detray status

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

A. Salzburger (CERN)

Why ? What ?

4.2.1 Polymorphism

Virtual functions cannot be called inside a CUDA kernel unless the objects are constructed there. The Curiously Recurring Template Pattern (CRTP) is a C++ design pattern that emulates the behaviour of dynamic polymorphism through having a base class which is a template specialization for the derived class itself.

The ACTS Kalman filter is designed to be independent of the detector's tracking geometry, which could contain surfaces of different concrete types for different tracking detectors. To realize this design pattern on accelerators, the surfaces are implemented with CRTP, instantiated outside of the Kalman filter, and fed to the algorithm. CRTP is successfully used to define the surfaces as shown in the code sample in Listing 1.

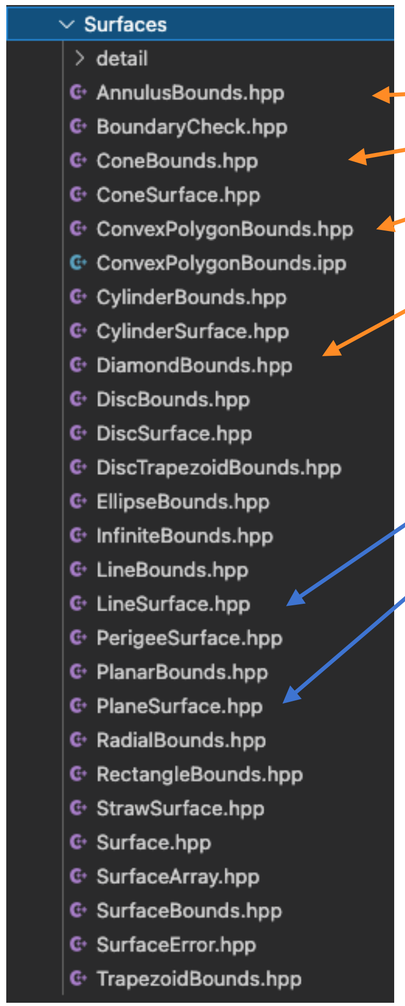
```
template <typename Derived>
inline const typename Derived::SurfaceBoundsType
    *Surface ::bounds() const {
  return static_cast<const Derived *>(this)->bounds();
}
```

Listing 1 Function definition in Surface base class using CRTP

X. Ai et al, "A GPU based Kalman Filter for Tracking"

https://arxiv.org/abs/2105.01796

ACTS Tracking Geometry



Polymorphism in Bounds

define different shapes on surfaces

Polymorphism in Surfaces

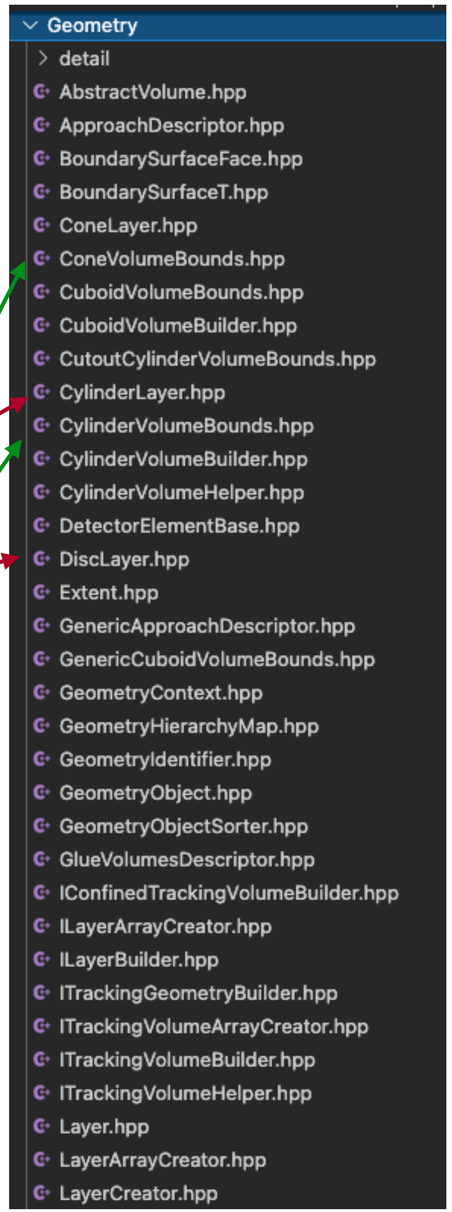
define different surface types
(coordinate systems)

