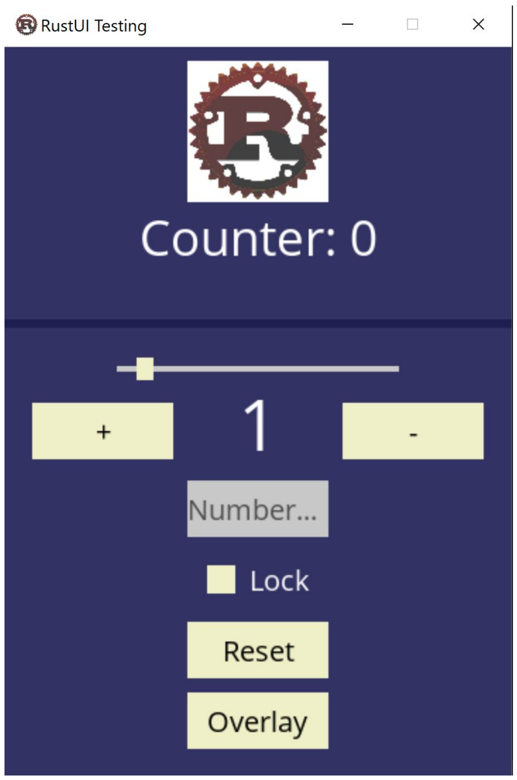
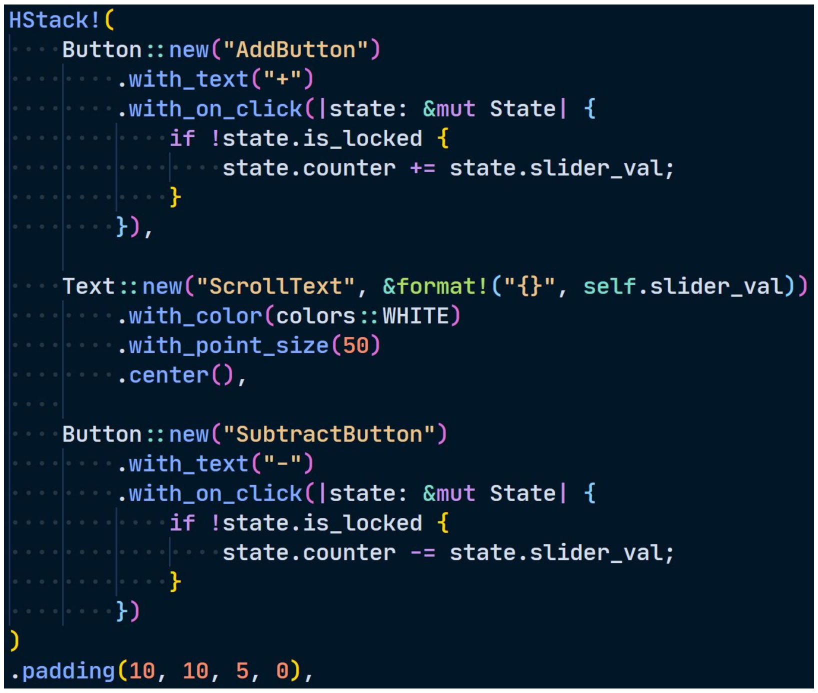
RustUI Writeup



Sections:

1 - RustUI



```
HStack!(
    Button::new("AddButton")
        .with_text("+")
        .with_on_click(|state: &mut State| {
            if !state.is_locked {
                state.counter += state.slider_val;
            }
        }),

    Text::new("ScrollText", &format!("{}", self.slider_val))
        .with_color(colors::WHITE)
        .with_point_size(50)
        .center(),

    Button::new("SubtractButton")
        .with_text("-")
        .with_on_click(|state: &mut State| {
            if !state.is_locked {
                state.counter -= state.slider_val;
            }
        })
)
.padding(10, 10, 5, 0),
```

RustUI is a user interface library inspired by SwiftUI. The goal of creating this library was to explore declarative-style syntax and its role in simplifying the UI-building experience. Pictured above is an example of this syntax in action. This particular portion of code is responsible for generating the [“+” “1” “-”] row pictured in the topmost image (full source code for the example can be found here).

