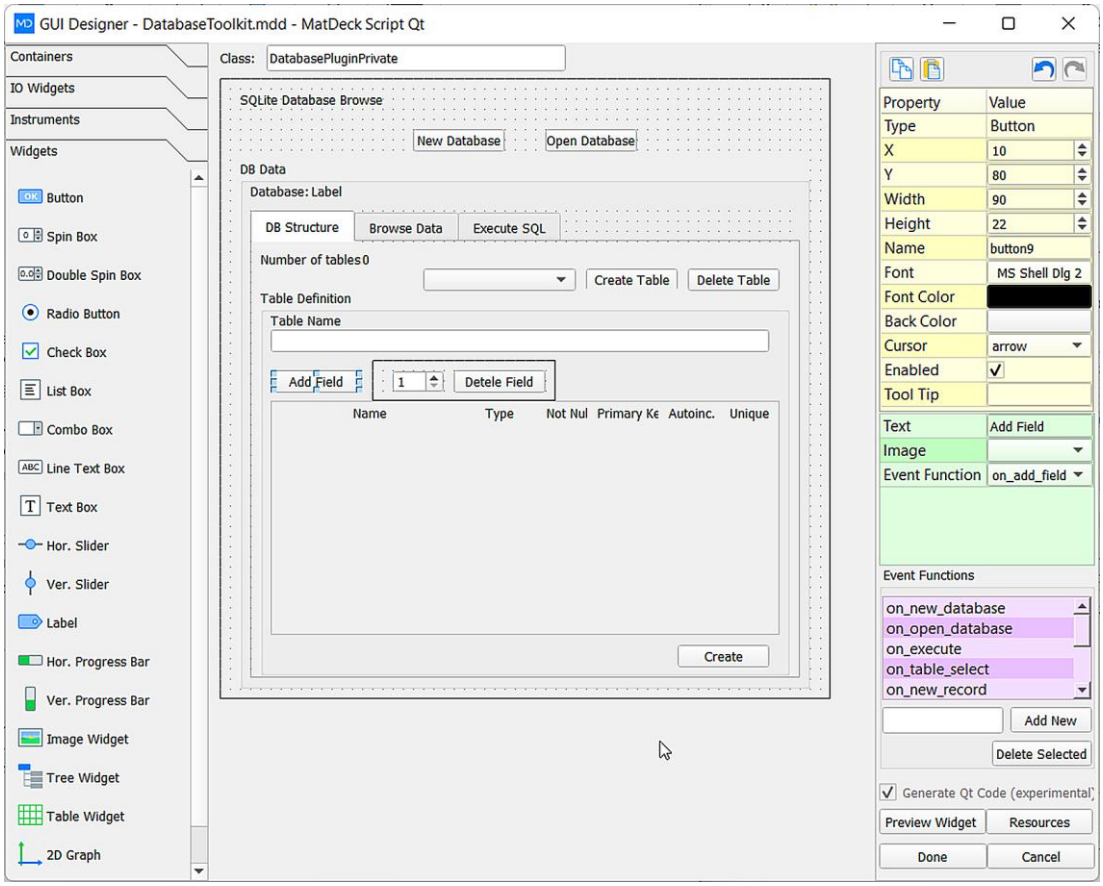
MatDeck

Rust (egui)
GUI Designer

Drag-and-drop GUI building for MatDeck · User manual



Contents

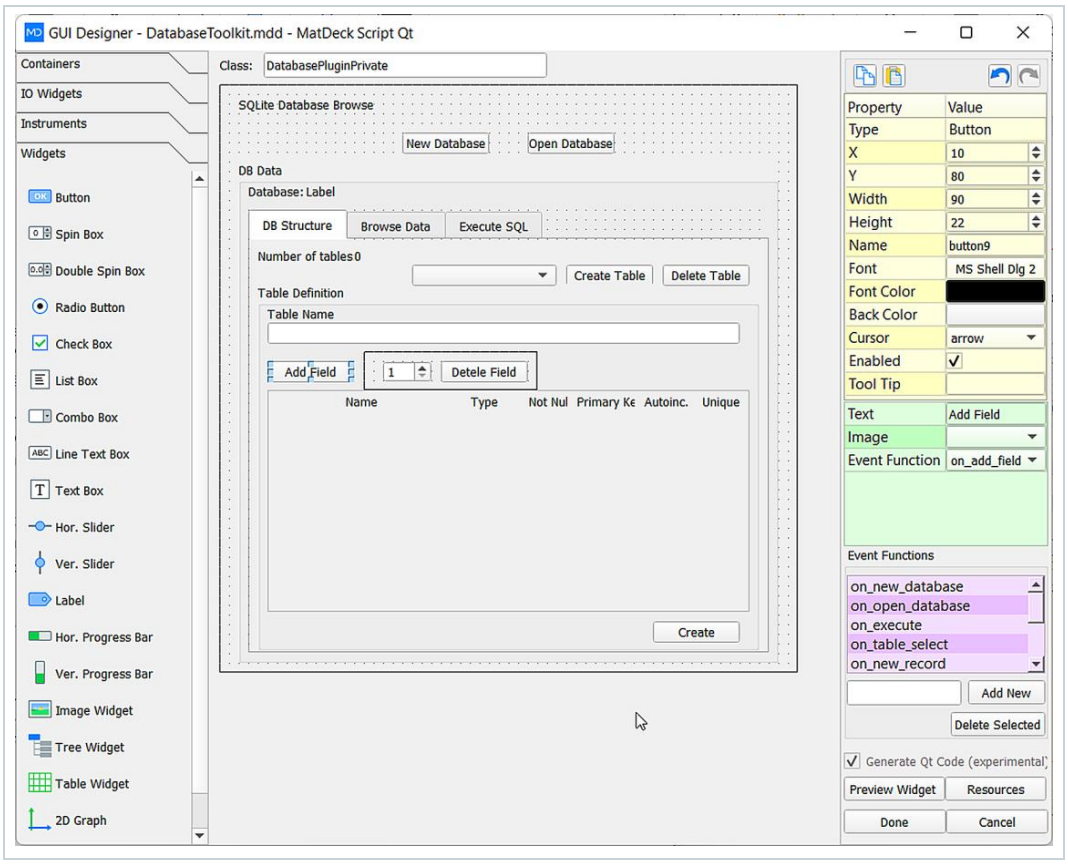
Introduction

The Unlimited Rust (egui) GUI Designer is available in MD Python Designer, Engineering Designer, Visionary Deck, MatDeck Student, MatDeck Academic, MatDeck Home and MatDeck. Its limited form is available in Lite MD Python Designer.

MatDeck products now offer nine GUI Designers — MatDeck Script (Qt runtime), MD Python, PySide2, Tkinter, Custom Tkinter, Kivy, Flet, Julia (QML) and Rust (egui). MD Python Designer and higher MD products have unlimited access to every GUI Designer, whereas Lite MD Python Designer is limited to only a few elements and MatDeck Free does not include any MD GUI Designer.

MD GUI Designers use drag-and-drop visual aids to build a fully functioning GUI with a sleek, modern interface; the widgets and components are generated automatically, leaving no boilerplate work to the user. MD Python Designer ships with every language, library, module and extension already installed, so no extra setup is needed and you can start creating straight away.

MD Python Designer also comes with a specialised IDE and GUI Designer for Rust (egui). As mentioned above, all necessary libraries and toolchains come preinstalled.



