AllPix^2 User Manual

Koen Wolters (koen.wolters@cern.ch)
Simon Spannagel (simon.spannagel@cern.ch)

July 6, 2022

Version v0.2

Contents

1. Quick Start

This chapter serves as a very quick introduction to AllPix^2 for users who prefer to start quickly and learn the details while simulating. The typical user should skip the next paragraphs and continue to the next Section 2 instead.

AllPix^2 is a generic simulation framework for tracker and vertex detectors. It provides a modular, flexible and user-friendly framework for the simulation of independent detectors. The framework currently relies on the Geant4 [1], ROOT [2] and Eigen3 [3] libraries, that need to be installed and loaded before using AllPix^2.

The minimal, default installation can be done by executing the commands below. More detailed installation instructions are found in Section 3.

```
$ git clone https://gitlab.cern.ch/simonspa/allpix-squared
$ cd allpix-squared
$ mkdir build && cd build/
$ cmake ..
$ make install
$ cd ..
```

The binary can then be executed with the example configuration file as follows:

```
$ bin/allpix -c etc/example.conf
```

Hereafter, the example configuration can be copied and adjusted to your own needs. This example contains a simple setup of two test detector. It simulates the whole process from the passage of the beam, the deposition of charges in the detectors, the particle propagation and the conversion of the collected charges to digitized pixel hits. All the generated data is finally stored on disk for later analysis.

After this quick start it is very much recommended to read the other sections in more detail as well. For quickly solving common issues the Frequently Asked Questions in Section 8 may be particularly useful.

2. Introduction

AllPix^2 is a generic simulation framework for silicon tracker and vertex detectors written in modern C++. It is the successor of a previously developed simulation framework called AllPix [4, 5]. The goal of the AllPix^2 framework is to provide a complete and easy-to-use package for simulating the performance of detectors from a general source of particles until the digitization of hits in the detector chip.

The framework builds upon other packages to perform tasks in the simulation chain, most notably Geant4 [1] for the deposition of charge carriers in the sensor and ROOT [2] for producing histograms and saving the produced data to storage. The core of the framework focuses on the simulation of charge transport in semiconductor detectors and the digitization to hits in the frontend electronics. The framework does not perform a reconstruction of the particle tracks.

AllPix^2 is designed as a modular framework, allowing for an easy extension to more complex and specialized detector simulations. A modular setup also allows to separate the core of the framework from the implementation of the algorithms in the modules, leading to a framework which is both easier to understand and to maintain. Besides modularity, the AllPix^2 framework was designed with the following main design goals in mind (listed from most to least important):

1. Reflects the physics
    - A run consists of several sequential events. A single event here refers to an independent passage of one or multiple particles through the setup
    - Detectors are treated as separate objects for particles to pass through
    - All of the information must be contained at the very end of processing every single event (sequential events)
2. Ease of use (user-friendly)
    - Simple, intuitive configuration and execution ("does what you expect")
    - Clear and extensive logging and error reporting
    - Implementing a new module should be feasible without knowing all details of the framework
3. Flexibility
    - Event loop runs sequence of modules, allowing for both simple and advanced user configurations
    - Possibility to run multiple different modules on different detectors
    - Limit flexibility for the sake of simplicity and ease of use

2.1. History

Development of AllPix (the original version) started around 2012 as a generic simulation framework for pixel detectors. It has been succesfully used for simulating a variety of different detector setups through the years. Originally written as a Geant4 user application the framework has grown 'organically' after new features continued to be added. Around 2016 discussions between collaborators started to discuss a rewrite of the software from scratch. Primary possibilities for improvements included better modularity, more extensive configuration options and an easier geometry setup.

Early development of AllPix^2 started in end of 2016, but most of the initial rework in modern C++ has been carried out in the framework of a technical student project in the beginning of 2017. The core of the framework starts to mature and initial versions of various generic core modules have been created at the time of writing.

2.2. Scope of this manual

This document is the primary User's Guide for AllPix^2. It presents all of the required to start using the framework. In more detail this manual is designed to:

- guide all new users through the installation
- introduce new users to the toolkit for the purpose of running their own simulations
- explain the structure of the core framework and the components it provides to the modules
- provide detailed information about all modules and how-to use and configure them
- describe the required steps for adding a new detector model and implementing a new module

In the manual an overview of the framework is given, more detailed information on the code itself can be found in the Doxygen reference manual. The reader does not need any programming experience to get started, but knowledge of (modern) C++ will be useful in the later chapters.

2.3. Support and reporting issues

We are happy to receive feedback on any problem that might arise. Reports for issues, questions about unclear parts, as well as suggestions for improvements, are very much appreciated. These should preferably be brought up on the issues page of the repository, which can be found at `https://gitlab.cern.ch/simonspa/allpix-squared/issues`.

3. Installation

After installing and loading the required dependencies, there are various options to customize the installation of AllPix^2. This chapter contains details on the standard installation process and information about custom installations.

3.1. Prerequisites

AllPix^2 should be able to run without problems on Mac as well as any recent Linux distribution. Windows is not officially supported and will likely never be. It could however be theoretically possible to install AllPix^2 using MinGW or Cygwin, but this has not been tested.

The core framework is separated from the individual modules and AllPix^2 has therefore only one required dependency: ROOT 6 (versions below 6 are not supported!) [2]. If the framework is run on a CERN cluster the default dependencies can be loaded from CVMFS as explained in Section 3.3. Otherwise all required dependencies need to be installed before building AllPix^2. Please refer to [6] for instructions on how to install ROOT. ROOT has several extra components and the GenVector package is required to run AllPix^2. This package is included in the default build.

For various modules additional dependencies are necessary. For details about the dependencies and their installation visit the module documentation in Section 6. The following dependencies are needed to compile the standard installation:

- Geant4 [1]: Used to simulate the geometry and deposit charges in the detector. See the instructions in [7] for details on how to install the software. All the Geant4 datasets are required to run the modules succesfully. Also GDML support could be enabled to save the Geant4 geometry for later review. Finally it is very much recommended to enable Qt visualization. A good set of CMake build flags to start with is the following:

  ```
  -DGEANT4_INSTALL_DATA=ON
  -DGEANT4_BUILD_MULTITHREADED=ON
  -DGEANT4_USE_GDML=ON
  -DGEANT4_USE_QT=ON
  -DGEANT4_USE_XM=ON
  -DGEANT4_USE_OPENGL_X11=ON
  -DGEANT4_USE_SYSTEM_CLHEP=OFF
  ```

- Eigen3 [3]: Used vector package to do Runge-Kutta integration in the generic charge propagation module. Eigen is available in almost all Linux distributions through the package manager. Otherwise it can be easily installed, because it is a header-only library.

Extra flags needs to be set for building an AllPix^2 installation without these dependencies. Details about these configuration options are given in Section 3.4.

3.2. Downloading the source code

The latest version of AllPix^2 can be fetched from the Gitlab repository at `https://gitlab.cern.ch/simonspa/allpix-squared`. This version is under heavy development, but should work out-of-the-box. The software can be cloned and accessed as follows:

```
$ git clone https://gitlab.cern.ch/simonspa/allpix-squared
$ cd allpix-squared
```

3.3. Initializing the dependencies

Before continuing with the build, the necessary setup scripts for ROOT and Geant4 (unless a build without Geant4 modules is attempted) should be run. In Bash on Linux this means executing the following two commands from the respective installation directories (replacing *<root_install_dir>* with the local ROOT installation directory and similar for Geant):

```
$ source <root_install_dir>/bin/thisroot.sh
$ source <geant4_install_dir>/bin/geant4.sh
```

On the CERN LXPLUS service a standard initialization script is available to load all dependencies from the CVMFS infrastructure. This script should be run as follows (from the main repository directory):

```
$ source etc/scripts/setup_lxplus.sh
```

3.4. Configuration via CMake

AllPix^2 uses the CMake build system to build and install the core framework and the modules. An out-of-source build is recommended: this means CMake should not be directly executed in the source folder. Instead a *build* folder should be created inside the source folder from which CMake should be run. For a standard build without any flags this implies executing:

```
$ mkdir build
$ cd build
$ cmake ..
```

CMake can be run with several extra arguments to change the type of installation. These options can be set with -D*option* (see the end of this section for an example). Currently the following options are supported:

- **CMAKE_INSTALL_PREFIX**: The directory to use as a prefix for installing the binaries, libraries and data. Defaults to the source directory (where the folders *bin/* and *lib/* are added).
- **CMAKE_BUILD_TYPE**: Type of build to install, defaults to `RelWithDebInfo` (compiles with optimizations and debug symbols). Other possible options are `Debug` (for compiling with no optimizations, but with debug symbols and extended tracing using the Clang Address Sanitizer library) and `Release` (for compiling with full optimizations and no debug symbols).
- **MODEL_DIRECTORY**: Directory to install the internal models to. Defaults to not installing if the **CMAKE_INSTALL_PREFIX** is set to the directory containing the sources (the default). Otherwise the default value is equal to the directory **CMAKE_INSTALL_PREFIX***/share/allpix/*. The install directory is automatically added to the model search path used by the geometry model parsers to find all the detector models.
- **BUILD_*module_name***: If the specific *module_name* should be installed or not. Defaults to ON, thus all modules are installed by default. This set of parameters have to be set appropriately for a build without extra dependencies as specified in 3.1.
- **BUILD_ALL_MODULES**: Build all included modules, defaulting to OFF. This overwrites any selection using the parameters described above.

