

C++ - BsCV:

PROJECT REPORT - 2D CREATION

By: Kevin Descharrieres

Professor: Yohan FOUGEROLLE

May 22, 2017

Contents

I. Goal of the project and definitions	3
A. Goal	3
B. Some definitions	3
II. Graphical User Interface	4
A. Usage	4
B. First GUI	5
C. Second GUI	6
III. Code and demonstration	7
A. Libraries	7
B. Load and save function	7
C. Shape window and signal	11
D. To go further	15
IV. Issues	18
A. Coordinates	18
B. Widget	18
V. Conclusion	19
VI. sources	19

I. GOAL OF THE PROJECT AND DEFINITIONS

A. Goal

The goal of this project is to create an interface of 2D creation easy to use. For that, I will define how to draw simple items like a point, a line, a circle, an arc... On this interface, it will be possible to create a project from A to Z without any example. But, it will be possible too to create something by an existing image. For example, if we want to create a table with a nice shape, it will be possible to load the image of this last one to follow the curves. Then, for mistakes, there will be a building tree in which one we will be able to click with the mouse to select the wrong item (highlighted) and delete it. To draw an item, the main thing is to use coordinates boxes, choose them and press the OK button. But, at the end, I would like to incorporate the mouse click to change directly the place of the building item and to put its coordinates in the boxes. After that, in the tree, at every items, there will be the size of each items. The last point of the project is the possibility to register the project in a files on the computer.

B. Some definitions

A class constructor is a special member function of a class that is executed whenever we create new objects of that class. It will have exact same name as the class and it does not have any return type at all.

```
explicit MainWindow(QWidget *parent = 0);
```

A destructor is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

```
~MainWindow();
```

The include directive include the contents of a standard header or the contents of a file.

```
#include "mainwindow.h"  
#include <QString>
```

II. GRAPHICAL USER INTERFACE

A. Usage

With the Qt software, there is a functionality to create directly GUIs without any line of code or almost.

This means to create a GUI will be used all the long of this project.

The great part of this work is the GraphicsView. Indeed, all the schemes or pictures will appear in this.

The "Go to the slot" way to create and give action to a button is very useful to do it without any mistake. When we are creating a button in the GUI interface, we just have to do a right click on the button and click on "Go to the slot" which sends us on the .cpp corresponding file. So we obtain something like the two following parts of code, the first one in the .cpp file and the second one in the .h file.

First one :

```
void MainWindow::on_pushButton() {  
}
```

Second one :

```
private slots:  
void on_pushButton();
```

With this method, the link between all files and the button is completely done, it lacks just to ask to the button to engage an action by adding it in the first of the two previous codes.

The "lineEdit" function is very useful to the user to send some informations to the software to run it as he wants. For example in our case, the user will enter some coordinates in some lineEdit to create an item at the right place on the screen.

Another good object which will be very useful for us is the comboBox. When I started, I did not know if it was better to use it or a simple dialogBox to create a type of repertoires of each created items. So my final choice is the comboBox because the user can see in one click all his actions since

the beginning of the session. For the rest of the report, the comboBox will be call "tree" because it is the name of this function in a conception software.

B. First GUI

The first GUI conception was for me a good beginning for my project. First of all, I learned with it to use some basic operations like the QPainter function for example which was a base of this project. The problem of this first GUI was the fact that everything was on the same window.(see it just under).

