*PinGUI v0.1 User Manual*

Copyright © 2017 Ľubomír Barantal

**1 About Library**

PinGUI is a free and open source library that lets you to create different gui elements, windows and menus for your games.

PinGUI is written in C++ programming language using libraries SDL2 (input, window management, possible sound (not yet), image/TTF loading) and OpenGL (rendering purposes).

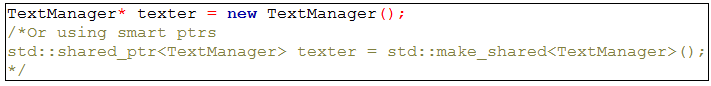
As there is no DLL you have to compile the library.

**2 Architecture of library**

While developing the library, the main idea was to separate everything as much as possible to give the developer a big overview of how and where is the code being handled. The main two core systems are called **TextManager** and **GUIManager**. TextManager can be used without the GUIManager, so it means that if you need only the text support functions, you have to include the TextManager to your project. However, GUIManager cannot exist without the TextManager. The library is manipulated via a core **singleton** called **PINGUI**.

**3 TextManager**

TextManager uses SDL\_ttf extension for loading different fonts and generating their textures which are then converted to OpenGL textures, so your OpenGL context will use them. This class is not a singleton, so an instance is needed to be created.



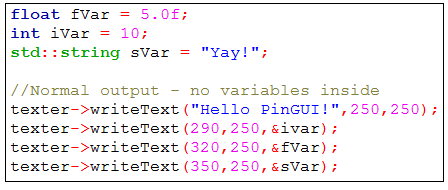
**3.1 Writing a text with TextManager**

For getting a text output, you have to use a prepared method called writeText. It is an overloaded function that can be used for 4 different cases :

* Static Text:

1. Normal text (just a plain string, not going to be changed frequently and if yes, you have to do it yourself).

* Dynamic text (it means that you put a variable inside it and it will be checked/changed frequently if the variable changes it value):
  1. Integer variable.
  2. Float variable.
  3. String variable.

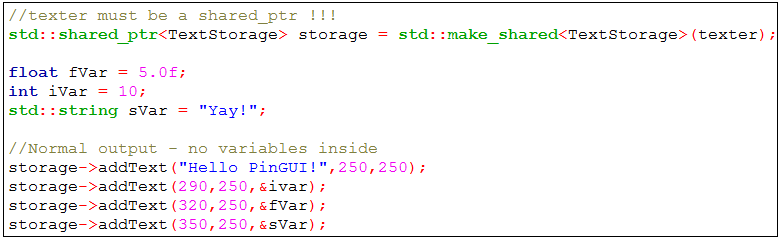


For now a pixel positions are used for placing different elements in this library but it may be changed during the development, so be sure to check the descriptions of every library version).

By default they return a created **shared\_ptr** if you needed to save it somewhere. (but it is saved in the TextManager´s memory anyway!)

**3.2 TextStorages**

TextStorages are feature of this library that provides you with creating a sub-storages of text input when needed. Usually GUI elements communicate with the TextManager via these objects. The main idea was to create an object, that would be able to create a multiple text input (like TextManager) and destroy it/hide it with a single function call. The only difference is that TextStorages rely on the TextManager and cannot be used alone. A way of use is almost same:



Then, by a single function call, all Text elements can be hidden via a TextStorage:

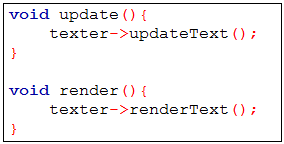
1.png

After a deletion, all Text elements included in the TextStorage are deleted.

You can also access the TextManager binded to TextStorage via the proper getter.

**3.3 Manipulating TextManager**

To get a proper output from your TextManagers you have to update and render the text in your functions positioned in game loop. It looks like this:

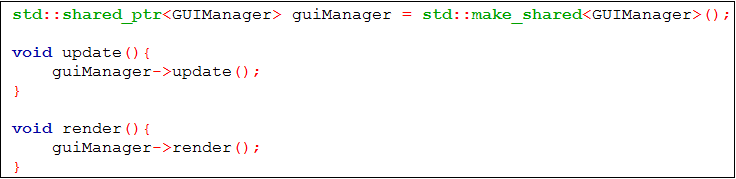


*Note: If you have multiple TextManagers you have to update 1 by 1(or use core PINGUI singleton methods – we will discuss it at the end).*

Then there are additional methods for changing your font (TTF), changing color of font or size.

