SPONSORED BY THE
Federal Ministry
of Education
and Research

EP-R&D
Programme on Technologies for Future Experiments

2022 ACTS Workshop

`detray` - tutorial

Joana Niermann

28.09.2022

Build the Project

Tutorial repository:

```
$ git clone https://github.com/beomki-yeo/detray_tutorial.git
$ cd detray_tutorial
$ mkdir build
$ cd build/
$ cmake ../ -DCMAKE_BUILD_TYPE=Release
$ make
```

Run executables (detray testing is also available):

```
# CUDA propagation
$ ./bin/detray_tutorial_propagator_cuda
```

The detray minimal build:

```
$ git clone https://github.com/acts-project/detray.git
$ cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo -S detray -B detray-build
$ cmake --build detray-build
```

Lots of configuration options:

- `DETRAY_CUSTOM_SCALARTYPE`
- algebra-plugins, e.g. `DETRAY_EIGEN_PLUGIN`
- `DETRAY_BUILD_CUDA`
- How to set up externals (e.g fetch them or pick up local installation)
- ...

Define a Detector

A detector type is defined by *metadata*:

```
struct telescope_metadata {

    // Define links to types
    enum mask_ids : unsigned int {
        e_rectangle2 = 0,
        e_unbounded_plane2 = 1,
    };
    enum material_ids : unsigned int {
        e_slab = 0,
    };

    using transform_store = static_transform_store<vector_t>;
    using material_definitions = tuple_vector_registry<material_ids, slab>;

    template <...>
    using surface_finder = surfaces_finder<...>;
};

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

See: `tests/common/include/tests/common/tools/detector_metadata.hpp`

Build a Detector

With the detector component types defined, build your detector, e.g.:

- 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 the detector's `add_objects_per_volume(...)` function.
- Example: `tests/common/include/tests/common/tools/create_toy_geometry.hpp`

. . . or set up a predefined detector:

- **Toy Detector:** Models the ACTS generic detector's pixel detector
- **Telescope Detector:** Construct a number of rectangular surfaces at predefined positions or along a pilot track. **Warning:** There are some pitfalls with this detector and it is not as well tested as the toy detector.

```
constexpr std::size_t n_brl_layers{4}; // up to 4 barrel layers
constexpr std::size_t n_edc_layers{3}; // up to 7 endacap layers
vecmem::host_memory_resource host_mr;

auto det = create_toy_geometry(host_mr, n_brl_layers, n_edc_layers);
```

Toy Detector - The TML Pixel Detector

Toy Detector

- Implement a small geometry, independent from io module
- All links are manually checked for consistency

The toy detector contains:

- A beampipe ($r = 27\,\mathrm{mm}$)
- An inner layer ($r_{\mathrm{min}} = 27\,\mathrm{mm}$, $r_{\mathrm{max}} = 38\,\mathrm{mm}$) with 224 pixel module surfaces
- A gap volume ($r_{\mathrm{min}} = 38\,\mathrm{mm}$, $r_{\mathrm{max}} = 64\,\mathrm{mm}$)
- An outer layer ($r_{\mathrm{min}} = 64\,\mathrm{mm}$, $r_{\mathrm{max}} = 80\,\mathrm{mm}$) with 448 pixel module surfaces
- Add grid will be added for local navigation.

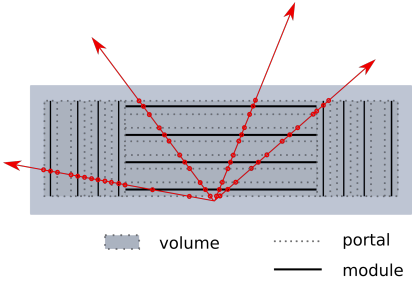
$\Rightarrow$ Provides a reliable, dynamically generated geometry that can be used for testing and rapid development.

$\Rightarrow$ Complexity (number of barrel and endcap layers) can be configured for easier debugging.

Geometry Validation

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.

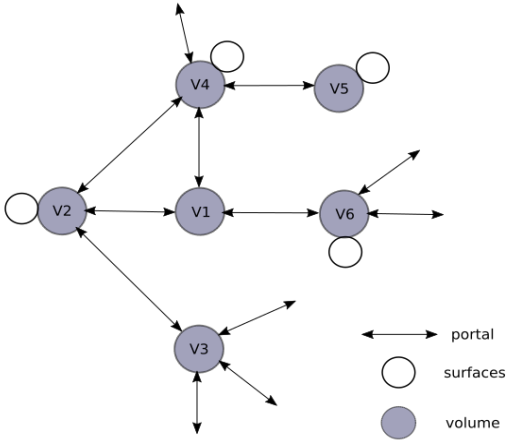



