MATRIX Documentation

Release 0.1

Matthias Benkort, Nicolas Gaborit, Arthur Manoha, Matthieu Pizenberg, Louis Viot

March 17, 2015

CHAPTER

ONE

CONTEXT AND OBJECTIVES

Arthur it is your part.

1.1 On-set special effects

1.2 Current solution

1.3 Objectives of the application

CHAPTER

TWO

USE OF THE APPLICATION

Here we will describ how to use the application. At the end of this chapter, you will know how to import pictures, launch a 3D reconstruction and see the results on the map and the 3D renderer.

Here is a video demo of our application which shows you quickly how to use it :

2.1 Managing workspaces and scenes

2.1.1 Workspaces

Here we will explain how to manage workspaces. They are working the same way than eclipse workspaces. In our software, you can create, delete, save or change workspace.

- To create a workspace : in the menu bar, click on *Workspace* then *New workspace*, or use the keyboard shortcut *Ctrl+Shift+N*
- To save your current workspace under a specific location : click on *Workspace* then *Save Workspace*, or use the keyboard shortcut *Ctrl+s*
- To open an already existing workspace : click on *Workspace* then *Open Workspace*
- To close the current workspace : click on *Workspace* then *Close Workspace*
- To change the current workspace to an already existing workspace : click on *Workspace* then *Change Workspace*

2.1.2 Scenes

To use scenes, you must be working in a specific workspace. Once the workspace is created, the application automatically creates a *default* scene. You can do the usual operation with scenes :

- To create a scene : click on *Scene* then *New scene* or use the keyboard shortcut *Ctrl+n*
- To delete the current scene : click on *Scene* then *Delete scene* or use the keyboard shortcut *Ctrl+d*
- To change the current scene : click on *Scene* then *Change scene*

In a scene, you can import pictures. You can either import them from a directory, or from a camera. We will be talking about how to import pictures from a camera in another part.

2.2 Managing the photo selection

Once photos have been imported in your scene, you will be able to manipulate them. But first, you have to know the different states a photo can have. It can either be:

- Discarded : you don't the photo to be in the next reconstruction or the photo was in thumbnail state, but you don't want to download the real picture from the camera
- New : the photo had just been imported or the photo was in discarded state, but you changed your mind and you actually want to download it.
- Processed : the photo has been processed by the reconstruction
- Rejected : the photo couldn't be used by the reconstruction processed and was rejected by openMVG
- Thumbnails : the photo is not the actual photo but just a thumbnail
- Valid : the photo is either in the state *New* or *Processed* (all the files that will be used for the next reconstruction)
- Reconstruction : the photo is being used for the reconstruction

Note that only photos in *New* and *Processed* state will be used for the next reconstruction.

Then, you can do a few things with them :

- You can manipulate them, sort them in any order you want.
- You can obviously select several photos by using *Ctrl* key or you can use the well known *Ctrl+Shift* key shortcut. Once a photo is selected, you can drag and drop it wherever you want.
- You can discard a photo, renew it (meaning it won't be discarded anymore), or you can delete it from the list.
- You can filter the picture list to display only photos in a specific state.

2.3 Camera connection

To connect a camera, simply plug in your camera and the application will detect it automatically. If it does, you will see a green *Connected* on the screen and the name of your camera (if gphoto is able to detect it).

Once your camera is connected, and once you have a functionnal workspace with the default scene or another scene, you can import pictures from it. To do so, simply follow this instruction :

- Click on *Scene* then *Import Pictures* then *From camera ...*

Note that this will import every pictures on your camera as thumbnails. This can take quiet a long time, even thumbnails are usually lightweight.

Once the import is done, you can choose which thumbnail to not import as real picture by setting there state to *Discarded*. Once your selection is done, simple click on *Tools* and then *Confirm thumbnails*

2.4 3D Reconstruction and Rendering

Once you have selected the photos you want in your next reconstruction (state *New* and *Processed*), you can launch the 3D reconstruction by :

- Clicking on *Tools* and then *Launch 3D reconstruction*

For the moment, the reconstruction will freeze the application and you might have to select a pair of photos in the terminal at a certain point of the reconstruction to feed openMVG. The application will then display the final point cloud result. You will see the point cloud from the point of view of one of the camera but you cannot in this version choose and switch camera. Note that somme function are ready to be used in the code to switch camera and change the point of view.

2.5 Map localization

Once you have a workspace and a scene with your pictures, our application extract the GPS data from the exif data in the photos. If a photo doesn't have exif data, then the GPS default value will be 0.

Then you have two options :

- If you switch off the map, then you will only see the renderer in the center of the screen once a reconstruction has been done.
- If you switch on the map, you will see some blue points on a map corresponding to the location where you take your pictures. If you click on a picture in the list, then the map will be centered on the point corresponding to this picture.

Points in the map can have different colors corresponding to their state :

- Blue : the picture is in state *New*
- Green : the picture is in state *Processed*
- Red : the picture is either in state *Rejected*, *Discarded* or *Thumbnail*

CHAPTER

THREE

ARCHITECTURE OF THE APPLICATION

Here we will first outline the architecture of the application and then enter in implementation details of each module.

3.1 Global architecture of the application

The application has been splitted into several independent modules. Before going further in the description of connections and contents of each modules, let's define the concept of *module* in our case.

3.1.1 What is called module

A *module* is related to a group of functions that are used to manipulate a specific entity. A module has no idea of the existence of other module and should never interact with another module directly. Instead of that, and thanks to the signals and slots system in Qt which may be seen as an implementation of the *publisher / subscriber* pattern, each module can emit signals whenever it wants to transmit some pieces of information to an interested module. In the same way, module can have slots that allow them to receive data from elsewhere.

3.1.2 Connections between modules

With this architecture, it is necessary to create top-level components that handle all connections between signals and slots of modules we want to link. This top-level components is the only one that import and know each module of the final application. Thus, if a module has to change because of any updates, the other modules will not have to be impacted by thoses changes; the top-level component is in charge of adapting the way they are connected if necessary. Also, the top-level component is responsible for both instantiating each module and handling errors coming from them.

On a first approach, we decided to only communicate with modules using signals and slots, and then, use the top-level component as a kind of big switchboard operator (like in 1950's/1960's) which may grab signals from one module, and dispatch them to other interested modules. However, there are some actions in the application that lead each time to identical successive calls to primary functions of several modules. Therefore, we've decided to use the top-level component to call directly functions of modules and transmit each result as parameters. Actually, signals and slots correspond to events and their handling. An event may require different actions on several modules. But, this actions are handled by the top-level component. The code below gives an overview of this principle (using the PyQt5 Syntax):

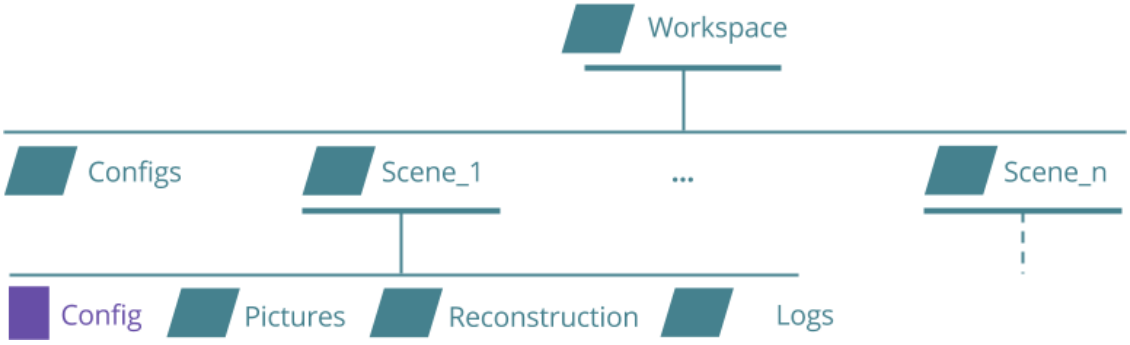
```
@pyqtSlot()
def slot_launchReconstruction():
  intermediateResult = module1.function1()
  module2.function2(intermediateResult)
  module3.function3()

...
```

```
view.sig_launchReconstruction(slot_launchReconstruction)
```

In such a way, when the signal **sig_launchReconstruction** is emitted by an entity (for instance, a button pressed in the graphical interface), the corresponding slot will proceed and call functions to carry out the reconstruction.