**4 GUIManager**

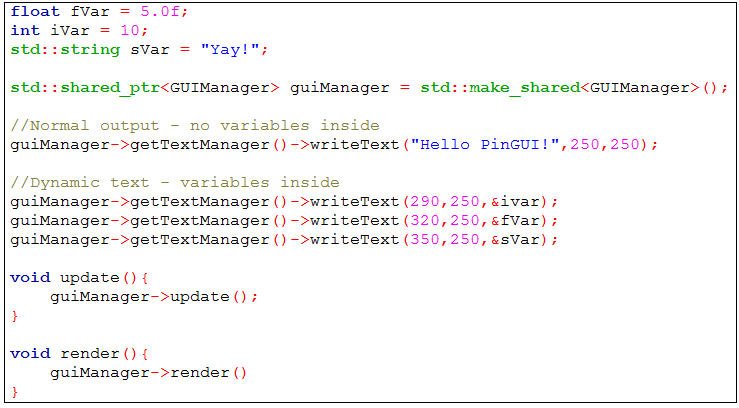
**GUIManager** is another core system used for creating and manipulating different GUI Elements. Creation and manipulation looks same like it was done for TextManager:



*Note: update() function has 1 implicit parameter called* ***allowCollision****. By default tis* ***true*** *but when its* ***false*** *the GUIManager will not check collisions.*

**Another note (important)**

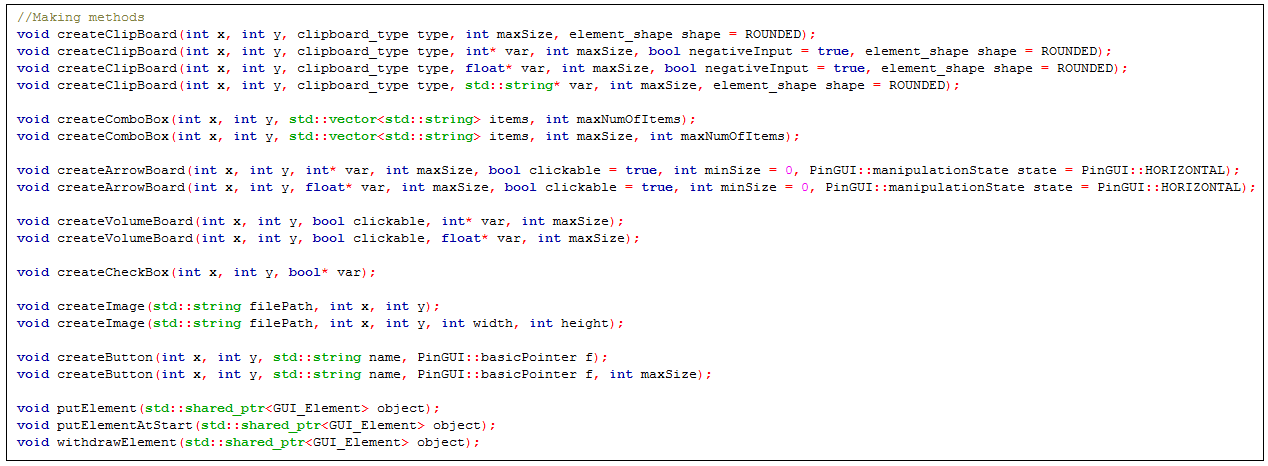
By default every GUIManager has its own TextManager that tisd for every GUI Element created with choosed GUIManager. If you do not want this feature you can pass a created one. However, if you do not want to create a TextManager separately (not recommended – more work to do to maintain it) tis better to use a TextManager binded to current GUIManager so the text elements will be connected if you decide to destroy a group of gui elements (or hide windows, etc.). The manipulation would look like this:



If you understand it, now you can access the TextManager and don´t need to update or render it separately, GUIManager will do it for you! The only change is that you need to acces these methods by several getters.

