



STATE OF RUST UI

Nico Burns
Software Engineer

May 6th, 2024

About me



- Full Stack developer for ~15 years
- Working with Rust since ~1.0
- Currently working on:
 - DioxusLabs' Blitz
 - Taffy (layout)
 - Servo
 - Modularisation
 - Layout
 - Connecting the Rust UI ecosystem
- Also maintain a curated crate directory “<https://blessed.rs>”

This talk will:



- Situate Rust UI within the wider context of UI general
- Describe the overall shape of the ecosystem
 - The sorts of solutions that people are building
 - Who is building things. And how they are coordinating with each other.
- Consider “Are we GUI yet”?
 - What capabilities do Rust UIs currently have or not have
 - What work is currently in-progress
 - Future challenges and opportunities
- Discuss Rust’s suitability for GUI development
 - Areas in which the language/toolchain could improve
- Discuss relationship between “native” and “web”



High-Level Overview

“Tier 1” UI Toolkits



- “Actually Native” (OS Native)
 - Win32, WPF, Cocoa (AppKit/UIKit)
 - SwiftUI, Jetpack Compose
- Traditional cross-platform
 - QT, GTK
- Modern cross-platform
 - React Native
 - Flutter
- Web-based
 - Electron, Capacitor
- Other (not the focus of this talk)
 - TUIs, Immediate Mode / Game UIs, Embedded, Document Rendering

90%* of apps
use one of these

*Numbers entirely
made up

Why do UI in Rust?



PRO

- Same reasons we love Rust in other domains:
 - Performance + Type System + Ergonomics, etc
- Cross platform with high fidelity
- Opportunity to replace lower-level C and C++ libraries without losing performance.
- Can potentially write the whole stack in one language
- Can target web (DOM) via WASM efficiently.

CON

- Can be verbose compared to other languages
- Difficult to manage shared state
- Poor compile times
 - Particularly important for UI which often can't be captured by types.
- Reinventing the wheel



- Mostly flutter-style and web-style
 - Flutter-ish: Iced, Slint, Xilem, Makepad, Floem, GPUI, Vizia, Freya, Ribir, KAS, Cushy
 - Web-ish: Servo, Tauri, Blitz (+ “actually web”: Dioxus, Leptos, Sycamore, etc)
 - Minimal “actually native”: FF Crash Reporter, GTK, rfd
- “Cambrian explosion”
 - Similar to frontend web framework ecosystem
 - Lots of options (some bigger than others)
 - Friendly competition
- Highly modular (sometimes)
 - Sharing of underlying libraries (Winit, AccessKit, WGPU, Cosmic-Text, Taffy, etc, etc)
 - Sharing across domains (App UI, Web Engines, Game Engines, even TUI)
 - Not everyone shares this philosophy + Not always practiced perfectly.
- Mix of community and commercially funded work

Wide Range of Projects



- **UI Frameworks:** Iced, Slint, Xilem, Makepad, Floem, GPUI, Vizia, Freya, Ribir, KAS, Cushy, zng
- **Immediate mode:** egui, rui
- **TUI:** ratatui
- **“Actually Native”:** gtk-rs, relm, FF Crash reporter, rfd, native-windows-gui
- **Frontend Web Frameworks:** Dioxus, Leptos, Yew, Sycamore, Sauron
- **Web engines:** Servo, Blitz, Firefox, Chrome, Tauri
- **Game engines:** Bevy, Fyrox
- **Document creation:** Typst, Prince
- **Text Editors:** Lapce, Zed
- **Terminals:** Alacritty, Cosmic Terminal



Are we GUI yet?

Are we GUI yet?