```
// Iterate through uniformly distributed momentum directions
for (const auto ray :
     uniform_track_generator<detail::ray>(theta_steps, phi_steps, ori)) {
    // Shoot ray through the detector and record all surfaces it encounters
    const auto intersection_record = particle_gun::shoot_particle(det, ray);  // :)

    // Create a trace of the volume indices that were encountered
    dindex start_index{0};
    auto [portal_trace, surface_trace] = trace_intersections(intersection_record, start_index);

    // Check correct portal linking
    is_consisten_linking &= check_connectivity(portal_trace);
}
```

Display the Portal Linking as a Graph

Geometry Linking Validation

- Compare ray scan with geometry linking graph.
- Provides a coarse, but automated check of geometric setup.



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
[...]
```

Geometry Validation

Navigation Validation

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

```
using inspector_t = aggregate_inspector<object_tracer_t, print_inspector>;
using navigator_t = navigator<decltype(det), inspector_t>;
[...]
for (auto track :
     uniform_track_generator<free_track_parameters>(theta_steps, phi_steps, ori, p_mag)) {

    // Get ground truth helix from track
    detail::helix helix(track, &B);

    // Record all surface-helix intersections
    const auto intersection_record = particle_gun::shoot_particle(det, helix);

    // Now follow that helix with the same track
    propagator_t::state propagation(track);
    // Get navigator object trace
    auto &inspector = propagation._navigation.inspector();
    auto &obj_tracer = inspector.template get<object_tracer_t>();
    auto &debug_printer = inspector.template get<print_inspector>();

    prop.propagate(propagation);
    std::cout << debug_printer.to_string();
}
```

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);
}
```

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.

$\Rightarrow$ Currently implemented: Navigation policies, pathlimit aborter, propagator inspectors.

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
    }
};
```

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<decltype(det)>;
using b_field_t = constant_magnetic_field<>;
using track_t = free_track_parameters;
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, track_t, 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_constants::um};
constexpr scalar step_constr{30 * unit_constants::cm};
constexpr scalar path_limit{60 * unit_constants::cm};

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

    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 p_state(track, actor_states);

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

    // Propagate the track
    is_success &= p.propagate(p_state);
}
```

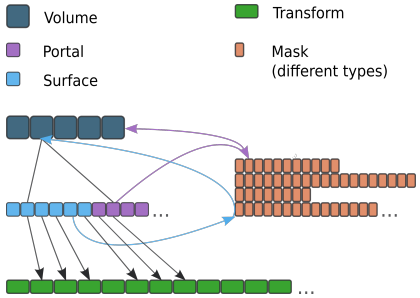
Backup

Detray Container Structure

In ACTS: Jagged memory layout of volumes containing layers (might be removed), containing surfaces.

Linking by Index

- Volumes keep index ranges into surface/portal containers.
- Surfaces/Portals keep indices into the transform and mask containers.
- Portals link to adjacent volume and next surfaces finder (local grid).
- Surfaces link back to mother volume.



⇒ Only transforms and masks contain geometric data, all other classes are uniquely used for container indexing.

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);
```