WCT Imaging Notes

Brett Viren

June 12, 2024

Contents

1 Concepts, data and processes **1**
- 1.1 Prototype . . . 1
- 1.2 Toolkit Interfaces . . . 3
- 1.3 Toolkit Data Interfaces . . . 4
- 1.4 Toolkit Component Interfaces . . . 5
- 1.5 Implementation notes . . . 6
  - 1.5.1 Slicing . . . 6
  - 1.5.2 Stripping . . . 6
  - 1.5.3 Tiling (ray clustering, layer overlaps) . . . 6
  - 1.5.4 Slice Overlaps . . . 8
  - 1.5.5 Solving . . . 8

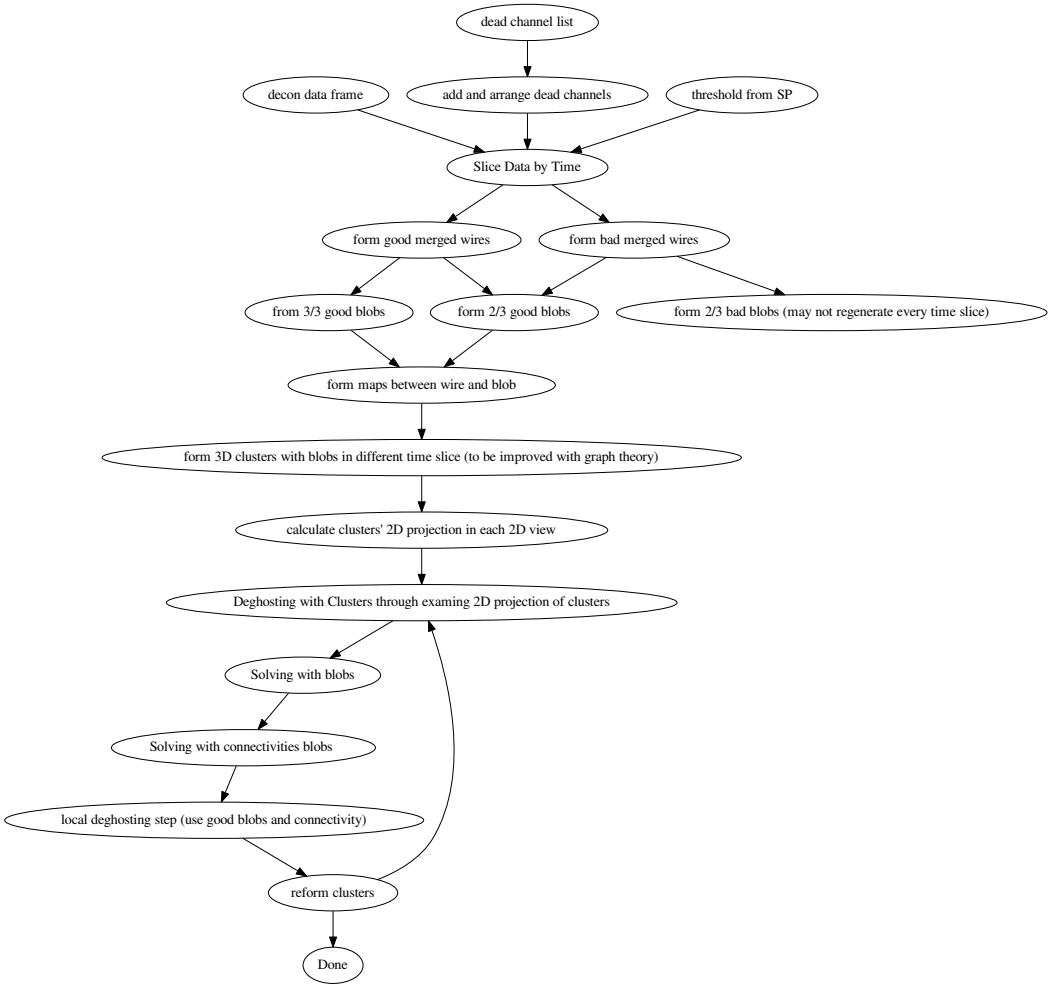
2 Tiling **9**

3 Live vs dead **10**

1 Concepts, data and processes

1.1 Prototype

The figure below holds an initial, high-level prototype conceptual flow graph showing the top-level algorithms as implemented in the Wire-Cell Prototype code base.



