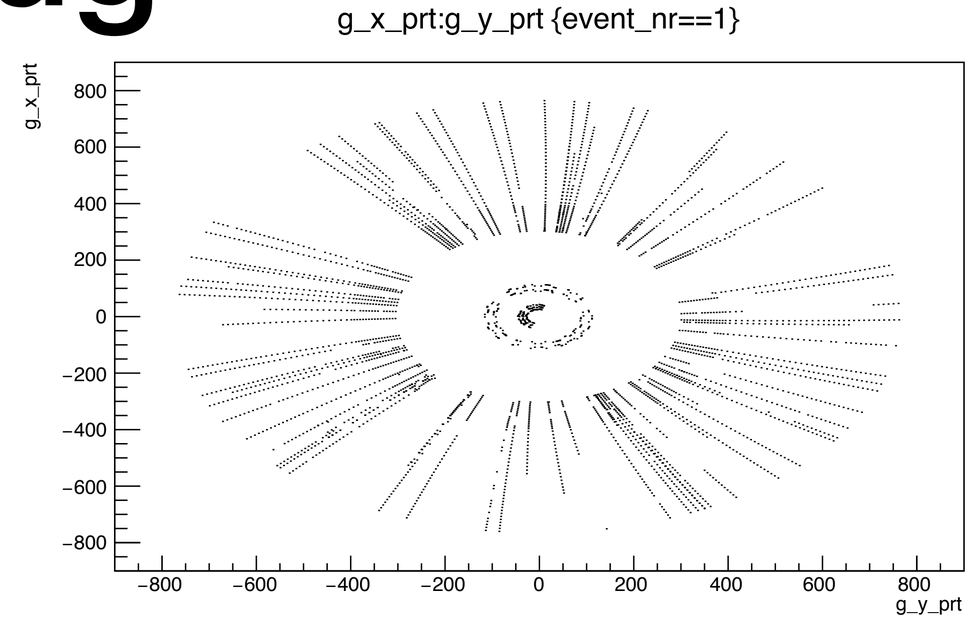
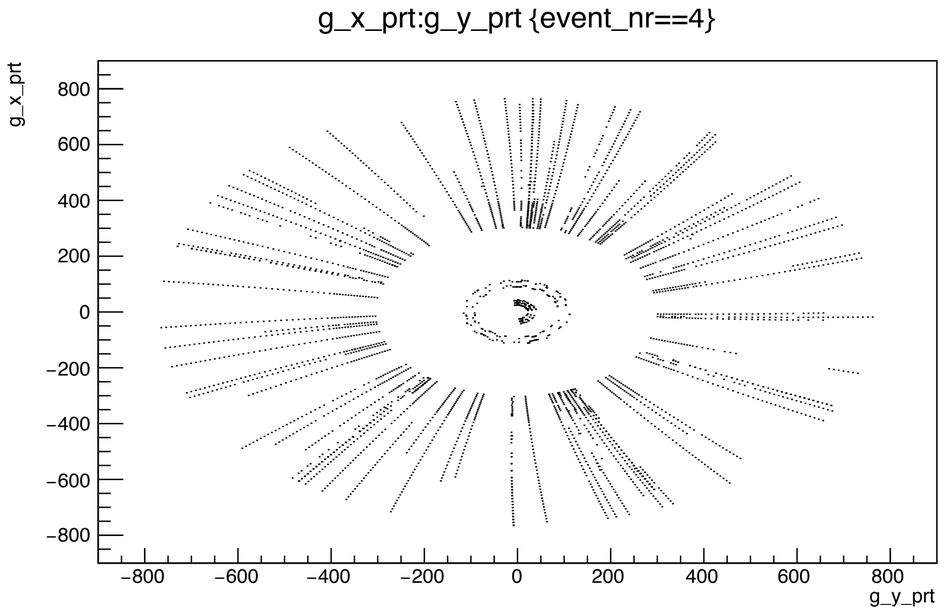
sPHENIX Navigation Bug (?)