An example of a custom installation with debugging, without the GeometryBuilderGeant4 module and installed to a custom directory, is shown below:

```
$ mkdir build
$ cd build
$ cmake -DCMAKE_INSTALL_PREFIX=../install/ \
        -DCMAKE_BUILD_TYPE=DEBUG \
        -DBUILD_GeometryBuilderGeant4=OFF ..
```

3.5. Compilation and installation

Compiling the framework is now a single command in the build folder created earlier (replacing *<number_of_cores>* with the number of cores to use for compilation):

```
$ make -j<number_of_cores>
```

The compiled (non-installed) version of the executable can be found at *src/exec/allpix* in the build folder. Running AllPix^2 directly without installing can be useful for developers, but is not recommended for normal users.

To install the library to the selected install location (defaulting to the source directory) requires the following command:

```
$ make install
```

The binary is now available as *bin/allpix* in the installation directory. The example configuration files are not installed as they should only be used as a starting point for your own configuration. They can however be used to check if the installation was succesful. Running the allpix binary with the example configuration (like `bin/allpix -c etc/example.conf`) should run without problems when a standard installation is used.

4. Getting Started

After finishing the installation the first simulations can be runned. This Getting Started guide is written with a default installation in mind, meaning that some parts may not work if a custom installation was used. When the *allpix* binary is used, this refers to the executable installed in bin/allpix in your installation path. Remember that before running any AllPix^2 simulation, ROOT and likely Geant4 should be initialized. Refer to Section 3.3 on instructions how to load those libraries.

4.1. Configuration Files

The framework has to be configured with simple human-readable configuration files. The configuration format is described in detail in Section 5.2.1. The configuration consists of several section headers within [and] brackets and a section without header at the start. Every section contain a set of key/value pairs separated by the = character. The # is used to indicate comments.

The framework has the following three required layers of configuration files:

- The **main** configuration: The most important configuration file and the file that is passed directly to the binary. Contains both the global framework configuration and the list of modules to instantiate together with their configuration. More details and an example are found in Section 4.1.3.
- The **detector** configuration passed to the framework to determine the geometry. Describes the detector setup, containing the position, orientation and model type of all detectors. Introduced in Section 4.1.2.
- The detector **models** configuration. Contain the parameters describing a particular type of detector. Several models are already shipped by the framework, but new types of detectors can be easily added. Please refer to Section 7.2 for more details about adding new models.

Before going into depth on defining the required configuration files, first more detailed information about the configuration values is provided in the next paragraphs.

4.1.1. Supported types and units

The AllPix^2 framework supports the use of a variety of types for all configuration values. The module specifies how the value type should be interpreted. An error will be raised if either the key is not specified in the configuration file, the conversion to the desired type is not possible, or if the given value is outside the domain of possible options. Please refer to the module documentation in Section 6 for the list of module parameters and their types.

Parsing the value roughly follows common-sense (more details can be found in Section 5.2.2). A few special rules do apply:

- If the value is a **string** it may be enclosed by a single pair of double quotation marks (`"`), which are stripped before passing the value to the modules. If the string is not enclosed by the quotation marks all whitespace before and after the value is erased. If the value is an array of strings, the value is split at every whitespace or comma (`,`) that is not enclosed in quotation marks.
- If the value is a **relative path** that path will be made absolute by adding the absolute path of the directory that contains the configuration file where the key is defined.
- If the value is an **arithmetic** type, it may have a suffix indicating the unit. The list of base units is shown in Table 1.

Table 1: List of units supported by AllPix^2

Quantity	**Default unit**	**Auxiliary units**
Length	mm (millimeter)	nm (nanometer)
		um (micrometer)
		cm (centimeter)
		dm (decimeter)
		m (meter)
		km (kilometer)
Time	ns (nanosecond)	ps (picosecond)
		us (microsecond)
		ms (millisecond)
		s (second)
Energy	MeV (megaelectronvolt)	eV (electronvolt)
		keV (kiloelectronvolt)
		GeV (gigaelectronvolt)
Temperature	K (kelvin)	
Charge	e (elementary charge)	C (coulomb)
Voltage	MV (megavolt)	V (volt)
		kV (kilovolt)
Angle	rad (radian)	deg (degree)

Combinations of base units can be specified by using the multiplication sign `*` and the division sign `/` that are parsed in linear order (thus $\frac{Vm}{s^2}$ should be specified as $V * m/s/s$). The framework assumes the default units (as given in Table 1) if the unit is not explicitly specified. It is recommended to always specify the unit explicitly for all parameters that are not dimensionless as well as for angles.

Examples of specifying key/values pairs of various types are given below

```
# All whitespace at the front and back is removed
first_string =   string_without_quotation
# All whitespace within the quotation marks is kept
second_string = "  string with quotation marks  "
# Keys are split on whitespace and commas
string_array = "first element" "second element","third element"
# Integer and floats can be given in standard formats
int_value = 42
float_value = 123.456e9
# Units can be passed to arithmetic type
energy_value = 1.23MeV
time_value = 42ns
# Units are combined in linear order
acceleration_value = 1.0m/s/s
# Thus the quantity below is the same as 1.0deg*kV*K/m/s
random_quantity = 1.0deg*kV/m/s*K
# Relative paths are expanded to absolute
# Path below will be /home/user/test if the config file is in /home/user
output_path = "test"
```

4.1.2. Detector configuration

The detector configuration consist of a set of section headers describing the detectors in the setup. The section header describes the names used to identify the detectors. All names should be unique, thus using the same name multiple times is not possible. Every detector should contain all of the following parameters:

- A string referring to the **type** of the detector model. The model should exist in the search path described in Section 5.4.2.
- The 3D **position** in the world frame in the order x, y, z. See Section 5.4 for details.
- The **orientation** specified as Z-X-Z Euler angle relative to the world axis. See Section 5.4 for details.

An example configuration file of one test detector and two Timepix [8] models is given below.

```
# name the first detector ‘telescope1‘
[telescope1]
# set the type to the test detector
type = "test"
# place it at the origin of the world frame
```

```
position = 0 0 0mm
# use the default orientation
orientation = 0 0 0

# name the second detector ‘telescope2‘
[telescope2]
# set the type again to Timepix
type = "timepix"
# place it 0.5 mm down on the z-axis from the origin
position = 0 0 -50mm
# use the default orientation
orientation = 0 0 0

# name the third detector ‘dut‘ (device under test)
[dut]
# set the type again to Timepix
type = "timepix"
# set the position in the world frame
position = 100um 100um -10mm
# rotate 20 degrees around the world x-axis
orientation = 0 20deg 0
```

This configuration file is used in the rest of this chapter for explaining concepts.

4.1.3. Main configuration

The main configuration consists of a set of section header that specify the modules used. All modules are executed in the linear order in which they are defined. There are a few section names that have a special meaning in the main configuration, which are the following:

- The **global** (framework) header sections: These are all the zero-length section headers (including the one at the start) and all with the header `AllPix` (case-sensitive). These are combined and accessed together as the global configuration, which contain all the parameters of the framework (see Section 4.2 for details). All key-value pairs defined in this section are also inherited by all individual configurations as long the key is not defined in the module configuration itself.
- The **ignore** header sections: All sections with name `Ignore` are ignored. Key-value pairs defined in the section as well as the section itself are redundant. These sections are useful for quickly enabling and disabling for debugging.

All other section headers are used to instantiate the modules. Installed modules are loaded automatically. If problems arise please review the loading rules described in Section 5.3.3.

Modules can be specified multiple times in the configuration files, but it depends on their type and configuration if this allowed. The type of the module determines how the module is instantiated:

- If the module is **unique**, it is instantiated only a single time irrespective of the amount of detectors. These kind of modules should only appear once in the whole configuration file unless a different inputs and outputs are used as explained in Section 4.3.2.
- If the module is **detector**-specific, it is run on every detector it is configured to run on. By default an instantiation is created for all detectors defined in the detector configuration file (see Section 4.1.2) unless one or both of the following parameters are specified.
  - **name**: An array of detector names where the module should run on. Replaces all global and type-specific modules of the same kind.
  - **type**: An array of detector type where the module should run on. Instantiated after considering all detectors specified by the name parameter above. Replaces all global modules of the same kind.

A valid example configuration using the detector configuration above could be:

```
# Key is part of the empty section and therefore the global sections
string_value = "example1"
# The location of the detector configuration should be a global parameter
detectors_file = "manual_detector.conf"
# The AllPix section is also considered global and merged with the above
[AllPix]
another_random_string = "example2"

# First run a unique module
[MyUniqueModule]
# This module takes no parameters
# [MyUniqueModule] cannot be instantiated another time

# Then run some detector modules on different detectors
# First run a module on the detector of type Timepix
[MyDetectorModule]
type = "timepix"
int_value = 1
# Replace the module above for ‘dut‘ with a specialized version
#   this does not inherit any parameters from earlier modules
[MyDetectorModule]
name = "dut"
```

```
23 int_value = 2
24 # Runs the module on the remaining unspecified detector 'telescope1'
25 [MyDetectorModule]
26 # int_value is not specified, so it uses the default value
```

This configuration can however not be executed in practice because MyUniqueModule and MyDetectorModule do not exist. In the next paragraphs an useful configuration file with valid configuration is presented. Before turning to the module parameters the global framework parameters are introduced first.

4.2. Framework parameters

The framework has a variety of global parameters that allow to configure AllPix^2 for different kind of simulations:

- **detectors_file**: Location of the file describing the detector configuration (introduced in Section 4.1.2). The only required global parameter: the framework will fail if it is not specified.
- **number_of_events**: Determines the total number of events the framework should simulate. Equivalent to the amount of times the modules are run. Defaults to one (simulating a single event).
- **log_level**: Specifies the minimum log level which should be written. Possible values include `FATAL`, `STATUS`, `ERROR`, `WARNING`, `INFO` and `DEBUG`, where all options are case-insensitive. Defaults to the `INFO` level. More details and information about the log levels and changing it for a particular module can be found in Section 4.4. Can be overwritten by the `-v` parameter on the command line.
- **log_format**: Determines the format to display. Possible options include `SHORT`, `DEFAULT` and `LONG`, where all options are case-insensitive. More information again in Section 4.4.
- **log_file**: File where output should be written to besides standard output (usually the terminal). Only writes to standard output used if option is not given. Another (additional) location to write to can be specified on the command line using the `-l` parameter.
- **output_directory**: Directory to write all output files into. Extra directories are created for all the module instantiations. This directory also contains the **root_file** parameter described below. Defaults to the current working directory with the subdirectory *output/* attached.