Some notes and interpretations on this are collected here. Please note, **these are WCP terms** and not fully carried over to WCT.

slice a Slice is filled by a ToyuBooNESliceDataSource. A slice represents a set of charge measurements (waveform samples) at a common time in a set of electronics channels. The "time" (`tbin`) of the slice is an integer, implicitly some bin index across the readout time. The duration of the slice is left fully implicit.

wire a GeomWire represents the logical address and physical location and extent of a segment of a conducting wire. It is conceptually equivalent to a WCT `IWire`.

merged-wire (colloquially, aka just "wire"), a MergeGeomWire represents a collection of `GeomWire` objects and an associated `tbin` of a time `Slice`. Conceptually it represents a 2D region of space. In the drift-transverse direction it is bounded by two extreme co-planar wires. In the drift-parallel by drift segment falling in the time slice.

cell (aka, "primitive-cell") a GeomCell represents a 2D region of space in the drift-transverse plane associated with a single wire from each wire plane. It has only transverse spacial information and neither time nor charge is associated.

blob (aka "cell" aka "merged-cell"), eg SlimMergeGeomCell is conceptually a collection of primitive-cells which are contiguous within wire boundaries or equivalently the spacial intersection of a collection of more than one merged-wire. Unlike a primitive-cell, a blob is associated with transverse space as well as time (drift position) and a measure of charge.

tiling eg LowmemTiling is a process which given a slice generates blobs that surround transverse regions that may have charge (ie, exclude those regions which are very likely to not have any charge).

parent-wire in the process of tiling, a blob may have smaller transverse extent than originally determined by a merged-wire. The original merged-wire is called a parent-wire to the final, smaller merged-wire.

1.2 Toolkit Interfaces

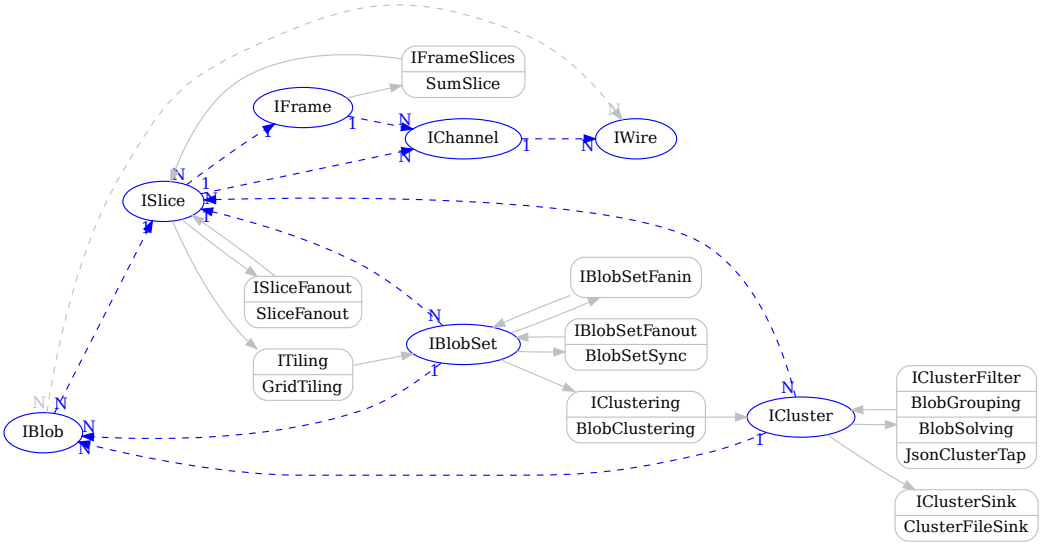
The `wire-cell-img` package provides components supplying the conceptual elements of the prototype described above. They are developed following this general strategy:

1. First, the interface base classes for the data products are made general to minimize their number. In particular, unique interfaces are not created to encode mere semantic differences (eg, "good" vs "bad" blob).

