TECHNOLOGY IN ACTION™

Developing Games on the Raspberry Pi

App Programming with Lua and LÖVE

—

Seth Kenlon

Apress®

Developing Games on the Raspberry Pi

App Programming with Lua and LÖVE

Seth Kenlon

Apress®

Developing Games on the Raspberry Pi: App Programming with Lua and LÖVE

Seth Kenlon
Wellington, New Zealand

ISBN-13 (pbk): 978-1-4842-4169-1 ISBN-13 (electronic): 978-1-4842-4170-7
https://doi.org/10.1007/978-1-4842-4170-7

Library of Congress Control Number: 2018966138

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4169-1. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to all programmers who work tirelessly on free (as in "liberty") and open software, not the least of whom are the Lua devs.

Table of Contents

About the Author

Seth Kenlon is a teacher, artist, D&D dungeon master, free software and free culture advocate, and UNIX geek. He has worked in the visual effects (VFX) (*The Hobbit*, *Deadpool*, *Valerian*) and computing industries (IBM, Red Hat), often at the same time. He is one of the maintainers of a Slackware-based multimedia production project.

About the Technical Reviewer

Sai Yamanoor is an IoT (Internet of Things) applications engineer working for an industrial gases company in Buffalo, NY. His interests, deeply rooted in DIY and open-source hardware, include developing gadgets that aid behavior modification. He has published two books with his brother and in his spare time, he likes to contribute to open source projects. You can find his project portfolio at http://saiyamanoor.com.

CHAPTER 1

Getting Started with the Raspberry Pi

Welcome to the exciting world of the Raspberry Pi and the Lua programming language. Whether you're already a programmer looking to learn about Lua, or the proud but confused new owner of a Raspberry Pi looking for a fun project, or a budding freelancer looking to get into mobile app development, or just a curious computer user looking to learn more, this book is your gateway into an exciting new world of fun with software.

To get through this book, you'll use two main tools: Lua and the Raspberry Pi.

Note This book requires no previous experience with computers or programming. Everything you need to know, you can learn from this book and diligent practice.

Lua is a small, fast, modern programming language that can be used for everything from system maintenance to graphics and standalone games. It's a leading scripting language in the video game and visual effects industry, and it is used for front-end development in several popular game engines. Learning Lua is not only a great way to learn programming, it's a pathway into the software development industry.

S. Kenlon, *Developing Games on the Raspberry Pi*,
https://doi.org/10.1007/978-1-4842-4170-7_1

The Raspberry Pi is, of course, a groundbreaking computer roughly the size of a mobile phone. It costs just $35 USD. Against all odds, the non-profit Raspberry Pi Foundation competes with dominating mega-corporations by selling an educational product loaded with open source software to students, teachers, and hobbyists like you. It's a great, affordable way to learn programming, open source, and how computers really work.

You may have acquired a Raspberry Pi for any variety of reasons, but here are the reasons that it was a good choice, and why it's the platform that this book uses:

- The Pi uses the ARM architecture, as opposed to the x86 architecture made popular by AMD and Intel. Most mobile phones use ARM chips, and mobile technology is the fastest-growing market for games. You don't have to develop games on ARM to publish games for mobile, but if you believe that knowing technology starts with using that technology, then $35 for a mobile game dev kit is a smart investment.
- The Raspberry Pi runs Linux, a free version of UNIX. You might not know UNIX yet, but if you're heading into the tech industry, the more you know about it, the better. UNIX knowledge is invaluable because most of the Internet is run on it, and it's the basis for Android phones, Steam machines, the PlayStation 4, and most of the film and TV visual FX industry. Besides that, it's a lot of fun.
- When computers first came out, it was expected that they would be tools that people could use to bring their ideas to life. It didn't matter whether your idea was great or small, you could make a computer do what you wanted it to do.

- Computers today are largely struggling to meet that goal. While programming on a Mac or a Windows PC is common, access to the full OS is restricted, and it can be expensive to keep up with the latest releases. There shouldn't be a barrier into computing. Use this book, a Raspberry Pi, and your passion for creativity and discovery to prove that programming is still for everyone.
- You can learn Windows or you can learn macOS, and either way, you learn either Windows or macOS. If you learn Linux, however, you learn *computing*. There will always be differences in how different platforms work, but an open source system like Linux lets you gain familiarity with the low-level computational basics shared by all computers, whether desktop, laptop, or mobile.