This is what a MD GUI Designer looks like. As you can see, the form itself contains no code and was created solely by dragging and dropping GUI elements into the work area.

> **Please note** The GUI Designer generates two files. One is the Rust source file; the other is used by the designer to save and remember what your GUI looked like. This second file has the .mpy extension and cannot be opened like a source file.

Available widgets

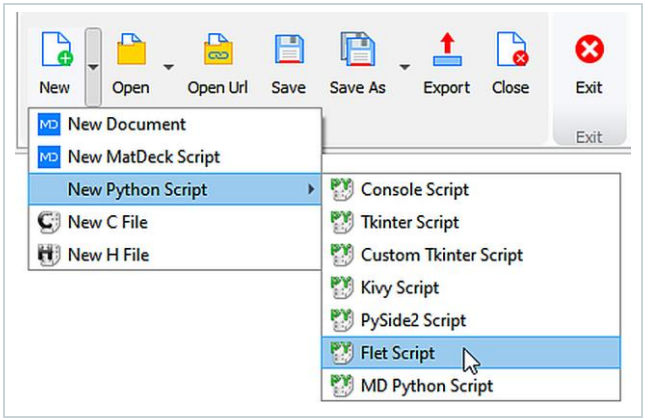
The Widgets tab of the Rust (egui) GUI Designer always provides the core controls, and the Containers, Inputs & data and Graphs groups add the elements listed below.

Category	Widgets available
Core widgets	Button, Label, Radio Button, Check Box, Combo Box, Line Edit, Text Box, Horizontal Slider, Vertical Slider, Horizontal Progress Bar
Containers	Root Widget, Group Box, Tab Widget, Scroll Area, Frame, Menu Bar, Tool Bar, Custom Widget
Inputs & data	Spin Box, Double Spin Box, List Box, Table, Tree Widget, Image, Vertical Progress Bar, Circular Progress, Toggle Switch, Spinner, Divider, Icon Button
Graphs	2D Graph, 3D Graph

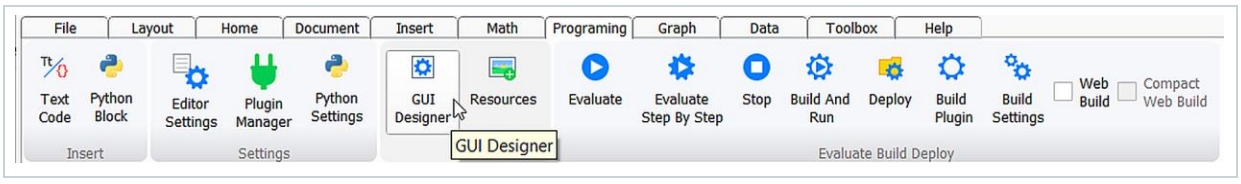
Feature support: the Rust (egui) designer generates layout code. Event-function stubs and the advanced property fields (tooltips, cursor and similar) are not exposed in the designer — behaviour and detailed styling are added directly in the generated Rust code.