- **root_file**: Location relative to the **output_directory**, where the ROOT output data of all modules will be written to. Default value is *modules.root*. The directories will be created automatically for all the module instantiations in this ROOT file.
- **random_seed**: Seed to use for the random seed generator (see Section 5.6.3). A random seed from multiple entropy sources will be generated if the parameter is not specified. Can be used to reproduce an earlier simulation run.
- **library_directories**: Additional directories to search for libraries, before searching the default paths. See Section 5.3.3 for details.
- **model_path**: Additional files or directories from which detector models should be read besides the standard search locations. Refer to Section 5.4.2 for more information.

With this information in mind it is time to setup a real simulation. Module parameters are shortly introduced when they are first used. For more details about these parameters the module documentation in Section 6 should be consulted.

4.3. Setting up the Simulation Chain

Below a simple, but complete simulation is described. A typical simulation in AllPix^2 contains at least the following components.

- A **model reader**, which reads the model paths and creates the detector models it can parse (should be all models specified in the detector configuration)
- The **geometry builder**, responsible for creating the external geometry (Geant4) from the internal geometry.
- The **deposition** module that deposits charge carriers in the detectors using the provided physics list and the geometry created above.
- A **propagation** module that propagates the charges through the sensor and assigns them to a pixel.
- A **digitizer** module which converts the charges in the pixel to a detector hit, simulating the frontend electronics response.
- An **output** module, saving the data of the simulation to the requested format.

In the example charges will be deposited in the three sensors from the detector configuration file in Section 4.1.2. Only the charges in the Timepix models are going to be propagated and digitized. The final results of hits in the device under test (dut) will be written to a ROOT histogram. A configuration file that implements this description is as follows:

```
# Initialize the global configuration
[AllPix]
# Run a total of 10 events
number_of_events = 10
# Use the short logging format
log_format = "SHORT"
# Location of the detector configuration
detectors_file = "manual_detector.conf"

# Read the default models
[DefaultModelReader]

# Read and instantiate the detectors and construct the Geant4 geometry
[GeometryBuilderGeant4]
# Size of the world
# TODO: this will be optional
world_size = 50mm 50mm 100mm

# initialize physics list, setup the particle source and deposit the
↪  charges
[DepositionGeant4]
# Use one of the standard Geant4 physics lists
physics_list = QGSP_BERT
# Use a charged pion as particle
particle_type = "pi+"
# Use a single particle in a single 'event'
particle_amount = 1
# Set the energy of the particle
particle_energy = 120GeV
# The position of the point source
particle_position = 0 0 1mm
# The direction of the source
particle_direction = 0 0 -1

# Specify a linear electric field for all detectors
# NOTE: This will be explained in more detail later in the manual
[ElectricFieldInputLinear]
# Voltage to calculate the electric field from
voltage = 50V

# Propagate the charges in the sensor
[SimplePropagation]
```

```
# Only propagate charges in the Timepix sensors
type = "timepix"
# Set the temperature of the sensor
temperature = 293K
# Propagate multiple charges together in one step for faster simulation
charge_per_step = 50

# Transfer the propagated charges to the pixels
# TODO: this module is going to be deleted
[SimpleTransfer]
max_depth_distance = 10um

# Digitize the propagated charges
[DefaultDigitizer]
# Noise added by the electronics
electronics_noise = 110e
# Threshold for a hit to be detected
threshold = 600e
# Noise of the threshold level
threshold_smearing = 30e
# Uncertainty added by the digitization
adc_smearing = 300e

# Save histogram to an output file
[DetectorHistogrammer]
# Save the histogram only for the dut
name = "dut"
```

The configuration above is available as *etc/manual.conf*. The detector configuration file in Section 4.1.2 can be found in *etc/manual_detector.conf*. The total simulation can then be executed by passing the configuration to the allpix binary as follows:

```
$ allpix -c etc/manual.conf
```

The simulation should then start. It should output similar output as the example found in Appendix A. The final histogram of the hits will then be availabe in the ROOT file *output/modules.root* as the local file *DetectorHistogrammer/histogram*.

If problems occur, please make sure you have an up-to-date and properly installed version of AllPix[2] (see the installation instructions in Section 3). If modules and models fail to load, more information about loading problems can be found in the detailed framework description in Section 5.

4.3.1. Adding new modules

Before going to more advanced configurations, a few simple modules are discussed which a user might want to add.

Visualization Displaying the geometry and the particle tracks helps a lot in both checking and interpreting the results. Visualization is fully supported through Geant4, supporting all the options provided by Geant4 [9]. Using the Qt viewer with the OpenGL driver is however the recommended option as long as the installed version of Geant4 supports it.

To start using the Qt viewer the `simple_view = 1` parameter should first be added to the **GeometryBuilderGeant4** section. This simplifies the visualization output and makes it several order of magnitude faster. Finally the example section below should be added at the end of the configuration file before running the simulation again:

```
[VisualizationGeant4]
# Setup to use the Qt gui
use_gui = 1
# Use the OpenGL driver
driver = "OGL"
# Point those macro files to the etc/ directory in the allpix source folder
# TODO: these macros should be optional
macro_init = "geant4_init.in"
macro_run = "geant4_run.in"
```

If it gives an error about Qt not being available the VRML viewer can be used as a replacement, but it is recommended to reinstall Geant4 with the Qt viewer included. To use the VRML viewer instead, follow the steps below:

- The `simple_view = 1` parameter should be added to the **GeometryBuilderGeant4** section.
- Then the viewer should be installed on your operating system. Good options are for example FreeWRL and OpenVRML.
- Subsequently two environmental parameters should be exported to inform Geant4 of the configuration. These include `G4VRMLFILE_VIEWER` which should point to the location of the viewer and `G4VRMLFILE_MAX_FILE_NUM` which should typically be set to 1 to prevent too many files from being created.
- Finally the example section below should be added at the end of the configuration file before running the simulation again:

```
[VisualizationGeant4]
# Use the VRML driver
driver = "VRML2FILE"
# Point those macro files to the etc/ directory in the allpix source
 ↪  folder
# TODO: these macros should be optional
macro_init = "geant4_init.in"
macro_run = "geant4_run.in"
```

Linear Electric Field The example configuration before already contained a module for adding a linear electric field to the sensitive detector. All detectors by default do not have any electric field. This will make the `SimplePropagation` module slow, because it will wait for the propagated charges to reach the end of the sensor, which can take a long time with diffusion solely. Therefore a simple linear electric field have been added to the sensors. The section below sets the electric field on every point in the pixel grid to the voltage divided by the thickness of the sensor.

```
# Add a linear electric field
[ElectricFieldInputLinear]
# Bias voltage used to create the linear electric field
voltage = 50V
```

More complex electric fields from TCAD can also be added as explained in more detail in Section 4.3.2.

4.3.2. Advanced configuration

Redirect Module Inputs and Outputs By default it is not allowed to have the same type of module (linked to the same detector), but in several cases it may be useful to run the same module with different settings. The AllPix^2 framework support this by allowing to redirect the input and output data of every module. A module sends it output by default to all modules listening to the type of object it dispatches. It is however possible to specify a certain name in addition to the type of the data.

The output name of a module can be changed by setting the **output** parameter of the module to a unique value. The output of this module is than not send anymore to modules without a configured input, because the default input listens only to data without a name. The **input** parameter of a particular receiving module should therefore be set to match the value of the **output** parameter. In addition it is allowed to set the **input** parameter to the special value $*$ to indicate that it should listen to all messages irrespective of their name.

An example of a configuration with two settings for digitization is shown below:

```
# Digitize the propagated charges with low noise levels
[DefaultDigitizer]
# Specify an output identifier
output = "low_noise"
# Low amount of noise added by the electronics
electronics_noise = 100e
# Default values are used for the other parameters

# Digitize the propagated charges with high noise levels
[DefaultDigitizer]
# Specify an output identifier
output = "high_noise"
# High amount of noise added by the electronics
electronics_noise = 500e
# Default values are used for the other parameters

# Save histogram for 'low_noise' digitized charges
[DetectorHistogrammer]
# Specify input identifier
input = "low_noise"

# Save histogram for 'high_noise' digitized charges
[DetectorHistogrammer]
# Specify input identifier
input = "high_noise"
```

Using TCAD Electric Field Simulations An electric field in the detectors can be specified using the .init format by using the ElectricFieldReaderInit module. The init format is format used by the PixelAV software [10, 11] after conversions from internal TCAD formats. These fields can be attached to specific detectors using the standard syntax for detector binding. An example would be:

```
[ElectricFieldReaderInit]
# Bind the electric field to the timepix sensor
name = "tpx"
# Name of the file containing the electric field
file_name = "example_electric_field.init"
```

An example electric field (which the name used above) can be found in the *etc* directory of the AllPix^2 repository.