3.1.3 A same working space

In order to simplify the manipulation of data within the application, we've decided that every pieces of data (photos, configuration files, exports and temporary files) will be placed under a same folder. This folder is called a *scene*. Then, as some parameters may be common to several scenes (that can be photos, or configs), all *scene* folders have been placed under a global parent folder called *workspace*. Then, it is possible for the user to create a *workspace* wherever he wants to but then, every single piece of data in that workspace should be manipulated by the application. A specific module in charge of those manipulations, the *workspace manager*, is detailed in the Components Implementation .



To keep the same spirit as before, modules don't have an idea of the structure of the workspace. That's why, every module which want to export data in or grab data from the files system may require I/O paths as arguments. We also completly move aside the knowing of the workspace in a given module. The top-level component previously presented has therefore no idea of how the workspace is structured and may ask the *workspace manager* for any filepath.

3.1.4 Implementation details

In order to build the whole GUI, we have been using the QML language to handle all displays and user interactions. Indeed, we follow here a *model / view pattern* (described in the next section), where the view is also responsible for the user inputs. Also, as we shared the whole application into several modules, the view will be splitted all the same into several little views called *Components*. The challenge here, is to make all modules and *Components* communicate to each other as they are written using different tools. On one side, we are using PyQt5 which is a Python API that maps the existing C++ API Qt used for graphical user interface purposes. On the other side, we are building the view using the QML language, which is part of the Qt API as well. In fact, Qt gives us the opportunity of making views using their declarative language, and also enable smart ways to communicate from these views to a model or at least, an operative and logic part of the application. This is possible via *Signals and Slots* described previously. For more details about the way is implemented the communication, please go ahead to the next section.

3.2 The model / view pattern

3.2.1 General principle

The *model / view pattern* is derived from the *model - view - controller* pattern tradionally used in web application and in a more general way, in any application that needs to keep a separated logic part. In our case, view and controller are

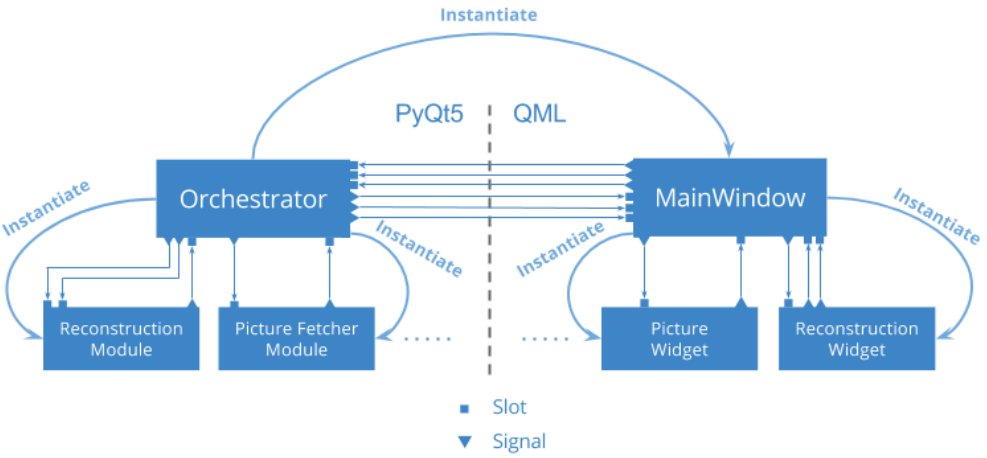
gathered into what we called view. This pattern is then called *model / view* and is the one on which Qt's philosophy relies on.

As the whole concept is well-explained in the Qt's documentation, we will only gives here a small synthesis. The idea of this pattern is to shared components into two kinds of components, depending of their responsabilities; Components responsibles for rendering item and interacting with the user are in the view part. The other, responsible for storing and managing the data, they are called models. Once a view component catches an action from the user, it may either inform the model that it should update itself; either ask the model for some data to display.

In most cases, models may exists and be used without a view. The view is only there to translate user action into concrete models calls. It is a convenient way for a user to manipulate the data whereas it is a convenient way for the programmer to supply and store the same data.

3.2.2 MATRIX model / view implementation

In the application, we followed and implemented the *model / view* pattern. As it has been said before, all view parts have been written using QML while models rely on Python and PyQt5. Beside, in the python's side, models live alongside other modules that are just interface to library or external ressources (for instance, the module who's handling gphoto2 doesn't define any model. It acts like a façade and offer control operations on a Camera). That's why, in order to manage all those modules, there is a top-level component called the *Orchestrator* in the Python's side. This is rigorously the same in the QML's side; Components may be seen as independant entities, all orchestrated by a top-level view called the *MainWindow*. As both side rely on the same philosophy, we'll describe only the Python's side and admit that things are transposable to the QML's side.



The Orchestrator's role is quite straightforward : it instantiates each module, instantiates the view, and watches for signals emitted either by modules, or either by the view. When a signal is catched, there are then two different strategies :

- Directly connect the signal to a corresponding slot
- Define a specific slot to handle the signal

In fact, we use the second case when the signal can't be connected directly because it involves several modules and/or models. The Orchestrator is then responsible for executing each action necessary to perform the global action required by the signal. It is common in our application that the orchestrator emits at the end a new signal toward the view to inform it that the action has been performed successfully.

As a matter of fact, communications between the QML's side and the Python's side are made only (or at least, mainly) between the Orchestrator and the Mainwindow. To be totally exact, actions that only require to retrieve data from a model corresponds to direct model method calls in QML. Thus some models (for instance, the *pictureModel*) define internal slots that are never used by the Orchestrator, but corresponds to callable functions in QML. Moreover, PyQt5

doesn't make any distinction between *INVOKABLE methods* and *SLOTS* from the Qt framework. That's why some slots special slots are necessary, but they should only be defined on models and models items and only on accessors. All other requests from the view should be transmitted to the MainWindow which will either handle it locally, or either transmit it to the Orchestrator.

3.3 Components implementation

3.3.1 Picture Fetcher

Contents

This module is made of a single class `Pygphoto` defined in the file pygphoto.py. It allows to interact with a USB camera using the gphoto2 tool. Common actions include : listing the names of the photos present in the camera, watching for new files, downloading photos individually etc.

Interface with the `gphoto2` command-line interface

The module uses directly the gphoto2 command-line interface for prototyping. It would be wiser to create a *C* module that uses the `C` API of gphoto2 called libgphoto2 and offer a interface for all the needed operations. This *C* module would be compiled as a shared library and imported in Python with ctypes.

However, the current implementation makes extended use of the command-line interface provided by `gphoto2`, through hardcoded calls and output parsing.

The calls to gphoto2 are made with the subprocess Python module.

Simple calls are done like so:

```
command = ["gphoto2", "--get-all-files"]
return_code = subprocess.call(command)
```

When the output need to be parsed, it is necessary to use the blocking `subprocess.check_output()` function and to decode the binary stream as an UTF-8 string:

```
command = ["gphoto2", "--summary"]
output_string = subprocess.check_output(command).decode("utf-8")
```

Active watching of the camera

When instantiated, the `Pygphoto` class create a daemon thread that actively watch for new camera connections/disconnections and pictures creation/deletion. The two associated signals `onCameraConnection` and `onContentChanged` are the only signals emitted by `Pygphoto`.