Joe Osborn
Oak Ridge National Laboratory
14/7/2020

MVTX Bug

- We found a bug where, on an event-by-event basis, only half of the tracks are fit with MVTX hits (even though the hits are supplied to the fitter)
- It is always one 180° half of the detector, and it appears random which half it is
- We didn’t notice it until running more than 1 track per event, and looking event-by-event





Navigator

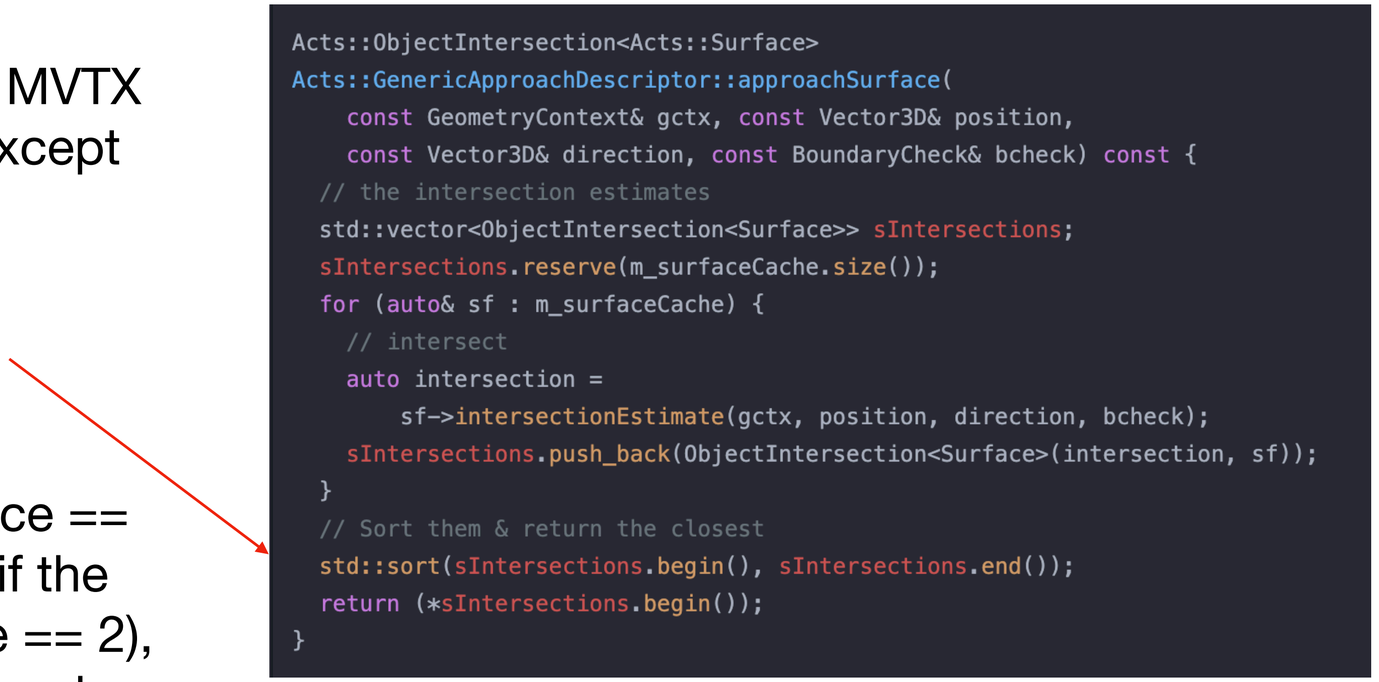
- Navigator searches for surface intersection in appropriate layer here
- approachSurface returns one of the two possible MVTX surfaces
- If it returns the “wrong” one, an invalid surfaceIntersection is returned and the path length to navigate is inf, so the navigator skips the MVTX

```
// Helper function to test intersection
auto checkIntersection =
    [&](SurfaceIntersection& sIntersection) -> SurfaceIntersection {
  // Avoid doing anything if that's a rotten apple already
  if (!sIntersection) {
    return sIntersection;
  }
  double cLimit = sIntersection.intersection.pathLength;
  // Check if you are within the limit
  bool withinLimit =
      (cLimit > oLimit and
       cLimit * cLimit <= pLimit * pLimit + s_onSurfaceTolerance);
  if (withinLimit) {
    // Set the right sign to the path length
    sIntersection.intersection.pathLength *=
        std::copysign(1., options.navDir);
    return sIntersection;
  } else if (sIntersection.alternative.status >=
             Intersection::Status::reachable) {
    // Test the alternative
    cLimit = sIntersection.alternative.pathLength;
    withinLimit = (cLimit > oLimit and
                   cLimit * cLimit <= pLimit * pLimit + s_onSurfaceTolerance);
    if (sIntersection.alternative and withinLimit) {
      // Set the right sign for the path length
      sIntersection.alternative.pathLength *=
          std::copysign(1., options.navDir);
      return SurfaceIntersection(sIntersection.alternative,
                                 sIntersection.object);
    }
  }
  // Return an invalid one
  return SurfaceIntersection();
};

// Approach descriptor present and resolving is necessary
if (m_approachDescriptor && (resolvePS || resolveMS)) {
  SurfaceIntersection aSurface = m_approachDescriptor->approachSurface(
      gctx, position, sDirection, options.boundaryCheck);
  return checkIntersection(aSurface);
}
```

