Developer Guide

Contents

1. About the structure

The structure of GraphMic is divided on two components

1.1. QtQuick and QML

The QtQuick framework includes the QML declarative scripting language, QML Objects are useful for the interaction with the user, the GraphMic user interface is conceded on QML, in *GraphMIC* it makes possible to:

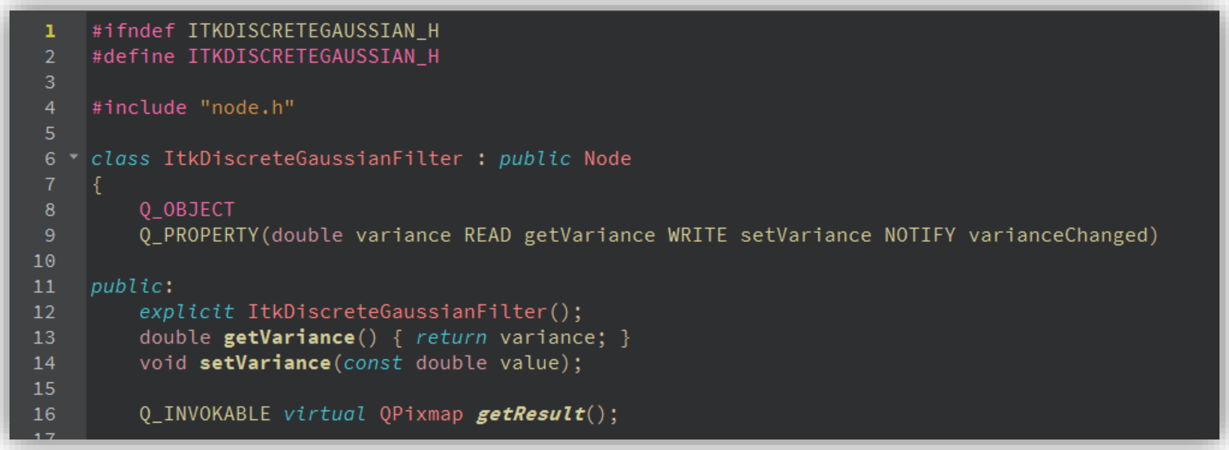
- Represent filters and images using *nodes* the user can drag on the Editor pane (using customs objects).
- Create a data flow between the created *nodes* connecting them using *edges*.
- Interact to mouse and key events like user inputs.
- Allow the user to change the filter values on a *filter node* and apply the filter with these new values by clicking on it.

1.2. C++ Model

Each filter node represents a C++ Class defined as ***<ClassName>Filter***, each filter class is registered as QML Object on the main function so it can be instantiated as QML Object. For more information go here.

A typical filter class declaration will be showed on the *Figure 1: Typical filter declaration, here the ItkDiscreteGaussianFilter.*

- The Q_OBJECT macro tells the compiler this class implements its owns signals and slots, so the meta object compiler ***moc*** need to run at first.
- The Q_PROPERTY macro will be used to tell the compiler the variables to be read through a given get method, write through a given set method and the changes will be notified using the given method next the ***NOTIFY*** macro
- The Q_INVOKABLE macro allows functions to be called from QML objects.



```cpp
#ifndef ITKDISCRETEGAUSSIAN_H
#define ITKDISCRETEGAUSSIAN_H

#include "node.h"

class ItkDiscreteGaussianFilter : public Node
{
    Q_OBJECT
    Q_PROPERTY(double variance READ getVariance WRITE setVariance NOTIFY varianceChanged)

public:
    explicit ItkDiscreteGaussianFilter();
    double getVariance() { return variance; }
    void setVariance(const double value);

    Q_INVOKABLE virtual QPixmap getResult();
```

Figure 1: Typical filter declaration, here the ItkDiscreteGaussianFilter class

2. Creating a new filter node

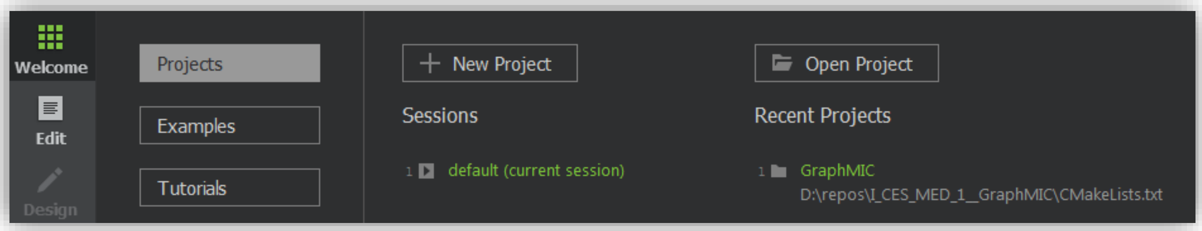
To create a new filter object, it is needed to create a new filter class, register this class as a QML object and add it into the window toolbar to make it visible for the user. As example we want to add the ***ITKMedianImageFilter*** as follows:

