AGH University of Science and Technology

Faculty of Physics and Applied Computer Science

Bachelor Thesis

RQtWrapper - Cross-Platform ROOT & QT Submerging Framework

Name and Surname: Michał Loska
Studies: Applied Computer Science
Thesis Supervisor: dr inż. Bartłomiej Rachwał

Kraków, January 2020

Oświadczenie studenta

Uprzedzony(-a) o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2018 r. poz. 1191 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony(-a) o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.) „Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta.", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Jednocześnie Uczelnia informuje, że zgodnie z art. 15a ww. ustawy o prawie autorskim i prawach pokrewnych Uczelni przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli Uczelnia nie opublikowała pracy dyplomowej w terminie 6 miesięcy od dnia jej obrony, autor może ją opublikować, chyba że praca jest częścią utworu zbiorowego. Ponadto Uczelnia jako podmiot, o którym mowa w art. 7 ust. 1 pkt 1 ustawy z dnia 20 lipca 2018 r. — Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.), może korzystać bez wynagrodzenia i bez konieczności uzyskania zgody autora z utworu stworzonego przez studenta w wyniku wykonywania obowiązków związanych z odbywaniem studiów, udostępniać utwór ministrowi właściwemu do spraw szkolnictwa wyższego i nauki oraz korzystać z utworów znajdujących się w prowadzonych przez niego bazach danych, w celu sprawdzania z wykorzystaniem systemu antyplagiatowego. Minister właściwy do spraw szkolnictwa wyższego i nauki może korzystać z prac dyplomowych znajdujących się w prowadzonych przez niego bazach danych w zakresie niezbędnym do zapewnienia prawidłowego utrzymania i rozwoju tych baz oraz współpracujących z nimi systemów informatycznych.

.....

(czytelny podpis)

Na kolejnych dwóch stronach proszę dołączyć kolejno recenzje pracy popełnione przez Opiekuna oraz Recenzenta (wydrukowane z systemu MISIO i podpisane przez odpowiednio Opiekuna i Recenzenta pracy). Papierową wersję pracy (zawierającą podpisane recenzje) proszę złożyć w dziekanacie celem rejestracji.

Abstract

The thesis showcases a specific scenario of combining two Frameworks - *Root* and *Qt* - allowing for easier and beneficial application building. At first, it is crucial to understand the idea of what Graphical User Interface is. It becomes clear that while *Qt* offers a highly developed tool for *GUI* building it lacks in the Data analysis and visualisation domain - for which *Root* was created. This join of technologies allows reusing previously written code for a new application - regardless of whether that code was *Qt* or *Root* based. The technical aspects of both Frameworks are showcased and the idea behind the term "submerging" is clarified. Examples of how the combination of both frameworks was achieved are also presented. Ultimately this project was made to allow for hassle-free integration of both environments for developers without the need for a thorough analysis of Frameworks mechanic.

Abstrakt

Praca Inżynierska opisuje metodologię łączenia oraz sposoby użycia dwóch zintegrowanych Framework'ów - *Root*'a oraz *Qt*. Ta integracja pozwala na łatwiejsze i korzystniejsze budowanie aplikacji okienkowych. Na początku kluczowym jest, aby zrozumieć pojęcie Graficznego Interfejsu Użytkownika. Staje się jasnym, że podczas gdy *Qt* jest wysokiej jakości narzędziem do budowania Graficznego Interfejsu Użytkownika, nie pozwala na prostą analizę oraz wizualizację danych - do których oryginalnie został stworzony *Root*. To połączenie technologii pozwala na ponowne użycie poprzednio napisanego kodu w celu tworzenia nowej aplikacji - niezależnie od tego, czy kod ten pochodzi z zakresu *Root*'a czy *Qt*. Aspekty techniczne obu Framework'ów zostają opisane, a logika tytułowego zagnieżdżenia wytłumaczona. Przykłady wykorzystania kombinacji obu Framework'ów także zostaje zaprezentowana. Podsumowując, projekt ten został stworzony, by pozwolić na prostą integrację dwóch środowisk dla użytku developer'ów bez potrzeby dokładnego zgłębiania oraz analizy technologii z osobna- lecz wykorzystania gotowego narzędzia, jakim jest *RQtWrapper*.

Contents

Chapter 1

Introduction

Applications are the programs we all use every day regardless of what platform it is based on. They make lives significantly easier. Many people would not even be able to imagine for a split second what would life look like if population still had to use physical maps instead of Satellite Navigation, not being able to communicate freely through online chats or even something which is becoming more and more common - online shopping.

At this point it may still be unclear what would the difference be between just an "Application" and a "GUI Application". The difference is very obvious to Computer Industry society. At the beginning of computers existence, the way people used them was quite distinct to the current standards. The very first "computer" or more specifically an "Electronic General-purpose computer" was the *ENIAC*. Unlike now- it did not have a keyboard nor a mouse but rather a specially made *punch card* which served a purpose of an I/O (Input/Output) communication between the computer and its operator. Because *ENIAC* did not have any static memory these cards were also used as a way to store data. On top of that, the size of the whole machine exceeded a size of an average house weighing roughly 30 tons with diameters of 2.4m x 0.9m x 30m occupying an area of 167 m^2. Sometime later, as computers evolved and became significantly more powerful, a new way of communicating with the machines arose. These were the applications which had used a terminal (also known as a command prompt or more generally *command-line interface - CLI*) which allowed users to enter given commands, interpret them successively and after that, execute a specific action and display the output in the console. The CLI first concept - called RUNCOM was invented in 1964 by MIT Computation Center staff member - Louis Pouzin. This standard has been a part of the computer world for over 50 years now and it does not seem like it is vanishing any time soon. Till this day it serves, its purpose.

Using text as a way of communicating with a computer is efficient but not in all

cases. With that in mind, Graphical User Interface (GUI) Applications were becoming more and more popular in the 1980s. As Apple released the *Apple Lisa* and IBM released *DOS* (Disk Operating System), it became apparent it was the path engineers had to follow. GUI applications allowed end-users to operate computers with very little knowledge of the highly complex language and instead gave the ability to engage the interaction with peripheral devices by just pointing and pressing buttons. As an example, let us consider a situation in which we would need to copy over files from one location to another. With *CLI* we need the knowledge of specific commands and the right way of executing them, whereas with *GUI* applications it usually only requires a few button presses. This shift from text-based oriented system to graphical ones allowed the society for hassle-free use of Computers. Both systems provide the same functionality but what varies is the different effectiveness and the final effect. However, even nowadays using a computer seems challenging for some people with the availability of various "easy-to-use" tools. Using a command prompt is still limited to a very small group of specialists. Because of that, a new trend has begun which allowed the applications for more approachable, better looking and easier to use systems. This tendency has allowed us to use our *PC*s (Personal Computers) freely for average needs. As this trend evolved and GUI application programming became popular and within people's reach, free tools were created. In this case, a tool could be a number of things, such as an *IDE* (Integrated Development Environment) or a suitable Framework. The thesis is focused on the second one. What Framework is, is a packed abstraction which provides a generic functionality which can be freely used by the end-user with no or little restrictions. Many of them were strictly made for a certain purpose and some were for general usage. As per usual - we cannot always achieve every desirable outcome without any sacrifices. Having said that it might hint why it could be beneficial to combine good sides of two frameworks such as *Root*[1] and *Qt* [2]. Both of them were made for creating *GUI* applications but for slightly different final purposes.

The ambition of this Thesis is to showcase the process, the upsides and the downsides and the technical implementation of merging of building applications with either of the frameworks and both combined. In the following chapters and sections *Qt* and *Root* will be presented with more detail as well as the final merging product - *RQtWrapper* which is based mainly around a pre-made code found on *Root Community Forum*[3]. The main motivation of creating a tool like this was to allow creators to use pre-built applications in either of the before mentioned frameworks and allow for easy integration. This approach increases the reusability of already implemented programs in Root and/or improving data analysis and visualization features within a *Qt* GUI Applications without needing to ditch the current project or switch technologies.

Chapter 2

Frameworks Basics

2.1 QT

Qt is a GUI creation open-source toolkit which because of its cross-platform capabilities makes it undoubtedly versatile, allowing programmers for creating applications for many platforms with just one development cycle. It supports platforms such as *Windows*, *macOS*, *Android* and embedded systems (figure 2.1). The key feature is that while we may be creating our application with just one platform in mind it will ultimately be compatible with multiple, with little or no change in the underlying codebase making it a proper native application with its native capabilities and performance.

Figure 2.1: Same Application on four various platforms [4].

This approach may not seem popular but the list of big companies using *Qt* for their application development is long and impressive and consists of: *Adobe Photoshot Album*, *Autodest Maya*, *Google Earth*, *TeamViewer* and many more.

Qt not only provides a library for easy GUI application creation but also a self-sufficient tool called *Qt Creator*. It serves a purpose of a tool which takes application creation from the level of written code with a lot of abstraction to a level of a Graphical application. Having said that it is known as a GUI Application made for creating other GUI applications. Knowledge of programming languages becomes of lower importance and becomes more accessible.

Let us consider a simple *Qt* made application. It seems just like a normal Notepad app most people are familiar with and actually it is one. It proves the point that creating simple applications is an easy task to do and many features already pre-defined within *Qt*. In this case, the application was written top to bottom in *Qt* and the same effect can be achieved on all available platforms.

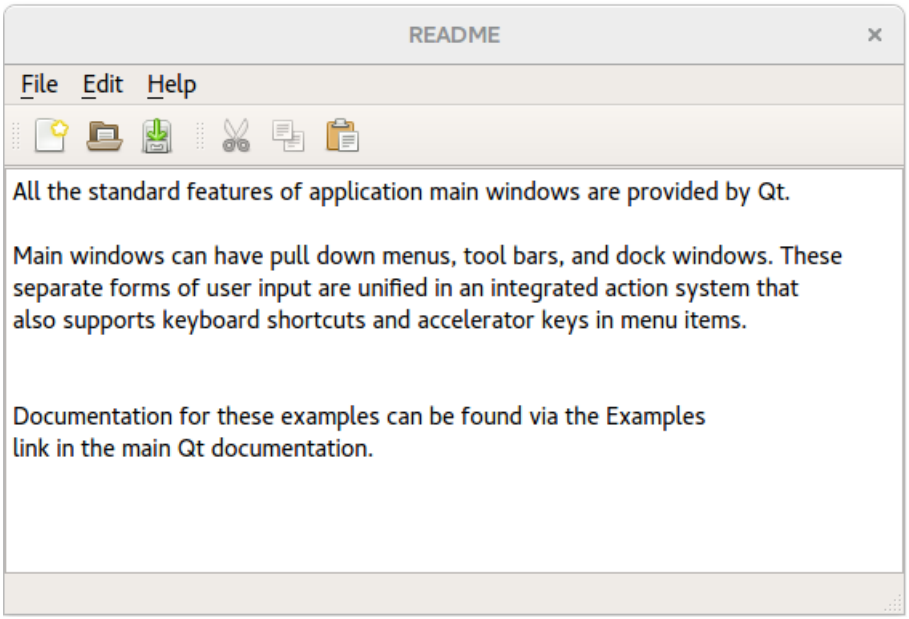

Figure 2.2: An example of a simple *Qt* application [5]

Creating Menu bars, Taskbars and other familiar solutions are a built-in option which is ready to be used or even dragged into the design pane in *Qt* Creator.

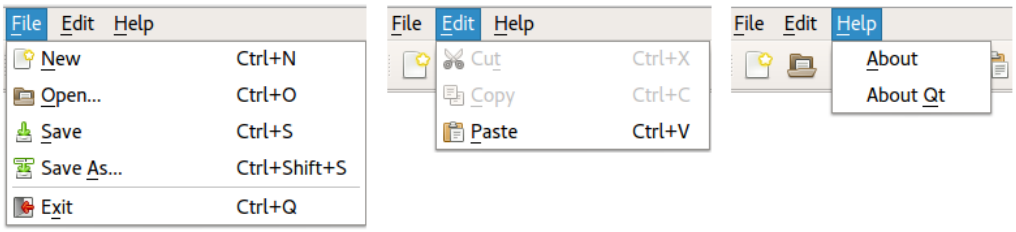

Figure 2.3: Menu bar of the *Qt* example application [5]

2.2 Root

Root is a modular scientific software toolkit developed by *CERN*(Conseil Européen pour la Recherche Nucléaire)[6]. At first, it was designed for particle physics data analysis but now it's purpose has grown and allows users for easy manipulation and analysis of big data, processing, statistical analysis, visualization and storage[7]. The library was maintained in *FORTRAN* for a long period of time until 2003 when it got displaced by *C++*. The main advantage of *Root* is that it has independent access to computers graphics subsystem and the operating system. As a whole platform, it provides a GUI and a GUI builder, container classes and a self-reliant *C++* script and command-line interpreter. While the built-in visualization and data analysis are at the highest level of finesse, the GUI builder it provides plays a very basic role in the whole package.