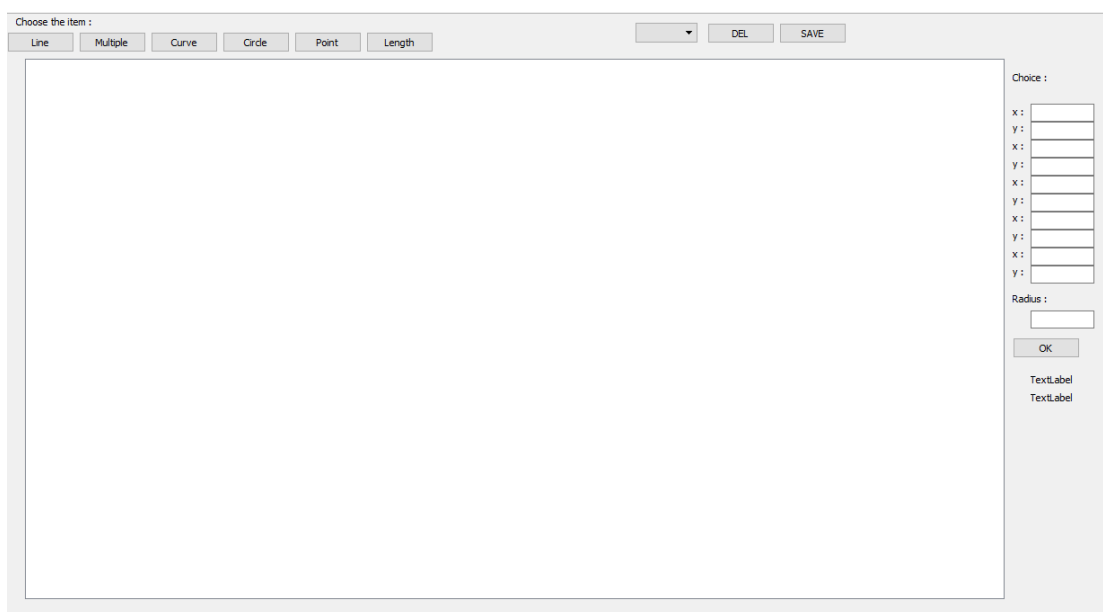


FIG. 1: First Interface

We can see on the previous picture what we said before. This representation shows us the global functionality of the interface, remember that the goal is to obtain something easy to use. On the first left hand corner, there is few buttons which are present to choose an item to draw by the user. Then, there is some other buttons which permit to load, to save and to give the dimensions of an item. On the right of the interface, we can see some lines in which one it is possible to write something. These lines are there to receive the coordinates of the next created item. But, as we will see in a future part, almost all the code was in a same file it was not at all optimized as I wanted at the beginning. On the first week of April, I decided to create a new project to obtain something

better than the first one.

C. Second GUI

For the second GUI, I took as example the first one with the idea to split the last one into three parts :

- The main window with the part of the draw and the part with coordinates.
- The second one with the buttons to activate.
- The third one is just an HELP window

The idea was to clear and clean the main window and permit to do the draw place bigger than in the first GUI. The functionality stays almost the same but with few shades :

- I deleted the "Dimensions" button which was replace by a .txt file
- I deleted the "OK" button to validate a choice, it will be automatic
- I added an "HELP" button to give some informations to the user if he needs.

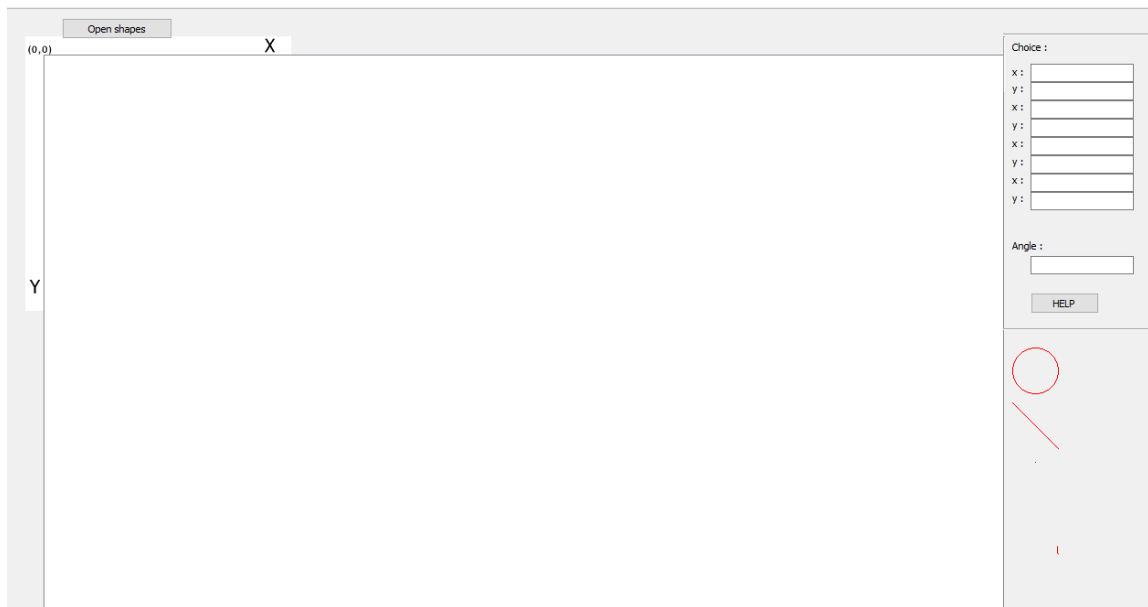


FIG. 2: Main Interface

III. CODE AND DEMONSTRATION

A. Libraries

In C++, at the beginning of each source code, there are the libraries that are used. In my case, these are the following ones :

```
#include <QString> //permits to use cipher as string
#include <QPainter> //permits to paint elements
#include <QGraphicsView> //permits to use the graphicsView
#include <QFileDialog> //permits to open dialog box
#include <QDir> //permits to find the direction of a file
#include <QImageWriter> //permits to save an image
#include <QMouseEvent> //check the action of the mouse
#include <QPainterPath> //class which contains some shapes
#include <QTextStream> //convenient interface to writing text
```

