Micmac's Qt tools

May 21, 2015

1 Visual interfaces "vCommands"

1.1 Introduction

Each command of MicMac can be called in a command line prompt with the general syntax:

```
mm3d Command arg1 arg2 ... argn  NameOpt1=Argot1 ...
```

For example, a possible call to the Tapas tool is:

```
mm3d Tapas  RadialStd ".*.PEF" Out=All
```

To help filling arguments for MicMac commands, visual interfaces based on Qt can be launched by adding the letter "v" in front of the command name. For example, a possible call to the Tapas visual interface is:

```
mm3d vTapas
```

An other possible call to the Tapas visual interface is:

```
mm3d vTapas  RadialStd ".*.PEF" Out=All
```

This will fill visual interface with corresponding arguments.

NB: this is also true for some commands in TestLib such as:

```
mm3d TestLib vOriMatis2MM
```

1.2 Compilation and code

Visual interfaces are available with option WITH_QT4 or WITH_QT5 activated.

```
cmake -DWITH_QT4=ON ..
```

or

```
cmake -DWITH_QT5=ON ..
```

If necessary, see CMakeLists.txt.

At revision 5520, code is located in:

```
include/general/visual_mainwindow.h
include/general/visual_buttons.h
src/util/visual_main_window.cpp
src/util/visual_buttons.cpp
src/util/visual_arg_main.cpp
src/util/arg_main.cpp
src/CBinaires/mm3d.cpp
```

1.3 How it works?

Each command has a set of mandatory arguments, and may have a set of optional arguments. Basically, a visual interface is shown by parsing the two lists of mandatory and optional arguments, getting their type, and depending on their type, displaying in a widget the corresponding selection object (ComboBox, buttons, text edition field, button, or SaisieBoxQT, etc.).

This is based on ElInitArgMain method from arg_main.cpp, which is usually called in the main method for each command in MicMac. This method fills the two lists of mandatory and optional arguments, following this syntax:

```
std::vector <char *>  ElInitArgMain
        (
            int argc,char ** argv,
            const LArgMain & LGlob,
            const LArgMain & L1,
            const std::string & aFirstArg = "",
            bool  VerifInit=EIAM_VerifInit,
            bool  AccUnK=EIAM_AccUnK,
            int   aNbArgGlobGlob = EIAM_NbArgGlobGlob
        );
```

LGlob contains the list of mandatory arguments, while L1 has optional arguments.
Adding an argument is usually done like this:

```
int Function_main(int argc,char ** argv)
{
    string aFullName, aOri, aPly, aOut;

    ElInitArgMain
            (
                argc,argv,
                LArgMain()  << EAMC(aFullName,"Full Name (Dir+Pat)")
                            << EAMC(aOri,"Orientation path")
                            << EAMC(aPly,"Ply file"),
                LArgMain()  << EAM(aOut,"Out",true,"Output filename")
                etc.
            );

    etc.
}
```

To transform an existing MicMac function into a function which can be launched in visual mode, one has to specify the type of the arguments. For example, here:

```
int Function_main(int argc,char ** argv)
{
    string aFullName, aOri, aPly, aOut;

    ElInitArgMain
            (
                argc,argv,
                LArgMain()  << EAMC(aFullName,"Full Name (Dir+Pat)",eSAM_IsPatFile)
                            << EAMC(aOri,"Orientation path",eSAM_IsExistDirOri)
                            << EAMC(aPly,"Ply file", eSAM_IsExistFile),
                LArgMain()  << EAM(aOut,"Out",true,"Output filename")
                etc.
            );

    etc.
}
```

The arguments types can be:

- eSAM_IsPatFile, for a pattern string,
- eSAM_IsBool, for a bool,
- eSAM_IsPowerOf2, for an integer power of 2
- eSAM_IsDir, for a directory string

— `eSAM_IsExistDirOri`, for an existing Orientation directory string,
— `eSAM_IsOutputDirOri`, for an Output Orientation directory string,
— `eSAM_IsExistFile`, for an existing file string,
— `eSAM_IsExistFileRP`, for an existing file to be given with a relative path
— `eSAM_IsOutputFile`, for an output file string,
— `eSAM_Normalize`, for a 2d box that has to be normalized (Box2dr),
— `eSAM_NoInit`, for an argument that has not been initialized,
— `eSAM_InternalUse`, for an argument that we don't want to display in the visual interface,
— `eSAM_None`, for a list of strings.

