

How to Make an RPG

Daniel Schuller

© 2017 Daniel Schuller

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein maybe reproduced, transmitted, stored or used in any form or by any means graphic, electronic or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, except as permitted under Section 107 or 108 of 1976 United States Copyright Act, without the prior written permission of the author.

Dungeons & Dragons is a registered trademark of Wizards of the Coast, Inc.

FINAL FANTASY is a registered trademark of Square Enix Co., Ltd.

Tiled is a copyright of Thorbjørn Lindeijer.

All other trademarks are the property of their respective owners.

All images ©Daniel Schuller unless otherwise noted.

Visit the book website at **howtomakeanrpg.com**.

Version 1.3, 8th April 2017

Acknowledgements

Many people helped make this book a reality and I'm grateful to them all.

Thank you to **Michael LaRocca** for editing the book. You can find out more about him at his site <http://michaeldits.com/>.

Thanks to the artists for their time, patience and excellent artwork.

Bertrand Dupuy "DaLk" created the tilesets for the arena as well as the enemy and player character artwork. You can see more of this work by following him on twitter @DaLk_Alts.

Maciej Kuczynski "DE" created the tilesets for the Dungeon game and the final Quest game. He also created the combat backgrounds and special effects. You can find more of his work at <http://pixeljoint.com/p/925.htm>.

Early Readers

Thanks to all the people who have been patient enough to deal with the early versions of this book and offer corrections.

Thanks to Aaryn Bryanton, Adam Howels, Allen Mills, Brant Saunders, Brian Lyons, Patrick Clapp, Paul David, Wiktor Kwapisiewicz and XuWei Li, for detailed feedback and corrections.

About the Author

Daniel Schuller is a British-born computer game developer who has worked and lived in the United States, Singapore, Japan, the United Kingdom and is currently based in Hong Kong. Schuller has developed games for Sony, Ubisoft, Naughty Dog, Redbull Wizards of the Coast, as well as briefly running his own company; Bigyama.

Schuller maintains a game development website at <http://godpatterns.com>. In addition to developing computer games, Schuller also studies Japanese and is interested in Artificial Intelligence and the use of games in education.

Contents

1 Introduction	12
Origins	13
How to Read	16
RPG Architecture	17
How Games Work	17
Game State	18
Tools	19
Lua	20
Dinodeck	20
Common Code	24
Map Editing	25
2 Exploration	26
Tilemaps	26
Let's Write Some Code	27
Basic Map Format	35
Faster Rendering with a Texture Atlas	41
Implementing a Texture Atlas	43
Rendering the Tiled Lua Map Format	49
Rendering a Tiled Map	54
A Tree Falls in the Forest and No One Is Around to Hear It	60
A 2D Camera	68
Review	72

From Maps to Worlds	72
Enter the Hero	72
Simple Collision Detection	96
Layers	103
Interaction Zones	115
Trigger	118
Next Steps	126
A Living World	126
Character Class	126
A New Controller	132
Adding NPCs to the Map	138
Data Driven Triggers	151
User Interface	155
Panel	155
Textboxes	168
Textbox Transitions	173
A Fixed Textbox	176
Choices	199
Fitted Textbox	215
Progress Bar	220
Scrollbar	227
Stacking Everything Together	234
Menus	238
Game Flow	239
Stacking States	241
An Abstraction to Help Make Menus	259
The Default State of the In-Game Menu	269
Creating the World	275
The World	278
The Item Menu	286
An Exploration Game	298

The Introduction Storyboard	298
The Storyboard Render Stack	309
Sound Events	324
Map Events	328
Scene Operation	330
NPCs In Cutscenes	332
NPC Movement	340
NPC Speech	346
Storyboard Namespace	355
Action	356
Intrigue Cutscene	381
The Main Menu	394
Fix Up	398
Conclusion	402

3 Combat	403
Stats	403
What is a Stat?	403
Derived stats	405
Why Use Stats At All?	405
Which Type of Stats Should a Game Have?	406
Implementing Stats	406
Modifiers	408
Equipment Stats	415
A Stat Set Function	417
Moving On	417
Levels	417
What is a Level?	417
How Levels Work	418
Why Use Levels	420
Level Formulae	421

Creating a Levelling Function	425
How Levels Affect Stats	425
Dice	425
Stat Growth	431
Level-Up Test Program	435
Next Steps	437
The Party	437
Implementing the Party	438
Party Member Definitions	438
The Party Class	442
The Actor Class	443
Status Screen Preparation	446
Hooking Up the Status Screen	460
Status Menu State	460
Joining the Party	469
Closing	480
Equipment	480
Defining Equipment	480
Equipment Icons	482
Designing New Equipment	484
Equip and Unequip functions	488
Comparing Equipment	490
Equipment Menu	494
Finding Equipment	519
Combat Flow	528
An Event Queue	529
Defining Some Events	538
CombatScene	543
Testing Combat	546
Combat State	548
Getting Ready for Combat	548

Combat Layout Definitions	554
The First Enemy	562
Combat States	564
Creating the Combat States	566
The Standby State	569
Registering the CombatStates	572
Screen Layout	575
Combat Actions	585
EventQueue	585
CETurn	589
CombatChoiceState	591
CombatTargetState	600
Attack Event	610
CSAttack	611
Storyboards for Combat Events	614
The Attack Event	615
Death	618
Reacting to Damage	625
Combat Effects	630
Enemies Fighting Back	639
Combat Formula	640
The EnemyAttack	642
Winners, Losers and Loot	646
Game Over	647
GameOverState Reborn	647
Triggering Game Over	651
Winning and the Combat Summary	654
XPSummaryState	656
LootSummaryState	677
Putting it all Together	685
Loot Tables	689

Oddment Table to the Rescue	690
Loot Drops	693
Advanced Combat	699
Melee Combat	699
Implementing the Combat Formula	704
A Text Animation Effect	708
Updating CombatState and CEAttack	711
Fleeing	716
Items	727
Magic	753
Special Moves	770
The Arena	787
A Slimmer Main File	787
Sensible Stats	788
Locking Abilities	792
The Arena Game Loop	795
Making the Arena Dangerous	809
4 Quests	815
Planning	815
Plot and Flow	815
Asset List	817
The Town	818
World Map	820
Cave	821
Return to the Town	821
Wiring Everything Up	822
The Town	822
The Setup	822
Teleport Triggers	824
Basic NPCs	827

Shops	833
Implementing the Shop State	839
The Inn and NPCs	864
The Major's Cutscene	870
The Big Wide World	874
The World Map	879
Random Encounters	879
Cave Trigger	892
The Cave Map	893
Adding Cave Monsters	894
Boss Fight	898
Door Puzzle	909
Random Encounter Setup	917
Game State	921
Tracking Game State	923
Persisting Puzzles	928
Persistent Chests	931
Saving and Loading	932
Save and Loading Lua Data	946
Save and Load From The Menu	948
Finale	960
The Showdown	960
Final Game Over	966
Closing	970
5 Appendix	971
Appendix A - Classes in Lua	971
Appendix B - State Machines	973
State Machine Uses	977
StateStack	978
Appendix C - Tweening	978

Chapter 1

Introduction

It's been over 30 years since the release of Dragon Quest, one of the first, popular, Japanese-style computer role-playing games. Since then RPGs¹ have gone from a niche genre to mass market. From mobile, to web, to massively multiplayer; nearly all games use mechanics originally from role-playing games.

RPGs excel at telling story, character development, giving the player a feeling of progression and creating environments that are rewarding to explore. Learning how these games work and how to build one lets you not only create an awesome RPG experience but helps you when designing *any* game.

In this book we'll develop a classic JRPG², a little like Final Fantasy 6. We'll go from rendering a sprite to the screen, to a living breathing world; with towns, quests, combat and loot.

The book is broken into three main sections.

1. Exploration
2. Combat
3. Quests

In the first section we create the world for the player to explore, as well as the user interface to interact with the world. The second section introduces monsters, loot, stats, leveling and combat. In the final section we create a fully featured mini-RPG to demonstrate how a full game can be created.

I hope you're ready to enjoy the ride!

We're going to cover a little history of where JRPG games came from. I think it's quite interesting but if you're raring to start making a game, skip it; you can always check it out later!

¹RPG is the common abbreviation for "role-playing game." In this book RPG almost exclusively refers to "computer role-playing games" and commonly refers to "Japanese-style computer role-playing games."

²Japanese role-playing game or Japanese style role-playing game.

Origins

In the mid 1800s, before Germany even existed, is the Kingdom of Prussia. At this time, like most other times, Prussia is at war; Prussia *excels* at war. One reason it's so good at war is its war games. Ancient kingdoms may have used chess to teach the art of war, but not Prussia; Prussia has Kriegsspiel³. Kriegsspiel, like chess, is a game of miniatures and rules but it's much more elaborate; it's an attempt to simulate real warfare. These games would be quite familiar to a modern gamer. Counters represent units. They use dice, and the concept of game turns with time limits, and a games master. They use many of the features seen in modern tabletop games. Kriegsspiel games, like so many inventions of war, quickly found a way into civilian life.

In 1911 England, H.G. Wells is a household name; the father of science fiction, eugenics supporter, utopian visionary, socialist, and he's just about to kick off recreational miniature wargaming. By this time Wells has already written "The Time Machine," "The Invisible Man" and "The War of the Worlds" as well as countless other science fiction and nonfiction books, but in 1911 he wrote something a little different; a book called "Floor Games." Floor Games was an early rulebook for a miniature war game, not a game to train Prussian generals, but a game to entertain. He followed up Floor Games with a book called "Little Wars," marking the beginning of tabletop gaming. Many people published their own takes on these games, drawing on the grim realities of the World Wars as well as famous battles from antiquity.

Let's leave war games spreading and evolving toward the modern day and jump to Cross Plains, Texas. Each year on the second weekend of June, you can see men parading around town, clad in furs and holding swords to celebrate the life of Robert E. Howard. In the 1920s, Howard started the Sword and Sorcery genre with pulp titles such as Conan the Barbarian as well as more eldritch works written with the encouragement of H.P. Lovecraft, creator of the Cthulhu mythos. Howard's influences came from British mythology, particularly the Picts (an Iron Age culture that lived in Scotland). Unfortunately his life ended tragically in 1936, at 30 years old; he shot himself. Around the same time in Oxford, England, J.R.R. Tolkien was circulating a manuscript for his book "The Hobbit or, There and Back Again." The Hobbit was published the following year and the fantasy genre went mainstream. In 1954, Tolkien's "Lord of the Rings" trilogy was published, further cementing high fantasy in the popular consciousness and inspiring generations of readers.

Two American readers and war game enthusiasts, inspired by Howard and Tolkien, were Gary Gygax and Dave Arneson. In 1974 they created the tabletop roleplaying game "Dungeons and Dragons." Much like the Prussian Kriegsspiel, it had a game master, turns, dice and a rulebook. By 1981 there were more than three million players. Gygax loved reading pulp fantasy and science fiction, especially Howard's brand of sword and sorcery. Appendix N of the 1st edition of Dungeons and Dragons is titled "Inspirational and Educational Reading." It is half a page of books including Howard, Lovecraft, Tolkien and many other fantasy authors.

³Literally "war play."

Dungeons and Dragons was more narrative focused than the war games that came before it. Instead of simulating battles it focused on the adventures of a few characters directed by a Dungeon Master who devised each game's unique plot. It's hard to overstate the influence of Dungeons and Dragons on computer role-playing games; experience points, levels, numbers representing strength intelligence and speed, a party of players, spells, magic - these all come from D&D⁴. Dungeons and Dragons also included iconic monsters, often sourced from fantasy books, such as dragons, goblins, orcs, jellies, skeletons, wights and zombies, to name but a few.

Dungeons and Dragons introduced cooperative storytelling and world building but it did something more subtle too; it took a fantasy world and represented parts of it with numbers. Every person in the world was a certain level, with certain stats; this imaginary world wasn't just literary, it was *quantified*, grounded in numbers, math and probabilities. Actions and events were represented by rules and dice. This idea of a quantified fantasy world resonated very strongly with a certain type of person; programmers.

In the 1970s, computing systems were starting to become a common sight at universities. At the University of Illinois, one such system was called Plato. Plato was a precursor to many of the systems and concepts we take for granted today. It had a graphical user interface, email, forums, chat rooms and multiplayer games. Plato was not built with games in mind and therefore the first two dungeon crawl games are lost to history, deleted by system administrators trying to conserve Plato's limited memory. The third dungeon crawl game, written by Gary Whisenhunt and Ray Wood in 1975, did survive and it was simply called "dnd," It was one of the first computer RPGs. It was heavily influenced by Dungeons and Dragons but was a much simpler game; players battled their way through successive levels of a dungeon searching for two items: an orb and a grail. All the basic elements of modern computer RPGs were present; inventory, equipment, monsters (including "slimes"), a "boss" monster (in this case a Golden Dragon), treasure and treasure chests, experience points and levels, stats, a quest and even graphics. dnd was popular on Plato and helped influence many other Plato games including one called "Oubliette"⁵.

Oubliette was released in 1977 and introduced its own battery of innovations; multiplayer, a party of adventurers, choosing a race and class for your party members and even 3d wireframe graphics to depict the dungeon. Oubliette and dnd were groundbreaking games but they were on the Plato system and few people had access - just students and university staff. To really take off, role-playing games needed to move to more mainstream platforms and go commercial. Andrew Greenberg and Robert Woodhead, inspired by playing Oubliette, created a game that did this called Wizardry.

