A Handy Tool for History Keeping of Geant4 Tracks – J4HistoryKeeper –

Sumie Yamamoto[a], *Keisuke Fujii*[b1], and *Akiya Miyamoto*[b],

[a] School of High Energy Accelerator Science, The Graduate University for Advanced Studies (Sokendai), Tsukuba, 305-0801, Japan
[b] High Energy Accelerator Research Organization (KEK), Tsukuba, 305-0801, Japan

Abstract

The Particle Flow Analysis (PFA) is currently under intense studies as the most promising way to achieve precision jet energy measurements required at the future linear e^+e^- collider. In order to optimize detector configurations and to tune up the PFA it is crucial to identify factors that limit the PFA performance and clarify the fundamental limits on the jet energy resolution that remain even with the perfect PFA and an infinitely granular calorimeter. This necessitates a tool to connect each calorimeter hit in particle showers to its parent charged track, if any, and eventually all the way back to its corresponding primary particle, while identifying possible interactions and decays along the way. In order to realize this with a realistic memory space, we have developed a set of C++ classes that facilitates history keeping of particle tracks within the framework of Geant4. This software tool, hereafter called J4HistoryKeeper, comes in handy in particular when one needs to stop this history keeping for memory space economy at multiple geometrical boundaries beyond which a particle shower is expected to start. In this paper this software tool is described and applied to a generic detector model to demonstrate its functionality.

Keywords: Geant4, History Keeping, PFA
PACS code: 07.05.Tp, 02.70.Uu

[1] Corresponding author.
E-Mail address: keisuke.fujii@kek.jp
TEL: +8-29-864-5373
FAX: +8-29-864-2580

1 Introduction

The experiments at the International Linear Collider[1] will open up a novel possibility to reconstruct all the final states in terms of fundamental particles (leptons, quarks, and gauge bosons) as viewing their underlying Feynman diagrams. This involves identification of heavy unstable particles such as W, Z, t, and even yet undiscovered new particles such as H through jet invariant-mass measurements. The goal is thus to achieve an jet invariant-mass resolution comparable to the natural width of W or Z[2]. High resolution jet energy measurements will thus be crucial, necessitating high resolution tracking and calorimetry as well as an algorithm to make full use of available information from them. With a currently envisaged tracking system[1] that aims at a momentum resolution of $\sigma_{p_T}/p_T = 5\times10^{-5}p_T[\mathrm{GeV/c}]$ or better, tracker information will be much more accurate than that from calorimetry for charged particles. This implies that the best attainable jet energy resolution should be achieved when we use the tracker information for charged particles and the calorimeter information only for neutral particles. This requires separation of calorimeter clusters due to individual particles and, in the case of charged particle clusters, their one-to-one matching to the corresponding tracks detected in the tracking system. This is the so-called Particle Flow Analysis (PFA) currently under intense studies[3].

For the PFA, it is hence desirable to have a highly granular calorimeter that allows separation of clusters due to a densely packed jet of particles. In practice the performance of the PFA depends not only on the hardware design of the detector system consisting of the tracker and the calorimeter but also on a particular algorithm one employs to separate calorimeter signals due to neutral particles from those due to charged particles. Various realistic algorithms are currently being tested by various groups[3].

For the optimization of detector configurations and the PFA algorithm, it is crucial to identify factors that limit the PFA performance and clarify the fundamental limits on the jet energy resolution that remain even with an infinitely granular calorimeter and an ideal algorithm to achieve perfect track-to-cluster matching. We hence need a tool to connect each calorimeter hit in particle showers to its parent charged track, if any, and eventually all the way back to its corresponding primary particle, while identifying possible interactions and decays that might have taken place along the way. In order to achieve this history keeping with a reasonable memory size, we need an algorithm to effectively achieve infinite calorimeter segmentation independently of the physical size of its readout cells as well as a mechanism to stop history keeping at various geometrical boundaries beyond which particle showering is expected.

We have developed a set of C++ classes that realize such a functionality within the framework of Geant4. We call this software tool J4HistoryKeeper hereafter, since it is the name of the central class of the package. Although J4HistoryKeeper was designed primarily for PFA studies, it has wider applications. It comes in handy in particular when one needs to stop history keeping for memory space economy at multiple user-registered geometrical boundaries. The software tool was implemented in a Geant4[6] based Monte Carlo simulator called JUPITER[4, 5] and has been used successfully for PFA studies, together with a smearing and reconstruction package called SATELLITES[5], running under a modular analysis frame work called JSF[7], both of which are based on ROOT[8]. The source code of a demo package of J4HistoryKeeper with slimmed up versions of JUPITER and SATELLITES is available from our "J4HistoryKeeper Sample Code Page"[9].

In the following sections, we first elucidate the concept of *Cheated PFA*, which can be considered as the primary application that J4HistoryKeeper is designed for. We then describe the software tool to keep history of particle tracks (Geant4 tracks) traced through a detector in Geant4 with emphasis put on its design philosophy. The subsequent section is devoted to its usage with a sample application to a generic ILC detector model to demonstrate its functionality.

