

# Simulating Water Cherenkov Detectors Using WCSim

Thomas Dealtry, Alexander Himmel, Johannes Hoppenau, Joseph Lozier

January 27, 2016

## Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Overview</b>   | <b>1</b>  |
| <b>2</b> | <b>The WCSim Geometry</b>   | <b>2</b>  |
| 2.1      | Hierarchy of Volumes . . . . .                                    | 2         |
| <b>3</b> | <b>Setup your own detector</b>                                    | <b>4</b>  |
| 3.1      | Parameters . . . . .  | 4         |
| 3.2      | Example . . . . .   | 8         |
| 3.3      | Warnings . . . . .  | 9         |
| 3.4      | Input Files . . . . .   | 10        |
| <b>4</b> | <b>DAQ classes - for dark noise, digitization, and triggering</b> | <b>10</b> |
| 4.1      | WCSimWCAddDarkNoise . . . . .                                     | 10        |
| 4.2      | WCSimWCDigitizer . . . . .  | 11        |
| 4.3      | WCSimWCTrigger . . . . .  | 12        |
| 4.4      | Input files . . . . .   | 13        |
| 4.4.1    | Digitizer options . . . . .                                       | 13        |
| 4.4.2    | Trigger options . . . . .   | 14        |
| 4.4.3    | Dark noise options . . . . .                                      | 15        |
| <b>5</b> | <b>Output Root File</b>   | <b>15</b> |
| 5.1      | The Class Hierarchy . . . . .                                     | 16        |
| 5.2      | How to Use the Files . . . . .                                    | 19        |

# 1 Overview

WCSim is a flexible Geant4-based simulation of a water-Cherenkov detector with top and side photo-multiplier tubes. Given basic parameters about the detector, it automatically lays out the PMTs so you can get started simulating events as soon as possible. This document will describe the detector geometry and all its elements. It describes how to set up a new custom detector and how to configure the simulation options. Finally, it describes the ROOT output file and how to access the stored data with some simple examples.

## 2 The WCSim Geometry

The `WCSimDetectorConstruction::ConstructWC()` method returns a pointer to a logical volume that contains an upright cylindrical detector. The function is defined in `src/WCSimConstructWC.cc`. The inner detector consists only of blacksheet and PMTs, and the optional top veto outer detector contains only blacksheet, whitesheet, and PMTs.

The PMTs and the blacksheet are organized into cells along the cylinder walls plus a top and bottom cap. This structure makes it easy to add outer detector PMTs to the sides and bottom or a steel structure behind the blacksheet. The method is written generically, making it possible to simulate many different detectors.

### 2.1 Hierarchy of Volumes

This section describes the the volumes that make up the detector. (Figure 1)

**ExpHall** is the world volume. It is not constructed in `src/WCSimConstructWC.cc` but in `src/WCSimDetectorConstruction.cc`.

**WC** is a tubs filled with air. At the moment it only contains one daughter volume:

**WCBarrel** is a tubs filed with water. It contains all of the current detector structure (PMTs and blacksheet), divided into the annulus and the caps:

**WCBarrelAnnulus** is the main part of the detector wall. It is divided into rings:

**WCBarrelRing** each one cell high, which are then divided into cells:

**WCBarrelCell** These cells contain one or more PMTs (**WCPMT**'s) and the blacksheet (**WCBarrelCellBlackSheet**). Each cell is flat and represents one modular detector section.

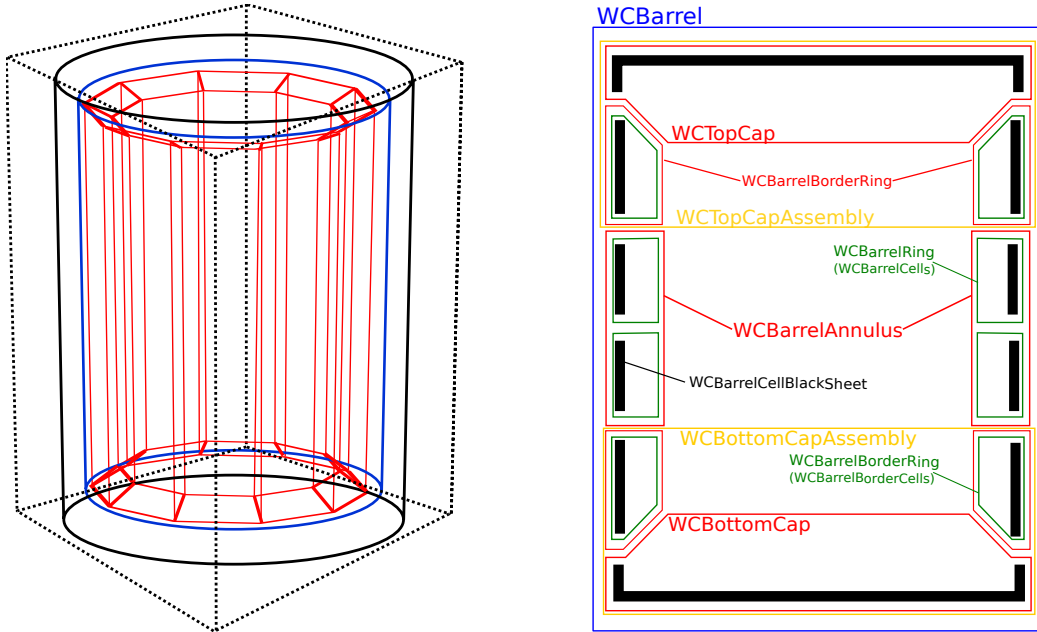
**WCTopCapAssembly** and **WCBottomCapAssembly** are two mirrored volumes that close the ends of the barrel. They are constructed by calling `src/WCSimWCSimDetectorConstruction::ConstructCaps(G4int zflip)`, where the argument, `zflip`, equals 1 to generate the **BottomCapAssembly** and -1 to generate the **TopCapAssembly**. This method allows symmetric changes to the caps by editing `ConstructCaps` and avoids using

Geant's built-in ReflectionFactory, which is incompatible with the OpticalSurface used to model certain PMT properties.