Polymorphism in Layers

Layers extend surfaces

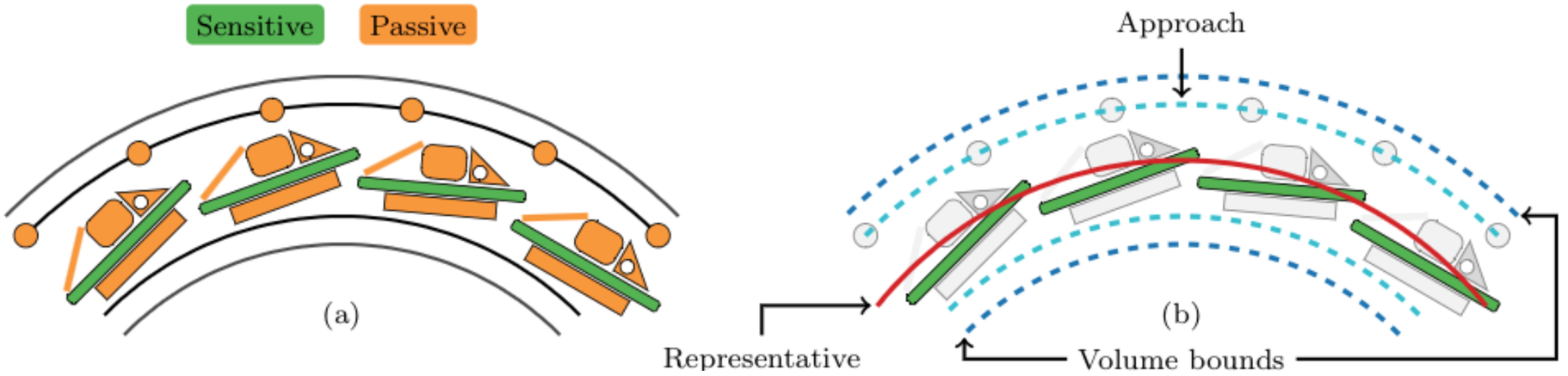
Polymorphism in VolumeBounds

Volumes have their shapes,
but are not typed otherwise



ACTS to detray

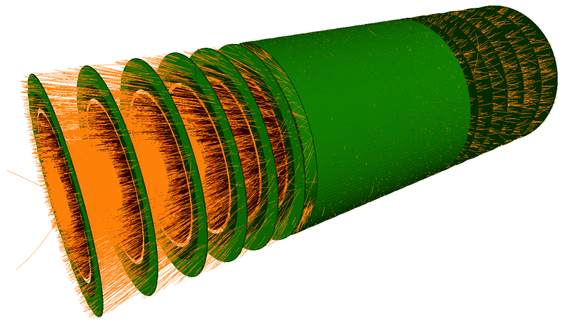
ACTS builds a very complex TrackingGeometry with interlinking (by bare pointers) the geometry objects for the navigation stream