2. Second, the process component (node) interfaces are made specific and granular in order to deconstruct the algorithms into reasonably small units. In some cases it is expected that this deconstruction may run counter to optimization so not all processes interfaces must necessarily have concrete implementations.

3. Third, and applicable to both data and processing, the data interfaces intentionally lack functionality. They are meant to provide `const` input to and finalized output from components. They are not necessarily meant to be useful as "working objects". Processing components typically should convert their inputs into more convenient and optimized data representations for their internal processing and then pack results into the output data type.

The following diagram illustrates the small zoo of imaging related data and process node interface classes.



Note, this is **not** a DFP diagram but it hints at one. A DFP `INode` interface class is drawn as a box and an `IData` interface class as an ellipse. A solid arrow indicates input/output of a data instance to/from the node's `operator()` method. An arrow label of 1 indicates a scalar `pointer` and an N indicates a queue (explicit for output, implicit for input). A dashed arrow indicates that the implementation of the node interface will internally produce some number of instances of the linked data interface. Finally a gray arrow indicates a line of provenance dependency and the tail label indicates if this is provided by a scalar or vector of *back pointers*. Details on each member of this zoo follow.

1.3 Toolkit Data Interfaces

The WCP implementation was developed in the context of the MicroBooNE detector which has one wire (segment) per channel (wire and channels are degenerate) while a general detector (an in particular, ProtoDUNE/DUNE APAs) has multiple wires per channel due to wrapping a continuous wire conductor around a supporting frame. In WCT supports wrapped wires so this degeneracy must be broken. At the start of "tiling", one must necessarily begin with a "channel-centric" view and ultimately end with "segments-centric" one. This is reflected in the differentiation of "strip" and "blob".

In part, the WCP implementation works progressively, producing new data types as its algorithms run in a chain. However, there are obvious cycles in the prototype flow diagram above and it makes reference to steps where prior input data products must be accessed. Thus, all major data interfaces will provide **provenance** information in the form of *back pointers* to the data (interfaces) which produced them.

First, a number of data interfaces are already well established for WCT simulation, noise filtering and signal processing:

IWire largely equivalent to WCP's `GeomWire`. It defines a single physical wire (segment) and its logical connection to a channel.

IChannel defines a single detector electronics channel in terms of its logical addressing and association with an ordered list of the `IWire` which make up the attached conductor.

IFrame the activity (waveforms) from a finite span of time and detector channels. The input to imaging is expected to be the output of signal processing and thus the IFrame represents unipolar, positive signals however the frame may densely cover the is span (zero padded) or it may be sparse.

The first two interfaces above are "activity-independent" in the sense that they are typically static over a job and do not depend on any detector data. The `IFrame` is in a sense "activity-centric". The data interfaces in support of imaging described next tend to be a mix of both categories:

ISlice a slice describes the detector activity at the channel level over some span of time. It provides this information in the form of a map between `IChannel` and a value (typically "charge"). As a slice is expected to represent a portion of an `IFrame` it carries a back pointer to one from which it was derived.

IStrip a strip describes a subset of `IChannel` from one `IAnodePlane` and for each a corresponding value. See `ISliceStripper`. (n.b., maybe "stripe" is a better term since the `IWire` objects may collectively wrap around an APA.)

IStripSet simply a collection of `IStrip` with an `ident` number.

IBlob a blob describes a volume which is delineated along the drift direction by a time slice and transversely by one or more per-plane pitch-intervals. These pitch-intervals are defined by a pair of `IWire` objects which are considered to logically and inclusively-bound the interval. Physically, the boundary of the interval may be considered to extend further by one-half pitch to either side of the interval. The number of per-plane pitch-intervals need not be equal to the number of wire planes however, more than one pitch-interval for any one plane is not defined. For each IWire in each pitch-interval of a blob, there is an associated value (eg "the wire's charge"). The meaning of this value is application specific (ie, whatever the component wants which produces some `IBlob`).