2.1. Choose the filter

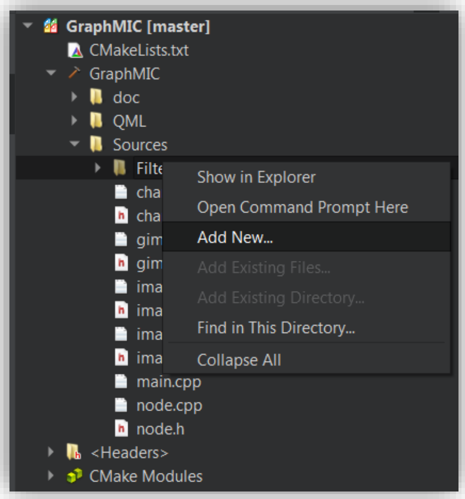
- Go to the documentation page of the filter you want to implement, for OpenCv or ITK (Another good website for ITK examples is https://itk.org/Wiki/Main_Page or https://itk.org/ItkSoftwareGuide.pdf)
- Read the documentation of the filter and check the input parameter you will need.
- For our example filer the documentation can be visited at: https://itk.org/Doxygen/html/classitk_1_1MedianImageFilter.html

2.2. Adding a new C++ filter class

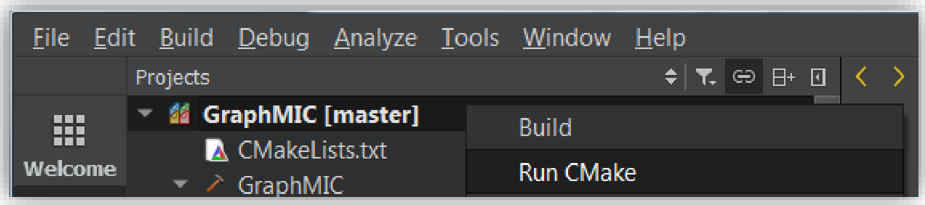
Open the project, if you open the project for first time, so follow the step 7 from the installation guide. Open the QtCreator and select by the Welcome Tab, the GraphMIC project.



- Go to the Source folder
- Go to the Filter folder
- Right click on the filter folder and click on ***Add New…***



- Select ***C++ Class,*** click on ***choose.***
- Name the class like the filter you want to implement. (for example, ***ItkMedianFilter***), look at the existing classes and the names on ***camel case***
- Click on finish.
- If you get a warning from Qt, ignore it
- Run CMake right clicking on the project folder and selecting “Run CMake”.