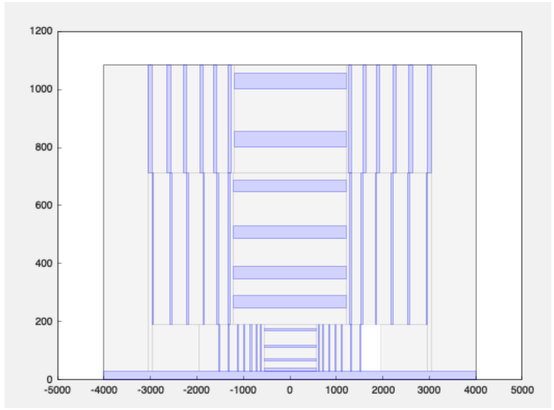


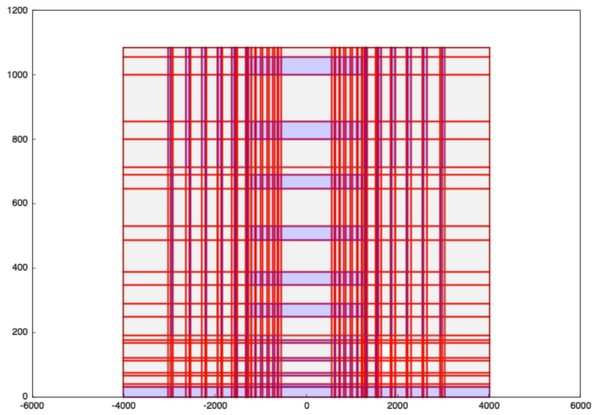
- once the geometry is built, all of those links could be done indexed based.
- That's what detray is trying to do.

ACTS to detray

ACTS builds a very complex TrackingGeometry with interlinking (by bare pointers) the geometry objects for the navigation stream



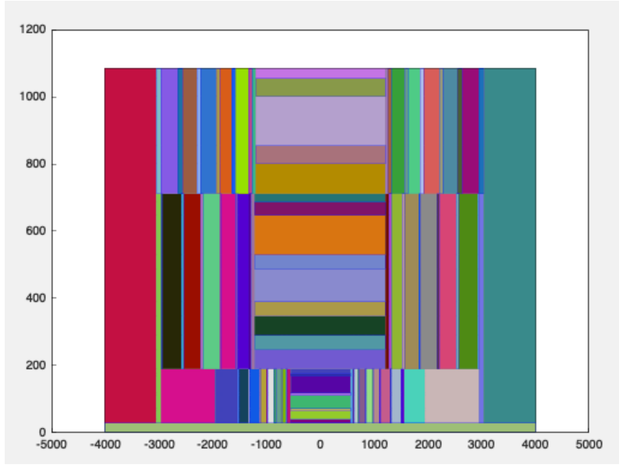




1) build detector in ACTS

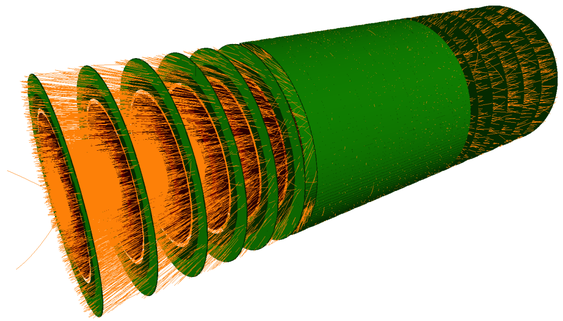
Currently done with csv

2) translate into detray



ACTS to detray

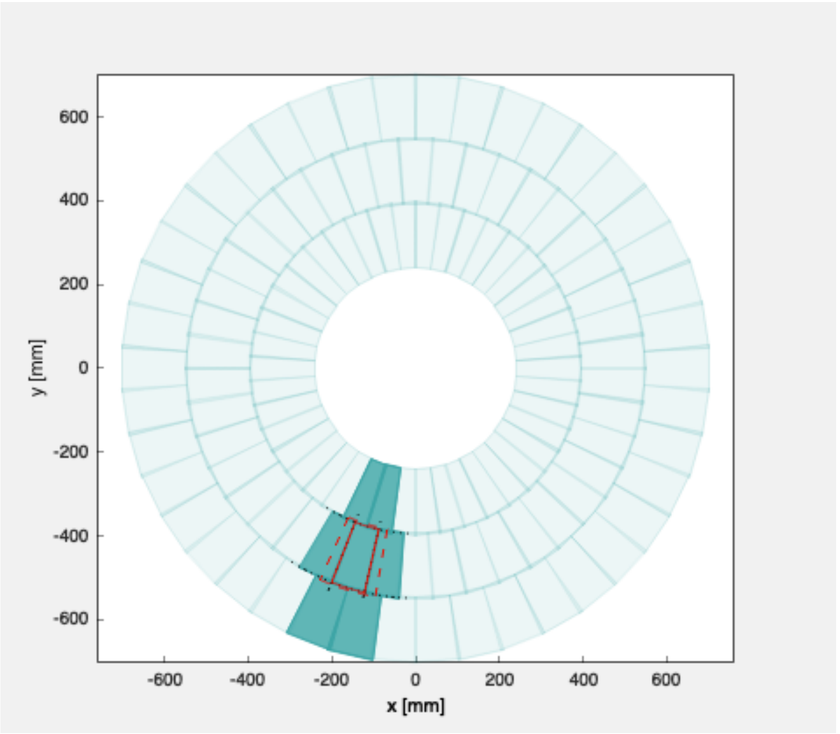
ACTS builds a very complex TrackingGeometry with interlinking (by bare pointers) the geometry objects for the navigation stream