IBlobSet simply a collection of `IBlob` with an `ident` number.

ICluster a cluster is collection of blobs represented as an undirected graph where edges represent "connectivity" between blobs which are in close proximity (by some clustering definition). FIXME: what else is needed, tbd.

1.4 Toolkit Component Interfaces

The WCP component (conceptual) interfaces is shown in the above figure in section 1.1. The current WCP implementation is also used to deconstruct algorithms into more granular parts and define their interfaces. Some existing interfaces are already applicable.

`IChannelSelector` produce an output frame containing traces from a an subset of channels of an input frame.

`IFrameFilter` generic frame modification.

The new interfaces:

`IFrameSlices` an `IQueuedoutNode` converting `IFrame` to a number of `ISlice` instances.

`ISliceStripper` an `IFunctionNode` converting an `ISlice` to an `IStripSet` holding `IStrip` instances. See `NaiveStripper`

`ITiler` an `IFunctionNode` which convert sets of strips into sets of blobs

`IBlobFilter` an `IFunctionNode` which produces an output set of blobs from an input set.

`IBlobClustering` produces an `ICluster` from a set of blobs

1.5 Implementation notes

This section holds notes on implementations.

1.5.1 Slicing

The `SumSlices` component produces a queued output of slices from an input frame. The frame is assumed to have any thresholds applied by setting any existing samples to exactly `0.0` or by providing a sparse frame.

Note that there is also a `SumSlicer` which produces an `ISliceFrame` collection of slices which otherwise works the same. However, it's monolithic coverage does not encourage future fine grained parallelism.

1.5.2 Stripping

The stripper is a function node. It takes in a single slice and produces a collection of strips. This 1-to-N mapping is done because it's a relatively fast operation, does not produce much data and all strips corresponding to a slice most likely always need to be used together and the overhead of passing individual strips through the execution graph just to recollect them is silly. If a reason to have individual strip flow is found, one may break this up in the future.

The `NaiveStripper` will produce strips in the simplest way possible. It does not provide any special support for dead channels or other detector pathology. However, it does handle arbitrary number of wire planes and treats generically wire planes which may or may not wrap. The algorithm is rather simple and based on forming a graph with two types of edges. First, it forms an edge from each `IChannel` in the input slice to each `IWire` that feeds that channel. It then walks all `IWire` in each wire plane in the anode in order of its wire-in-plane index and adds an edge from an `IWire` to its neighbor if both are associated with an input channel. A wire is considered a neighbor if it is within the configured "gap" parameter. By default, a wire must have a WIP index one higher to be considered a neighbor. Once this graph is constructed then all *connected subgraphs* are found. Each such subgraph spans a strip.

1.5.3 Tiling (ray clustering, layer overlaps)

Tiling is the process of determine regions ("blobs") which likely surround charge given wires which likely measured that charge as determined by their channels reading activity in the given time slice. Given a detector with wrapped wires it is not trivial to know on which wire current was induced and thus one must initially assume all wires attached to a "hit" channel may have had nearby charge. This then leads to "ghosts" which must be later removed. However a strong negative remains true: if a working channel measures no activity (above threshold) then all attached wires should not contribute to any "blobs".

Building "blobs" is then an exercise of identifying regions in the plane which have nearby active wires from all planes. Naive tiling algorithms can be rather expensive with worse case being N^{2+3n} where \$N is the number of wires per plane and n is the number of planes. However, substantial optimization is possible if the wire planes have certain uniform geometry. Specifically, if all wires in a given plane share a single angle and a single perpendicular distance between any two neighbors (the pitch) it is possible to construct geometric queries which require constant time to satisfy after a brief one time calculation of some coefficients.