Opening a MD GUI Designer

The first thing to do is select the Rust Script. This is done by hovering over the small triangle next to the New icon.

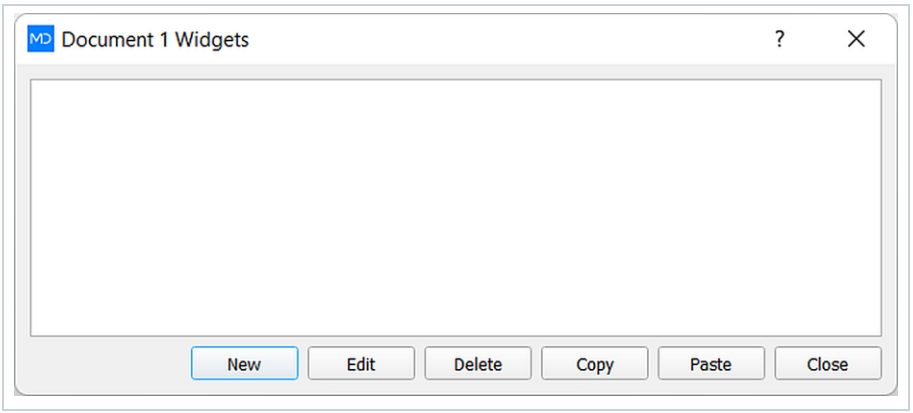


To open the Rust (egui) GUI Designer, go to the Programming tab and press the ‘GUI Designer’ button (Picture 1).



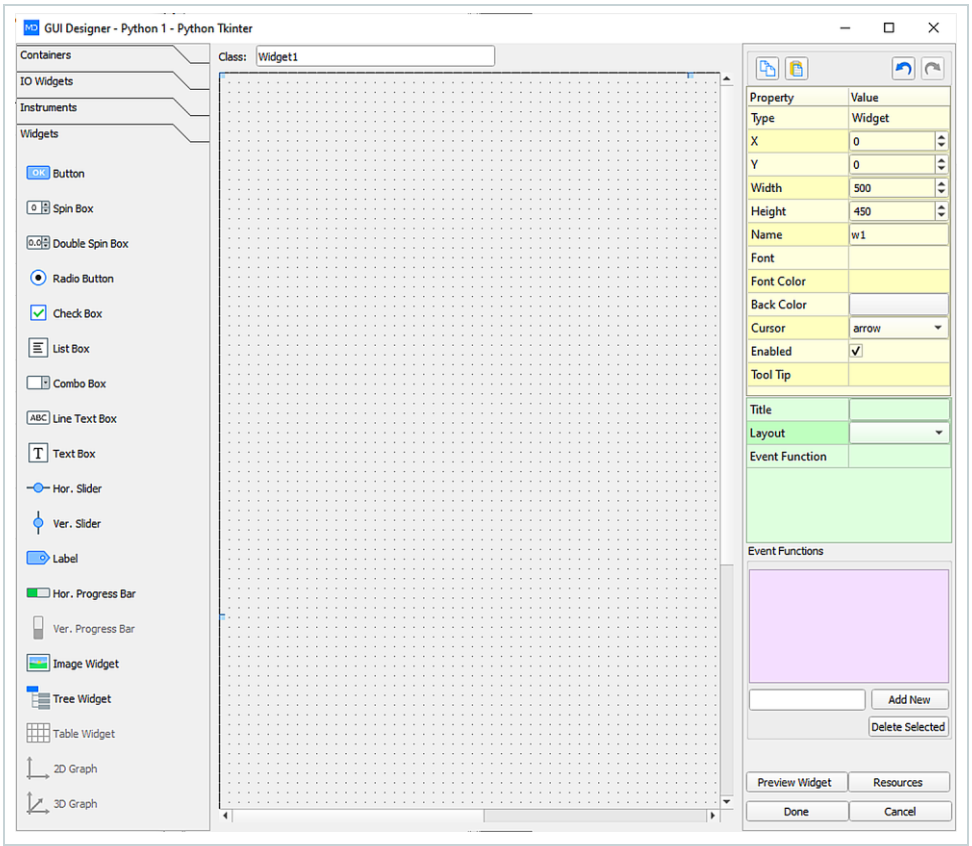
Picture 1: Widgets button

A new window opens containing a list of all widgets created in the current Rust Script (Picture 2). From this window you can create a new widget, or edit or delete an existing one. To open the Rust (egui) GUI Designer, click New; the Edit button only opens the designer when a widget already exists.



Picture 2: Document widgets editor

To create a GUI, press New and an empty Rust (egui) GUI Designer tab opens. When doing so, make sure no existing widget is selected in the window. To edit or delete a pre-existing widget, select it first, then press the appropriate button at the bottom of the tab. The default look of an empty Rust (egui) GUI Designer is shown below.