The `CameraWatcher` internal class is responsible for the active watching. The threading is made by moving the instance of `CameraWatcher` to a `QThread` at initialisation. Using `QThread` allows for easy asynchronous communication with the `CameraWatcher`'s thread through the use of *Qt*'s signals.

Note: As the `CameraWatcher` class is internal, `Pygphoto` must forward every signals emitted by `CameraWatcher` to the exterior. The same goes for slots connections. As a consequence, every slot and signal of `CameraWatcher` is duplicated in the the parent `Pygphoto` class.

Camera locking

The camera device can be busy for several reasons. This usually end up throwing the following error message

```
*** Error ***
An error occurred in the io-library ('Could not lock the device'): Camera is already in use.
*** Error (-60: 'Could not lock the device') ***
```

Sometimes it comes from other processes or daemons accessing the camera. We do not do anything about them and we did not explore this issue further. Sometimes, however, the lock comes from simultaneous calls to `gphoto2` made by the application. To avoid such problems, we use a thread synchronization lock every time we make a call to `gphoto2`:

```
self._camera_lock = threading.Lock()
command = [Pygphoto._GPHOTO, "--auto-detect"]
with self._camera_lock:
    output = subprocess.check_output(command).decode("utf-8")
# The call has returned, the lock is released
# Continue working with output...
```

The lock is internal to the `Pygphoto` class, the only class that can make calls to `gphoto2`. The lock issue is thus hidden from the exterior. It is important to note that the `Pygphoto` class should not be instantiated twice, and could actually be implemented as a singleton class.

3.3.2 Picture Manager

This section describes several things that, once combined, allow the application to manage pictures. Models, entities and manager will be described. We'll also write a bit about the storage system.

Picture model

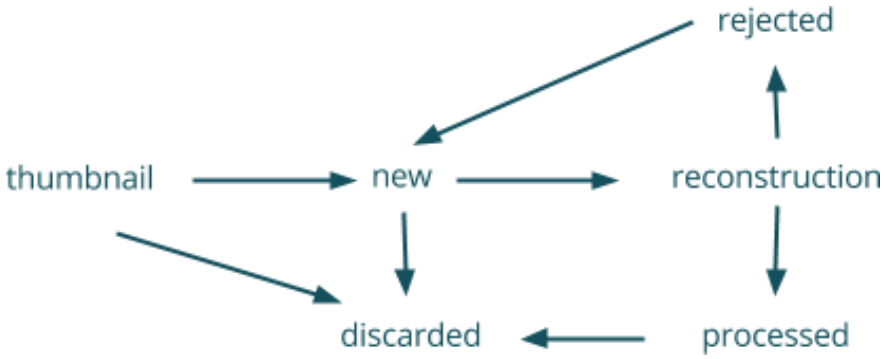
In order to manipulate pictures in the *Picture Manager*, we've defined a specific model in Python. Besides, this model is also used by the *Workspace Manager* and the *Map Viewer* as they share common needs that may be represented into one single model. Concretly our Python implementation is classic Python class that inherit from the QAbstractListModel of Qt (or more exactly, the PyQt5 representation). Then, the model is implementing required methods to access data, set data and easily manipulate rows. We've strictly respected Qt documentation and headers to write functions content; thus, Qt's doc could be used to describe each implemented function. There is also a bunch of fonctionalities needed for our typical application purpose such as specific accessors or hydratation method.

Also, the picture is implementing a *METAClass Savable* that allow it to be stored and loaded by the *Workspace Manager* into and from the workspace. This is possible with only four methods calls on an instance of the model, which are : serialize, unserialize, save and load. Their meanings are pretty straightforward.

Picture model's items : Picture

Each model is a kind of container that store certain kind of item. In our case, those items are pictures, or at least, our idea of a picture. In Matrix, a picture represent several things : a path to an image file, a name, a status, a latitude, a longitude, a date of creation, a color and an icon. All those attributes are called *roles* in the Qt world. Some of them

could be directly obtained from another (color and icon from status for instance) and thus, they are just programming sugars. However, most of the role are straightforward, except maybe the status role which we'll describe below.



Thereby, a picture can have a status. The actual goal of this status is either to know which pictures to select for a reconstruction, or either to inform users about how the application is considering the picture. A picture can start in only two states : **THUMBNAIL** or **NEW**; **THUMBNAIL** after an import from a camera, and new after either an import from a computer, or a real import from thumbnail into HQ pictures. After a reconstruction, pictures can be in the state **PROCESSED** or **REJECTED** depending of the result of the openMVG computes. Only **NEW** and **PROCESSED** pictures are taken for a reconstruction. During the reconstruction, they are in the state **RECONSTRUCTION**. Finally, pictures can be manually discarded to go in the state **DISCARDED**. It allows user to discard thumbnail he doesn't want to import, or to discard photos that should not be used for the reconstruction, even if they are ready. Also, GPS coordinates of a picture are obtained from the EXIF data of a picture, using a python implementation of the tool exiftool.

Picture manager

It is really common that the model and the model manager are gathered into one single component. The model is a specific component that holds data and gives simple accessors on those data. A manager, is typically another component that supplies high-level methods to manipulate and retrieve data from the model. When models are quite simple, they may be fusionned with the manager. As a result, it is almost our case.

In the application, we've define a manager as an upper layer, upon the model described previously. What we called manager is a QSortFilterProxyModel, and this is the component supplied to the different QML views. This manager allow us to create proxy model, i.e. models that are subsets from our real model. The main purpose is to have an easy control on sorting and filtering element inside the model. As we defined several states for each pictures, they could be filtered using this state. Thus, instead of instanciating a new model each time a filter is required, we use a proxy model that re-use the real model and simplify a part of the work. Therefore, some accessors are directly defined on the manager to allow QML's component to retrieve some information directly from the model, without emitting a slot.

Also, we've experimented some issues using the proxy model. Indeed, re-instantiating a proxy model to replace an existing one in the QML's side causes a core dumped. The reason is still unknown. In order to change the source model, you would rather use appropriate method of the proxy model.