To be general, we dispense with the notion of physical wires and instead discuss in terms of abstract rays which are line segments defined in terms of their 3D end points. Rays may be associated with wire segments, with the line running half way between two wires or with vertical/horizontal bounds of the active area of a wire plane. Conceptually, we will define two classes of coordinate systems in addition to a shared, global Cartesian system.

First, an ordered pair of parallel rays define an orthogonal coordinate system (ROCS) which may be translated and rotated about one axis of the global system. We define:

c^l the location of the center of the first ray in ROCS l, this first ray is indexed as $i = 0$.

p^l a vector which connects and is mutually orthogonal to both rays in ROCS l and pointing from the first to the second ray (which is considered to be indexed as $i = 1$.

The direction of p^l defines an axis for ROCS l and its magnitude defines a length scale. The second axis is take as in the mutual direction of the rays in ROCS l and no particular length scale is identified yet.

A second ROCS m is now considered which has rays at some non-zero angle with ROCS l. The set of crossing points of rays from each form a regular, non-orthogonal 2D grid. We define:

r_{ij}^{lm} a vector giving the location (expressed in global Cartesian coordinates) of the crossing point of the i^{th} ray in ROCS l and the j^{th} ray in ROCS m, $l \neq m$ and with one set of rays having a non-zero angle w.r.t. the other.

w^{lm} a relative vector giving the displacement between the intersections of a pair of neighboring rays of ROCS m with a ray in ROCS l. That is, this vector allows one to "hop" along a ray in l from one crossing point of a ray in m to the crossing point of its neighbor.

Without loss of generality one ray in each ROCS is given index $i = 0$ and it is then easy to calculate pair-wise crossing points for the "zero rays" of two ROCS or the zero crossing of l and m, r_{00}^{lm}. It is then trivial to define the crossing point of ray i from ROCS l and any ray j from ROCS m as $r_{ij}^{lm} = r_{00}^{lm} + jw^{lm} + iw^{ml}$.

In forming "blobs" it is typical that one must test if a particular point is "inside" some boundary defined by a pair of rays. This is equivalent to calculating the pitch location of the point in the ROCS coordinate system. If the only points which may be tested are themselves calculated as r_{ij}^{lm} then one evaluate the pitch location in a third ROCS in terms of the original crossings of two other ROCS, $P_{ij}^{lmn} = (r_{ij}^{lm} - c^n) \cdot \hat{p}^n$. Expanding, one arrives at, $P_{ij}^{lmn} = r_{00}^{lm} \cdot \hat{p}^n + jw^{lm} \cdot \hat{p}^n + iw^{ml} \cdot \hat{p}^n - c^n \cdot \hat{p}^n$. Note, that this decomposes into the a form of two three dimensional tensors $P_{ij}^{lmn} = ja^{lmn} + ia^{mln} + b^{lmn}$ where b^{lmn} is symmetric about the exchange of l and m. The size of each tensor dimension is that of the number of ROCS considered.

Finally, one may divide this tensor by the magnitude of the pitch of the third ROCS in order to find the index of the closest ray. This pitch index is $I_{ij}^{lmn} = P_{ij}^{lmn} / |p^n|$.

1.5.4 Slice Overlaps

Tiling leaves ghosts. One way to remove them is to check each cluster against those produced from a neighboring time slice and remove any that lack an "overlaps". To leverage the optimization techniques developed for tiling above, the definition of overlap should be expressed in the same terms of two-layer corners and their pitch distance in a third layer which can then be compared against pitch intervals in the layers of clusters in the "other" time slice.

In checking for overlap of clusters between two slices there is a danger of naively creating an $\mathcal{O}(N^2)$ operation, where N is the number of clusters in each slice. Even with the optimizations developed for tiling one can exploit the ordering of wires (rays) in a plane (layer).

fast overlap To perform this, a fast lookup is needed which goes from a pitch index range to the clusters which overlap it. This is similar to an Activity but which holds a reference to its cluster so that if an overlap in the first layer is found one may continue on to the other layers of the overlapping cluster.

mark and sweep In comparing a cluster in slice t to those in slice s we also want to learn which clusters in slice s where never found to overlap and to do so with having to do symmetric comparisons (ie, do NOT check each in t against each in s and then each in s with each in t). When any overlap is found some marking on both clusters is needed. After overlap comparisons, remove all that lack markings.