My primary focus while developing the syntax for the library was clarity. Libraries such as Python’s tkinter, for example, can quickly become tedious, requiring the user to follow a pattern of “declare widget, define properties, define layout, add to view”. I wanted to create a syntax allowing for all of this in one go where a reader of this code can simply look at the UI declaration and know exactly what will be produced.

Furthermore, state management is a common problem--particularly in Rust--when it comes to UI. To solve this, I require the user to pass a mutable reference to their state struct to be maintained by the backend. While this means the user can no longer interact with this struct manually (unless the state is an Rc<RefCell>>, see postmortem (**11**)), doing this works around Rust’s strict lifetime and mutability rules. As seen above, widget callbacks allow for easy state mutation. I will go more in depth about state in section **10**.

2 - Learning Rust

When I decided to create a UI library in Rust, I knew very little about Rust and even less about graphical applications. To learn Rust initially, I decided to try something simple: a linked list. Little did I know that the first rule of Rust is “don’t make a linked list”. Although I have yet to create a linked list in Rust, I did learn (painfully and slowly) about what makes Rust such a good language: the borrow checker, lifetimes, options/results, mutability, functional style, etc.

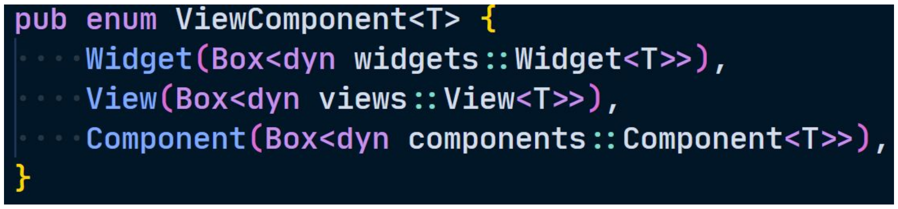
3 - Learning SDL2

At first, Rust seemed like a rather difficult language where nothing works as you would expect it to. However, I decided that the best way to learn Rust was to jump into my second task: learning SDL2. To do so, I used the sdl2 library, a great, idiomatic wrapper around the SDL2 C API. I began by writing a simple paint program, then a raycaster, and finally a sprite animator. Doing this gave me a decent understanding of rendering, input handling, and the basic feel of Rust. Although I learned more about SDL2 and less about Rust as a language, I felt confident enough after these projects to begin tackling RustUI.

4 - Widgets & Views

I began writing widgets with object storage in mind. I therefore began by creating a widget trait, something that I could use to derive multiple widgets and store them in a single view data structure. After creating buttons, checkboxes, and text widgets, I began working on views. Inspired by SwiftUI, I created macros for instantiating views (VStacks and HStacks). These macros, although poorly executed in retrospect (**11**), allowed for views to be built up of widgets, and allowed views to be nested inside each other. Then, after creating a naive sizing API, I was able to layout widgets and nested views. I ran into a new problem, however: how can I store either a widget *and* a view inside the same data structure?