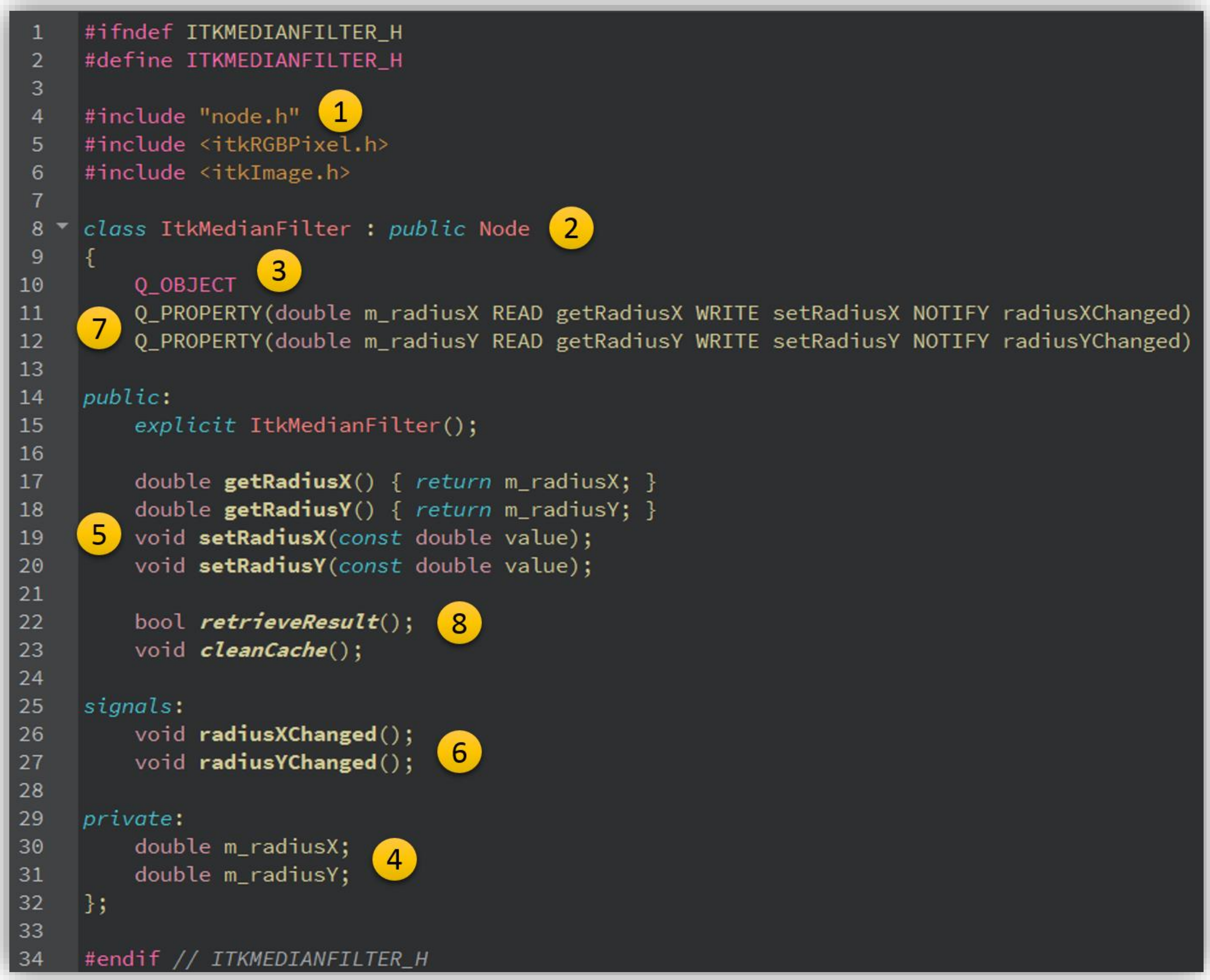
- Now a both header und source files from your class are available on the project pane.

2.3. Implementing the filter class

Go to the header file of your new class

The filter classes are very similar only the filter parameters might change, so you can define your class using the definition of an existing filter class

For our example the median filter needs two parameters: ***m_radiusX*** and ***m_radiusY***, so we create these two variables on our class ***ItkMedianFilter***, add its get, set methods and ***Q_PROPERTIES.***



```cpp
#ifndef ITKMEDIANFILTER_H
#define ITKMEDIANFILTER_H

#include "node.h"
#include <itkRGBPixel.h>
#include <itkImage.h>

class ItkMedianFilter : public Node
{
    Q_OBJECT
    Q_PROPERTY(double m_radiusX READ getRadiusX WRITE setRadiusX NOTIFY radiusXChanged)
    Q_PROPERTY(double m_radiusY READ getRadiusY WRITE setRadiusY NOTIFY radiusYChanged)

public:
    explicit ItkMedianFilter();

    double getRadiusX() { return m_radiusX; }
    double getRadiusY() { return m_radiusY; }
    void setRadiusX(const double value);
    void setRadiusY(const double value);

    bool retrieveResult();
    void cleanCache();

signals:
    void radiusXChanged();
    void radiusYChanged();

private:
    double m_radiusX;
    double m_radiusY;
};

#endif // ITKMEDIANFILTER_H
```

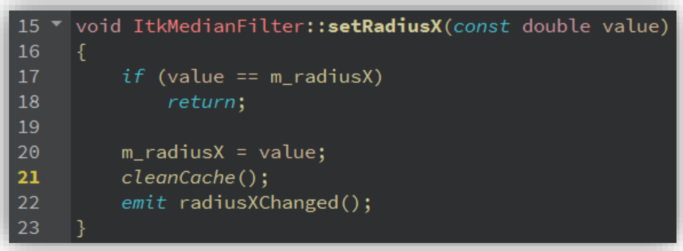
Figure 2: ItkMedianFilter Class, implementation example.