2 Concept of Cheated PFA

In principle Monte Carlo simulations allow us to use so-called Monte-Carlo truth and enable us to unambiguously separate calorimeter hits due to different incident particles, thereby performing perfect clustering. By linking so-formed calorimeter clusters to corresponding charged particle tracks in the tracking system again using Monte-Carlo truth, we can achieve the situation with the perfect PFA. We call this Cheated PFA (CPFA) since it involves cheating by using Monte-Carlo truth, which is impossible in practice. The concept of the CPFA is detailed in this section so as to clarify the philosophy behind the design of the software tool.

2.a Perfect Clustering and Perfect Track-Cluster Matching

For the CPFA, the history of Geant4 tracks should be kept on a track-by-track basis starting from a primary track at the interaction point. The history of all the secondary tracks together with the original one should be recorded until they hit any one of pre-registered boundaries beyond which particles may start showering. At such a boundary we create a virtual hit called `PHit`. Calorimeter hits by Geant4 tracks in a particle shower will then be tagged with this `PHit`. By collecting all the calorimeter hits with the same `PHit` we can hence form a calorimeter cluster without any confusion (see Fig.1).

Since the `PHit` carries the information of its parent track, one-to-one matching between the calorimeter cluster and its corresponding charged particle track in the tracking system is possible. Once matched, we just lock the calorimeter cluster as linked to a charged track and just use the tracker information. Calorimeter clusters with no matching charged tracks are hereafter called neutral PFOs, while all the charged tracks are called charged PFOs regardless of whether there are corresponding calorimeter clusters or not.

It is also important to record the mother-daughter correspondence for particles decayed in a tracking volume so as to estimate their effects on the PFA performance. The mother-daughter correspondence is book-kept together with the other information on the daughter track such as its particle ID, position, and momentum, in a so-called `BreakPoint` object which is created at the beginning of each track. The information stored in the `BreakPoint` objects will be used to follow particle decays observed as kinks or V^0s in the tracking volume and to assign correct particle masses to charged PFOs. This bookkeeping comprises the major role of the history keeper.

2.b Infinite Calorimeter Segmentation

For a realistic calorimeter design, the granularity of the calorimeter, or equivalently the cell size, is finite and hence the signals created by shower particles stemming from different parent

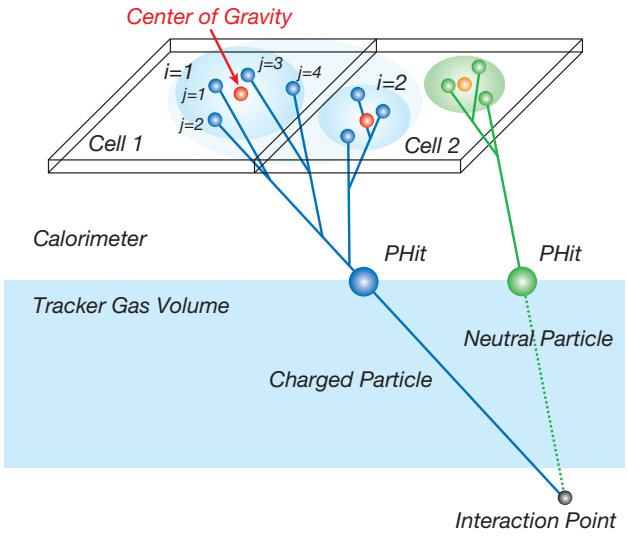

Figure 1: *Schematics showing the cheated PFA concept. Only two cells in a single sampling layer of the calorimeter are shown to simplify the picture though in practice much more cells are expected to be hit over different sampling layers.*

particles sometimes merge into a single hit degrading the cluster separation capability. In order to investigate to what extent this limits the PFA performance, we need to know the performance expected for perfect cluster separation. It is, however, impracticable to implement infinitely fine segmentation even in a Monte Carlo detector simulator because of memory space limitation. In order to overcome this drawback, we exploit the following trick.

In each hit cell, say cell i, we separately store the energy sum of hits originating from the same `PHit`:

$$E_i^c \;=\; \sum_j E_{i,j}$$

and their center of gravity:

$$\mathbf{x}_i^c \;=\; \sum_j E_{i,j}\,\mathbf{x}_{i,j} \,/\, E_i^c,$$

instead of using the cell center as the hit position. In the above expression $E_{i,j}$ and $\mathbf{x}_{i,j}$ are the energy deposit and the position of j-th hit in cell i with the same `PHit`. Denoting the total energy of the cluster originating from the same `PHit` by

$$E^c \;=\; \sum_i E_i^c,$$

we can then calculate its cluster center as

$$\mathbf{x}^c \;=\; \sum_i E_i^c\,\mathbf{x}_i^c \,/\, E^c \;=\; \sum_i\sum_j E_{i,j}\,\mathbf{x}_{i,j} \,/\, \sum_i\sum_j E_{i,j},$$

showing that the center of gravity calculated this way precisely coincides the one would-be obtained when the segmentation is infinitely fine. It should be also emphasized here that hits from different `PHits` make multiple centers of gravity in the same cell, which can be later separated even though they are in the same cell, thereby realizing the infinite segmentation in effect.