Focusing more on the actual usage of *Root*, its application can be immediately seen in many industries. Starting from medical data analysis of various medical examinations, through the automotive sector for data analysis and plotting of live data feeds up to Aviation for airflow analysis. All of these applications are what the *Root* framework was designed for.

Let us consider Spirometry as an example. It is the most common medical examination of breath and lung functioning - more specifically, the volume and/or flow of the air during an inhale and exhale. A successful test can be conducted by graph analysis which often is being generated as patient breaths. *Root* Framework provides all the tools needed for the implementation of such an application.

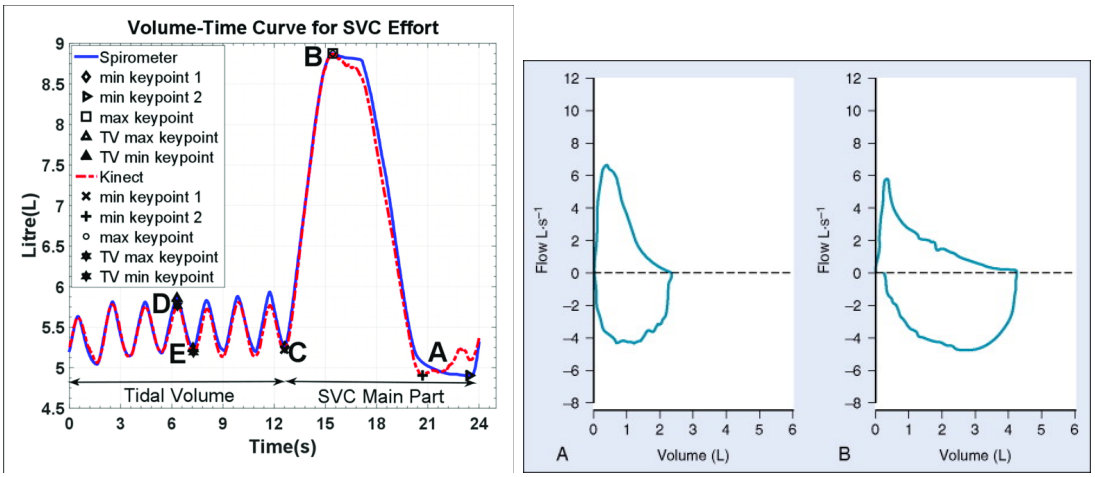

Figure 2.4: An example of a spirometry analysis app implementable in *Root*

More scientific use cases where *Root* has its application is still strictly connected with High Energy Physics, Astrophysics, Medical Physics and many many more. Let us have a look at examples of what *Root* was originally designed for:

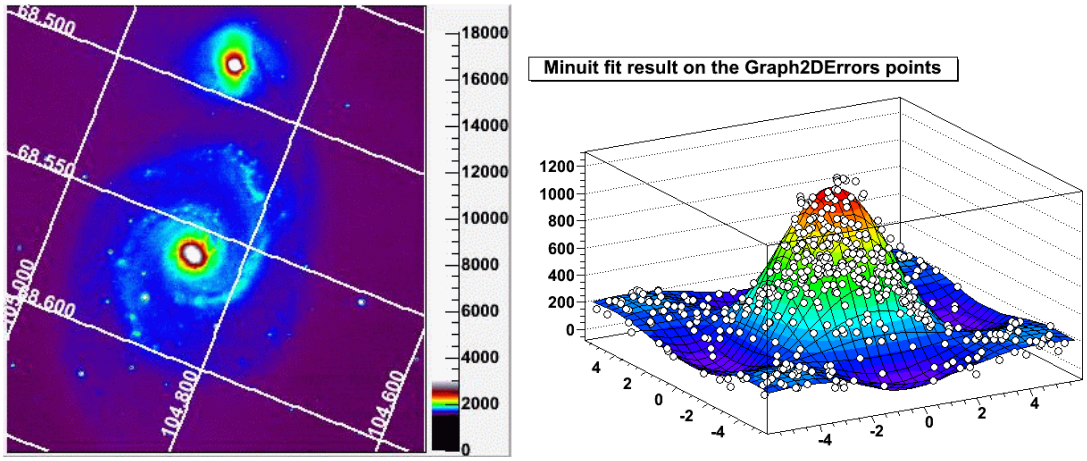

Figure 2.5: Astrophysics visualization of the Galaxy[8] and a Minuit Fit[9]

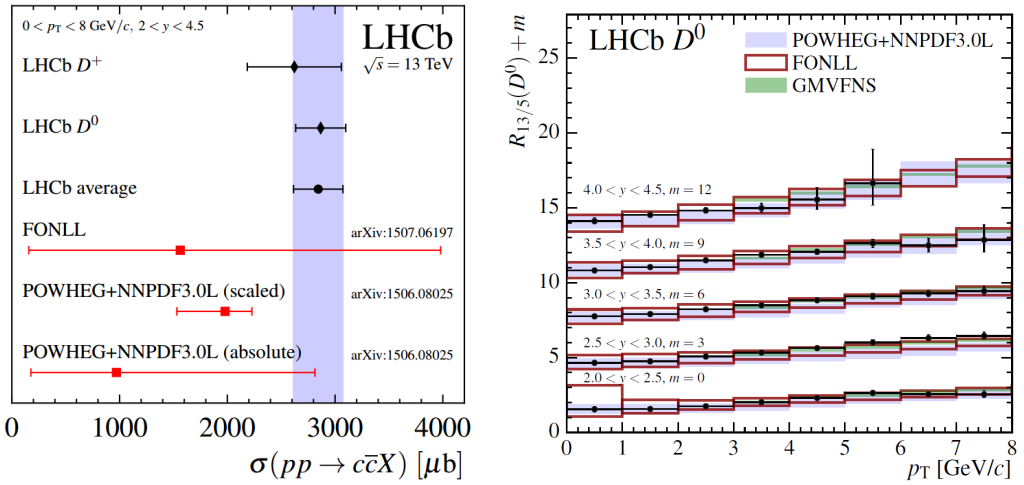

Figure 2.6: Visualization of High Energy Physiscs applications[10]

2.3 Motivation behind combining of the frameworks

Having analysed both Frameworks/Libraries should be enough to grasp the idea of their purpose. *Qt* seems more general and for generic GUI application building and *Root* while it also gives the ability to build GUI it seems more oriented around the Data Analysis and visualization. At this point, it may be clearer why the combination of the two makes sense. It will allow the end-user to create a GUI application combining two strengths of both libraries making for a great mix of refined and tested solutions. In this case, the merge is not interchangeable, but one way only by sub-merging *Root* capabilities into *Qt*'s GUI.

The following Chapters cover the technical aspects of how the sub-merge was achieved and what are the limitations and upsides of this implementation

Chapter 3

Frameworks Architecture

Every Technology has its flow and procedures - QT and Root are no different. To understand how the sub-merging was accomplished we first have to analyze both of the frameworks separately. Their basics as well as the specifics connected strictly with *RQtWrapper*'s implementation. The idea of a program executing commands line by line is fairly straightforward- The program starts, executes the commands, asks the user for input, reads from a file, outputs data to the console and once everything is done the program finishes. With GUI applications this approach is rarely the same. Imagine any other Window application you use on a daily basis. An application which allows users to check bus time schedule does not close just after having looked up a single time for a given bus. It allows the user to gain access to it as long as desired.

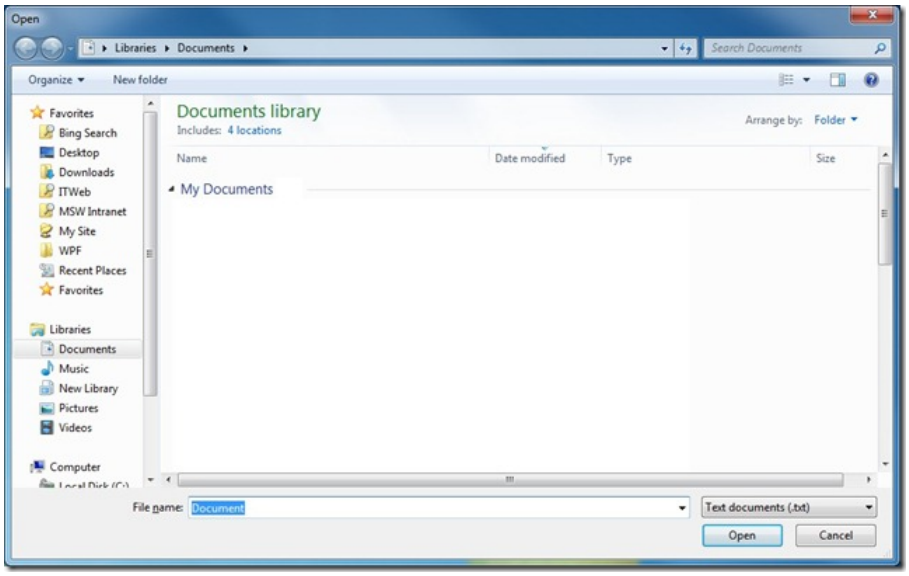

Figure 3.1: Windows File Dialog window [11]

Having looked at Figure 3.1 [11] we can see that this window will stay opened and

at the top until we proceed with a specific action. This action is usually called an *Event* within this context. In our case, the application is waiting for a button press which will *handle* a certain action.

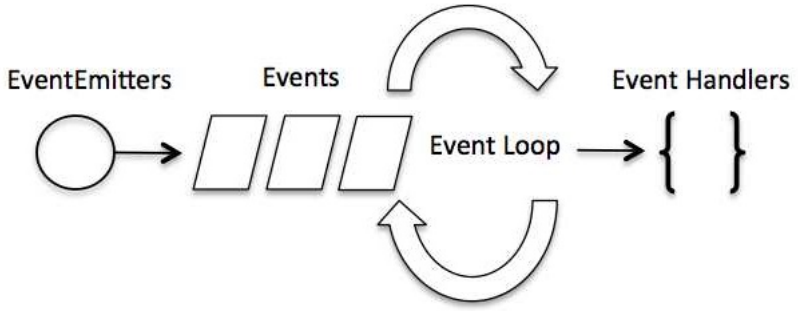

Figure 3.2: Event loop in GUI applications[12]

All this is managed by the Event Loop. Usually, an event is emitted at one point of the program and handled (received) in another part of the program. In Figure 3.2 we can see that dependency. One element emits the signal which is then queued up and once it goes to the Event Loop it eventually ends up handled in the Event handler.

To put all of this in simple terms: An event loop is exactly what it's called. We could even imagine it as an infinite while loop which executes until the user presses the *"X"* button in the window which could be interpreted just like a *break;* would in *C++*. Event Emitter could be something as simple as a met condition or a button which when pressed emits a signal. The signal, on the other hand, could be understood as a connection between the *Event Emitter* and the *Event Handler*. The Handler is usually just a function which is only called when the event is triggered and executes predefined actions.

Another important matter to remember about in Window Applications is the hierarchy. Every application is divided into smaller portions such as Windows, panes, tabs or buttons. All of them need to correspond to the hierarchy by owning a parent. Every element has it's own parent and the whole hierarchy has a master-parent which is responsible for all of the actions during the run time of the application. The life span of these individual elements is strongly dependant on the other elements. We cannot (or we should not be able to) close the main window while we still have a warning pop up dialogue opened. That would ruin the hierarchy by deallocating the parent before the child expires. The whole process is very similar to the idea of a FIFO (First In - First Out) stack. You cannot remove a second element from the top until the top gets removed.

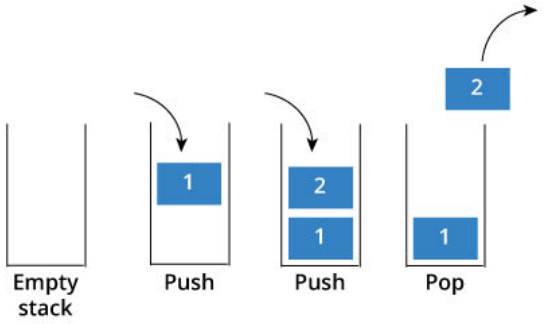

Figure 3.3: Graphical representation of a FIFO stack [13]

For the sake of an example let us assume that the element number 1 pushed onto the stack is the main window of our GUI application. During the use of our application, we decide to close the main window but our work so far has not been saved. A warning dialogue appears telling us to either save our work or close without saving. That warning dialogue is the element number 2 that is being pushed onto the stack. As the graphical representation shows- we cannot close the main window until we close the dialogue first. The very right part of Figure 3.3 shows the moment of closing the warning dialogue allowing us to exit out of the application by closing the main window.

