Easy GUI
- User Manual -

Bogdan-Nicolae David
david.bnicolae@gmail.com

October 11, 2022

Contents

1 Introduction

This document will provide you a broad overview of what this extension has to offer and how it simplifies UI creation, as well as detailed guides of how to use each major component of the extension. Do note that this document **does not** replace the documentation. If you want specific details on how to use each function, please check the documentation that comes with the release package.

1.1 Application Layout

Similar to website, this extension divides the user interface into two major ares:

- The visible area
- The backstage (invisible area)

The **visible** area refers to what is visible on the screen. When working on this side you would typically use Menus or other components such as: Labels, Buttons, and so on.

The **backstage** is defined by what controls the components visible to the user. When working on this side, you would typically use predefined methods (such as On Click) or Routines.

1.2 File structure

The file structure is always up to you, however we recommend a standardized structure to ease the code production and understanding:

- Main.cpp (The main source file)
- Menus.hpp (Header which includes all the application menus)
- Routines.hpp (Header which includes all the application routines)
- Menus (Folder containing each menu)
- Routines (Folder containing each routine)

The application comes with a demo project structured according to this layout. If you look through the source files provided with the demo, you will find that the code is easy to understand and maintain. For instance, whenever you would add a new Menu, you would create the appropriate source file in the **Menus** folder and include it via the appropriate header file.

2 Application control

In this section we will dive into the invisible side of the application, and give you an overview about Routines, what they do and how to use them.

2.1 Routines

Routines are the primary way of controlling the application flow. A routine is made out of two functions:

- A **trigger** which tells the routine when to fire
- A **action** which tells the routine what to do after it's fired.

For instance, a trigger might tell a routine to fire only when the mouse is pressed, and an action could tell it to create an animation around the cursor. It is important to know that whenever an event occurs, **all** routines will fire if they are triggered by the event.

For example, when a mouse is pressed, a routine could be fired because the mouse was pressed, and another one could be fired because a general mouse-related event occurred.

2.1.1 Triggers

Triggers are what tell a routine when to fire. They are special functions which should always take this form:

bool [Trigger name] (const sf::Event event)

In general, it is considered good practice that each routine has its own trigger and action. This is because if two routines are triggered by the same trigger, they can be merged into one (thus removing the redundancy).

When creating a routine **you** are responsible for creating its own trigger and action and assigning them to the routine. However, the extension comes with two predefined routines: **OnClick** and **OnHover**.

2.1.2 OnClick() method

The **OnClick()** method will fire whenever the mouse is **pressed**. What it does is it loops through all the components visible on the screen and if they are configured with an **onClick** action, they will trigger that action if the mouse if and only if over the said component.

2.1.3 OnHover() method

The **OnHover()** method will fire whenever the mouse is **moved**. What it does is it loops through all the components visible on the screen and if they are configured with an **onHover** action, they will trigger that action regardless of the position of the mouse.

Because of this, it is wise to configure the function with two different actions: one to be executed when the mouse is **over the component**, and another to be executed when the mouse is **elsewhere**.

2.2 Exceptions

The extension also provides a set of exceptions which are thrown whenever something goes wrong. It is important to write a proper exception handler to avoid the application crashing whenever a font is missing or something else malfunctions. There are two main categories of exceptions:

- Application exceptions
- Asset exceptions

Application exceptions occur at application level and typically involve general components such as Menus or even the application itself. Usually an application exception is thrown whenever the expected flow is disturbed (e.g. the application cannot start)

Asset exceptions occur at lower levels, and involve a single component within a menu (such as a Label or a Button). Usually these exceptions do not have critical impact and whenever one is thrown the worst thing is that the affected component cannot be rendered.

For more information about exceptions please checkout the Doxygen documentation.

3 Menus

In this section we will present an overview about the visible side of the application: Menus and how to manage them.

3.1 Creating menus

After instantiating the application, you can create a menu via the designated function: **addMenu**. The function will create a new menu and immediately return a pointer to the menu, so that you can use it to add components to the menu.

If you ever want to change between visible menus, you can use the **setActiveMenu** function. Furthermore, all menus can be retrieved with the **getMenu** function.

3.2 Adding components

After you have created the menu, you will want to add components to it. Otherwise it would be empty. Each component requires its own parameteres, and you will have to consult the Documentation for specific components.

However, below you have an example of how to add a label to a menu. You can also find more examples in the demo project that comes with the binary.