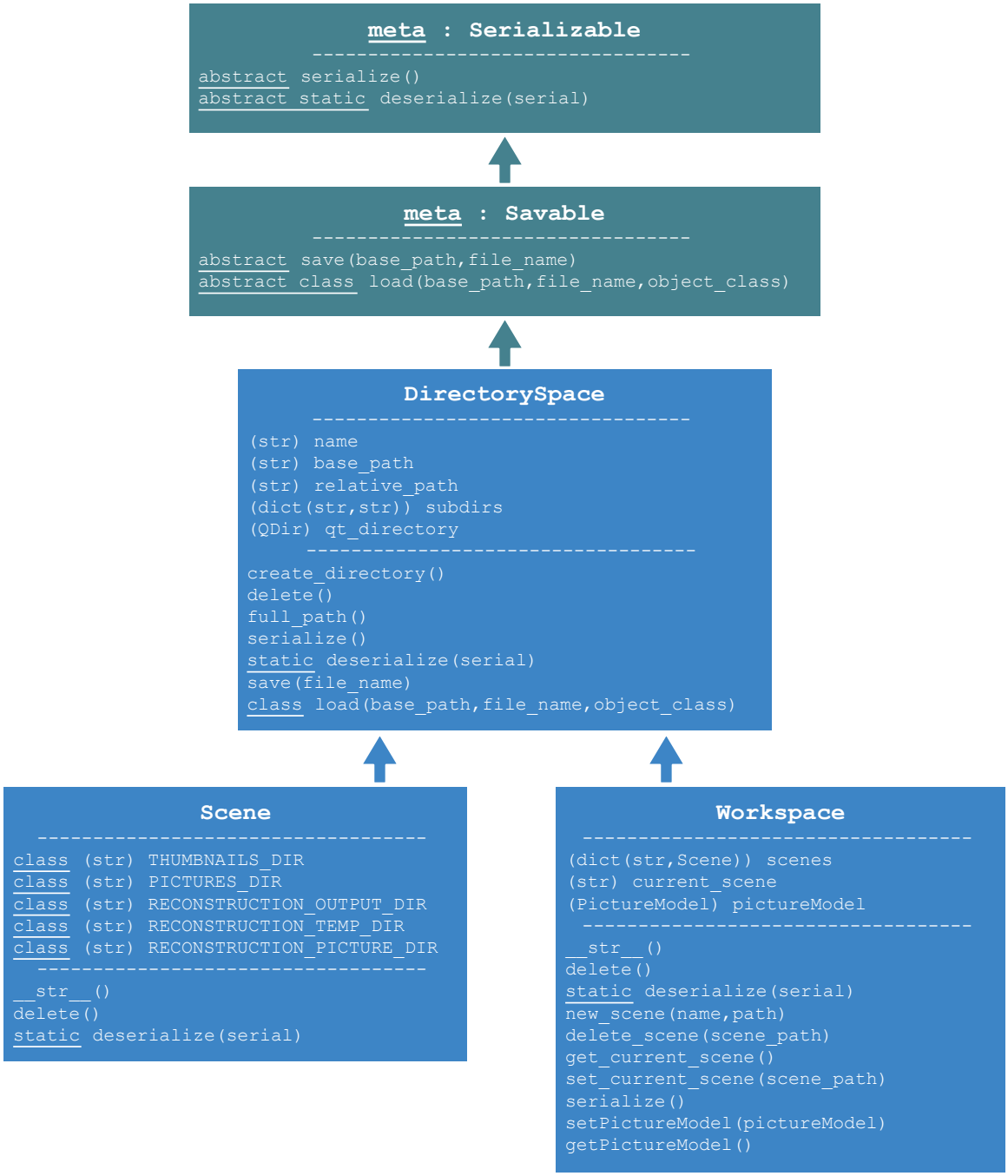
3.3.3 Workspace Manager

The WorkspaceManager package is made of 4 different classes :

- WorkspaceManager
- DirectorySpace
- Workspace
- Scene

The WorkspaceManager class is serving as an interface between the ochestrator and the effective workspaces and scenes.

Internally the three other classes (DirectorySpace, Workspace and Scene) handle the creation, manipulation and destruction of workspaces and scenes. In order to make a DRY (Don't Repeat Yourself) code, every feature shared between workspaces and scenes is in the parent class DirectorySpace. You have a simple class diagram of the workspaces and scenes in the figure below :



Attention: Normally, the core features of workspaces and scenes were tested many times and are stable. However, during the last two days before the demo, some important features needed to interact with external modules (pictureManager, reconstructionManager, ...) were added without the time to do meticulous testing. Thus it is possible that some bugs occur while manipulating scenes in the application. But we are confident that such bugs might be easily corrected.

3.3.4 Recontruction Manager

This section describes several things about the module used to launch the recontruction. This module is included in the project as a git submodule and has it's own github repository (OpenMVGFeeder). Right now, the reconstruciton manager has two versions (the older is the one included in the final version of the application).

Interface with the openMVG command line

The reconstruction manager is simply a front end to the openMVG command line. It is very similar with the Picture Fetcher module : we also use the subprocess submodule from python to call directly openMVG through the command line.

This version is pretty basic and doesn't handle many things we would have wanted to implement like detection of new point cloud file. That's why there is another version, but we didn't have the time to integrate it in our last version of the application.

Recontruction Manager improvements

In this improvement, we first added a function, `pointCloudDetected` able to emit a signal when a new point cloud is detected in a specific folder (given as a parameter).

When the orchestrator creates a reconstruction manager, it creates a new object with default atttributes :

- A list of `.ply` files it already has : initialize empty.
- A new signal called *newPointCloud*
- A new *QFileSystemWatcher_* initially watching nothing and that we connect with the pointCloudDetected (described previously) slot.

Then, when we launch a new recontruction, we call the `launchReconstruction` function. This function is working like in the first version, except that every time it is called, we add the openMVG output directory to the list of folder watched by the file watcher. Every time a new point cloud is detected in this folder, we copy it in the output folder used by the 3D renderer (the path is given as a parameter).

3.3.5 3D Rendering Module

The 3D renderer is C++ module that has to be compile as a QML plugin to be used in the application. The communication between the module and the orchestrator can only be made through QML. To display openGL under QML, we used the QML scene graph.

openGL implementation and `.ply` reader

The C++ code for the openGL implementation is based on this tutorial . There are many differences as we had to use this module in a Qt context : we had to convert every C++ openGL function in Qt openGL functions.

Those are really often the same function, only the name changes. Only Buffer Object and Vertex Array Object are different since they are Qt object that we have to create before using them.

The `.ply` reader can be found in an old commit made in the openMVG repository. This reader is the one initializing the position and color buffers of the renderer.

When a point cloud has been read by the reader, we initialze a list of camera coordinates simply by reading the end of the position buffer (because cameras are found at the end of the ply file). We then initialize the default camera to the first camera : that's the default point of view used for the renderer. We can switch this point of view, in the camera list order, simply by calling the nextCam function. The renderer is still pretty basic, but what is interesting here is the conversion between the C++ openGL implementation to a Qt implementation. We then describe how to use this renderer as a QML plugin. The `pathply` attribute and the `nextCam` function are two good examples of interactions between QML and the plugin, and are two ways of communicating (by setting attributes directly from QML, or by calling a C++ function also directly from QML).

The QML Plugin

To create a new QML plugin, we followed the QT documentation. The plugin is in the `PointCloud.cpp` file as a new class extending QQmlExtensionPlugin. As said before, we used the QML scene graph to display openGL under QML. As explained in the tutorial, we had to separate the PointCloud, which lives in the GUI thread, and the PointCloudRenderer, which lives in the rendering thread. When the beforeRendering signal from the window is emitted , at the start of every frame before the window rendering, any OpenGL draw calls that are made as a response to this signal will stack under the Qt Quick items. We then simply connect the `beforeRendering` signal to the renderer paint function to paint the PointCloud.

Communication between QML and the plugin

With our plugin, you can either communicate by directly calling C++ function from QML, or by dynamically setting attribute in QML. We implemented both ways of communication :

- You can set in the QML PointCloud object a path to a ply file used for the rendering. The communication is not made directly from QML to the renderer, you have to use the PointCloud to communicate. To set the ply path, we added a setter and a getter, and also a Qt property in order to make QML aware on how to set and get the attribute.
- You can call directly from QML the nextCam function. To do that, you also have to go through the point cloud object and not directly call the renderer function. You have to add a Qt invokable function which calls the renderer function.

Note that you really have to be aware of the whole logic behind the scene graph in order to synchronize everything. You especially have to use the QQuickItem function like `sync` to be able to stack the openGL calls under the Qt Quick item.

CHAPTER

FOUR

CODE DOCUMENTATION

Here comes the semi-automatically generated code documentation. For now the commands used are :

```
sphinx-apidoc -f -o source/code ../MatrixGUI
sphinx-apidoc -f -o source/code/Python ../MatrixGUI/Components/Python
sphinx-apidoc -f -o source/code/PyQt ../MatrixGUI/Components/PyQt
```

Note: In order to understand google docstring format you have to use `napoleon` (see http://sphinx-doc.org/ext/napoleon.html#module-sphinx.ext.napoleon) This requires Sphinx 1.3.