- Library Ecosystem Capabilities
 - Windowing
 - Rendering
 - Layout
 - Input
 - Accessibility & Automation
 - State Management & Reactivity
 - Widgets & “Styling”
 - Everything else



- Developer Experience
 - Documentation
 - Compile Times
 - Rust language issues



Library Ecosystem Capabilities

Windowing



NEEDS

- Create windows
- HiDPI (scale factor)
- Multiple windows
- Sub-windows
 - Embed within existing application
 - Popup menus / Modals
- Mobile
 - Activities
 - Widgets (homescreen, lock screen, start menu, sidebar, etc)
- Web
- Other (full screen, resizing, blur, etc)

AVAILABLE SOLUTIONS

- Winit
 - Glazier -> Merging with Winit
 - Winit refactoring to trait-based API
 - Needs more API coverage
 - Needs better extensibility
 - Platform-specific functionality
- Baseview and nih-plug
 - DAW Plugins
 - Q: Can It be used with winit?
- Makepad
 - Code generation

Rendering: low-level



- Software (CPU) rendering
 - Softbuffer
- OpenGL
 - Glutin, Glow
- Vulkan / Metal / DX12
 - Ash, Metal, d3d12, etc
- WGPU



Traditional Shaders
Compute Shaders

Rendering: High level



Library	Rendering	Users	Notes
skia-safe	CPU/OpenGL/Metal/Vulkan/dx12	Chromium, Vizia, Freya	Fast, capable but C++ and hard to build. Good short-term option.
tiny-skia	CPU	Iced	Rust port of Skia. But slower and CPU only. Good option for pure-rust gpu rendering.
webrender	OpenGL	Firefox, Servo, zng	Fast, capable but poorly documented. May be more usable soon.
femtovg	OpenGL	Slint	Simple API with average performance. Lacks some advanced features.
vger-rs	wgpu	Floem	Good performance but some missing features (like raster images)
vello	wgpu (compute)	Xilem, Blitz	Cutting edge but still immature.
(raw) wgpu	wgpu	Iced, Bevy, Inlyne	You may not even need a high-level library

In general: Poor support for blur, shadows and other similar effects.

Many high-level renderers do not integrate well with other custom rendering code.

Rendering: glyphs



- `wr_glyph_renderer`
 - Delegates to system libraries
 - Freetype on linux
- `swash`
 - Pure rust scaling/hinting
 - Path rendering with “zeno”
 - Fake bold/italic
- `skrifa`
 - Successor to Swash’s hinting
 - From “fontations” project
 - Does scaling / hinting. Only needs a path renderer

Other: Fake bold / Italic.

```
pub trait OutlinePen {  
    // Required methods  
    fn move_to(&mut self, x: f32, y: f32);  
  
    fn line_to(&mut self, x: f32, y: f32);  
  
    fn quad_to(&mut self, cx0: f32, cy0: f32, x: f32, y: f32);  
  
    fn curve_to(  
        &mut self,  
        cx0: f32,  
        cy0: f32,  
        cx1: f32,  
        cy1: f32,  
        x: f32,  
        y: f32  
    );  
  
    fn close(&mut self);  
}
```

Rendering: system compositor



- Use cases:
 - Power-efficient scrolling
 - Embedding: video, webviews, platform-native widgets, other app's content
- Platform support:
 - Windows 8, macOS, iOS, Android, Wayland.
 - Emulate for X11, older windows, web.
- Missing piece of Rust ecosystem
 - WebRender supports the model (used in Firefox)
 - Abandoned Planeshift library (github.com/pcwalton/planeshift)
 - But nothing easily usable

Layout: Open vs. Closed

- Open layout:

- Methods on a trait that allow for arbitrary algorithm to be implemented in "userland"
- Win32, Cocoa, GTK, Flutter
- Iced, Xilem, etc

- Closed layout

- Fixed set of built-in layout widgets/algorithms
- Web, React Native
- Floem, GPUI, Vizia, Blitz

Open layout still requires decisions:

- Who decides final size? (parent / child)
- What information is available? (parent size? sibling sizes?)
- Compare: Web vs. Flutter vs. SwiftUI



```
pub trait Layout {  
    fn compute_child_layout(  
        &mut self,  
        node_id: NodeId,  
        inputs: LayoutInput  
    ) -> LayoutOutput;  
}
```

