Build Reusable Game State Management

In *Hands-on Rust*, readers built and used the *Flappy Dragon* project as an introductory example to Rust game development. Flappy Dragon is similar to the classic—and popular—game, Flappy Bird. Instead of guiding a bird, players take on the role of a majestic dragon. Despite the character change, the primary objects remain the same: avoid obstacles and don’t hit the ground.

Because the game logic and player controls are relatively simple, Flappy-like games are sometimes referred to as the “Hello, World” of game development. These types of games are not only fun to build but also extensible: you can add menus, game over screens, track scores, increase difficulty, and best of all, you can add your own visual flair.

Over the next four chapters, you’ll build a new, graphical version of Flappy Dragon using both the Bevy engine and the library you built in *Part II, Your First Library*, on page ?. The initial setup is very similar to the example from part one—so let’s start with a premade base.

**Setting up Flappy Dragon**

You can find the initial implementation of Flappy Dragon in the code/FlappyIntro/flappy_dragon_base/ directory of the accompanying source code. You’ll also need my_library, in the state it was in at the end of part one of this book. You can find it in the code/FirstLibraryDocs/my_library/ directory.

To get started, you’ll need to create a new project for Flappy Dragon. Select a base directory and change directory to it. Then, create a new project with Cargo:

```bash
$ cd (path)
$ cargo new flappy_dragon
```

Next, copy the source code into the new project structure:
1. Copy flappy_dragon_base (the directory, not just the contents) into the
   (path)/flappy_dragon directory from code/FlappyIntro/flappy_dragon_base/.
2. Copy my_library (again, the directory) into the code/FirstLibraryDocs/my_library/
directory.

You should now have a directory structure that looks like this:

(path)
   flappy_dragon
       flappy_dragon_base
           assets
           src
           Cargo.toml
       my_library
           src
           Cargo.toml
   src
       Cargo.toml

Let’s add flappy_dragon_base and my_library to the top-level workspace. Open
(path)/flappy_dragon/Cargo.toml and add the following lines to the [workspace] section:

[workspace]
members = [
    "flappy_dragon_base",
    "my_library"
]

It’s always a good idea to warn users when they try to run the top-level
workspace. Open (path)/flappy_dragon/src/main.rs and change it to warn the user
that they are trying to run the top-level workspace:

```rust
fn main() {
    println!("Please run the flappy_dragon_base project instead.");
}
```

Finally, link the library to the project. Open (path)/flappy_dragon/Cargo.toml and
add the my_library and bevy to the [dependencies] section:

```
[package]
name = "flappy_dragon_base"
version = "0.1.0"
edition = "2021"

[dependencies]
bevy = "0.12"
my_library = { path = "../my_library" }
```

You can now run the project:

```bash
cd (path)/flappy_dragon/flappy_dragon_base
```
cargo run

The game should start, and look like this:

![Flappy Dragon - Bevy Edition](image)

### Paths in the Downloaded Source Code

The downloadable source code includes multiple versions of both `flappy_dragon_base` and `my_library`. The Rust workspace system doesn’t like it when projects have the same name inside a workspace, so the downloadable version changes project names in `Cargo.toml`. As you follow along with the book, update your project and keep the names the same, which will make it easier to follow along.

### Understanding the Flappy Dragon Code

This implementation does not include anything you haven’t already used in the book so far, so there’s not much to explain. However, there are some things to know before you get started with the updates.
Flappy Dragon uses two graphics, both located in the assets directory. The dragon graphic is freely available from OpenGameArt.¹ The wall graphic was quickly put together in The Gimp.²

Most games follow a similar overall game cycle that includes an optional loading screen, a main menu, playing the game, and finally a game over screen. When players reach the game over screen, they typically either exit the game or play again. You can visualize this game cycle as follows:

In this chapter, you’ll create the main menu and game over screens for Flappy Dragon and Pig. More specifically, you’ll build a reusable menu system suitable for inclusion in any game you create, as you’ll see when you add these items to Pig (you’ll do this near the end of the chapter). As you work through this chapter, you’ll learn more about generics, traits, and state management. You’ll also learn how to use macros to shorten complicated syntax.

**Understanding Bevy and Game States**

Before you create your state management system, it’s worth taking a moment to study how Bevy handles application state. At the top level, a Bevy game maintains a state variable. State is stored as a resource and can be accessed in systems. And Bevy’s scheduler uses the game’s state to determine what systems to run.

States operate in phases:

- When a state becomes active, any systems attached to its `OnEnter` phase runs.
- With every “tick,” the active state’s `Update` systems are executed.

¹. [https://opengameart.org/content/flappy-dragon-sprite-sheets](https://opengameart.org/content/flappy-dragon-sprite-sheets)
². [https://www.gimp.org/](https://www.gimp.org/)
When a state is deactivated, systems contained in the state’s `OnExit` run. You specify which systems run for each state in your app builder by adding `system sets`. We’ll model `GameState` as an enumeration—it isn’t provided by Bevy, you will have to make one to represent your game state. In this example, we’ve added a `PlayMyGame` state to illustrate how states can be connected to Bevy systems:

```rust
// System runs for all states
.add_systems(Update, System1)

// System runs when State becomes active
.add_systems(OnEnter(GameState::PlayMyGame), setup)

// System runs each tick State is active
.add_system(OnUpdate(GameState::PlayMyGame), run)

// System runs when State becomes inactive
.add_system(OnExit(GameState::PlayMyGame), exit)
```

As a rule of thumb, try to make states self-contained. A state should perform set up and tag entities as belonging to the state. When the state exits, it should clean up any entities tagged as belonging to that state. This ensures that states can operate independently, which is especially important in a library, since you—the author—have no idea what a library consumer might do in the states you didn’t design.

**Modeling Game State**

Let’s start by organizing the code in `my_library`.

Your building a framework to handle game flow, so create a module named `bevy_framework`. The framework will include multiple modules, so let’s put it inside a folder. Create a new directory named `my_library/src/bevy_framework`. In that folder, create an empty file named `mod.rs`. Now, include the file in `my_library` by opening `my_library/src/lib.rs` and adding the following code:

```rust
mod bevy_framework;
pub use bevy_framework::*;
```

`bevy_framework` is now part of `my_library`, and it exports any public members to library consumers.

<table>
<thead>
<tr>
<th>Engines vs. Frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td>A framework is a skeleton organization, while an engine actually does the work. Bevy is an engine: it takes over control of your</td>
</tr>
</tbody>
</table>
Engines vs. Frameworks

A framework is more forgiving, offering a suggested approach to a task.

Flappy Dragon is a simple game: you’re either playing the game or you’re not. Adding in menus, you can model the states as follows:

```rust
#[derive(Clone, Copy, PartialEq, Eq, Debug, Hash, Default, States)]
enum GamePhase {
    MainMenu,
    Flapping,
    GameOver,
}
```

Pig is a bit more complicated, with several states that activate while the game is running. Pig’s game phase enumeration will need expanding to also include the game menus:

```rust
#[derive(Clone, Copy, PartialEq, Eq, Debug, Hash, Default, States)]
enum GamePhase {
    MainMenu,
    Start,
    Player,
    Cpu,
    End,
    GameOver,
}
```

Adding the MainMenu and GameOver items adds a little complexity to your workflow—but don’t worry, you’re going to automate most of the task of using it in your library. Let’s start building a state management plugin for Bevy.

Layered States

Before version 0.11, Bevy supported nesting states inside other states, and states with parameters. The Bevy Engine team decided upon a simpler state system. It works fine—but you’ll have to add generic states into your state enumeration every time.