Exploiting detector symmetries for photon visibility

Gianluca Petrillo

SLAC National Accelerator Laboratory, U.S.A.

LArSoft Coordination Meeting, April 9, 2019

Current photon visibility and shortcomings

`PhotonVisibilityService` interfaces LArSoft with a “photon library”, a 3D mapping describing the fraction of light received by *each* optical detector, emitted from any point of the argon volume. In a similar fashion, it may also serve timing information and more.

- currently implemented as a (seriously big) table
- takes loads of memory, lot of storage
- covers a single volume, and the representation in memory is contiguous (not sparse)
- ICARUS has two identical cryostats: photon library has to cover a box containing both, plus a lot of inactive material in between

We felt it.

A workaround

I have put together a workaround:

- introduced an intermediate *mapping* layer:
  - $\Rightarrow$ with the proper transformation, the library can cover only part of the detector volume, and the service will make up for the rest by symmetry
  - 1) transform detector space into the (voxelised) one of the library to access it
  - 2) transform optical detector numbering to decode the library content
- two additional steps required to get visibility of point $\vec{p}$ from optical detector C:
  1. *(new)* transform $\vec{p}$ using symmetry rules, to make it point to a place covered by the library ($\vec{p}'$)
  2. *(new)* transform the requested channel C into the correct channel C' in the library
  3. let the library convert $\vec{p}'$ into a voxel, and into the visibility for the requested channel
- `PhotonVisibilityService` internally applies both mappings (e.g. `GetVisibility()`)
- sometimes information for *all optical channels* is requested; in that case, now return a special class that lazily applies proper mapping when used (e.g. `GetAllVisibilities()`)

How it works for ICARUS

The optical detectors of ICARUS have:

- the two cryostats (*T300 modules*) are nominally identical and completely independent
- 180 PMT's per cryostat, 90 behind each of the two anodes (cathode is in the middle)
- the library: we map 180 channels from the full active volume inside one cryostat
- mappings:

spatial mapping

map the volume of second cryostat into the first one:

$\vec{p}$ on C:0 $\vec{p} \to \vec{p}$

$\vec{p}$ on C:1 $\vec{p} \to \vec{p} - \Delta\vec{p}_C$

with $\Delta\vec{p}_C$ the difference in position of the two cryostats

channel mapping

map the channels of the second cryostat to the first:

$\vec{p}$ on C:0 $\#0 \div 179 \to \#0 \div 179$,
$\#180 \div 359 \to 0.0$ (no visibility)

$\vec{p}$ on C:1 $\#0 \div 179 \to 0.0$ (no visibility),
$\#180 \div 359 \to \#0 \div 179$

How it could work for SBND

SBND!

- single cryostat, cathode in the middle, two identical TPCs at the sides
- the library: mapping sources from *one* of the TPCs into *all* the optical detector
- mappings:

spatial mapping		*channel mapping*	
map the volume of second TPC into the first		swap the channels of the two TPCs	
$\vec{p}$ on T:0	use position as is	$\vec{p}$ on T:0	use channels as they are
$\vec{p}$ on T:1	mirror the point into T:0	$\vec{p}$ on T:1	swap channel numbers between T:0 and T:1

- *ideally* one could simulate *one fourth of a single TPC* and use the symmetry to multiply it eightfold and cover everything
- note the “could” in the title: this is just a suggestion
  - $\rightarrow$ and of course it all depends on how symmetric the detector *actually* is

In principle, this is possible for ICARUS as well (on top of the mapping already described).

How it works for MicroBooNE

I am singling out MicroBooNE for no particular reason, but the point here is:

- there are no symmetries to be exploited
- an identity mapping can be used
- it's provided, it's called `PhotonMappingIdentityTransformations`, and it's the default
- so nothing to be seen, unless MicroBooNE wants to

More information

For future reference, the additional material in this presentation includes:

- hints on how to update existing code and configuration
- some questions and my answers (short Q&A)

How to implement your own mapping is *not* described here.
The best starting point is to look how it is done in ICARUS:
`icaruscode/Light/LibraryMappingTools`.