Extra Open: Flutter's "Sliver"
system for virtualised layout

Layout: Box Layout

In general:

- Tree of boxes
- Fixed/Extrinsic/Intrinsic/Smart sizing
- 1-dimensional (stack/flexbox)
- 2-dimensional (grid)
- Z-order (ability to place boxes on top of each other)



- Web like (Flexbox, CSS grid, etc):
 - + Familiar, expressive, well-specified
 - - Performance, some find confusing
 - React Native, Flutter, 3rd-party
 - Available via Taffy library
 - Floem, GPUI, Blitz, (Xilem?)
- Simplified (Stretch/Fill)
 - + Performance, simpler
 - - Expressivity, maturity of impls.
 - GTK, SwiftUI, Sciter
 - Available via Morphorm crate
 - Iced, Vizia
- Constraint based
 - + Most expressive
 - - !performance!, can be confusing
 - Apple AutoLayout, (QT?)
 - Ratatui

Layout: Text

NEEDS

- Shaping
- Bidi
- Rich text (font sizes, styles)
- Mixed content
(e.g. inline images / widgets)
- Excluded areas
- Floated content
- Padding/border
- Selection (hit testing)
- Editing

AVAILABLE SOLUTIONS

- cosmic-text
- parley
- Future: servo's layout crate?

For shaping:

- harfbuzz (sys, rust-harbuzz, harfbuzz_rs)
- rustybuzz
- allsorts
- swash
- Future: fontations?

Input

Basics

- Mouse (coords/wheel/click)
 - Out of bounds mouse events
- Touch screen
 - multitouch, pressure
- Touchpad gestures
- Keyboard
 - Dead keys
 - Configurable layouts
- Decent-ish support in winit

IME (Input Method Editor)

- Used for:
 - Composed characters (latin accents, CJ characters)
 - Emoji keyboards
 - Autocorrect (some platforms?)
 - Voice input (AI input?)
- Different APIs on different platforms
- Basic support in winit (needs to be better)
- Linebender implementing for Android (+ others?)

State Management / Reactivity



- Traditionally imperative
 - GTK, Win32, Cocoa, etc
- Modern frameworks are declarative
 - React, Flutter, SwiftUI, Jetpack Compose
- Can separate “view” and “widget” layers:
 - React: DOM, Native, SVG, PDF
 - Dioxus, Leptos, Xilem

Different styles:

- Coarse-grained vs. fine grained
- Borrow checking: Callbacks to managed state (Dioxus/Leptos) vs. message passing (Iced/Relm)
- Dynamically typed vs. macro-based vs. type/trait based

```
fn app() -> Element {  
    let mut count = use_signal(|| 0);  
  
    rsx! {  
        h1 { "High-Five counter: {count}" }  
        button { onclick: move |_| count += 1, "Up high!" }  
        button { onclick: move |_| count -= 1, "Down low!" }  
    }  
}
```

```
use leptos::*;  
  
#[component]  
pub fn SimpleCounter(initial_value: i32) -> impl IntoView {  
    // create a reactive signal with the initial value  
    let (value, set_value) = create_signal(initial_value);  
  
    // create event handlers for our buttons  
    // note that `value` and `set_value` are `Copy`, so it's super easy  
    let clear = move |_| set_value(0);  
    let decrement = move |_| set_value.update(|value| *value -= 1);  
    let increment = move |_| set_value.update(|value| *value += 1);  
  
    // create user interfaces with the declarative `view!` macro  
    view! {  
        <div>  
            <button on:click=clear>Clear</button>  
            <button on:click=decrement>-1</button>  
            // text nodes can be quoted or unquoted  
            <span>"Value: " {value} "!"</span>  
            <button on:click=increment>+1</button>  
        </div>  
    }  
}
```