Bug (?)

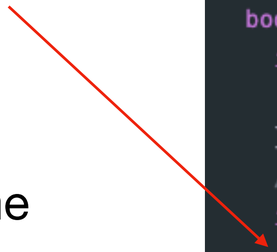
- Traced code and identified source of bug here
- The intersectionEstimate returns two MVTX surfaces, each with identical geoID except approach surface identifier
- std::sort sometimes selects the first surface, sometimes the second
- If the first is selected (approach surface == 1), navigation visits MVTX surfaces - if the second is selected (approach surface == 2), an invalid SurfaceIntersection is returned and an infinite path length step is returned to the navigator, and it skips the MVTX



```
Acts::ObjectIntersection<Acts::Surface>
Acts::GenericApproachDescriptor::approachSurface(
    const GeometryContext& gctx, const Vector3D& position,
    const Vector3D& direction, const BoundaryCheck& bcheck) const {
  // the intersection estimates
  std::vector<ObjectIntersection<Surface>> sIntersections;
  sIntersections.reserve(m_surfaceCache.size());
  for (auto& sf : m_surfaceCache) {
    // intersect
    auto intersection =
        sf->intersectionEstimate(gctx, position, direction, bcheck);
    sIntersections.push_back(ObjectIntersection<Surface>(intersection, sf));
  }
  // Sort them & return the closest
  std::sort(sIntersections.begin(), sIntersections.end());
  return (*sIntersections.begin());
}
```

Bug (?)

- The sorting of the intersections is where the error comes in
- When sorting the two possible intersections, when the pathLengths are negative it goes to the incorrect surface and then skips the MVTX
- When the pathLengths are positive it proceeds as intended
- So the bug must be associated to, for some reason, sometimes the intersection path lengths are negative, and this must be somehow associated to a 180 deg azimuthal region (?). Is there some geometry convention that is mismatched?



```
/// Default constructor
Intersection() = default;

/// Bool() operator for validity checking
explicit operator bool() const { return (status != Status::missed); }

/// Smaller operator for sorting,
/// - it respects the validity of the intersection
/// @param si is the intersection for testing
bool operator<(const Intersection& si) const {
  if (status == Status::unreachable) {
    return false;
  }
  // Now check the pathLength
  if (si.status != Status::unreachable) {
    return (pathLength < si.pathLength);
  }
  // The current one wins, no re-ordering
  return true;
}
```