Preparing Your Pi

Believe it or not, one of the strengths of the Raspberry Pi is that it is low power. If you develop on a low-powered computer, then you broaden your audience because not everyone has the latest and greatest gaming rig or mobile device. Indeed, developing on a Pi is perfect for targeting the mobile market, because the Pi shares a lot with the internal hardware of mobile phones.

In the same spirit of inclusiveness, you don't actually have to have a Raspberry Pi to follow along with this book. You can buy any System-on-a-Chip (SoC) device; common ones include the BeagleBone, Banana Pi, Odroid, and the Pine64.

This book is general enough to cover whatever SoC device you use and any Linux or UNIX operating system. Technically, you can even use a spare computer instead, although you'll need to install Linux on it first or boot from a Linux USB drive. The important thing is to get through this section and end up with some computing device loaded with a UNIX or Linux operating system.

The advantage of a genuine Raspberry Pi is that it is thoroughly documented. There are lots of tutorials on raspberrypi.org to help you through anything you don't understand, and there's little to no variation in what you see on a Pi compared to what you see in this book.

Depending on where you buy your Raspberry Pi, you might find that the OS (called either Raspbian or NOOBS) is included in the box. That's fine for normal use, but when programming, it's best to have access to the latest development libraries. Raspbian isn't known for providing the most recent software tools, so this book uses a Linux OS called Fedberry, derived from the popular Fedora distribution of Linux. You can either purchase a spare microSD card to use with this book, or use the microSD card that came with your Pi, as long as you accept that the contents of your card will be replaced with a different OS.

If you purchased a Raspberry Pi that didn't include the OS on an SD card, or if you purchased a different SoC device that doesn't come already set up, then you have a computer that doesn't know what to do when you turn it on. It needs an operating system, and it's a great learning experience for you to install one.

Installing Linux onto Your Pi with Etcher

To install an operating system on your Pi or SoC device, you need a microSD card and an OS image file. OS images are available from `fedberry.org/#download`. Use the Fedberry "minimal" image file.

Caution This process *erases* the card, so don't use one containing photos, videos, or other data that you care about.

There are many ways to get a disk image onto a microSD card. The following is the easy method, and it's the same whether you run Linux, macOS, or Windows on your personal computer.

1. If you have not already done so, download the Fedberry LXQT image from `https://github.com/fedberry/fedberry/releases`. This image provides a basic OS with a few extra applications. You will manually install a full development environment later.
2. On your personal computer, download and install the Etcher application from `www.balena.io/etcher/`. For both Etcher and Fedberry, you need a tool to unzip archives. If you run Linux on your personal computer, then you already have one; otherwise, download and install 7zip from `www.7-zip.org` for Windows or Keka from `www.keka.io` for macOS.
3. Put the microSD card into your computer. If your computer doesn't have an SD card slot, you must purchase a microSD card reader.
4. Once the OS image has downloaded and Etcher has been installed, launch the Etcher application.
5. In the Etcher window, select the Fedberry image file from where it is saved on your hard drive, probably in your `Downloads` directory (see Figure 1-1).

Figure 1-1. *Etcher in action*

6. Select the SD card as the destination.
7. And finally, click the Flash button.

You can now skip to the "Writing Your First Lua Script" section.

Preparing Your Off-Brand SoC

If you only have a SoC board that is not made by the Raspberry Pi Foundation, then the OS images for the Raspberry Pi probably won't work on your device. But you can still use this book!

Your first step is to visit the website of your device's manufacturer. They probably offer an OS for the device they produce, and since they are targeting their own device, the OS image is likely a prebuilt image to copy to your SD card using the Etcher application. This process is described in *Installing Linux onto your Pi with Etcher*.

If you cannot find an official image for your device, the next step is to do an Internet search for the name of your device plus a query such as "Linux image". It helps to know which chip your device is based upon,

too, since sometimes generic OS images target the chip rather than every possible brand name applied to a system built around that chip. Whether you have an Allwinner, armv6, armv7, Tegra, or something else entirely, there's a good chance that somewhere on the Internet, there are a few hardworking hackers supporting your device.