Iced

```
#[derive(Debug, Clone, Copy)]
pub enum Message {
    Increment,
    Decrement,
}
```

```
#[derive(Default)]
struct Counter {
    value: i32,
}
```

Now, let's show the actual counter by putting it all together in our **view logic**:

```
use iced::widget::{button, column, text, Column};

impl Counter {
    pub fn view(&self) -> Column<Message> {
        // We use a column: a simple vertical layout
        column![
            // The increment button. We tell it to produce an
            // `Increment` message when pressed
            button("+").on_press(Message::Increment),

            // We show the value of the counter here
            text(self.value).size(50),

            // The decrement button. We tell it to produce a
            // `Decrement` message when pressed
            button("-").on_press(Message::Decrement),
        ]
    }
}
```

Finally, we need to be able to react to any produced **messages** and change our **state** accordingly in our **update logic**:

```
impl Counter {
    // ...

    pub fn update(&mut self, message: Message) {
        match message {
            Message::Increment => {
                self.value += 1;
            }
            Message::Decrement => {
                self.value -= 1;
            }
        }
    }
}
```



Xilem

```
fn app_logic(data: &mut u32) -> impl View<u32, ()>, Element = impl Widget> {
    Column::new((
        Button::new(format!("count: {}", data), |data| *data += 1),
        Button::new("reset", |data| *data = 0),
    ))
}
```

Accessibility & Automation

ACCESSIBILITY

- Need to:
 - Expose widget tree to system via accessibility APIs (for e.g. screen readers to access)
 - Drive widget updates in reaction to accessibility events
 - Implement focus & keyboard navigation
 - Global font-size / zoom control
 - And much more...
- AccessKit:
 - Abstracts system accessibility APIs
 - Supported: Windows/macOS/Linux
 - Planned: iOS/Android/Web
 - Winit adapter available
 - Adoption: Xilem, Vizia, Freya, zng

AUTOMATION / INTROSPECTION

- Automated testing APIs also:
 - Query UI state
 - Programmatically drive UI
- Devtools
 - e.g. Layout inspector, state inspector
 - Need live updates
 - Edit as well as inspect
 - Floem, Freya have some but bad
 - See: Masonry vision article

← Very hard to implement on web if not rendering to DOM

Everything else (the long tail...)



- System Menus
- File dialogs
- Clipboard
- Drag & Drop between apps
- Tray icons
- Default app file/url handling
- Location
- Sharing
- Biometrics
- Camera
- Storage Access (Photo Gallery, etc)
- Accelerometer
- Bluetooth / NFC
- Look at React Native and Flutter ecosystems for just how deep this rabbit hole goes
- Some of this (e.g. HTTP, SQLite) can be fulfilled by the general Rust ecosystem.
- But there is a LOT



Developer Experience

Documentation


- End-user documentation

- Reference (rustdoc)
- Guide level
- Examples
- Changelog
- Roadmap
- Comparison to similar libraries
- Feature support table
- Where to find community

- Developer documentation

- Code comments
- Git commit messages
- Build/test instructions
- Contribution guidelines
- Architecture docs
- Code / module structure docs
- Key concepts, types, and code flow

People use and contribute to documented libraries!



Dioxus Labs

[Learn](#) [Blog](#) [Awesome](#) [docs.rs](#)

Introduction

Getting Started

Guide

Reference

Your First Component

State

Data Fetching

Full Code

RSX

Components

Props

Event Handlers

Hooks

User Input

Context

Dynamic Rendering

Routing

Resource

UseCoroutine

Spawn

Assets

Choosing A Web Renderer

Desktop

Mobile

Web

OS

Search the docs

CTRL + /

Deploy

Introduction

Dioxus is a portable, performant, and ergonomic framework for building cross-platform user interfaces in Rust. This guide will help you get started with writing Dioxus apps for the Web, Desktop, Mobile, and more.

```
use dioxus::prelude::*;

pub fn App() -> Element {
    let mut count = use_signal(|| 0);

    rsx! {
        h1 { "High-Five counter: {count}" }
        button { onclick: move |_| count += 1, "Up high!" }
        button { onclick: move |_| count -= 1, "Down low!" }
    }
}
```

High-Five counter: 0

Up high!