Consider the *Figure 2: ItkMedianFilter Class, implementation example.* by implementing your filter class, pay attention to the following steps:

1. The class should include the ***node.h*** header.
2. The class should inherit from the **Node** class as ***public***.
3. The class should include the ***Q_OBJECT*** macro.
4. In case the filter you want to implement needs some input values, so you need to create this input values as ***member variable***.
5. Implement for each parameter variable the get and set methods.
6. Add a Notify ***Signal*** for each ***member variable*** you will use as parameter.
7. Add the ***Q_PROPERTIES*** for each ***member variable*** on the class, using its get, set methods and signals. This will be needed to exchange data between the model and the view on the QML site.
8. Implement the ***retrieveResult()*** function.

2.4. Defining the filter class in the source file

- Go to the source file of your class.
- Define your constructor giving a default value for each input parameter (Look at the documentation).
- Define the get and sets methods, make sure you call the function ***cleanCache() on the set methods.***



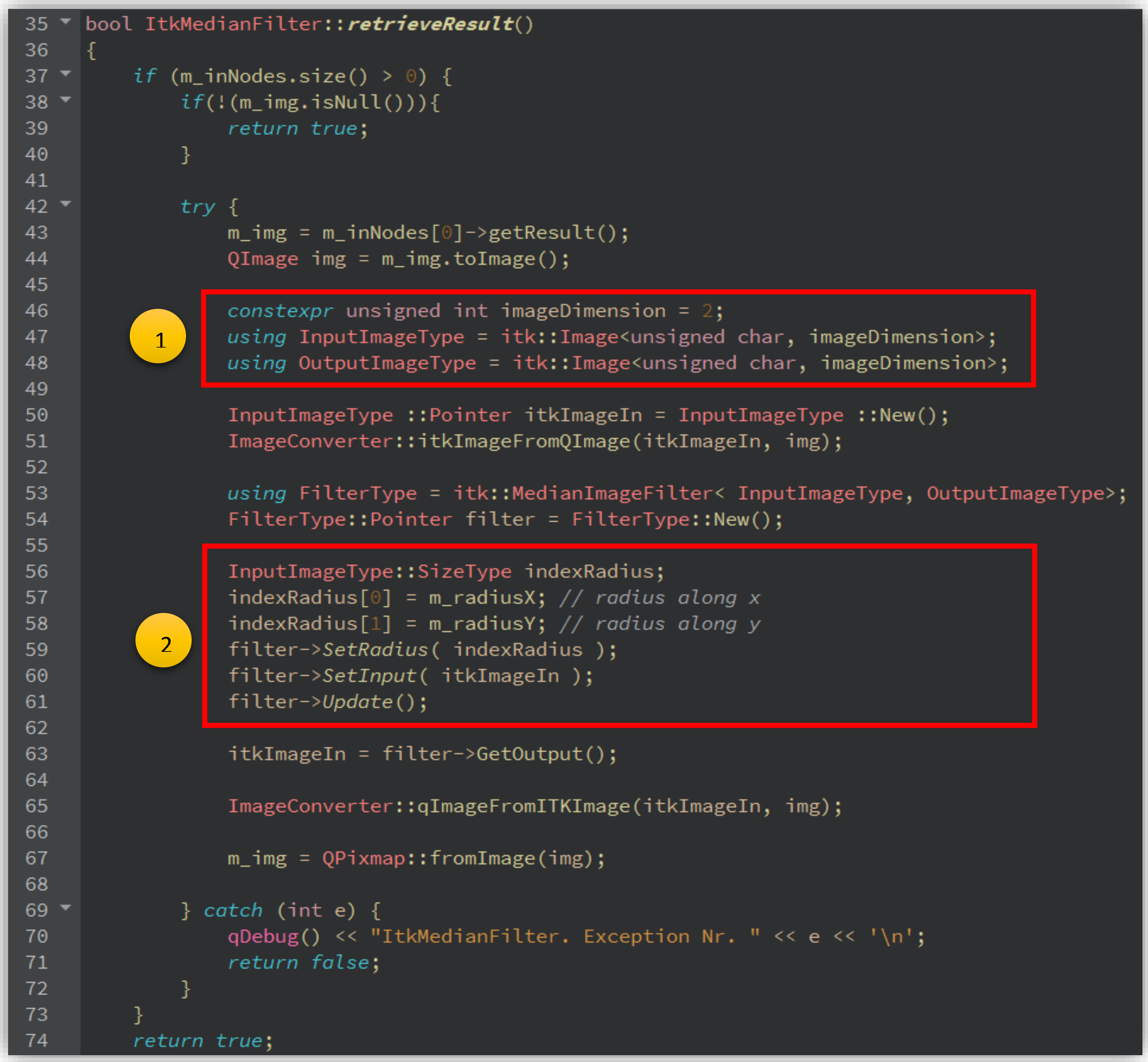
```
void ItkMedianFilter::setRadiusX(const double value)
{
    if (value == m_radiusX)
        return;

    m_radiusX = value;
    cleanCache();
    emit radiusXChanged();
}
```