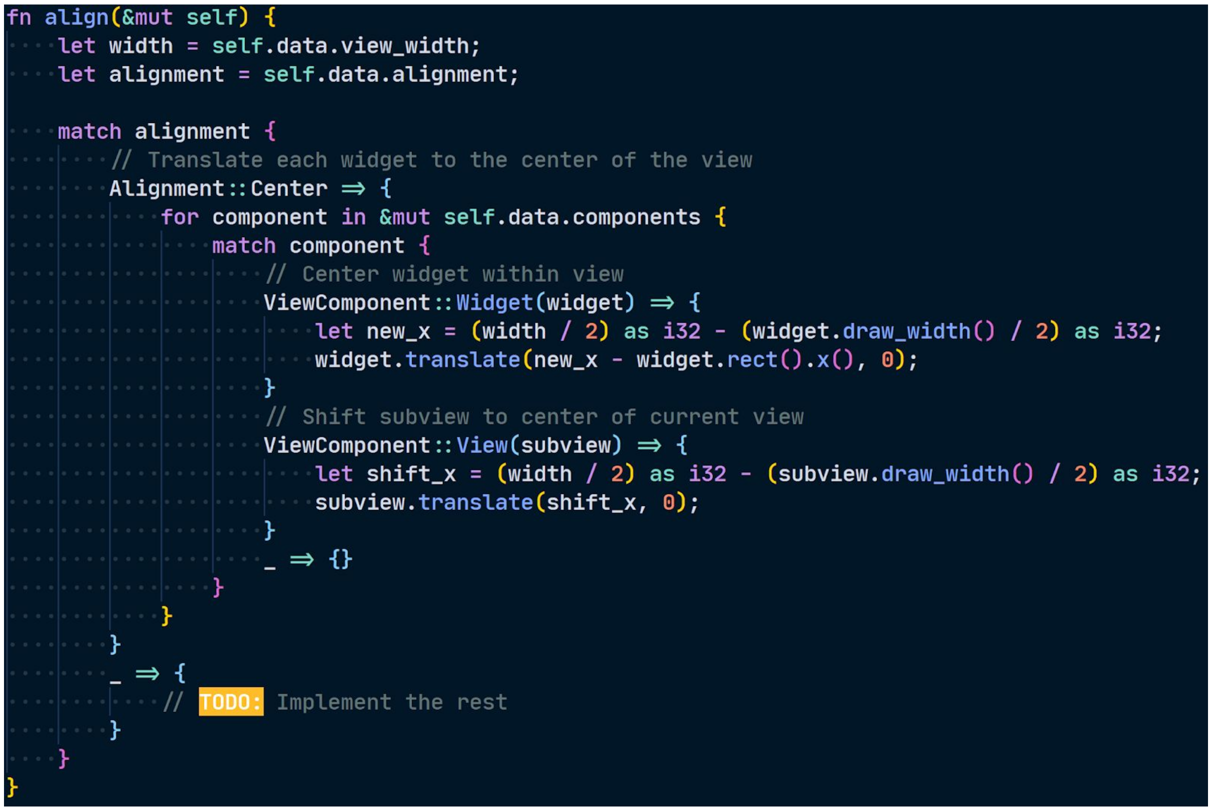
5 - Enums for Dynamic Typing



```rust
pub enum ViewComponent<T> {
    Widget(Box<dyn widgets::Widget<T>>),
    View(Box<dyn views::View<T>>),
    Component(Box<dyn components::Component<T>>),
}
```

At first, I tried using Rust's any module. Simply put, this was beyond my skills at the time. While investigating this topic, though, I learned about the power of Rust's enums. Because I knew exactly which types could be stored within a view, I could simply create an enum with variants for each of these types (where "T" is the state type). This allowed me to layout views composed of different components and handle them within my instantiation macros.

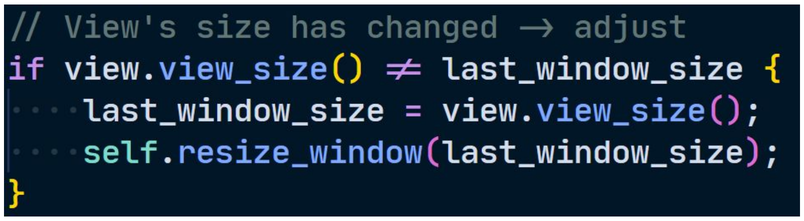
6 - View Layout



```rust
fn align(&mut self) {
    let width = self.data.view_width;
    let alignment = self.data.alignment;

    match alignment {
        // Translate each widget to the center of the view
        Alignment::Center => {
            for component in &mut self.data.components {
                match component {
                    // Center widget within view
                    ViewComponent::Widget(widget) => {
                        let new_x = (width / 2) as i32 - (widget.draw_width() / 2) as i32;
                        widget.translate(new_x - widget.rect().x(), 0);
                    }
                    // Shift subview to center of current view
                    ViewComponent::View(subview) => {
                        let shift_x = (width / 2) as i32 - (subview.draw_width() / 2) as i32;
                        subview.translate(shift_x, 0);
                    }
                    _ => {}
                }
            }
        }
        _ => {
            // TODO: Implement the rest
        }
    }
}
```

Laying out views turned out to be quite simple. My Widget, View, and later Component traits all require a "draw_width" and "draw_height" function. Views can then layout objects using these methods like pictured above--the VStack alignment function. Many of these functions end up being recursive by nature, assisting in this process. For example, the "subview.translate" call will in turn call "subview.translate" for any nested views and "widget.translate" for any nested widgets, thus enforcing proper alignment. Similar mechanisms apply for widgets and their own components (such as nested text within a button).