4.1 MatrixGUI

4.1.1 orchestrator module

class orchestrator.**Orchestrator**

Bases: *orchestratorSlots.OrchestratorSlots*

MAIN_VIEW = '/home/matthieu/GIT/ENSEEIHT/3A/PL_POPART/MATRIX_matthieu/docs/MainWindow.qml'

OPENMVG_BUILD_DIR = '/home/matthieu/GIT/ENSEEIHT/3A/PL_POPART/MATRIX_matthieu/docs/Components/PyQ

QML_PACKAGE = '/home/matthieu/GIT/ENSEEIHT/3A/PL_POPART/MATRIX_matthieu/docs/Components/QML'

QML_PLUGIN = '/home/matthieu/GIT/ENSEEIHT/3A/PL_POPART/MATRIX_matthieu/docs/Components/QML/3dRend

RESOURCES = '/home/matthieu/GIT/ENSEEIHT/3A/PL_POPART/MATRIX_matthieu/docs/Resources'

connectEverything()

Every connections between slots and signals are done here

4.1.2 orchestratorSlots module

class orchestratorSlots.**OrchestratorSlots**

Bases: PyQt5.QtCore.QObject

cameraConnection(*isConnected*)

change_scene(*path*)

change_workspace(*path*)

close_workspace(*path*)

confirmThumbnails()

deletePictures(*indexes*)
A slot that handle picture deleting

Parameters **indexes** (*list<QVariant>*) – Indexes of pictures to delete

delete_scene(*path*)

delete_workspace(*path*)

discardPictures(*indexes*)
A slot that handle picture discarding

Parameters **indexes** (*list<QVariant>*) – Indexes of pictures to discard

filterPictures(*status*)
A slot that handle filtering within pictures

Parameters **status** (*int*) – The status that should be filtered, according to PictureState

importPictures(*picturesFiles*)
A slot that handle the picture import from a camera

Args: picturesFiles (list<QUrl>): The list of pictures to be imported

importThumbnails()

launchReconstruction()

movePictures(*indexes*, *startIndexTo*)
A slot that handle the reorganization between pictures. If more than one index are supplied, the first pictures selected will be moved at indexTo, and the other will be appended

Args: indexes (list<int>): Indexes to be moved startIndexTo (int): The destination start index of all pictures

newPictures(*newPictures*, *deletedPictures*)
Handle an update from the camera to manage new or deleted pictures.

Args: newPictures (list<str>): A list of newly found pictures deletedPictures (list<str>): A list of all previously existing pictures now deleted by user

new_scene(*name*)

new_workspace(*name*, *path*)

onCameraConnection

open_workspace(*path*)

picturesUpdated

reconstructionChanged

renewPictures(*indexes*)
A slot that handle picture renewing, i.e, that allow rejected or discarded pictures to be used again

Parameters **indexes** (*list<QVariant>*) – Indexes of pictures to renew

save_workspace()

workspaceAvailable

4.2 Python

4.2.1 Persistence package

Submodules

Persistence.Savable module

class Persistence.Savable.**Savable**
Bases: Serializable.Serializable

An object that can be saved and loaded in the file system.

classmethod load(*base_path*, *file_name*, *object_class*)
Recreate an object from the file

Parameters

- **base_path** (*str*) – The path of the directory containing the file.
- **file_name** (*str*) – The name of the file to load.
- **object_class** (*class*) – The class of the object to recreate.

save(*base_path*, *file_name*)
Save the object in the file system.

Persistence.Serializable module

class Persistence.Serializable.**Serializable**
Bases: object

static deserialize(*serial*)

serialize()

Persistence.Utils module

class Persistence.Utils.**Utils**
Bases: object

Useful functions.

static valid_name(*s*)
Transform a string in order to get a valid filename

This method may produce invalid filenames such as ”.”

Module contents

4.3 PyQt

4.3.1 PictureFetcher package

Submodules

PictureFetcher.pygphoto module

class `PictureFetcher.pygphoto.`**`Pygphoto`**(*watch_camera=False*, *watch_files=False*)
Bases: `PyQt5.QtCore.QObject`

Allows simple operations on a USB connected camera by interfacing the gphoto2 command line tool.

This class allows to interact with a USB camera. List the names of the photos present in the camera, watch for new files, and eventually download the photos individually. Needs a QApplication to be instantiated in order to watch properly for events.

Parameters

- **`watch_camera=False`** (*bool*) – Should the daemon thread watch for camera connection.
- **`watch_files=False`** (*bool*) – Should the daemon thread watch for files creation/deletion.

`check_camera_connected`()

Returns A boolean indicating the presence of a connected camera.

Raises `CalledProcessError` – when gphoto2 raised an error.

`download_all`(*output_dir*, *overwrite=True*, *thumbnail=False*)
Download all the files present on the camera.

Overwrites preexisting files. Faster than 'download_files()'.

Parameters

- **`output_dir`** (*str*) – The directory where the files should be downloaded to.
- **`overwrite=True`** (*bool*) – If any existing file with the same name should be overwritten.
- **`thumbnail=False`** (*bool*) – Download the thumbnail instead of the file.

Returns The return code returned by the gphoto2

`download_file`(*filename*, *output_dir*, *overwrite=True*, *thumbnail=False*)
Download the file name "filename" and copy it to the given path.

Parameters

- **`filename`** (*str*) – The name of the file to download.
- **`output_dir`** (*str*) – The directory where the files should be downloaded to.
- **`overwrite=True`** (*bool*) – If any existing file with the same name should be overwritten.
- **`thumbnail=False`** (*bool*) – Download the thumbnail instead of the file.

Returns 0 if succeeded. Else returns the error code returned by gphoto.

download_files(*filename_list*, *output_dir*, *overwrite=True*, *thumbnail=False*)
Download the whole list of files to the ouput directory

This is equivalent to calling download_file on every file in the "filename_list", but should be faster for a large number of files.

Parameters

- **filename_list** (*list*) – The name list of files to download.
- **output_dir** (*str*) – The directory where the files should be downloaded to.
- **overwrite=True** (*bool*) – If any existing file with the same name should be overwritten.
- **thumbnail=False** (*bool*) – Download the thumbnail instead of the file.

Returns **files_list** – The paths list of all downloaded files.

Return type list

onCameraConnection
`pyqtSignal(bool)` This signal indicates if a camera is connected.

onContentChanged
`pyqtSignal(list, list)` When watching the camera for new files, emit this signal when there has been some changes in the camera filesystem. Arguments are the lists of new files and deleted files

query_camera_name()

Returns The camera model name, or None.

query_file_list()
Generate the list of filenames for all the files present on the first camera found by requesting directly the camera.

Returns The list of all filenames present on the camera.

Raises `CalledProcessError` – when gphoto2 raised an error.

query_storage_info()

Returns A dictionnary of values concerning memory usage {free, occupied, total} containing values in KB.

Raises `CalledProcessError` – when gphoto2 raised an error.

set_watching_camera(*value*)
Set whether the component should watch for presence of a connected camera.

Parameters **value** (*bool*) – True for watching.

set_watching_files(*value*)
Set whether the component should watch for changes in the camera filesystem.

Parameters **value** (*bool*) – True for watching.

Module contents

4.3.2 PictureManager package

Submodules

PictureManager.exiftool module

PyExifTool is a Python library to communicate with an instance of Phil Harvey's excellent ExifTool command-line application. The library provides the class *ExifTool* that runs the command-line tool in batch mode and features methods to send commands to that program, including methods to extract meta-information from one or more image files. Since `exiftool` is run in batch mode, only a single instance needs to be launched and can be reused for many queries. This is much more efficient than launching a separate process for every single query.

The source code can be checked out from the github repository with

```
git clone git://github.com/smarnach/pyexiftool.git
```

Alternatively, you can download a tarball. There haven't been any releases yet.