Down low!

Dioxus is heavily inspired by React. If you know React, getting started with Dioxus will be a breeze.

This guide assumes you already know some Rust! If not, we recommend reading [the book](#) to learn Rust first.

On this page

Features

Multiplatform

Stability

Edit this page!

Go to version

< 0.5

< 0.4

< 0.3

Liveview

Fullstack

Router

Example Project

Reference

Cookbook

Publishing

Anti-patterns

Error Handling

Integrations

State Management

Testing

Examples

Tailwind

Custom Renderer

Optimizing

CLI

Create a Project

Configure Project

Translate HTML

Contributing

Project Structure

Walkthrough of Internals

Guiding Principles

Roadman

https://dioxuslabs.com/learn/0.5/

Iced Guide



Introduction

Learning the Basics

1. Architecture
2. First Steps
3. The Runtime
4. More to come!

Appendix

5. Additional Resources



iced — A Cross-Platform GUI Library for Rust



Introduction

[iced](#) is a cross-platform GUI library for [Rust](#). It is inspired by [Elm](#), a delightful functional language for building web applications.

As a GUI library, iced helps you build *graphical user interfaces* for your Rust applications.

iced is strongly focused on **simplicity** and **type-safety**. As a result, iced tries to provide simple building blocks that can be put together with strong typing to reduce the chance of **runtime errors**.




















This book will:


















- Introduce you to the fundamental ideas of iced.
- Teach you how to build interactive applications with iced.
- Emphasize principles to scale and grow iced applications.




















Before proceeding, you should have some basic familiarity with Rust. If you are new to Rust or feel lost at some point, I recommend you to read [the official Rust book](#).

Iced Examples

 **hecrj** Improve layout of ferris e

Name
 ..
 arc
 bezier_tool
 checkbox
 clock
 color_palette
 combo_box
 component
 counter
 custom_quad
 custom_shader
 custom_widget
 download_progress
 editor
 events
 exit

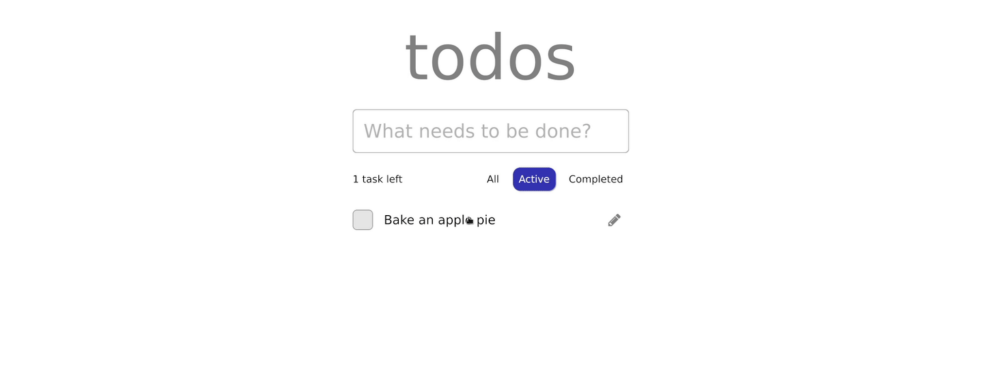
 sierpinski_triangle
 slider
 solar_system
 stopwatch
 styling
 svg
 system_information
 the_matrix
 toast
 todos
 tooltip
 tour
 url_handler
 vectorial_text
 visible_bounds
 websocket
 README.md

 ferris
 game_of_life
 geometry
 gradient
 integration
 layout
 lazy
 loading_spinners
 loupe
 modal
 multi_window
 multitouch
 pane_grid
 pick_list
 pokedex
 progress_bar
 qr_code
 screenshot
 scrollable

Todos

A todos tracker inspired by [TodoMVC](#). It showcases dynamic layout, text input, checkboxes, scrollables, icons, and async actions! It automatically saves your tasks in the background, even if you did not finish typing them.

