The Pragmatic Programmers

Advanced Hands-on Rust

Level up Your Coding Skills

Herbert Wolverson

edited by Tammy Coron

Preface

Are you ready to accelerate your Rust development?

In this book, you'll gain intermediate to advanced Rust skills as you build a game development toolkit. Each chapter includes hands-on, practical development, creating tools to help you unleash your creativity and quickly build your own games.

As you work through this book, you'll:

- Create reusable libraries and improve your code with testing, benchmarking and optimization.
- Discover how to unlock the power of Rust's trait and generic meta-programming systems to create code that adapts to fit your needs.
- Customize libraries with feature flags and language syntax with macros.
- Master concurrency with threads and asynchronous programming.
- Find out how to structure your games with reusable state management, menus, user interface elements and asset management.
- Learn how to quickly build games with reusable physics and collision detection.
- Make your games pop with animations, particles and rendering tricks including parallax layering.

This book is the ideal follow-up to take the skills you gained in *Hands-on Rust* *[Wol21]* to the next level or it can be read stand-alone once you've mastered basic Rust.

Who Should Read this Book?

If you've just completed *Hands-on Rust*, this book is the next step. However, it doesn't require that you've read *Hands-on Rust*. If you have mastered basic Rust and want to take your skills to the next level, this book can help you discover: benchmarking, testing, optimization, library design, generic programming, macros, and designing reusable code to make your project development—game or otherwise—easier.

What's in this Book?

This book teaches about library design, gradually introducing you to intermediate and advanced Rust concepts. Each chapter contains hands-on, practical examples of the concepts you are learning—made fun by using them to build and improve games.

In Chapter 1, Set up Rust and the Bevy Engine, on page ?, you'll ensure that you have a working Rust setup, and setup the Bevy Engine.[1] You'll build a basic example—moving a character around the screen—and will become familiar with the basic concepts that make the Bevy Engine tick.

In Chapter 2, Create and Test Your First Library, on page ?, you'll create your first Rust library. Starting with "hello, library world," you'll build up to generating random numbers as a library service. You'll unit test your library functions, and integration test your systems by building a simple dice game. This builds up to your first interaction with generic functions: adapting your random number generation to fit the type of data requested by the function caller.

Moving on to Chapter 3, Optimize and Benchmark Your Library, on page ?, you'll use Rust and Criterion to benchmark your random number generation library. Benchmarking allows you to prove that your optimizations helped and is an important part of software development. Once you've established baseline performance, you'll use feature flags to offer different random number generation algorithms to the library consumer and will benchmark each one. Once you've selected an appropriate default—the fastest algorithm—you'll wrap your random number generator in a Bevy plugin and offer different scheduling and mutability characteristics with the use of interior locking. Once again, you'll unit test and integration test your library with the Pig dice game.

In Chapter 4, Document Your Library, on page ?, you'll focus on another important aspect of library development: documentation. Rust includes tools to make documenting libraries easier—even unit-testing your documentation examples. If you're ever going to share your library, good documentation is essential.

1. https://bevyengine.org/

Chapter 5, Build Reusable Game State Management, on page ? starts by creating a Bevy version of Flappy Dragon. Since most arcade-style games require a main menu and game over screen, this chapter walks you through creating a generic system for loading menus and applying them to games you create. State management can quickly become complicated, with states requiring sub-states to track progress through more complicated games. Because even the generic state-management syntax is becoming complicated, you'll create macros to create a more usable syntax. You'll then apply what you've learned to both Flappy Dragon and Pig.

Chapter 6, Manage Your Game Assets, on page ? continues the theme of automating common tasks with library code by tackling game assets. You'll build an asset manager that lets you specify which sprites and sounds to load up-front and provides clear error messages if an asset is unavailable. Bevy loads assets asynchronously in the background. This can be wonderful for large titles, but it can also lead to embarrassing pauses as your game finishes loading a sprite and it suddenly appears on the screen. This chapter will teach you how to avoid this by wrapping your asset loading in a loading screen, showing progress as assets load—and not running the game before everything is ready.

In Chapter 7, Teach Your Dragon to Fly, on page ?, you'll flesh out Flappy Dragon. You'll combine your asset system with per-frame animation and timers to provide smoothly rendered animations to the game. The chapter will teach you a few "tricks," such as layering sprites and scrolling graphics at different speeds and scales to provide a "parallax" effect—giving the player an impression of forward movement. Simple movement will be replaced with the beginnings of a time-independent physics simulation, ensuring that your game runs the same on old or new computers. These are all useful techniques, so they'll be wrapped in library code and made available for all of your future games.

Chapter 8, Build Obstacles and Collision Detection, on page ? continues the physics theme. Instead of hard-coding collision logic, you'll build a generic collision detection system that allows you to create events whenever entities of different types overlap, allowing you to expand Flappy Dragon's gameplay and easily provide collision detection in your games. Simply comparing each entity's position and "hit box" with every other entity can be time consuming; this chapter will explore different ways to optimize the process.

Changing gears a bit, *the (as yet) unwritten Chapter 10, Welcome to Mars Base One,* introduces a new game: Mars Base One. Mars Base One is a physics-based game in which you fly a spaceship around a colony on Mars, as you try to avoid crashing into walls and rescue colonists. You'll stress test your random number generator by procedurally generating a colony, guaranteeing different gameplay each time, and using "seeds" to allow you to use a fixed design while debugging.