Type has not to be specified for an integer, a float, a point (`Pt2di`, `Pt2dr`), a box terrain (`Box2dr`).

To check if a visual interface has to be launched a global variable `MMVisualMode` is set to `true` in `GenMain` in `src/CBinaires/mm3d.cpp`. When calling `mm3d` for a visual interface, we first run the `Function_main` with its `ElInitArgMain` to fill the visual interface, then we run a second time `mm3d` with `MMVisualMode` set to `false`, to take into account modifications done in the visual interface, and to run the actual process.

At the first call to `Function_main`, we just want to go through `ElInitArgMain`, to show the visual interface, so we need to exit this function without doing the main process. This is why we have to add after `ElInitArgMain`:

```
if (MMVisualMode) return EXIT_SUCCESS;
```

Another small trick is done to enable user to set some arguments directly in the command line (which will be automatically filled in the visual interface). If command line contains more than `mm3d vCommmand`, we initialize arguments with these values by a call to `ElInitArgMain` in `arg_main.cpp`.

```
if(argc > 1)
{
 MMVisualMode = false;
 ElInitArgMain(argc,argv,LGlob,L1,aFirstArg,VerifInit,AccUnK,aNbArgGlobGlob);
 MMVisualMode = true;
}
```

NB: there is a bug in this part, since we check if an argument has been modified in the visual interface, and this state is set to unchanged when we call `ElInitArgMain` twice.

In `ElInitArgMain`, `aFirstArg` is used to set the widget title.

1.4 `visual_MainWindow` class

In `include/general/visual_mainwindow.h`, we define a class derived from `QWidget`, `visual_MainWindow`. A `visual_MainWindow` is mainly composed of 2 `QGridLayout` where mandatory and optional arguments are displayed in rows. Optional arguments are sorted with regard to their name (`cMMSpecArg::NameArg()`). At the widget's bottom, a button "`Run command`" runs the command with the selected arguments (*slot* `onRunCommandPressed`) ; a checkbox "`Show dialog when job is done`" allows user to continue working, and force a dialog to pop up when process is finished.

Depending on the argument's type, specific objects are created in the corresponding row (see `buildUI` method). A label is added systematically at the left (see`add_label` method), using argument comment (for mandatory argument) or name (for optional argument). A tool tip is added for optional arguments with comment, and is shown (when available) by putting cursor over the argument name.

Dealing with files: many commands need several files, and sometimes several directories, which may be located in the same directory. To help choosing these files or directory, we store the first directory (`mLastDir`). We also store this directory in application settings, so that, at next call, the first open dialog is set to the last directory.

Pushing "`Run command`" button, we parse vector `vInputs` that stores the argument (as `cMMSpecArg`), and build command line `aCom` for `mm3d`, adding only arguments that has been changed (see `cMMSpecArg::IsDefaultVal`

1.5 Specific functions: vTapioca, vMalt, vC3DC, vSake

Some functions have a slightly different workflow and behavior than the majority. These functions need to choose between several modes before calling `ElInitArgMain`. This mode is recovered with a `QInputDialog`. See for example `CPP_Tapioca.cpp`.

`vMalt` has also some specific behavior depending on mode (Ortho, UrbanMNE, GeomImage). This is dealt in function `visual_MainWindow::moveArgs` with boolean `_bMaltGeomImg`. `vMalt` has also two small special behaviours, dealt with function `isFirstArgMalt`: disabling mandatory argument edition after choosing mode in `add_select`, and recovering mode in the final command line in `onRunCommandPressed`.