Choosing the Propagation Modules *This section is not written yet.*

4.4. Logging and Verbosity Levels

AllPix^2 is designed to identify mistakes and implementation errors as early as possible and tries to give the user a clear indication about the problem. The amount of feedback can be controlled using different log levels. The global log level can be set using the global parameter **log_level**. The log level can be overridden for a specific module by adding the **log_level** parameter to that module. The following log levels are currently supported:

- **FATAL**: Indicates a fatal error that should and will lead to direct termination of the application. Typically only emitted in the main executable after catching exceptions, because exceptions are the preferred way of error handling as discussed in Section 5.7. An example of a fatal error is an invalid configuration parameter.
- **STATUS**: Important informational messages about the status of the simulation. Should only be used for informational messages that have to be logged in every run (unless the user wants to only fetch fatal errors)
- **ERROR**: Severe error that should never happen during a normal well-configured simulation run. Frequently leads to a fatal error and can be used to provide extra information that may help in finding the reason of the problem. For example used to indicate the reason a dynamic library cannot be loaded.
- **WARNING**: Indicate conditions that should not happen normally and possibly lead to unexpected results. The framework can however typically continue without problems after a warning. Can for example indicate that a output message is not used and that a module may therefore do unnecessary work.
- **INFO**: Informatic messages about the physics process of the simulation. Contains summaries about the simulation details of every event and for the overall simulation. Should typically produce maximum one line of output per event.
- **DEBUG**: In-depth details about the progress of the framework and all the physical details of the simulation. Produces large volumes of output per event usually and this level is therefore normally only used for debugging the physics simulation of the modules.
- **TRACE**: Messages to trace what the framework or a module is currently doing. Does not contain any direct information unlike the **DEBUG** level above, but only indicates which part of the module or framework is currently running. Mostly used for software debugging or determining the speed bottleneck in simulations.

It is not recommended to set the **log_level** higher than **WARNING** in a typical simulation as important messages could be missed.

The logging system does also support a few different formats to display the log messages. The following formats are supported for the global parameter **log_format** and for the module parameter with the same name that overwrites it:

- **SHORT**: Displays the data in a short form. Includes only the first character of the log level followed by the section header and the message.
- **DEFAULT**: The default format. Displays the date, log level, section header and the message itself.
- **LONG**: Detailed logging format. Besides the information above, it also shows the file and line where the log message was produced. This can help in debugging modules.

More details about the logging system and the procedure for reporting errors in the code can be found in Section 5.6.1 and 5.7.

4.5. Storing Output Data

Saving the output to persistent storage is of primary importance for later review and analysis. AllPix^2 primarily uses ROOT for storing output data, because it supports writing arbitrary objects and is a standard tool in High-Energy Physics. The `ROOTObjectWriter` automatically saves all the data objects written by the modules to a TTree [12] for more information about TTrees). The module stores all different object types to a separate tree, creating a branch for every combination of detector and the name given to the output as explained in Section 4.3.2. For each event, values are added to the leafs of the branches containing the data of the objects. This allows for easy histogramming of the acquired data over the total run using the ROOT utilities.

To save the output of all objects an `ROOTObjectWriter` has to be added with a `file_name` parameter (without the root suffix) to specify the file location of the created ROOT file in the global output directory. The default file name is `data` which means that **data.root** is created in the output directory. To replicate the default behaviour the following configuration can be used:

```
# The object writer listens to all output data
[ROOTObjectWriter]
# specify the output path (can be omitted as it the default)
file_name = "data.root"
```

Besides using the generated tree for analysis it is also possible to read all the object data back in, to propagate it to further modules. This can be used to split the execution of several parts of the simulation in multiple independent steps, which can be executed after each

order. The tree data can be read using a `ROOTObjectReader` module, that automatically dispatches all objects to the right detector with the correct name in the same event, using the internal name of the stored data. An example of using this module is the following:

```
# The object reader dispatches all objects in the tree
[ROOTObjectReader]
# path to the output data relative to the configuration file
file_name = "../output/data.root"
```

5. The AllPix^2 Framework

The framework is split up in the following four main components that together form AllPix^2:

1. **Core**: The core contains the internal logic to initiate the modules, provide the geometry, facilitate module communication and run the event sequence. The core keeps its dependencies to a minimum (it only relies on ROOT) and remains separated from the other components as far as possibble. It is the main component discussed in this section.
2. **Modules**: A set of methods that execute a part of the simulation chain. These are build as separate libraries, loaded dynamically by the core. The available modules and their parameters are discussed in more detail in Section 6.
3. **Objects**: Objects are the data passed around between modules using the message framework provided by the core. Modules can listen and bind to messages with objects they wish to receive. Messages are identified by the object type they are carrying, but they can also be named to allow redirecting data to specific modules facilitating more sophisticated simulations. Messages are meant to be read-only and a copy of the data should be made if a module wishes to change the data. All objects are contained into a separate library, automatically linked to every module. More information about the messaging system and the supported objects can be found in Section 5.5.
4. **Tools**: AllPix^2 provides a set of header-only 'tools' that provide access to common logic shared by various modules. An example is a Eigen Runge-Kutta solver and a set of template specializations for ROOT and Geant4 configuration. More information about these can be found in Section 9. This set of tools is different from the set of core utilities the framework provides by itself, which are part of the core and explained in 5.6

Finally AllPix^2 provides an executable which instantiates the core, passes the configuration and runs the simulation chain.

In this chapter, first an overview of the architectural setup of the core is given and how it interacts with the total AllPix^2 framework. Afterwards, the different subcomponents are discussed and explained in more detail. Some C++ code will be provided in the text, but readers not interested may skip the technical details.

5.1. Architecture of the Core

The core is constructed as a light-weight framework that provides various subsystems to the modules. It also contains the part responsible for instantiating and running the modules from the supplied configuration file. The core is structured around five subsystems of which

four are centered around managers and the fifth contain a set of simple general utilities. The systems provided are:

1. **Configuration**: Provides a general configuration object from which data can be retrieved or stored, together with a TOML-like [13] file parser to instantiate the configurations. Also provides a general AllPix^2 configuration manager providing access to the main configuration file and its sections. It is used by the module manager system to find the required instantiations and access the global configuration. More information is given in Section 5.2.
2. **Module**: Contain the base class of all the AllPix^2 modules and the manager responsible for loading and running the modules (using the configuration system). This component is discussed in more detail in Section 5.3.
3. **Geometry**: Supplies helpers for the simulation geometry. The manager contains all registered detectors. A detector has a certain position and orientation linked to an instantiation of a particular detector model. The detector model contains all parameters describing the geometry of the detector. More details about the geometry and detector models is provided in Section 5.4.
4. **Messenger**: The messenger is responsible for sending objects from one module to another. The messenger object is passed to every module and can be used to bind to messages to listen for. Messages with objects are also dispatched through the messenger to send data to the modules listening. Please refer to Section 5.5 for more details.
5. **Utilities**: The framework provides a set of simple utilities for logging, file and directory access, random number seeding and unit conversion. An explanation how to use of these utilities can be found in Section 5.6. A set of C++ exceptions is also provided in the utilities, which are inherited and extended by the other components. Proper use of exceptions, together with logging informational messages and reporting errors, make the framework easier to use and debug. A few notes about the use and structure of exceptions are given in Section 5.7.

5.2. Configuration and Parameters

Modules and the framework are configured through configuration files. An explanation how to use the various configuration files together with several examples are provided in Section 4.1. All configuration files follow the same format, but the way their input is interpreted differs per configuration file.

5.2.1. File format

Throughout the framework a standard format is used for the configuration files, a simplified version of TOML [13]. The rules for this format are as follows:

1. All whitespace at the beginning or end of a line should be stripped by the parser. Empty lines should be ignored.
2. Every non-empty line should start with either `#`, `[` or an alphanumeric character. Every other character should lead to an immediate parse error.
3. If the line starts with `#`, it is interpreted as comment and all other content on the same line is ignored
4. If the line starts with `[`, the line indicates a section header (also known as configuration header). The line should contain an alphanumeric string indicating the header name followed by `]` to end the header (a missing `]` should raise an exception). Multiple section header with the same name are allowed. All key-value pairs following this section header are part of this section until a new section header is started. After any number of ignored whitespace characters there may be a `#` character. If that is the case, the rest of the line is handled as specified in point 3.
5. If the line starts with an alphanumeric character, the line should indicate a key-value pair. The beginning of the line should contain an string of alphabetic characters, numbers and underscores, but note that it may not start with an underscore). This string indicates the 'key'. After a optional number of ignored whitespace, the key should be followed by an =. Any text between the = and the first `#` character not enclosed within a pair of `"` characters is known as the non-stripped 'value'. Any character from the `#` is handled as specified in point 3. If the line does not contain any non-enclosed `#` character the value ends at the end of the line instead. The 'value' of the key-value pair is the non-stripped 'value' with all whitespace in front and the end stripped.
6. The value can either be accessed as a single value or an array. If the value is accessed as an array, the string is split at every whitespace or `,` character not enclosed in a pair of `"` characters. All empty entities are not considered. All other entities are treated as single values in the array.
7. All single values are stored as a string containing at least one character. The conversion to the actual type is executed when accessing the value.
8. All key-value pairs defined before the first section header are part of a zero-length empty section header

5.2.2. Accessing parameters

All values are accessed via the configuration object. In the following example the key is a string called **key**, the object is named **config** and the type **TYPE** is a valid C++ type that the value should represent. The values can be accessed via the following methods:

```
// Returns true if the key exists and false otherwise
config.has("key")
// Returns the value in the given type, throws an exception if not
 ↪  existing
config.get<TYPE>("key")
// Returns the value in the given type or the provided default value if
 ↪  it does not exist
config.get<TYPE>("key", default_value)
// Returns an array of single values of the given type; throws if the key
 ↪  does not exist
config.getArray<TYPE>("key")
// Returns an absolute (canonical if it should exist) path to a file
config.getPath("key", true /* check if path exists */)
// Return an array of absolute paths
config.getPathArray("key", false /* check if paths exists */)
// Returns the key as literal text including possible quotation marks
config.getText("key")
// Set the value of key to the default value if the key is not defined
config.setDefault("key", default_value)
// Set the value of the key to the defaults array if key is not defined
config.setDefaultArray<TYPE>("key", vector_of_default_values)
```

The conversions to the type are using the `from_string` and `to_string` methods provided by the string utility library described in Section 5.6.3. These conversions largely follows the standard C++ parsing, with one important exception. If (and only if) the value is retrieved as any C/C++ string type and the string is fully enclosed by a pair of `"` characters, they are stripped before returning the value (and strings can thus also be given without quotation marks).