**WCCap** This volume contains all of the cap PMTs (**WCPMT**) and the blacksheet behind them (**WCCapBlackSheet**) which extends around the cylinder edges to connect to:

**WCB BarrelBorderRing** This volume connects the annulus to the cap. It is essentially the uppermost (or lowermost) ring of PMTs, but has a modified bounding-box geometry and is contained in the CapAssembly because it must mate at the corner with the caps. (N.B. See 3.3, for information on corner geometry.) The border ring is divided into cells:

**WCB BarrelBorderCell** that contain the same number of PMTs (**WCPMT**) and the same blacksheet (**WCB BarrelCellBS**) as the normal annulus cells.



**Figure 1:** On the left is a 3D schematic of the cylindrical detector and on the right is a 2D cross-section that better shows the volume hierarchy contained in **WCB Barrel**. The outermost level is **ExpHall** (dotted black line) which is an air-filled rectangle. It contains **WC** (narrow black line), the volume returned by `WCSimDetectorConstruction::ConstructWC()`. Inside it is the water-filled cylinder **WCB Barrel** (blue) whose sub-volumes are labeled on the right and described in the text below.

**WCExtraTower** is a tower that is narrower than the normal cells that is added if the number of PMTs circumferentially is not divisible by the number of PMTs horizontally in each cell. It is divided into cells:

**WCExtraTowerCell** that contains the remaining PMTs (**WCPMT**) and blacksheet (**WCTowerBlackSheet**) (figure 2). The cap assemblies contain a corresponding this corresponding extra cell called **WCExtraBorderCell**.

**WCPMT** is a single cylindrical volume containing a single PMT model that is placed many times in the detector. The volume is described in `src/WCSimWCSimConstructPMT.cc`. The surrounding volume is cylindrical with a coordinate origin where its axis intersects the blacksheet (not a standard tubs) (red box, 5). The PMT is made up of two sub-volumes, both spherical caps: the hollow glass outer face of the PMT (**GlassFaceWCPMT**) and the inner vacuum (**InteriorWCPMT**). The optical inner coating of the PMT glass is modeled as an **OpticalSurface** (not a volume) (**GlassCathodeSurface**) between the glass and vacuum. (See also 5.) Any components added to this single volume (e.g., an acrylic pressure vessel) are replicated and placed along with every PMT.

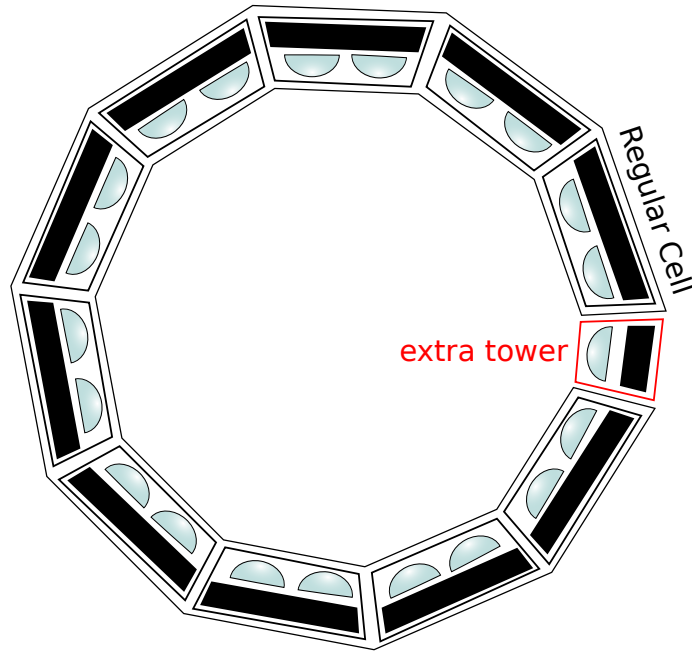
### 3 Setup your own detector

The dimensions of the detector to be setup must be described in member variables of `WCSimDetectorConstruction` before calling `WCSimDetectorConstruction::ConstructWC()`. This section describes which geometric parameters must be set. The easiest and least error-prone way to set the variables is to add a method to the `WCSimDetectorConfigs.cc` file that is a copy of an existing detector configuration with the necessary changes. You can either call this function in the constructor of `WCSimDetectorConstruction` or add it to the detector messenger (`src/WCSimDetectorMessenger.cc`) if you want call it in a macro file or you want to change the detector setup dynamically. If you want to do this, you first have to use the command that calls your function (`/WCSim/WCgeom <geometry name>`) to set the parameters, and afterwards the detector construction command (`/WCSim/Construct`) to (re)construct the geometry.

