# Wire-Cell Toolkit Rectangles

Brett Viren

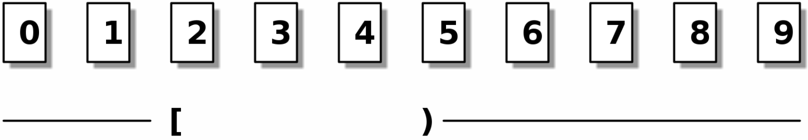
April 24, 2023

# Topics

- Interval trees, sets and maps
- Boost Implementation
- Extend to 2D: Rectangles
- Application to deghosting

# Intervals

Consider the 1D number line, either integer/discrete or real/continuous valued.




An *interval* is some finite, contiguous subset of the number line,

- Here, confine to the *right-open* inteval: $[2,5)$.
- 2 is in the interval, 5 is not (*ie*, just like Python/C++ iterator ranges)
- $[2,5)$ and $[5,7)$ are not overlapping

# Interval tree - a binary tree of intervals

For $n$ stored intervals (and $m$ returned), naive operations require at least $\mathcal{O}(n)$.
With a tree structure, expect:

| | |
|---|---|
| creation | $\mathcal{O}(n \log n)$ |
| insert/delete | $\mathcal{O}(\log n)$ |
| memory | $\mathcal{O}(n)$ |
| point query | $\mathcal{O}(\log n + m)$ |
| interval query | $\mathcal{O}(\log n)$ |

For brief description of the data structure and algorithms,
https://en.wikipedia.org/wiki/Interval_tree

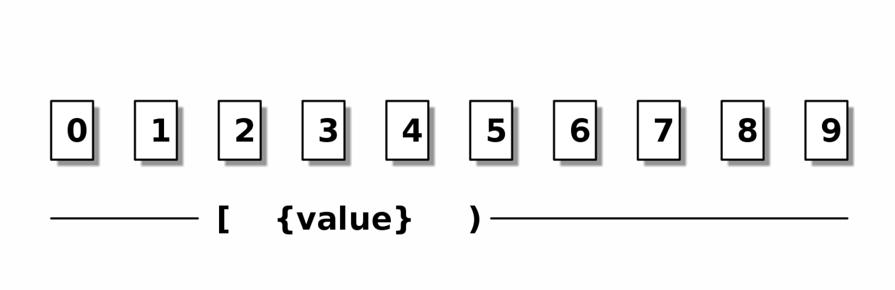
# Interval set

Adds *set-theoretic* operations: *union* (addition), *difference* (subtraction, symmetric or asymmetric) and *intersection*.

Set addition and interval overlaps:

- `s1 = {[1, 4)}`
- `s2 = {[2, 5)}`
- `s1 + s2 = {[1, 2), [2, 4), [4, 5)}`

# Interval maps - associate values with intervals.




An interval map holds an interval's value(s) in a **set**.

# Interval map: aggregation on overlap

In pseudocode:

```
interval_map m;

m.add([2,5), v1) -> { [2,5) -> {v1} }

m.add([1,4), v2) -> { [1, 2)->{v2},
                      [2, 4)->{v1,v2},
                      [4, 5)->{v1} }
```

# Boost Interval Container Library (boost::icl)

- https://www.boost.org/doc/libs/master/libs/icl
- Supports intervals, interval sets and interval maps.
- Provides family of free functions and operators.
    - creation, set operations, queries, iteration