5.3. Modules and the Module Manager

AllPix^2 is a modular framework, the core idea is to separate functionality in various independent modules. The modules are defined in the subdirectory *src/modules/* in the repository. The name of the directory is the unique name of the module. The suggested naming scheme is CamelCase, thus an example module would be *GenericPropagation*. There are two different kind of modules which can be defined:

- **Unique**: Modules for which always a single instance runs irrespective of the number of detectors.
- **Detector**: Modules that are specific to a single detector. They are replicated for all required detectors.

The type of module determines the kind of constructor used, the internal unique name and the supported configuration parameters. More details about the instantiation logic for the different kind of modules can be found in 5.3.3.

5.3.1. Files of a Module

Every module directory should at the minimum contain the following documents (with `ModuleName` replaced by the name of the module):

- **CMakeLists.txt**: The build script to load the dependencies and define the source files
- **README.md**: Short documentation of the module
- `ModuleName`**.tex**: Full documentation of the module for this Users Manual
- `ModuleName`**Module.hpp**: The header file of the module (note that another name can be used for this source file, but that is deprecated)
- `ModuleName`**Module.cpp**: The implementation file of the module

The files are discussed in more detail below. All modules that are added to the *src/modules/* directory will be build automatically by CMake. This also means that all subdirectories in this module directory should contain a module with a *CMakeLists.txt* to build the module.

More information about constructing new modules can be found in Section 7.1.

CMakeLists.txt Contains the build description of the module with the following components:

1. On the first line either ALLPIX_DETECTOR_MODULE(MODULE_NAME) or ALLPIX_UNIQUE_MODULE(MODULE_NAME) depending on the type of the module defined. The internal name of the module is saved to the ${MODULE_NAME} variable which should be used as argument to the other functions. Another name can be used as well, but below we exclusively use ${MODULE_NAME}
2. The next lines should contain the logic to load the dependencies of the module (below is an example to load Geant4). Only ROOT is automatically included and linked to the module.

3. A line with ALLPIX_MODULE_SOURCES(${MODULE_NAME} `sources`) where `sources` should be replaced by all the source files of this module
4. Possibly lines to include the directories and link the libraries for all the dependencies loaded earlier as explained in point 2. See below for an example.
5. A line containing ALLPIX_MODULE_INSTALL(${MODULE_NAME}) to setup the required target for the module to be installed to.

An example of a simple CMakeLists.txt of a module named `Test` which requires Geant4 is the following

```
# Define module and save name to MODULE_NAME
# Replace by ALLPIX_DETECTOR_MODULE(MODULE_NAME) to define a detector module
ALLPIX_UNIQUE_MODULE(MODULE_NAME)

# Load Geant4
FIND_PACKAGE(Geant4)
IF(NOT Geant4_FOUND)
    MESSAGE(FATAL_ERROR "Could not find Geant4, make sure to source the Geant4 environment\n$ source YOUR_GEANT4_DIR/bin/geant4.sh")
ENDIF()

# Add the sources for this module
ALLPIX_MODULE_SOURCES(${MODULE_NAME}
    TestModule.cpp
)

# Add Geant4 to the include directories
TARGET_INCLUDE_DIRECTORIES(${MODULE_NAME} SYSTEM PRIVATE ${Geant4_INCLUDE_DIRS})

# Link the Geant4 libraries to the library
TARGET_LINK_LIBRARIES(${MODULE_NAME} ${Geant4_LIBRARIES})

# Provide standard install target
ALLPIX_MODULE_INSTALL(${MODULE_NAME})
```

README.md *This section is not written yet.*

ModuleName.tex *This section is not written yet.*

`ModuleName`Module.hpp and `ModuleName`Module.cpp All modules should have both a header file and a source file. In the header file the module is defined together with all its method. Brief Doxygen documentation should be added to explain what every method does. The source file should provide the implementation of every method and also its more detailed Doxygen documentation. Not a single method should be defined in the header to keep the interface clean.

5.3.2. Module structure

All modules should inherit from the `Module` base class which can be found in *src/core/module/Module.hpp*. The module base class provides two base constructors, a few convenient methods and several methods to override. Every module should provide a constructor taking a fixed set of arguments defined by the framework. This particular constructor is always called during construction by the module instantiation logic. The arguments for the constructor differs for unique and detector modules. For unique modules the constructor for a `TestModule` should be:

```
TestModule(Configuration config, Messenger* messenger, GeometryManager*
↪  geo_manager): Module(config) {}
```

It is clear that the configuration object should be forwarded to the base module.

For unique modules the first two arguments are the same, but the last argument is a `std::shared_ptr` to the linked detector instead. It should always forward this provided detector to the base class, besides the configuration. Thus a constructor of a detector module should be:

```
TestModule(Configuration config, Messenger* messenger,
↪  std::shared_ptr<Detector> detector): Module(config, detector) {}
```

All modules receive the Configuration object holding the config parameters for that specific object, which can be accessed as explained in Section 5.2.2. Furthermore, a pointer to the Messenger is passed which can be used to both bind variables to receive and dispatch messages as explained in 5.5. Finally either a pointer to the GeometryManager is passed, which can be used to fetch all detectors, or a instance of the specifically linked detector. The constructor should normally be used to bind the required messages and set configuration defaults. In case of failure an exception can be thrown from the constructor.

In addition to the constructor every module can override the following methods:

- `init()`: Called after loading and constructing all modules and before starting the event loop. This method can for example be used to initialize histograms.

- `run(unsigned int event_number)`: Called for every event in the simulation run with the event number (starting from one). An exception should be thrown for every serious error, otherwise an warning should be logged.
- `finalize()`: Called after processing all events in the run and before destructing the module. Typically used to save the output data (like histograms). Any exceptions should be thrown from here instead of the destructor.

5.3.3. Module instantiation

The modules are dynamically loaded and instantiated by the Module Manager. Modules are constructed, initialized, executed and finalized in the linear order they are defined in the configuration file. Thus the configuration file should follow the order of the real process. For every non-special section in the main configuration file (see 5.2 for more details) a corresponding library is searched which contains the module. A module has the name **libAllPixModule**`ModuleName` reflecting the `ModuleName` of a defined module. The module search order is as follows:

1. The modules already loaded before from an earlier section header
2. All directories in the global configuration parameter *library_ directories* in the provided order if this parameter exists
3. The internal RPATH of the executable, that should automatically point to the libraries that are build and installed together with the executable.
4. The other standard locations to search for libraries depending on the operating system. Details about the procedure Linux follows are found in [14].

If the module definition is successful it is checked if the module is an unique or a detector module. The instantiation logic determines an unique name and priority, where a lower number indicates a higher priority, for every instantiation. The name and priority for the instantation are determined differently for the two types of modules:

- **Unique**: Combination of the name of the module and the **input** and **output** parameter (both defaulting to an empty string). The priority is always zero.
- **Detector**: Combination of the name of the module, the **input** and **output** parameter (both defaulting to an empty string) and the name of detector this module runs on. If the name of the detector is specified directly by the **name** parameter the priority is zero. If the detector is only matched by the **type** parameter the priority is one. If the **name** and **type** are both not specified and the module is instantiated for all detectors there priority is two.

The instantiation logic only allows a single instance for every unique name. If there are multiple instantiations with the same unique name the instantiation with the highest priority is kept (thus the one with the lowest number). Otherwise if there are multiple instantiations with the same name and the same priority an exception is raised.

5.4. Geometry and Detectors

Simulations are frequently run on a set of different detectors (such as as a beam telescope and a device under test). All these individual detectors together is what AllPix^2 defines as the geometry. Every detector has a set of properties attached to it:

- A unique **name** to refer to the detector in the configuration.
- The **position** in the world frame. This is the position of the geometric center of the sensitive device (sensor) given in world coordinates as X, Y and Z (note that any additional components like the chip or the PCB are ignored when determining the geometric center).
- The **orientation** given as Euler angles using the extrinsic Z-X-Z convention relative to the world frame (also known as the 1-3-1 or the "x-convention" and the most widely used definition of Euler angles [15]).
- A **type** of a detector model. The model defines the geometry and parameters of the detector. Multiple detectors can share the same model (and this is in fact very common). Several ready-to-use models are shipped with the framework.
- An optional **electric field** in the sensitive device. An electric field can be added to a detector by a special module as shown in Section 4.3.1.

The detector configuration is provided in the special detector configuration which is explained in Section 4.1.2.

The detectors can be accessed by name through the GeometryManager. If the module is a detector-specific module its related Detector can be accessed through the `getDetector()` method (returns a null pointer for unique modules) as follows:

```
void run(unsigned int event_id) {
    // Returns the linked detector
    std::shared_ptr<Detector> detector = this->getDetector();
}
```

5.4.1. Coordinate systems

All detectors have a fixed position in the world frame which has an arbitrary origin. Every detector also has a local coordinate system attached to it. The origin of this local coordinate system does not necessarily correspond with the geometric center of the sensitive device, which is the center of orientation of the detector in the global frame. The origin of the local coordinate system is instead based on the pixel grid in the sensor. This allows for simpler calculations that are also easier to read.