PyExifTool is licenced under GNU GPL version 3 or later.

Example usage:

```
import exiftool

files = ["a.jpg", "b.png", "c.tif"]
with exiftool.ExifTool() as et:
    metadata = et.get_metadata_batch(files)
for d in metadata:
    print("{:20.20} {:20.20}".format(d["SourceFile"],
                                    d["EXIF:DateTimeOriginal"]))
```

class `PictureManager.exiftool.`**`ExifTool`**(*executable_=None*)

Bases: `object`

Run the *exiftool* command-line tool and communicate to it.

You can pass the file name of the `exiftool` executable as an argument to the constructor. The default value `exiftool` will only work if the executable is in your `PATH`.

Most methods of this class are only available after calling *start()*, which will actually launch the subprocess. To avoid leaving the subprocess running, make sure to call *terminate()* method when finished using the instance. This method will also be implicitly called when the instance is garbage collected, but there are circumstance when this won't ever happen, so you should not rely on the implicit process termination. Subprocesses won't be automatically terminated if the parent process exits, so a leaked subprocess will stay around until manually killed.

A convenient way to make sure that the subprocess is terminated is to use the *ExifTool* instance as a context manager:

```
with ExifTool() as et:
    ...
```

> **Warning:** Note that there is no error handling. Nonsensical options will be silently ignored by exiftool, so there's not much that can be done in that regard. You should avoid passing non-existent files to any of the methods, since this will lead to undefied behaviour.

running
A Boolean value indicating whether this instance is currently associated with a running subprocess.

execute(**params*)
Execute the given batch of parameters with `exiftool`.

This method accepts any number of parameters and sends them to the attached `exiftool` process. The process must be running, otherwise `ValueError` is raised. The final `-execute` necessary to actually run the batch is appended automatically; see the documentation of *`start()`* for the common options. The `exiftool` output is read up to the end-of-output sentinel and returned as a raw `bytes` object, excluding the sentinel.

The parameters must also be raw `bytes`, in whatever encoding exiftool accepts. For filenames, this should be the system's filesystem encoding.

Note: This is considered a low-level method, and should rarely be needed by application developers.

---

execute_json(**params*)
Execute the given batch of parameters and parse the JSON output.

This method is similar to *`execute()`*. It automatically adds the parameter `-j` to request JSON output from `exiftool` and parses the output. The return value is a list of dictionaries, mapping tag names to the corresponding values. All keys are Unicode strings with the tag names including the ExifTool group name in the format <group>:<tag>. The values can have multiple types. All strings occurring as values will be Unicode strings. Each dictionary contains the name of the file it corresponds to in the key `"SourceFile"`.

The parameters to this function must be either raw strings (type `str` in Python 2.x, type `bytes` in Python 3.x) or Unicode strings (type `unicode` in Python 2.x, type `str` in Python 3.x). Unicode strings will be encoded using system's filesystem encoding. This behaviour means you can pass in filenames according to the convention of the respective Python version – as raw strings in Python 2.x and as Unicode strings in Python 3.x.

get_metadata(*filename*)
Return meta-data for a single file.

The returned dictionary has the format described in the documentation of *`execute_json()`*.

get_metadata_batch(*filenames*)
Return all meta-data for the given files.

The return value will have the format described in the documentation of *`execute_json()`*.

get_tag(*tag*, *filename*)
Extract a single tag from a single file.

The return value is the value of the specified tag, or `None` if this tag was not found in the file.

get_tag_batch(*tag*, *filenames*)
Extract a single tag from the given files.

The first argument is a single tag name, as usual in the format <group>:<tag>.

The second argument is an iterable of file names.

The return value is a list of tag values or `None` for non-existent tags, in the same order as `filenames`.

get_tags(*tags*, *filename*)
Return only specified tags for a single file.

The returned dictionary has the format described in the documentation of *`execute_json()`*.

get_tags_batch(*tags*, *filenames*)
Return only specified tags for the given files.

The first argument is an iterable of tags. The tag names may include group names, as usual in the format <group>:<tag>.

The second argument is an iterable of file names.

The format of the return value is the same as for *execute_json()*.

start()
Start an exiftool process in batch mode for this instance.

This method will issue a UserWarning if the subprocess is already running. The process is started with the -G and -n as common arguments, which are automatically included in every command you run with *execute()*.

terminate()
Terminate the exiftool process of this instance.

If the subprocess isn't running, this method will do nothing.

PictureManager.exiftool.**executable = 'exiftool'**
The name of the executable to run.

If the executable is not located in one of the paths listed in the PATH environment variable, the full path should be given here.

PictureManager.exiftool.**fsencode**(*filename*)
Encode filename to the filesystem encoding with 'surrogateescape' error handler, return bytes unchanged. On Windows, use 'strict' error handler if the file system encoding is 'mbcs' (which is the default encoding).

PictureManager.pictureManager module

class PictureManager.pictureManager.**MetaPictureModel**
Bases: PyQt5.QtCore.pyqtWrapperType, Savable.Savable

class PictureManager.pictureManager.**Picture**(*resourcesPath*, *path*, *latitude*, *longitude*, *date=None*, *status=0*)
Bases: object

A container used to store all data about a particular picture. It reflects an xml structure and is used to manipulate photos as dataModel along the use of the application

circleColor

icon

serialize()
Serialize a Picture object.

class PictureManager.pictureManager.**PictureManager**
Bases: PyQt5.QtCore.QSortFilterProxyModel

computeCenter()
Compute the coordinates of the center associated to all pictures

Returns The coordinates, latitude then longitude

Return type list<float>

count()
An alias to be used in QML as a property. We can"t use the rowCount method as it should remain available inside the class.

Returns The number of element in the model

Return type int

deleteAll(*rows*)
Remove pictures from the model

Parameters **rows** (*list<QModelIndex>*) – The related rows in the proxy model

discardAll(*rows*)
Change the status of pictures to DISCARDED or THUMBNAIL_DISCARDED

Parameters **rows** (*list<QModelIndex>*) – The related rows in the proxy model

getName(*index*)

move(*initRow*, *finalRow*)
Move a row from an index to another. Exactly, put initRow after finalRow if moving up to down, and before finalRow if moving down to up

Parameters

- **initRow** (*QModelIndex*) – The starting index
- **finalRow** (*QModelIndex*) – The final index

renewAll(*rows*)
Change the status of pictures DISCARDED, THUMBNAIL_DISCARDED or REJECTED to NEW

Parameters **rows** (*list<QModelIndex>*) – The related rows in the proxy model

class PictureManager.pictureManager.**PictureModel**(*resourcesPath*, *listPictures=[]*, *parent=None*)
Bases: PyQt5.QtCore.QAbstractListModel

Represent and handle a list of pictures as a ListModel. Directly implements QAbstractListModel.

PATH_ROLE
int

Role that handle the picture's path name of an item

NAME_ROLE
int

Role that handle the picture's name of an item

DATE_ROLE
int

Role that handle the picture's date of creation

STATUS_ROLE
int

Role that handle the picture's status of an item

ICON_ROLE
int

Role that handle the picture's icon of an item

ITEM_ROLE
int

Role related to the whole item / picture

LATITUDE_ROLE
int

Role related to the latitude of an item

LONGITUDE_ROLE
int

Roled related to the longitude of an item

COLOR_ROLE
int

Role that handle the color of an item on the map

COLOR_ROLE = 264

DATE_ROLE = 259

ICON_ROLE = 261

ITEM_ROLE = 306

LATITUDE_ROLE = 262

LONGITUDE_ROLE = 263

NAME_ROLE = 258

PATH_ROLE = 257

STATUS_ROLE = 260

add(*picture*, *index=None*)
Add a picture to the model

picture – The picture to add index – The index where the picture should be inserted. If None, the picture will be appended at the end.

data(*index*, *role=258*)
Retrieve a piece of information from an item (a picture) of the model

Parameters

- **index** (*int*) – The index of the element
- **role** (*int*) – The role of the element we are interested in

Returns The requested element or data related to this element.