Finally, if all else fails, you can turn to the two most reliable OS providers in the modern world: Debian Linux and NetBSD. These groups justifiably pride themselves on providing an operating system that runs on nearly every device you can think to put an OS onto (and a few that you wouldn't).

Debian Linux is available from debian.org. Depending on your device, you may have to do a little research on `wiki.debian.org/InstallingDebianOn` to understand how an install is done, but the good news is that it's almost certainly possible.

NetBSD is available from `wiki.netbsd.org/ports/evbarm`. The install process for NetBSD is remarkably easy, but the setup afterward is considerably more complex, especially if you're not familiar with UNIX yet.

If this is the route you are taking, you should take a little extra time to set up your system and to get familiar with it before continuing this book. The instructions in this book are mostly universal, but instructions on installing software or configuring sound outputs and other details may differ depending on your device and operating system.

Using This Book Without a SoC Device

If you don't have and cannot get a Raspberry Pi or other SoC, then you can use a traditional computer to work through this book, even a very old one. You'll get all the same benefits as those using a Pi: you'll learn programming, you'll learn Linux, and you'll learn all about the software development process, but you will have to work a little harder to get set up.

To set up a computer to use with this book, install Fedora Linux from `spins.fedoraproject.org/en/lxqt/` so that your environment mirrors the one in this book.

Caution This *erases* all the content on your computer, so use a spare computer that doesn't contain data you care about.

It's out of the scope of this book, but there are many ways to run Linux on a computer, and technically, any of them are probably acceptable for this book. For instance, you can run Linux off of a USB drive or DVD using porteus.org, or you can run Linux in a virtual machine using virtualbox.org. Whatever you choose, you have to translate what is in this book for what you are using. In other words, it's easier to just get a Pi and follow along, but it's not strictly required.

If this is the route you are taking, you should take a little extra time to set up your system and to get familiar with it before continuing this book. The instructions in this book are mostly universal, but instructions on installing software or configuring sound outputs and other details may differ, depending on your device and operating system.

First Boot

Assuming that you have your Pi plugged into a monitor, keyboard, mouse, and Ethernet, you can finally boot into your fresh, new Linux operating system. The first time you boot, you are asked to configure your system.

1. Configure your network to connect wirelessly to the Internet. If you are connected to the Internet over an Ethernet cable, then you can skip this category.
2. Set your time zone. To have your Pi get the correct time and day from the Internet, enable NTP in the upper-right corner of the time zone screen.

3. Set the administrative password.
4. Create a user and set a user password. Set the user as an administrator (see Figure 1-2). Take note of your username and password. You will need them often!

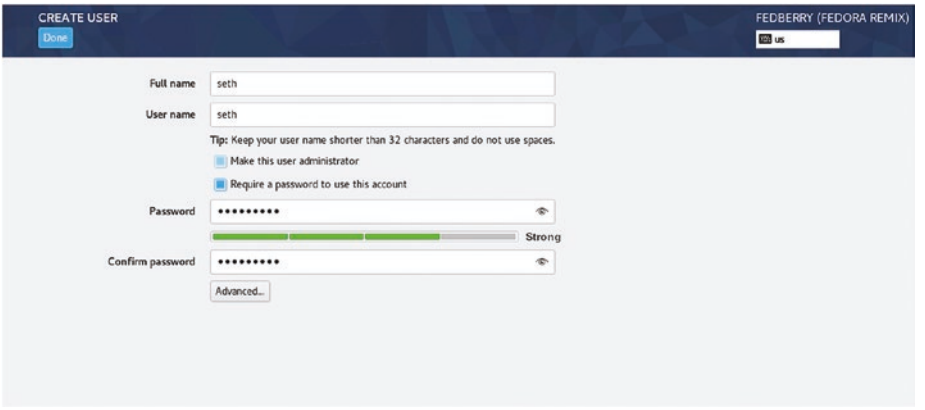

Figure 1-2. *Setting up FedBerry with the Anaconda installer*

5. Click the **Finish Configuration** button in the bottom right of the main screen to continue booting.

Note From this point on, the term *Pi* is meant to encompass whatever device you are using to follow along with this book.

When Fedberry has booted, you are left at the login screen. You'll log in to the desktop soon, but you got this book so that you could learn to code, so it's worth looking behind the scenes. Press Ctrl+Alt+F3 to switch to a text login screen instead.