While the origin of the local coordinate system depends on the type of the model, there are fixed rules for the orientation of the coordinate system. The positive z-axis should point from the side of the sensor where collection takes place upwards (normal to the collection plane) to the other side of the sensitive device. The x-axis should point in one of the arbitrary two directions in the plane of the pixel grid. The y-axis should then be normal to both the x and the z-axis in such a way that a right-handed coordinate system is constructed. The 2D pixel grid is therefore in the XY-plane.

5.4.2. Detector models

Different types of detector models are already available and shipped with the framework. Every models extends from the `DetectorModel` base class which defines the minimum parameter of a detector model in the framework:

- The coordinate of the rotation center in the local frame. This is the location of the local point which is defined as position in the global frame.
- The position of the bottom-left (minimum) of the sensor (thus the sensitive device) in the local frame. This determines the excess of the sensor and the size of the guard rings around the pixel grid.
- The total size of the sensor (thus the top-right corner offset from the bottom-left minimum position).
- The number of pixels in the sensor. Every pixel is an independent block replicated over the XY plane of the sensor. The number of pixel defaults to one if it is not overridden, which means the default sensor has no replicated blocks.
- The size of an individual pixel. The multiplication of the pixel size and the number of pixels should not exceed the sensor.

This standard detector model can be extended to provide a more detailed geometry as required by certain modules. Currently the only included advanced detector model is the `PixelDetectorModel` , which apart from the sensor also includes guards rings, bump bonds, a readout ASIC chip, a PCB and a cover layer.

To fetch a detector model as `PixelDetectorModel`, the base class should be downcasted as follows (the downcast return a null pointer if it is not a PixelDetectorModel).

```
// Detector is a pointer to a Detector object
std::shared_ptr<PixelDetectorModel> model =
 ↪  std::dynamic_pointer_cast<PixelDetectorModel>(detector->getModel());
if(model != nullptr) {
    // The model of this Detector is a PixelDetectorModel
}
```

For more details about the different types of supported models and how to add your own new model, Section 7.2 should be consulted.

Many detector models are shipped with the framework in the configuration format introduced in Section 5.2.1. Other models can however be used in addition. To support different detector models and configuration formats the framework supports different types of model readers. These model readers search the directories in the following order:

1. If defined, the paths in the *models_path* parameter provided to the model reader module (for example the `DefaultModelReader`) or the global *models_path* parameter if no module-specific one is defined. Files are read and parsed directly. If the path is a directory, all files in the directory are added (not recursing into subdirectories).
2. The location where the models are installed to (see the MODEL_DIRECTORY variable in Section 3.4).
3. The standard data paths on the system as given by the environmental variable $XDG_DATA_DIRS with the allpix-directory appended. The $XDG_DATA_DIRS variable defaults to */usr/local/share/* (thus effectively */usr/local/share/allpix*) followed by */usr/share/* (effectively */usr/share/allpix*).

The framework provides a `DefaultModelReader` module to read all the default models in the framework. Every simulation should include this module at the beginning of the configuration file, unless another detector model reader is used.

5.5. Passing Objects using Messages

Communication between modules happens through messages (only some internal information is shared through external detector objects and the dependencies like Geant4). Messages are templated instantiations of the `Message` class carrying a vector of objects. A typedef is typically added by the object to provide an alternative name for the message directly linking to the carried object. The message system has a dispatching part and a receiving part.

The dispatching module can specify an optional name, but most modules should not specify this directly. If the name is not directly given (or equal to -) the **output** parameter of the module is used to determine the name of the message, defaulting to an empty string. Dispatching the message to their receivers then goes by the following rules:

1. The receiving module the will only receive a message if it has the exact same type as the message dispatched (thus carrying the exact same object). If the receiver is however listening to the `BaseMessage` type it will receive all dispatched messages instead.
2. The receiving module will only receive messages with the exact same name as it is listening for. The module uses the **input** parameter to determine to which message names the module should listen. If the **input** parameter is equal to `*` the module should listen to all messages. Every module listens by default to messages with no name specified (thus receiving the messages of default dispatching modules).
3. If the receiving module is a detector module it will only receive messages that are bound to that specific detector or messages that are not bound to any detector.

An example how to dispatch, in the `run()` function of a module, a message containing an array of `Object` types bound to a detector named `dut` is provided here:

```
void run(unsigned int event_id) {
    std::vector<Object> data;
    // .. fill the data vector with objects ...

    // The message is dispatched only for 'dut' detector
    std::shared_ptr<Message<Object>> message =
↪   std::make_shared<Message<Object>>(data, "dut");

    // Send the message using the Messenger object
    messenger->dispatchMessage(message);
}
```

5.5.1. Methods to process messages

The message system has multiple methods to process received messages. The first two are the most common methods and the third should only be used if necessary. The options are:

1. Bind a **single message** to a variable. This should usually be the preferred method as most modules only expect one message to arrive per event (as a module should typically send only one message containing the list of all the objects it should send).

An example of how to bind a message containing an array of **Object** types in the constructor of a detector `TestModule` would be:

```
TestModule(Configuration, Messenger* messenger,
 ↪  std::shared_ptr<Detector>) {
    messenger->bindSingle(this,
                          /* Pointer to the message variable */
                          &TestModule::message,
                          /* No special messenger flags */
                          MsgFlags::NONE);
}
std::shared_ptr<Message<Object>> message;
```

2. Bind a **set of messages** to an vector variable. This method should be used it the module can (and expects to) receive the same message multiple times (possibly because it wants to receive the same type of message for all detectors). An example to bind multiple messages containing an array of **Object** types in the constructor of a detector `TestModule` would be:

```
TestModule(Configuration, Messenger* messenger,
 ↪  std::shared_ptr<Detector>) {
    messenger->bindMulti(this,
                          /* Pointer to the message vector */
                          &TestModule::messages,
                          /* No special messenger flags */
                          MsgFlags::NONE);
}
std::vector<std::shared_ptr<Message<Object>>> messages;
```

3. Listen to a particular message type and execute a **listening function** as soon as an object is received. Should be used for more advanced strategies for fetching messages. The listening module should not do any heavy work in the listening function as this is supposed to take place in the `run()` method instead. An example of using this to listen to a message containing an array of `Object` types in a detector `TestModule` would be:

```
TestModule(Configuration, Messenger* messenger,
 ↪  std::shared_ptr<Detector>) {
    messenger->registerListener(this,
                                /* Pointer to the listener method */
                                &TestModule::listener,
                                /* No special message flags */
                                MsgFlags::NONE);
}
```

```
void listener(std::shared_ptr<Message<Object>> message) {
    // Do something with received message ...
}
```

5.5.2. Message flags

Various flags can be added to the bind function and listening functions. The flags enable a particular behaviour of the framework (if the particular type of method supports the flag).

- **REQUIRED**: Specify that this message is required to be received. If the particular type of message is not received before it is time to execute the run function, the run is automatically skipped by the framework. This can be used to ignore modules that cannot do any action without received messages, for example propagation without any deposited charges.
- **NO_RESET**: Messages are by default automatically reset after the `run()` function executes to prevent older messages from previous runs to appear again. This behaviour can be disabled by setting this flag (this does not have any effect for listening functions). Setting this flag for single bound messages (without ALLOW_OVERWRITE) would cause an exception to be raised if the message is overwritten in a later event.
- **ALLOW_OVERWRITE**: By default an exception is automatically raised if a single bound message is overwritten (thus setting it multiple times instead of once). This flag prevents this behaviour. It is only used for variables to a single message.
- **IGNORE_NAME**: If this flag is specified, the name of the dispatched message is not considered. Thus the **input** parameter is ignored and forced to the value `*`.

5.5.3. Object types

All supported objects that can be transferred between modules are shipped with the framework in the Objects library. This list of objects currently consists of the following:

- **DepositedCharge**: Set of charges at a specific position in the sensor of a detector. Deposited by an ionizing particle crossing the active material of the sensor.
- **PropagatedCharge**: Charge at a specific position after propagation.
- **PixelCharge**: Set of charges at a particular pixel in the pixel grid.

5.6. Logging and other Utilities

The AllPix^2 framework provides a set of utilities that can be attributed to two types:

- Two utilities to improve the usability of the framework. One of these is a flexible and easy-to-use logging system, introduced below in Section 5.6.1. The other is an easy-to-use framework for units that supports converting arbitrary combinations of units to an independent number which can be used transparently through the framework. It will be discussed in more detail in Section 5.6.2.
- A few utilities to extend the functionality provided by the C++ Standard Template Library (STL). These are provided either to simplify access to the STL (like the random seed generator utility) or to provide functionality the C++14 standard lacks (like filesystem support). The utilities are used internally in the framework and are only shortly discussed here. The utilities falling in this category are the random seed generator (see Section 5.6.3), the filesystem functions (see Section 5.6.3) and the string utilies (see Section 5.6.3).

5.6.1. Logging system

The logging system is build to handle input/output in the same way as `std::cin` and `std::cout`. This approach is both very flexible and easy to read. The system is globally configured, thus there exists only one logger, and no special local versions. To send a message to the logging system at a level of **LEVEL**, the following can be used:

```
1 LOG(LEVEL) << "this is an example message with an integer and a double "
  ↪ << 1 << 2.0;
```

A newline is added at the end of every log message. Multi-line log messages can also be used: the logging system will automatically align every new line under the previous message and will leave the header space empty on the new lines.

The system also allows for producing a message which is updated on the same line for simple progress bar like functionality. It is enabled using the `LOG_PROCESS(LEVEL, IDENTIFIER)` macro (where the `IDENTIFIER` is a special string to determine if the output should be written to the same line or not). If the output is a terminal screen the logging output will be colored to make it prettier to read. This will be disabled automatically for all devices that are not terminals.