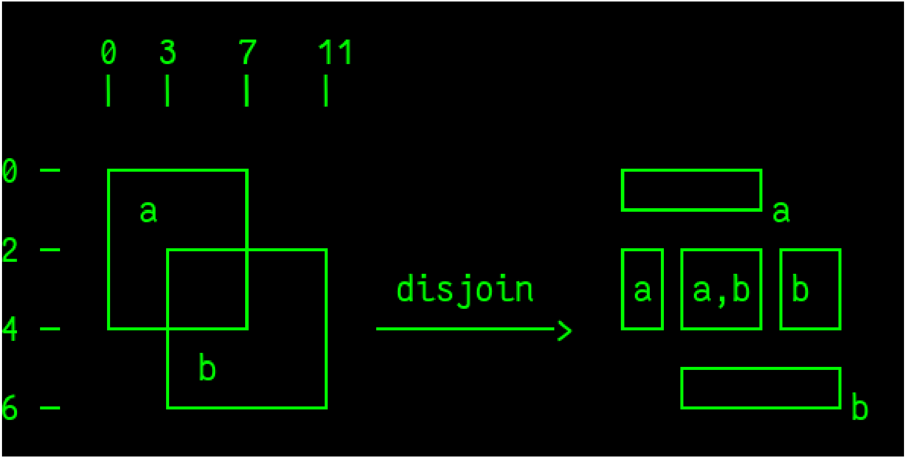
# boost::icl interval map example

```
using key_t = int;
using val_t = double;
using imap_t = boost::icl::interval_map<key_t, val_t>;
using interval_t = boost::icl::interval<key_t>::interval_type;

imap_t m;
m += std::make_pair(interval_t::right_open(0, 7), 42.0);

auto qi = interval_t::right_open(1, 2);
auto mq = m & qi; // "bitwise and"
for (const auto& [i, s] : mq) {
    cout << "in interval " << i << " we have set {";
    for (const auto& v : s) { cout << " " << v; }
    cout << " }\n";
}
```

# Extend interval map to 2D - `WireCell::Rectangles`




(thank `herbstluftwm` for the artwork)

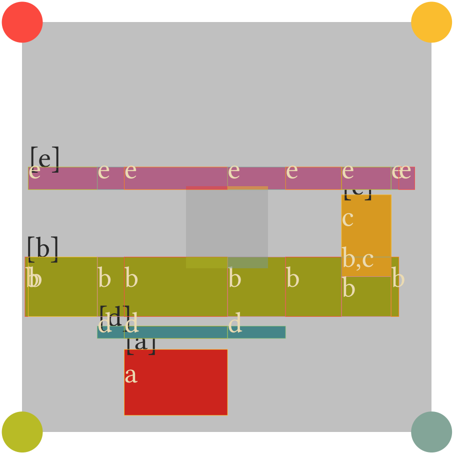
# Dimensional hierarchy of interval maps.

**Very** simple implementation.

```
using xkey_t = ...;
using ykey_t = ...;
using value_t = ...;
using set_t = std::set<value_t>;
// Intervals along the vertical Y-axis
using ymap_t = boost::icl::interval_map<ykey_t, set_t>;
// Intervals along the horizontal X-axis
using xmap_t = boost::icl::interval_map<xkey_t, ymap_t>;
```

Essentially, X dimension interval map maps to a Y dimension interval map which finally maps to a set of values.
Everything else in `Rectangles` merely provides some syntactic sugar.

# Example - random rectangles




- Initial rectangles: solid color holding black letter data, *eg*: `[a]`.
- 2D overlaps are out outlined rectangles holding a set of white letters.
- Central gray square is a query:

```
X<- [400,600) Y<- [400,600)
X-> [400,501.495) Y-> [400,408.36) {e}
X-> [400,501.495) Y-> [572.966,600) {b}
X-> [501.495,600) Y-> [400,408.36) {e}
X-> [501.495,600) Y-> [572.966,600) {b}
```

The dimensional hierarchy strategy to extend to 2D results in each X-interval projecting across **all** rectangles. This causes segmentation of rectangles which do not overlap in Y with the one providing the X-interval.

# Cluster Shadows

A **Geometric Cluster** is a *connected component* of a `b-b` graph

- `ICluster` already has this `b-b` subgraph embedded
- `b-b` edges formed between `b` nodes in neighboring slices which overlap.
- Todo: form these `b-b` edges given dead/bad channels

A **Blob Shadow** (see last presentation)

- Describes overlap of two blobs in one view.
- Can be wire-type or channel-type shadow.
- Results in a `b-b` **blob shadow graph**, edge is the shadow.

A **Cluster Shadow Graph** combines tese two `b-b` graphs

- Makes a `g-g` graph, each `g` is "geometric cluster"
- A `g` holds a `Rectangles` of the cluster for each view
- A `g-g` edge for any cluster pair with non-zero shadow

# Application: blob deghosting

A ghost blob truly has no charge and tends to have shadows with real blobs.
Use the `Rectangles` from Cluster Shadows to find them:

- Iterate over `b-b` edges of a CS graph.
- Get set-difference and/or set-intersection of the CS `Rectangles.`
- Compare area or charge*area or charge/area in diff/inter to total.
- Define selection criteria for ghosts.
  - compare against `BlobDepoFill` true charge blobs

This work is still a big TODO.