Writing Your First Lua Script

The modern computing public likes to think that computers have evolved into interactive virtual worlds, but programmers know the truth: computers—whether it's a server, a desktop, or a mobile phone—are merely highly efficient calculators that get instructions in the form of *plain text*. When you switch a Linux computer to a text console, you're seeing the not-so-secret side of the operating system that responds to text commands. That's great if you're an experienced UNIX user, but it can be crippling if you don't know what commands exist, much less which commands to use. Throughout the course of this book, you'll get familiar with useful commands for Linux. Most commands you learn apply to any UNIX system, although some are particular to Fedora. Instead of listing common commands and expecting you to memorize them, however, this book uses and explains commands throughout so that you get familiar with them by using them.

First, you need to log in with a username and password. Use the username and user password that you created during setup. When you type in your password, it *appears* that nothing is happening; that's to be expected, just keep typing.

Once you're logged in, you are given a shell prompt that ends with a dollar sign ($). This means that your computer is ready for a command.

To program in Lua, you need to have Lua installed. In Linux, most of the "obvious" software that users need is stored in repositories of applications on remote servers. You can think of it as an app store (although it predates app stores by at least a decade).

Fedberry includes Lua by default, but this is a good exercise nevertheless, as it demonstrates how to confirm that an application you need is indeed installed.

The Fedora `dnf` command searches and installs software from Fedberry repositories. You will use this command a lot throughout this book, so you will become familiar with it, but for now just type this:

```
$ sudo dnf install lua
```

Enter your password when prompted. Remember, when you type your password, the cursor won't move.

Note If you installed a different OS onto your Pi, then the command is probably different. For example, on NetBSD the command is `pkg_add lua53`. Refer to the OS image's documentation for help.

For your first foray into Lua, you're going to program a simple dice-rolling game that pits the user against the computer to see who can roll the highest number on a 20-sided virtual die.

So far, you've been controlling your computer with a language called Bash. To switch to Lua, launch a Lua interpreter by typing

```
$ lua
```

It may not look that different, but you probably notice that your shell prompt has changed from a $ to a > symbol. Not all programming languages have an interactive prompt like this, but it's a good way to get to know a language before embarking on a big project with it.

Programming languages have lots of built-in functions that you can use. These functions are called *methods* or, unsurprisingly, *functions*. They are organized into libraries.

For instance, the `print()` function in Lua's basic library prints text to the screen. Try this:

```
> print("hello world")
hello world
```

You can also have Lua print numbers.

```
> print(23)
23
> print(21+(378/18))
42.0
```

Or both.

```
> print("The answer is "..21+(378/18))
The answer is 42.0
```

Rolling Virtual Dice

For your first program, you need random numbers so that you can mimic a die roll competition. Computers are producing numbers all the time, but how do you access those numbers? Can you think of something within a computer's normal routine that would produce numbers? If you can't think of anything, try looking up from your screen at the room around you. Is there anything in your physical space that could provide a more or less random number at a glance?

After some thought, you might realize that computers usually keep track of time, just like a clock in the real world does. It's not perfect, but it's a reliable source of numbers.

Lua has many libraries filled with specialized functions. The `os` library contains the `time` function, which returns the current time, in seconds, since 1 January 1970 (the UNIX Epoch). That's a lot of numbers, especially in the context of a dice game where you only need up to 20. Setting that aside for now, try using the `os.time()` function yourself.

```
> os.time()
1524967695
```

When you use a function, you are "calling" it. The empty parentheses at the end of the `os.time()` function call allows you to send information to the function when calling it. The `os.time()` function doesn't require any information from you to do its job, so the parentheses are left empty. Functions like `print()`, and other advanced functions that you will use later (some of which you yourself will write), require more information.

There are a few problems with using `os.time()`as a stand-in for a die roll. The `os.time()` function returns a very large number, and it's not very random.

There are a few ways to take a large number and reduce it to something within a given range. One easy way uses grade school math: take any number and divide it by your maximum desired value, and use the remainder (the "modulo" in computer terminology) as your result. For instance, if you have the number 103 and divide it by 20, you get 5 with a modulo of 3. In computer science, the % sign is used to do division and return only the modulo. Try it in Lua.

```
> 103%20
3
> os.time()%20
6
> os.time()%20
12
```

The modulo of `os.time()` has some degree of variance, depending on the time at which you call it. This introduces a perception of randomness. You can test this by trying to predict what your "roll" will be just before calling `os.time()`. It's pretty difficult to predict.