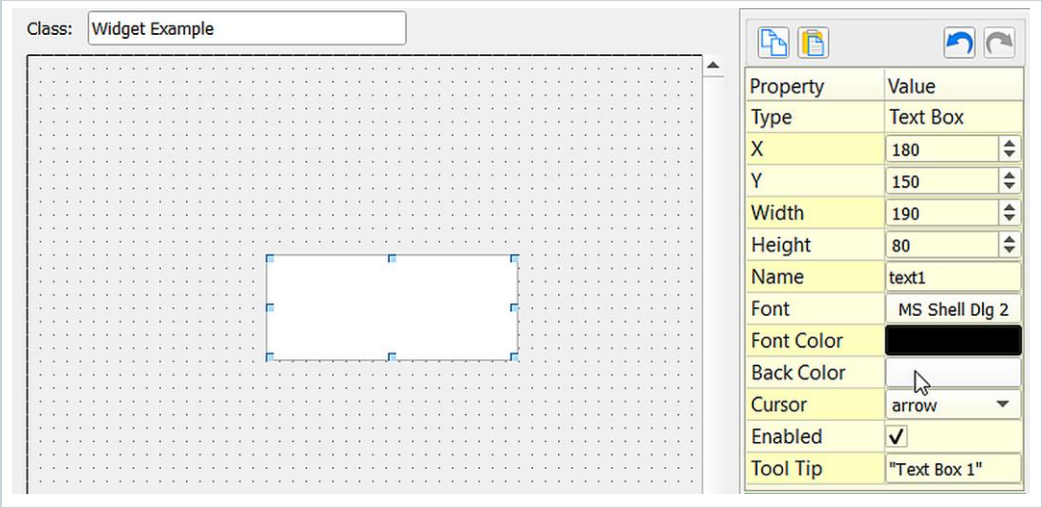


Picture 3: GUI Designer

On the left side of the Rust (egui) GUI Designer are all the GUI elements; the middle is the working area, where widgets and elements are placed. To place an element, click the preferred GUI element and then click where on the working area you want it.

For example, to add a Text Box, click the Text Box button on the left and then click anywhere on the working area.

On the right-hand side, the properties of the selected element are displayed and can be modified (Picture 4). In the example below we have also set the widget name to 'Widget Example' — you do this by typing the preferred name in the field above the main working area.

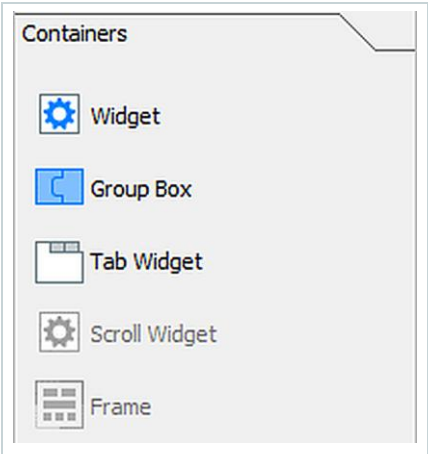


Picture 4: GUI Element Property

Please note that to connect or interact between GUI elements you must add the interaction code yourself by editing the generated Rust code. We focus on saving you time and effort creating the GUI and on simplifying the layout work; what your code ultimately does is left to your imagination.

Containers

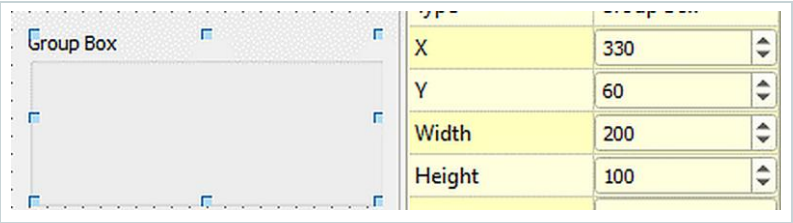
Opening the Containers menu from the top-left of the Rust (egui) GUI Designer reveals a list of container elements. Containers are placed on the main widget area and act as carriers for other GUI elements.



Picture 5: Containers

To insert a container, use the same method as any other GUI element: select the container, then click where on the working area you want to place it.

When a container or element is selected it can be resized by dragging the blue handles around it, or by changing its Width and Height values in the Property table on the right.

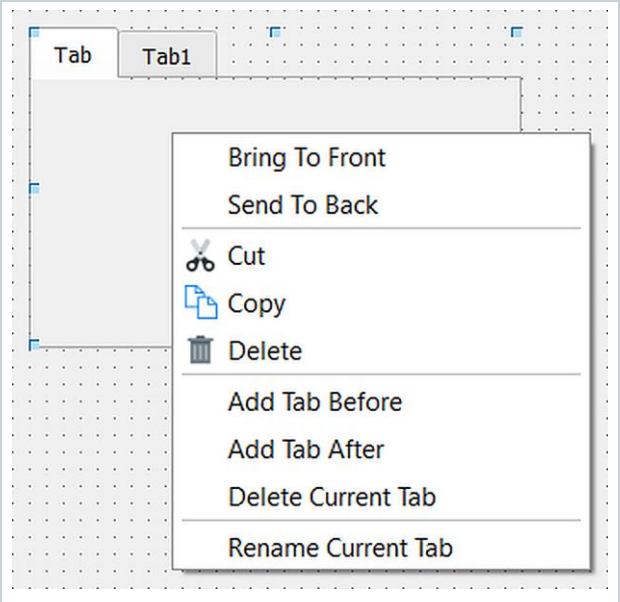


Picture 6: Resizing of object

To preview your widget and check your progress at any point, press the 'Preview Widget' button in the bottom-right corner. It opens a new interactive window where you can test your widget.