The example code is located in the [main](#) file.



You can run the native version with `cargo run` :

```
cargo run --package todos
```

Makepad's docs

makepad-widgets v0.6.0

Makepad widgets

[Readme](#) 4 Versions Dependencies Dependents

makepad-widgets

Overview

This is the top-level crate for Makepad Framework, a next-generation UI framework for Rust. Applications built using Makepad Framework can run both natively and on the web, are rendered entirely on the GPU, and support a novel feature called live design.

Live design means that Makepad Framework provides the infrastructure for other applications, such as an IDE, to hook into your application and change its design while your application keeps running. To facilitate this, the styling of Makepad Framework applications is described using a DSL. Code written in this DSL is tightly integrated with the main Rust code via the use of proc macros.

An IDE that is live design aware detects when changes are made in DSL code rather than in Rust code, so that instead of triggering a full recompilation, it can send the changes to the DSL code over to the application, allowing the latter to update itself. (The [makepad-studio](#) crate contains a working prototype of an IDE that will eventually be capable of this, but it is still under heavy development.)

This crate contains a collection of basic widgets that almost every application needs. At the moment of this writing, the following widgets are supported:

- Windows
- Dropdown menus
- Docks
- Splitters
- Tab bars
- Frames
- Scrollbars
- File trees
- Labels
- Buttons
- Checkboxes
- Radio buttons
- Color pickers

In addition to these widgets, this crate also contains re-exports of two lower level crates, namely [makepad-draw-2d](#), which contains all code related to drawing applications, and [makepad-platform](#), which contains all platform specific code. Finally, it contains a collection of base fonts.

In short, to build an application in Makepad Framework, most of the time this crate is the only one you'll need.

Caveats

Although Makepad Framework is complete enough that you can write your own applications with it, it is still under heavy development. At the moment, we only support Mac and web, (though we intend to add support for Windows and Linux very soon). We also make no guarantees at this point with respect to API stability. Please keep that in mind should you decide to use Makepad Framework for your own applications. Finally, we still lack significant functionality in the areas of font rendering, internationalization, etc.



Examples

Simple Example

A very simple example, consisting of window with a button and a counter, can be found in the [makepad-example-simple](#) crate. It's fairly well commented, so this should be a good starting point for playing around with Makepad Framework.

Ironfish (Electronic Synthesizer)

A much more impressive example can be found in the [makepad-example-ironfish](#) crate. Ironfish is an electronic synthesizer written entirely in Makepad Framework. If you want to get an idea of the kind of applications you could build with Makepad Framework, this is the example for you.

Contact

If you have any questions/suggestions, feel free to reach out to us on our discord channel: <https://discord.com/invite/urEMqtMcSd>



Click or press 'S' to search, '?' for more options...



Crate [makepad_widgets](#) 

[source](#) · [\[-\]](#)

Re-exports

```
pub use crate::data_binding::DataBindingStore;
pub use crate::data_binding::DataBindingMap;
pub use crate::tab::TabClosable;
pub use crate::scroll_bars::ScrollBars;
pub use crate::scroll_shadow::DrawScrollShadow;
pub use crate::scroll_bar::ScrollBar;
pub use crate::slides_view::SlidesView;
pub use crate::widget::WidgetSet;
pub use crate::widget::WidgetSetIterator;
pub use crate::widget::WidgetUid;
pub use crate::widget::WidgetDraw;
pub use crate::widget::WidgetDrawApi;
pub use crate::widget::CreateAt;
pub use crate::widget::WidgetActions;
pub use crate::widget::WidgetActionsApi;
pub use crate::widget::WidgetActionItem;
pub use crate::widget::WidgetRef;
pub use crate::widget::Widget;
pub use crate::widget::WidgetRegistry;
pub use crate::widget::WidgetFactory;
pub use crate::widget::WidgetAction;
pub use crate::widget::DrawStateWrap;
pub use makepad_draw::makepad_platform;
pub use makepad\_draw;
pub use makepad\_derive\_widget;
```



Crate **parley** 

[source](#) · [\[-\]](#)

[\[-\]](#) Rich text layout.