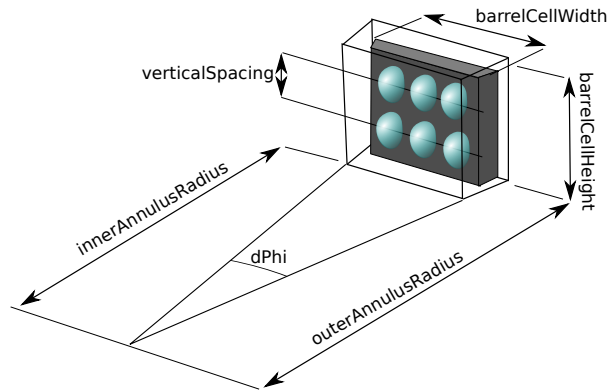
#### 3.1 Parameters

To set up a new detector geometry the following parameters must be set:

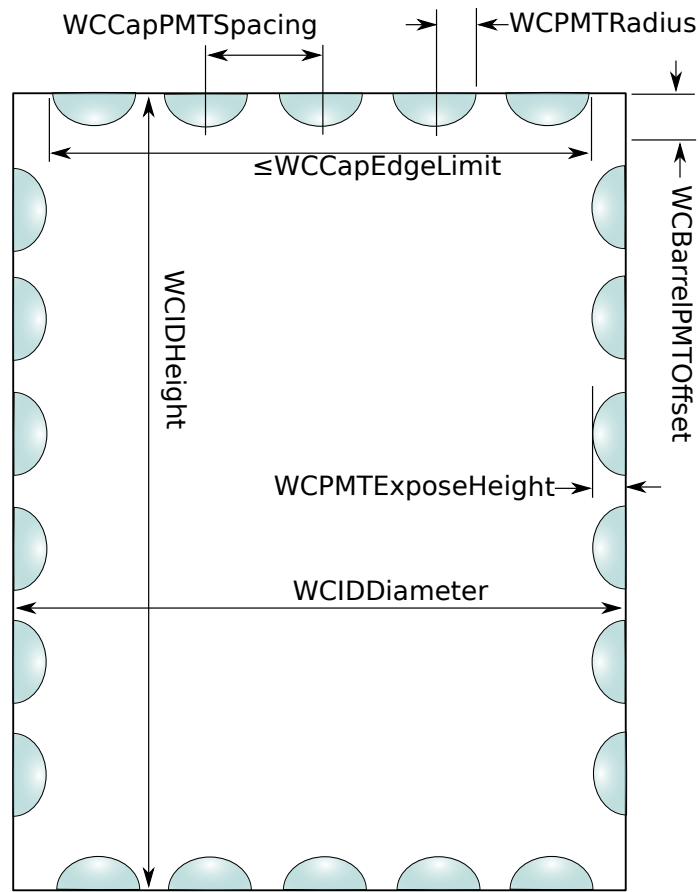
**CreatePMTObject("PMTType")** instantiates a PMT of type **PMTType** as defined in the `WCSimPMTObject.cc` file. This function also returns the pointer to the **PMTObject** and stores the pointer in memory to be accessed by other files which use the PMT properties (for example, the pe conversion in `WCSimWCPMT.cc`). The current options are **PMTType** = **PMT8inch**, **PMT10inch**, **PMT10inchHQE**, **PMT12inchHQE**, **PMT20inch**, or **HPD20inchHQE**.



**Figure 2:** If the number of PMTs in one cell (in this example, 2) does not divide the total number of PMTs in one ring (in this example, 2), there is an extra tower that contains the remaining PMTs



**Figure 3:** Each cell holds blacksheet and multiple PMTs. All labeled lengths are calculated automatically. In the current version, the PMTs are distributed equally in horizontal and vertical direction.

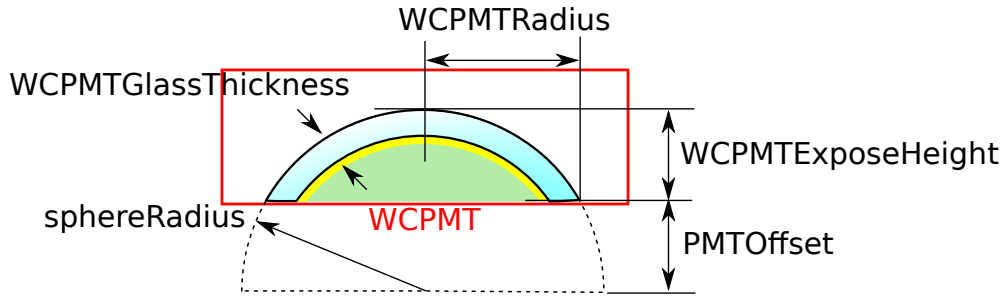


**Figure 4:** These are among the variables you have to set to create a new detector geometry.

**WCPMTRadius**, **WCPMTExposeHeight** (see figure 5) are the radius at blacksheet and height above blacksheet, respectively, of the PMTs. This information is retrieved using the pointer that is returned by **CreatePMTObject**.

**WCPMTGlassThickness** the thickness of the glass face. This information is retrieved using the pointer that is returned by **CreatePMTObject**.

**WCIDDiameter**, **WCIDHeight** These two variables are used to setup the size of the detector. The height is the distance between the inner surfaces of the top and bottom blacksheets and the diameter is two times the shortest distance between the inner surface of the wall blacksheet and the center of the detector (see figure 4). This shortest radius occurs at the center of normal (not extra) cells, and is perpendicular to the blacksheet.