3 Tool Design

Before coding our tool for history keeping, we set the following guideline to fulfill the required functionality discussed in the last section: a) there must be a versatile mechanism to register user-defined physical volumes whose specified boundaries can be used to define a `PHit` that marks the source point of a particle shower, b) whether a track is allowed to create a `PHit` or not depends on whether the track originates from any pre-created `PHit` or not, which should be checked on a track-by-track basis at the beginning of its tracking, c) the history keeping is to be done on the track-by-track basis by creating a `BreakPoint` object at the beginning of each track if there is no `PHit` from which the track stems, and d) the history keeping should be realized making maximum use of existing Geant4 facilities within the framework of JUPITER, e) JUPITER should produce Monte-Carlo truths (i.e. exact hits) and their smearing should be done later in SATELLITES as needed.

The following is a sketch of the tool design we adopted according to the guideline:

1. The history keeping is to be done on the track-by-track basis using `J4TrackingAction` that inherits from `G4UserTrackingAction`. Its `PreUserTrackingAction` method is hence called at the beginning of a new track. The `PreUserTracingAction` method serially invokes `PreTrackDoIt` method of each offspring of `J4VSubTrackingAction` pre-registered to the `J4TrackingAction` object. Likewise, its `Clear` method serially invokes `Clear` method of individual offsprings of `J4VSubTrackingAction`.

2. `J4VSubTrackingAction` is an abstract class that serves as a base class for user-defined sub-actions taken by `J4TrackingAction` thereby extending the `G4UserTrackingAction` functionality. It has a method called `Clear` to reset the object state.

3. `J4HistoryKeeper` is implemented as a derived class from `J4VSubTrackingAction` and, in its `PreTrackDoIt` method, scans through a collection of pre-registered `J4PHitKeeper` objects corresponding to a collection of bounding surfaces. It then creates a `J4BreakPoint` object if none of them has been hit by any ancestors of the new track,

4. `J4PHitKeeper` also inherits from `J4VSubTrackingAction`. Its `PreTrackDoIT` method checks if this new track is stemming from any pre-created `PHit`, and, if not, resets its sate to allow creation of a new `PHit`. When its corresponding boundary is hit by the current track, a `PHit` object is created, if it is allowed, to tag subsequent daughter tracks possibly created in a shower.

The flow of tracking related to J4HistoryKeeper is shown in Fig. 2.

3.a Extension of G4UserTrackingAction

The `G4UserTrackingAction` class provides one with a handy tool to perform a user-defined action on a track-by-track basis. In its original form, however, it allows only a single action. In order to extend its functionality to accept multiple user-defined actions, we have introduced the concept of `SubTrackingAction` as sketched above.