In a sense, this checking for overlap with neighboring time slice is **almost** a continuation of clustering by adding three new layers of activity and limiting comparison to three previous layers. However, it doesn't quite work as we must allow each slice to form its novel clusters before reducing them by the overlap check. One could imagine a variant on cluster-the-overlap which plays with the slice size. For example, some loss of spatial resolution is lost when the time slice is formed over some number of ticks. For example, instead of slicing at 4 ticks, one could make an extended tiling of 6 layers (ignoring vert/horiz layers) with each triple spanning 2 ticks.

1.5.5 Solving

The tiling step typically leaves "ghosts" which are blobs (aka ray clusters) which surround some region that does not actually contain any ionization activity. The ambiguities which are inherent in having a limited number of layers makes it impossible to remove these ghosts with just the tiling algorithm. Tiling only uses the existence of non-zero signal activity but does not use its magnitude. Using this information it is possible to construct a linear equation in terms of measured signal activity and constrained by the geometry. In the form of a matrix equation: $m = CGb$ where:

m a vector spanning the set of N_{ch} channels and holding their activity measurement

C an association matrix of $N_{cn} \times N_w$ which records which wires are attached to which channels.

G an association matrix of $N_w \times N_b$ which records which wires are overlapping to produce a given blob.

b a vector of length N_b giving the amount of ionization activity (in same units as m) of each blob.

In the "solving" step we wish to solve for b given the other tensors. At their largest, these tensors span spaces defined by all channels, all possible wires and all possible overlaps from a single wire strip from each plane and across all anode faces. Reducing the size of this problem the following strategies are available:

- all channels with activity below threshold are removed, in fact they and their channels are not even considered during tiling.
- all wires not contributing to a "blob" resulting from a tiling can be removed from the problem. If all wires associated with a channel are so removed so will the channel, even if it has activity above threshold.
- a set of wires contributing to one or more blobs may be grouped so that the sum of activity is considered.

For this last one, grouping must respect the mutual partial shadowing that will occur between blobs in a given layer. That is, if two blobs have intersection and difference then three wire groups (aggregate strips) should be formed with a pair of strips corresponding to one cluster, a pair to the other and one strip which is in both pairs.

It's also noted that the tiling is performed based on the arrangement of wire in planes of a single anode face. Thus G is composed of off-diagonal blocks consisting of zeros. The non-zero (but sparse) diagonal block matrices (G_0, G_1) each span the wires of one face and b is likewise separable as $b = (b_0, b_1)$ and their produce it block diagonal $Gb = (G_0 b_0, G_1 b_1)$. Application of C mixes the block when wire conductors wrap around two anode faces.

Next, it all zero blobs can be removed which removes wires and ultimately

2 Tiling

The `img/test/test-pdsp.jsonnet` file can configure to run either full or fast sim+sigproc (fast is the new `DepoSplat` component).

```
rm -f test-pdsp.npz test-pdsp-*.json && wire-cell -c img/test/test-pdsp.jsonnet
```

This makes a bunch of JSON files, one per slice. Any number of these can be can converted to a VTK unstructured grid file (`.vtu`) of blobs:

```
python util/test/blob2tvtk.py test-pdsp test-pdsp-???.json
```

It also makes a Numpy file which can be converted to a VTK point (`.vtp`) file of depos:

```
./util/test/depo2tvtk.py test-pdsp.npz - 3695*units.mm/speed
# --> makes test-pdsp-depos.vtp
```

As hinted by the argument here, there is currently an arbitrary time offset that needs to be applied to get depos and blobs to line up in the display.