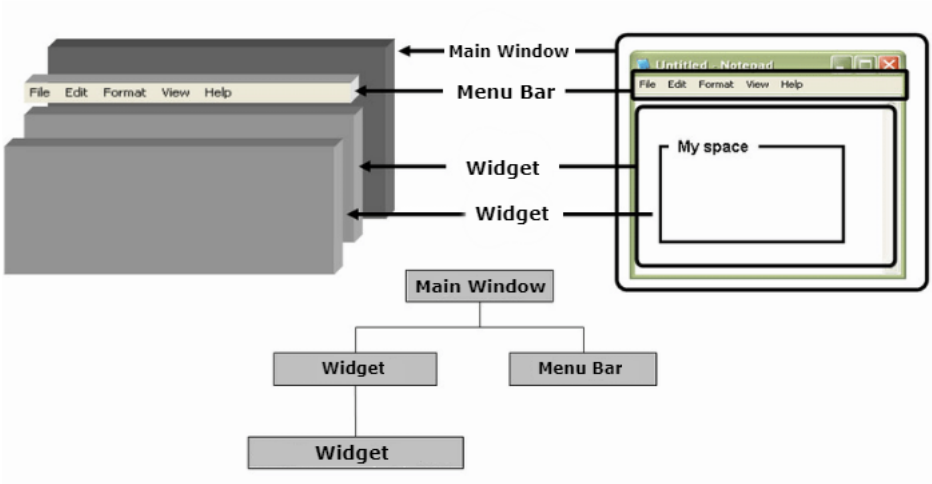

Figure 3.4: Correspondence of GUI to class hierarchy [14]

The above image (figure 3.4) represents the correspondence of the GUI elements as well as the class and parent/child hierarchy

3.1 QT

Every program, library or framework has its hierarchical beginning and a start point. For any *C++* written code it usually is the well known main function. In the case of *Qt* it is no different. Let us have a look at a piece of code which resembles the minimal needed source code for a *Qt* application:

```
1 #include <QApplication>
2
3 int main(int argc, char **argv){
4  QApplication app (argc, argv);
5  return app.exec();
6 }
```

Listing 3.1: A minimal Qt Application [15]

In this case, calling the above function an application may be an overstatement. The reason behind why is very simple. If the code from the Listing 3.1 was run it would be apparent that absolutely nothing would show up. Unlike with the next listing:

```
1 #include <QApplication>
2 #include <QPushButton>
3
4 int main(int argc, char **argv){
5  QApplication app (argc, argv);
6
7  QPushButton button ("Hello world !");
8  button.show();
9
10  return app.exec();
11 }
```

Listing 3.2: A minimal Qt Application with a visible window [15]

As the effect of running the code from Listing 3.2 the following window will appear:

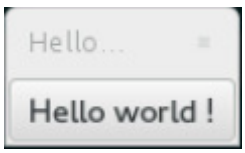

Figure 3.5: Simple app with a button only [5]

The image from figure 3.5 is in its original size. What has happened here is fairly simple. We have created an instance of background manager(*QApplication)* and an instance of a button (*QPushButton*) and we explicitly called a *show()* method on it forcing it to become visible. In order to comprehend further steps of advanced GUI building we have to understand the difference between *QApplication* and *QPushButton*.

3.1.1 QApplication

QApplication is a very important - if not the most important - class in *Qt* GUI application building. It is not strictly connected with the graphical or physically visible part of an application. It plays the role of background and more Back-end element. It handles input arguments, widget specific initialization and finalization and finally, it takes care of the before-mentioned Event Loop. It is important to note that there is always just one *QApplication* instance in every app - no matter how many windows it has. That said, we can assume that *QApplication* could also be called a brain of a GUI application - we cannot see it but it does all of the needed operations in the background without us noticing it.

3.1.2 QWidget

Following the structure of the previous section, this one should be about the *QPushButton*. Instead, it is about the *QWidget* which is a broader class from which a lot of others derive (such as *QPushButton*).

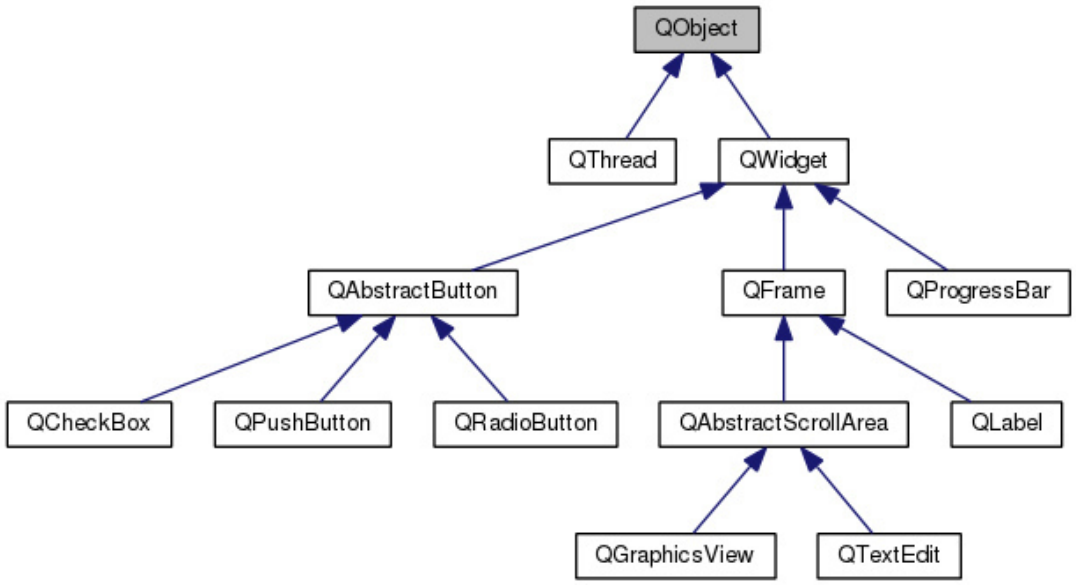

Figure 3.6: Basic Qt classes hierarchy [5]

Having analyzed Figure 3.6, we see that *QPushButton* derives from *QWidget*. This means that every other class inheriting from *QWidget* could be replaced with the button from Listing 3.2. An application with a *QLabel*, *QCheckBox* or a *QProgressBar* would also work and look correspondingly.

While the *QApplication* plays the role of the brain, any *QWidget* based class could be our eyes and ears - or more generally- a sense. It allows us for the Interaction with the program and the back-end of the code. *QWidget* technically speaking plays the role of a plain container and a potential parent widget in bigger applications where the need for nesting multiple Text Edit fields or buttons is required. We can achieve that by specifying a parent of the newly created object:

```
1 QWidget(QWidget *parent = nullptr,
2         Qt::WindowFlags f = Qt::WindowFlags())
```

Listing 3.3: QWidget constructor declaration [15]

By default the parent is *nullptr* which in simple terms means that the new element (ex. QPushButton) is not associated with any other parent and is not a part of an already existing structure/hierarchy. That matter will be highlighted in later sections.

3.1.3 QMainWindow

QMainWindow is a class which provides a toolkit or a framework on its own for building GUI applications. It is specifically useful for more advanced applications with elements such as Menu Bars or Tool Bars. It can be thought of as a wrapper for content that is already standardized because of its predefined layout.

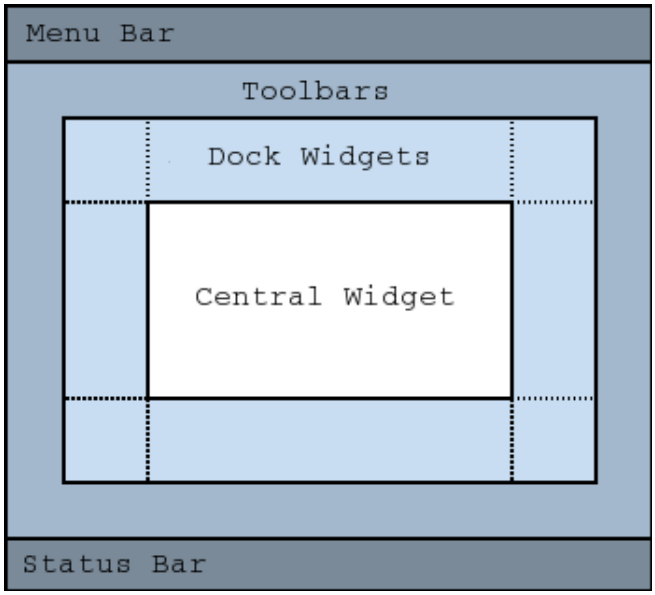

Figure 3.7: *QMainWindow* Layout [16]

On top of that, it also provides a place for any kind of widget in the middle. Setting a widget as a central one is not required but provides the layout which fits it to the main window instance.
The approach of using *QMainWindow* seems natural even when taking its name into consideration. It even implies it should be the **Main Window** of any project and be responsible for managing the children within the hierarchy. In the following part of the Thesis, we will be using the *QMainWindow* as a standard in every example.

```
1 class MainWindow : public QMainWindow {
2 public:
3     MainWindow(QWidget *parentWidget = nullptr);
4 private:
5     QTextEdit * m_textEdit;
6     QPushButton * m_button;
7 };
```

Listing 3.4: QMainWindow derived class [15]

With our MainWindow class defined we can continue with instantiating it and the *QApplication*. At this point we assume the *m_textEdit* and *m_button* members were initialized in this manner:

```
Window::Window(QWidget *parentWidget){}
    ...
    QWidget * centralWidget = new QWidget(window);
    window->setCentralWidget(centralWidget);
    m_textEdit = new QTextWidget(centralWidget);
    m_button = new QPushButton(centralWidget);
    ...
}
```

Listing 3.5: Part of *Window* Constructor definition [15]

We can now see the before mentioned structure of objects making up for one big logical hierarchy. This allows *Qt* to manage its resources and to always know which window is on the top. This is particularly useful for memory deallocation.

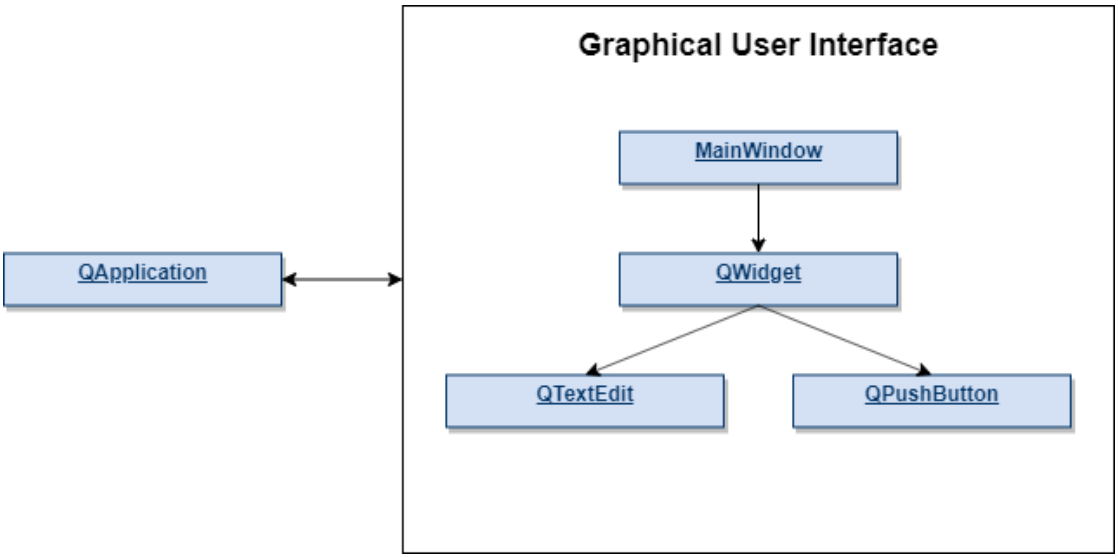

Figure 3.8: Application class hierarchy

The final outcome of the application:

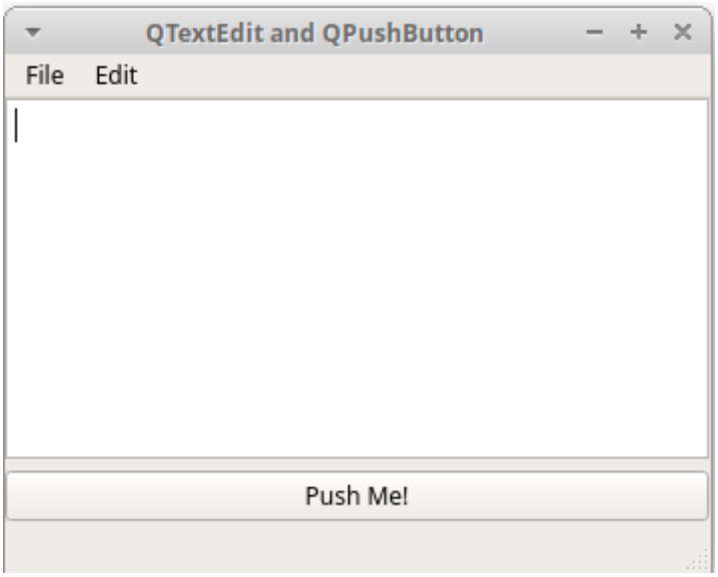

Figure 3.9: Application with QTextEdit and QPushButton based ontop of QWidget within QMainWindow

At this point, we may assume that the "Push Me!" button is implemented further and can print out "Hello World!" text to the Text Field Above it. The main function workflow:

```
1 int main(int argc, char **argv){
2     QApplication app (argc, argv);
3
4     Window window;
5     window.show();
6
7     return app.exec();
8 }
```

Listing 3.6: Main function [15]

3.1.4 QLayouts

QLayout is an abstract base class inherited by the classes *QBoxLayout*, *QGridLayout*, *QFormLayout*, and *QStackedLayout* [17]. Its main purpose is to provide its basic functionality such as setting size constraints, adding items (widgets) in a specific order and the key feature to this thesis - the automatic size fitting. Because of many (potential) nested widgets within themselves it is always important to take care of the sizes of elements in the hierarchy. In our case, every widget had to be set to the size of the preceding widget making the proportions constant throughout the whole lifetime of the program. Hierarchy is the key to success with *Qt* and it is even more significant when *Root* is submerged within *Qt*. When building an app every element needs to be in the right place of the hierarchy (the tree) in order to achieve the desired outcome. *QLayouts* take this rule with a pinch of salt causing a lot of confusion. There is a verity of types of *QLayout*. The most common ones are:

- QVBoxLayout - arranges inner elements Vertically
- QHBoxLayout - arranges inner elements Horizontally
- QGridLayout - arranges inner elements in a from of a Grid
- QFormLayout - arranges inner elements in form of a (2 column) Form

There is one missing part in Listing 3.5. That missing part is the *QLayout Implementetion* which allowed for the clean look and arrangement of the *QTextEdit* and the *QPushButton* instances inside of the *QMainWindow*. These three lines may seem unnecessary but if it was not for them the application would not look smart at all, but the elements in the centralWidget could be overlapping. The missing implementation looks like in the following listing (listing 3.7).

```
1  Window::Window(QWidget *parentWidget){}
2  ...
3  QWidget * centralWidget = new QWidget(window);
4      window->setCentralWidget(centralWidget);
5      m_textEdit = new QTextWidget(centralWidget);
6      m_button = new QPushButton(centralWidget);
7      ... //Missing part:
8      QVBoxLayout * layout = new QVBoxLayout(centralWidget);
9      layout->addWidget(m_textEdit);
10     layout->addWidget(m_button);
11 }
```

Listing 3.7: Part of *Window* Constructor definition with QLayout [15]

3.2 Root

Root Framework gives us a lot of flexibility with both GUI building and data analysis and Visualization. It is no surprise that while *Root* is superb at what it was originally designed for, it is lacking in the GUI building scope. Hence why we will only focus on the scientific usage of the Framework and its integrability with *Qt* in the next chapter.

```
#include "TCanvas.h"
#include "TH1D.h"

void DrawHistogram(){
    TCanvas * canvas;
    canvas->Clear();
    canvas->cd();
    canvas->SetBorderMode(0);
    canvas->SetFillColor(0);
    canvas->SetGrid();
    gStyle->SetOptStat(0);
    auto h1 = new TH1D("Gaus Histogram","Gaus Histogram", 50, -5, 5);

    TRandom3 r;
    for (int i=0;i<10000;i++) {
        h1->Fill(r.Gaus(0,1));
    }
    h1->Draw();
    h1->GetXaxis()->SetTitle("x");
    canvas->Modified();
    canvas->Update();
}
```

Listing 3.8: Code used for Histogram generation

Having added *Root* into our project we can create a very simple Histogram visualization of Gauss Distribution. It is noticeable that the amount of code written to achieve the final effect is low, making *Root* efficient in use.

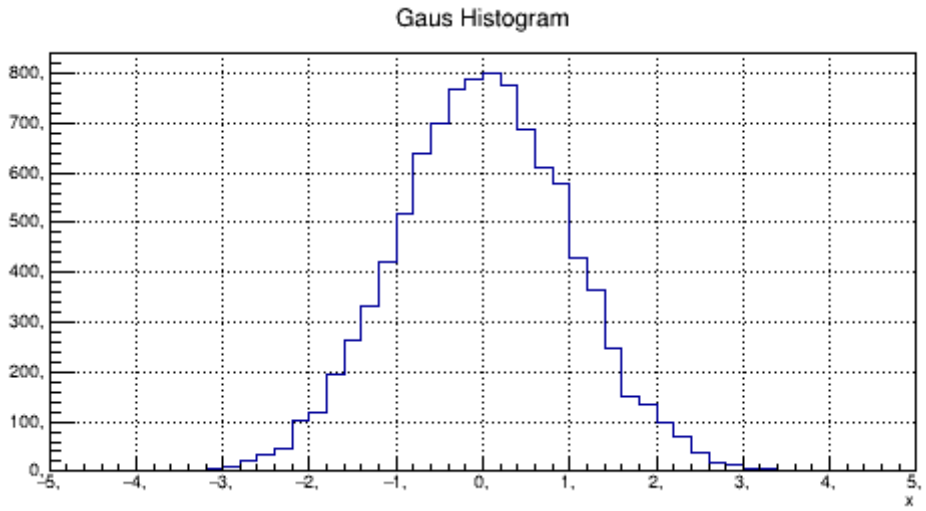

Figure 3.10: Gauss Distribution visualization from Listing 3.8 created in *Root*

The graph has been drawn on an instance of *TCanvas* which provides a wide range of tools for customization for any type of needs.

Let us look at an example showcasing an application with no scientific background but just a plain calendar application. It is a very simple program with very little implemented functionality but it proves that normal GUI applications can be created using *Root*.

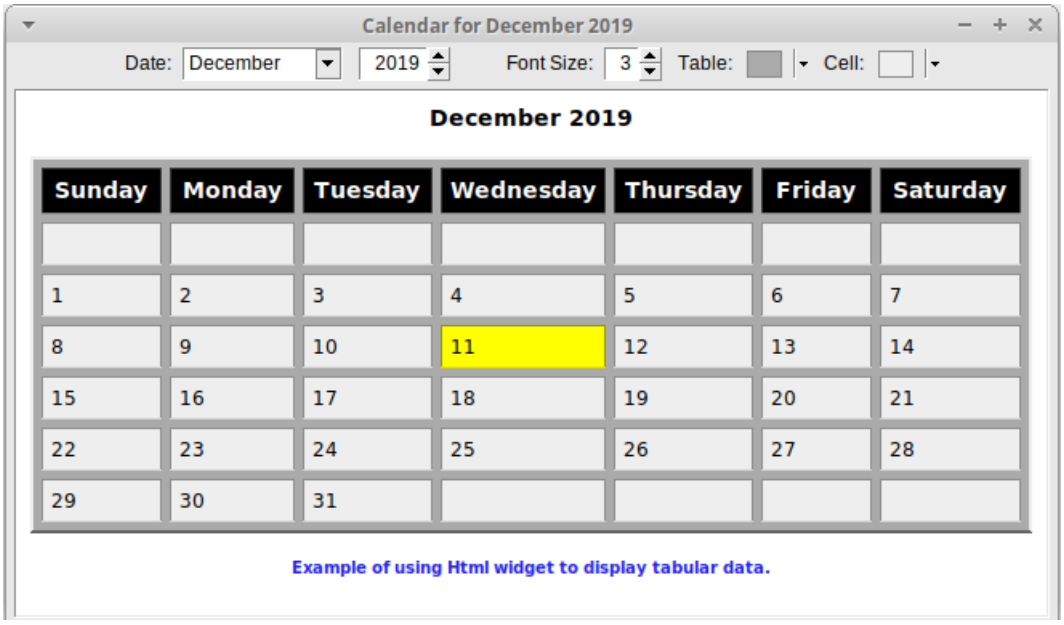

Figure 3.11: Calendar app in *Root*[18]

3.2.1 TApplication

TApplication is a crucial part of any *Root* based program. Following the scheme of the previous chapter, it is suitable to refer to it as the brain of the Framework[19]. This Class creates the necessary Application Environment that interfaces to the windowing system and it is responsible for the greatly important Event Loop and the Event Handling. As expected it can be instantiated only once.

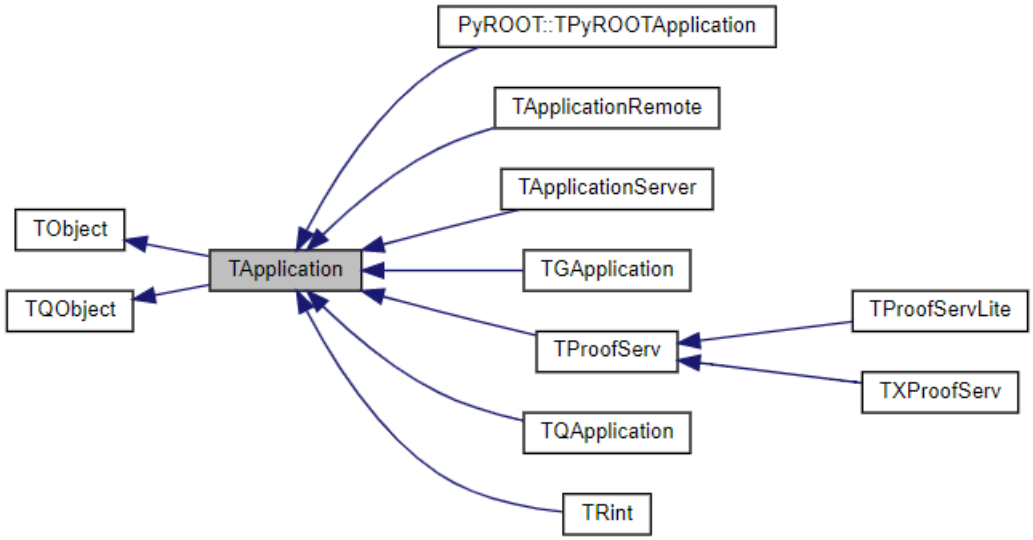

Figure 3.12: Inheritance diagram of TApplication *Root*[19]

Looking at Figure 3.12 even without having analysed it thoroughly, it is clear how important *TApplication* really is. Just like in the case of *Qt*'s *QApplication* it is an entity that only exists in the background of the program which by any means does not degrade its importance. It plays the role of an interconnecting bridge between the operational power and its management and the end-user with the use of GUI.

3.2.2 TCanvas & TPad

TCanvas while it may just be the most important class to focus on within the whole thesis it is also a very complex and important element. Technically speaking it is an area mapped to a window which supports Drawing and Painting plots, histograms etc. within itself. Any *TCanvas* based space can be subdivided into in independent graphical areas called *TPads*. Being more specific the drawing process takes place on a *TPad* instance which is built into every instance of *TCanvas*. Every Canvas has at least one Pad. This provides an ability to display two separate plots in just one window while being autonomous and self-sufficient. Adding a new element into a pad is generally performed by the Draw method of the object

class. It is important to realize that pads are linked to a list of references to the original objects. What it means in practice is that when we call the Draw method on an instance of a Histogram the operation only stores a reference to the histogram object and not its graphical representation. For example when we perform a change to the histogram the actual content of the histogram object is changed - not the TPad's [20].

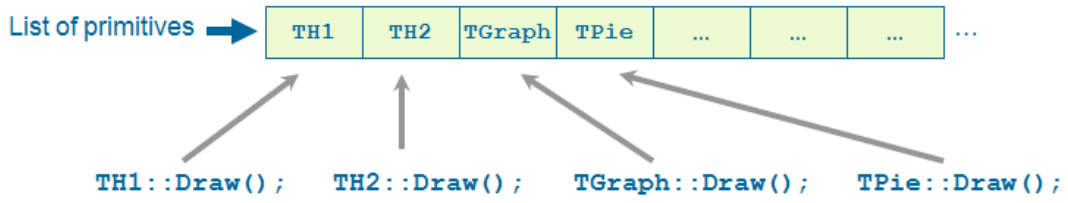

Figure 3.13: Tpad's list of drawable primitives [20]

The Convention used in *Root* is that the Draw method only adds a reference to the object while the actual drawing is performed when the canvas receives a signal to be painted. When a Canvas or a Pad is repainted, the member function Paint is run recursively for the remaining nested Pads. This signal is emitted in many other places such as the Update method in *TCanvas* which performs it for all of its pads [20].