`J4VSubTrackingAction`

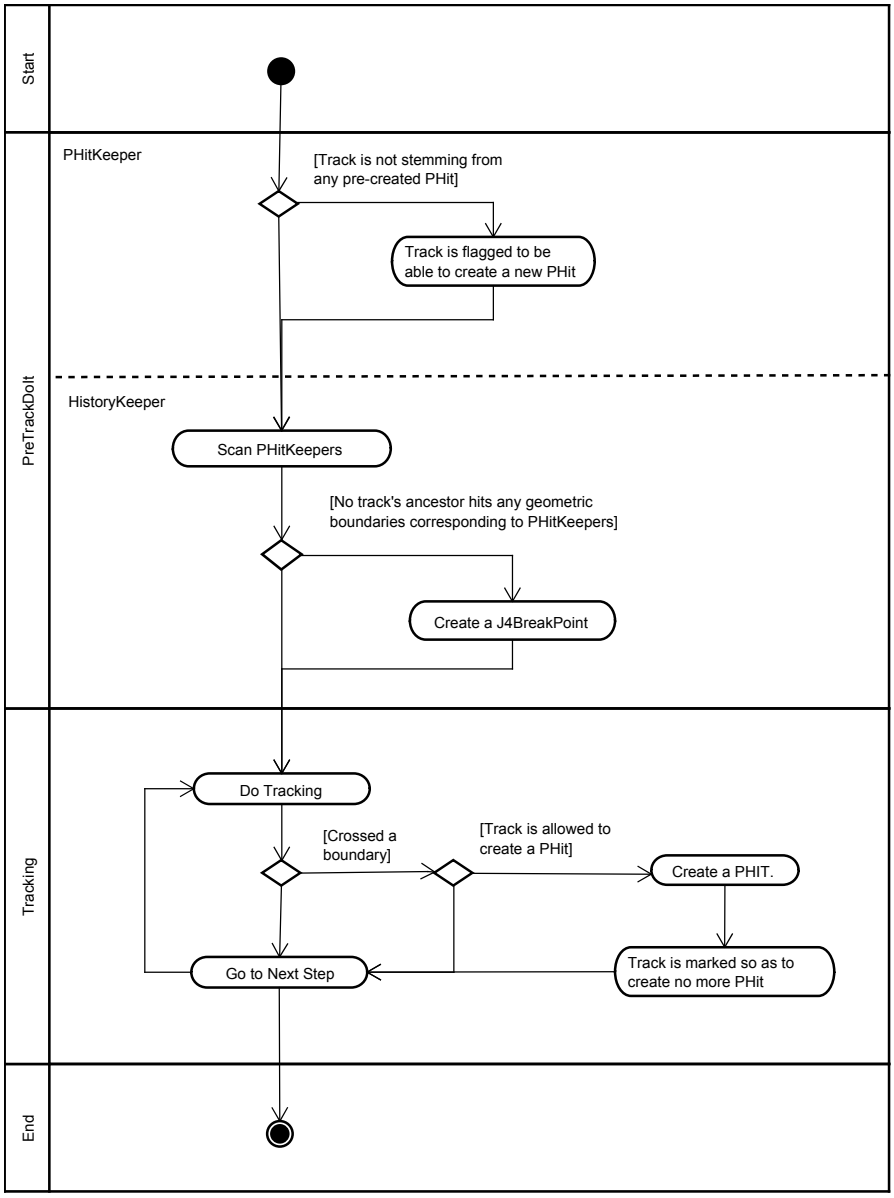

Figure 2: *The flow diagram for Geant4 tracking related to J4HistoryKeeper.*

The abstract base class `J4VSubTrackingAction` has the following methods:

`void PreTrackDoIt(const G4Track *aTrack = 0) = 0`
which is pure virtual and to be implemented by its derived class to take a sub-tracking action for the given track.

`void Clear()`
which does nothing, and to be overridden in the derived class as needed.

This class just specifies the interface and requires its users to implement the methods listed above.

`J4TrackingAction`

The `J4TrackingAction` class is a singleton inheriting from `G4UserTrackingAction`. It has, among others, an STL vector as a data member to store pointers to objects derived from the `J4VSubTrackingAction` class. Its major methods include the following:

`static J4TrackingAction *GetInstance()`
which returns the pointer to the single instance of `J4TrackingAction`.

`void Add(J4VSubTrackingAction *aSta)`
which registers a user-defined object derived from `J4VSubTrackingAction`. When `*aSta` has already been registered, the pre-registered one is erased and the new entry is appended.

`void PreUserTrackingAction(const G4Track *aTrack)`
which loops over the registered offsprings of `J4VSubTrackingAction` and invokes their `PreTrackDoIt` methods.

`void Clear()`
which loops over the registered offsprings of `J4VSubTrackingAction` and invokes their `Clear` methods.

The class diagram for `J4TrackingAction` and `J4VSubTrackingAction` is shown in Fig. 3.

3.b P-Hits and P-Hit Keeper

`PHit` is a generic name for a `Pre-Hit` or a `Post-Hit`, which stands for a virtual hit created on a boundary of a `G4PhysicalVolume` beyond which particle showering is expected. The `PHit` creation is done in the user-overridden `ProcessHits` method of a user-defined virtual detector derived from `G4SensitiveDetector` corresponding to the physical volume. Notice that `PHits` are created for all kinds of particles, even neutrinos, that pass through the boundary. One `PHit` class is defined inheriting from the `J4VTrackerHit` class for each such boundary. The `J4VTrackerHit` class carries basic track hit information such as track ID, particle ID, position, momentum, TOF, energy deposit, etc. and setters and getters to access them. An individual `PHit` class has a data member to store `PHit` ID and a static data member to store the current `PHit` ID, which can be retrieved by a static method to mark calorimeter hits as needed. The class diagram for `J4VTrackerHit`, `J4PHitKeeper`, and related classes is shown in Fig. 4.

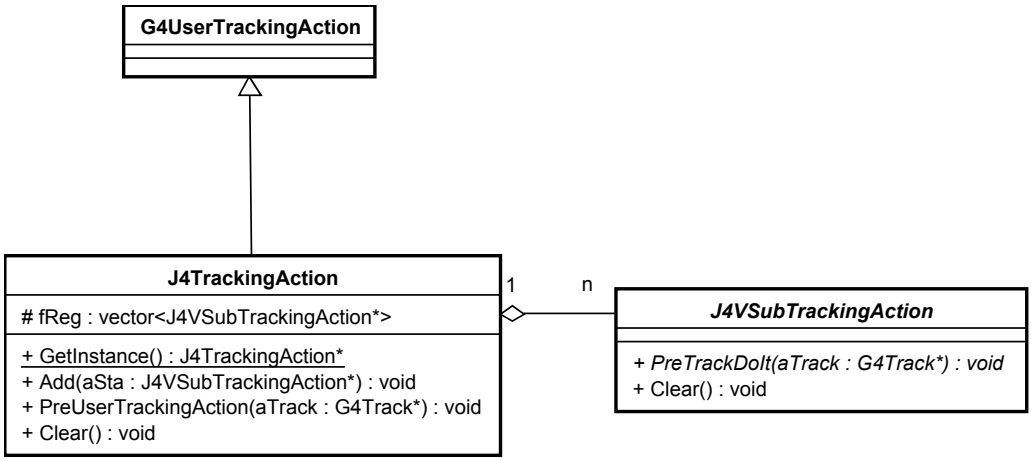

Figure 3: *Class diagram for* `J4TrackingAction` *and* `J4VSubTrackingAction`.