**Figure 5:** The PMTs are segments of spheres. All parts are contained in the bounding cylinder, **WCPMT** (red). The PMT glass (**GlassFaceWCPMT**, blue) and the sensitive volume, the inner vacuum (**InteriorWCPMT**, green) are contained within. Also present is the optical coating between the glass and vacuum (**GlassCathodeSurface**, yellow). To define the geometry of the PMTs you need to set the **WCPMTGlassThickness**, the **WCPMTRadius**, and the **WCPMTExposeHeight**

**WCPMTperCellHorizontal**, **WCPMTperCellVertical** The cap volumes contain vertical space in stripes along the edges of the detector to make room for the cap PMTs (N.B. See 3.3, for information on corner geometry). This variable defines the width of these stripes. Specifically, the offset is the vertical distance from the inner surface of the cap blacksheet to the upper edge of the top cell. This edge is half the vertical PMT spacing vertically above the center of each PMT in the uppermost (or lowermost) ring.

**WCPMTperCellHorizontal**, **WCPMTperCellVertical** are two integers that define the arrangement of PMTs within each cell, the product of which gives the number of PMTs in each cell.

**WCPMTperCellHorizontal** defines the number of PMTs circumferentially around the detector. If **WCPMTperCellHorizontal** does not divide this number, there will be an extra cell in each ring, which contains the remaining PMTs.

**WCPMTperCellVertical** defines how many rings of cells there will be vertically. The total number of PMTs in a vertical column will be the product of this number and **WCPMTperCellHorizontal**.

**WCPMTSpacing** defines the center-to-center spacing of the PMTs on the top and the bottom caps of the detector.

**WCPMTEdgeLimit** is the maximum distance between the center of the cap and the outer edge of a cap PMT (the edge is **WCPMTRadius** away from the center of the PMT, whose coordinates on the cap are half-integer multiples of **WCPMTSpacing**). This length has to be smaller than half the **WCIDDiameter**. Otherwise there may be

PMTs that intersect the edge of the caps. The WCSimConstructWC places PMTs on the caps in a grid subject only to this constraint. Note that the four centermost cap PMTs are equidistant from the cylinder axis.

**WCBlackSheetThickness** the thickness of the blacksheet.

**WCAddGd** a boolean that, when true, dopes the water volume with .01% Gadolinium by mass.

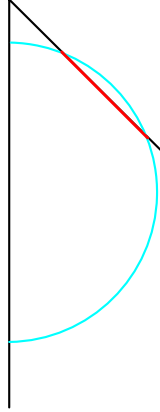
All other values needed to set up the geometry are derived from these variables.

### 3.2 Example

```
void WCSimDetectorConstruction::SetSuperKGeometry()
{
    WCSimPMTObject * PMT = CreatePMTObject("PMT20inch");
    WCPMTName = PMT->GetPMTName();
    WCPMTExposeHeight = PMT->GetExposeHeight();
    WCPMTRadius = PMT->GetRadius();
    WCPMTGlassThickness = PMT->GetPMTGlassThickness();
    WCIDDiameter          = 33.6815*m; //16.900*2*
                                   //cos(2*pi*rad/75)*m;
    WCIDHeight            = 36.200*m;
    WCBarelPMTOffset      = 0.0715*m; //offset from vertical
    WCBarelNumPMTHorizontal = 150;
    WCBarelNRings          = 17.;
    WCPMTperCellHorizontal = 4;
    WCPMTperCellVertical   = 3;
    WCCapPMTSpacing        = 0.707*m; // distance between centers
                                   // of top and bottom pmts
    WCCapEdgeLimit         = 16.9*m;
    WCBlackSheetThickness  = 2.0*cm;
    WCAddGd                = false;
}
```

This is the Super-K setup. This method is located at the beginning of `src/WCSimWCSimDetectorConfigs.cc`. It is called in the constructor of `WCSimDetectorConstruction`. In SK the PMTs are arranged in  $4 \times 3$  cells (`WCPMTperCellHorizontal` and `WCPMTperCellVertical`). All in all there are 51 rings of PMTs (3 lines in each cell times 17 lines of cells (`WCBarelNRings`)). Each line contains 150 PMTs (`WCBarelNumPMTHorizontal`). As 150 divided by 4 is 37.5, there are 37 regular  $4 \times 3$  cells and one  $2 \times 3$  cell in one ring. Note that Super-K's plans specify a 16.9 meter radius to the corner between cells, some trigonometry was required to translate this to a perpendicular distance, almost 6 cm less.





**Figure 6:** The topmost and bottommost PMTs could intersect the border of the cells.

The vertical spacing of the PMTs is

$$\frac{WCIDHeight - 2 \cdot WBarrelPMTOffset}{WBarrelNRings \cdot WCPMTperCellVertical} = 0.707m.$$

between the bottom (and top) blacksheet and the cells on the wall, there is a gap of 7.15 cm ( $WBarrelPMTOffset$ ).

The caps are completely filled with PMTs, because  $WCapEdgeLimit$  is equal to the detector diameter.

### 3.3 Warnings

If the PMTs have a large  $WCPMTExposeHeight$  and there is not enough space between the PMTs and the borders of the cells, the PMTs could intersect the edge of the border cells, because the border of these cells are slanted (see figure 6).

This can also happen at the caps if  $WCapEdgeLimit$  is close to the inner radius of the detector and  $WBarrelPMTOffset$  is small.

During the setup an incomplete checking for obvious overlaps occurs. You should see a lot of the following lines:

```
Checking overlaps for volume WBarrelPMT ... OK!
```

If you see warnings instead, it is likely that there are too large or too many PMTs. An absence of warnings does not mean that there are no overlaps.

There is no check if the placement of the PMTs on caps is correct. It would take too much time to do this check every time, because there are too many PMTs in side a single volume.