Summary

- I have coded tools to reduce the memory impact of the optical library
- test passed: compared with LArSoft `v08_14_00`, photons from `LArG4` are exactly the same (5 ICARUS events, 10 SBND events)
- code is in branches `feature/gp_PhotonVisTransformations`
  - → also provided for dunetpc (update) and icaruscode (custom mapping)
  - → nothing needed for ArgoNeuT, LArIAT, SBND and MicroBooNE
- requesting to merge it into LArSoft

Additional material

Updating the code (I)

This is a breaking change. I tried to make it as little breaking as possible:

- the return types of some `PhotonVisibilityService` methods has changed; I recommend to use the `auto` no-brainer:

```
  //get the visibility vector
  const float* PointVisibility = pvs.GetAllVisibilities(&xyz_segment[0]);
```

`larana/OpticalDetector/FlashHypothesisCreator.cxx` from LArSoft `v08_14_00`

becomes

```
  //get the visibility vector
  auto const& PointVisibility = pvs.GetAllVisibilities(&xyz_segment[0]);
```

- these return values behave as pointers (→ and better)
  - they refer to the data, they don't own it
  - they support indices: `float const v = PointVisibility[opDetID];`
  - they support testing: `if (PointVisibility) //...`
  - → they know how much data: `unsigned int const nChannels = PointVisibility.size();`
  - → they support iteration: `for (float v: PointVisibility) //...`

Updating the code (II)

- sometimes the actual type is needed:

```
TF1* ParPropTimeTF1;
float const* ReflT0s;
```

`larsim/LArG4/OpFastScintillation.hh` from LArSoft `v08_14_00`

becomes

```
#include "larsim/PhotonPropagation/PhotonVisibilityTypes.h" // phot::MappedT0s_t
// [...]
phot::MappedFunctions_t ParPropTimeTF1;
phot::MappedT0s_t ReflT0s;
```

Updating the code (III)

Configuration:

- specific mappings are implemented as *art* tools; e.g. for ICARUS:

```
PhotonVisibilityService: {
  # ...
  Mapping: {
    tool_type: ICARUSPhotonMappingTransformations
  }
}
```

- the tool `PhotonMappingIdentityTransformations` (provided) is a pass-through mapping
- the tool `PhotonMappingIdentityTransformations` is selected if no mapping is configured

This should make the old configuration work seamlessly.

Updating the code (IV)

More obscure breaking change is in `sim::PhotonVoxelDef`.
This is what was relevant for DUNE:

```
  void GetNeighboringVoxelIDs
    (const TVector3& v, std::vector<NeiInfo>& ret) const;
```

filling `ret` with 8 neighbouring voxels unless the point `v` is invalid, is replaced by:

```
   template <typename Point>
   std::optional<std::array<NeiInfo, 8U>> GetNeighboringVoxelIDs
     (Point const& v) const;
```

Beside templates[1], to avoid dynamic allocation, a “optional” STL array is now returned (`phot::PhotonVisibilityService::doGetVisibilityOfOpLib()` shows how to use it).

[1] Because it’s me after all.

Q&A (I)

Q: Is this the solution?
A: Diego Garcia-Gamez is working on a full parametrization of the visibility. That feels a better solution, if we can find such a parametrization. But that might be a much harder job for each experiment.

Q: ... and that will make the work you presented today obsolete?
A: More or less. It is still possible to use this mapping, but once we go analytic we can as well implement it directly in the analytic library rather than as an intermediate layer.

Q: Additional mapping... is it slower?
A: Yes. And if your module spends all its time querying the library, you might even notice. Me? I ran 5 ICARUS events through LArG4, taking each 6.72 s (`v08_14_00`) and 6.77 s (`feature/gp_PhotonVisTransformations`). I call it “within fluctuation”, and the slowdown “negligible”.

Q&A (II)

Q: To use this feature will we need a new photon library?

A: Probably; and the tools to make it are not necessarily readily available. The new library should map only the non-redundant space, only the relevant optical detectors, and an experiment could even consider to trade the gain of symmetry for a higher mapping granularity.

Q: Can DUNE far detectors take advantage of this?

A: This feature was not designed with DUNE in mind (see the first slide of this presentation for hints). Nevertheless, I believe that at least on first approximation, it may help.

The optical detector mapping will be more complicated when trying to correctly describe the TPCs at the border.

Q: Can multiple libraries be used? (e.g. a “corner” one and a ”bulk” one)

A: Nope. It’s possible to extend the system though — I haven’t put much thought into it.

Q&A (III)

Q: Where is the documentation?
A: These slides are some documentation already. The software infrastructure is documented in Doxygen, but that will only show how to use the new objects that `PhotonVisibilityService` returns.

Q: Where is the code?
A: There are numerous changes, all stored in `feature/gp_PhotonVisTransformations` branches, for:

- larcoreobj... oh, I introduced a geographical ID for optical detectors too, it's called `geo::OpDetID`!
- larcorealg for metaprogramming stuff (and assignation of `geo::OpDetID`)
- lardataalg (new mapping container objects)
- larsim (the mappings)
- larana (updates only)

Trying to sneak it into LArSoft...