Re-exports

```
pub use context::LayoutContext;  
pub use font::FontContext;  
pub use layout::Layout;  
pub use fontique;  
pub use swash;
```

Modules

[context](#) Context for layout.
[font](#)
[layout](#) Layout types.
[style](#) Rich styling support.

Structs

[Font](#) Owned shareable font resource.

cosmic-text (docs.rs)

Crate [cosmic_text](#) 

[source](#) · [-]

COSMIC Text

This library provides advanced text handling in a generic way. It provides abstractions for shaping, font discovery, font fallback, layout, rasterization, and editing. Shaping utilizes rustybuzz, font discovery utilizes fontdb, and the rasterization is optional and utilizes swash. The other features are developed internal to this library.

It is recommended that you start by creating a [FontSystem](#), after which you can create a [Buffer](#), provide it with some text, and then inspect the layout it produces. At this point, you can use the [SwashCache](#) to rasterize glyphs into either images or pixels.

```
use cosmic_text::{Attrs, Color, FontSystem, SwashCache, Buffer, Metrics, Shaping};

// A FontSystem provides access to detected system fonts, create one per application
let mut font_system = FontSystem::new();

// A SwashCache stores rasterized glyphs, create one per application
let mut swash_cache = SwashCache::new();

// Text metrics indicate the font size and line height of a buffer
let metrics = Metrics::new(14.0, 20.0);

// A Buffer provides shaping and layout for a UTF-8 string, create one per text widget
let mut buffer = Buffer::new(&mut font_system, metrics);

// Borrow buffer together with the font system for more convenient method calls
let mut buffer = buffer.borrow_with(&mut font_system);
```



```
// Set a size for the text buffer, in pixels
buffer.set_size(80.0, 25.0);

// Attributes indicate what font to choose
let attrs = Attrs::new();

// Add some text!
buffer.set_text("Hello, Rust! 🦀\n", attrs, Shaping::Advanced);

// Perform shaping as desired
buffer.shape_until_scroll(true);

// Inspect the output runs
for run in buffer.layout_runs() {
    for glyph in run.glyphs.iter() {
        println!("{:?}", glyph);
    }
}

// Create a default text color
let text_color = Color::rgb(0xFF, 0xFF, 0xFF);

// Draw the buffer (for performance, instead use SwashCache directly)
buffer.draw(&mut swash_cache, text_color, |x, y, w, h, color| {
    // Fill in your code here for drawing rectangles
});
```

taffy (docs.rs)



Crate [taffy](#)  [source](#) · [-]

Taffy

Taffy is a flexible, high-performance library for **UI layout**. It currently implements the Flexbox, Grid and Block layout algorithms from the CSS specification. Support for other paradigms is planned. For more information on this and other future development plans see the [roadmap issue](#).

§ Architecture

Taffy is based on a tree of “UI nodes” similar to the tree of DOM nodes that one finds in web-based UI. Each node has:

- A [Style](#) struct which holds a set of CSS styles which function as the primary input to the layout computations.
- A [Layout](#) struct containing a position (x/y) and a size (width/height) which function as the output of the layout computations.
- Optionally:
 - A `Vec` set of child nodes
 - “Context”: arbitrary user-defined data (which you can access when using a “measure function” to integrate Taffy with other kinds of layout such as text layout)

Usage of Taffy consists of constructing a tree of UI nodes (with associated styles, children and context), then calling function(s) from Taffy to translate those styles, parent-child relationships and measure functions into a size and position in 2d space for each node in the tree.