1.6 `BoxClip` and `BoxTerrain`

Some functions need a 2d rectangular selection information (mainly to perform computations on reduced area). Most of the time, argument name are `BoxClip` and `BoxTerrain`. Previously, user had to measure two corners coordinates in image, and sometimes normalize these coordinates, then type argument for example, `BoxClip=[0.13,0.11,0.89,0.87]`. Here, we use a new tool based on `SaisieQT` (see next chapter), called `SaisieBoxQT`, which is launched with "Selection editor" button (see visual_MainWindow::onSaisieButtonPressed). User first choose which image to open, then draw a rectangle by click-n-drag in the image. User can also edit the rectangle selection afterward, by clicking close to a corner and draging it. We must deal here with 3 cases: true image coordinates, normalized image coordinates, and box terrain coordinates. Normalization is specified using `eSAM_Normalize`. Difference between box image and box terrain is done with the argument type (`Box2di` or `Box2dr`). For a box terrain, a `FileOriMnt xml` has to be read to convert image coordinates to terrain coordinates through function `visual_MainWindow::transfoTerrain`.

2 SaisieQT

2.1 Introduction

`SaisieQT` is a Qt application gathering a set of commands designed to mimic and extend the `SaisiePts` X11 tool originally used for measuring data in images for `MicMac`. It is cross-platform. At revision 5520, there is 6 Qt tools: `SaisieMasqQT`, `SaisieAppuisInitQT`, `SaisieAppuisPredicQT`, `SaisieCylQT`, `SaisieBascQT` and `SaisieBoxQT`.

`SaisieMasqQT`, `SaisieAppuisInitQT`, `SaisieAppuisPredicQT`, `SaisieCylQT` and `SaisieBascQT` are designed as independent applications, while `SaisieBoxQT` is, for now, only called through the visual interfaces (it does not output measures in a `xml` file, it only sends data through *slot/signal* connections.

2.2 Compilation and code

SaisieQT tools are available with option `WITH_QT4` or `WITH_QT5` activated.

```
cmake -DWITH_QT4=ON ..
```

or

```
cmake -DWITH_QT5=ON ..
```

If necessary, see `CMakeLists.txt` and `src/SaisieQT/CMakeLists.txt`.

At revision 5520, code is located in:

```
src/uti_phgrm/CPP_SaisieQT.cpp
src/SaisieQT/
include/qt/
```

When building binaries, one has to copy translation files `.qm` and style sheet `.qss` from `include/qt/` in the same directory, next to `bin/` directory. Scripts that build binaries and packages already do this, but this is to be remembered.

2.3 How it works?

`SaisieQT` is a unique binary, based on a core structure deriving from `QMainWindow` (for now) and from `GLWidgetSet`: `SaisieQtWindow`. To follow and conform to the *universal* command `mm3d` an alias command is defined in `src/uti_phgrm/CPP_SaisieQT.cpp` so

```
mm3d SaisieMasqQT IMG_5059.JPG
```

actually runs:

```
SaisieQT SaisieMasqQT IMG_5059.JPG
```

`SaisieQT` then dispatches to each function main in `SaisieQT/main/saisieQT_main.cpp` depending on second argument (`SaisieMasqQT` etc.).

All applications share the same style sheet, loaded in `saisieQT_main.cpp` and stored in `include/qt/style.qss`.

Each application has its own settings (stored depending on the OS). Most of the settings can be edited through a `QDialog`: `cSettingsDialog` defined in `include_QT/Settings.h`.

Switching between each application in code is managed with private member `_appMode` of `SaisieQtWindow` class. Corresponding enum is defined in `include_QT/Settings.h`.

Some of the Saisie Qt tools make use of Elise library, and use the same core as `SaisiePts` to compute 3D points from image measures, epipolar lines, etc. To mimic the way `SaisiePts` works, a class `cVirtualInterface` has been created in `include/SaisiePts/SaisiePts.h`. This class owns all methods that are shared both by `SaisiePts` and `SaisieQT`. Two classes derive from this mostly virtual class: `cX11_Interface` in `SaisiePts`, and `cQT_Interface` in `SaisieQT`. `cQT_Interface` needs `cAppli_SaisiePts` and a `SaisieQtWindow` to be instantiated.

2.4 SaisieMasqQT

`SaisieMasqQT` has 2 modes: a 2D mask selection mode, like X11 `SaisieMasq`, and a 3D mask selection mode, useful for `C3DC` command. `SaisieMasqQT` uses the same command line arguments as `SaisieMasq` which are read in `saisieMasq_ElInitArgMain`, function common to both. Theses arguments are provided to `SaisieQtWindow` afterward.