**4.1 GUI Elements from GUIManager**

GUIManager tisd for creating a simple gui elements via proper functions:

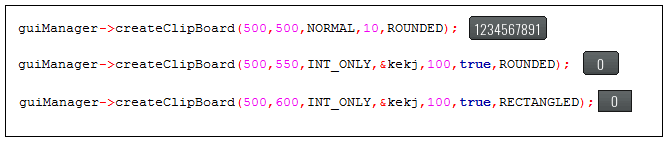


**4.1.1 Clipboard**

You have to know the following: Clipboard=textbox. tis used to load input from user, strings, numbers and whatever you want to know. Usual parameters are:

* X position
* Y position
* Type – special enum for clipboards
  1. NORMAL – text input, accepts everythings
  2. INT\_ONLY – only integers
  3. INT\_FLOAT – integers+floats(point numbers)
  4. UNCLICKABLE – cannot access via click
* maxSize – this variable has different behavior based on its type – when using NORMAL it specifies the number of letters (chars) that can fit inside the textbox while in other case it specifies the maximum value (the maximum number of letters will be then calculated automatically)
* element\_shape – defines the shape of textbox(RECTANGLED or ROUNDED)

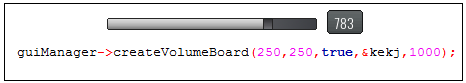
In numeric textboxes you have to (or not – its implicitly set to true) define it it can accept also negative values with bool negativeIntput.



**4.1.2 Volumeboard (slider)**

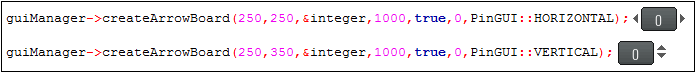
Sliders are used to store numerical variables, parameters (except positions)

* clickable – true (you can click on the textbox and change it with keyboard typing) or false (you cannot click the textbox)
* pointer to a variable (int or float)
* maximum size for variable



**4.1.3 ArrowBoards**

Used for numerical variables, min value in constructor defines the minimal value, and enum for the style of arrows(HORIZONTAL,VERTICAL – do not forget the PinGUI:: prefix) – ratio can be also changed (by default ratio is 1 – it means it will inc/dec variables by this ratio)



**4.1.4 Crossbox (checkbox)**

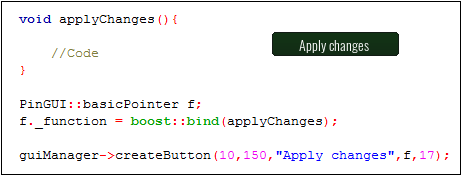
Used for checking input for booleans.

1.png

**4.1.5 Buttons**

Buttons are used for executing a functions after click. Parameters:

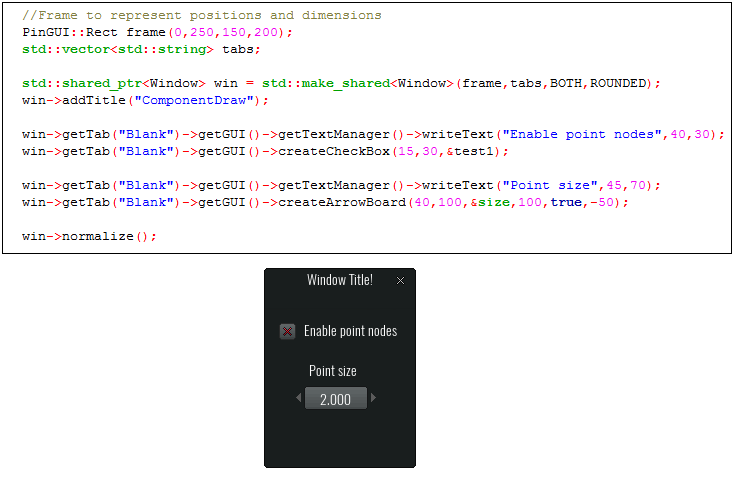
* positions
* Name of button
* Function to be executed after click (created with help of boost::bind or std::bind [boost recommended])
* Last integer is not required ( tis explicitly used for setting the length of the name, to create bigger buttons)



Special case of Button is **WindowButton**. It has a parameter as a pointer (shared\_ptr ! ) to a Window element.