More details about the various logging levels can be found in Section 4.4.

5.6.2. Unit system

Correctly handling units and conversions is of paramount importance. Having a separate C++ type for all different kind of units would however be too cumbersome for a lot of operations. Therefore the units are stored in standard C++ floating point types in a default unit which all the code in the framework uses for calculations. In configuration files as well as for logging it is however very useful to provide quantities in a different unit.

The unit system allows adding, retrieving, converting and displaying units. It is a global system transparently used throughout the framework. Examples of using the unit system are given below:

```
// Define the standard length unit and an auxiliary unit
Units::add("mm", 1);
Units::add("m", 1e3);
// Define the standard time unit
Units::add("ns", 1);
// Get the units given in m/ns in the defined framework unit mm/ns
Units::get(1, "m/ns");
// Get the framework unit of mm/ns in m/ns
Units::convert(1, "m/ns");
// Give the unit in the best type as string (lowest number higher than one)
//   input is default unit 2000mm/ns and 'best' output is 2m/ns (string)
Units::display(2e3, {"mm/ns", "m/ns"});
```

More details about how the unit system is used within AllPix can be found in Section 4.1.1.

5.6.3. Internal utilities

Filesystem Provides functions to convert relative to absolute canonical paths, to iterate through all files in a directory and to create new directories. These functions should be replaced by the C++17 filesystem API [16] as soon as the framework minimum standard is updated to C++17.

String utilities The STL only provides string conversions for standard types using `std::stringstream` and `std::to_string`. It does not allow to parse strings encapsulated in pairs of `"` characters and neither does it allow to integrate different units. Furthermore it does not provide wide flexibility to add custom conversions for other external types in either way. The AllPix[2] `to_string` and `from_string` do allow for these flexible conversions and it it extensively used in the configuration system. Conversions of numeric types with a

unit attached are automatically resolved using the unit system discussed in Section 5.6.2. The AllPix^2 tools system contain extensions to allow automatic conversions for ROOT and Geant4 types as explained in Section 9.1. The string utilities also include trim and split strings functions as they are missing in the STL.

Random seed generator Generating random number with a high level of entropy is very important for running Monte-Carlo processes in high-energy physics. Good random generators are already included in both the standard library and various external libraries. Seeding those random generators with enough entropy or seeding them with a predefined seed instead (to allow for reproducing a certain simulation) should however be a framework task to ensure that seeds are transferred in a well-defined way. This small utility library provides the required method to provide a seed to all random generators. With the STL random libraries it can be used as shown in the following example `init()` method:

```
void init() {
    // Create a standard 64 bit Mersene Twister 19937 generator
    std::mt19937_64 random_generator;

    // Get a seed from the framework
    uint64_t seed = get_random_seed();

    // Seed the generator with the random seed provided by the framework
    random_generator.seed(seed);

    // Use the random generator to produce a gaussian distribution
    std::normal_distribution<double> gauss_distribution(0, 1);
    double gauss_number = gauss_distribution(random_generator);
}
```

5.7. Error Reporting and Exceptions

AllPix^2 generally follows the principle to throw exceptions in all cases where something is definitely wrong, it should never try to circumvent problems. Also error codes are not supposed to be returned, only exceptions should be used to report fatal errors. Exceptions are also thrown to signal for errors in the user configuration. The asset of this method is that configuration and code is more likely to do what they are supposed to do.

For warnings and informational messages the logging should be used extensively. This helps in both following the progress of the simulation as well as for debugging problems. Care should however be taken to limit the amount of messages outside of the `DEBUG` and `TRACE` levels. More details about the log levels and their usage is given in Section 4.4.

The base exceptions in AllPix are available in the utilities. The most important exception base classes are the following:

- **ConfigurationError**: All errors related to incorrect user configuration. Could be a non-existing configuration file, a missing key or an invalid parameter value.
- **RuntimeError**: All other errors arising at run-time. Could be related to incorrect configuration if messages are not correctly passed or non-existing detectors are specified. Could also be raised if errors arise while loading a library or running a module.
- **LogicError**: Problems related to modules that do not properly follow the specifications, for example if a detector module fails to pass the detector to the constructor. These methods should never be raised for a well-behaving module and should therefore not be triggerable by users. Reporting these type of errors can help developers during their development of new modules.

Outside of the core framework, exceptions can also be used directly by the modules. There are only two exceptions which should be used by typical modules to indicate errors:

- **InvalidValueError**: Available under the subset of configuration exceptions. Signals any problem with the value of a configuration parameter that is not related to either the parsing or the conversion to the required type. Can for example be used for parameters where the possible valid values are limited, like the set of logging levels, or for paths that do not exist. An example is shown below:

```
1  void run(unsigned int event_id) {
2      // Fetch a key from the configuration
3      std::string value = config.get("key");
4
5      // Check if it is a 'valid' value
6      if(value != 'A' && value != "B") {
7          // Raise an error if it the value is not valid
8          //   provide configuration object, key and an explanation
9          throw InvalidValueError(config, "key", "A and B are the only
   ↪  allowed values");
10     }
11 }
```

- **ModuleError**: Available under the subset of module exceptions. Should be used to indicate any runtime error in a module that is not directly caused by an invalid configuration value. For example if it is not possible to write an output. A reason should be given to indicate what the problem is.

6. Modules

This section is not written yet.

7. Module & Detector Development

7.1. Implementing a New Module

Before creating a module it is essential to read through the framework module manager documentation in Section 5.3, the information about the directory structure in Section 5.3.1 and the details of the module structure in Section 5.3.2. Thereafter the steps below should provide enough details for starting with a new module `ModuleName` (constantly replacing `ModuleName` with the real name of the new module):

1. The whole directory contents of *src/modules/DummyModule/* should copied to *src/modules/`ModuleName`/*.
2. The DummyModule.hpp should be renamed to `ModuleName`Module.hpp and the DummyModule.cpp to `ModuleName`Module.cpp.
3. The CMakeLists.txt has to be modified depending on the module type. Depending on the type of the module the first line is different. If the new module is a unique module it should be ALLPIX_UNIQUE_MODULE(MODULE_NAME), if it is a detector-specific module it should be ALLPIX_DETECTOR_MODULE(MODULE_NAME). Next, the source file created in the previous step has to replace the original dummy source file.
4. The header and source files have to be implemented following the constructor conventions for the specific type of module.
5. The initial documentation in the README.md and the LaTeX-file `ModuleName`.tex can already be started with before the module is implemented.
6. Now the the initial parts of the constructor, and possible the `init`, `run` and/or `finalize` methods can be written, depending on what the new module is supposed to do.

After this, it is up to the developer to implement all the required functionality in the module. Keep considering however that at some point it may be beneficial to split up modules to support the modular design of AllPix^2. Various sources which may be primarily useful during the development of the module include:

- The framework documentation in Section 5 for an introduction to the different parts of the framework.
- The module documentation in Section 6 for a description of functionality other modules already provide and to look for similar modules which can help during development.
- The Doxygen (core) reference documentation included in the framework .

- The latest version of the source code of all the modules (and the core itself). Freely available to copy and modify under the MIT license at `https://gitlab.cern.ch/simonspa/allpix-squared/tree/master`.

Any module that may be useful for other people can be contributed back to the main reposity. It is very much encouraged to send a merge-request at `https://gitlab.cern.ch/simonspa/allpix-squared/merge_requests`.

7.2. Adding a New Detector Model

This section is not written yet.

8. Frequently Asked Questions

How do I run a module only for one detector?

This is only possible for detector modules (which are constructed to work on individual detectors). To run it on a single detector one should add a parameter `name` specifying the name of the detector (as given in the detector configuration file).

How do I run a module only for a specific detector type?

This is only possible for detector modules (which are constructed to work on individual detectors). To run it for a specific type of detectors one should add a parameter `type` with the type of the detector model (as given in the detector configuration file by the `model` parameter).

How can I run the exact same type of module with different settings?

This is possible by using the `input` and `output` parameters of a module that specialize the location where the messages from the modules are send to and received from. By default both the input and the output of module defaults to the message without a name.

How can I temporarily ignore a module during development?

The section header of a particular module in the configuration file can be replaced by the string `Ignore`. The section and all of its key/value pairs are then ignored.

Can I get a high verbosity level only for a specific module?

Yes, it is possible to specify verbosity levels and log formats per module. This can be done by adding a `log_level` and/or `log_format` key to a specific module to replace the parameter in the global configuration sections.

9. Additional Tools & Resources

9.1. ROOT and Geant4 utilities

This section is not written yet.

9.2. Runge-Kutta solver

This section is not written yet.

9.3. TCAD Electric Field Converter

This section is not written yet.

9.4. Simple Usage Examples

This section is not written yet.

10. Acknowledgments

- **Mathieu Benoit**, **John Idarraga**, **Samir Arfaoui** and all other contributors to the first version of AllPix, for their earlier work that inspired AllPix^2.
- **Neal Gauvin** for interesting discussion, his experiments with TGeo and his help implementing a visualization module.
- **Paul Schütze** for contributing his earlier work on simulating charge propagation and providing help on simulations with electric fields.
- **Marko Petric** for his help setting up several software tools like continous integration and automatic static-code analysis.

We would also like to thank all others not listed here, that have contributed to the source code, provided input or suggested improvements.