Wizardry ran on the Apple II, one of the first and most popular mass-produced computers. Wizardry was a dungeon crawler similar to Oubliette but with a better name and better platform. Released in 1981, it proved immensely popular and generated numerous sequels.

⁴D&D is a popular abbreviation for Dungeons & Dragons.

⁵Oubliette, literally "forgotten place," is a reference to a type of dungeon.

We're going to back up a bit now. As mentioned earlier, Plato was a wonderful system but not everyone had access to it and innovation happened in other places too. Nassau Bay, Texas, is where America's astronauts live, and not just astronauts but nearly everyone involved in the space industry. It's a pretty brainy place, a mix of scientists and adventurers. One of those astronauts is Owen K. Garriott who would travel into space in 1973. Thirty-five years later he was in the desert steppes of Kazakhstan, probably feeling quite nervous, as he watched his son, Richard Garriott, make his own journey into space.

When Richard was 13, a NASA doctor told him he'd never be an astronaut because of his poor eyesight, so he had to turn his mind to other endeavors. Six years later, Richard published his first game, Akalabeth. Akalabeth, like Wizardry, was developed for the Apple II and was a hit, selling over 30,000 copies. It was the first game to have a top-down overworld map with the view switching to first person on entering a dungeon.

Richard had been playing D&D with his friends for years, and he'd read the Lord of the Rings and The Chronicles of Narnia, all unknowing preparation to create one of the most important computer series ever, the Ultima Series. Ultima 1 was first released in 1981, and by the end of the series Richard was able to fund his own journey into space.

At the start of the 1980s, Wizardry and Ultima were the hottest games on the Apple II. Over in Japan, Koichi Nakamura and Yuri Horii played Ultima and Wizardry while working at Chunsoft. Chunsoft made games published by Enix. In 1986, together with Dragon Ball manga artist Akira Toriyama, they created one of the first console role-playing games, Dragon Quest. They wanted to make a game that was more accessible than Ultima or Wizardry. Dragon Quest took the overhead map of Ultima and combined it with the first-person combat and random battles of Wizardry. They stripped out classes, used a single character instead of a party, streamlined and simplified the inventory, and made the graphics more cartoonish and friendly. Dragon Quest is *the* most influential Japanese role-playing game. It created many of the conventions that persist today.

In the West, Dragon Quest didn't do well. It came over to North America toward the end of the NES⁶ era but failed to replicate the success it saw in Japan.

In 1987, a game that did manage to capture the attention of the West was released; Square's Final Fantasy. At the time, Square was a small studio and hadn't released many games. Final Fantasy was programmed by Nasir Gebelli, an American-educated Iranian, and was designed by Hironobu Sakaguchi, who named the game after his own personal situation; Sakaguchi had decided that if this game did not sell well he'd quit the industry and return to university. These days Sakaguchi lives in Hawaii and spends a lot of his time surfing, so it's safe to say things worked out.

When taking into account all the various versions, Final Fantasy 1 alone has sold over 2 million copies. Final Fantasy tried to steer away from the cutesy style of Dragon Quest toward something more realistic (as realistic as one can get in 16x16 pixels). In Sakaguchi's own words:

⁶Nintendo Entertainment System. The first popular home video game console.

"We made a concerted effort to be different. The graphics were part of that effort."

The basics of Final Fantasy were still very Dragon Quest - two scales, one for traveling on a overworld map and another for towns and dungeons; a battle mode; and menus for inventory and equipment. Final Fantasy gave the player control of a party of characters, like Wizardry, as opposed to the lone knight of Dragon Quest. The battle mode was a little different; instead of battling enemies head-on like Dragon Quest, it was side on - showing the player characters as well as the monsters.

From that time on, Dragon Quest and Final Fantasy have released sequel after sequel, and many other companies have released games in the same mold.

JRPGs have quite a storied history. From ancient board games like chess we got the wargames of the Prussian generals. Dungeons and Dragons came from these wargames mixed with the fiction of H.G. Wells, Howard, Tolkien and Lovecraft. As computers became more widespread, the games became digital, the most notable games being Wizardry and Ultima. These games spread to Japan where they were refined, simplified and mixed with a little manga art to form the console RPG as we know it today. As for the future? Well if you're reading this book, maybe that's in your hands!

How to Read

This book shows you how to build an RPG like Final Fantasy 6. The book is broken into three major parts:

1. Exploration - dealing with maps, triggers, animation, cutscenes, moving around the world and setting up the user interface.
2. Combat - dealing with stats, leveling, equipment, monsters and combat simulation.
3. Quests - making a full game, saving and loading.

Each part of the book has a set of step-by-step examples that are referred to in the text. You can find these examples at the same place you downloaded the book. The file names are:

1. example_1_explore.zip
2. example_2_combat.zip
3. example_3_quest.zip
4. example_intro.zip

Each zip file contains multiple example projects. You do not need to download any other program to run the examples. Run the file dinodeck_win.exe if you're on Windows, or

dinodeck_mac, if you're on Mac, in same directory as the example you want to test. The example_intro zip file contains examples for the introduction and appendices.

As you read the book, also read the code listings. Try out the examples projects for each part of the book. Typing out the code is a good way to get a feel for what's happening. Try to reason through the code to understand how it's working.

It's hard to accurately type out a full program without making mistakes! Don't be afraid to look at the example projects if you can't get something to work. Also feel free to read ahead, look at future examples and generally play with the code.

RPG Architecture

Japanese-style RPGs break down into separate systems very cleanly. Dinodeck takes care of the low-level actions, which means we don't have to worry about low-level systems like interfacing with the graphics or sound card, loading files from disk, rendering text and so on.

The systems we'll be build, roughly in order, are:

- Maps - loading and rendering
- Movement system and animations, collision detection
- Entities such as the player - animations and definition
- User Interface - menus, etc.
- Trigger system so players can interact with maps and other entities
- Cutscenes
- World State tracking - for quests, opened chests, stats, spells, item database, etc.
- Inventory System
- Saving and Loading
- Combat System

Some systems are dependent on others; you can't have a movement system until there's a map to move around on. You can't have a combat system without being able to track stats, display menus, have an inventory system and load entities.

How Games Work

Computers are fast. It's hard to imagine how fast they are. Even a computer like the NES can execute hundreds of thousands of instructions per second. For games, computers calculate all the information about the game and then draw that information to a framebuffer that's sent to the screen. Games running at 60 FPS do this 60 times in a second. This means in 0.01666 seconds a game checks and updates the following:

1. Player input
2. AI
3. Enemy position and animations
4. Particles
5. Sound effects and music
6. User interface and text

Then it renders the entire game world to the screen.

The majority of games are structured in the same way; they're built around an update loop. The update loop is a special function that the operating system calls as often as it can. The update function handles all the game logic and tells the graphics card what to draw on screen. The quicker your update loop executes, the faster your game.

Game State

In a typical JRPG you start in a small town, free to wander around and interact with the townspeople. Walking to the edge of the town lets you leave and you enter a different game mode; the world map.

These two types of maps can be thought of as *game states*.

The world map acts very much like the local map but at a larger scale; you can see mountains and towns, instead of trees and fences. While on the world map, if you walk back into the town the mode reverts to the local map.

In either the world map or local map you can bring up a menu to check out your characters, and sometimes on the world map (or in a dungeon) you'll be thrown into combat. Figure 1.1 describes these game modes and transitions; this is the basic flow of JRPG gameplay and is what we'll base our game states on.

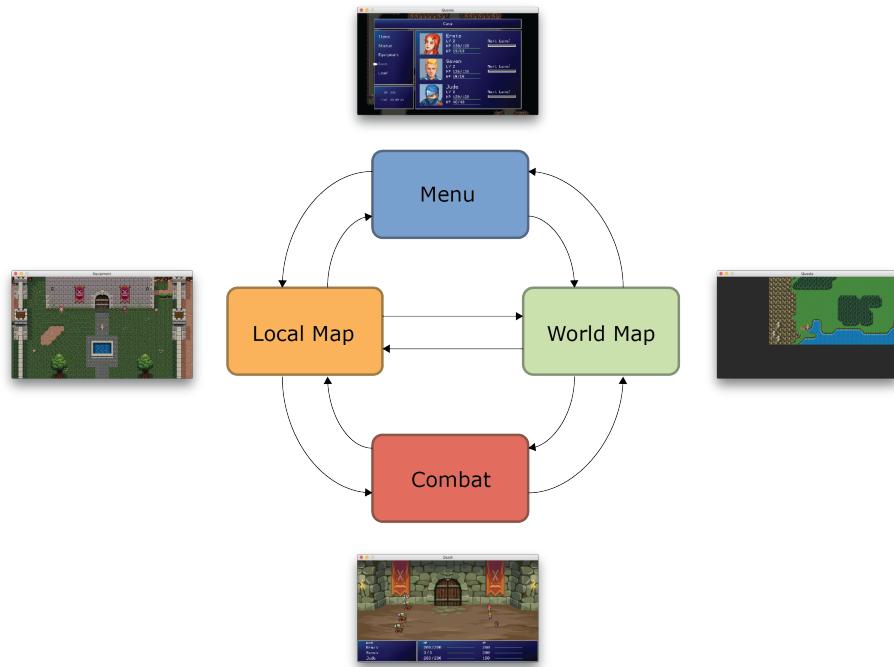


Figure 1.1: The general structure of a JRPG.

These states are, for the most part, independent of each other. Let's say we want to change the way the character moves, so it's more like *Zelda : A Link to the Past* than *Final Fantasy 6*. In this case we only need to edit the exploration state; the other states are unaffected. If we want to change the combat to something more real time and actiony, again we can just change one state.

Breaking a game into separate states silos off complex game systems and makes it easier to build and reason about the game as a whole.

Tools

All the code and examples in the book run on top of a bespoke engine called Dinodeck, written in C++. Games in Dinodeck are created using the Lua scripting language⁷. You don't need to download anything apart from a nice text editor; I recommend [Sublime Text](#).

Dinodeck is used because then you own the entire stack. There's no black-box third party engine or licensing costs; you're free to do whatever you want with the code. It's

⁷Dinodeck uses a Lua variant called LuajIT that's extremely fast.

also built around the idea of assisting developers in making games rapidly. The game code is quite general and can be ported to any modern engine.

The complete documentation for Dinodeck is available at <http://dinodeck.com/docs/1.0/> and you can get the full source code at <http://dinodeck.com/source/1.0/>.

Lua

Lua is an elegant language that's popular in the game industry. It's been used to script software from Grand Theft Auto to World of Warcraft and even to write viruses for international espionage⁸! It has a lot of breadth.

This book does not cover the Lua language, it's simple enough that if you have programming experience then you can pick it up as we go.

To get a better grounding in Lua check out these links:

- [Programming in Lua](#) - the official programming guide.
- [Quickstart Lua](#) - a quick start guide I wrote if you're already familiar with similar languages.
- [Lua Demo](#) - Lua interactive console where you can write programs online. I use this sometimes for testing small snippets of code.

In the book we use a simple class system built in Lua. For more details please see [Appendix A - Classes](#).

Dinodeck

The Dinodeck engine is one file; Dinodeck.exe on Windows or just Dinodeck on Mac. This exe file never changes; the files in the same directory change for each different game. To run a game, Dinodeck looks for a file called settings.lua. The settings file tells Dinodeck some basic information about how to run the game.

Listing 1.1 is a simple settings file.

```
name = "Hello World"
width = 740
height = 480
manifest = "manifest.lua"
main_script = "Main.lua"
on_update = "update()"
webserver = false
```

⁸The Flame computer virus was partially written in Lua. You can read more here:
<http://www.wired.com/2012/05/flame/>

Listing 1.1: A basic Dinodeck settings file.

The settings file is pure Lua code. Dinodeck checks for each of the following variables:

- **name** - The name of the game. The title of the window will be set to this.
- **width**⁹ - Width⁹ of the game render area in pixels.
- **height**¹⁰ - Height¹⁰ of the game render area in pixels.
- **manifest** - A list of all the assets the game uses and where to find them.
- **main_script** - Name of a script asset defined in the manifest file that contains the main update loop. This is the only Lua script that's run. It's responsible for loading all other scripts as needed.
- **on_update** - The name of the update function to call.
- **webserver** - Dinodeck can run a local webserver for debugging.

After Dinodeck runs the settings file, it parses the manifest.lua file to get the location of all the assets for the game. The manifest file can include different types of resources: scripts, sounds, textures, etc. Listing 1.2 shows a simple manifest file.

```
manifest =
{
    scripts =
    {
        ['Main.lua'] =
        {
            path = "main.lua"
        },
    },
    textures =
    {
        ['logo.png'] =
        {
            path = "logo.png",
        },
    }
}
```

Listing 1.2: A simple two-file manifest.

If you look at Listing 1.1 you'll see Dinodeck expects to run a "Main.lua" file, and every frame it will try to call "update()", a Lua function. This means Main.lua should define a function called update.

⁹The width and height represent the render area in pixels. These numbers don't include the window frame.

¹⁰The width and height represent the render area in pixels. These numbers don't include the window frame.

Dinodeck Game Loop

The main game loop for Dinodeck is specified in the `settings.lua` file but, generally, it's a global function call `update` in your `main.lua` file. The `update` function is called each frame. After the `update` function is called the screen is updated. This repeats as fast as the computer can manage.