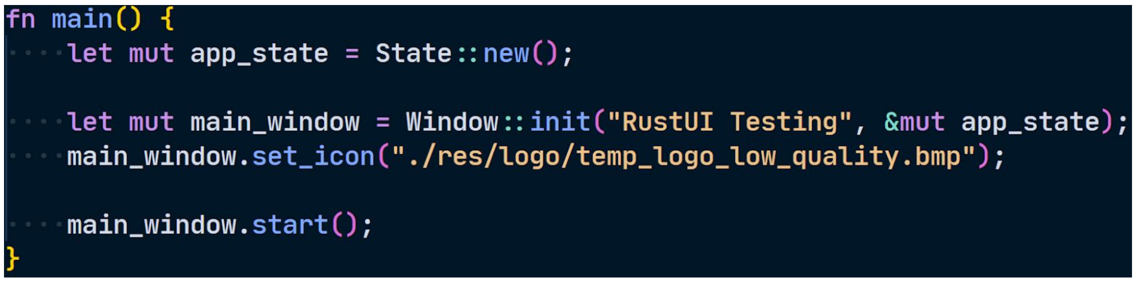
7 - Window Layout



```rust
// View's size has changed → adjust
if view.view_size() ≠ last_window_size {
    last_window_size = view.view_size();
    self.resize_window(last_window_size);
}
```

Window layout is done by simply querying the root view. Because views are inherently tree structures, the “main” view, the one referenced by the backend at runtime, must contain, and therefore fit, all subcomponents. Thus, querying this view’s size (which also accounts for padding) will give the required window size. The user can also add “.fixed_width”, “.fixed_height”, or “.fixed_size” if a constant size is desired.

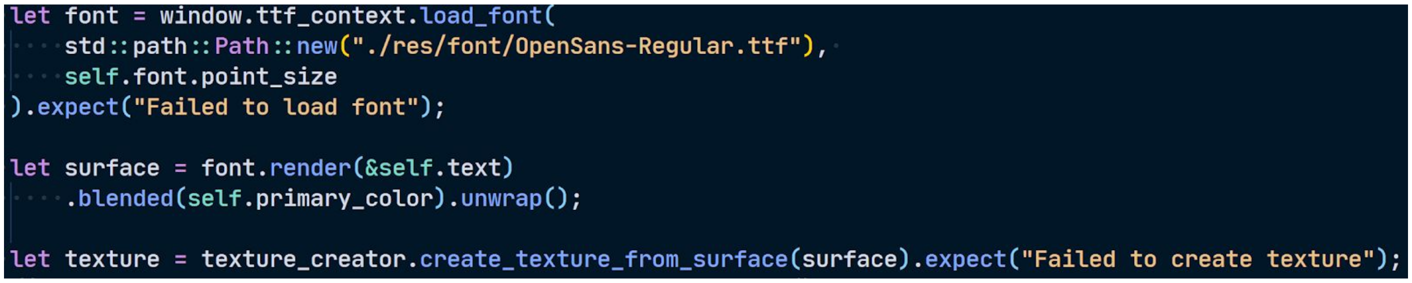
8 - The Backend



```rust
fn main() {
    let mut app_state = State::new();

    let mut main_window = Window::init("RustUI Testing", &mut app_state);
    main_window.set_icon("./res/logo/temp_logo_low_quality.bmp");

    main_window.start();
}
```

The backend handles all states and offers a simple wrapper around SDL2. For example, if a widget can be focused (locks inputs such as text inputs), hovered, or clicked, the backend maintains this state. You might also notice that no view is being passed before starting the application. This is because the state is required to implement a view-generating trait (more details in State (**10**)). Thus, the state type can live within the backend.

9 - Text



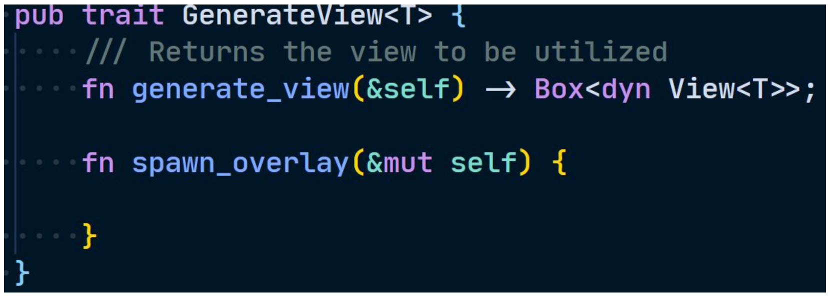
```rust
let font = window.ttf_context.load_font(
    std::path::Path::new("./res/font/OpenSans-Regular.ttf"),
    self.font.point_size
).expect("Failed to load font");

let surface = font.render(&self.text)
    .blended(self.primary_color).unwrap();

let texture = texture_creator.create_texture_from_surface(surface).expect("Failed to create texture");
```