Some rules of thumb to avoid overlaps are:  $WBarrelPMTOffset > WCPMTExposeHeight$  or  $WCIDDiameter / 2 - WCapEdgeLimit > WCPMTExposeHeight$  ensures cap PMTs are fully within the cap volume, and vertical spacing of the PMTs  $/ 2 > WCPMTExposeHeight$

+ `WCPMTRadius` ensures the top and bottom ring PMTs are within `WBarrelBorderRing`. These rules are general, and PMTs may be placed closer with careful attention to geometry. Collisions are not an issue in Super-K, and newer detectors with smaller, more-efficient (and thus sparser) PMTs should have little issue provided reasonable specifications of `WCCapEdgeLimit` and `WBarrelPMTOffset`.

### 3.4 Input Files

Some tuning parameters are found in `jobOptions.mac` and `tuning_parameters.mac`, which are in separate files because they must be loaded at specific times during initialization. The bulk of options, however, may be found in `WCSim.mac`, the default input file. `WCSim` is a GEANT4 program and accepts GEANT4 commands as an input file or at runtime. GEANT4 documentation describes the commands not detailed here.

`/WCSim/WCgeom` selects a set of geometry parameters (see 3.1.) It does not create a new geometry. New geometries are easily added and a full list of those already available may be found in `src/WCSimDetectorMessenger.cc`

`/WCSim/Construct` constructs the detector geometry in memory based on the previously selected parameters. Takes no arguments.

`/WCSim/PMTQEMethod` Selects the quantum efficiency method. Possible arguments are: `Stacking_Only`, in which the QE is applied to reduce the total number of photons when the photons are generated; `Stacking_And_SensitiveDetector`, which the (constant) QE at the most efficient wavelength is applied at photon creation, then the remaining (wavelength-dependent) loss is applied at the detector; and `SensitiveDetector_Only`, in which QE is applied at the detector only.

`/WCSim/PMTColEff` Selects wavelength-dependent (on) or -independent (off) quantum efficiency model.

`/WCSim/SavePi0` Selects whether or not  $\pi^0$ -specific information is saved, options are `true` and `false`.

## 4 DAQ classes - for dark noise, digitization, and triggering

### 4.1 `WCSimWCAddDarkNoise`

In order for the effect of dark noise to be included correctly in the simulation it must be added throughout the time of the event. The easiest way to do this would be to add dark noise in a large window that is sure to incorporate any late activity (e.g. Michel electrons). Unfortunately this is computationally expensive, so this class has various options to add the dark noise only at relevant times.

**In a predefined absolute time window** Set the first/last time and fill in the area in-between with dark noise.

**In a predefined time window around each hit** Set the duration of the window (note that noise is added in the range hit time  $- \tau/2$  to hit time  $+ \tau/2$ , where  $\tau$  is the window value in the .mac file).

The optimal mode and values need to be tuned to each simulation type (e.g. electron particle gun, muon particle gun, beam, atmospherics, ...).

## 4.2 WCSimWCDigitizer

This part of the code uses a base class (WCSimWCDigitizerBase), and concrete implementations (e.g. WCSimWCDigitizerSKI). The purpose of the class is to take an input WCSimWCDigitsCollection ("WCRawPMTSignalCollection", the collection of hits including dark noise) and output a WCSimWCDigitsCollection ("WCDigitizedStoreCollection", the collection of digits). In this, a "hit" is a photoelectron (from a Cherenkov photon or dark noise) depositing charge on a PMT, and a "digit" is an integration of photons which includes electronics threshold effects.

In order to implement your own digitizer class, you must create your own class derived from WCSimWCDigitizerBase in WCSimWCDigitizer.{hh,cc} and implement the following:

**constructor** This must:

- Call the base constructor i.e. WCSimWCDigitizerBase(name, myDetector, myMessenger, DigitizerType\_t)
- Set triggerClassName
- Call GetVariables()

**void DigitizeHits(WCSimWCDigitsCollection\*)** Should do charge integration, pulse fitting, etc.

**static void Threshold(double& pe, int& iflag)** Should set conditions (e.g. charge cut) when a digit is rejected

**int GetDefaultDeadTime()**

**int GetDefaultIntegrationWindow()**

If your new class has any options (e.g. integration time, deadtime, ...) these options should be added to WCSimWCDigitizerBase and not the derived class. This is due to the way that the options are read in.

The complete steps to add a new digitizer class are as follows:

- Create the new class, derived from WCSimWCDigitizerBase.

- Add the new digitizer type to `WCSimEnumerations.hh`.
- Add the creation of the new digitizer class to the `WCSimEventAction` constructor.
- Add the new option to the list of allowed digitizers in `WCSimWCDAQMessenger` (there are 2 places in the code to add this!).
- If there are any new-digitizer-specific options:
  - Add the new parameters to the `WCSimWCDigitizerBase` class, including set methods.
  - Add the new options to the `WCSimWCDAQMessenger` class, including calls to the new `WCSimWCDigitizerBase` set methods.
  - Document the new options in this document, and in `daq.mac`.

`WCSimWCTriggerBase` assumes that “`WCDigitizedStoreCollection`” has, on a given PMT, its digits ordered in time. This ordering should be maintained in all new digitizer classes, in order for trigger classes to function correctly.

### 4.3 `WCSimWCTrigger`

This part of the code uses a base class (`WCSimWCTriggerBase`), and concrete implementations (e.g. `WCSimWCTriggerNDigits`). The purpose of the class is to take an input `WCSimWCDigitsCollection` (“`WCDigitizedStoreCollection`”, the collection of digits) and output a `WCSimWCDigitsCollection` (“`WCDigitizedCollection`”, the collection of triggered digits), and a series of vector’s with the trigger times, trigger types, and extra trigger information.