Delta Time

Dinodeck has a global function called `GetDeltaTime` that returns the duration of the last frame in seconds. Usually this number is very small. For instance, in a 60 frames per second game the `GetDeltaTime` function will return 0.016666.

The delta time number is used mainly to get smooth animations, independent of how choppy a game's frame rate might be.

For instance, say you're moving a ball right on the screen and you know it's moving 5 meters per second. To work out how much to move it in a single frame, we first need to know how much time has passed since we last moved it. This number is the delta time and it tells us that some fraction of a second has passed. Then we can calculate the distance to move the ball during that time and move it by that amount. The calculation looks like this (if we assume 1 Dinodeck unit is 1 meter):

```
local dt = GetDeltaTime()
ball.position.x = ball.position.x + (5 * dt)
```

Space in Dinodeck

In Dinodeck, position 0, 0 is the middle of the screen. The top left of the screen is `-width/2, height/2` where the width and height are the pixel values defined in the `settings` file¹¹. Figure 1.2 shows the coordinates of the Dinodeck space.

¹¹You can get these values programmatically using the `System` library and the calls `System.ScreenWidth()` and `System.ScreenHeight()`.

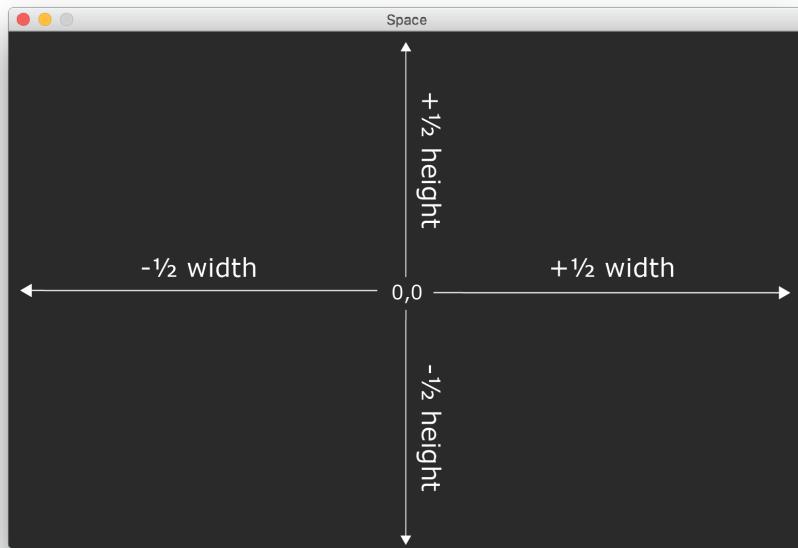


Figure 1.2: How Dinodeck defines its space.

Sprites are drawn from their center. This means if you create a sprite in Dinodeck at 0, 0 and draw it to the screen, it's centered on the center of the window.

Hello World

What's an introduction to an engine without a Hello World program? There's an example project containing the code, manifest and settings files in example/hello_world. Listing 1.3 shows the main.lua file in full.

```
LoadLibrary("Renderer")

gRenderer = Renderer>Create()
gRenderer:AlignText("center", "center")

function update()
    gRenderer:DrawText2d(0, 0, "Hello World")
end
```

Listing 1.3: Hello in Dinodeck.

Listing 1.3 begins by loading the Renderer library. The renderer library lets us create renderers and draw things to the screen. In the next line we create a renderer and store it as gRenderer. The lower case g is used to indicate this is a global variable. Next we tell the renderer to make all subsequent text calls be centered on the horizontal and vertical. This will make our text center nicely on screen.

Then after the setup we define our update function. This function is referenced in the settings.lua file and is called once per frame. In the update loop we tell the renderer to draw "Hello World" at position 0, 0. You can see the result in Figure 1.3.

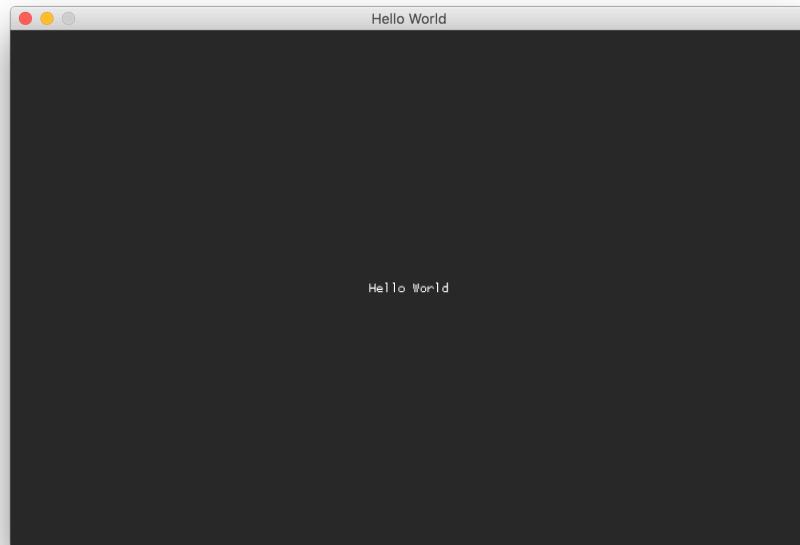


Figure 1.3: A Dinodeck Hello World app.

Feel free to play around with this code. Nothing bad can happen!

Common Code

Some code is fundamental and useful for all game projects. The two pieces of code we make a lot of use of in the book are State Machine class and Tween class. For more details on these pieces of code check out **Appendix B - State Machine** and **Appendix C - Tweening**.

Map Editing

To make the maps for the games in the book we use a free program called Tiled. You can download this software from <http://www.mapeditor.org/>. The book uses Tiled 0.9.1 (changes in versions are likely to be minor).

Chapter 2

Exploration

All world building starts with a map, so that's where we'll start on our RPG adventure.

Tilemaps

Computer role-playing games display their environments to the user in many different ways.

Most modern RPGs create and display the game world entirely in 3d. 3d worlds represent locations accurately and are easier to reuse and animate than 2d sprites.

Games such as Final Fantasy 7 and Baldur's Gate represent locations using a single large image. These images are far bigger than a single screen can display at one time. As the player moves around the environment, they see different parts of the image. The environments are created in 3d, then rendered out as a 2d image. Extra detail is added using a paint program, and then it's ready to use in the game.

Classic RPGs, like the one we're going to create, use *tilemaps* to represent the world.

Early PCs and consoles such as the SNES¹ had limited memory. Large images, like those of Final Fantasy 7, can't fit in such a small amount of memory. Therefore maps were made out of lots of smaller images called tiles. For example, the SNES game, The Legend of Zelda: A Link to the Past, has a display resolution of 256 by 224 pixels. To display a field of grass taking up the entire screen, you need one big 256 x 224 pixel image, but using a grass tile means you only need a single 16 x 16 pixel image drawn 244 times (the number of tiles required to totally fill the screen).

Using a tilemap not only saves memory, it makes it easier to produce maps and game content. Once you have a good palette of tiles for, say, a forest, then you can create many different forest maps quickly, using hardly any additional memory.

¹Super Nintendo Entertainment system. Successor to the NES.

Let's Write Some Code

To get a feel for how tilemaps work, there's nothing better than diving into some code. We'll start by drawing a single tile. Open project tilemap-1 in the examples folder. Here you'll find an image called grass_tile.png and a simple empty project. The project's manifest tells it to load the tile image when it runs.

Open main.lua and write as follows in Listing 2.1.

```
LoadLibrary("Renderer")
LoadLibrary("Sprite")
LoadLibrary("Texture")

gRenderer = Renderer>Create()

gTileSprite = Sprite>Create()
gTileSprite:SetTexture(Texture.Find("grass_tile.png"))

function update()
    gRenderer:DrawSprite(gTileSprite)
end
```

Listing 2.1: Drawing a tile sprite. In main.lua.

This code loads the libraries we need to draw graphics to the screen. Then it creates the renderer, creates a sprite, finds the image to use as a texture, and assigns it to the sprite with the SetTexture call. By default sprites are rendered at position 0, 0 which is the center of the screen and sprites are centered around their midpoint. If you run the code you'll see something similar to Figure 2.1.

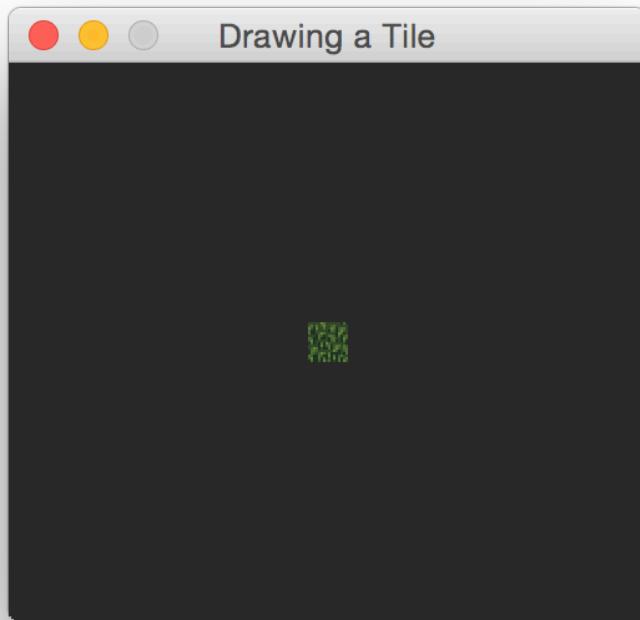


Figure 2.1: Rendering a single tile.

Figure 2.1 shows our grass tile sprite displayed in the center of the screen. There's a working project that shows this in the examples folder named tilemap-1-solution. Now that we have a single tile rendering, can you figure out how we'd fill the entire display?

If not, then let's go through it step by step! First we need the position where we want to draw the first tile. The top left corner seems a sensible place; we can get that by taking our $0, 0$ position, in the center of the display, and subtracting half the display width and half the display height. See Listing 2.2.

```
gLeft = -System.ScreenWidth() / 2  
gTop = System.ScreenHeight() / 2
```

Listing 2.2: Getting the top left screen position.

This code gives us the position of the pixel in the top left corner. You can see it visualized in Figure 2.2. The code that draws this pixel is available in example tilemap-2.

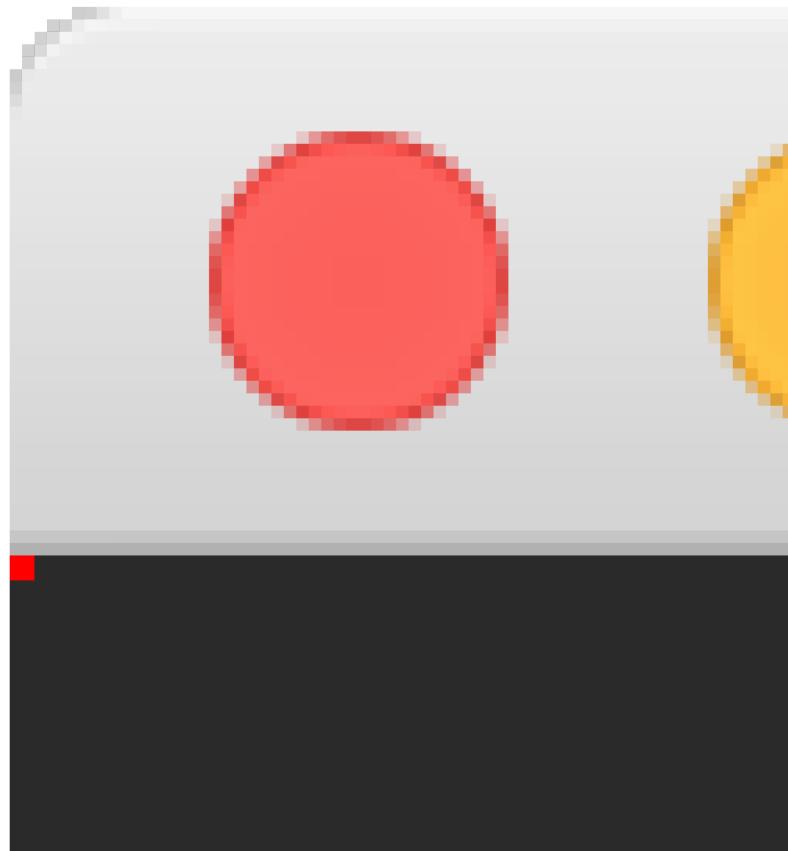


Figure 2.2: The top left pixel in the corner of the screen.

We have the top left position of the screen but tiles are center-aligned. If we position tile at the top left pixel we get a situation like Figure 2.3; the tile is mostly offscreen! (Code rendering the tile like this is available in the tilemap-3 example.)

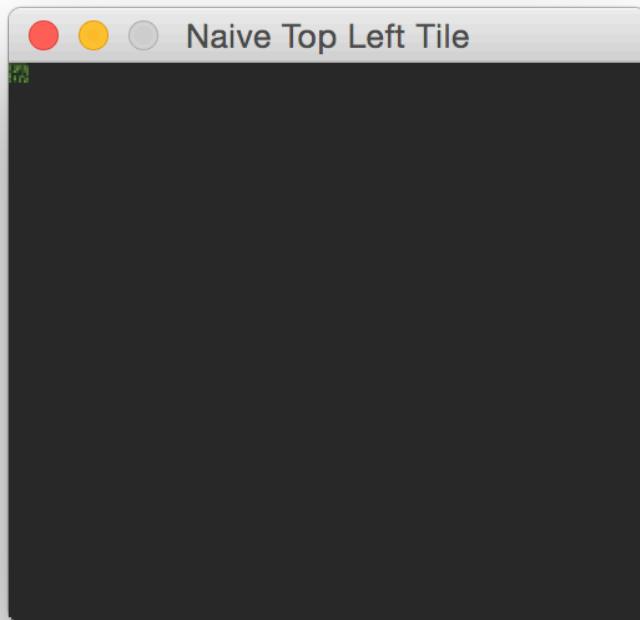


Figure 2.3: Rendering a tile centered on the top left pixel.