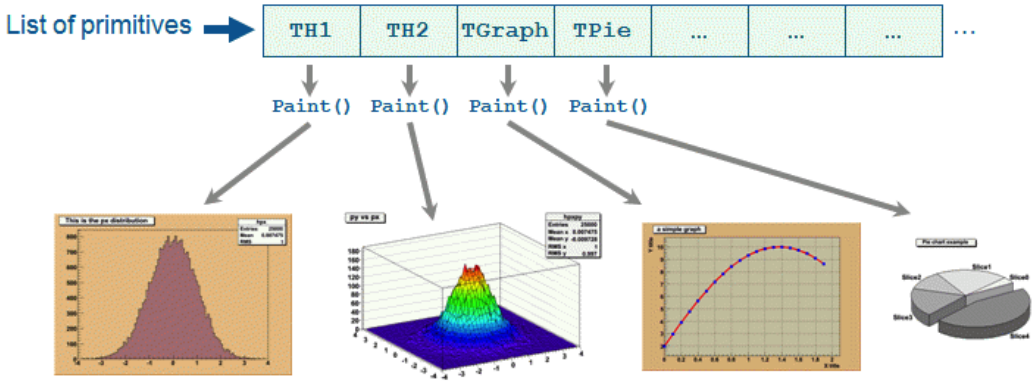

Figure 3.14: Tpad's list of drawable primitives 2 [20]

```
void DrawHistogram(){
    TCanvas * canvas;
    canvas->Clear();
    canvas->cd();
    ...
    auto h1 = new TH1D("Gaus Histogram","Gaus Histogram", 50, -5, 5);
    ...
    h1->Draw();
    ...
    canvas->Modified();
    canvas->Update();
}
```

Listing 3.9: Part of the code used for Histogram generation

Analyzing the previously used Listing we notice that there is a lot of operations done to the canvas instance. Most of them refer to the built-in TPad or TPads- depending on their amount. As mentioned before we can see previously mentioned functions:

- *Clear()* - clears the contents of all the contained *TPad*s.
- *cd()* - returns a pointer to a *TPad* to be drawn on. If no index specified it returns the very first one.
- *Draw()* - adds a reference of the *h1* to the *canvas*.
- *Modified()* - emits a signal handling specific TPad functionalities respectively.
- *Update()* - Paints or Repaints all of the TPads contained by the *canvas*.

Let us see an example of a double TPad Application (figure 3.15).

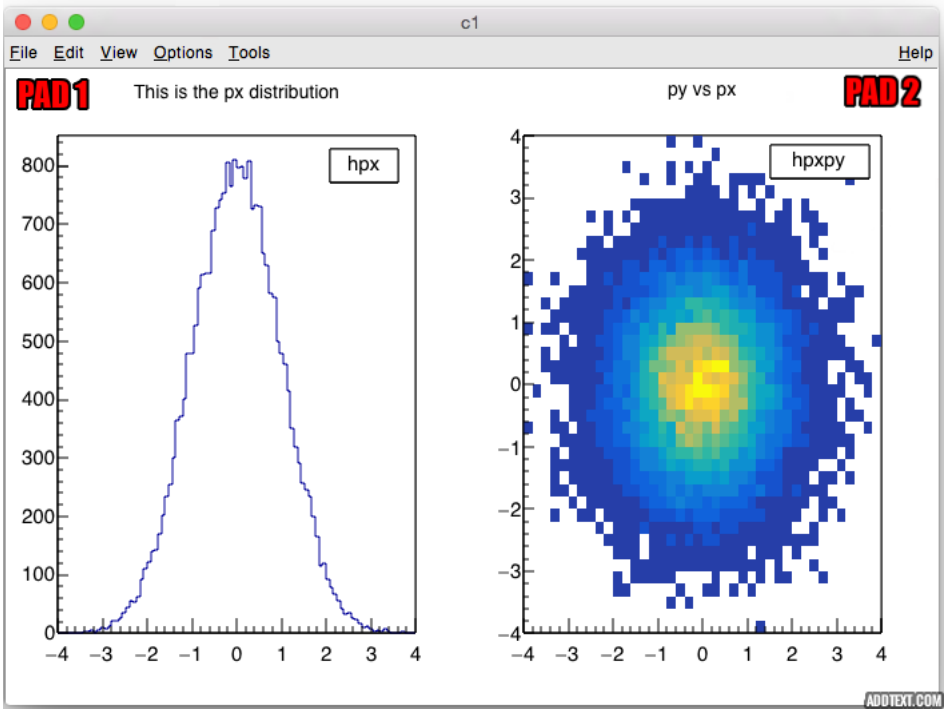

Figure 3.15: TCanvas with double TPad [20]

Chapter 4

RQtWrapper Architecture

Having gotten familiar with the pure foundation of GUI applications and specifics of each Framework we may proceed to the main part of the thesis. The main goal of creating the *RQtWrapper* was allowing the developers to:

- Reuse their already existing Qt GUI and implement a *Root* application taking advantage of it's functionalities.
- Implement existing *Root* application (most likely some kind of a Visualization) into a new GUI.
- Build a new application form scratch while maintaining the best of both worlds - the great experience of *Qt*'s GUI building and *Root*'s data analysis and visualisation.

Regardless of the final form of application, the aim was to make the combination of the Frameworks as easy as possible. Thus far the before mentioned combination has never been properly specified. In nature the way these two Frameworks live with each other is by merging *Root*s fairly small application within a possibly big and advanced *Qt* based GUI. This enforces the "sub-merging" naming, meaning that the *Root* based visualisation application is only a part of a bigger *Qt* application.

The picture below this paragraph reflects a **concept only** of RQtWrapper's potential use case. The combination of a clear and easy to maintain GUI and a useful and fully packed visualisation is an appealing conception.

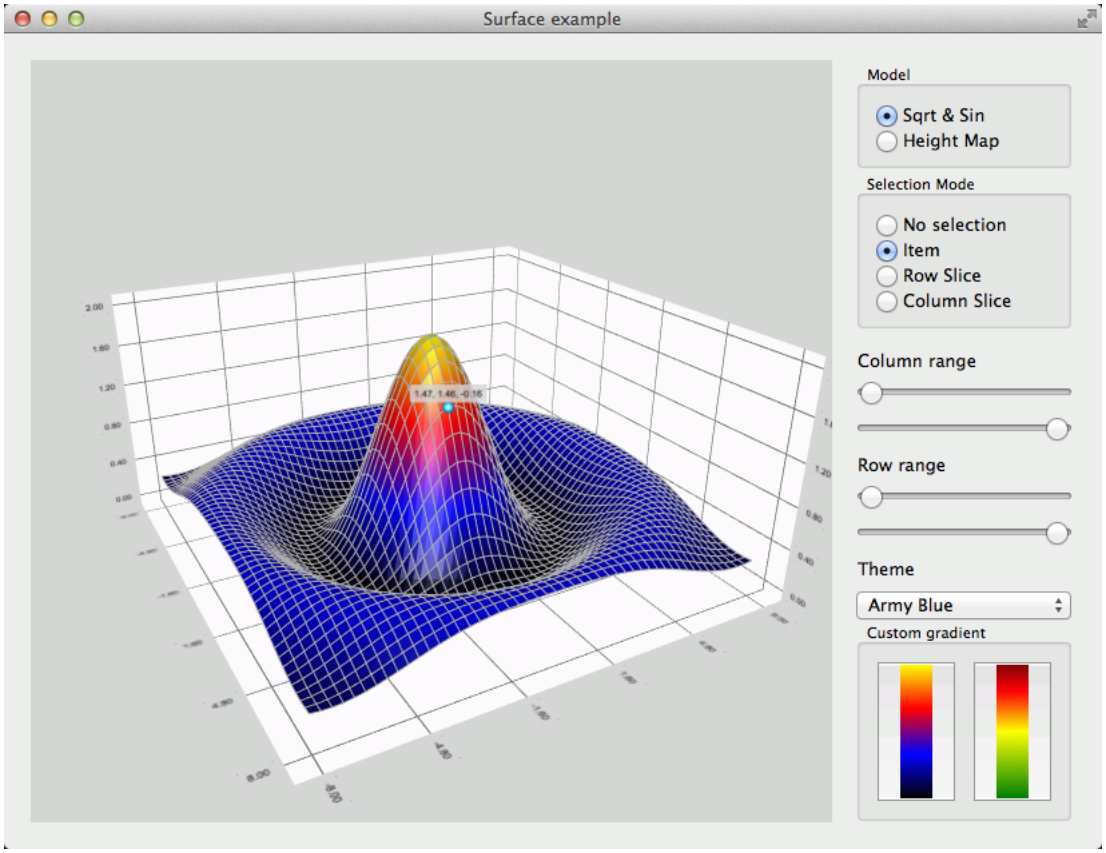

Figure 4.1: Concept of an application using RQtWrapper [21]

In the following chapters, the technical aspects of the implementation will be covered.

4.1 RQtApplication

RQtApplication is a class which unites the functionalities and capabilities of the corresponding classes: *QApplication* and *TApplication*. It merges them into one class allowing for a simple and clear instantiation without fear and uncertainty of any incompatibility. It becomes the ultimate brain of RQtWrapper based applications. Just like with the base Frameworks it becomes responsible for monitoring and managing the back end of the application. Both of the *QApplication* and *TApplication* management systems which are hidden behind the cover of *RQtApplication* work separately and as expected when on their own. They still manage independent event loops, event handling and memory allocation and deallocation.

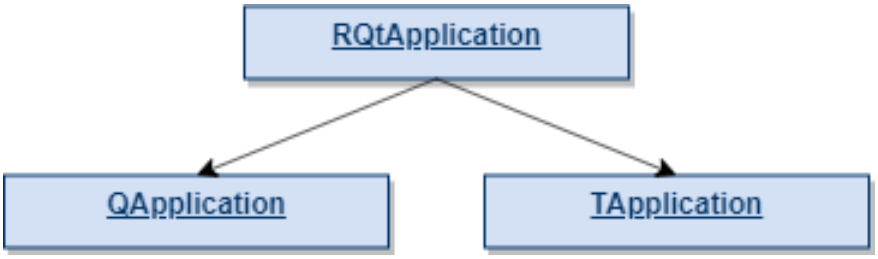

Figure 4.2: RQtApplication Inheritance Diagram

The way it was technically accomplished was by making *RQtApplication* derive from both *QApplication* and *TApplication*. This way all of the constructor parameters get passed into constructors of each base class respectively. On top of that, we needed to make sure *Root*s inner loop could be propagated. In order to achieve that a *QTimer* class was used. It provides repetitive and single-shot timers which emit necessary signals. In this case, the respective signal would be the *Root*'s ProcessEvent() which handles all of the currently pending events such as timers or sockets.

```
RQtApplication::RQtApplication(..)
   : QApplication(...), TApplication(...){
    timer = new QTimer(this);
    QObject::connect(timer, &QTimer::timeout,
                     [](){gSystem->ProcessEvents();});
    timer->setSingleShot(false);
    timer->start(20);
}
```

Listing 4.1: Root Inner Loop integration

The above solution allowed for processing *Root*'s events every single 20ms according to *Qtimer*'s expiration rate.

4.2 RQtWidget & RQtCanvas

RQtWidget and *RQtCanvas* despite their contradictory naming are strictly inter-connected. While both derive from the same base class - *QWidget* their purpose is slightly different. The *RQtWidget*, as the name suggest serves a purpose of a widget- or more specifically a container for the *RQtCanvas*, therefore also for our drawables.

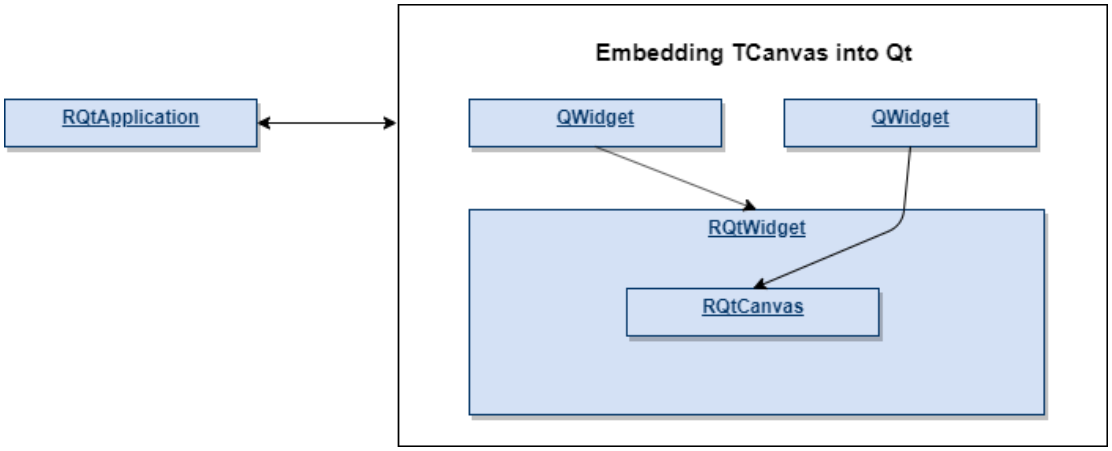

Figure 4.3: *RQtWidget* and *RQtCanvas* inheritance Graph

As can be seen in figure 4.3, once again the *RQtApplication* and the *RQtWidget* combined with *RQtCanvas* are separated visually but they cooperate during the whole lifetime of a program. Going a step deeper it is noticeable that *RQtCanvas* is, in fact, a class member of the *RQtWidget*, and its creation is done by its parent constructor.