- Override the virtual function ***retrieveResult()*** *as follows:*

Consider the *Figure 3: itkmedianfilter.cpp, definition of retrieveResult()*, the new class needs also to overwrite the virtual function ***node::retrieveResult().***
The filter will be implemented on the Try block. The definition of the ***ItkMedianFilter::retrieveResult()*** function follows these 6 steps:

1. The first ***If*** clause will check if there is an input, so this input is an ***QPixmap***, we convert it on a ***QImage*** format to access to its pixels values. (line 37)
2. The second clause will check if the ***m_img*** variable is set (this variable holds the input image), in this case the current value will be loaded. (line 38)
3. On the Try block, we get the image input, **prepare the ITK filter** with the proper parameters: ***InputImageType***, ***OutputImageType*** and ***ImageDimension***. (line 43-50)
4. We use the ***ImageConverter::itkImageFromQImage(TargetITKImage, FromQImage)*** function to convert the input ***QImage*** to a ***ITK Image***. (line 51)
5. We create the ***ITK::MedianImageFilter*** pointer using the given ***InputImageType*** and ***OutputImageType*** parameters from step 3 (line 53-54), and set the values given from the user. (line 56-61)
6. The ***ITK Image*** output from filter will be converted to an ***QImage*** to continue the data flow. (line 63)
7. In order to continue the flow between the nodes, we need to convert out filter output to an ***QImage***, so we call the ***ImageConverter::qImageFromITKImage(TargetQImage, FromITKImage)*** function. (line 65)
8. In case an exeption occurred, then it will be catched on the catch block. (line 69-72)
9. If the Filter was correct implemented, so the function ***retrieveResult()*** returns the flag true.(line 74)



```cpp
bool ItkMedianFilter::retrieveResult()
{
    if (m_inNodes.size() > 0) {
        if(!(m_img.isNull())){
            return true;
        }

        try {
            m_img = m_inNodes[0]->getResult();
            QImage img = m_img.toImage();

            constexpr unsigned int imageDimension = 2;
            using InputImageType = itk::Image<unsigned char, imageDimension>;
            using OutputImageType = itk::Image<unsigned char, imageDimension>;

            InputImageType ::Pointer itkImageIn = InputImageType ::New();
            ImageConverter::itkImageFromQImage(itkImageIn, img);

            using FilterType = itk::MedianImageFilter< InputImageType, OutputImageType>;
            FilterType::Pointer filter = FilterType::New();

            InputImageType::SizeType indexRadius;
            indexRadius[0] = m_radiusX; // radius along x
            indexRadius[1] = m_radiusY; // radius along y
            filter->SetRadius( indexRadius );
            filter->SetInput( itkImageIn );
            filter->Update();

            itkImageIn = filter->GetOutput();

            ImageConverter::qImageFromITKImage(itkImageIn, img);

            m_img = QPixmap::fromImage(img);

        } catch (int e) {
            qDebug() << "ItkMedianFilter. Exception Nr. " << e << '\n';
            return false;
        }
    }
    return true;
```

Figure 3: itkmedianfilter.cpp, definition of retrieveResult()

Take care of the valid parameter types and values for your filter, the *Figure 3: itkmedianfilter.cpp, definition of retrieveResult()* shows the position where this parameters might change for each ITK filter. The number 1 represent the ***input pixel types*** and image dimension, the number 2 represents the ITK ***filter parameters***.

2.5. Register the class as QtQuick QML Object

Once we have already prepared the class following successfully the step 2, so we should register this class **on the *main.cpp* file** *as QML Type*, it will make possible to use this class on QML files declaring it as normal QML Object.