Tabs and element options

When a Tab widget is used, the context menu lets you rename the current tab or insert a new one — right-click the widget to open it.



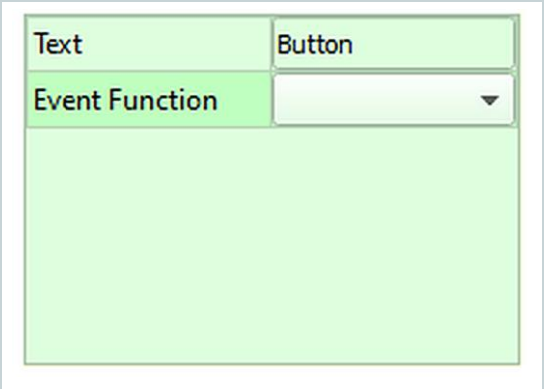
Tab widget context menu

Several elements expose a small editing panel (colour-coded green) for their element-specific options — a button's text, a spin box's range, and so on. These panels are shown in the widget descriptions below.

The Rust (egui) designer focuses on generating the layout: it does not create event-function stubs, so interactions are wired up directly in the generated code (see 'Interacting with generated GUI code').

Widget settings

Button



When a Button is inserted, the green panel above appears on the left. The ‘Text’ field sets the visible label on the button. Click handling is added directly in the generated code.

Check Box

A Check Box lets the user toggle an option on or off. The ‘Text’ field sets its label (default ‘Check Box’) and the ‘Value’ attribute sets whether it starts checked.

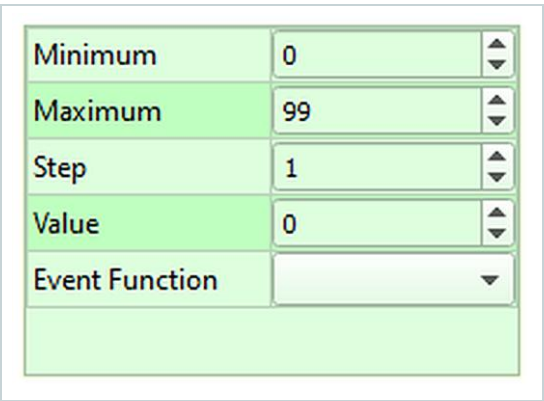
Combo Box

A Combo Box presents a drop-down list of selectable entries. Its rows are populated from code.

Line Edit and Text Box

A Line Edit (Line Text Box) accepts a single line of typed text, while a Text Box accepts multiple lines. Both read and write their contents from code.

Spin Box

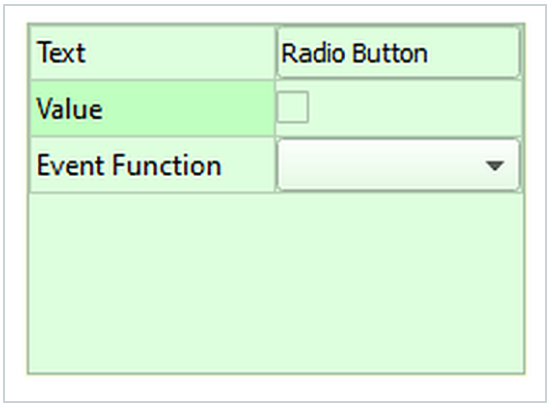


When a Spin Box is inserted, element-specific settings appear. Set the range with the ‘Minimum’ and ‘Maximum’ fields, the increment with ‘Step’, and the starting value with ‘Value’.

Double Spin Box

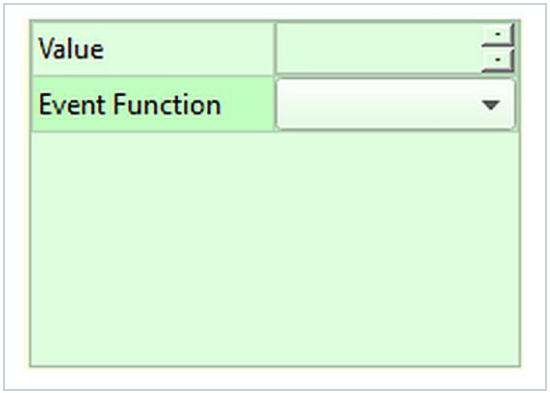
A Double Spin Box behaves like a Spin Box but accepts fractional (floating-point) values. Use ‘Minimum’, ‘Maximum’, ‘Step’ and ‘Value’ to configure its range and precision.