The tile is centered on the top left pixel so we need to shift it left and down by half the tile's width and half the tile's height. Try updating your main.lua code to Listing 2.3 and then check out where the tile renders. Example tilemap-4 shows the tile in this improved position.

```
LoadLibrary("Renderer")
LoadLibrary("Sprite")
LoadLibrary("System")
LoadLibrary("Texture")

gRenderer = Renderer.Create()
gLeft = -SystemScreenWidth() / 2
gTop = System.ScreenHeight() / 2
```

```
gTileSprite = Sprite.Create()

gGrassTexture = Texture.Find("grass_tile.png")
gTileWidth = gGrassTexture:GetWidth()
gTileHeight = gGrassTexture:GetHeight()

gTileSprite:SetTexture(gGrassTexture)
gTileSprite:SetPosition(gLeft + gTileWidth / 2, gTop - gTileHeight / 2)

function update()
    gRenderer:DrawSprite(gTileSprite)
end
```

Listing 2.3: Rendering tiles flush with the window.

In this code snippet we've included the `System` library to get the height and width of the window. We get the height and width of the tile by calling `GetWidth` and `GetHeight` on the texture. The result can be seen in Figure 2.4.

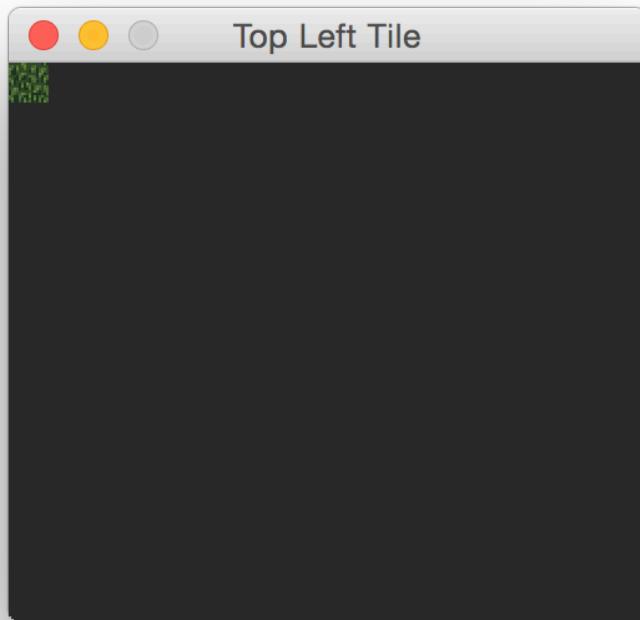


Figure 2.4: Run the code and you'll see the tile is flush with the top left corner of the window.

The tile is perfectly flush to the top left corner. Let's calculate how many tiles we need to draw in a row and how many in a column to fill the screen. We divide the display width and height by the tile width and height. If the result is a fraction we round up because if we only drew 12 of 12.36 tiles we'd have an ugly black space at the end of our display. The ugly black space would be the same size as that tile fraction! Rounding up in Lua is done using the `math.ceil` function.

Let's do the column and row calculations as in code Listing 2.4. Example tilemap-5 demonstrates the calculations.

```
gTexture = Texture.Find("grass_tile.png")
gTileWidth = gTexture:GetWidth()
gTileHeight = gTexture:GetHeight()
```

```

gDisplayWidth = System.ScreenWidth()
gDisplayHeight = System.ScreenHeight()

gTilesPerRow = math.ceil(gDisplayWidth/gTileWidth)
gTilesPerColumn = math.ceil(gDisplayHeight/gTileHeight)

```

Listing 2.4: Calculating rows and columns required to fill a window.

If the resolution of the display is 256x244 pixels, and the dimension of the tile is 16x16 pixels, then the number of tiles per row is $256 / 16 = 16$. The number of tiles per column is calculated the same way, $224 / 16 = 14$. Calculating this tells us how many tiles to draw to cover the entire display area.

Let's draw the first row using a loop, as in Listing 2.5. This code is available in example project tilemap-6.

```

gTop = gDisplayHeight / 2 - gTileHeight / 2
gLeft = -gDisplayWidth / 2 + gTileWidth / 2

function update()
    for i = 0, gTilesPerRow - 1 do
        gTileSprite:SetPosition(gLeft + i * gTileWidth, gTop)
        gRenderer:DrawSprite(gTileSprite)
    end
end

```

Listing 2.5: Drawing a row of tiles. In main.lua.

Run the code from Listing 2.5 and you'll see it draws a single row of grass tiles, covering the top of the screen. To make this happen we rendered the first tile at the top left corner, then for each successive tile we increased the X coordinate by one tile width, drawing one after another until we reached the right side of the window.

In the code, the for-loop starts with *i* equaling 0. This means the end point of the loop must be *gTilesPerRow - 1*; if it was not minus 1 then we'd end up drawing an extra tile off the right hand side of the screen. If *i* started being equal to 1 then we wouldn't need to subtract 1 from the *gTilesPerRow* but that would make the later math more complicated.

Now we've got a nice strip of grass. Let's add the columns and create a full field! We're going to add an extra for-loop as shown in Listing 2.6. Example tilemap-7 has the code so far.

```
function update()
    for j = 0, gTilesPerColumn - 1 do
        for i = 0, gTilesPerRow - 1 do

            gTileSprite:SetPosition(
                gLeft + i * gTileWidth,
                gTop - j * gTileHeight)

            gRenderer:DrawSprite(gTileSprite)
        end
    end
end
```

Listing 2.6: Drawing rows and columns of tiles. In main.lua.

And that's it! Running the code should get something like Figure 2.5. Our first RPG field to explore (or cultivate if all this grass has inspired you to make an RPG / farming simulator like Harvest Moon).

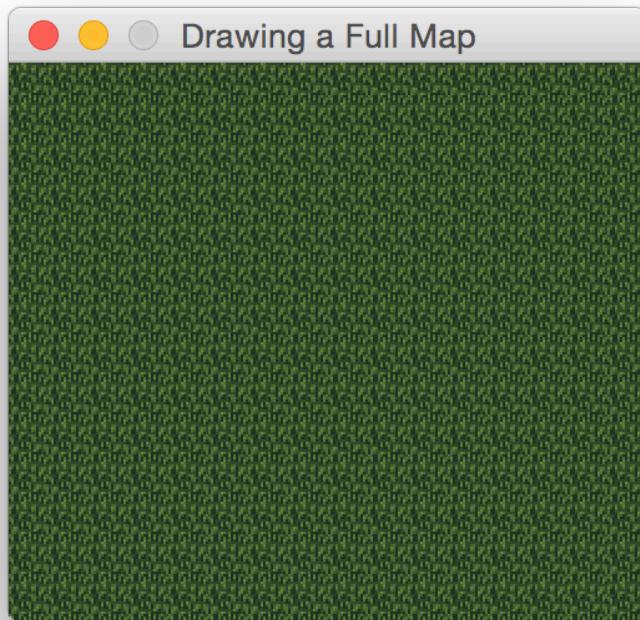


Figure 2.5: A full field of grass made from a single grass tile.

Basic Map Format

Our grass field is *pretty* awesome but most games need more than just plain after plain of grass. Open the tilemap-8 example. This is a basic empty project that has ten different tiles. Mainly grass and dirt. We'll use these to render a more interesting map. Take a look in manifest.lua and you'll see how the tile images are linked to the project; this will be useful for when you want to add your own tiles! We're going to extend this example project to use a "map definition" table to render the map.

First let's list all the tile textures in a table so we can easily reference them using a number, Listing 2.7.

```
gTextures =  
{  
    Texture.Find("tiles_00.png"),  
    Texture.Find("tiles_01.png"),  
    Texture.Find("tiles_02.png"),  
    Texture.Find("tiles_03.png"),  
    Texture.Find("tiles_04.png"),  
    Texture.Find("tiles_05.png"),  
    Texture.Find("tiles_06.png"),  
    Texture.Find("tiles_07.png"),  
    Texture.Find("tiles_08.png"),  
    Texture.Find("tiles_09.png"),  
    Texture.Find("tiles_10.png"),  
}
```

Listing 2.7: Adding tiles. In main.lua.

Here a plain grass texture is the first entry and can be accessed using the number 1 as an index. (Lua indices count up from 1 not 0 like many other programming languages. This can be a little confusing when starting with Lua.)

The grass texture, tiles_00.png, can be accessed by writing gTextures[1]. Each texture is mapped to a number. Numbers are easier to use for a map format because they're simpler data types than textures and easier to save and load from disk.

Maps are made up from tiles. Each tile has an X, Y coordinate to indicate its position on the map. A common arrangement is shown in Figure 2.6.

1, 1	2, 1	3, 1	4, 1	5, 1	6, 1	7, 1	8, 1
1, 2	2, 2	3, 2	4, 2	5, 2	6, 2	7, 2	8, 2
1, 3	2, 3	3, 3	4, 3	5, 3	6, 3	7, 3	8, 3
1, 4	2, 4	3, 4	4, 4	5, 4	6, 4	7, 4	8, 4
1, 5	2, 5	3, 5	4, 5	5, 5	6, 5	7, 5	8, 5
1, 6	2, 6	3, 6	4, 6	5, 6	6, 6	7, 6	8, 6
1, 7	2, 7	3, 7	4, 7	5, 7	6, 7	7, 7	8, 7

Figure 2.6: A map of tiles, with the X, Y coordinates for each tile.

Using a coordinate system like Figure 2.6 means if we're at tile 8, 6 and want to access the tile directly above, we just need to subtract one from the y coordinate. Getting a neighboring tile is a common operation in games using a tilemap. For instance, if the player is moving up you can quickly check if the tile above is a wall or not and if the player is allowed to move there.

Let's make a Lua table that describes a map layout. We'll refer to this kind of table as a *map definition* table. A first attempt at a map format might look like Listing 2.8 below.

```
-- Map is 8 x 7 to fit the display exactly
gMap =
{
    {1,1,1,1,1,1,1}, -- 1
    {1,1,1,1,1,1,1}, -- 2
    {1,1,1,1,1,1,1}, -- 3
    {1,1,1,1,1,1,1}, -- 4
```

```

{1,1,1,1,1,1,1},      -- 5
{1,1,1,1,1,1,1},      -- 6
{1,1,1,1,1,1,1},      -- 7
}
local topLeftTile = gMap[1][1]
local bottomRightTile = gMap[7][8]

```

Listing 2.8: A table describing a tilemap. In main.lua.

In Listing 2.8, the table `gMap` contains a table entry for each row of the map. The number of rows is equal to the map's height. Each row table contains a tile for each column in the map. All row tables are the same length and the length is equal to the map's width. The tile entries are all equal to 1 which references the plain grass tile.

This table structure has the nice advantage that it can be accessed by using `gMap[x][y]` which is simple to use when coding. But when it comes to access speed and storing it in memory, recording the map's width and height and having one long array of tiles is more efficient. This is shown in Listing 2.9.

This setup is more efficient because the map is one continuous block of memory, making it quick to access and modify. If the map is a table of tables then the memory might be scattered around and the computer will need to find the memory for each row before modifying it.

```

-- 1 indexes the plain grass tile.
gMap =
{
    1,1,1,1,1,1,1,      -- 1
    1,1,1,1,1,1,1,      -- 2
    1,1,1,1,1,1,1,      -- 3
    1,1,1,1,1,1,1,      -- 4
    1,1,1,1,1,1,1,      -- 5
    1,1,1,1,1,1,1,      -- 6
    1,1,1,1,1,1,1,      -- 7
}
gMapWidth = 8
gMapHeight = 7

local topLeftTile = gMap[1]
local bottomRightTile = gMap[gMapWidth*gMapHeight] -- 56

```

Listing 2.9: A more efficient tilemap format. In main.lua.

Now that our tiles are in one continuous array, they're harder to access using x, y coordinates. Let's add a function to make things simple again.

The function is shown in Listing 2.10 and uses X, Y coordinates starting from 0, 0 instead of 1, 1. This makes the math simpler. Starting from 0, 0 means the bottom right corner tile is now `gMapWidth - 1, gMapHeight - 1`.

```
function GetTile(map, rowsize, x, y)
    x = x + 1 -- change from 1 -> rowsize
              -- to          0 -> rowsize - 1
    return map[x + y * rowsize]
end

local topLeftTile = GetTile(gMap, gMapWidth, 0, 0)
local bottomRightTile = GetTile(gMap, gMapWidth,
                                gMapWidth - 1, gMapHeight - 1)
```

Listing 2.10: GetTile function to get a tile from the more efficient map format. In main.lua.

In Listing 2.10 each row has `rowSize` number of tiles in it; the width of the map. The `y` coordinate is multiplied by the map width to get us to the correct row, then the `x` coordinate is added to this to get us to the correct tile. If you don't get it immediately, try working through the code with a pen and paper, and it should all fall into place!