SDL2 has a TrueType font extension, allowing for easy font rendering. The Rust SDL2 wrapper, however, contains an issue caused by lifetimes. My approach to fonts was to store them in a map, but font objects inherit the lifetimes of the TrueType context. While this would normally be fine, Rust does not allow structs to contain references to other items in the same struct (to prevent dangling pointers). This meant I could not have an object that owned both the TrueType context and the loaded fonts--a rather annoying limitation.

With my current knowledge of Rust, I work around this issue, but while creating RustUI, this issue caused me days of frustration. When Rust beginners complain about fighting the borrow checker and its rules, this is exactly what they are referring to. Hours of annotating lifetimes only to get "...but data from x flows into y" errors made me want to give up on font rendering altogether.

The easiest solution would have been to create a new struct at the root of my application which stores both the TrueType context and the application struct which nests the fonts. Implementing this seemed very incorrect though, so I instead chose to simply reload the font each render cycle and worry about this when it mattered. This hurt performance significantly (thus making the library impractical to use), but at least I could continue developing the library.

10 - State



```rust
pub trait GenerateView<T> {
    /// Returns the view to be utilized
    fn generate_view(&self) -> Box<dyn View<T>>;

    fn spawn_overlay(&mut self) {

    }
}
```

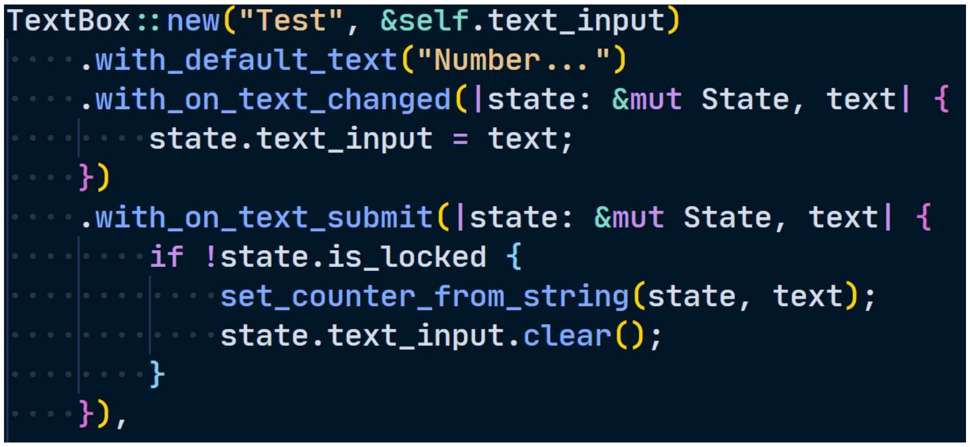
As hinted at earlier, user state is simply a struct with a "generate_view" method. By requiring this trait, the user is free to represent their data however they like, and they are free to generate a view from the data however they like. I quickly ran into an issue, however, in that views cannot actually modify the state.

I decided to take the functional approach to this, passing the state to view elements via event callbacks, and replacing the old state each render cycle rather than mutating it. This led to my current design which essentially treats view components as buffers where information is gathered each frame, then the state is updated with that information before a new view is generated from that new state.

Once again, looking back, this is a bad approach to state (purely functional languages have optimizations specifically for this kind of “mutation”--Rust does not). At the time, I experimented with “Rc” and “Cell” types, but these were beyond my understanding. Now, of course, I understand that a simple Rc<RefCell>> would have solved all my problems, likely leading to an entirely different library architecture. To me, though, this again points out Rust’s greatest weak point: it is anything but beginner friendly. Writing effective Rust requires an understanding of not only the borrow checker, but also the various idiomatic, “Rusty” patterns and type systems.

Ultimately, treating view components as state-modifying buffers slowed down my development (and the library’s performance), causing me to solve problems that never should have been there to begin with. My TextBox implementation showcases almost all of these flaws.