1) build detector in ACTS

Currently done with csv

2) translate into detray



Local navigation resolved by index grid

detray::detector

```
/** Indexed detector definition.
 *
 * This class is a heavy templated detector definition class, that connects
 * surfaces, layers and volumes via an indexed system.
 *
 * @tparam array_type the type of the internal array, must have STL semantics
 * @tparam tuple_type the type of the internal tuple, must have STL semantics
 * @tparam vector_type the type of the internal array, must have STL semantics
 * @tparam alignable_store the type of the transform store
 * @tparam surface_source_link the type of the link to an external surface source
 * @tparam bounds_source_link the type of the link to an external bounds source
 * @tparam surfaces_populator_type the type of populator used to fill the surfaces grids
 * @tparam surfaces_serializer_type the type of the memory serializer for the surfaces grids
 *
 */
template <template <typename, unsigned int> class array_type = darray,
          template <typename...> class tuple_type = dtuple,
          template <typename> class vector_type = dvector,
          typename alignable_store = static_transform_store<vector_type>,
          typename surface_source_link = dindex,
          typename bounds_source_link = dindex,
          typename surfaces_populator_type = attach_populator<false, dindex, vector_type>,
          typename surfaces_serializer_type = serializer2>
class detector
```

sfge_writer
sge_writer

Array, tuple & vector type can be changed (e.g. using vecmem::*)

detray::detector

```
private:
    std::string _name = "unknown_detector";
    vector_type<volume> _volumes = {};

    vector_type<surfaces_finder> _surfaces_finders;

    volume_grid _volume_grid = volume_grid(std::move(axis::irregular{{}}), std::move(axis::irregular{{}}));
```

Contains a **vector_type** of volumes

Contains **surfaces/portals/bounds**

```
private:
    /** Volume section: name */
    std::string _name = "unknown";

    /** Volume index */
    dindex _index = dindex_invalid;

    /** Index into the surface finder container */
    dindex _surfaces_finder_entry = dindex_invalid;

    /** Bounds section, default for r, z, phi */
    array_type<scalar, 6> _bounds = {0.,
                                     std::numeric_limits<scalar>::max(),
                                     -std::numeric_limits<scalar>::max(),
                                     std::numeric_limits<scalar>::max(),
                                     -M_PI, M_PI};

    /** Surface section */
    constituents<surface, surface_mask_container> _surfaces;

    /** Portal section */
    constituents<portal, portal_mask_container> _portals;
};
```