So, we will need to add the following line to the code (take care of the used name convetion):

```
qmlRegisterType<ItkMedianFilter>("Model.ItkMedian",1,0,"ItkMedian");
```

The syntax for this function should be as follows:

```
qmlRegisterType<ClassName>("Model.FilterName",X,Y,"FilterName");
```

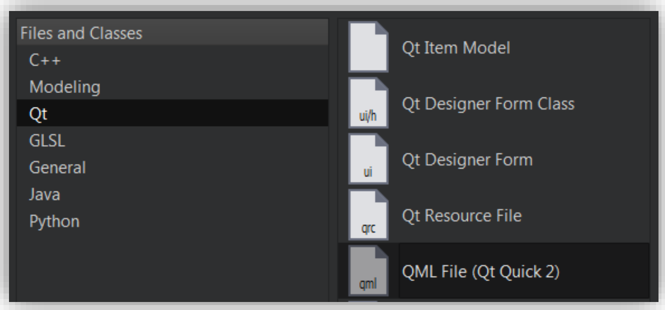
Whit X, Y as the package version. We will always use the package version X= 1 and Y = 0. Make sure you include your class header file, so we must add the line:

```
#include "itkmedianfilter.h"
```

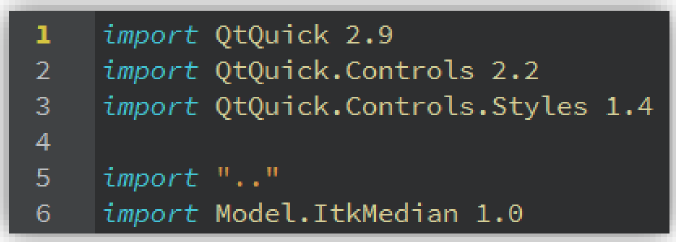
2.6. Create your QML file

In this step you should create a QML file on the ***QML project folder*** to be able to use our filter class in our user interface.

- right click on the QML/file folder, select ***Add new***
- Select Qt, and QML File as the following image



We turn into the QML coding, so we ***create*** a new file to hold the created class on the QML folder, we name this file ***ITKMedian.qml,*** like the name of the filter you want to create.



```
import QtQuick 2.9
import QtQuick.Controls 2.2
import QtQuick.Controls.Styles 1.4

import ".."
import Model.ItkMedian 1.0
```

Figure 4: Importing the needed components

We add the showed import packages like on *Figure 4: Importing the needed components* to the top of the QML file.

For your class you need to change the ***line 6,*** and import the ***filter name you registered*** before on the step ***2.5***, the numbers following the import package are taken as the package version.

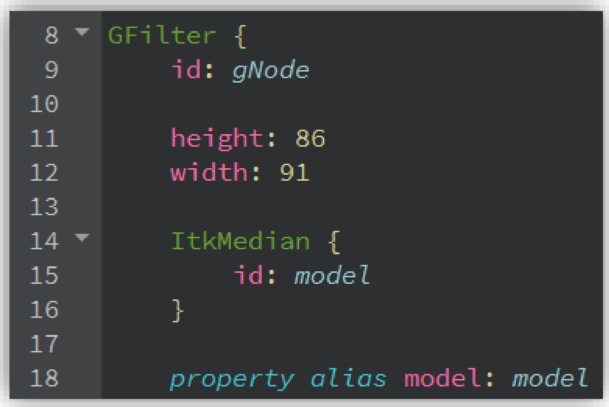
If the Filter needs 2 inputs, you must instantiate an QML Object from the class ***GFilter2Ports.qml*** as parent, so just write ***GFilter2Ports*** on line 8.

If the Filter needs 3 inputs you must instantiate the parent ***GFilter3Ports.qml***, so write ***GFilter3Ports***.

If your filter just need one input, so use the ***GFilter*** object as you see on the *Figure 5: Filter object on QML..*

For the median filter we just need one input, this will contain all the needed implementations for a basic node (dragging, connections through edges...), so we use the ***GFilter*** object.