In order to implement your own trigger class, you must create your own class derived from `WCSimWCTriggerBase` in `WCSimWCTrigger.{cc, hh}` and implement the following:

**constructor** This must:

- Call the base constructor i.e. `WCSimWCTriggerBase(name, myDetector, myMessenger)`
- Set `triggerClassName`
- Call `GetVariables()`

**void DoTheWork(`WCSimWCDigitsCollection*`)** Calls the relevant trigger algorithm(s). If calling multiple algorithms, also handles the creation/deletion of intermediate `WCSimWCDigitsCollection`’s.

**void Alg\*(`WCSimWCDigitsCollection*`)** Your new algorithm. Should fill the vector’s: `TriggerTimes`, `TriggerTypes`, `TriggerInfos` with relevant information. Currently all algorithms are added to `WCSimWCTriggerBase` in order for new triggers to use multiple algorithms without copying code.

The following default values of trigger options can also be implemented, if an NDigits-like trigger is used in the class:

`int GetDefaultMultiDigitsPerTrigger()`

`int GetDefaultNDigitsWindow()`

`int GetDefaultNDigitsThreshold()`

`int GetDefaultNDigitsPreTriggerWindow()`

`int GetDefaultNDigitsPostTriggerWindow()`

If your new class has any options (e.g. threshold, ...) these options should be added to `WCSimWCTriggerBase` and not the derived class. This is due to the way that the options are read in.

The complete steps to add a new trigger class are as follows:

- Add the new algorithm to `WCSimWCTriggerBase`.
- Add the new trigger type to `WCSimEnumerations.hh` (if applicable).
- Create the new class, derived from `WCSimWCTriggerBase`.
- Add the new option to the list of allowed triggers in `WCSimWCDAQMessenger` (there are 2 places in the code to add this!).
- Add the creation of the new trigger class to the `WCSimEventAction` constructor.
- If there are any new-trigger-specific options:
  - Add the new parameters to the `WCSimWCTriggerBase` class, including set methods.
  - Add the new options to the `WCSimWCDAQMessenger` class, including calls to the new `WCSimWCTriggerBase` set methods.
  - Document the new options in this document, and in `daq.mac`.

## 4.4 Input files

### 4.4.1 Digitizer options

The digitizer class to use is chosen in `WCSim.mac`, the default input file.

`/DAQ/Digitizer` selects the digitizer to use. Available arguments include `SKI`. New digitizers are easily added and a full list of those already available may be found in `src/WCSimWCSimWCDAQMessenger.cc`.

Digitizer-specific options are specified in `daq.mac`.

`/DAQ/DigitizerOpt/IntegrationWindow` selects how long the digitizer integrates for. The default is class specific (for SKI it is 400 ns).

`/DAQ/DigitizerOpt/DeadTime` selects for how long after creating a digit the digitizer is dead for. The default is class specific (for SKI it is 0 ns).

#### 4.4.2 Trigger options

The trigger class to use is chosen in `WCSim.mac`, the default input file.

`/DAQ/Trigger` selects the trigger class to use. Available arguments include `NDigits`. New triggers are easily added and a full list of those already available may be found in `src/WCSimWCSimWCDAQMessenger.cc`.

Trigger-specific options are specified in `daq.mac`.

`/DAQ/MultiDigitsPerTrigger` specifies whether to allow multiple digits per PMT per trigger, or restrict the saved digits in a trigger to the first on a PMT. The default is class specific (for `NDigits` it is false).

`/DAQ/TriggerNDigits/Threshold` selects the threshold number of hits for the `NDigits` trigger. The default is class specific (for `NDigits` it is 25).

`/DAQ/TriggerNDigits/Window` selects the time window to apply the `NDigits` trigger to. The default is class specific (for `NDigits` it is 200 ns).

`/DAQ/TriggerNDigits/AdjustForNoise` specifies whether the `NDigits` threshold should be automatically increased to take account of the average dark noise rate. The default is true.

`/DAQ/TriggerNDigits/PreTriggerWindow` selects how far in the past (relative to the trigger time) to save digits to the output file. The value is forced negative. The default is class specific (for `NDigits` it is  $-400$  ns).

`/DAQ/TriggerNDigits/PostTriggerWindow` selects how far in the future (relative to the trigger time) to save digits to the output file. The value is forced positive. The default is class specific (for `NDigits` it is  $+950$  ns).

`/DAQ/TriggerSaveFailures/Mode` selects which triggers to save. Allowed options are:

0. Save only events which pass triggers;
1. Save both events which pass triggers, and events which fail triggers (with a dummy trigger time);

2. Save only events which fail triggers (with a dummy trigger time).

The default mode is 0.

**/DAQ/TriggerSaveFailures/TriggerTime** selects the dummy trigger time for events which fail all triggers. The default is 100 ns.

**/DAQ/TriggerSaveFailures/PreTriggerWindow** selects how far in the past (relative to the trigger time) to save digits to the output file. The value is forced negative. The default is  $-400$  ns.

**/DAQ/TriggerSaveFailures/PostTriggerWindow** selects how far in the future (relative to the trigger time) to save digits to the output file. The value is forced positive. The default is  $+950$  ns.

#### 4.4.3 Dark noise options

Dark noise options may be chosen in `WCSim.mac`, the default input file.