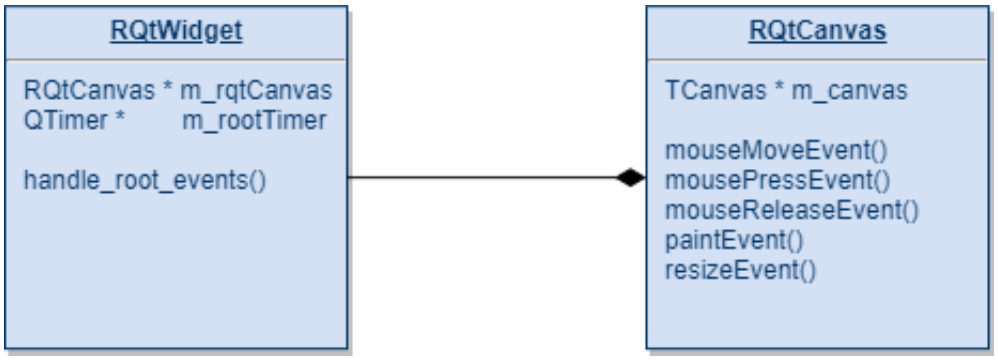

Figure 4.4: Relation between *RQtWidget* and *RQtCanvas*

While RQtWidget takes care of:

- Handling initiation of *Root* Events (ex. automatic resizing).
- Being base container for the app structure and it can be implemented in *Qt* code.
- Creation of RQtCanvas.

At the same time *RQtCanvas* is responsible for holding the actual instance of *TCanvas* and handling events corresponding to Mouse movement, button press, release, painting and resizing. RQtWidget being a container inheriting from *QWidget* can be treated just like any other QWidget within the program structure. This means that whenever we finish designing our *Qt* based GUI we can simply add an instance of *RQtWidget* on top of it, provided it has received a correct parent into its constructor. It is crucial for the functioning of *RQtWidget*.

```
1 ...
2 QWidget * mainWidget = new QWidget(QMainWindow instance);
3 RQtWidget * embeddedRQtWidget = new RQtWidget(mainWindow);
4 RQtWidget * outterRQtWidget = new RQtWidget();
5 ...
```

Listing 4.2: Root Inner Loop integration

GUI applications usually implement a concept of passing a parent to a child object which appends them to the already existing hierarchy. If we skip that and create an object without providing a parent it is usually successfully created but it's not added to the hierarchy. It creates a new structure of objects on its own.

Code from the listing 4.2 is no different in this matter - *embeddedRQtWidget* will get added into the already application hierarchy while *outterRQtWidget* will not and it will exist as an outer, totally separate entity from the original application. This will generate another window apart from the main one. This exact scenario will be covered in the first section of the next Chapter.

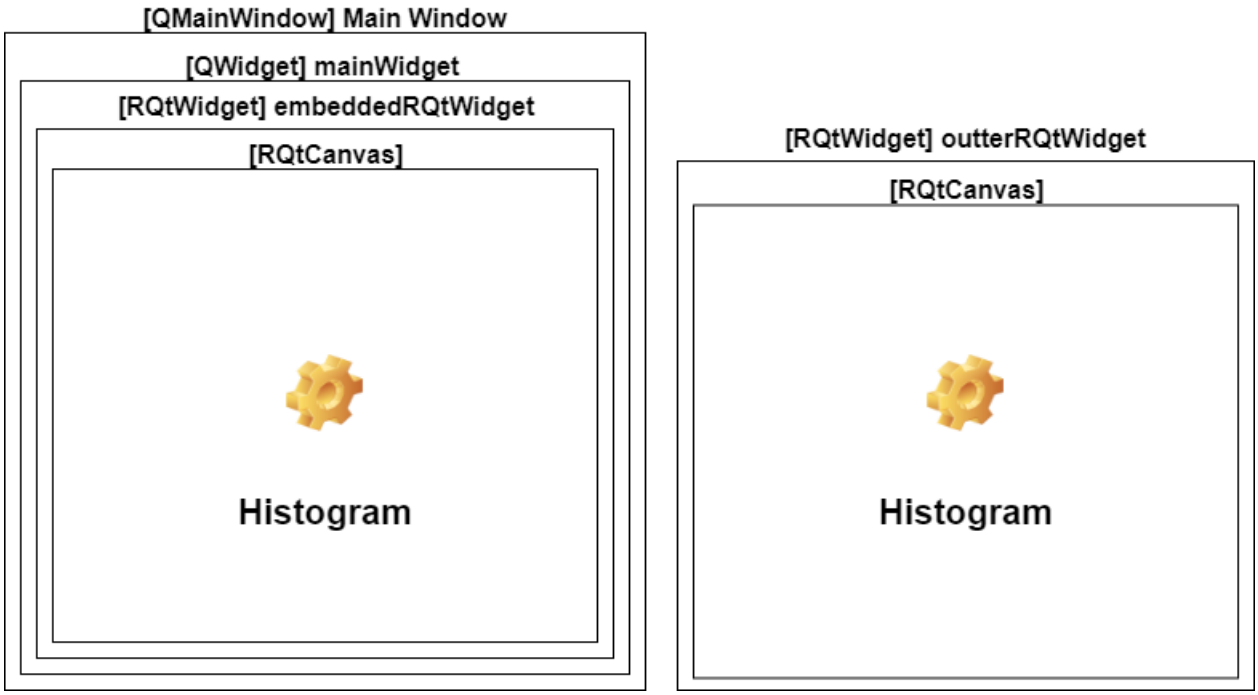

Figure 4.5: *embeddedRQtWidget* and *outterRQtWidget* based on Listing 4.2

This may seem like a useless error-prone bug but in fact, it is a feature the end user may need. It may come in handy in situations in which only a pop-up window is needed for a short period of time as a Loop Up reference.

Chapter 5

RQtWrapper - Example Applications

The objective of this chapter is to explain and show the crucial parts of the actual use of previously introduced *RQtWrapper*. This merge may find it's application in much bigger projects than in the simple examples covered (for highlighting purposes) in this chapter. Nevertheless, the following samples should be enough to grasp the basic idea and required workflow scheme of the *RQtWrapper*.

At this point, as expected we require the back end managers for both *Qt* and *Root* background processing. Hence why we create an instance of an *RQtApplication* to satisfy the requirements of the consisting frameworks.

```
1 int main(int argc, char **argv) {
2     RQtApplication rqtApplication(argc, &argc, argv);
3
4     MainWindow myapp;
5     myapp.show();
6     QObject::connect(qApp, SIGNAL(lastWindowClosed()), qApp, SLOT(quit()));
7
8     return rQtApplication.exec();
9 }
```

Listing 5.1: Root Inner Loop integration

5.1 Example 1 - Single Canvas

The first example is a basic one. However, It still does utilize a lot of *Root*s and *Qt*s inbuilt functionalities and it precisely showcases the workflow with the *RQtWrapper*. As the name implies- the application only has one inbuilt canvas. This example consists of the very simple structure of only *QMainWindow*, a central *QWidget* and an *RQtWidget* as well as it's member - the *RQtCanvas*.

Seeing the already familiar workflow of *Qt* we see an instance of a *QMainWindow* based MainWindow class. To also cover a scenario of closing the last opened *Qt* Window properly we need to connect an according slots and signals. This basically means that when *QMainWindow* based window gets shutdown the RQtApplication shutdowns as well.

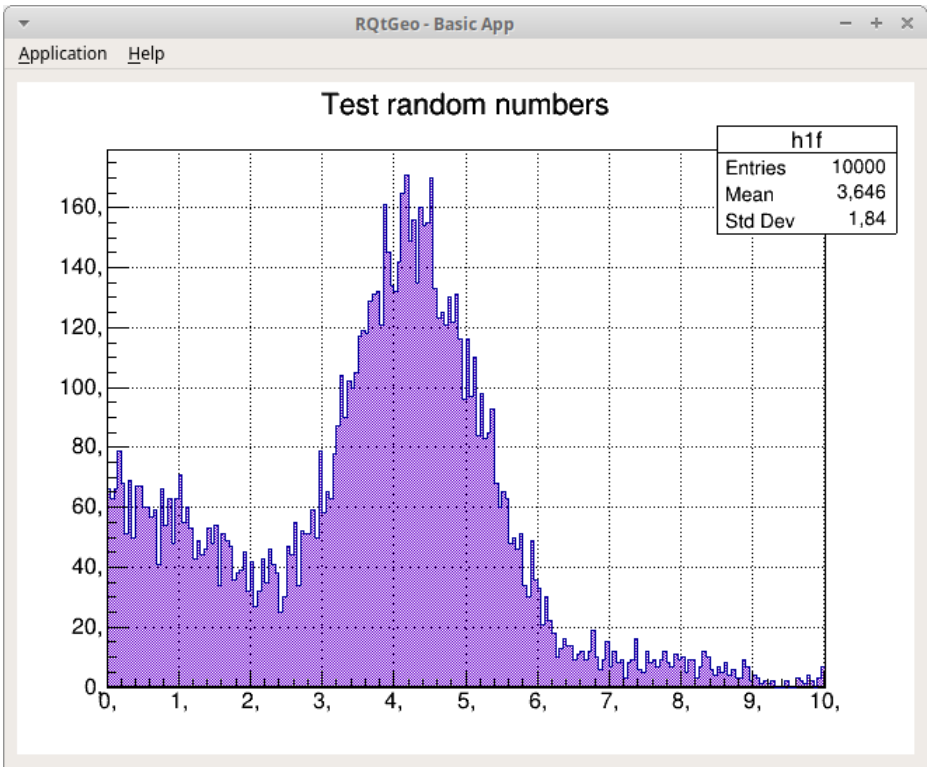

Figure 5.1: Full view of the first example application

Seeing a fully functioning application consisting of both frameworks may be exciting but let us check all of the *Root*'s functionalities such as:

- Editing Histogram's elements and resizing of the window causes the histogram to resize accordingly.
- Creating a new instance of *RQtWidget* in an outter window in RunTime.
- Duplicating the instance of our canvas into a new window.

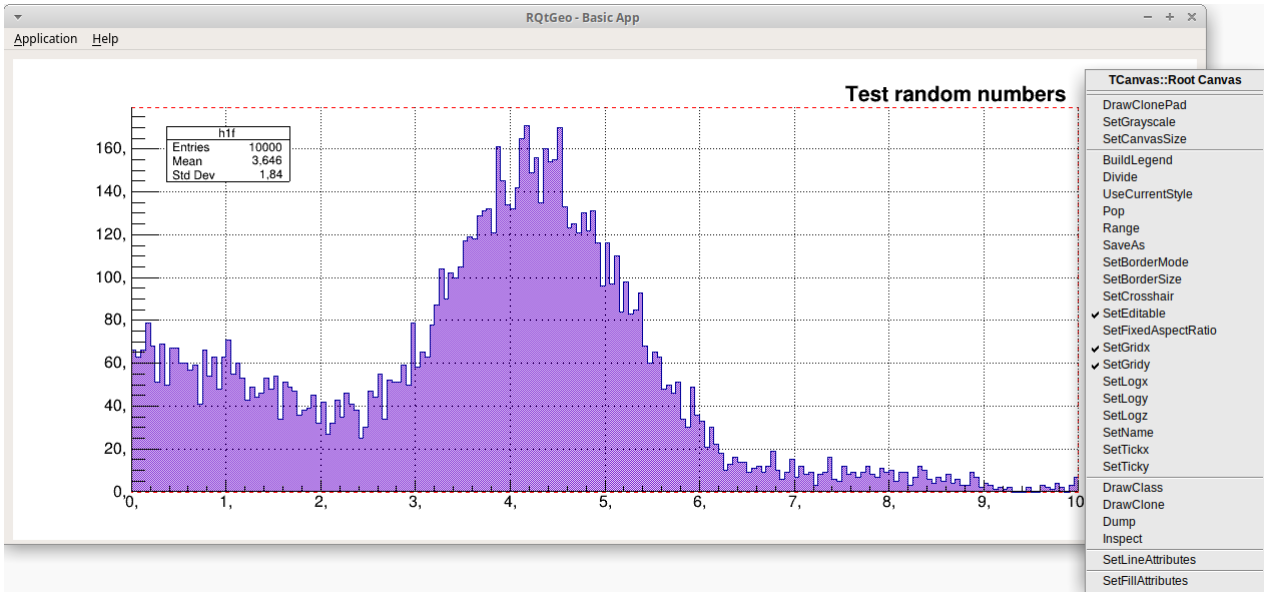

Figure 5.2: A resized window with applied edits and active edit side pane

Seeing the Figure 5.2 it is immediately noticeable what has been changed and what *RQtWrapper* has to offer with its integration. First of all, we resized the main window as well as the inner Histogram which is done seamlessly. Secondly, we have altered the size and position of the plot's legend and changed its title style. Last but not least, by right-clicking on the Canvas activated *Root*s tool pane with a variety of different options to choose from.