detray::surface_base

For surfaces & portals

Link/index into transform container

Link/index into mask container*

Link/index to surface

Link to source, e.g. Acts::Surface*

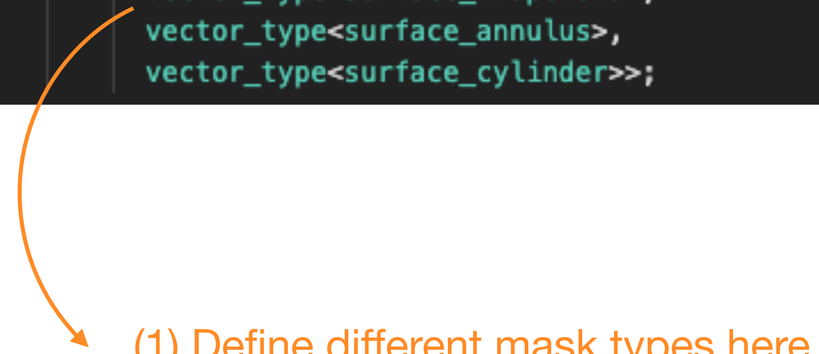
```
/** Templated surface base class for detector surfaces and portals
 *
 * @tparam transform_link the type of the transform/transform link for global 3D to local 3D frame
 * @tparam mask_link the type of the mask/mask link representation
 * @tparam volume_link the typ eof the volume/volume link representation
 * @tparam source_link the type of the source/source link representation
 */
template <typename transform_link, typename mask_link = dindex, typename volume_link = dindex, typename source_link = bool>
class surface_base
{
public:

    /** Constructor with full arguments - move semantics
     *
     * @param trf the transform for positioning and 3D local frame
     * @param msk the mask/mask link for this surface
     * @param vol the volume link for this surface
     * @param src the source object/source link this surface is representing
     *
     **/
    surface_base(transform_link &&trf, mask_link &&mask, volume_link &&vol, source_link &&src)
        : _trf(std::move(trf)), _mask(std::move(mask)), _vol(std::move(vol)), _src(std::move(src))
    {
    }
```

detray::surface_base

And how it deals with polymorphism

```
/// Surface components:
/// - surface links
using surface_links = array_type<dindex, 1>;
/// - masks, with mask identifiers 0,1,2
using surface_rectangle = rectangle2<planar_intersector, __plugin::cartesian2, surface_links, 0>;
using surface_trapezoid = trapezoid2<planar_intersector, __plugin::cartesian2, surface_links, 1>;
using surface_annulus = annulus2<planar_intersector, __plugin::cartesian2, surface_links, 2>;
using surface_cylinder = cylinder3<false, cylinder_intersector, __plugin::cylindrical2, surface_links, 3>;
/// - mask index: type, entry
using surface_mask_index = array_type<dindex, 2>;
using surface_mask_container = tuple_type<vector_type<surface_rectangle>,
                                          vector_type<surface_trapezoid>,
                                          vector_type<surface_annulus>,
                                          vector_type<surface_cylinder>>;
```



(1) Define different mask types here

detray::surface_base

And how it deals with polymorphism

(2) When resolving, use variadic unrolling

Premise:

a) as there are not many mask types, there's little to lose here

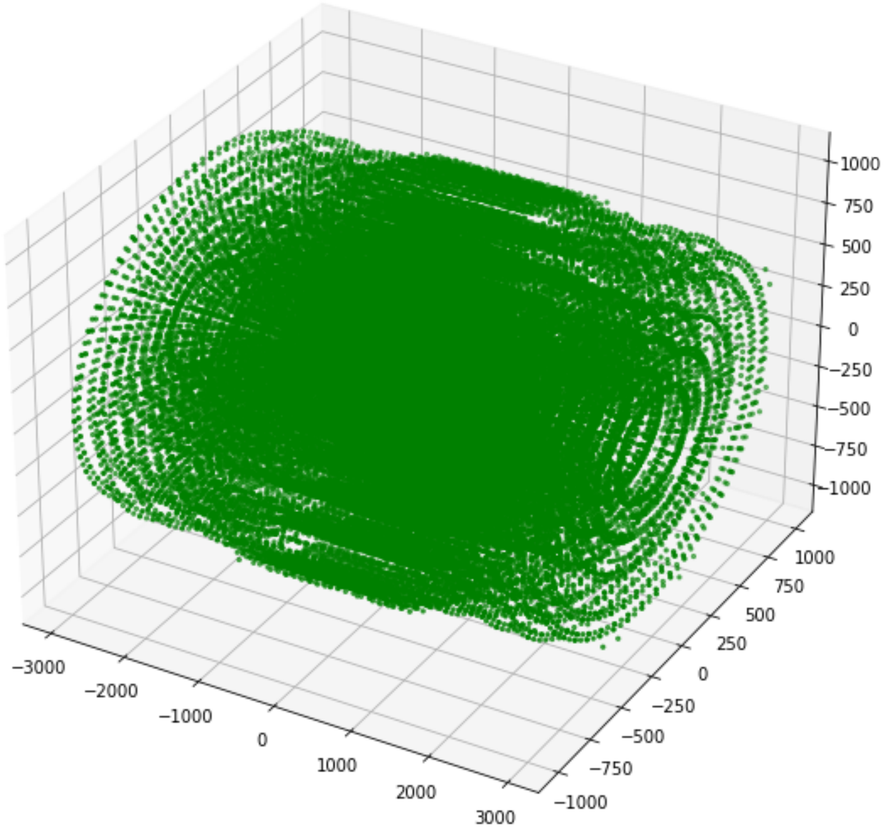
b) access to actual mask is again via index

```
template <typename intersection_type,
          typename links_type,
          typename track_type,
          typename mask_container,
          typename mask_range,
          dindex first_mask_context,
          dindex... remaining_mask_context>
bool unroll_intersect(intersection_type &intersection,
                      links_type &links,
                      const track_type &track,
                      const transform3 &ctf,
                      const mask_container &masks,
                      const mask_range &range,
                      dindex mask_context,
                      std::integer_sequence<dindex, first_mask_context, remaining_mask_context...> available_contices)
{
    // Pick the first one for interseciton
    if (mask_context == first_mask_context)
    {
        auto isg = intersect_by_group(track, ctf, std::get<first_mask_context>(masks), range);
        intersection = std::get<0>(isg);
        links = std::get<1>(isg);
        return true;
    }
    // The reduced integer sequence
    std::integer_sequence<dindex, remaining_mask_context...> remaining;
    // Unroll as long as you have at least 2 entries
    if constexpr (remaining.size() > 1)
    {
        if (unroll_intersect(intersection, links, track, ctf, masks, range, mask_context, remaining))
        {
            return true;
        }
    }
    // Last chance - intersect the last index if possible
    return last_intersect<intersection_type,
                          links_type,
                          track_type,
                          mask_container,
                          mask_range,
                          std::tuple_size_v<mask_container> - 1>(intersection, links, track, ctf, masks, range, mask_context);
}
```

sfge_writer Aa Abl .* No results

sge_writer AB

Potential & status




```
[detray] hits / missed / total = 272490 / 187977510 / 188250000
-------------------------------------------------------------
Benchmark                   Time             CPU   Iterations
-------------------------------------------------------------
BM_INTERSECT_ALL 1435555750 ns   1435182000 ns            1
salzburg@andimacbookprom1 detray %
```