`J4PHitKeeper`

The `J4PHitKeeper` class inherits from `J4VSubTrackingAction`. It serves as a base class for a `PHitKeeper` class defined for an individual `PHit` class corresponding to a boundary beyond which particle showering is expected. The `J4PHitKeeper` class has data members to store i) the current incident track ID (`fInTrackID`) that is expected to create or has already created a `PHit`, ii) the track ID (`fTopTrackID`) of the next track to be processed, if any, after the offsprings from the `PHit` are exhausted, and iii) a flag (`fIsPHitCreated`) to tell whether a `PHit` has been created or not. The major methods of `J4PHitKeeper` are listed below:

`void PreTrackDoIt(const G4Track *)`
: implements the corresponding base class pure virtual method so as to reset `fInTrackID` and `fTopTrackID` to `std::numeric_limits<int>::max()` and `fIsPHitCreated` to `false` upon encountering a new track which has a track ID smaller than `fTopTrackID`.

`G4bool IsNext()`
: returns `false` if a `PHit` has already been created. If not, it updates `fInTrackID` and `fTopTrackID` and returns `true` to tell the caller (the `ProcessHits` method of the sensitive detector defining the virtual boundary) that a new `PHit` is to be created.

`void Reset(G4int k = std::numeric_limits<int>::max())`
: resets `fInTrackID` and `fTopTrackID` to `k`.

`G4bool IsPHitCreated()`
: returns `fIsPHitCreated`, which is `true` if a `PHit` has been created, and `false` otherwise.

The algorithm of `J4PHitKeepr` heavily depends on Geant4's default track stacking scheme, which is worth explaining here for readers unfamiliar with it. By default Geant4 uses two types of track stacks, a Primary Stack (*PS*) and a Secondary Stack (*SS*).

At the beginning of each event, primary particles $1, \cdots, n$ are pushed into *PS*. According to the "last in first out" rule, the top entry, track n, is popped out for tracking. Notice that there remains $n-1$ tracks in *PS* at this point. All the secondary particles produced while track

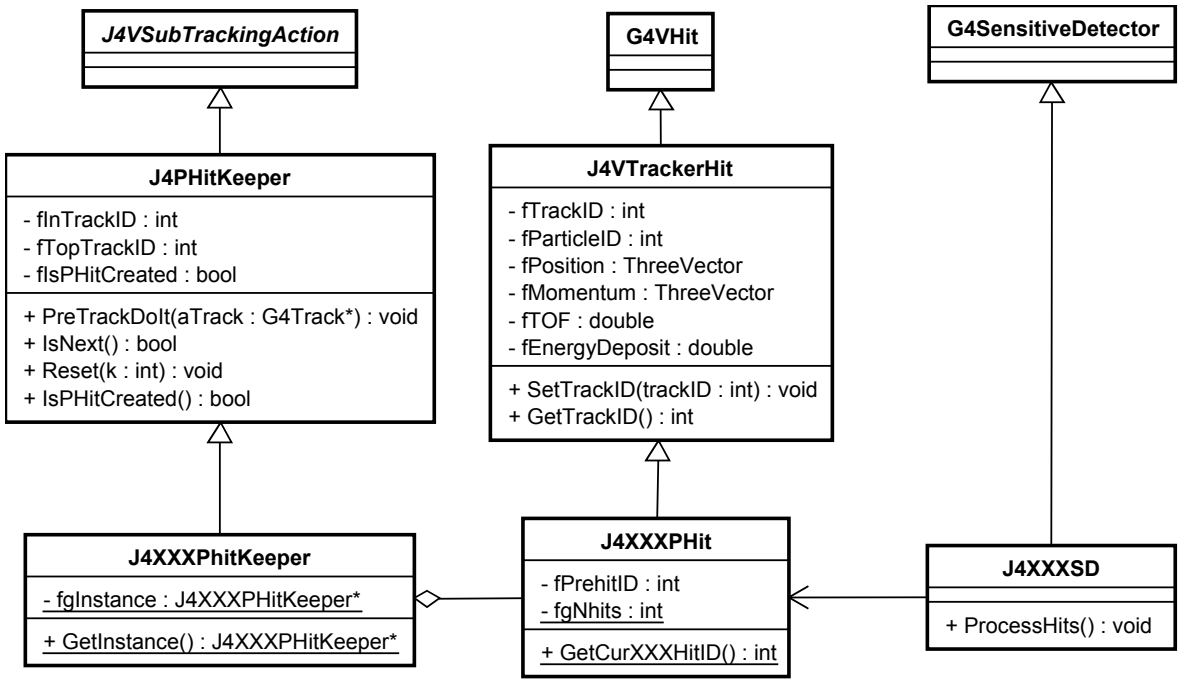

Figure 4: *The class diagram for* `J4PHitKeeper`*,* `J4VTrackerHit`*, and related classes.*

n is being processed are pushed into SS. Let us assume that there will be m secondary particles stacked into SS by the time track n is disposed of. All of these m secondary particles in SS are moved to PS upon the death of track n and numbered serially as tracks $n+1, \cdots, n+m$. Notice that there are $n+m-1$ tracks in PS at this point since track n has been popped out and disposed of.