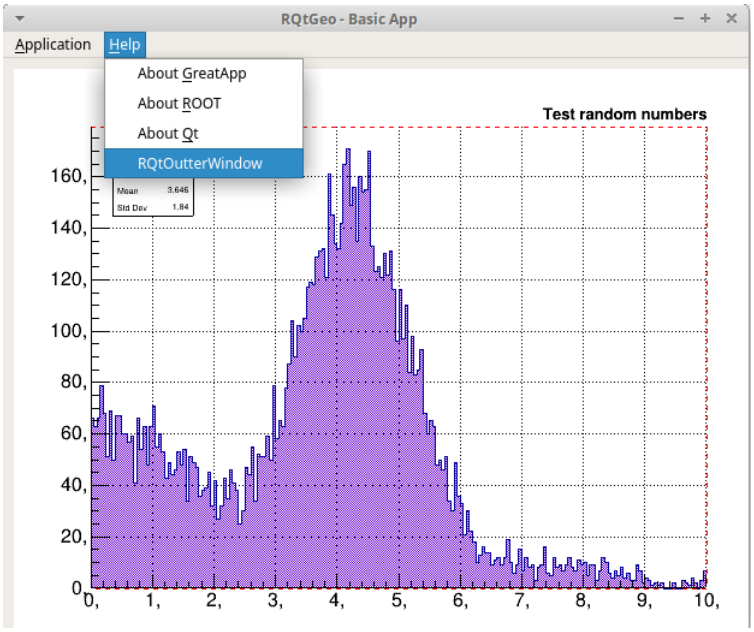

Figure 5.3: Button creating a new outer window with an independent Canvas

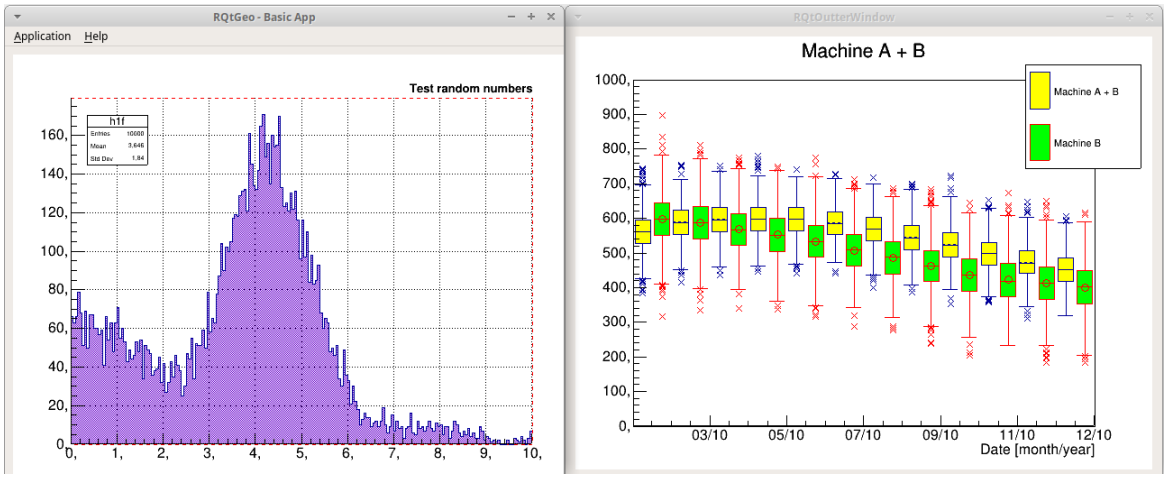

Figure 5.4: Extra window generated in RunTime

5.2 Example 2 - Dual Canvas

The principle of this Example is to showcase that all of the features still work as expected with more than just one canvas within the Hierarchy of the application.

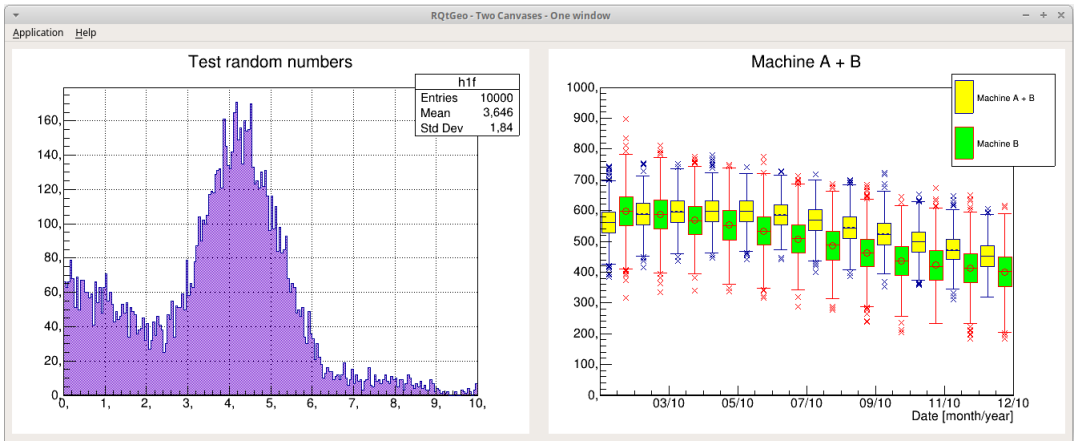

Figure 5.5: Example 2

Seeing the figure 5.7 it is clear that no matter how many *RQtWidget*s we embedd into the application it will still work as expected.

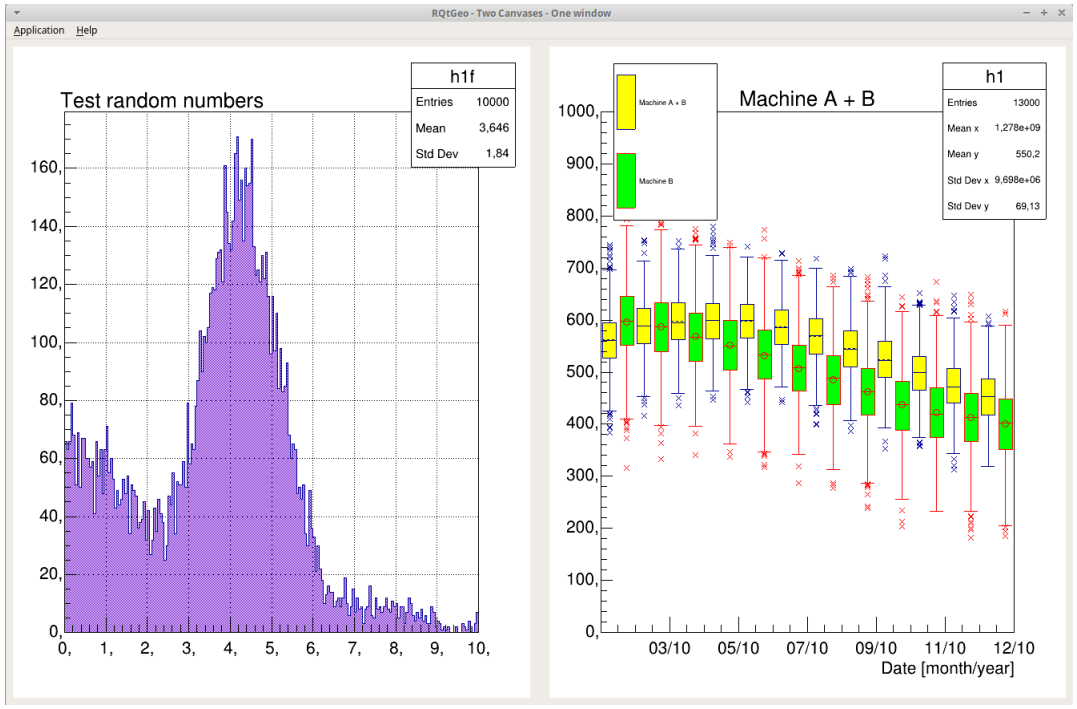

Figure 5.6: Example 2

5.3 Example 3 - Canvases within Tabs

In the case of this example the purpose is to highlight that three different canvases run in parallel in different areas which are switchable when tabs get reselected. This example has an entity of *QTabWidget* implemented. It allows for changing views of a window in Run Time. The first tab contains a simple plot, the second tab contains a candle plot and the third tab contains a button which generates an outer canvas window upon press.

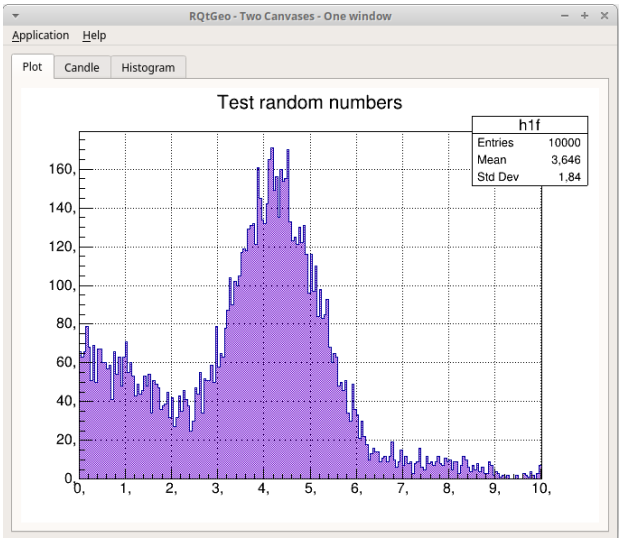

Figure 5.7: Example 3 - First Tab

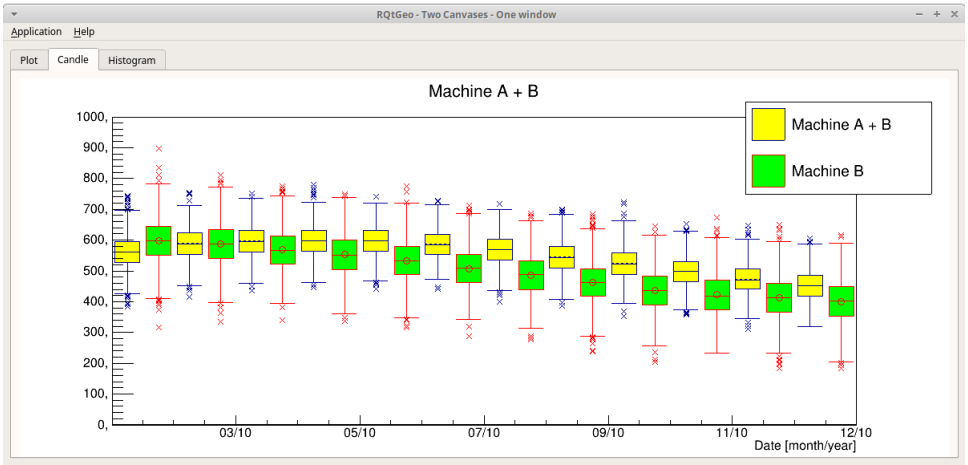

Figure 5.8: Example 3 - Second Tab

In the previous two Figures, the generic features or editing and resizing were showcased. In the following one, another way of allowing end-users to generate an outer window is highlighted. Having seen this in all three examples thus far it is clear that this feature is self-reliant and does not depend upon any application related structure.

