**Rust GUI Using GTK**

[[](https://medium.com/@coderhack.com?source=post_page-----96def5980e6a--------------------------------)](https://medium.com/@coderhack.com?source=post_page-----96def5980e6a--------------------------------)

[[CoderHack.com](https://medium.com/coderhack-com?source=post_page-----96def5980e6a--------------------------------)](https://medium.com/coderhack-com?source=post_page-----96def5980e6a--------------------------------)

[Technocrat](https://medium.com/@coderhack.com?source=post_page-----96def5980e6a--------------------------------)

·

Follow

Published in

[CoderHack.com](https://medium.com/coderhack-com?source=post_page-----96def5980e6a--------------------------------)

·

9 min read

·

Sep 16

37

Rust is a modern systems programming language focused on safety, speed, and concurrency. It has become popular in recent years for building fast and robust software. However, Rust’s dedication to safety and low-level control can make GUI development seem challenging.

Fortunately, there are a few GUI crates for Rust that provide safe and idiomatic wrappers around existing frameworks. One such option is the gtk crate, which provides bindings for the GTK+ 3 GUI library.

GTK+ is a popular cross-platform GUI toolkit used to build graphical applications. It works on Linux, Windows, and macOS and has bindings in many languages. The gtk crate allows Rust developers to leverage GTK+ and create fast, responsive GUI applications with a native look and feel.

In this article, we’ll explore how to build GUI applications in Rust using the gtk crate. We’ll cover:

* Setting up a GTK+ development environment
* Creating windows and basic widgets
* Using layout managers to arrange widgets
* Connecting to signals and setting up callbacks
* Drawing custom graphics
* And more!

By the end, you’ll be familiar with Rust GTK+ development and ready to build your own desktop applications!

To follow along, you’ll need:

* The Rust programming language installed. You can get the latest version at [www.rust-lang.org.](http://www.rust-lang.org./)
* The GTK+ 3 development libraries. On Linux, you can install with apt, yum or similar. On macOS, use Homebrew. And on Windows, the GTK+ runtime libraries.

With the prerequisites installed, let’s dive in and build our first Rust GTK+ app!

**Setting up the development environment**

To get started with Rust GUI development using GTK+, you first need to set up your development environment.

**Install Rust**

You will need to install the latest version of Rust. You can download installers for your operating system from [rustup.rs](https://rustup.rs/). Rustup will install Cargo, Rust’s package manager, as well as the Rust toolchain which includes the Rust compiler rustc and the Rust build tool rustup.

**Install GTK+ dependencies**

GTK+ has a few dependencies you need to install:

* libgtk-3-dev - GTK+ 3 development files
* libcairo2-dev - Cairo 2D graphics library development files
* libpango1.0-dev - Font layout and rendering library development files
* libffi-dev - Development files for libffi foreign function interface library

You can install these on Ubuntu with:

sudo apt install libgtk-3-dev libcairo2-dev libpango1.0-dev libffi-dev

**Create a new Rust project**

You can now create a new Rust project using Cargo:

cargo new my-gtk-app

This will create a folder my-gtk-app with a Cargo.toml and src/main.rs. You're now ready to start building a GTK+ app in Rust!

In the next section, we’ll look at creating our first window in GTK+ using Rust.

**Creating a Window**

To build a GUI app, we first need a window to hold our UI elements. In GTK+, windows are represented by the gtk::Window widget. We can create a basic window like this:

use gtk::{Window, WindowType};  
  
fn main() {  
 let window = Window::new(WindowType::Toplevel);  
 window.set\_title("My Window");  
 window.set\_default\_size(350, 70);  
  
 window.show\_all();  
}

This creates a top level window with a title “My Window” and a default size of 350x70. The show\_all() method shows the window and any child widgets on the screen.

The gtk::WindowType enum has several variants to specify the type of window you want:

* Toplevel - A regular window, used for most "main" application windows.
* Popup - A popup menu or tooltip window.
* Utility - A small temporary window like a splash screen or modal dialog.
* Dialog - A dialog window.
* etc.

You can set various properties on the window like:

* set\_title() - The titlebar text
* set\_default\_size() - The default width and height
* set\_position() - The window position
* set\_resizable() - Whether the window can be resized
* set\_modal() - Whether the window is modal (blocks other windows)
* And many more!

Windows are the foundation of any GTK+ app and most widgets are added to a window. In the next section, we’ll look at adding some basic widgets to our window.

**Adding Widgets**

GTK+ provides a variety of widgets that can be added to windows to build interactive user interfaces. Some of the commonly used widgets are:

**Button**

A button is a classic widget for implementing user commands. You can specify the button text as well as an icon.

let button = gtk::Button::with\_label("Click me!");

**Label**

A label widget provides a text string that the user cannot edit. It’s useful for headings, instructions or captions.

let label = gtk::Label::new("Hello World!");

**Entry**

An entry widget allows the user to enter and edit text strings. You can use it to get input from the user.

let entry = gtk::Entry::new();

**TextView**

A text view widget provides a scrollable view onto a text buffer. It allows the user to enter and edit multi-line text.

let text\_view = gtk::TextView::new();  
let text\_buffer = gtk::TextBuffer::new();  
text\_view.set\_buffer(Some(&text\_buffer));

These widgets can be added to a window using the add() method. For example:

window.add(&button);  
window.add(&label);  
window.add(&entry);  
window.add(&text\_view);

This will simply add the widgets to the window. In order to organize them properly, we need to discuss layout managers, which we’ll cover in the next section.