Inside this ***GFilter*** Object, you need to ***create an instance of your filter class (lines 14-16) with the id: model*** and a ***property alias to the model*** (line 18) like on the *Figure 5: Filter object on QML.*



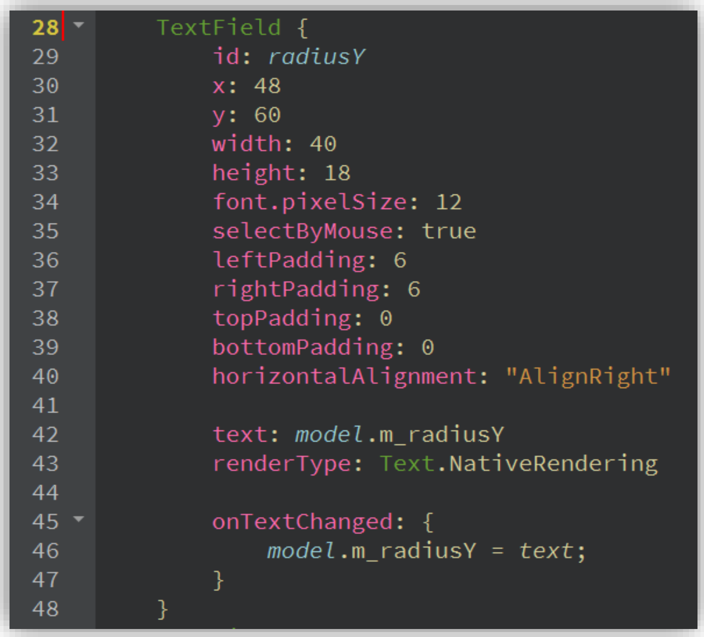
```
 8  GFilter {
 9      id: gNode
10
11      height: 86
12      width: 91
13
14      ItkMedian {
15          id: model
16      }
17
18      property alias model: model
```

Figure 5: Filter object on QML.

Create a ***Label*** to hold the filter name, for us **"ITKMedian"**.

Create a ***Label*** and a ***TextField*** for each input *parameter* your filter needs, the ***Label*** just holds the *parameter* name and the ***TextField*** will contain the *parameter* value, make sure you give each ***TextField*** a valid ***id***.

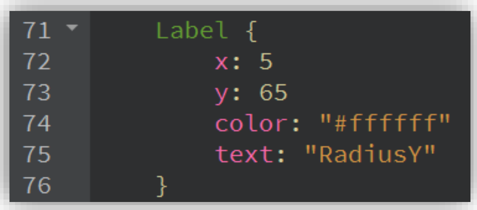
For us, the ***ItkMedian*** needs two inputs: ***RadiusX*** and ***RadiusY,*** so we created the corresponding ***labels*** and ***textfields***.



```
28      TextField {
29          id: radiusY
30          x: 48
31          y: 60
32          width: 40
33          height: 18
34          font.pixelSize: 12
35          selectByMouse: true
36          leftPadding: 6
37          rightPadding: 6
38          topPadding: 0
39          bottomPadding: 0
40          horizontalAlignment: "AlignRight"
41
42          text: model.m_radiusY
43          renderType: Text.NativeRendering
44
45          onTextChanged: {
46              model.m_radiusY = text;
47          }
48      }
```

Figure 6: TextField for an input parameter

Consider the *Figure 6: TextField for an input parameter,* for each input *parameter*, you should create a ***TextField*** like this for each input parameter, it is recommended to use the *parameter name* as ***id (line 29)***, its ***text*** property should hold the *parameter* value from your filter class ***model (line 42),*** the ***onTextChanged*** event should be implemented changing the ***model*** *parameter* every time the ***TextField text*** changes (***Lines 45-47***).



```
Label {
    x: 5
    y: 65
    color: "#ffffff"
    text: "RadiusY"
}
```

Figure 7: Parameter name "RadiusY", represented with a Label.

The parameter name for the ***TextField radius***, will be showed using a Label, see *Figure 7: Parameter name "RadiusY", represented with a Label.*

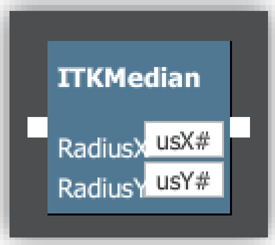
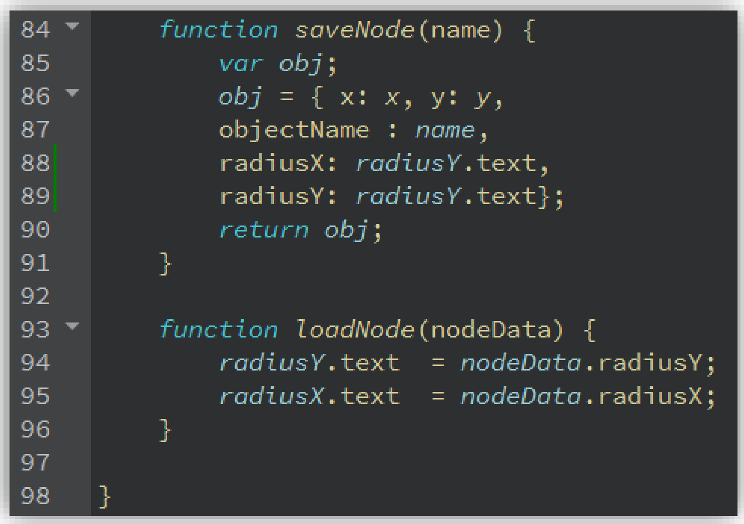