All the data in `SaisieMasqQT` are rendered in an OpenGL context. These data are stored in `cGLData` class. We use an ortho projection to render them (see `MatrixManager::mglOrtho`). The main container, after `SaisieQtWindow`, is `GLWidget` (derived from `QGLWidget`). Projection matrix and projection functions (from image to window, and back) can be found in `MatrixManager` class.

2D mode

In 2D mode, `SaisieMasqQT` loads one image, and displays it in the center of the viewport.

An image, at first glance, is stored as a `cMaskedImageGL` (`3DObject.h`) which contains both image data and mask information.

To deal with some very big images, we show a rescaled image in full size, and draw only visible tiles, at full scale when zooming in image. In this case, while loading image, a scale factor is computed, and a rescaled image is stored, next to the original image in `cMaskedImageGL`, and a vector of full scale image tiles in `cGLData::_glMaskedTiles`.

An image is drawn as a `GL_QUAD` (see `cImageGL::drawQuad`). When a mask has been measured, it is blended over the image (see `cMaskedImageGL::draw()`).

Drawing vector data (polygons, points, text) is done in `GLWidget::overlay()`.

Editing a mask is done in `cGLData::editImageMask` (`cGLData.cpp`) by drawing a polygon (class `cPolygon`).

3D mode

3D mode allows loading a ply file (only point clouds, for now). 6 ply formats are currently managed (xyz, xyzrgb, xyz nx ny nz, xyzrgba, xyz nx ny nz rgb, xyz nx ny nz rgba) (see `GlCloud::loadPly`).

There is two interaction modes: selection (one can draw a polygon, as in 2D mode), and move (rotate or translate camera). In `GLWidget` switching between the two modes is done with `getInteractionMode()` and `m_interactionMode`. In the *gui*, `F9` shortcut switches between the 2 modes.

Editing a mask is done in `cGLData::editCloudMask` (`cGLData.cpp`). Editing a mask consists of two different operations: for each 3d point, decide if it is inside or outside the mask, and also store the mask selection information (to be able to recover the mask from other `mm3d` commands, such as `C3DC`). A mask consists of the intersection of several 3D cones, each 3D cone being defined by a 3D polygon (the cone section) and a direction. 3D polygon is built from the 2D polygon drawn in viewport and its camera and matrix information. Cone direction is known from camera orientation. So for each couple (polygonal selection-openGL camera), a virtual 3D cone has to be stored (see *"Conventions for 3D selection tool"* paragraph in `DocMicMac.pdf`). These information are stored in a vector of `selectInfos`, in `HistoryManager`. This allows to undo/redo actions, and also to edit actions through a `QAbstractTableModel` (QTableView `tableView_Objects`).

2.5 SaisieAppuisInitQT and SaisieAppuisPredicQT

In `SaisieAppuisInitQT` and `SaisieAppuisPredicQT`, we use the same `cPolygon` object to draw a set of points, but we don't draw lines between points. This is done with boolean _bShowLines in `cPolygon`, which can be checked with method `cPolygon::isLinear()`.

2.6 SaisieBascQT

Mode=0 in `src/uti_phgrm/CPP_SaisieQT.cpp`

At this point, `SaisieBascQT` has exactly the same behavior as `SaisieBasc`: lines are drawn as two points, while it may be useful to display the complete line.

2.7 SaisieCylQT

Mode=1 in `src/uti_phgrm/CPP_SaisieQT.cpp`

2.8 SaisieBoxQT

`SaisieBoxQT` is a very simple use of `SaisieQtWindow`: it shows an image, allow to draw a `cRectangle`, which is a `cPolygon` with 4 points defined in `cObject.h`. At this point, `SaisieBoxQT` is only meant to communicate with a visual interface (such as `vMalt`) and we only send a *signal* with `void newRectanglePosition(QVector <QPointF> points)` in `GLWidget::mouseMoveEvent`. This *signal* is connected to `onRectanglePositionChanged` in `visual_MainWindow::onSaisieButtonPressed` in `visual_mainWindow.cpp`. `SaisieBoxQT` is instantiated in `visual_mainWindow` constructor (`visual_mainWindow.cpp`).