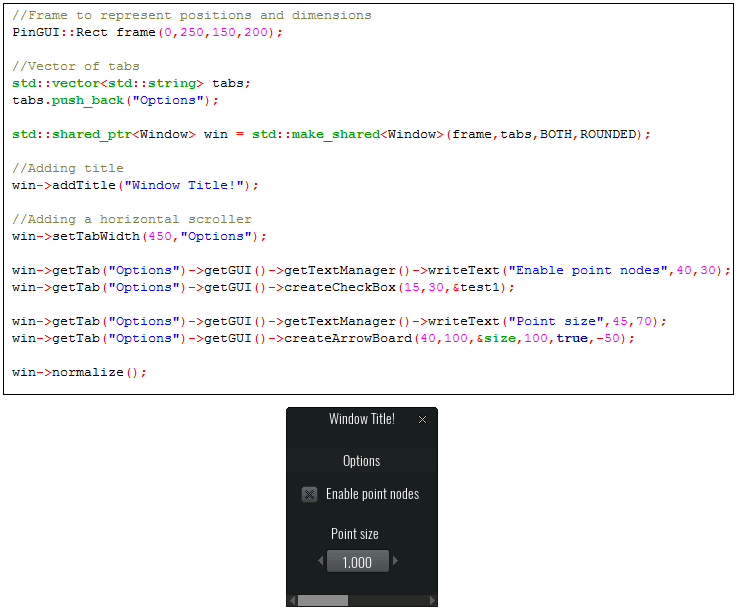
**4.1.6 Windows**

Windows are pretty specific, so they are not created via GUIManager and need to be created explicitly. Creation looks like this:



PinGUI::Rect represents the frame used to create. It has 4 parameters: x and y position, width and height. Then there is a vector of tabs – strings are used to represent each. If you create a window with 0 tabs a default tab – „Blank“ – is created and used for maintaining the content of the window. You access gui and text functions via proper getters.

Now different approach:



The difference is that you created a tab. Tabs define different window contents, so the user can choose whatever he/she wants. All you have to do is to put a tab name into your tabName vector and in getter (win->getTab() ) call proper tab. If you call non-existing tab PinGUI will throw error and close program.

Also you can see we defined explicitly width of the created tab. Width is now bigger than the width of the whole window frame so library will make a scroller for you. Same thing can be done on height.

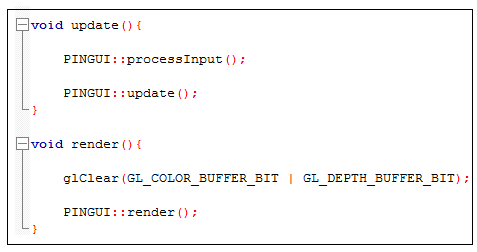
Very important is the last function. **NORMALIZE.** It means that for every GUI element inside the window tab will have [0,0] as the start point of the window tab. It makes things easier.

**5 PINGUI singleton**

You can define everything by yourself explicitly, but the best thing is to use a main singleton – PINGUI. It takes care of created windows, filter them to update only the visible windows and store them. It also updates them and renders. You can see its use in demo main.cpp



*Note: Use the singleton also in your update methods and render, so it will be easier to update all the stuff.*



**6 Different graphics**

All of the graphics is loaded via SheetManager. The main idea is that it uses a parts of every element and then when you enter their dimensions it just uses them to create a texture of your element. You can redesign these graphics, but if you have its positions (x,y) or dimensions (width,height) you have to change also the macros in SheetManager.hpp.

**7 Future of library**

The library is still in early development stage so it means the way of how it work can be changed. If you have any questions you can ask me on [l.pinsius@gmail.com](mailto:l.pinsius@gmail.com) or ask on github/warn about existing errors.