So far, only trial and error example runs :-(

10k tracks, trying each surface ~1.5 s

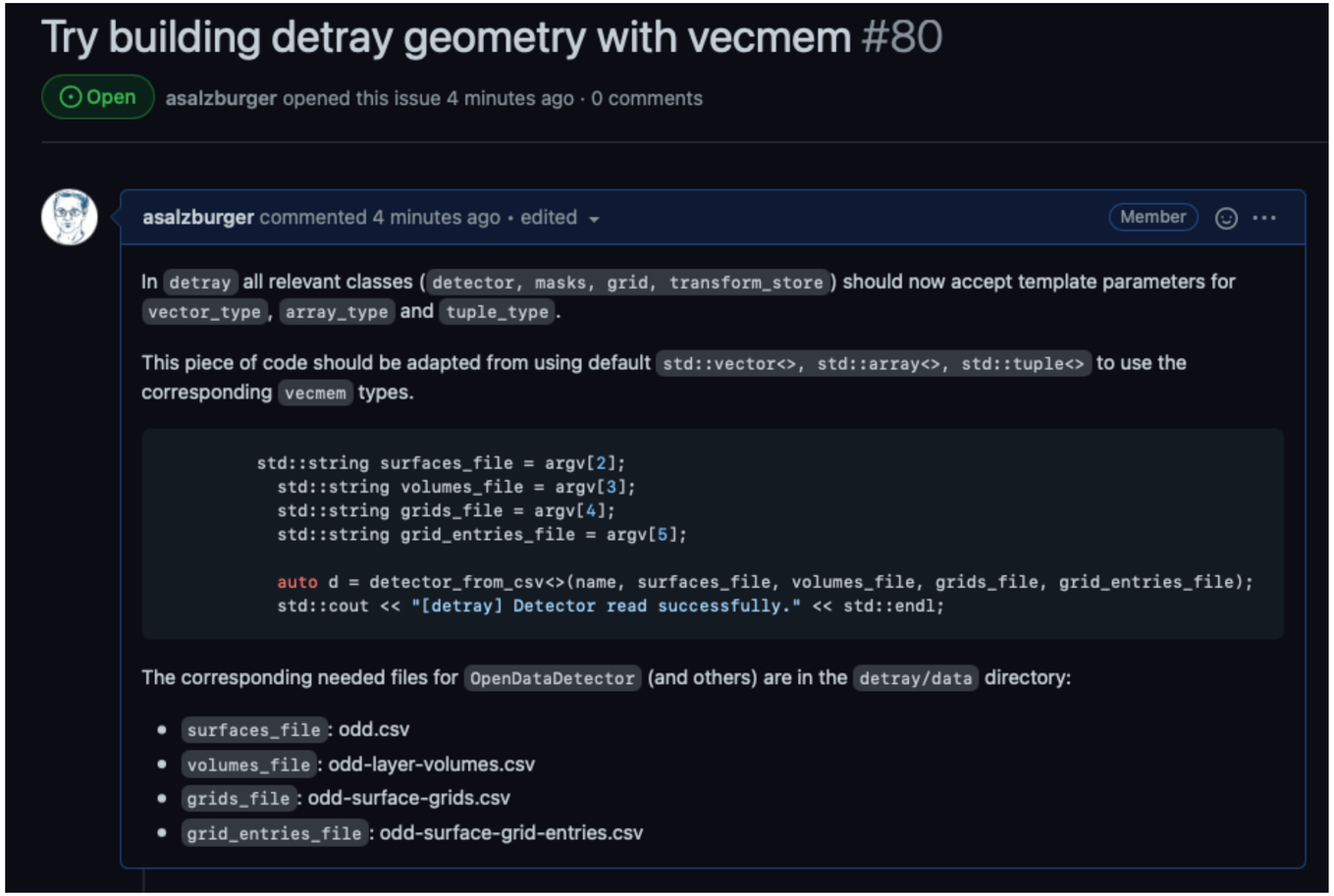
(1 intersection ~ 7 ns)

With navigation potential to save

~ (n_modules/n_candidates) … up to factor ~1000

Next steps

Build it with vecmem



Try building detray geometry with vecmem #80

Open asalzburger opened this issue 4 minutes ago · 0 comments

asalzburger commented 4 minutes ago • edited Member

In `detray` all relevant classes (`detector, masks, grid, transform_store`) should now accept template parameters for `vector_type`, `array_type` and `tuple_type`.

This piece of code should be adapted from using default `std::vector<>, std::array<>, std::tuple<>` to use the corresponding `vecmem` types.

```
std::string surfaces_file = argv[2];
  std::string volumes_file = argv[3];
  std::string grids_file = argv[4];
  std::string grid_entries_file = argv[5];

  auto d = detector_from_csv<>(name, surfaces_file, volumes_file, grids_file, grid_entries_file);
  std::cout << "[detray] Detector read successfully." << std::endl;
```

The corresponding needed files for `OpenDataDetector` (and others) are in the `detray/data` directory:

- `surfaces_file`: odd.csv
- `volumes_file`: odd-layer-volumes.csv
- `grids_file`: odd-surface-grids.csv
- `grid_entries_file`: odd-surface-grid-entries.csv