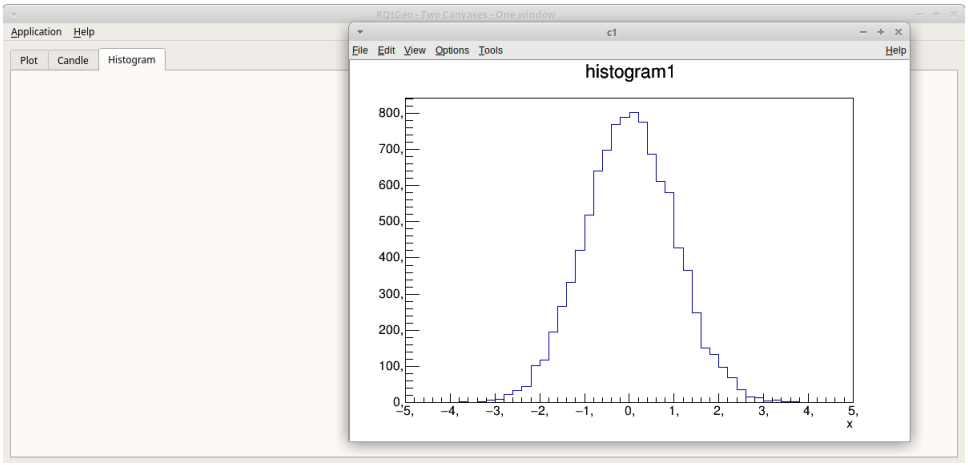

Figure 5.9: Example 3 - Third tap with outer Window

5.4 Example 4 - Multiple Canvases

The fourth and last example is the most advanced one. It touches the aspect of inserting multiple canvases into multiple *Qwidget*s and *QLayout*s. The structure may be a bit complicated therefore it may be easier to understand it through a graph:

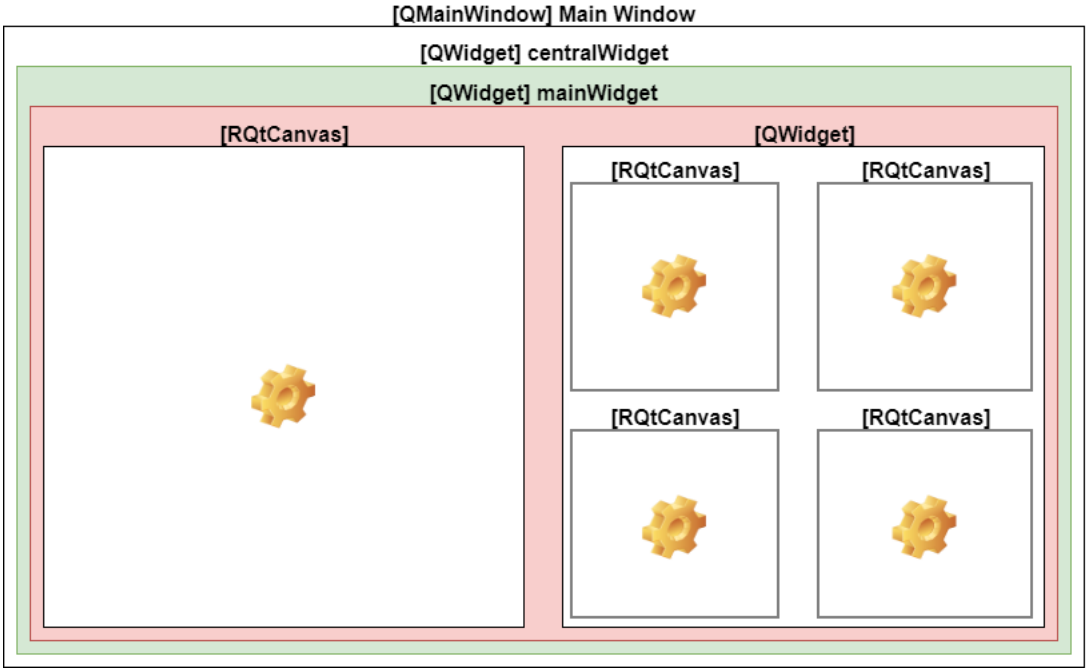

Figure 5.10: Example 4 - widgets layout

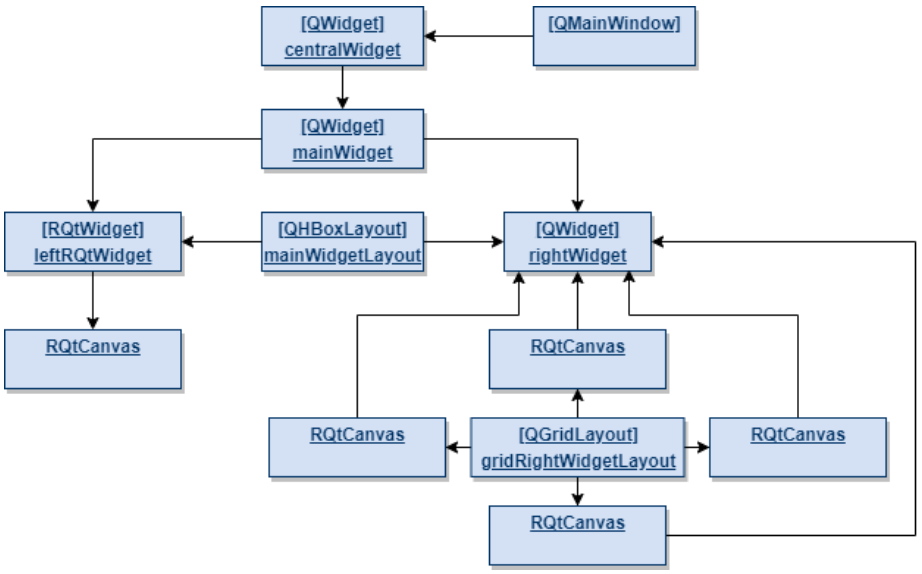

Figure 5.11: Example 4 - widgets layout - second view

Not only this example may seem very similar to the previous ones but also the level of complexity may seem no different. It may be true but this time there is a very crucial part to focus on. This example relies heavily on *Qlayout*s which hold the whole widget layout together. Firstly we used a Horizontal Layout to set up *leftRQtWidget* and *rightWidget* side by side. This allowed for placing one canvas on the left half of the window, and the remaining four in a grid pattern on the right side. On top of that, some of the crucial *QWidget*'s are coloured to make the structure clear. The layout of a grid in the right-hand side of the window is provided by the *QGridLayout* which is a child of *rightWidget*. What makes this important is the fact that if it was not for these layouts this arrangement would not be possible. Another feature of *QLayout*s is the fact that they automatically set their contents - the canvases - accordingly to be fitted with the container of higher class(*rightWidget)*.

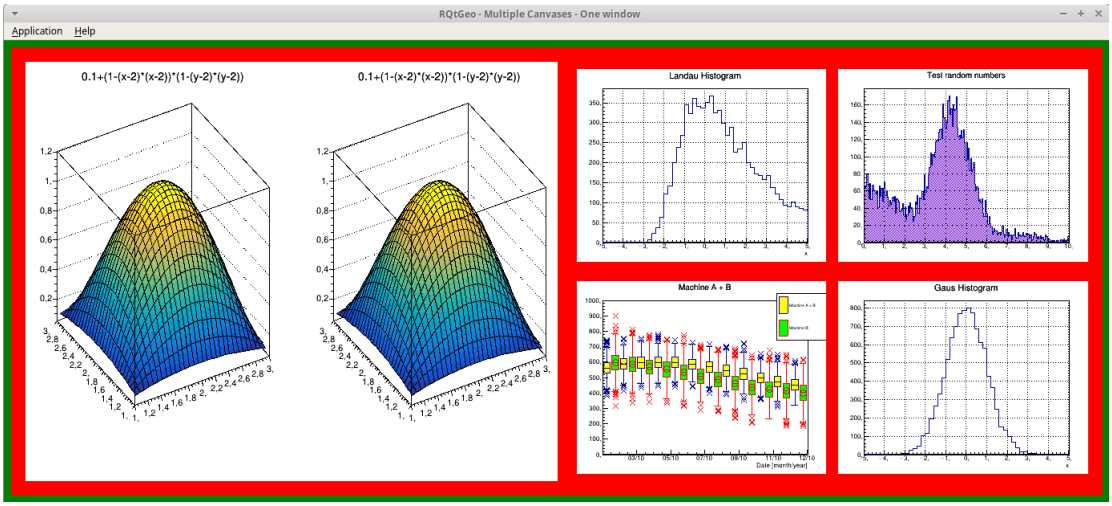

Figure 5.12: Example 4 - widgets layout - main

Manipulation of every individual canvas is also possible, as well as all of the previously mentioned features. These very simple examples, yet advanced enough for showcasing purposes were the essence of the thesis.

Chapter 6

Conclusion And Prospects

The *RQtWrapper* project has begun as an idea of allowing people to reuse their existing code and publish a tool which would let other developers for seamless integration of various code puzzles together. The Wrapper which got created is a really powerful tool if used in a proper way. Having said that we always have to remember to keep attention to the very small detail and know the limitations of the mechanism. Speaking of the limitations, there are some. In specific terms: the application will only work with *TCanvas* instances. Any application which utilizes a different base drawing class (*TGFrame* for example) will not work as this approach is not suitable to wrap over an entire *Root* GUI application. This is a serious limitation but could be thought of as a constraint. Wrapping over an entire application is surely easy and simple but it takes away the smart and clean look which we could, in the end, achieve by sub-merging *Root* within *Qt*. At various stages of development, there were attempts of implementing a level of rigidity which would enforce a specific architecture of any future *RQtWrapper* based application. This idea has been dropped due to potential future limitations for end-users at the level of implementation. This could be something which could be added into the project in the form of a non-mandatory addition.

This work is at this stage fully functional however further development could be conducted.

Bibliography

[1] *Root Main Page*. [Online]. Available: https://root.cern.ch/

[2] *Qt Main Page*. [Online]. Available: https://www.qt.io/

[3] A. *bellenot*, *Code on which RQtWrapper is based*, 2019. [Online]. Available: https://root-forum.cern.ch/t/root-canvas-in-qt5-revisited/34485/9

[4] *Qt Multiple Platforms showcase*, 2012. [Online]. Available: https://www.youtube.com/watch?v=6ubOItPgQzc

[5] *QT example application - notepad*. [Online]. Available: https://doc.qt.io/qt-5/qtwidgets-mainwindows-application-example.html

[6] *Qt Main Page*. [Online]. Available: https://home.cern/

[7] *Definition of what Root is*. [Online]. Available: https://root.cern.ch/

[8] *A visualization of the Galaxy in Root*. [Online]. Available: https://root.cern.ch/galaxy-view-0

[9] *A visualization of Minuit Fit*. [Online]. Available: https://root.cern.ch/minuit-fit-result-graph2derrors-points

[10] *An example of HEP in Root application*. [Online]. Available: https://home.cern/

[11] *Windows File Dialog window*. [Online]. Available: https://blogs.msdn.microsoft.com/wpfsdk/2010/03/26/openfiledialog-sample/

[12] *GUI application event loop graphics*. [Online]. Available: http://lifexasy.com/nodejs˙event˙loop

[13] *FIFO stack image*. [Online]. Available: https://dev.to/rinsama77/data-structure-stack-and-queue-4ecd

[14] *GUI window Hierarchy.* [Online]. Available: https://www.researchgate.net/figure/3-example-of-a-Java-GUI-and-its-hierarchy-of-containers˙fig18˙265248151

[15] *Qt Basics Tutorial.* [Online]. Available: https://wiki.qt.io/Qt˙for˙Beginners

[16] *QMainWindow Basics Tutorial.* [Online]. Available: https://doc.qt.io/qt-5/qmainwindow.html#details

[17] *Quote from Qt Documentation.* [Online]. Available: https://doc.qt.io/qt-5/qlayout.html#details

[18] *Calendar app built with Root.* [Online]. Available: https://root.cern.ch/root/html608/calendar˙8C.html

[19] *TApplication Definition and Graphs.* [Online]. Available: https://root.cern.ch/doc/v610/classTApplication.html

[20] *TPad Definition and Graphs.* [Online]. Available: https://root.cern.ch/doc/master/classTPad.html

[21] *An example of how Root could be used within QT's GUI. This example does not reflect such combination. It is used only for a concept example.* [Online]. Available: https://doc-snapshots.qt.io/qt5-5.9/qtdatavisualization-surface-example.html

[22] *First visualization of root application for spirometry.* [Online]. Available: https://www.researchgate.net/figure/Volume-Time-curve-of-spirometer-and-model-estimates-reference-points-are-for-tidal˙fig3˙313419071

[23] *Second visualization of root application for spirometry.* [Online]. Available: https://www.sciencedirect.com/topics/medicine-and-dentistry/spirometry

[24] *TCanvas Definition and Graphs.* [Online]. Available: https://root.cern.ch/doc/master/classTCanvas.html#a33bbb4c95ac1a51b433c8c42274e65cd