11 - Retrospective



```
TextBox::new("Test", &self.text_input)
    .with_default_text("Number...")
    .with_on_text_changed(|state: &mut State, text| {
        state.text_input = text;
    })
    .with_on_text_submit(|state: &mut State, text| {
        if !state.is_locked {
            set_counter_from_string(state, text);
            state.text_input.clear();
        }
    }),
```

Considering that RustUI was my first real look into Rust, it was successful in meeting its goals. I started this project in hopes of exploring declarative UI patterns, and I found these to be quite effective. While the library itself is nothing exceptional with many flaws worthy of a rewrite, my focus was almost purely on the API. In other words, I wanted to create an intuitive UI library for the *user*.

Looking at a more featured example, the library succeeds where I feel that FFI wrappers and imperative-style libraries fail: readability. Consider the TextBox example above. Even if a reader knew nothing about the library, what happens is perfectly clear. There is a TextBox. It has default text. It has an “on_text_changed” event. It has an “on_text_submit” event. There is no nesting of labels or objects; there is no imperative code which comes with repeated “object.property” calls that can be spread throughout a disorganized block of code. Even the application itself is quite clear with obvious functions like “set_icon”, “init”, and “start”.

1.11 - Future Work

RustUI is not meant for real-world use. This would have been nice, but my knowledge of both graphical applications and Rust itself was essentially nonexistent before starting this project, leading to a number of usability issues (mainly performance). Were I to continue this project, however, there are a number of improvements and additions to be made.

The first issue is the graphics. Because I used SDL2's canvas API, I was essentially limited to textures and rectangles. Furthermore, I did not leave room for animations or easing functions. Solving the graphical capability issue is easy enough: create a simple OpenGL abstraction for drawing textures and shapes. Similarly, an animation trait could be created and implemented for view components, allowing for the modification of fields over time.

The second issue is state management. In Rust, the idea of interior mutability addresses almost all of my issues with RustUI. The idea here is that rather than worrying about lifetimes, shared references allow for **runtime** safety checks. This is important in cases such as UI libraries because the user can guarantee that it is safe to mutate certain fields on otherwise immutable objects. For example, a TextBox widget could simply hold a **shared reference** to some String owned by the user state. In practice, this is exactly as easy as it sounds, although actual implementation varies on use case (see above link on interior mutability). The Rust compiler would never allow an actual mutable reference to such a String because immediately, no more mutable references to the state are allowed to exist as per borrow checker rules. Shared references simply act as single-threaded mutexes, moving safety checks to runtime. This could have fixed my issue with view-state persistence (treating view elements as buffers).

A third issue is the view elements themselves. I ended up essentially copying/pasting much of my code (mainly for view types) when I could have simply created default-implemented traits. I chose to put most of my functionality into major traits when I could have created smaller, shared traits, allowing for easier extensibility. For example, all view types have "update", "render", and sizing functions. This could have been a trait of its own, perhaps "Renderable".

Finally, I want to address the view generation macros. What I should have done was use the macros as an alias for "vec!", then layout the views at another point. Instead, I ended up with a rather confusing and repetitive series of operations that attempt to layout, initialize, and align the views in non-intuitive, disparate ways.

12 - Conclusion

The featured example of RustUI showcases the benefits of declarative UI patterns. The interface can be described clearly, and features can be added using self-describing builder methods. Furthermore, minimizing backend complexity (windowing, event handling, icons, etc.) keeps the library simple and easy to use. Just call "init" and "start", and you are good to go.

While this limits features like custom renderers and windowing backends, I believe that simplicity is the key to developing a good **and** likable library (I'm looking at you, OpenGL). Simplicity does not necessarily imply limiting features, rather, it means keeping the end user in mind, and catering to their use cases. The user of a library like RustUI would simply want to create a UI. The user base I imagined would not care about writing their own backend just to put a button in a window. Similarly, a library like SDL2 might seem "complex", but in reality, its clear focus and target audience makes it a rather simple library to use; thus, I would consider SDL2 to be a good example of a “good and likable” library.

Finally, I would like to touch on the idea of failure: I would be happy to say that RustUI was a failure. I would never recommend it for real-world use, and I would also say it needs a complete rewrite to address the performance and extensibility issues behind this conclusion. However, what I learned from this project ultimately makes it a success. Not only did I learn more about Rust, I learned about SDL2, graphical applications, library design/implementation, and many good programming practices. The latter of these outweighs any shortcomings of this project and is a direct result of my so-called failures.