Note Press the up arrow on your keyboard to recall the previous Lua function call without all the typing.

After trying to predict your roll 20 or 30 times, do you see any problems or patterns in the `os.time()` solution?

You might notice that making the same call to `os.time()` in rapid succession betrays its very predictable pattern of incrementing steadily once per second.

Using Variables and User Input

Computers are programmed. They don't exactly produce random events, because they only do exactly as they have been programmed. Yet few computers are dormant; usually, they have been programmed to interact and respond to human input. There's nothing quite as unpredictable as the human mind, so why not use it to introduce some randomness to the dice roll?

It's too obvious to just ask the human player for a random number, especially if they know the goal of the game. If you know the goal of a game is to roll 20 on a virtual 20-sided die, then any good gamer is going to "randomly" choose 20 the majority of the time. So instead, you can ask your human player for some input and then use that input as a seed of randomness.

Ignoring that this is happening on a computer in a programming language you don't know yet, try to think of some ways you could trick a player into providing you with a random value.

Here are some ideas:

- Ask the user to provide a three-digit number and add it to `os.time()` as an offset.
- Ask the user for two numbers. Use the difference between the two numbers as an offset.
- Ask the user for the name of an animal or a color. Count the number of letters in the answer and use that number as an offset.
- Ask for two numbers, divide their sum by 20, and use the modulo as the offset.

You can probably imagine even more ways, but to implement any of them, you need to know how to get input from your user. As you might guess, getting input from a user is a common task in programming, so Lua has a function for that as a part of its input/output library, called `io`. The problem is that Lua doesn't inherently know what to do with input. Watch what happens if you use the `read` function, and then type `hello world` as input.

```
> io.read()
hello world
hello world
```

Lua just repeats what you give it. That's not very useful, and that's exactly why variables were invented. A variable is like an empty box, and you can put anything into the box that you need to store for later. You can put a word (or *string* in programming lingo), a number, or even an image or sound effect. Variables are surprisingly easy to set and easy to use once you need them.

```
> seed=io.read()
103
> seed%20
3.0
```

A new variable, in this example called `seed`, is created because you use the = after a word that Lua otherwise does not recognize. Whatever `io.read` gets from the user is placed into the variable you created. From then on, you can call the variable just as you call functions, and use whatever is inside.

Using variables, you can create interactive applications. Write a dice-rolling application based on your new understanding of variables. Of course, since you're running Lua as an interactive session, your program gets written and runs all at the same time, but that's enough for a proof of concept.

Here is a version of a simple dice rolling game:

```
> computer=os.time()
> seed=io.read()
104
> player=os.time()+seed
> print("The computer rolled "..computer%20)
The computer rolled 6
> print("You rolled "..player%20)
You rolled 18.0
```

You have written your first fully functional program! It's not a fancy game, and the only way to play it is to type it manually into a Lua prompt, but the logic and the results are sound. In the next chapter, you will create a Lua script file so that a more advanced version of this simple dice game can be run like a normal application. In the meantime, practice creating and using variables, and try to come up with alternative random number engines.

When you're ready to leave the Lua prompt, call the Lua exit function.

```
> os.exit()
```

When you see a $ prompt again, you're back at your Bash shell.

To power off your Pi for the day, use the `poweroff` command.

```
$ sudo poweroff
```

Homework

I may as well admit to you that Lua actually already has a random number function as part of its `math` library. Like your own versions of random number generation, it too requires a seed, but it uses a lot of math tricks and entropy to generate a number within whatever range you specify. Here's how it works:

```
> math.randomseed(os.time())
> math.random(1,20)
6
> math.random(1,20)
11
> math.random(1,20)
1
> math.random(1,20)
17
```

How did I find out that Lua had a random number function? How can you find out what other features Lua has that I haven't told you about? The answer to both questions is documentation.

Any good programming language is fully documented so that programmers know what the language can do. You're a programmer now, so you should browse through Lua's reference manual, available at `lua.org/manual/5.3/#index`. Much of it won't make sense to you yet, and there are several conventions of code documentation that can be confusing, but knowing where to find the functions available to you is a hugely important part of learning to code.