In the (as yet) unwritten *Chapter 11, Build a Mining Outpost on Mars,* , you'll optimize both your library and Mars Base One. Level creation takes some time, so rather than "hitch" the game during level generation, you'll spin level-generation off to a thread and use synchronization primitives to start the game once the level is available. Rendering thousands of wall pieces is also slow, so you'll apply some optimization techniques to combine the walls into a single render mesh, rendering the level thousands of times faster.

the (as yet) unwritten *Chapter 12, Add Miners and Energy Shields,* will flesh out Mars Base One and teach you some more rendering tricks. You'll add colonists to rescue, particle effects to convey a greater sense of motion and activity, and a Heads-Up Display (HUD) to better convey progress and game status. Your ship will gain a shield, so you don't die so quickly. At the end of the chapter, Mars Base One is a complete game.

In the (as yet) unwritten *Chapter 13, Build a High Score Server,* , you'll integrate your game with the Internet. You'll build a high-score server, accepting scores from people who play your game, and providing a list of high scores to display on the main menu. In the process, you'll build full round-trip serialization and deserialization, work with TCP sockets and safely transmit and receive data from other computers. You'll also practice some safe error handling techniques; the server isn't always available, and random people might send you malicious data.

the (as yet) unwritten *Chapter 14, Share Your Library,* focuses on sharing your library with others. Learn about licensing, publishing your library—or sharing it via Github—and gain a good idea of what's required to maintain an open source library.

What's not in this Book?

This book does not include the basics of getting started with Rust. If you need to start at the beginning, *Hands-on Rust* is a great start. This book also doesn't contain very advanced Rust—you won't learn the intricacies of directly managing memory, transmuting types, or working with cutting-edge features such as Generally Associated Types (GATs).

If you're an experienced Rust developer and want to learn more about structuring games or using the Bevy Engine, this book can still help you.

How to read this Book

If you're coming from *Hands-on Rust*, you'll want to read this book in order. If you're an experienced Rust developer, you'll probably want to skim some of the more introductory sections and focus on the areas you need to learn.

Tutorials are as much about the journey as the destination. Working through this book should give you ideas and inspire you to create other games and programs. Building generic library toolkits is a great way to accelerate your development; whenever you have a new idea, you no longer need to spend a lot of time writing boilerplate, you can reuse your previous work without resorting to copy and paste.

Conventions Used in this Book

The code accompanying this book is wrapped in a Rust *workspace*. Workspaces allow you to combine several projects into a single code-base, and when you compile an example, the resulting build reuses dependencies. Without a workspace, every project would require its own copy of Bevy and all of its dependencies, which can lead to using a huge amount of disk space.

The code is divided into directories as follows:

```
root
   /example_name (e.g. FirstLibraryCreate)
      /src        --- the source code for this example
      /assets     --- game assets required for this example
      Cargo.toml  --- build information for this example
  /src            --- a simple root program that reminds you to run an
                      example, rather than the workspace as a whole.
  Cargo.toml --- the master workspace control, listing all projects.
```

You can run code examples by navigating to example_name and typing cargo run for programs, and cargo test for libraries.

Rust doesn't permit multiple projects of the same name to exist within a workspace. Because of this restriction, you'‘ll find that some projects have a slightly different name inside their Cargo.toml descriptions. As you work through building my_library, the example code will change the name from my_library to my_library_benchmark as you progress through each stage of creating my_library. You *don't* need to change the names in your source code unless you explicitly want to match the example code.

When a file was created in a previous part of the book and a later example makes changes, changes have been highlighted with an arrow. For example:

```
fn do_something() {
```

```
➤   add_this_function();
  }
```

When a file has changed and requires that you remove an entry, a comment is added:

```
  fn do_something() {
➤   // remove: add_this_function();
  }
```

Source Code for this Book

(To be updated with a link to the GitHub repo and PragProg source download)

Online Resources

Here are some online resources that can help you:

- *Rust By Example* provides a good, example-driven introduction to the Rust Language.[2]
- The Rust Programming Language supplies in-depth concepts and tutorials to learn the finer details of Rust. It is also available online.[3]
- *The Rust Standard Library* documentation provides detailed descriptions of everything found in Rust's std library. It's a great reference when you can't remember how something works.[4]
- *The Unofficial Bevy Cheat Book* is a great help when working the Bevy engine.[5]
- *Bevy - The Book* provides a lot of great documentation for the Bevy engine.[6]

Wrapping Up

Whether you've just finished *Hands-on Rust* and want to continue your journey, or have mastered some of the basics of the Rust programming language and are ready to move onto more advanced concepts, this book can help. It's exciting to build a new game and see your hard work come to fruition. It's even more exciting when you have created a toolkit to help you avoid repeating yourself in every new project. Let's get started by setting up Rust and Bevy.

2. https://doc.rust-lang.org/rust-by-example/
3. https://doc.rust-lang.org/book/
4. https://doc.rust-lang.org/std/index.html
5. https://bevy-cheatbook.github.io/
6. https://bevyengine.org/learn/book/introduction/