B. Load and save function

At the beginning of this project, I had not a good feeling with the practice of Qt and the first error was to put every line of code in the same file. The result was not so bad for a first experience in GUI and it worked globally properly as I wanted but I was blocked quickly for some functions. For example, the fact that I wanted to do a "software" with more than one window for the functioning obliged me to reconsider all the structure of the code to use some manipulations learned before during lectures. We will see later which type of structures.

```

//Load function

QString fileName = QFileDialog::getOpenFileName(this, tr("Open_
File"),

QDir::currentPath()

,

tr("Images_(*.jpeg_
*.jpg)"));

QImage image(fileName);
QPixmap pixmap(fileName);
scene->addPixmap(pixmap);
ui->graphicsView->setScene(scene);
ui->graphicsView->show();

```

```

//Save function

QGraphicsView* view = new QGraphicsView(scene);
QString fileName = QFileDialog::getSaveFileName(this,
tr("Save_Image"), "
",
tr("Image(*.jpeg_*.
jpg);
All_Files(*)"));

QPixmap pixMap = QPixmap::grabWidget(view);

//http://stackoverflow.com/questions/7451183/how-to-create-
image-file-from-
//qgraphicsscene-qgraphicsview
pixMap.save(fileName);

```

These two previous functions are opening a dialogBox to find a path as followed.

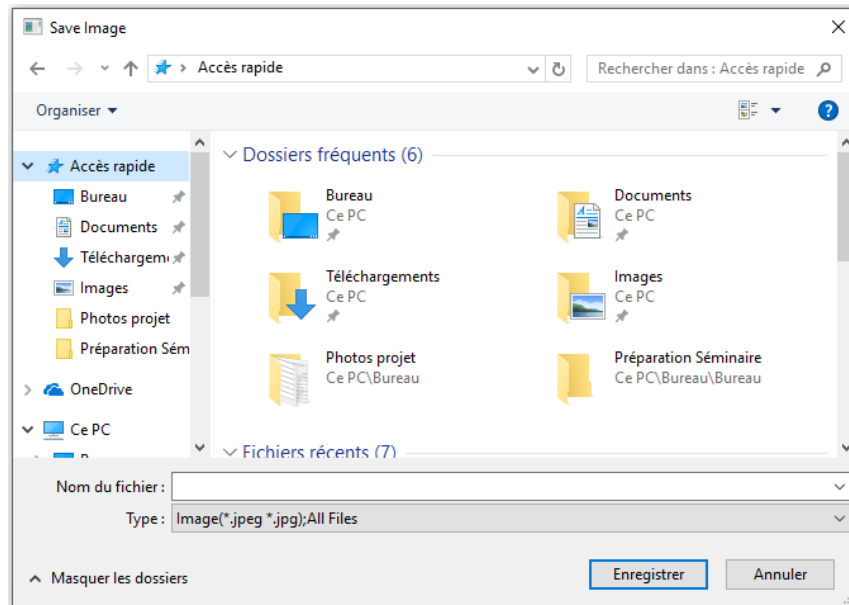


FIG. 3: Dialog Box

Now, an important part of this project is "How to draw a shape"? The following code gives us an example with the drawing of a line.

```
#include <QPainter>

ui->graphicsView->setScene(scene);

//transform the lineEdit texts into integers
xValue1 = ui->lineEdit->text().toInt();
yValue1 = ui->lineEdit_2->text().toInt();
xValue2 = ui->lineEdit_3->text().toInt();
yValue2 = ui->lineEdit_4->text().toInt();

//creation of the pen (color and width)
QPen blackPen(Qt::black);
blackPen.setWidth(4);

//creation of the corresponding line with coordinates and the
    using pen
lines = scene->addLine(xValue1,yValue1,xValue2,yValue2,blackPen
    );
```

I started to include the library "QPainter" which permitted me to use the function "addLine".