You can individually load a `*.vt*` file into `paraview`. This can be tedious if one is in a debug loop and constantly reloading. That can be solved by making a Paraview data set file (`.pvd`) to include some number of files:

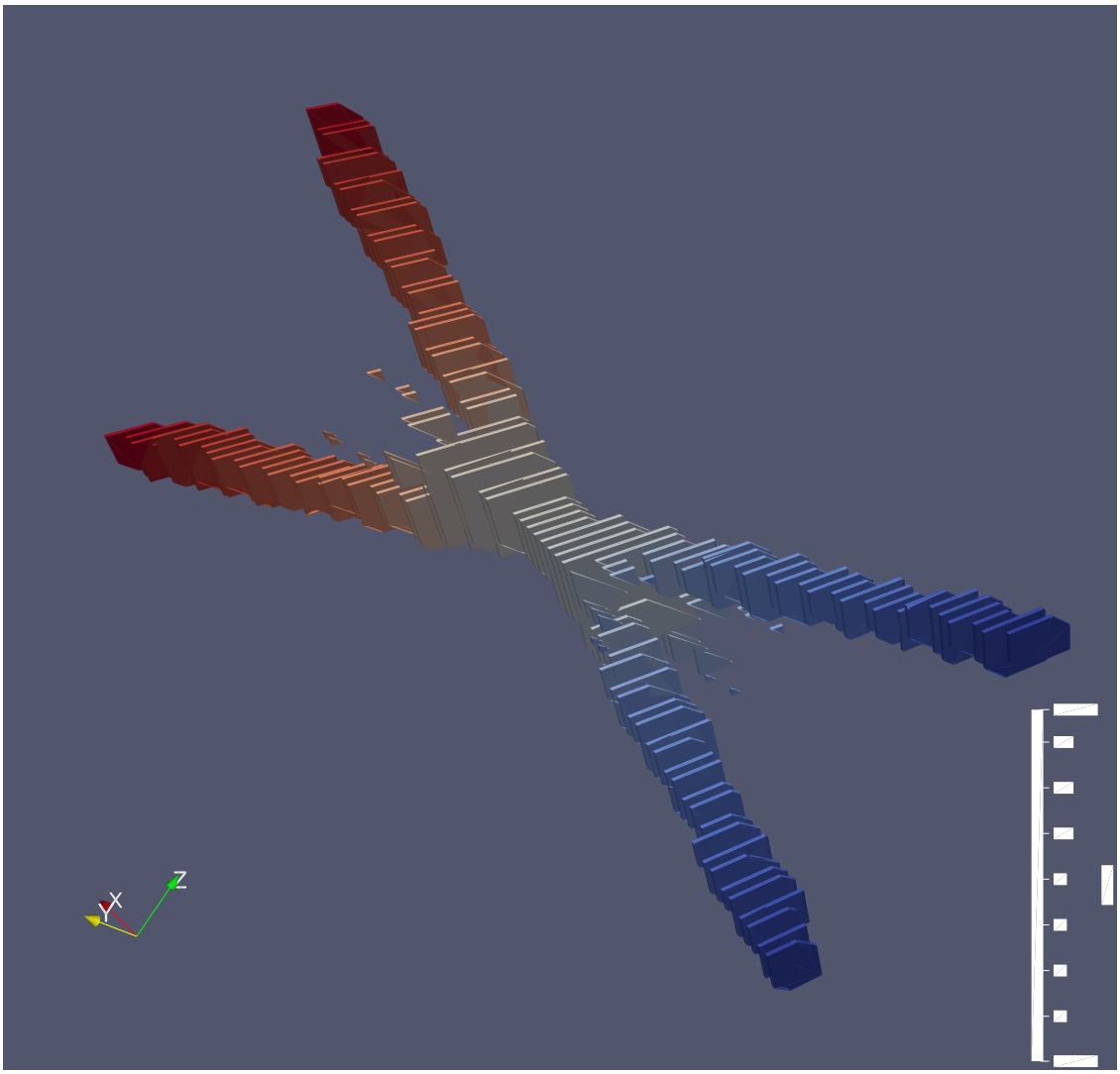
```
<VTKFile type="Collection" version="0.1" byte_order="LittleEndian">
  <Collection>
    <DataSet timestep="0001" part="001" file="test-pdsp-blobs.vtu" name="blobs"/>
    <DataSet timestep="0001" part="002" file="test-pdsp-points.vtp" name="points"/>
  </Collection>
</VTKFile>
```

Then, when para paraview opens, just click “Apply”.