The key point is to bookmark the secondary track which is to be created just after the creation of a `PHit` by the track which has been being processed, track n in the present case. The track ID with the bookmark will be `fTopTrackID` $= n+k'+1$ where $k'(\leq m)$ is the number of secondary particles in SS at the time of the `PHit` creation. Further `PHit` creation is to be forbidden until it becomes necessary.

The top of the stack, track $n+m$, is popped out and to be processed as before. Track $n+m$ will produce further m' secondary particles to be pushed into PS upon its death and to be numbered as tracks $n+m+1, \cdots, n+m+m'$.

This procedure is repeated and after some time all the secondary particles originating from the track created the last `PHit` will be disposed of and the next track to be popped out from PS will have a track ID that is smaller than that of the last bookmarked one, `fTopTrackID`. This signals a new incident track which is allowed to create a new `PHit`. By repeating this procedure until all the tracks in PS are exhausted, we can mark all the calorimeter hits with corresponding `PHits`.

3.c Break Points and History Keeper

The purpose of the history keeper is to allow us to trace back to kink and V^0 particles that decay before entering calorimeters so as to correctly link clusters to tracks. As sketched above, the his-

tory keeper is implemented as a J4VSubTrackingAction so as to create a J4BreakPoint object for each new track until a PHit is created on any of the pre-registered boundaries beyond which particle-showering is expected. The class diagram for J4BreakPoint and J4HistoryKeeper is shown in Fig. 5.

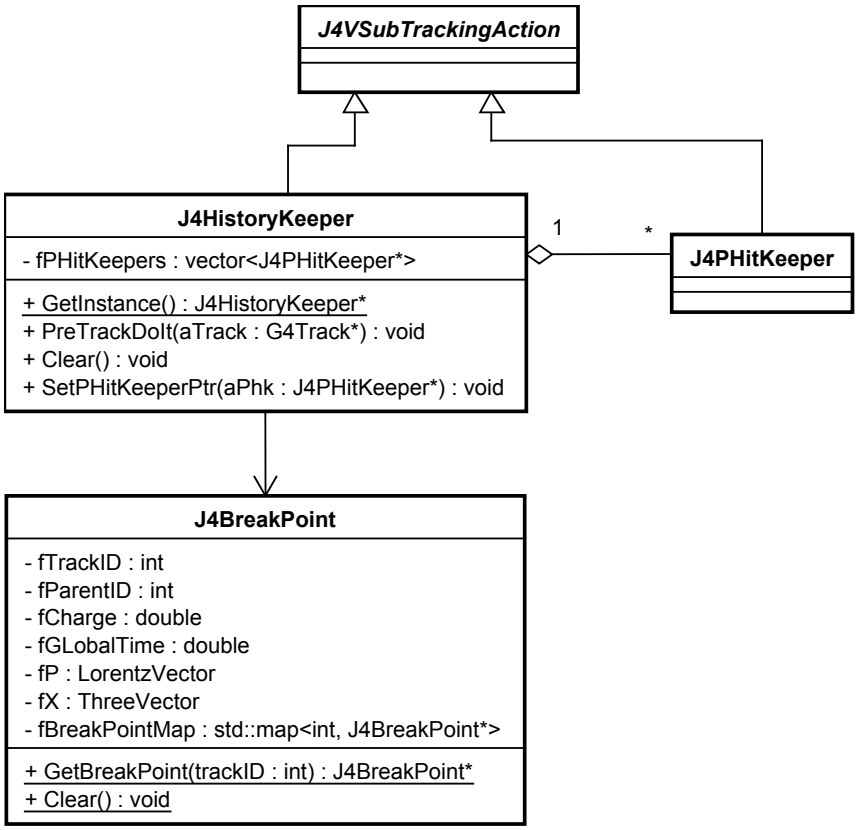

Figure 5: *Class diagram for* J4HistroyKeeper *and* J4BreakPoint.

J4BreakPoint

The J4BreakPoint class has data members to store the information about a track at its starting position such as track ID (fTrackID), parent track ID (fParentID), charge, particle ID, time, position, 4-momentum, etc.. In addition it has a static data member called fgBreakPointMap, which is an STL map that links track ID to a J4BreakPoint object. Besides the getters to these data members, J4BreakPoint has the methods listed below:

`static J4BreakPoint *GetBreakPoint(G4int trackID)`
: returns the pointer to the J4BreakPoint object corresponding trackID.

`static void Clear()`
: clears the track-to-break-point map.

J4HistoryKeeper

The J4HistoryKeeper class is a singleton that inherits from J4VSubTrackingAction. It has an STL vector (fPHitKeepers) as a data member to store registered J4PHitKeepers that correspond to boundaries beyond which particle-showering is expected. As sketched above, it scans through these pre-registered J4PHitKeeper objects to make sure that none of them has a PHit, and then creates a J4BreakPoint object. The major methods of J4HistoryKeeper are listed below:

`static J4HistoryKeeper *GetInstance()`
: returns the pointer to the single instance of J4HistoryKeeper.