A. Output of Example Simulation

Possible output for the example simulation in Section 4.3 is given below:

```
(S) Welcome to AllPix v0.2beta1+5^gf61a69d
(S) Initialized PRNG with seed 7964691284681267564
(S) Loaded 8 modules
(S) Initializing 14 module instantiations
(I) [I:DepositionGeant4] Not depositing charges in telescope2
   because there is no listener for its output
(I) [I:DepositionGeant4] Not depositing charges in dut because
   there is no listener for its output
(S) Initialized 14 module instantiations
(S) Running event 1 of 10
(I) [R:DepositionGeant4] Deposited 29995 charges in sensor of
   detector telescope1
(I) [R:SimplePropagation:telescope1] Propagated 29995 charges in
   600 steps in average time of 11.3608ns
(I) [R:SimpleTransfer:telescope1] Transferred 29995 charges to 1
   pixels
(I) [R:DefaultDigitizer:telescope1] Digitized 1 pixel hits
(W) [R:DefaultDigitizer:telescope1] Dispatched message Message<
   allpix::PixelHit> from DefaultDigitizer:telescope1 has no
   receivers!
(S) Running event 2 of 10
(I) [R:DepositionGeant4] Deposited 84794 charges in sensor of
   detector telescope1
(I) [R:SimplePropagation:telescope1] Propagated 84794 charges in
   1696 steps in average time of 11.3571ns
(I) [R:SimpleTransfer:telescope1] Transferred 84794 charges to 1
   pixels
(I) [R:DefaultDigitizer:telescope1] Digitized 1 pixel hits
(W) [R:DefaultDigitizer:telescope1] Dispatched message Message<
   allpix::PixelHit> from DefaultDigitizer:telescope1 has no
   receivers!
(S) Running event 3 of 10
(I) [R:DepositionGeant4] Deposited 27363 charges in sensor of
   detector telescope1
(I) [R:SimplePropagation:telescope1] Propagated 27363 charges in
   548 steps in average time of 11.387ns
(I) [R:SimpleTransfer:telescope1] Transferred 27363 charges to 1
   pixels
(I) [R:DefaultDigitizer:telescope1] Digitized 1 pixel hits
```

```
(W) [R:DefaultDigitizer:telescope1] Dispatched message Message<
    allpix::PixelHit> from DefaultDigitizer:telescope1 has no
    receivers!
(S) Running event 4 of 10
(I) [R:DepositionGeant4] Deposited 32071 charges in sensor of
    detector telescope1
(I) [R:SimplePropagation:telescope1] Propagated 32071 charges in
    642 steps in average time of 11.3811ns
(I) [R:SimpleTransfer:telescope1] Transferred 32071 charges to 1
    pixels
(I) [R:DefaultDigitizer:telescope1] Digitized 1 pixel hits
(W) [R:DefaultDigitizer:telescope1] Dispatched message Message<
    allpix::PixelHit> from DefaultDigitizer:telescope1 has no
    receivers!
(S) Running event 5 of 10
(I) [R:DepositionGeant4] Deposited 45021 charges in sensor of
    detector telescope1
(I) [R:SimplePropagation:telescope1] Propagated 45021 charges in
    901 steps in average time of 11.3894ns
(I) [R:SimpleTransfer:telescope1] Transferred 45021 charges to 1
    pixels
(I) [R:DefaultDigitizer:telescope1] Digitized 1 pixel hits
(W) [R:DefaultDigitizer:telescope1] Dispatched message Message<
    allpix::PixelHit> from DefaultDigitizer:telescope1 has no
    receivers!
(S) Running event 6 of 10
(I) [R:DepositionGeant4] Deposited 33678 charges in sensor of
    detector telescope1
(I) [R:SimplePropagation:telescope1] Propagated 33678 charges in
    674 steps in average time of 11.3607ns
(I) [R:SimpleTransfer:telescope1] Transferred 33678 charges to 1
    pixels
(I) [R:DefaultDigitizer:telescope1] Digitized 1 pixel hits
(W) [R:DefaultDigitizer:telescope1] Dispatched message Message<
    allpix::PixelHit> from DefaultDigitizer:telescope1 has no
    receivers!
(S) Running event 7 of 10
(I) [R:DepositionGeant4] Deposited 40388 charges in sensor of
    detector telescope1
(I) [R:SimplePropagation:telescope1] Propagated 40388 charges in
    808 steps in average time of 11.3542ns
(I) [R:SimpleTransfer:telescope1] Transferred 40388 charges to 1
    pixels
```

```
(I) [R:DefaultDigitizer:telescope1] Digitized 1 pixel hits
(W) [R:DefaultDigitizer:telescope1] Dispatched message Message<
    allpix::PixelHit> from DefaultDigitizer:telescope1 has no
    receivers!
(S) Running event 8 of 10
(I) [R:DepositionGeant4] Deposited 26686 charges in sensor of
    detector telescope1
(I) [R:SimplePropagation:telescope1] Propagated 26686 charges in
    534 steps in average time of 11.3863ns
(I) [R:SimpleTransfer:telescope1] Transferred 26686 charges to 1
    pixels
(I) [R:DefaultDigitizer:telescope1] Digitized 1 pixel hits
(W) [R:DefaultDigitizer:telescope1] Dispatched message Message<
    allpix::PixelHit> from DefaultDigitizer:telescope1 has no
    receivers!
(S) Running event 9 of 10
(I) [R:DepositionGeant4] Deposited 32342 charges in sensor of
    detector telescope1
(I) [R:SimplePropagation:telescope1] Propagated 32342 charges in
    647 steps in average time of 11.3717ns
(I) [R:SimpleTransfer:telescope1] Transferred 32342 charges to 1
    pixels
(I) [R:DefaultDigitizer:telescope1] Digitized 1 pixel hits
(W) [R:DefaultDigitizer:telescope1] Dispatched message Message<
    allpix::PixelHit> from DefaultDigitizer:telescope1 has no
    receivers!
(S) Running event 10 of 10
(I) [R:DepositionGeant4] Deposited 34287 charges in sensor of
    detector telescope1
(I) [R:SimplePropagation:telescope1] Propagated 34287 charges in
    686 steps in average time of 11.376ns
(I) [R:SimpleTransfer:telescope1] Transferred 34287 charges to 1
    pixels
(I) [R:DefaultDigitizer:telescope1] Digitized 1 pixel hits
(W) [R:DefaultDigitizer:telescope1] Dispatched message Message<
    allpix::PixelHit> from DefaultDigitizer:telescope1 has no
    receivers!
(S) Finished run of 10 events
(I) [F:DepositionGeant4] Deposited total of 386625 charges in 1
    sensor(s) (average of 38662 per sensor for every event)
(I) [F:SimplePropagation:telescope1] Propagated total of 386625
    charges in 7736 steps in average time of 11.3702ns
```

```
(I) [F:SimpleTransfer:telescope1] Transferred total of 386625
    charges to 1 different pixels
(I) [F:SimpleTransfer:telescope2] Transferred total of 0 charges
    to 0 different pixels
(I) [F:SimpleTransfer:dut] Transferred total of 0 charges to 0
    different pixels
(I) [F:DefaultDigitizer:telescope1] Digitized 10 pixel hits in
    total
(I) [F:DefaultDigitizer:telescope2] Digitized 0 pixel hits in
    total
(I) [F:DefaultDigitizer:dut] Digitized 0 pixel hits in total
(S) Finalization completed
(S) Executed 14 instantiations in 11 seconds, spending 92% of
    time in slowest instantiation SimplePropagation:telescope1
```

References

[1] S. Agostinelli et al. "Geant4 - a simulation toolkit". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 506.3 (2003), pp. 250–303. ISSN: 0168-9002. DOI: https://doi.org/10.1016/S0168-9002(03)01368-8.

[2] *ROOT - An Object Oriented Data Analysis Framework*. Vol. 389. Sept. 1996, pp. 81–86.

[3] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. 2010. URL: http://eigen.tuxfamily.org.

[4] Mathieu Benoit et al. *The AllPix Simulation Framework*. Mar. 21, 2017. URL: https://twiki.cern.ch/twiki/bin/view/Main/AllPix.

[5] Mathieu Benoit, John Idarraga, and Samir Arfaoui. *AllPix. Generic simulation for pixel detectors*. URL: https://github.com/ALLPix/allpix.

[6] Rene Brun and Fons Rademakers. *Building ROOT*. URL: https://root.cern.ch/building-root.

[7] Geant4 Collaboration. *Geant4 Installation Guide. Building and Installing Geant4 for Users and Developers*. 2016. URL: http://geant4.web.cern.ch/geant4/UserDocumentation/UsersGuides/InstallationGuide/html/.

[8] X. Llopart et al. "Timepix, a 65k programmable pixel readout chip for arrival time, energy and/or photon counting measurements". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 581.1 (2007). VCI 2007, pp. 485–494. ISSN: 0168-9002. DOI: http://dx.doi.org/10.1016/j.nima.2007.08.079.

[9] Geant4 Collaboration. *Geant4 User's Guide for Application Developers. Visualization*. 2016. URL: https://geant4.web.cern.ch/geant4/UserDocumentation/UsersGuides/ForApplicationDeveloper/html/ch08.html.

[10] Morris Swartz. *A detailed simulation of the CMS pixel sensor*. Tech. rep. 2002.

[11] Morris Swartz. "CMS pixel simulations". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 511.1 (2003), pp. 88–91.

[12] Rene Brun and Fons Rademakers. *ROOT User's Guide. Trees*. URL: https://root.cern.ch/root/htmldoc/guides/users-guide/Trees.html.

[13] Tom Preston-Werner. *TOML. Tom's Obvious, Minimal Language*. URL: https://github.com/toml-lang/toml.

[14] Michael Kerrisk. *Linux Programmer's Manual. ld.so, ld-linux.so - dynamic linker/loader*. URL: http://man7.org/linux/man-pages/man8/ld.so.8.html.

[15] Eric W. Weisstein. *Euler Angles. From MathWorld – A Wolfram Web Resource*. URL: http://mathworld.wolfram.com/EulerAngles.html.

[16] Beman Dawes. *Adopt the File System TS for C++17*. Feb. 2016. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0218r0.html.