**/DarkRate/SetDarkRate** selects the dark noise rate. Common values are 0 kHz (i.e. off), 4.2 kHz (SKI default), and 8.4 kHz (20" HPDs and box and line PMTs).

**/DarkRate/SetConvert** converts dark noise frequency before digitization to after digitization by setting suitable factor. Common values are 1.367 (normal PMTs), 1.120 (HPDs), 1.126 (box and line PMTs).

**/DarkRate/SetDarkMode** selects how to add dark noise. Allowed options are:

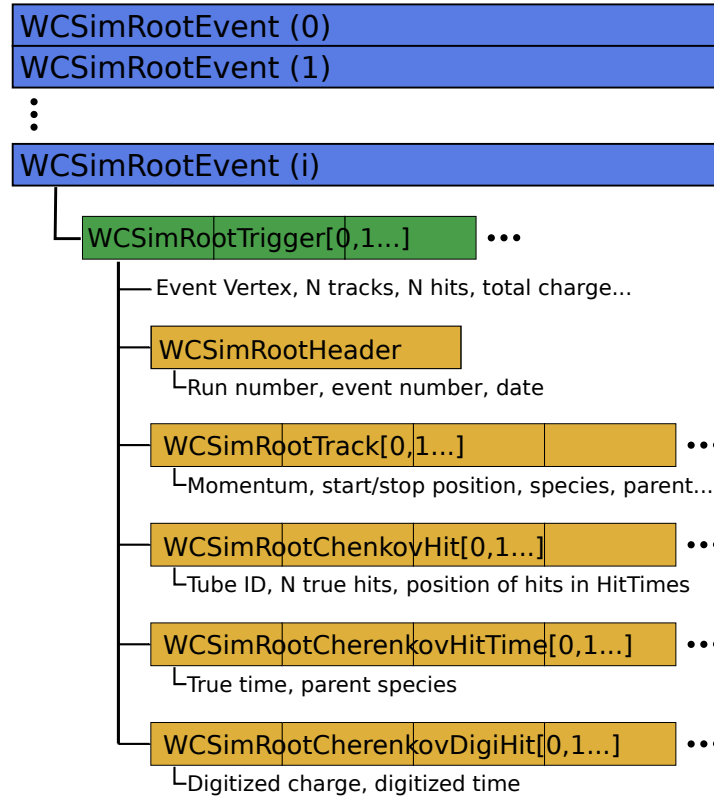
0. Adds dark noise in range `/DarkRate/SetDarkLow` to `/DarkRate/SetDarkHigh`;
1. Adds dark noise in a  $\pm(\text{DarkRate/SetDarkWindow})/2$  window around each hit.

The default mode is 1.

**/DarkRate/SetDarkLow** When using dark mode 0, add dark noise in a window starting at this value.

**/DarkRate/SetDarkHigh** When using dark mode 0, add dark noise in a window ending at this value.

**/DarkRate/SetDarkWindow** When using dark mode 1, add dark noise in a  $\pm(\text{DarkRate/SetDarkWindow})/2$  window around each hit.



**Figure 7:** The class hierarchy of the WCSimRootEvent written to the output file.

## 5 Output Root File

WCSim writes the results of the simulation in a root file. You can set the name and path of this file in the WCSim.mac file using the `/WCSimIO/RootFile` command. To read from the root file, the command `gSystem.Load("<WCSim Directory>/libWCSimRoot.so")` should be run in root to load shared classes. This shared object library is created by running "gmake shared" on the command line.

A typical analysis will loop through events (one per simulated initial vertex), and in each event loop through the observed triggers (usually one for the initial particles and sometimes additional triggers for delayed decay products). In each trigger, you access a list of digitized hits, each containing the charge, time, and ID of the hit PMT. To get the location of the hit, you use the geometry object to access the PMT object that corresponds to that Tube ID. This structure avoids having to store a list containing every PMT in every sub-event.



The files `sample-root-scripts/read_PMT.C` and `sample-root-scripts/testgeo.C` provide general examples for how to read `wcsimT` and `wcsimGeoT`, and are probably the best starting point for custom analysis. There is also some annotated example code in the next subsection.

## 5.1 The Class Hierarchy

The root file itself contains 2 TTrees, each with only 1 branch containing a custom object: `wcsimT` with branch `wcsimrootevent` and `wcsimGeoT` with branch `wcsimrootgeom`. The first has an entry corresponding to each GEANT event and contains the truth and hit data, while the second has only 1 entry which contains the geometry information for the simulated detector. Below is a description of the class hierarchy for these two objects.

**WCSimRootEvent** is a container for the observed triggers. It always contains at least 1 trigger (number 0) which contains the information about the initial particle tracks given to GEANT. If there are delayed decay particles, these “sub-events” are added as additional triggers numbered from 1 onwards.