Return type QVariant

static deserialize(*serial*)
Recreate a pictureModel object from its serialization.

Parameters **serial** (*dict()*) – The serialized version of a pictureModel object.

insertRow(*row*, *parent=<PyQt5.QtCore.QModelIndex object>*)
An implementation of the parent method insertRow. It add an empty row at the given index See Qt Documentation for more details <3.

Parameters

- **row** (*int*) – The index of tization between pictures. If more than one index are supplied, the first he future row. If superior to model size, the row will be appended.
- **parent** (*QModelIndex*) – The parent index, always default in our case.

Returns Return True if the row have been successfully inserted.

Return type bool

instantiateManager()
Create an instance of this model manager. The manager add an indirection that allow, for instance, filtering.

Returns The picture manager corresponding to that model

Return type *PictureManager*

classmethod load(*base_path*, *file_name*, *object_class=None*)
Recreate a pictureModel object from a file.

Parameters

- **base_path** (*str*) – The path of the directory containing the file.
- **file_name** (*str*) – The name of the file to load.
- **object_class** (*class*) – The class of the object to recreate.

populate(*picturesFiles*, *status=0*)
Populate the model, i.e. add instance of pictures element. Element are added with the status "NEW"

Parameters

- **picturesFiles** (*list<str>*) – List of path to the different pictures
- **status** (*int*) – The initial status to assign to the item

printData()

removeDiscardedThumbnails()

removeRow(*row*, *parent=<PyQt5.QtCore.QModelIndex object>*)
Remove a picture from the model

row – The picture"s index

removeRows(*row*, *count*, *parent=<PyQt5.QtCore.QModelIndex object>*)
Remove contiguous pictures from the model

Parameters

- **row** (*int*) – The first picture's index
- **count** (*int*) – Number of picture to remove
- **parent** (*QModelIndex*) – The parent row

roleNames()
An accessor to roles names

rowCount(*parent=<PyQt5.QtCore.QModelIndex object>*)
Return the number of element within that model

save(*base_path*, *file_name*)
Save the object in the file system.

serialize()
Serialize a pictureModel object.

setData(*index*, *value*, *role*)
Set a particular value in the data model using his role.

index – The index of the related element value – The new value role – The related role

thumbnails()

validFiles()
Retrieve all files that may be used for the reconstruction; i.e. with status : NEW or PROCESSED

Returns The list of all valid pictures in that model

Return type list<Picture>

class PictureManager.pictureManager.**PictureState**
Bases: object

An Enumeration to handle all different state in which a picture may be

DISCARDED = 6

NEW = 0

PROCESSED = 3

RECONSTRUCTION = 1

REJECTED = 2

THUMBNAIL = 4

THUMBNAIL_DISCARDED = 5

Module contents

4.3.3 WorkspaceManager package

Submodules

WorkspaceManager.DirectorySpace module

class WorkspaceManager.DirectorySpace.**DirectorySpace**(*name=''*, *base_path=''*, *relative_path=''*)
Bases: Savable.Savable

A space based on a directory in the file system.

name
str

The name of the space.

base_path
str

The absolute path of the directory containing the space.

relative_path
str

The path of the directory space relatively to base_path.

subdirs
dict(str,str)

The useful subdirectories (path,name). The paths are the keys

qt_directory
QDir

The directory corresponding to that workspace

create_directory()
Effectively creates the directory thanks to Qt object QDir

delete()
Delete the directory space (and its contents).

Raises AssertionError – If the directory does not exist or can not be deleted.

static deserialize(*serial*)

full_path()

classmethod load(*base_path*, *file_name*, *object_class=None*)
Recreate a DirectorySpace object from a file.

Parameters

- **base_path** (*str*) – The path of the directory containing the file.
- **file_name** (*str*) – The name of the file to load.
- **object_class** (*class*) – The class of the object to recreate.

save(*file_name*)
Save the object in the file system.

serialize()

WorkspaceManager.Scene module

class WorkspaceManager.Scene.**Scene**(*name*, *base_path*, *relative_path=''*)
Bases: DirectorySpace.DirectorySpace

A scene containing all its images.

Class Attributes: RECONSTRUCTION_OUTPUT_DIR RECONSTRUCTION_TEMP_DIR RECONSTRUCTION_PICTURE_DIR

name
str

The name of the scene. It must be unique in the workspace. For example : "Shooting ENSEEIHT". Default is "scene_n" where n is the current number of scenes in workspace.

base_path
str

The path of the scene's workspace.

relative_path
str

The path of the scene relatively to base_path.

subdirs
dict(str,str)

The useful subdirectories (path,name).

PICTURES_DIR = 'pictures_set'

RECONSTRUCTION_OUTPUT_DIR = 'reconstruction_output'

RECONSTRUCTION_PICTURE_DIR = 'reconstruction_pictures'

RECONSTRUCTION_TEMP_DIR = 'reconstruction_temp'

THUMBNAILS_DIR = 'thumbnails'

__str__()
Change displaying of a scene.

Example

```
>>> print(scene)
```

delete()
Delete the scene and remove its access from the workspace.

It must not be called by itself but by the workspace !

static **deserialize**(*serial*)
Regenerate a Scene object from serialized data (JSON like).

Parameters **serial** (*dict()*) – The serialized version of a Scene object.

get_reconstruction_out_dir()

get_reconstruction_picture_dir()

get_reconstruction_temp_dir()

get_thumbnails_dir()

WorkspaceManager.Workspace module

class WorkspaceManager.Workspace.**Workspace**(*name=''*, *base_path=''*, *relative_path=''*)
Bases: DirectorySpace.DirectorySpace

A workspace containing its own configuration and scenes.

name
str

The name of the workspace.

base_path
str

The absolute path of the directory containing the workspace.

relative_path
str

The path of the workspace relatively to base_path.

subdirs
dict(str,str)

The useful subdirectories (path,name).

scenes
dict(str,Scene)

The dictionnary of the scenes it contains. Keys are the scene paths.

current_scene
str

The path of the current scene.

qt_directory
QDir

The directory corresponding to that workspace.

pictureModel
PictureModel

A model for the view.

__str__()
Change displaying of a workspace.

Example

```
>>> print(workspace)
```

delete()
Delete the workspace (and its contents).

It must not be called by itself but by the workspace manager !

Raises `AssertionError` – If the workspace directory does not exist or can not be deleted.

delete_scene(*scene_path*)
Delete the scene identified by its local path.

Parameters **scene_path** (*str*) – The path (relatively to the workspace) of the scene to delete.

static deserialize(*serial*)
Recreate a Workspace object from its serialization.

Parameters **serial** (*dict()*) – The serialized version of a Workspace object.

getPictureModel()

get_current_scene()
Get the current scene of the workspace.

Returns The current scene.

Return type *Scene*

Raises

- `AssertionError` – If no current scene.
- `AssertionError` – If the current scene has disappeared.

new_scene(*name=''*, *path=''*)
Add a new scene to the workspace.

Parameters

- **name** (*str*) – The name of the new scene
- **path** (*str*) – The local path of the scene (inside workspace)

Raises `AssertionError` – If a scene with the same path already exists.

serialize()
Serialize a Workspace object.

setPictureModel(*pictureModel*)

set_current_scene(*scene_path*)
Set the current scene.

Parameters **scene_path** (*str*) – The relative path of the scene which will be the current scene.

Raises `AssertionError` – If the scene does not exist in the workspace.

WorkspaceManager.WorkspaceManager module

class `WorkspaceManager.WorkspaceManager.`**WorkspaceManager**(*pictureModel*)
Bases: `object`

Manages the workspace.