Time to build a simple map. We're going to use different tiles and add a dirt path to our map! See Listing 2.11.

```
gMap =
{
    1,1,1,1,5,6,7,1,    -- 1
    1,1,1,1,5,6,7,1,    -- 2
    1,1,1,1,5,6,7,1,    -- 3
    3,3,3,3,11,6,7,1,   -- 4
    9,9,9,9,9,9,10,1,   -- 5
    1,1,1,1,1,1,1,1,    -- 6
    1,1,1,1,1,1,2,3,    -- 7
}
gMapWidth = 8
gMapHeight = 7
```

Listing 2.11: A more interesting tilemap. In main.lua.

We already have a render loop to display a new map but we need to make a few modifications. We need to get the current tile coordinate, look up the tile in our map, and set the sprite to the corresponding texture. Listing 2.12 shows how to do it.

```
function update()
    for j = 0, gMapHeight - 1 do
        for i = 0, gMapWidth - 1 do
            local tile = GetTile(gMap, gMapWidth, i, j)
            local texture = gTextures[tile]
            gTileSprite:SetTexture(texture)
            gTileSprite:SetPosition(
                gLeft + i * gTileWidth,
                gTop - j * gTileHeight)
            gRenderer:DrawSprite(gTileSprite)
        end
    end
end
```

Listing 2.12: Update render loop for the map. In main.lua.

An example project running this code is available in tilemap-8-solution. Run the code and you'll get something like Figure 2.7. Try changing the tile numbers in the map table and you'll be able to see how you can modify the map.

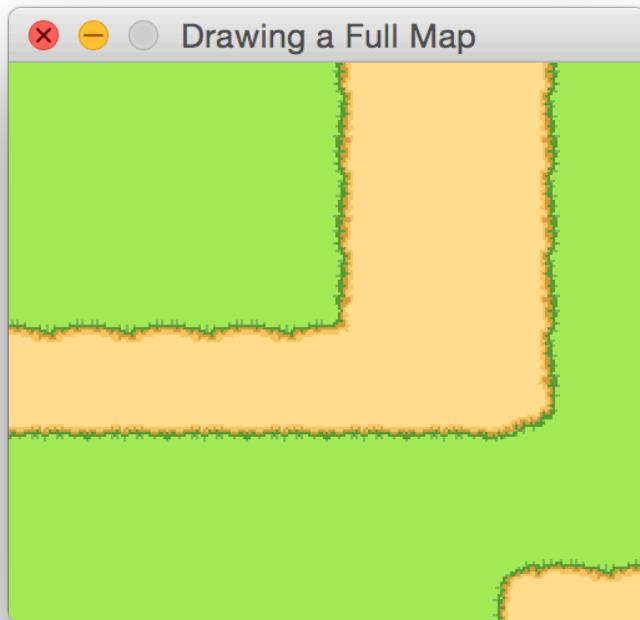


Figure 2.7: Rendering a map using a map definition table.

Maybe you can now begin to imagine how to use this technique to create a full RPG world. Before we start designing worlds, though, we're going to make our rendering faster.

Faster Rendering with a Texture Atlas

Currently every tile has its own image file. This can be slow. To make things faster we'll use one texture that contains all the tile images. This combined image file is called a *texture atlas*. You can see an example in Figure 2.8.

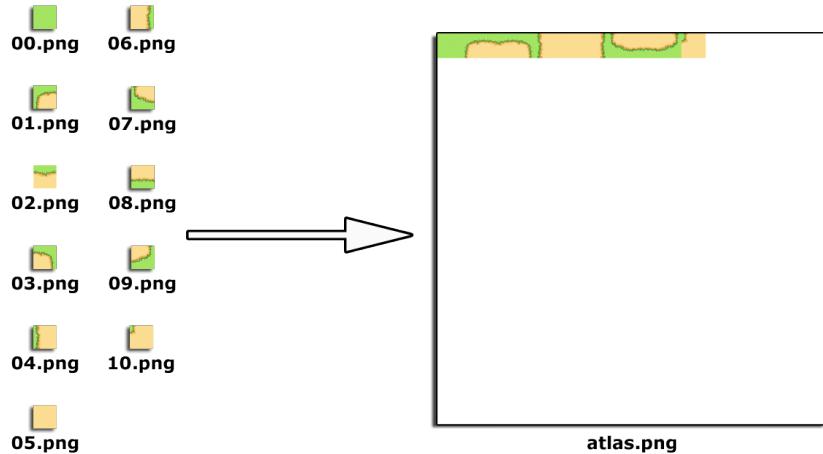


Figure 2.8: Combining lots of small images into a larger texture atlas.

Why are many files slower to use than a single file? Well, there's a degree of overhead when using a file; you have to tell the operating system you want the file, assign memory for it, parse it, and then free the memory. If you open twenty files you're doing that twenty times, but if you open one file you're only doing it once.

Even after we get the files into memory, there's still a speed difference between one texture and twenty. Every time we use a different texture we need to tell the graphics card. This can be a slow operation. If we use a texture per tile we'll be swapping textures often! The worst case would be once per tile which for an 8 by 7 map would be 57 times! Of course this gets even worse the larger the map, or imagine a map with layers, particle effects and characters! Using a texture atlas solves this problem by storing a large number of images in a single texture.

Mobile devices and some graphics also have restrictions about image sizes, only allowing power of 2 images. That is, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, etc. These size images fit in memory more efficiently. To use a non-power of 2 sprite, in this case, you must use an atlas.

A texture atlas is faster than individual tiles but it can be a pain to make and maintain. While you're developing a game, individual images are fine; you can leave using a texture atlas until the end of development. In our case, texture atlases are a perfect match for tileset, so it's easy and fast for us to use them.

Once we put all our tiles into one texture like in Figure 2.9, we have a problem; how do we make the tile sprites display the correct image? Well, we can use some tricks to make the sprite only display a small section of the texture atlas. Let's look at some code to make that happen.



Figure 2.9: A partial view of the 512 by 512 texture atlas image.

Implementing a Texture Atlas

To use a texture atlas we need to know about *UV coordinates*. UV coordinates describe which part of a texture to draw. U and V are like x, y coordinates (U is like X and V like Y); together they describe a point on a 2d image. UV coordinates range from 0 - 1²; you can think of 0 as 0% and 1 as 100%. Figure 2.10 below shows an example texture with some additional UV information.

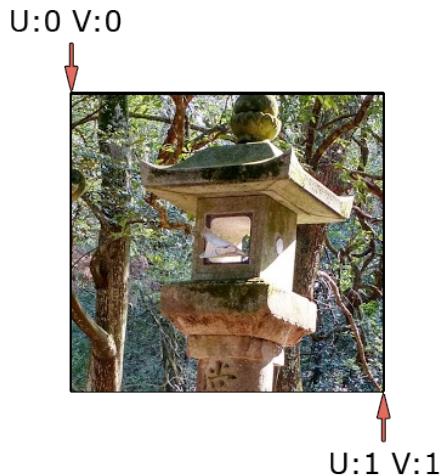


Figure 2.10: A texture as drawn with the default coordinates UV coordinates called out.

If you look at Figure 2.10, you'll see the default UVs for a sprite. The top left coordinate is 0, 0 and the bottom is 1, 1. This draws the sprite using the entire texture.

Let's try changing the UVs. If we make the top left equal to 0.5, 0.5 we get something like Figure 2.11. The sprite stays the same size. Its size is totally independent from its UV settings. Because our top left corner is set to the middle of the texture and our

²There are certain cases where moving outside of the 0 - 1 range makes sense but it's nothing we need to worry about when making our game.

bottom right corner is set to the bottom right of the texture, we draw the bottom right quarter of the image. If UV coordinates are new to you then it's worth experimenting. Check out tilemap-9 in the example folders tilemap-9. Change around the UVs and I'm sure you'll be able to imagine how we can alter the UVs to cut out any sprite we want.

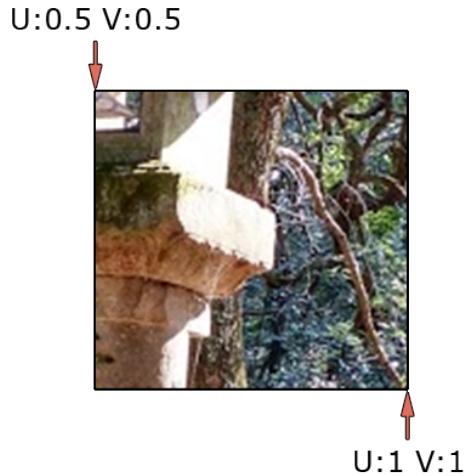


Figure 2.11: A texture draw with starting UVs of 0.5, 0.5 and ending UVs of 1,1.

To get the tiles out of the texture atlas we need to know the U,V coordinates for each tile. (Example tilemap-10 shows how we might calculate the UVs for each image in our texture atlas.)

Open example tilemap-11 to get a project referencing the texture atlas (shown in Figure 2.9) and with a simple map but incomplete render loop. We'll complete this example and use the texture atlas to draw our map.

Each type of tile has its own set of U,V coordinates. One coordinate pair is for the top left and one is for the bottom right, equalling four numbers in total. We'll store these numbers in a list. The first two numbers in the list are for the top left and the second two numbers are for the bottom right. We'll call this list gUVs, and it's shown in Listing 2.13. The list has 11 entries for our 11 tiles. Add this list to your own project.

```
gUVs =
{
  -- Left      Top      Right      Bottom
  -- U         V        U          V
  {0,           0,       0.0625,    0.0625},
  {0.0625,     0,       0.125,    0.0625},
  {0.125,      0,       0.1875,   0.0625},
```

```

{0.1875,      0,      0.25,      0.0625},
{0.25,        0,      0.3125,    0.0625},
{0.3125,     0,      0.375,     0.0625},
{0.375,       0,      0.4375,    0.0625},
{0.4375,     0,      0.5,       0.0625},
{0.5,         0,      0.5625,    0.0625},
{0.5625,     0,      0.625,     0.0625},
{0.625,       0,      0.6875,    0.0625},
}

```

Listing 2.13: The UV coordinates for each tile in the atlas. In main.lua.

Listing 2.13 has a lot of numbers - where did they come from? They were calculated! UVs for uniform images, like tilesets, are easy to calculate. First we work out the size of a tile in the texture atlas by dividing the tile width and height by the atlas width and height. In this case our tile width and height are 32 pixels and the atlas width and height are 512 pixels. Dividing 32 by 512 gives 0.0625, which the width (and height) of a tile in the texture atlas where the texture atlas width (and height) is 1. This is shown visually in Figure 2.12.

512

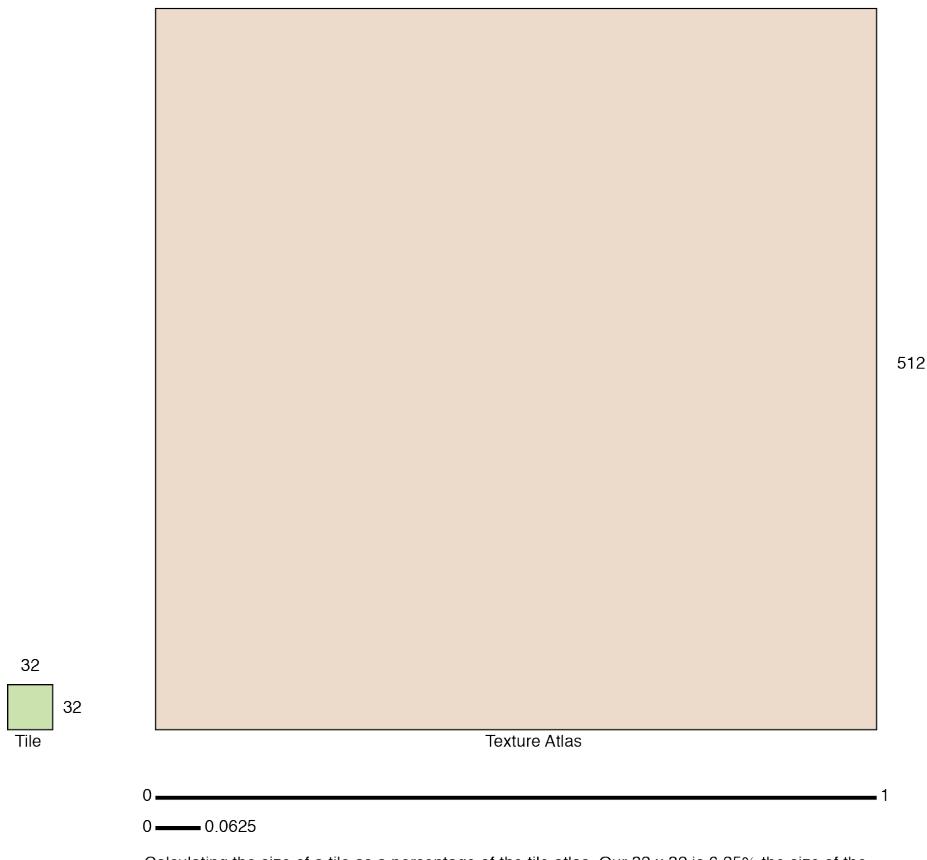


Figure 2.12: The proportions of a tile compared to a texture atlas.

We use the tile width and height to cut tiles out of the texture atlas. The UVs for the first tile are simple. The top left is $0, 0$; to get the bottom right we add the tile width and height. Therefore the bottom right is $0.0625, 0.0625$.