**Layout Managers**

To organize widgets in a window, GTK+ provides various layout managers. These allow you to arrange widgets in horizontal boxes, vertical boxes, grids, and more.

**BoxLayout**

The Box container arranges child widgets in a single row (horizontally) or column (vertically). We can use the gtk::Orientation enum to specify the orientation.

For example, to arrange buttons in a horizontal row, we can use:

use gtk::{Button, Box, Orientation};  
  
let box\_ = Box::new(Orientation::Horizontal, 5);  
box\_.pack\_start(&Button::new\_with\_label("Button 1"));  
box\_.pack\_start(&Button::new\_with\_label("Button 2"));  
box\_.pack\_start(&Button::new\_with\_label("Button 3"));

This will create a Box with horizontal orientation and a spacing of 5 pixels between children. We add three buttons to this box, so they are arranged in a horizontal row.

Similarly, for a vertical column:

use gtk::{Button, Box, Orientation};  
  
let box\_ = Box::new(Orientation::Vertical, 10);  
box\_.pack\_start(&Button::new\_with\_label("Button 1"));   
box\_.pack\_start(&Button::new\_with\_label("Button 2"));  
box\_.pack\_start(&Button::new\_with\_label("Button 3"));

This will arrange the buttons in a vertical column with 10 pixels spacing.

**GridLayout**

For a more complex layout, the GridLayout can arrange widgets in a table-like structure with rows and columns. We can specify:

* The number of rows and columns
* Spacing between rows/columns
* Padding around the edges

For example, a 3x2 grid with 10 pixel spacing would be:

use gtk::{Button, Grid};  
  
let grid = Grid::new();  
grid.set\_row\_spacing(10);  
grid.set\_column\_spacing(10);   
grid.attach(&Button::new\_with\_label("Button 1"), 0, 0, 1, 1);   
grid.attach(&Button::new\_with\_label("Button 2"), 1, 0, 1, 1);  
grid.attach(&Button::new\_with\_label("Button 3"), 0, 1, 1, 1);   
grid.attach(&Button::new\_with\_label("Button 4"), 1, 1, 1, 1);

This will layout the buttons in a 3x2 grid with 10 pixel spacing.

The attach() method attaches a widget to the grid at a given row, column, and spanning certain rows/columns.

**Additional Layouts**

GTK+ includes a few more layout managers like Fixed which allows manually placing widgets at fixed positions, Stack which stacks widgets on top of each other, Viewport which clips its child to a specific size, etc. These provide a lot of flexibility in how you want to layout an interface.

The Rust bindings for all these GTK+ layout managers are available in the gtk crate, so you have the full power of GTK+ layouts in Rust! Let me know if you have any other questions!

**Signals and Callbacks**

Signals and callbacks are an important concept in GTK+ development. They allow you to add interactivity and functionality to your GUI apps.

A signal is emitted when some event occurs, like a button click or text entry. Your app can connect callback functions to these signals, which will be executed when the signal is emitted.

For example, you can connect a callback to the “clicked” signal of a Button. When the user clicks the button, the “clicked” signal will be emitted and your callback will run.

Here’s an example of connecting a callback to the “clicked” signal of a Button in Rust:

// Create a button   
let button = gtk::Button::with\_label("Click me!");  
  
// Connect the "clicked" signal to a callback   
button.connect\_clicked(move |button| {  
 println!("Button was clicked!");  
});  
  
// Add the button to the window  
window.add(&button);

This will print “Button was clicked!” to the console whenever the button is clicked.

Some other common signals and events you may use include:

* “activate” signal on Entries, fired when the user presses Enter
* “changed” signal on Entries, fired when the text changes
* “delete-event” signal on Windows, fired when the user closes the window
* Keyboard and mouse events
* And many more!

Using signals and callbacks, you can make your GTK+ apps interactive and dynamic. They are a core part of GTK+ development with Rust.

Let me know if you would like me to explain anything in this section in more detail! I can also provide more examples and snippets if needed.

**Drawing Area**

The gtk::DrawingArea widget allows you to create custom graphics. It provides a blank canvas that you can draw on using the Cairo graphics library.

To use a DrawingArea, you simply create the widget and connect a "draw" signal callback. In the callback, you can render graphics on the drawing area.