This function needs some coordinates and the definition of a pen to work. Lets start with the first need. The coordinates will be define in a "lineEdit" which are used as followed:

```
//transform the lineEdit texts into integers  
xValue1 = ui->lineEdit->text().toInt();
```

The previous code is defining the first coordinate of a point, the x one. The "xValue1" will take the value which is place in the first lineEdit which is itself on the UI. The only difficulty here is to think to transform the text in the lineEdit into an integer.

So now the second need is the "pen" which will be used to draw the line because without it, the function will send us an error during the compilation time.

```
//creation of the pen (color and width)  
QPen blackPen(Qt::black);  
blackPen.setWidth(4);
```

Here, I just defined a pen name "blackPen" which will be black with a width of 4 pixels.

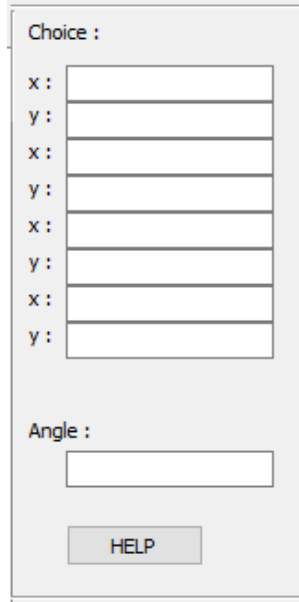


FIG. 4: Choice

To improve the usage of the software, I decided to add a function using the position of a mouse

click on the window. The idea was to put in the lineEdits the x and y coordinates of the position of the mouse at the clicking point. The idea was to permit to the user to click somewhere in the window (1 time for a point or 2 times for a line) to choose coordinates without any calculus:

```
void MainWindow::mousePressEvent(QMouseEvent *event)
{
    //if the left clic is pressed
    if(event->button()==Qt::LeftButton)
    {
        //put coordinates in lineEdit and lineEdit_2
        ui->lineEdit ->setText(QString::number(event->x()));
        ui ->lineEdit_2 ->setText(QString::number(event->y()));
    }
    //if the right clic is pressed
    else if(event->button()==Qt::RightButton)
    {
        //put coordinates in lineEdit_3 and lineEdit_4
        ui->lineEdit_3 ->setText(QString::number(event->x()));
        ui ->lineEdit_4 ->setText(QString::number(event->y()));
    }
}
```

As explained in the previous part of source code, the two if events will just determine if the user pressed the left button or the right one and send coordinates in the good lineEdits.

C. Shape window and signal

As I said previously in the report, a main problem was the fact that every lines of codes was in the same file, so I decided to use SIGNAL and SLOT to remedy to this problem. A signal will be transmitted when a specific action will be performed by the user. Once this signal is sent, it will be received by a predefined slot. Their construction in Qt is therefore very precise:

shapes.h

```

#ifndef SHAPES_H
#define SHAPES_H

class shapes : public QDialog
{
signals:
    //signal to give the decision
    void choice(int decision);
};

#endif // SHAPES_H

```

mainwindow.h

```

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

class MainWindow : public QMainWindow
{
private slots:
    //slot to receive signal from shapes
    void changeDecision(int decision);
};

#endif // MAINWINDOW_H

```

The two previous codes are the header of the two linking files. The first one shows us the definition of the signal which will be emit and the second one is for the definition of the slot which will receive the signal.

shapes.cpp

```

void shapes::on_buttonLine_clicked()
{
    //emition of a signal
    emit choice(1);
}

```

mainwindow.cpp

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    //connection by signal and slot between shapes and
    mainwindow
    connect(shapess, &shapes::choice,
            this, &MainWindow::changeDecision);
}

void MainWindow::changeDecision(int decision)
{
    if (decision == 1)
    {
        //Action to performe
    }
}
```

The .cpp files will permit to the user to emit the signal and applied an action. Here the signal will be emit when a button will be push and the next performed action will be determine thanks to it. The link is performed by:

```
connect(shapess, &shapes::choice,
        this, &MainWindow::changeDecision);
```

The explanation is the following one : basically, we connect here the "choice" from shapes to "changeDecision" from mainwindow. The choice value will be pass as argument.

We will see now how to open other windows. The goal of this manipulation is to make bigger the drawing window and permit to the user to place the shapes window as he wants as we can see

after.

mainwindow.cpp

```
//Opening of the window shapes
void MainWindow::on_pushButton_released()
{
    shapess->show();
}
```

mainwindow.h