Figure 8: Designer view of the ITKMedianFilter.qml

If you click on the left pane, on the Design tab you can see the result of your code, like on the *Figure 8: Designer view of the ITKMedianFilter.qml.*

The last step consist on implementing two functions ***saveNode()*** and ***loadNode()*** its implementation are same for each filter, but the ***ids*** might vary. If the filter you want to implement **have no input parameters**, you don't need to implement these functions.



```
84      function saveNode(name) {
85          var obj;
86          obj = { x: x, y: y,
87          objectName : name,
88          radiusX: radiusY.text,
89          radiusY: radiusY.text};
90          return obj;
91      }
92
93      function loadNode(nodeData) {
94          radiusY.text  = nodeData.radiusY;
95          radiusX.text  = nodeData.radiusX;
96      }
97
98  }
```

Figure 9: ITKMedianFilter.qml, functions saveNode() and loadNode()

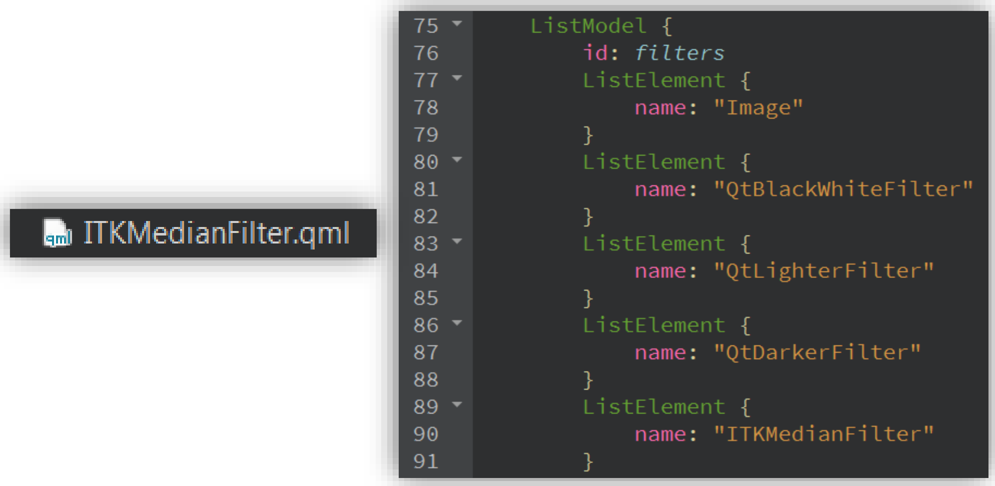
Consider the *Figure 9: ITKMedianFilter.qml, functions saveNode() and loadNode(),* the lines 84-87 und 90 are same for each filter node, the next lines 88-89 will vary depending on the input *parameters* and the ***TextField*** id's.

On the function ***loadNode(),*** we will read the values from the saved ***nodeData*** JSON object and load it into the ***TextFields*** representing the saved configuration values.

2.7. Update the toolbar buttons

On the ***GToolBar.qml*** go to the ***ListModel*** and **add** your filter as a new ***ListElement*** object, see ***Figure 10: ListModel and ListElements***.

The name of the new ***ListElement*** should be exactly the ***name of the QML file*** you created without the ***.qml*** ending.



ITKMedianFilter.qml

```
    ListModel {
        id: filters
        ListElement {
            name: "Image"
        }
        ListElement {
            name: "QtBlackWhiteFilter"
        }
        ListElement {
            name: "QtLighterFilter"
        }
        ListElement {
            name: "QtDarkerFilter"
        }
        ListElement {
            name: "ITKMedianFilter"
        }
```

Figure 10: ListModel and ListElements