Here’s a basic example that draws a blue circle on a DrawingArea:

use gtk::{DrawingArea, DrawingAreaExt};   
use glib::SignalHandlerId;  
use cairo::{Context, Rectangle};  
  
struct MyWindow {  
 drawing\_area: DrawingArea,  
}  
  
impl MyWindow {  
 fn new() -> Self {  
 let window = Window::new(WindowType::Toplevel);  
  
 let drawing\_area = DrawingArea::new();  
 drawing\_area.set\_size\_request(200, 200);  
  
 window.add(&drawing\_area);  
  
 let draw\_handler = drawing\_area.connect\_draw(|\_, cr| {  
 cr.set\_source\_rgb(0.0, 0.0, 1.0); // Blue  
  
 let rect = Rectangle {   
 x: 0.0,   
 y: 0.0,  
 width: 200.0,   
 height: 200.0   
 };  
  
 cr.arc(100.0, 100.0, 50.0, 0.0, 2.0 \* std::f64::consts::PI);  
 cr.fill(rect);  
 });  
  
 MyWindow {  
 drawing\_area,  
 draw\_handler   
 }  
 }  
}

We create a DrawingArea and set a size request of 200x200 pixels. We connect the "draw" signal to a callback which will be invoked whenever the drawing area needs to be redrawn.

In the callback, we create a Cairo context cr which can be used to render graphics. We set the source color to blue and then draw a circle with a radius of 50 pixels centered in the middle of the drawing area. Finally we fill the entire drawing area rectangle with the blue color, creating a blue circle on a blue background.

This is a very simple example, but it demonstrates how the DrawingArea can be used to create custom graphics and animations in your GTK applications using Cairo. Let me know if you have any other questions!

**Additional Topics**

Beyond the basics we have covered so far, GTK+ and the gtk-rs crate provide many more advanced features for building GUI applications. Here we will briefly discuss some of these additional topics.

**OpenGL Support**

For 3D graphics and hardware acceleration, GTK+ integrates well with OpenGL. The gtk::GLArea widget can be used to draw OpenGL content. The [gtk-rs OpenGL example](https://github.com/gtk-rs/examples/blob/master/opengl/src/main.rs) shows how to set up an OpenGL context and draw a rotating cube with OpenGL in a GTK+ application using Rust.

**Internationalization**

To support multiple languages, GTK+ provides excellent localization support. String resources can be translated and the correct translation for the user’s locale is automatically used. The [gtk-rs i18n example](https://github.com/gtk-rs/examples/tree/master/i18n) demonstrates internationalization of a simple “Hello World” GTK+ app in Rust.

**Architecture of a Larger GTK App**

For building more complex applications with multiple windows, it is important to maintain a clean architecture. Some key principles for larger GTK+ apps in Rust include:

* Separating the GUI (views) from the application logic (models)
* Using signals and callbacks to communicate between views and models
* Employing a state machine or similar to manage the overall application state
* Abstracting repetitive view code into reusable widgets

The [gtk-rsငtodo app](https://github.com/gtk-rs/examples/tree/master/todo-app) is an example of a larger GTK+ app in Rust following these principles.

**Other Features**

* Accessibility
* Clipboard
* Drag and drop
* And [many more](https://www.gtk.org/features.php)! The gtk-rs crate provides a safe API for most of GTK+’s functionality, so building complex GUIs in Rust is very capable.

Let me know if you would like me to explain anything in this section in more detail. I aimed for an overview highlighting some of the major additional topics, but can dive deeper into any part of this if needed for your article.

**Conclusion**

We have covered the basics of building GUI applications in Rust using the GTK+ framework. Rust and GTK+ is a combination that provides both high performance and scalability. Some of the main points we have discussed are:

* Rust is a systems programming language focused on safety, speed and concurrency. It works well for building desktop applications.
* GTK+ is a mature, cross-platform GUI toolkit that works with Rust.
* We can use the gtk crate to build GTK+ applications in Rust.
* We discussed various widgets like Window, Button, Label, Entry, etc. and how to use them.
* We looked at layout managers like BoxLayout and GridLayout to arrange widgets.
* We handled signals and callbacks to add interactivity to our apps.
* We used DrawingArea to create custom graphics and Cairo for more complex drawing.
* We briefly looked at more advanced topics like OpenGL support, Internationalization, etc.

Rust and GTK+ may have a learning curve, but can be used to build fast, robust and good looking desktop applications. I hope this article has provided you a good overview of GUI development in Rust. Let me know if you have any other questions!

Some useful resources for learning more:

* The GTK+ documentation: <https://gtk-rs.org/gtk4-rs/stable/>
* The Rust documentation: <https://doc.rust-lang.org/>
* GTK+ 3 Tutorial: <https://runescape.wiki/w/Gtk%2B_3_Tutorial>
* Various Rust GTK+ code samples: <https://gtk-rs.org/gtk4-rs/stable/code-samples.html>
* GTK+ widgets reference: <https://developer.gnome.org/gtk3/3.24/> I hope this article has been helpful to you!



Photo by [Thomas Tastet](https://unsplash.com/@thomastastet?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

If you found it helpful please support me with 1) click some claps and 2) share the story to your network. Let me know if you have any questions on the content covered.

Feel free to contact me at coderhack.com(at)gmail.com