To get the top left of the second tile we just need to add the width to the U coordinates of the first tile, giving us $0.0625, 0, 0.125, 0.0625$. To get the rest of the tiles on this row, we can just keep adding the width to the last tile's U coordinates.

When we come to the end of the texture atlas width, we reset the U position to 0 and increase the V by 0.0625 , which is one tile height. Now we can write code to make it happen automatically! Example tilemap-10 uses the function shown in Listing 2.14 to generate the UVs for each tile.

```

function GenerateUVs(texture, tileSize)

    local uvs = {}

    local textureWidth = texture:GetWidth()
    local textureHeight = texture:GetHeight()
    local width = tileSize / textureWidth
    local height = tileSize / textureHeight
    local cols = textureWidth / tileSize
    local rows = textureHeight / tileSize

    local u0 = 0
    local v0 = 0
    local u1 = width
    local v1 = height

    for j = 0, rows - 1 do
        for i = 0, cols - 1 do
            table.insert(uvs, {u0, v0, u1, v1})
            u0 = u0 + width
            u1 = u1 + width
        end
        u0 = 0
        v0 = v0 + height
        u1 = width
        v1 = v1 + height
    end
    return uvs
end

```

Listing 2.14: Calculating UV coordinates for tiles in a texture atlas, when tiles are square. In main.lua.

The GenerateUVs function chops up an image into regular sized tiles of tileSize and returns a table of UV coordinates.

With the gUVs table we're ready to use the texture atlas to draw the tiles in the update loop.

```

gTileSprite = Sprite.Create()
gTileSprite:SetTexture(gTextureAtlas)

function update()
    for j = 0, gMapHeight - 1 do
        for i = 0, gMapWidth - 1 do

```

```

local tile = GetTile(gMap, gMapWidth, i, j)
local uvs = gUVs[tile]
gTileSprite:SetUVs(unpack(uvs))

gTileSprite:SetPosition(
    gLeft + i * gTileWidth,
    gTop - j * gTileHeight)
gRenderer:DrawSprite(gTileSprite)
end
end
end

```

Listing 2.15: Using the texture atlas to render tiles. In main.lua.

Listing 2.15 shows the main loop converted to use a texture atlas. Instead of looking up the texture, the texture is now the same for all tiles.

We now look up each tile's UVs in gUVs and get a table of coordinates back. The returned tables are the same we saw in Listing 2.13. We set the sprite's UVs to using unpack to give us the correct image for the tile.

If you're new to Lua, the unpack function might be unfamiliar. unpack takes a table and explodes all the entries. Listing 2.16 shows how unpack is used.

```

a = unpack({1})
print(a) -- 1

a, b = unpack({1, 2})
print(a, b) -- 1 2

a, b, c = unpack({1, 2, 3})
print(a, b, c) -- 1 2 3

```

Listing 2.16: Unpack examples.

In Listing 2.15, `gTileSprite:SetUVs(unpack(uvs))` is the equivalent of `gTileSprite:SetUVs(uvs[1], uvs[2], uvs[3], uvs[4])`.

That's all there is to the render function. Once the UVs are set, the sprite is rendered the same as before.

Running the code displays the same tilemap as tilemap-8-solution but renders it in a more efficient way. The entire solution for using a texture atlas is available in the examples folder as tilemap-11-solution.

Rendering the Tiled Lua Map Format

Have you experimented with creating your own maps yet? If not give it a go now; try to create a new original map or two.

Back? Right, I'm sure you now know that creating maps by hand in the editor is tedious! Don't worry, there's a better way; a map editor. When creating content, such as maps, for your RPG, you're sure to come across tedious, repetitive tasks. If you come across a task you do a lot then consider writing a tool to automate it or make it easier.

A map editor is a tool that makes creating maps easier and faster. We could write our own map editor but luckily there's already a fully featured open source editor, called Tiled, that suits our needs well. There's a version in the programs folder, or you can grab the latest version from <http://www.mapeditor.org/>.

Run the Tiled program and you'll be presented with a window as shown in Figure 2.13. Tiled is a very feature-complete map-making program, but we'll only cover the basics; if you find yourself using it often then it's worth investigating its advanced features.

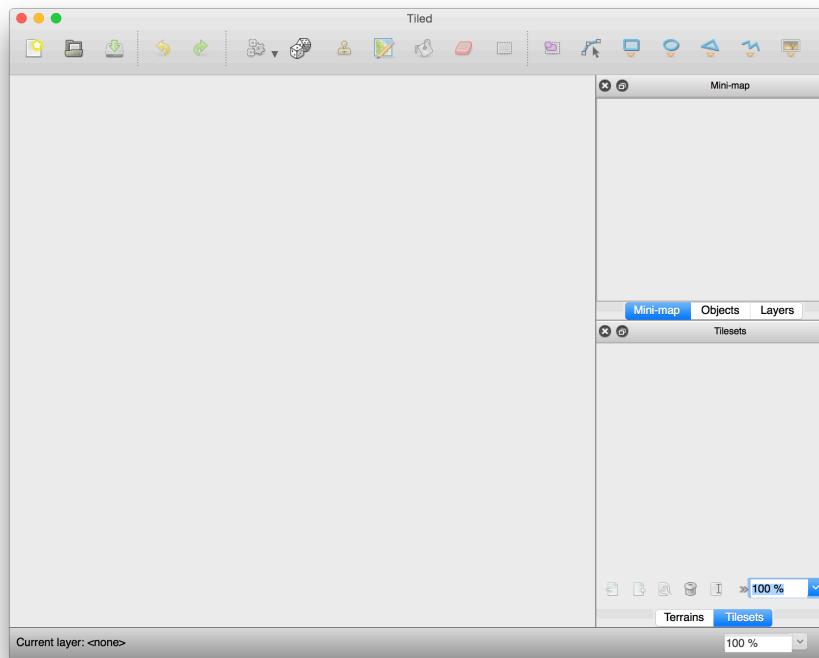


Figure 2.13: The Tiled program.

We'll start by using our texture atlas to create a new map using Tiled. Then we'll

write some code to display the Tiled map in our game. Tiled makes this very easy because it can export maps as Lua code.

Start a new project by choosing File > New from the menu bar. This brings up the dialog in Figure 2.14 asking what kind of map to create.

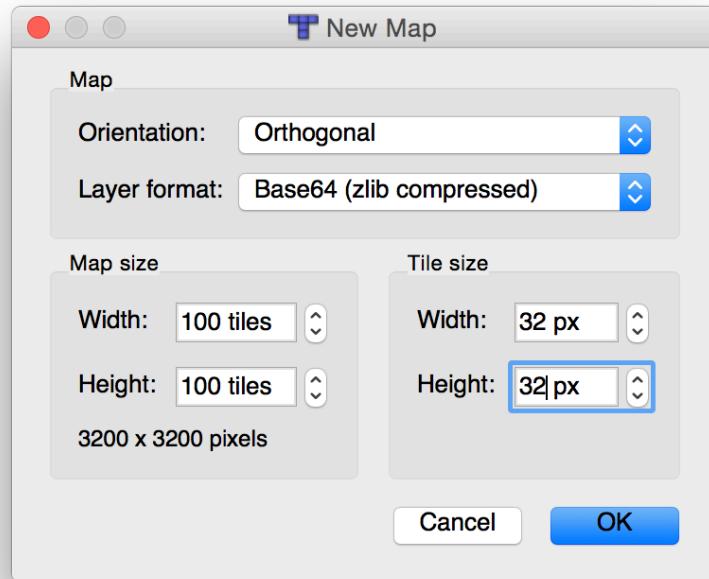


Figure 2.14: The Tiled Dialog to create a new map.

The first prompt is “Map Orientation,” with a default value of “Orthogonal.” This is what we want. Orthogonal means that the map is presented from a bird’s eye view. The alternative options are “Isometric” and “Isometric (staggered).” These use tiles that give the illusion that you’re seeing the map from an angle. All three orientation types are shown in Figure 2.15. Tactical RPGs often favor isometric or staggered isometric maps.

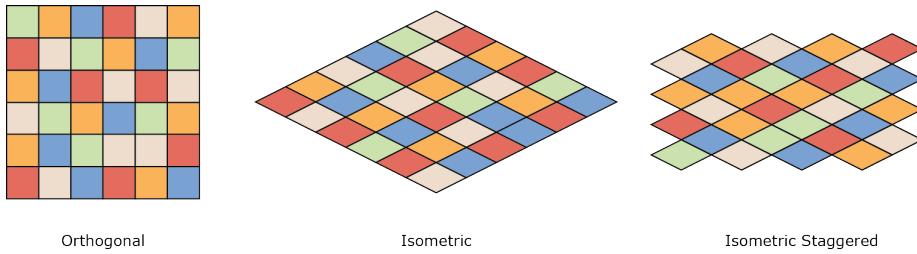


Figure 2.15: The three possible tiled map options in the Tiled program.

The bottom left of the New Map dialog, in Figure 2.14, has options for the size of the map in tiles. The default is 100 x 100. Let's change this to be 8 x 7 so it exactly fills the screen size we're using. The final section of the dialog is the size of the tile in pixels; the tiles in our atlas are 32 x 32 pixels so make sure that's set as the value. Press OK and you'll be shown a blank gray screen.

To start creating our map we need to tell Tiled to use our atlas as the tileset. In the example project tilemap-12 there's a file called `atlas.png`, the same texture atlas used in the last section. We'll use this texture to draw our map. In Tiled go to the Menu Bar and choose Map > New Tileset as shown in Figure 2.16.

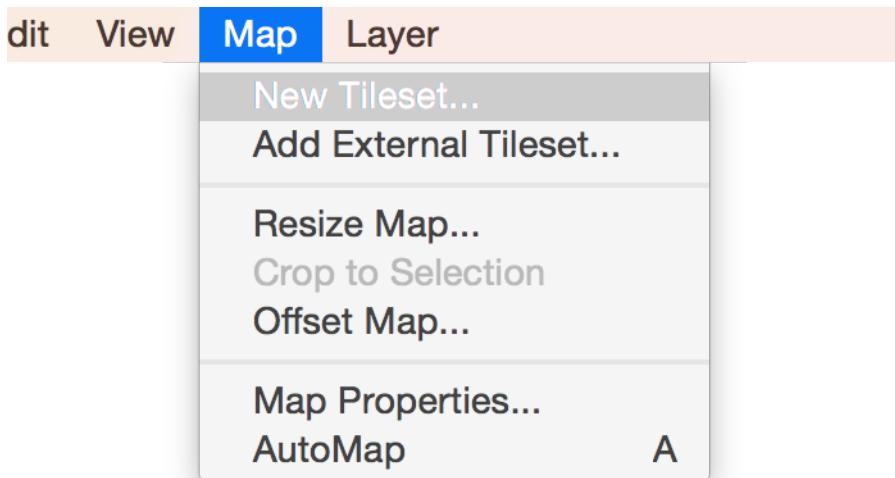


Figure 2.16: Adding a New Tileset to a map.

Figure 2.17 shows the Tileset dialog. The first field is a textbox asking for the tileset name; the name doesn't matter and is only for our reference. We can name our tileset

something simple like "atlas." The second textbox asks for the actual image file of the tileset. Click the Browse button and choose atlas.png in the tilemap-12 folder.

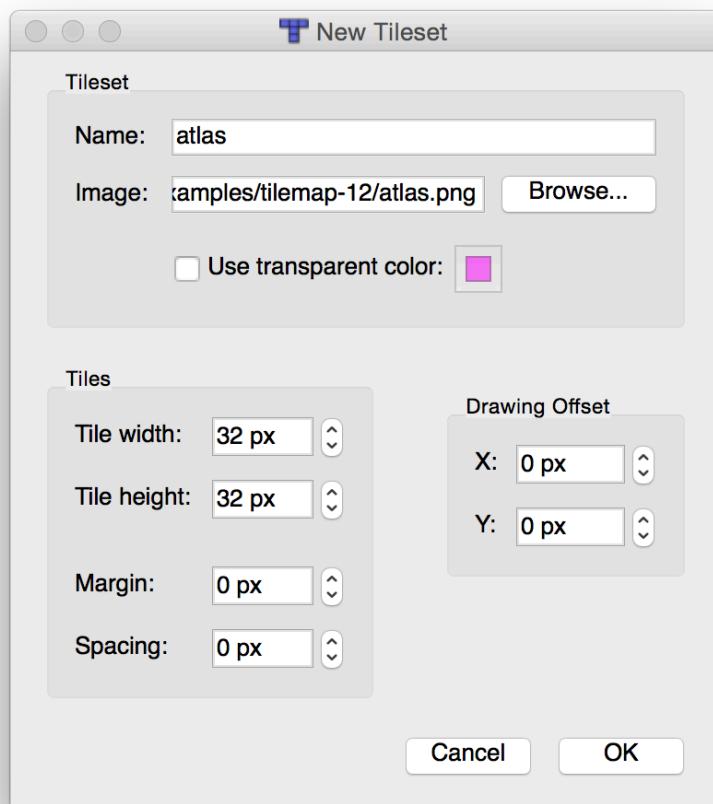


Figure 2.17: New Tileset Dialog.

The third field is a checkbox controlling transparent color use. This can be ignored as the png format handles transparency for us.

On the bottom right there's a section labelled Drawing Offset. Our files don't use an offset, so this too can be ignored.