```
#include "shapes.h"
```

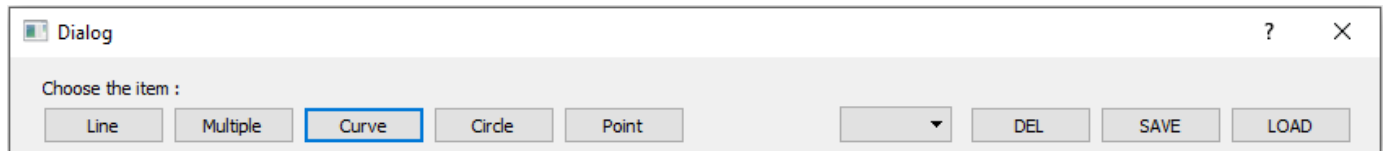


FIG. 5: Shapes Window

To add every items in the tree, their is a simple line of code to remember : shapes.cpp

```
ui->comboBox->addItem("Line_" + QString::number(l));
l++;
```

By this line, we can easily imagine the way of working of this. We will just add an item in the comboBox which is present in the UI. Then, this item is defined by the name of the item (here the "line") and the number which is corresponding in real time to "l" (a counter) in our case. This operation is repeated each time an item is created. See as follow :

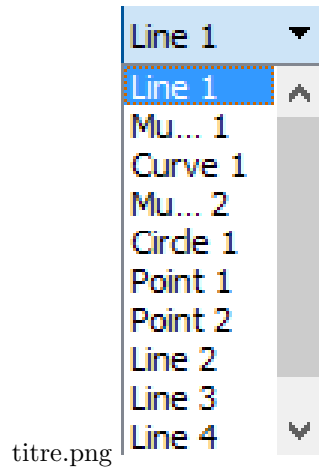


FIG. 6: Tree

D. To go further

To avoid to overload the existing windows or create another one, I decided to create a .txt file to list every actions of the user with the name of the created items plus the coordinates which are used in real time. With this file, the user could easily do the same work or avoid to do the same mistakes just by looking at it.

```
//Writing in a .txt file
    QFile file("name_of_the_file.txt");
    if (!file.open(QIODevice::WriteOnly | QIODevice::Append
    ))
        return;

    QTextStream out(&file);
    out << "Here_what_you_need_to_write" ;
```

With a good syntax and management, this document will become a real gold mine.

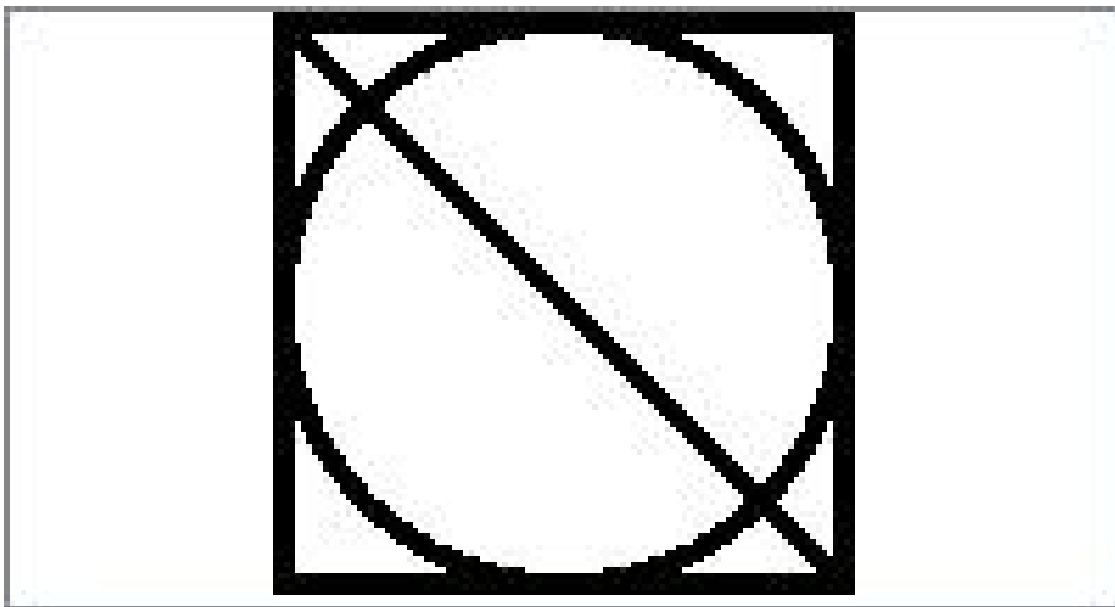


FIG. 7: Shapes example

As we seen previously, a .txt file is created and update at every creation of a shape. The associate text of the previous picture is:

```
line =>  
xValue1 = 0  
yValue1 = 0  
xValue2 = 100  
yValue2 = 100  
  
line =>  
xValue1 = 0  
yValue1 = 0  
xValue2 = 100  
yValue2 = 0  
  
line =>  
xValue1 = 0
```



```
yValue1 = 0
xValue2 = 0
yValue2 = 100

line =>
xValue1 = 100
yValue1 = 100
xValue2 = 0
yValue2 = 100

line =>
xValue1 = 100
yValue1 = 100
xValue2 = 100
yValue2 = 0

Circle =>
xValue1 = 0
yValue1 = 0
xValue2 = 100
yValue2 = 100
```

The "HELP" window was not really expected but when I tried my own project, I saw that it is not always easy to create an item without any explanation. That's why, I took a little of time to create another GUI class named "help". In this GUI, we can find some explanations with schemes on the mean which is used to create circle and arc with this software. Furthermore, We can find in the HELP window the basic usages of the software like how to enter coordinates, how to load an image and so one.

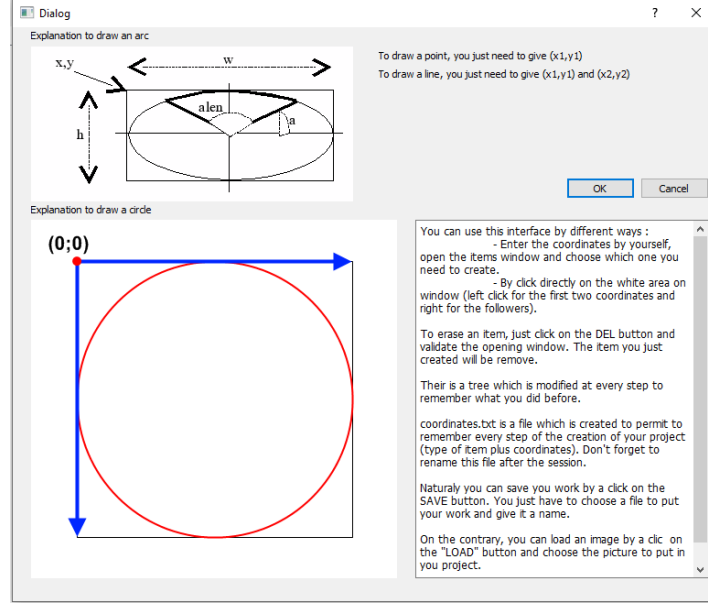


FIG. 8: Help Window

IV. ISSUES

In this section, I will speak briefly about two principals problems that I have just before the dead line.

A. Coordinates

The first problem concerns the coordinates of the mouseClic. In fact, when we are clicking on the graphicsView, the coordinates which appear in the lineEdits are linking we the coordinates of the great window. By that way, their is always a shift in the draw. This shift is proportional to the size of the shape. For example, if we are drawing a line between (0,0) and (100,100) the shift will be lesser than the second point was (150,150).

B. Widget

The second problem concerns the creation of a widget that I wanted to show in real time which shapes is using. To use a widget, we have to create the link directly thanks to the GUI conception.

By a right click, the idea is to overcome the widget as a new class. I wanted to send it by signal the an information and make appears the corresponding shape. So, in my opinion, their is a special manipulation to send a signal to a widget because I used the same method as earlier but the signal does not seem to pass. The widget is working but maybe it lacks a line of code in my project to do it properly.

V. CONCLUSION

During this project, I tried to put at use my knowledge acquired about Qt during the lessons and practicals. Despite the fact that I had to restart the all project at the beginning of April because my files was not at all optimize, this project was for me rewarding in the acquirement of new knowledge. I am now able to create an application in C++ which is using some libraries like QPainter or QMouseEvent...

By this project it is possible to draw some basic D dimensions elements and save them on a computer. To go further, it could be possible to apply it in 3 dimensions and adding some skins to the elements.

VI. SOURCES

- <http://stackoverflow.com/questions/7451183/how-to-create-image-file-from-qgraphicsscene-qgraphicsview>
- <http://www.qtcentre.org/threads/22149-How-to-draw-a-semi-circle-or-arc-of-ellipse>
- <http://doc.qt.io/>
- <https://professor.yohan.fougerolle.com>