High-level API vs. Low-level API

Taffy has two APIs: a high-level API that is simpler and easier to get started with, and a low-level API that is more flexible gives greater control. We would generally recommend the high-level API for users using Taffy standalone and the low-level API for users wanting to embed Taffy as part of a wider layout system or as part of a UI framework that already has it’s own node/widget tree representation.

High-level API

The high-level API consists of the [TaffyTree](#) struct which contains a tree implementation and provides methods that allow you to construct a tree of UI nodes. Once constructed, you can call the [compute_layout_with_measure](#) method to compute the layout (passing in a “measure function” closure which is used to compute the size of leaf nodes), and then access the layout of each node using the [layout](#) method.

When using the high-level API, Taffy will take care of node storage, caching and dispatching to the correct layout algorithm for a given node for you. See the [TaffyTree](#) struct for more details on this API.

Examples which show usage of the high-level API include:

- [basic](#)
- [flexbox_gap](#)
- [grid_holy_grail](#)
- [measure](#)
- [cosmic_text](#)

In particular, the “measure” example shows how to integrate Taffy layout with other layout modalities such as text or image layout when using the high level API.

§ Low-level API

The low-level API consists of a [set of traits](#) (notably the [LayoutPartialTree](#) trait) which define an interface behind which you must implement your own tree implementation, and a [set of functions](#) such as [compute_flexbox_layout](#) and [compute_grid_layout](#) which implement the layout algorithms (for a single node at a time), and are designed to be flexible and easy to integrate into a wider layout or UI system.

When using this API, you must handle node storage, caching, and dispatching to the correct layout algorithm for a given node yourself. See the [crate::tree::traits](#) module for more details on this API.

Examples which show usage of the low-level API are:

- [custom_tree_vec](#) which implements a custom Taffy tree using a `Vec` as an arena with `NodeId`’s being index’s into the `Vec`.
- [custom_tree_owned_partial](#) which implements a custom Taffy tree using directly owned children with `NodeId`’s being pointers.
- [custom_tree_owned_unsafe](#) which implements a custom Taffy tree using directly owned children with `NodeId`’s being pointers.

Compile times



- Compiler improvements
 - Macro caching
 - Faster / Incremental linking (mold/wild)
 - Faster codegen / JIT (rustc_codegen_clif)
 - Improve parallelism (e.g. frontend)
 - Stable ABI / dynamic loading
- Toolchain improvements
 - Better support for code generation / feature flags
 - Binary deps (branch switches are painful)
- Hot reloading
 - Dioxus / Leptos / Bevy have this
- Directly optimize crates (reduce bloat)
 - Platform crates (windows-rs, etc)
 - Common crates (syn, serde, etc)

Rust Language Issues (1/2)



- Use of Rc/Arc is verbose. Implicit clone would help (opt-in per type?)
- Methods cannot borrow only some fields of their struct
 - Works for closures. Main blocker is syntax
- Lack of support for "partial default".
 - Named/Optional arguments and/or per-field defaults for structs
 - Would also benefit API clients (similarly high-level code)

Rust Language Issues (2/2)



- Orphan rules / delegation
 - Harms modularity and interoperability across the ecosystem
- Specialization
 - Would allow for more ergonomic APIs that automatically special-case certain types
 - A kind of “overloading” - again very nice for high-level APIs
- External build system / Code generation
 - Could help reduce bloat (compile times)
 - Useful for customizing crates (e.g. stylo)



Takeaways

The Road Ahead



Short-Medium Term

- Getting text right (Layout, IME)
- Accessibility / Automation / Introspection
- Winit improvements
- Build out widget library(s)
- Document, Document, Document!

Medium-Long Term

- Compiler / toolchain improvements
- System Compositor
- Long tail of integrations

THANK YOU

Any Questions?

GOSIM 2024
EUROPE