The margin and spacing settings (in the Tile section) can be ignored, but tile width and tile height should both be set to 32. Then press OK. In the bottom right of the Tiled

window (see Figure 2.18) the tiles have been correctly cut up and added to a palette. We can use the tile palette to paint the map. This process can be repeated for any texture atlas. A full game will need several tilesets.

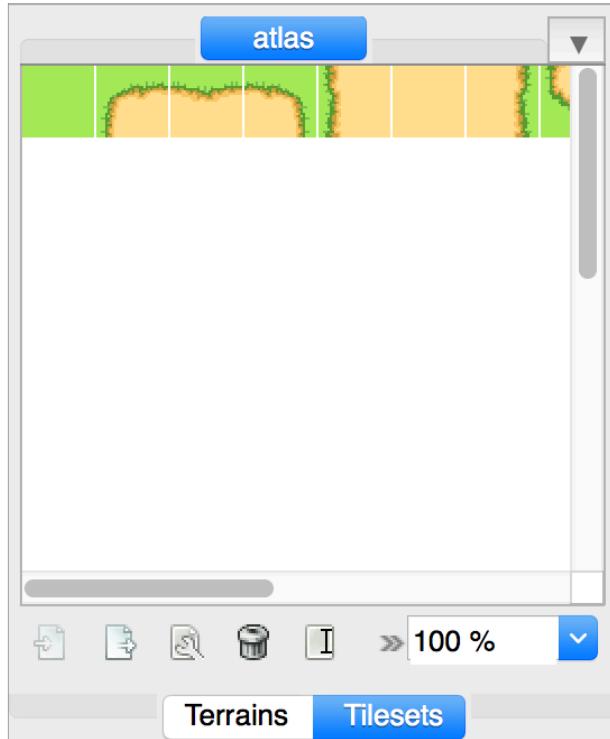


Figure 2.18: Our atlas loaded and ready to use in Tiled.

With the tiles added we can start creating maps! Click on a tile, then click on the map to place it. The drawable area can be quite hard to see for a small map like the one we're making, by default it's centered in the gray pane below the tab labelled "Untitled.tmx".

Tiled has a few different drawing tools. They're available in the toolbar at the top of the Tiled window. The two most useful tools are the paint bucket for filling a large area and the eraser for removing a tile. The rest of the tools in the toolbar are for more advanced usage and are worth learning as you start to create bigger maps. For now use the simpler tools and atlas to create a map to your satisfaction. Once you're happy with the map, you can save it using Tiled's file format .tmx. Save often, keep the tmx files in source control³ if you're using it, and remember to keep backups. You don't want to spend hours creating a beautiful map only to lose it due to a power cut or hard drive

³Source control is a name of for a type of program that programmers use to keep their work safe. It lets

failure! The example map I created can be seen in Figure 2.19 and in the examples folder under tilemap-12-solution.

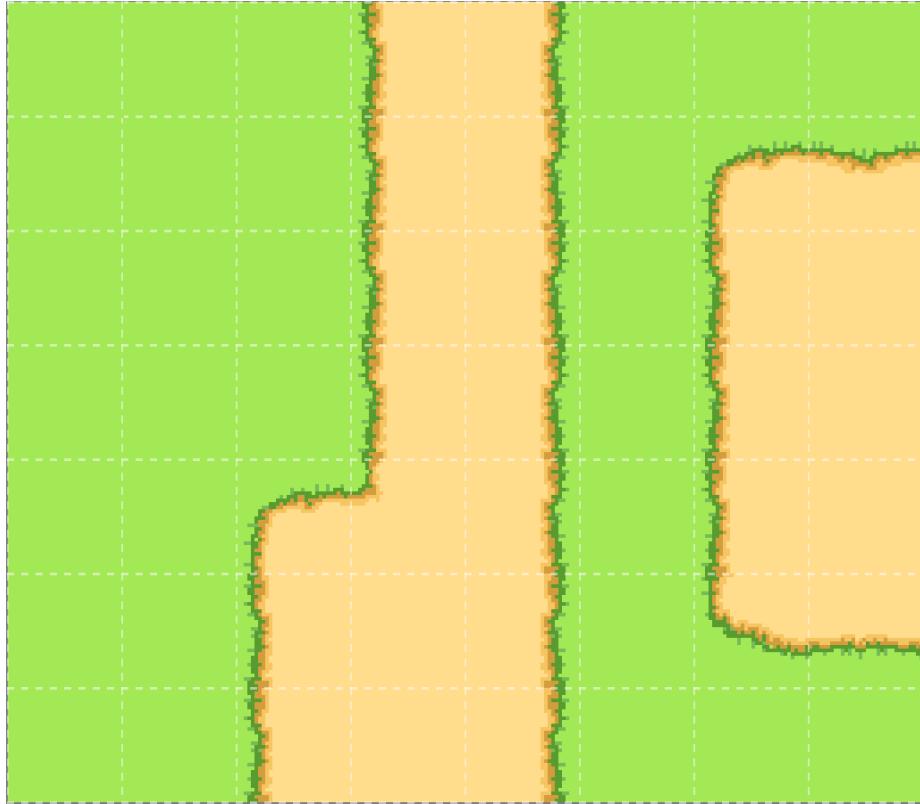


Figure 2.19: An example map created in Tiled.

Rendering a Tiled Map

Let's update our code to use a Tiled map. First tell Tiled to export the map as a Lua file by choosing File > Export As and selecting "Lua file" in the Save As Type combobox.

In example tilemap-13 there's an exported map file and the texture atlas. If you've made your own map then replace `example_map.lua` with your map.

Open up `example_map.lua` and let's look at Listing 2.17 to see what Tiled has written.

you easily revert to earlier versions of your project if something goes wrong, and makes it easy to save your work remotely so if your computer breaks you don't lose your project. It's worth learning more about if you're going to make a big project!

```

return {
    version = "1.1",
    luaversion = "5.1",
    orientation = "orthogonal",
    width = 8,
    height = 7,
    tilewidth = 32,
    tileheight = 32,
    properties = {},
    tilesets = {
        {
            name = "atlas",
            firstgid = 1,
            tilewidth = 32,
            tileheight = 32,
            spacing = 0,
            margin = 0,
            image = "../tilemap-12-solution/atlas.png",
            imagewidth = 512,
            imageheight = 512,
            properties = {},
            tiles = {}
        }
    },
    layers = {
        {
            type = "tilelayer",
            name = "Tile Layer 1",
            x = 0,
            y = 0,
            width = 8,
            height = 7,
            visible = true,
            opacity = 1,
            properties = {},
            encoding = "lua",
            data = {
                1, 1, 1, 5, 7, 1, 1, 1,
                1, 1, 1, 5, 7, 1, 2, 3,
                1, 1, 1, 5, 7, 1, 5, 6,
                1, 1, 1, 5, 7, 1, 5, 6,
                1, 1, 2, 11, 7, 1, 5, 6,
                1, 1, 5, 6, 7, 1, 8, 9,
                1, 1, 5, 6, 7, 1, 1, 1
            }
        }
    }
}

```

```
        }
    }
}
```

Listing 2.17: The contents of example_map.lua.

As you can see from Listing 2.17 Tiled's map format contains a lot of data. Most of this data is irrelevant for our needs. There's a return statement at the start of the file which will return this big map definition table as soon as the file is loaded. We're going to wrap the code in a function so it's easier for us to use. Listing 2.18 shows how to change the code to make a function called CreateMap1; the function definition is added to the first line and an end statement is appended to the end of the file.

```
function CreateMap1()
    return {
        -- Map definition omitted
    }
end
```

Listing 2.18: Modifying the map file so it's easier for us to use.

In the map definition the tilesets table has a field, image, pointing to our texture atlas. Change this line so it's like Listing 2.19. This lets Dinodeck find the correct texture.

```
image = "atlas.png",
```

Listing 2.19: Changing the image field for Dinodeck to use.

That's the only modification we need to make to the map. If you're making a lot of maps, this is a good place to write a tool to format them automatically!

In the main.lua file somewhere near the top, create the map, as shown in Listing 2.20.

```
LoadLibrary("Renderer")
LoadLibrary("Sprite")
LoadLibrary("System")
LoadLibrary("Texture")
LoadLibrary("Asset")

Asset.Run("example_map.lua")
gTiledMap = CreateMap1()
```

Listing 2.20: Creating the map. In main.lua.

We store the Tiled map data structure in `gTiledMap` (as we saw in Listing 2.17) ready to render. Example tilemap-13 has an empty render function, so let's write the render code next.

The Tiled map makes use of a tilesset. Tilessets are just a type of texture atlas. Our texture atlas / tilesset is used by the project under the name `atlas.png`. We know how to calculate the UV settings for each tile, so there's no need to hard code them. Add the `GenerateUVs` function to the `main.lua`.

```
function GenerateUVs(tileWidth, tileHeight, texture)

    -- This is the table we'll fill with UVs and return.
    local uvs = {}

    local textureWidth = texture:GetWidth()
    local textureHeight = texture:GetHeight()
    local width = tileWidth / textureWidth
    local height = tileHeight / textureHeight
    local cols = textureWidth / tileWidth
    local rows = textureHeight / tileHeight

    local ux = 0
    local uy = 0
    local vx = width
    local vy = height

    for j = 0, rows - 1 do
        for i = 0, cols - 1 do

            table.insert(uvs, {ux, uy, vx, vy})

            -- Advance the UVs to the next column
            ux = ux + width
            vx = vx + width

        end

        -- Put the UVs back to the start of the next row
        ux = 0
        vx = width
        uy = uy + height
        vy = vy + height
    end
    return uvs
end
```

Listing 2.21: Calculate the UV for the texture atlas. In main.lua.

Add a call to this function, passing in some of the parameters from the Tiled map data structure, as shown in Listing 2.22.

```
gTextureAtlas = Texture.Find(gTiledMap.tilesets[1].image)
gUVs = GenerateUVs(
    gTiledMap.tilesets[1].tilewidth,
    gTiledMap.tilesets[1].tileheight,
    gTextureAtlas)
```

Listing 2.22: Create the UVs for a Tiled map. In main.lua.

In this code snippet we create a texture using the name of the first tileset in the Tiled map table. We then use the texture and width and height settings to create a table of UVs.

We have the texture `gTextureAtlas` and the tile UVs `gUVs`, so let's write the render code!

```
gMap = gTiledMap.layers[1]
gMapHeight = gMap.height
gMapWidth = gMap.width
gTileWidth = gTiledMap.tilesets[1].tilewidth
gTileHeight = gTiledMap.tilesets[1].tileheight
gTiles = gMap.data

gTop = gDisplayHeight / 2 - gTileHeight / 2
gLeft = -gDisplayWidth / 2 + gTileWidth / 2

function update()
    for j = 0, gMapHeight - 1 do
        for i = 0, gMapWidth - 1 do
            local tile = GetTile(gTiles, gMapWidth, i, j)
            local uvs = gUVs[tile]
            gTileSprite:SetUVs(unpack(uvs))
            gTileSprite:SetPosition(
                gLeft + i * gTileWidth,
                gTop - j * gTileHeight)
            gRenderer:DrawSprite(gTileSprite)
        end
    end
end
```

Listing 2.23: The render loop to draw a Tiled map. In main.lua.

The code to render a Tiled map is extremely similar to how we were rendering our own map format previously. The main difference is we're now pulling some of the data out of the Tiled map table. We've moved the gTop and gLeft variables below where gTileWidth and gTileHeight are defined. Run the code and it will display the map.

We can now display maps from Tiled, meaning we can create new game areas much more rapidly and easily! The code demonstrating this Tiled map rendering is available as tilemap-13-solution in the examples folder.

Rendering a different map

Let's test our code by having it render a different map. Open example tilemap-14 which contains a new Tiled map and atlas. This atlas uses different sized tiles and the map itself is much larger than the one we've been using so far.

Run the program and you'll get something like Figure 2.20 displaying a brand new map.

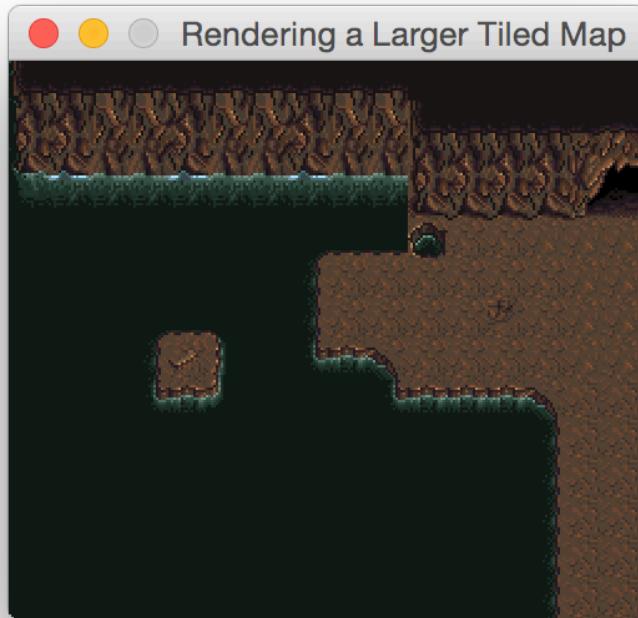


Figure 2.20: Reusing our code to render a different map.

Example tilemap-14 contains the new Tiled .tmx file; open it in Tiled and you'll see it's vastly bigger than the small portion we're displaying. Our window only shows 72 tiles but we render every single tile in the map! We're rendering a total of 8192 tiles, and if our map was bigger we'd draw even more! We should only draw the tiles that are inside, or partially inside, the view area. Let's think about how to do that next.