Hit “s” and then select and in “Selection Display Inspector” you can label cells or points by various values such as “indices” which can then be matched to WC debugging logs.

A per-slice blobs can be viewed:

```
python util/test/blob2tvtk.py test-pdsp-748 test-pdsp-748.json
paraview test-pdsp-748-blobs.vtu
```



On Mate desktop, paraview renders tool tips with white text on yellow background which is unreadable. This solution works. Install `qtconfig`, “Appearance” tab, “Tune Palette...” button “ToolTipBase” and “ToolTipText” under “Central color roles”. Pick nicer color (I just made text black) and save.

3 Live vs dead

Channels are considered “live” if they are functioning properly or they are “dead”. A live channel may produce zero signal for some period of time. That is interpreted as measuring an absence of

activity. A dead channel may produce non-zero signal for some period of time. That is interpreted as inconclusive and ignored. A channel may change between live and dead state throughout a given frame.

There are many ways to determine "live" or "dead". Statically configured "bad channel lists" or dynamically determined regions of "badness" are two general ways. These judgments on the space of channel and tick are represented by a "bad" channel mask map provided in an `IFrame` *eg* as constructed by `ChargeErrorFrameEstimator`.

Given the tomographic nature of wire-readout it is not feasible to deal with "dead" channels with some kind of ficducial cut. Instead, the distinction of "live" vs "dead" channels are propagated through the WC imaging stage. This leads to seven, mutually-exclusive types of blobs:

- **3-view live blobs** formed from three views each providing live channels.
- **2-view live blobs** as above but one view consists of dead channels. There are three sub-types of 2-view live blobs, one for each plane supplying the dead channels.
- **2-view dead blobs** or simply **dead blobs**. The measures of two views consist of dead channels. Again, there are three sub-types where depending on which view provides the live view.

Both live and dead blobs are interpreted as "likely contains ionization signal" while any volume not contained by a blob is considered "likely does not contain ionization signal". 2-view live blobs are largely considered as somewhat more ambiguous but otherwise generally similar to 3-view live blobs. Dead blobs are used differently than live blobs. For example, later clustering may allow two otherwise disconnected live blobs to become connected if the path between them goes through a dead blob but not form a connection between those two live blobs if no dead blob intercedes.

The `GridTiling` component that performs tiling is ignorant of any of this live vs dead interpretation. It merely consumes the *activity map* of an `ISlice`. This maps a channel to a charge value and a charge uncertainty. The amount of charge is ignored in tiling and only the existence of the channel in this map and knowledge of the locations of the wires feeding each channel is considered in tiling.

An instance of the component `MaskSlices` is added as a predecessor to `GridTiling` in the DFP graph. `MaskSlices` applies a live/dead interpretation in the process of consuming an input `IFrame` and of preparing an output stream of `ISlice` instances and in particular generating their activity maps. A signal sample from a live channel that is within the span of slice being created is added to that channels entry in the activithy map. A signal value is taken from one named set of tagged traces and the signal uncertainty from another. Any channel that is determined to be "dead" in a slice has its entry in the activity map set to a special activity of 0 ± 10^{12}, by convention. This determination of "dead" may override an otherwise previously counted "live" channel.

A `MaskSlices` is further configured with three lists of plane indices:

`active_planes` the planes considered to generally be alive

`masked_planes` planes to ignore when checking for entries in the "bad" channel mask map.

`dummy_planes` planes for which all channels are set to the special activity value