Will handle all interactions between user and modules and the working space. This class is responsible for managing files inside the workspace and for communicating any change through signals.

workspaces
dict(str,Workspace)

All the workspaces.

current_workspace
str

The current workspace.

workspaces_model
QStringListModel

The list model (for qml) of the workspaces.

scenes_model
QStringListModel

The list model (for qml) of the scenes in the current workspace.

pictureModel
PictureModel

The picture model

change_scene(*scene_path*)
Change the current scene identified by its path in the current workspace.

Parameters **scene_path** (*str*) – The relative path of the scene to select in the current workspace.

Raises `AssertionError` – If the scene directory does not exist.

change_workspace(*workspace_path*)
Change the current workspace.

Parameters **workspace_path** (*str*) – The absolute path of the workspace to select.

close_workspace(*workspace_path*)
Close the workspace (it is not visible anymore in the list of open workspaces. :param workspace_path: The absolute path of the workspace to close.

delete_scene(*scene_path*)
Delete the scene identified by its path in the current workspace.

Parameters **scene_path** (*str*) – The relative path of the scene to delete in the current workspace.

Raises AssertionError – If the scene directory does not exist or can not be deleted.

delete_workspace(*workspace_path*)
Delete the workspace identified by its abolute path.

Parameters **workspace_path** (*str*) – The absolute path of the workspace to delete.

Raises AssertionError – If the workspace directory does not exist or can not be deleted.

getPictureModel()

get_current_scene()

get_current_workspace()
Get the current workspace.

Returns The current workspace.

Return type *Workspace*

Raises

- AssertionError – If no current workspace.
- AssertionError – If the current workspace has disappeared.

get_picture_dir()
Retrieve the folder where all pictures are stored

Returns The absolute path to the folder

Return type str

get_scene_output_dir()
Returns the absolute path of the output directory for ply files.

Returns The absolute path of directory.

Return type str

get_scene_temp_output_dir()
Returns the temporary output directory for scene reconstructions.

Returns The absolute path of the temporary directory.

Return type str

get_selected_picture_dir()
Returns the absolute path of the directory containing the pictures used for reconstruction.

Returns The absolute path of the picture directory.

Return type str

get_thumbnails_dir()

import_pictures(*picturesPath*)
Import pictures from an external location into the workspace :param picturesPath: The list of path to import

new_scene(*name=''*, *path=''*)
Create a new scene in the current workspace.

Parameters

- **name** (*str*) – The name of the scene.
- **path** (*str*) – The relative path of the scene in the workspace.

Raises AssertionError – If there is no current workspace or the scene already exists.

new_workspace(*name=''*, *base_path=''*)
Create a new workspace.

Parameters

- **name** (*str*) – The name of the workspace.
- **base_path** (*str*) – The absolute path of the directory that will contain the new workspace.

open_workspace(*directory_path*, *file_name*)
Open an existi ng workspace from a save file of the workspace.

Parameters

- **directory_path** (*str*) – The absolute path of the directory containing the file.
- **file_name** (*str*) – The name of the file containing the save of the workspace.

prepare_reconstruction(*picturesList*)
Copy all file describe in picturesList inside the temporary reconstruction folder

Parameters **picturesList** (*list<Picture>*) – The list of pictures that should be copied

save_workspace(*workspace_path*, *file_name='workspaceSettings'*)
Save the workspace in a file.

Parameters

- **workspace_path** (*str*) – The absolute path of the workspace to save.
- **file_name** (*str*) – The name of the file to save into.

setPictureModel(*pictureModel*)

set_current_scene(*scene_path*)
Change the current scene identified by its path in the current workspace.

Parameters **scene_path** (*str*) – The relative path of the scene to select in the current workspace.

Raises AssertionError – If the scene directory does not exist.

set_current_workspace(*workspace_path*)
Set the current workspace.

Parameters **workspace_path** (*str*) – The absolute path of the workspace which will be the current workspace.

Raises `AssertionError` – If the workspace does not exist in the workspace manager.

update_scenes_model()
Updates the attribute scenes_model.

update_workspaces_model()
Updates the attribute workspaces_model.

Module contents

CHAPTER

FIVE

BUILDING THAT DOCUMENTATION

The objective is to have an easily maintainable and yet simple and powerful documentation. For all these reasons we chose Sphinx. Sphinx has many benefits :

- A simple syntax (rST)
- Provides a simple navigation
- Manages source code blocks
- Manages source code doc comments
- Manages images
- Manages mathematics
- Manages LaTeX compilation

You will find an introduction to rST syntaxe here : http://sphinx-doc.org/rest.html

5.1 Sphinx Installation

You can find all the details here : http://sphinx-doc.org/install.html. The simplest way is to use Python package manager (pip) :

```
$ sudo pip install sphinx
```

Attention: If you use some Ubuntu derived distribution, you might need to use `pip3` instead of `pip`.

5.2 Initialize the documentation

Sources of the documentation will reside in the `docs` folder. Use the `sphinx-quickstart` command to start the documentation :

```
$ mkdir docs
$ cd docs/
$ sphinx-quickstart
```

The default configuration is almost ok but we need to change some answers :

```
> Separate source and build directories (y/n) [n]: y
> autodoc: automatically insert docstrings from modules (y/n) [n]: y
> mathjax: include math, rendered in the browser by MathJax (y/n) [n]: y
```

At the end of the script, Sphinx configuration has been initialized. It has setup 2 folders : `build` and `source`. It also created a `Makefile` allowing to easily build the doc with commands such as :

```
$ make html
```

5.3 Configuration of Sphinx documentation

The `sphinx-quickstart` script generated files allowing us to configure Sphinx. Here are some additional information to setup your documentation.

5.3.1 Host your documentation on Github

If you have a Github repository you can host your documentation on the Github pages of your repository. See https://help.github.com/articles/what-are-github-pages/ for more explanations on it.

To activate the Github pages of your repository you only have to create a new branch called `gh-pages` and to push it on origin.

```
$ git checkout --orphan gh-pages
$ git rm -rf .
$ echo "My page" > index.html
$ git add index.html
$ git commit -am "Premier commit sur les pages Github"
$ git push origin gh-pages -u
```

5.3.2 Configure HTML build

In practice we need to build into the `gh-pages` branch. But it is not immediatly possible since we are not `gh-pages` branch when compiling.

The easiest solution according to me is to have 2 folders. One with the standard repository, and the other one with only the `gh-pages` branch (so it is lightweight) of the same repository. Then you only have to configure your Sphinx build so that it builds into the folder containing the repository on the `gh-pages` branch.

Example for creating the second folder :

```
$ pwd
/home/........./MATRIX
$ cd ../
$ mkdir MATRIX_gh-pages
$ cd MATRIX_gh-pages/
$ git clone -b gh-pages --single-branch git@github.com:mpizenberg/MATRIX.git .
```

Now we just need to modify the make file `MATRIX/docs/Makefile` :

```
BUILDDIR = ../../MATRIX_gh-pages

html:
   $(SPHINXBUILD) -b html $(ALLSPHINXOPTS) $(BUILDDIR)
```

5.3.3 Use the theme rTD (readTheDocs)

You only need to install the theme with pip.

```
$ sudo pip install sphinx_rtd_theme
```

Then in the file `conf.py` :

```
import sphinx_rtd_theme
html_theme = "sphinx_rtd_theme"
html_theme_path = [sphinx_rtd_theme.get_html_theme_path()]
```

Once it is installed, there is one last manipulation. You have to add an empty file entitled `.nojekyll` in the `gh-pages` branch. Otherwise Github will ignore folders starting with _. It would be a problem since the folder `_static` contains CSS styles and images needed for the theme.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex

o

p

w

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

W