`void PreTrackDoIt(const G4Track *)`
: implements the corresponding base class pure virtual method. It scans through the pre-registered J4PHitKeeper objects in fPHitKeepers to make sure that none of them has a PHit by calling their IsPHitCreated() method. It then creates a J4BreakPoint object.

`void Clear()`
: calls J4BreakPoint::Clear().

`void SetPHitKeeperPtr(J4PHitKeeper *aPhk)`
: pushes back the input J4PHitKeeper pointer into fPHitKeepers.

S4BreakPoint

Upon the completion of Monte Carlo truth generation by JUPITER, each J4BreakPoint object is copied to its SATELLITE dual, an S4BreakPoint object. The S4BreakPoint object inherits from ROOT's TObjArray and stores pointers to its daughter S4BreakPoints, if any. It has additional methods such as

`void LockAllDescendants()`
: which flags all of its descendants as locked. This functionality proves handy to avoid double counting of energies.

`TObject *GetPFOPtr()`
: which returns the pointer to its corresponding Particle Flow Object (PFO), if any.

`void SetPFOPtr(TObject *aPfo)`
: which is the setter corresponding to GetPFOPtr to be invoked from a PFO maker.

The class diagram for S4BreakPoint is shown in Fig. 6.

4 Tool Usage

What a tool user has to do for the history keeping is as follows:

- Inheriting G4SensitiveDetector, create a sensitive detector class, say J4XXXSD, that corresponds to a boundary on which a PHit object (J4XXXPHit) is to be created for each particle that is expected to produce a shower beyond that boundary.

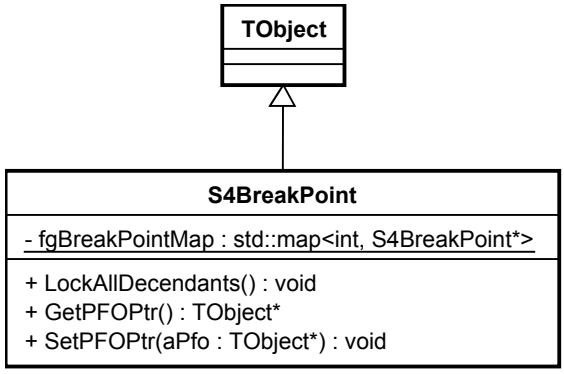

Figure 6: *Class diagram for* `S4BreakPoint`.

- Inheriting `J4PHitKeeper`, create a `J4XXXPHitKeeper` as a singleton to book-keep `J4XXXPHit`s.
- In `J4XXXSD`'s constructor, do

```
J4XXXPHitKeeper *aPhk = J4XXXPHitKeeper::GetInstance();
J4TrackingAction::GetInstance()->Add(aPhk);
J4TrackingAction::GetInstance()->Add(J4HistoryKeeper::GetInstance());
J4HistoryKeeper::GetInstance()->SetPHitKeeperPtr(aPhk);
```

  in order to register the `J4XXXPHitKeeper` to `J4TrackingAction` and to `J4HistoryKeeper`.
- In `J4XXXSD`'s `ProcessHits(···)` method, do

```
if (J4XXXPHitKeeper::GetInstance()->IsNext()) {
// create and store a J4XXXPHit object
}
```

- In `ProcessHits(···)` of each calorimeter sensitive detector, which usually corresponds to a single calorimeter cell, store the centers of gravity and energy deposits of particles from different `PHit`s as different calorimeter hits even in the same cell and mark them with the current `PHit` ID obtainable from an appropriate `J4PHitKeeper` object.

This ensures the history keeping to be continued until any one of the pre-registered boundaries is hit and beyond which the calorimeter hits are marked with the `PHit` ID put to the `PHit` created on that boundary.

As an example of J4HistoryKeeper application, a typical multi-particle event detected with a generic e^+e^- collider detector model is shown in Fig. 7. It is an $e^+e^- \to q\bar{q}$ event at the center of mass energy of 350 GeV. Circles in the figure indicate, from outside to inside, the outer and the inner boundaries of the barrel calorimeter, and the inner wall of the central tracker, a Time Projection Chamber (TPC), respectively. The calorimeter inner radius of 210 cm sets the scale of the detector model. Both the tracker and the calorimeter are placed in a solenoidal magnetic field of 3 Tesla. Using the J4HistoryKeeper package, calorimeter signals are matched to their corresponding track signals and are painted with the same color. With

J4HistoryKeeper, the centers of gravity of the calorimeter signals can be calculated as with infinite segmentation, which proves extremely useful in investigating the ultimate performance of the detector system with infinitely granular calorimetry.