Radio Button



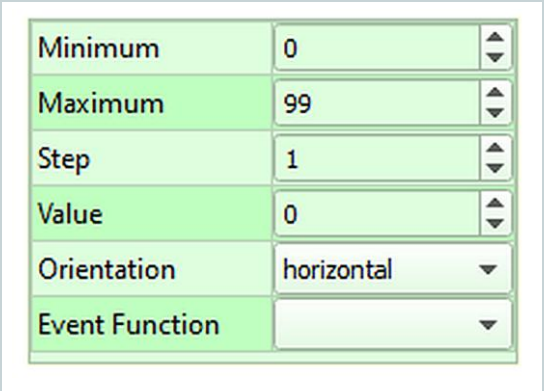
When a Radio Button is inserted, the settings above appear. Set its label in the ‘Text’ field (default ‘Radio Button’) and its default state with the ‘Value’ attribute. Within a container that holds several radio buttons, only one can be checked at a time.

List Box



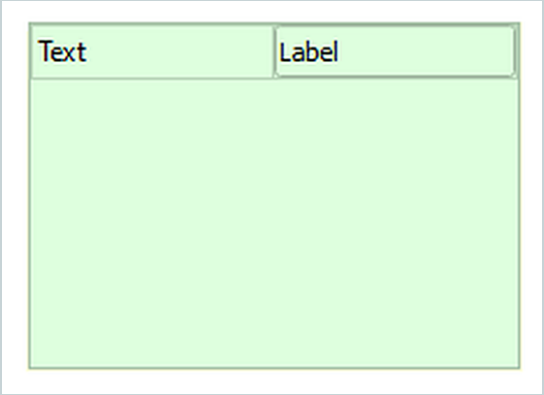
When a List Box is inserted, the following options appear. Use the ‘Value’ attribute to define its default value; its rows are populated from code.

Horizontal and Vertical Slider



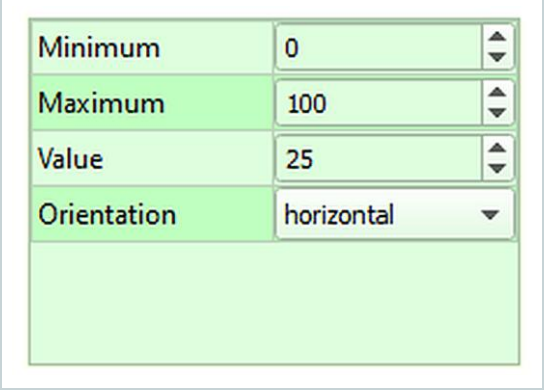
When a Slider is inserted, these options appear. Set the range with ‘Minimum’ and ‘Maximum’, the increment with ‘Step’, and the starting value with ‘Value’. The ‘Orientation’ drop-down switches between horizontal and vertical. Both horizontal and vertical sliders are on the Widgets tab.

Label



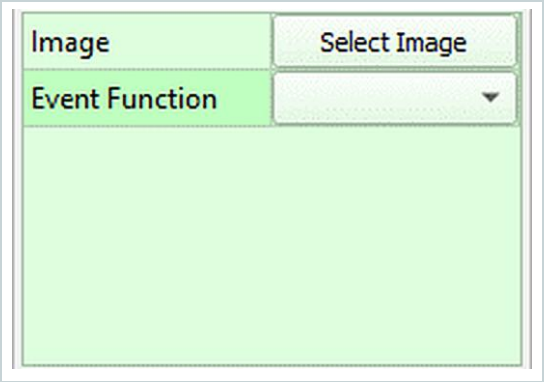
A Label has a single customisable setting: its visible text, which you set in the ‘Text’ field at the top.

Progress Bars



When a Progress Bar is inserted, set the allowed range with ‘Minimum’ and ‘Maximum’ and the starting value with ‘Value’. The ‘Orientation’ drop-down switches between horizontal and vertical; both the horizontal and vertical progress bars are available in this designer.

Image



When an Image element is inserted, set the picture with the ‘Image’ field at the top. Click handling is added in the generated code.

Circular Progress

A Circular Progress indicator shows completion as a ring rather than a bar. Set its range with ‘Minimum’ and ‘Maximum’ and its current value with ‘Value’.

Toggle Switch

A Toggle Switch is an on/off control styled as a sliding switch. The ‘Value’ attribute sets its default state.

Divider

A Divider draws a thin separating line to group parts of the layout; it has no interactive value.

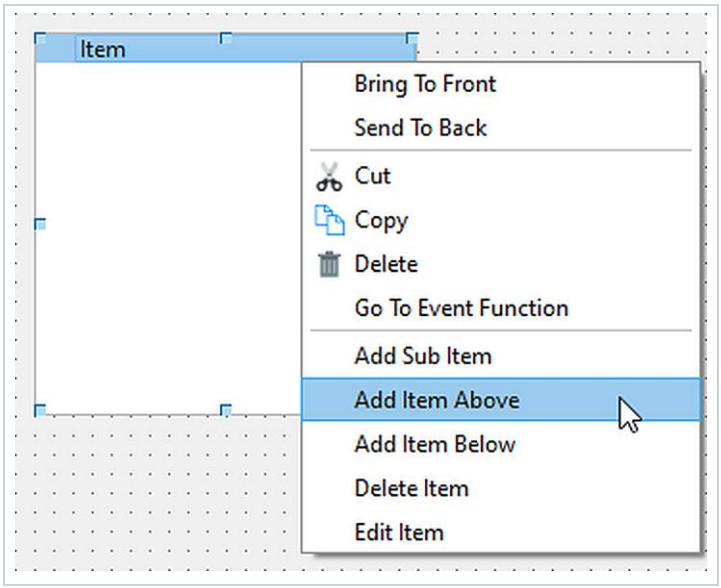
Icon Button

An Icon Button behaves like a Button but shows an icon instead of, or alongside, a text label. Set its icon with the ‘Image’ field; click handling is added in the generated code.

