    actor_chain<dtuple, propagation::print_inspector, pathlimit_aborter>;
using propagator_t = propagator<stepper_t, navigator_t, actor_chain_t>;
```

# Full Chain

### Run track loop

```
constexpr scalar overstep_tol{-7. * unit<scalar>::um};
constexpr scalar step_constr{5 * unit<scalar>::cm};
constexpr scalar path_limit{60 * unit<scalar>::cm};

for (auto track :
     uniform_track_generator<track_t>(theta_steps, phi_steps, ori, mom)) {

    track.set_overstep_tolerance(overstep_tol);

    // Build actor states and tie them together
    propagation::print_inspector::state print_insp_state{};
    pathlimit_aborter::state pathlimit_aborter_state{path_limit};
    actor_chain_t::state actor_states = std::tie(
        print_insp_state, pathlimit_aborter_state);

    // Init propagator state
    propagator_t::state state(track, d.get_bfield(), d);

    // Set step constraints (the most strict will be applied)
    state._stepping
        .template set_constraint<step::constraint::e_accuracy>(step_constr);

    // Propagate the track
    is_success &= p.propagate(state, actor_states);
}
```