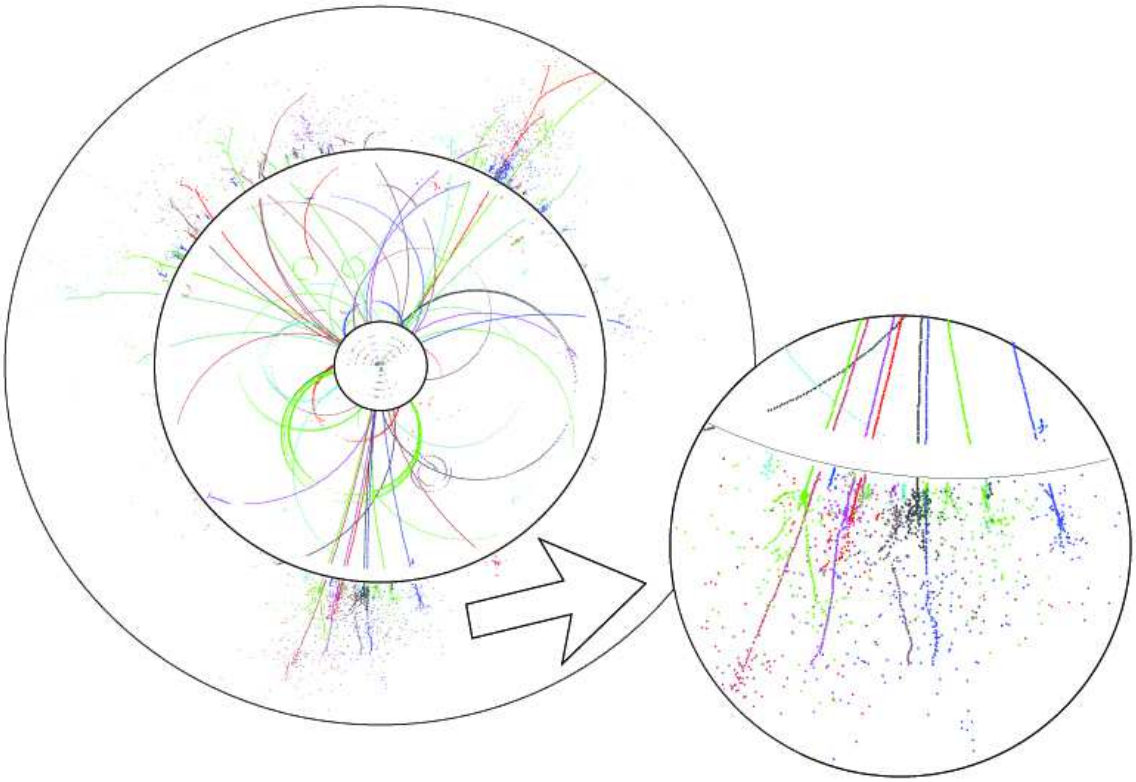

Figure 7: *A typical $e^+e^- \to q\bar{q}$ event with a hard gluon emission at the center of mass energy of 350 GeV, viewed from the beam direction. Matching calorimeter and tracker signals are identified by using the J4HisotryKeeper package and painted with same colors. Three circles indicate, from outside to inside, the outer and the inner boundaries of the calorimeter, and the inner wall of the TPC. Points inside the TPC inner wall are signals by silicon trackers.*

5 Conclusion

We have developed a software tool, J4HistoryKeeper, for history keeping of Geant4 tracks. J4HistoryKeeper records history of Geant4 tracks starting from the interaction point until they reach any of user-registered geometrical boundaries. The tool allows us to record their positions, momenta, trackIDs, TOFs, etc. at their birth points as well as at the user-registered boundaries. The flexible registration capability of the user-given boundaries and a mechanism to achieve effectively-infinite segmentation with a finite readout cell size comprise the core features of the package. These features provide J4HistoryKeeper with wider applications though it has been designed primarily for PFA studies. The package comes in handy in any application where history keeping is necessary but particle showering and consequently memory need explosion

are expected beyond multiple boundaries of a complicated detector geometry. The tool has been used for the so-called Cheated PFA to investigate limiting factors and ultimate performance of jet energy measurements at the future linear e^+e^- collider.

Acknowledgments

The authors would like to thank T. Yoshioka, H. Ono, T. Takeshita, and other members of the JLC-Software group for useful discussions and helps. This work was supported in part by the Creative Scientific Research Grant No. 18GS0202 of the Japan Society for Promotion of Science (JSPS) and the JSPS Core University Program.

References

[1] http://www.linearcollider.org/ and references therein.

[2] JLC group, KEK Report 92-16, December, 1992.

[3] M.A.Thomson, arXiv:physics/060726; V.Morgunov and A.Raspereza, arXiv:physics/0412108; T.Yoshioka, ECONF C0508141:ALCPG1711,2005; S.Magill and S.Kuhlmann, SLAC-PUB-12203, in the Proceedings of 2005 International Linear Collider Workshop (LCWS 2005), Stanford, California, 18-22 Mar 2005, pp 1015.

[4] ACFA Linear Collider Working Group, KEK Report 2001-11, August (2001), http://acfahep.kek.jp/acfareport/.

[5] Proceedings of the APPI Winter Institute, KEK Proceedings 2002-08, July (2002).

[6] http://wwwinfo.cern.ch/asd/geant4/G4UsersDocuments/UsersGuides/ForToolkitDeveloper/html/.

[7] http://acfahep.kek.jp/subg/sim/simtools/.

[8] http://root.cern.ch/.

[9] http://www-jlc.kek.jp/subg/offl/cpfa/.