- `GetTrigger(int i)` - Return trigger number `i`, a `WCSimRootTrigger*`
- `GetNumberOfEvents()` - Total observed triggers
- `GetNumberOfSubEvents()` - Number of sub-event triggers (`GetNumberOfEvents()-1`)
- `HasSubEvents()` - Return true if there is more than 1 trigger

**WCSimRootTrigger** Container for all the information associated with a single trigger

- `GetHeader()` - return the header with run and event numbers, etc.
- `GetTriggerType()` - return the trigger type enumeration
- `GetTriggerInfo()` - return additional trigger information (e.g. the number of digits in the trigger decision window that caused the trigger to trigger)
- `GetPi0Info()` - return `Pi0` information if it was set to be stored in the mac file
- `GetMode()` - interaction mode code number
- `GetVtx(int i)` - event vertex, 0=x, 1=y, 2=z
- `GetNpar()` - number of true particles
- `GetNtrack()` - number of true particle tracks
- `GetTracks()` - TClonesArray of true particle tracks
- `GetNumTubesHit()` - number of tubes with a true hit (quantum efficiency is already applied) (Note: “true” hit means either a photon or dark noise hit)
- `GetNcherenkovhits()` - number of tubes with a true hit (quantum efficiency is already applied)
- `GetCherenkovHits()` - true PMT hits in each PMT (quantum efficiency is already applied)

- `GetNcherenkovhittimes()` - number of true hits (quantum efficiency is already applied)
- `GetCherenkovHitTimes()` - the true times of all the hits (quantum efficiency is already applied)
- `GetNcherenkovdigiHits()` - number of digitized hits
- `GetSumQ()` - sum of digitized charge
- `GetCherenkovDigiHits()` - digitized hits e.g. charge read out by the simulated electronics

**WCSimRootHeader** is a simple container for the run number, event number, and date of the event.

**WCSimRootTrack** is a true track of a particles generated in the simulation. It contains all the information about the track, like particle species, mass, momentum, the start and top volumes, and the parent species. In each trigger the number of tracks is given by `GetNTrack()`.

**WCSimRootCherenkovHit** These hits are records of photons hitting the PMTs before the digitization step (and associated threshold, etc.). In each trigger the number of true Cherenkov hits is given by `GetNcherenkovhits()`.

- `GetTotalPe(0)` - the position in the array of `HitTimes`
- `GetTotalPe(1)` - the number of true photons that hit this PMT, which is also the number of entries in that list that belong to this PMT
- `GetTubeID()` - tube id number

**WCSimRootCherenkovHitTime** This list stores the true time and parent ids of each Cherenkov photon. So, by looking up the photons associated with a particular PMT as above, the particles which contributed light to a particular phototube can be determined.

- `GetTruetime()` - true time of the hit
- `GetParentID()` - ID number of parent, allowing each photon to be traced to a specific true particle

**WCSimRootCherenkovDigiHit** These hits are the final output of the simulation. In each trigger the number of digitized hits is given by `GetNcherenkovdigiHits()`. The charge and time variables are those returned by the simulated electronics.

- `GetQ()` - the total charge measured by the PMT
- `GetT()` - the measured time of the hit
- `GetPhotonIds()` - the position in `WCSimRootCherenkovHitTime` of the raw hits that contribute to this digit. This allows tracing from digit to specific true particle, or noise
- `GetTubeId()` - ID number of the PMT

**WCSimRootGeom** has methods `GetWCCylRadius()`, `GetWCCylLength()`, `GetWCPMTRadius()` and `GetWCNumPMT()`, which return, respectively, the radius and length of the detector

cylinder, the PMT radius, and the total number of PMTs. The class can also return PMT objects by tube number via `GetPMT(i)`.

**WCSimRootPMT** contains information for each PMT in the detector.

- `GetTubeNo()` - the tube ID.
- `GetCylLoc()` - 0 for a PMT on the top cap, 2 for a PMT on the bottom cap, and 1 for a PMT for a wall PMT.
- `GetPosition(j)` - where `j` is 0, 1, or 2. Returns the x, y, and z coordinates of the center of the sphere that forms the PMT.
- `GetOrientation(j)` - where `j` is 0, 1, or 2. Returns the x, y, and z components of the vector describing the direction the PMT faces.

## 5.2 How to Use the Files

There are example scripts showing how to use the root files in sample-root-scripts, but here are the basics of how to get the information out of the root file.

First, you need to load the root library into memory and assign the two WCSim branches:

```
gROOT->Load("libWCSimRoot.so");

TTree *wcsimT = f->Get("wcsimT");
WCSimRootEvent *wcsimrootevent = new WCSimRootEvent();
wcsimT->SetBranchAddress("wcsimrootevent",&wcsimrootevent);

TTree *wcsimGeoT = f->Get("wcsimGeoT");
WCSimRootGeom* wcsimrootgeom = new WCSimRootGeom();
wcsimGeoT->SetBranchAddress("wcsimrootgeom",&wcsimrootgeom);
wcsimrootgeom->GetEntry(0);
```

Since the geometry tree has only one entry, you may as well load it right away. You loop through the events as with any root tree. However, to get at any real information from the events, you will need to load the triggers. The first trigger contains the main event information so here we will only load the first trigger. Then from the trigger we can load and loop through all the digitized hits.

```
wcsimT->GetEntry(ev);
WCSimRootTrigger *wcsimroottrigger = wcsimrootevent->GetTrigger(0);

int ncherenkovdigiHits = wcsimrootevent->GetNcherenkovdigiHits();
// Loop through elements in the TClonesArray
for (int i=0; i<ncherenkovdigiHits; i++) {
    WCSimRootCherenkovDigiHit *hit = (WCSimRootCherenkovDigiHit*)
        (wcsimrootevent->GetCherenkovDigiHits()->At(i));

    double charge = hit->GetQ();
}
```

If you want to know the position of the hit we extracted above, we use the geometry tree to look up that information based on the TubeID.

```
int tubeId = hit->GetTubeId();  
WCSimRootPMT pmt = wcsimrootgeom->GetPMT(tubeId);  
double pmtX = pmt.GetPosition(0);  
double pmtY = pmt.GetPosition(1);  
double pmtZ = pmt.GetPosition(2);
```