Spinner

A Spinner is an animated busy indicator that shows work is in progress; it has no value to set.

Tree Widget



Tree widget context menu

When a Tree Widget is inserted, set its icon with the 'Image' field at the top. To add an item, right-click the tree and choose Add Item; to add a sub-item, right-click the parent item and choose Add Sub Item; to rename an item, right-click it and choose Edit Item. Item behaviour is wired in the generated code.

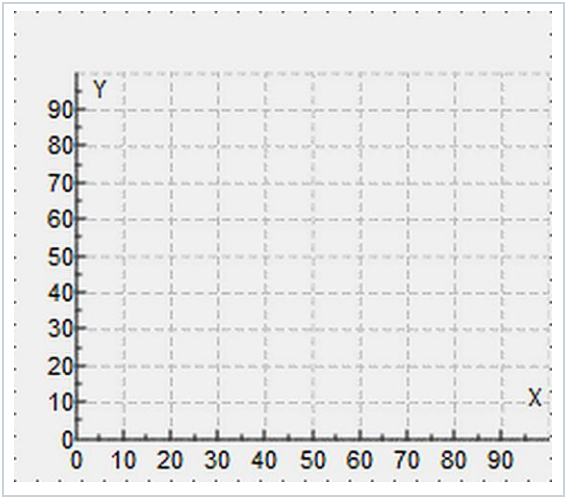
Tables and 2D/3D graphs have no property-panel settings; all of their properties are set from their context menus, and their values are assigned from code (see below).

Using tables, 2D and 3D graphs

Graphs and tables cannot be populated in the GUI Designer itself; they are customised there, but their values are only assigned from code.

2D graphs

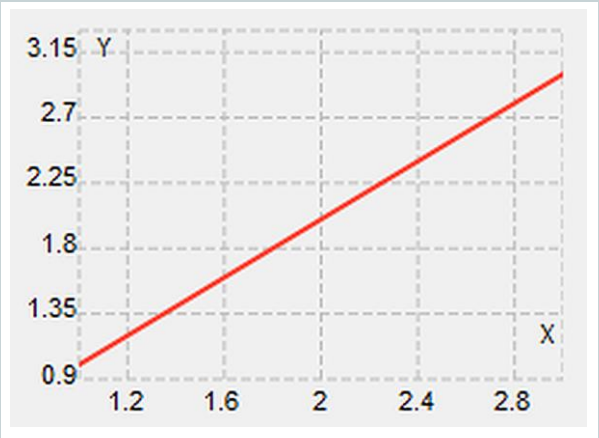
Assign a 2D graph's value with an array of x/y columns.



2D graph

```
set_widget_value(&mut self.graph2d1, vec![vec![1.0, 2.0, 3.0], vec![1.0, 2.0, 3.0]]);
```

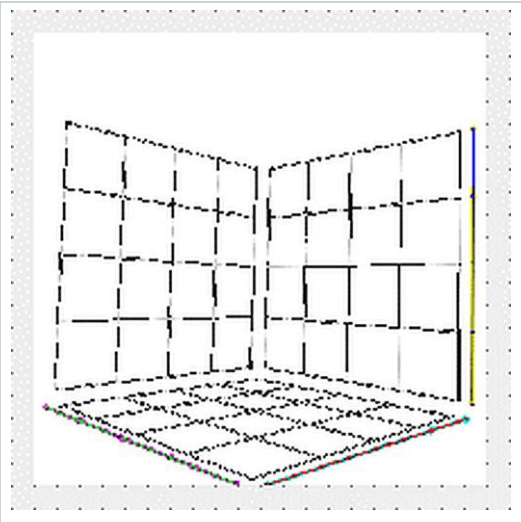
With its value assigned it looks like this:



2D graph with values

3D graphs

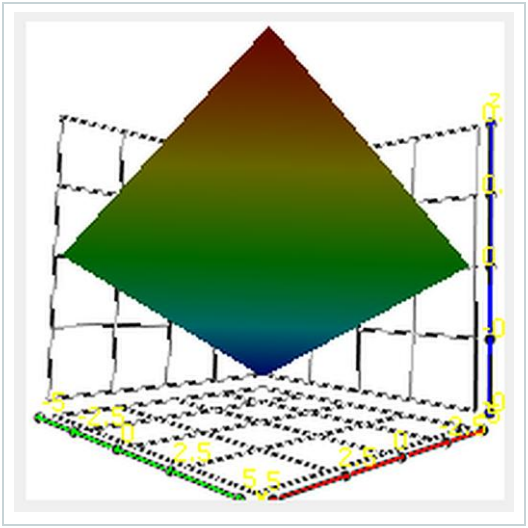
Assign a 3D graph's value with a surface generated in code.



3D graph

```
let a = surface3d(|x, y| (x + y).sin(), -5.0, 5.0, 50, -5.0, 5.0, 50);
set_widget_value(&mut self.graph3d1, a);
```

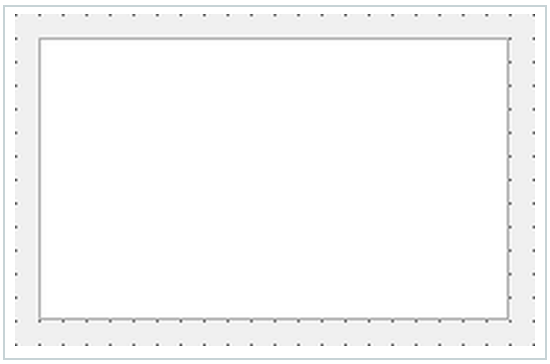
With its value assigned it looks like this:



3D graph with values

Tables

Assign a table's value with an array, just like a 2D graph. Columns are assigned automatically; change them with the table_create() function.



Table

```
set_widget_value(&mut self.table1, vec![vec![1, 2, 3], vec![1, 2, 3]]);
```

With its value assigned it looks like this:

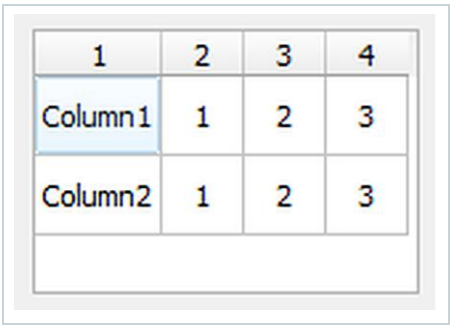
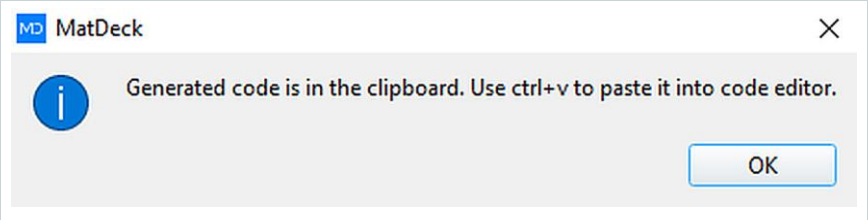


Table with values

Finishing up

When you finish your widget design, press ‘Done’. The designer generates the widget code and pastes it into the script you are working in. The first time you press ‘Done’ for a new widget, the notification window (Picture 8) appears. After that first time, code updates silently.



Picture 8: Notification window

Using the generated Rust (egui) code

> **Note** The Rust (egui) designer generates idiomatic Rust code; the excerpt below illustrates the structure of your generated file.

```
use eframe::egui;

#[derive(Default)]
struct Widget1 {
    // your fields
}

impl eframe::App for Widget1 {
    fn update(&mut self, ctx: &egui::Context, _frame: &mut eframe::Frame) {
        egui::CentralPanel::default().show(ctx, |ui| {
            // Layout generated by the Rust (egui) GUI Designer
            if ui.button("Button").clicked() {
                // your code
            }
            // your widgets
        });
    }
}

fn main() -> eframe::Result<()> {
    let options = eframe::NativeOptions::default();
    eframe::run_native(
        "Widget1",
        options,
        Box::new(|_cc| Box::new(Widget1::default())),
    )
}
```

Widget code

Add your interaction logic to this file. Because the Rust (egui) designer does not generate event-function stubs, behaviour is written directly in the code — see the next section.

> **Notice** The preferred workflow is to create and change widget designs in the GUI Designer, then generate or update the code from there. If you edit the code directly and afterwards open and save changes in the GUI Designer, any edits made directly in the code will be lost.

Interacting with generated GUI code

The GUI Designer generates the code that builds the visual side of the GUI — element positions, titles and properties are all coded as designed. Because the Rust (egui) designer does not add event-function slots, you wire behaviour directly in the generated Rust code.

Reading and updating widgets

egui is immediate-mode: widget state lives in your struct and is read or written each frame, and each widget returns a response you can test. For example, driving a progress bar from a slider:

```
egui::CentralPanel::default().show(ctx, |ui| {
    let response = ui.add(egui::Slider::new(&mut self.value, 0..=100));
    if response.changed() {
        self.progress = self.value as f32 / 100.0;
    }
    ui.add(egui::ProgressBar::new(self.progress));
});
```

Executing and outputting GUIs

Once a design is complete, the generated code is placed straight into the script. GUIs can be output in the following ways:

- Build and Run
- Deploy (native executable)
- Preview Widget

Building and running

The Rust (egui) designer generates a Cargo-ready program. Use Build and Run to compile and launch it, or Deploy to produce a distributable native executable.

The Programming toolbar