A Tree Falls in the Forest and No One Is Around to Hear It

If we only draw the tiles we can see in the window then the code will run faster. Let's change the code to make this happen.

First we need to figure out which tile intersects the top left corner of the viewport. Figure 2.21 shows the relationship between the viewport's top left pixel location and the tile it intersects. The viewport is shown as a red square. The top left corner of the viewport intersects the tile at X: 2, Y: 2, highlighted in yellow on the figure. In Figure 2.21 the map's top left pixel location is 0, 0 but it may be at any location.



Figure 2.21: The viewport's relationship to a large map.

Let's create a function that takes in a point and returns the coordinates of the tile under that point. If the point is outside of the map, then we'll force it into the map's bounds by clamping the value. There's an implementation of such a function in Listing 2.24.

```

function PointToTile(x, y, tileSize, left, top, mapWidth, mapHeight)

    -- Tiles are rendered from the center so we adjust for this.
    x = x + tileSize / 2
    y = y - tileSize / 2

    -- Clamp the point to the bounds of the map
    x = math.max(left, x)
    y = math.min(top, y)
    x = math.min(left + (mapWidth * tileSize) - 1, x)
    y = math.max(top - (mapHeight * tileSize) + 1, y)

    -- Map from the bounded point to a tile
    local tileX = math.floor((x - left) / tileSize)
    local tileY = math.floor((top - y) / tileSize)

    return tileX, tileY
end

```

Listing 2.24: Code to map a point to a tile. In main.lua.

The basic way to map a point to a tile is to divide the point by the size of the tile. This works perfectly as a solution if your tiles are drawn from the topleft point, and the map is located at 0, 0. In a game with maps larger than the screen this won't always be the case.

As the viewport moves over a large map, the top left point of the viewport moves over many different tiles. We need to account for this by relating the top left corner of the map to the top left corner of the view.

Let's take a closer look at the PointToTile function in Listing 2.24. The first two parameters x and y are the point we want to test to find out which tile they intersect. The tileSize is the size of tile in pixels. The parameters top and left are the pixel positions of the top left corner of the map. The top and left values are used to offset x and y when calculating which tile the point is in. The mapWidth and mapHeight parameters are the width and height of the map in tiles.

Open tilemap-15 from the examples folder and you can see this code in action. If you run the example, you'll see the mouse pointer is drawn as a red dot with text below describing the tile it intersects, as shown in Figure 2.22. Try changing the position of the map by altering the gLeft and gTop variables. Try 0, 0 and see how everything still works.



Figure 2.22: Transforming a point to a tile location. The tilemap-15 example program.

This code, mapping a point to a tile, is a little complicated. If you really want to get it, delete the function in tilemap-15 and try to rewrite it yourself! You may want a bit of graph paper to help.

Now that we've created the PointToTile function we can get the tiles at the top left and bottom right of the viewport. Then we only need to render tiles in between these two. Instead of rendering thousands of tiles we'll render less than one hundred, a pretty good saving for such a small function!

Open example tilemap-16. It includes the PointToTile function but the update still renders every tile in the map. We're going to change that! Let's get the top left and bottom right viewport tiles, as shown in Listing 2.25. Copy this code after the PointToTile function definition.

```

gTileLeft, gTileTop = PointToTile(
    -System.ScreenWidth() / 2,
    System.ScreenHeight() / 2,
    gTileWidth,
    gLeft, gTop,
    gMapWidth, gMapHeight)
gTileRight, gTileBottom = PointToTile(
    System.ScreenWidth() / 2,
    -System.ScreenHeight() / 2,
    gTileWidth,
    gLeft, gTop,
    gMapWidth, gMapHeight)

```

Listing 2.25: Getting top left and bottom right tiles shown in the viewport.

These functions are looking pretty messy, but don't worry. We'll pull everything together into a tidy class soon! To get the top left tile we pass in the top left pixel position and to get the bottom right tile we pass in the bottom right pixel position. That's pretty straightforward. This gives us all the information we need to limit rendering to those tiles in the window. Navigate to the update function and change the loop so it appears as in Listing 2.26.

```

function update()

    for j = gTileTop, gTileBottom do
        for i = gTileLeft, gTileRight do

```

Listing 2.26: Efficient render loop. In main.lua.

Run the code and you'll see nothing! It looks exactly the same, which is what we want, no graphical changes but much much faster! The example tilemap-16-solution has all the code we've added up to this point.

Making a Map Class

Let's bring all our small test programs together into a clean, reusable class. Open example tilemap-17 and create a new file called Map.lua. Copy in the code from Listing 2.27. This is an empty class that we're going to flesh out. Tilemap-17's main.lua file is nearly empty. Much of the code we had in main.lua is going to go into our map class.

```

Map = {}
Map.__index = Map
function Map:Create(mapDef)

```

```

local this = {}
setmetatable(this, self)
return this
end

function Map:Render(renderer)
end

```

Listing 2.27: An empty map class. In Map.lua.

The Map constructor takes a single parameter, `mapDef`. The `mapDef` parameter is a Tiled map definition. As before, we assume all maps have a single layer and single texture atlas. Let's add some fields to the constructor to store information from `mapDef`. We'll also get the texture, create a tile sprite, and generate UVs for the texture map. These new fields are shown in Listing 2.28.

```

function Map:Create(mapDef)
    local layer = mapDef.layers[1]
    local this =
    {
        mX = 0,
        mY = 0,

        -- To track the camera position
        mCamX = 0,
        mCamY = 0,

        mMapDef = mapDef,
        mTextureAtlas = Texture.Find(mapDef.tilesets[1].image),

        mTileSprite = Sprite.Create(),
        mLAYER = layer,
        mWidth = layer.width,
        mHeight = layer.height,

        mTiles = layer.data,
        mTileWidth = mapDef.tilesets[1].tilewidth,
        mTileHeight = mapDef.tilesets[1].tileheight
    }
    this.mTileSprite:SetTexture(this.mTextureAtlas)

    -- Top left corner of the map
    this.mX = -System.ScreenWidth() / 2 + this.mTileWidth / 2
    this.mY = System.ScreenHeight() / 2 - this.mTileHeight / 2

```

```

-- Additional fields
this.mWidthPixel = this.mWidth * this.mTileWidth
this.mHeightPixel = this.mHeight * this.mTileHeight
this.mUVs = GenerateUVs(mapDef.tilesets[1].tilewidth,
                        mapDef.tilesets[1].tileheight,
                        this.mTextureAtlas)

setmetatable(this, self)
return this
end

```

Listing 2.28: Adding the map fields. In Map.lua.

Our test programs had many global variables. They are now members of the Map class. We've added `mCamX` and `mCamY` to represent the pixel position for the top left corner of the viewport. By changing the values we can move around the map. The `mWidthPixel` and `mHeightPixel` fields store the width and height of the entire map in pixels.

Next let's add functions. We're not adding `GenerateUVs` to the Map class because it will be used by many different classes. For now we'll leave it in `main.lua`. Add the functions from Listing 2.29.

```

function Map:PointToTile(x, y)

    -- Tiles are rendered from the center so we adjust for this.
    x = x + self.mTileWidth / 2
    y = y - self.mTileHeight / 2

    -- Clamp the point to the bounds of the map
    x = math.max(self.mX, x)
    y = math.min(self.mY, y)
    x = math.min(self.mX + self.mWidthPixel - 1, x)
    y = math.max(self.mY - self.mHeightPixel + 1, y)

    -- Map from the bounded point to a tile
    local tileX = math.floor((x - self.mX) / self.mTileWidth)
    local tileY = math.floor((self.mY - y) / self.mTileHeight)

    return tileX, tileY
end

function Map:GetTile(x, y)
    x = x + 1 -- change from 1 -> rowsize
              -- to          0 -> rowsize - 1
    return self.mTiles[x + y * self.mWidth]
end

```

```

end

function Map:Render(renderer)

    -- Get the topleft and bottomright pixel of the camera
    -- and use to get the tile
    local tileLeft, tileBottom =
        self:PointToTile(self.mCamX - System.ScreenWidth() / 2,
                        self.mCamY - System.ScreenHeight() / 2)

    local tileRight, tileTop =
        self:PointToTile(self.mCamX + System.ScreenWidth() / 2,
                        self.mCamY + System.ScreenHeight() / 2)

    for j = tileTop, tileBottom do
        for i = tileLeft, tileRight do

            local tile = self:GetTile(i, j)
            local uvs = self.mUVs[tile]

            self.mTileSprite:SetUVs(unpack(uvs))

            self.mTileSprite:SetPosition(self.mX + i * self.mTileWidth,
                                         self.mY - j * self.mTileHeight)

            renderer:DrawSprite(self.mTileSprite)
        end
    end
end

```

Listing 2.29: Adding the remaining map functions. In Map.lua.

Listing 2.29 adds the previously global functions to the Map class. These functions are simpler. They don't need as many parameters because they can access the internal Map class fields.

In the Render function, the mCamX and mCamY fields are used to get the top left and bottom right tiles using PointToTile. Then each tile in between is rendered out.

That's the map class finished for now, so let's use it. Open manifest.lua and make it look like Listing 2.30 by adding the Map.lua file reference. This is how we add any code file to a project. I won't show the exact manifest code every time we add a new code file but you can always refer to Listing 2.30 for a reminder!

```

manifest =
{
    scripts =
    {
        ['main.lua'] =
        {
            path = "main.lua"
        },
        ['Map.lua'] =
        {
            path = "Map.lua"
        },
        ['larger_map.lua'] =
        {
            path = "larger_map.lua"
        }
    },
}

```

Listing 2.30: Adding the new Map class to the manifest. In manifest.lua.

We've created the Map class and added it to our project, so it's now ready to use. Open main.lua and alter the code to match Listing 2.31.

```

LoadLibrary("Renderer")
LoadLibrary("Sprite")
LoadLibrary("System")
LoadLibrary("Texture")
LoadLibrary("Asset")

Asset.Run("Map.lua")
Asset.Run("larger_map.lua")

function GenerateUVs(tileWidth, tileHeight, texture)
    -- code omitted
end

local gMap = Map:Create(CreateMap1())
gRenderer = Renderer:Create()

function update()
    gMap:Render(gRenderer)
end

```

Listing 2.31: Using the map class. In main.lua.

The code in Listing 2.31 shows how using a map class makes the code simpler. Run the project and you'll see it renders the map as before. The full working code is in tilemap-17-solution.

GenerateUVs looks a little out of place now. For future examples we'll move the function to a new code file called Util.lua. The Util file holds our utility functions. If you want to use Util.lua in your own project now, remember to add it to the manifest file and call Asset.Run on it.

We've made our map rendering code fast and clean but the camera is still fixed. To make a real game we need to move the camera. Let's add simple camera controls next!

A 2D Camera

A 2d camera is a combination of the viewport (which we already have) and simple functions to position it. We want to be able to give the camera a position and have it render that part of the world. For a demo we'll add keyboard input so we can pan the camera around the map. We'll use the Translate function of the Renderer to move the position of the Renderer around the world.

Example tilemap-18 has the latest code, or you can continue working with the code you have. To check keyboard input we call LoadLibrary("Keyboard"). This adds many global variables, including KEY_LEFT, KEY_UP, KEY_RIGHT and KEY_DOWN, which correspond to the arrow keys on the keyboard. Copy the code from Listing 2.32 to add checks detecting which keys are pressed. When we press an arrow key, the camera moves one pixel in that direction.

```
-- code omitted
LoadLibrary("Keyboard")

-- code omitted

function update()
    gRenderer:Translate(-gMap.mCamX, -gMap.mCamY)
    gMap:Render(gRenderer)

    if Keyboard.Held(KEY_LEFT) then
        gMap.mCamX = gMap.mCamX - 1
    elseif Keyboard.Held(KEY_RIGHT) then
        gMap.mCamX = gMap.mCamX + 1
    end

    if Keyboard.Held(KEY_UP) then
        gMap.mCamY = gMap.mCamY + 1
    elseif Keyboard.Held(KEY_DOWN) then
```

```
    gMap.mCamY = gMap.mCamY - 1
end
end
```

Listing 2.32: Moving around the map. In main.lua.

In the update loop Keyboard.Held is called each frame. It returns True if the key is held down and False otherwise. If we press and hold the left arrow key, each frame we'll move the camera left because the mCamX value is decreased by 1 pixel. Run the program and try moving around with the keys. It's important that camera coordinates only ever be whole numbers because of the way we're mapping 2d pixels to the 3d environment.

At the top of the update function there's a call to Render:Translate which handles moving the renderer around the world. Now that we can look around the world, everything is starting to feel much more gamelike! Imagine panning up over a map as a game begins. This a good point to take a little time out and congratulate yourself!

Moving the camera in various directions is fine, but we also need to tell it to jump to various points - go to position x:100, y:-300 for instance. Add the function shown in Listing 2.33. The Y is negated to match the Tiled's pixel coordinates. The further down on the map, the more negative the Y value. It doesn't really matter either way, it just affects how we refer to a pixel position on the map.

```
function Map:Goto(x, y)
    self.mCamX = x
    self.mCamY = -y
end
```

Listing 2.33: Simple code to focus the camera on a point. In Map.lua.

Example tilemap-19 demonstrates Goto in action. On the example map there's an interesting little rock at x:1984 y:832. Figure 2.23 shows what happens when we focus on it. The little rock appears in the top left of the window! That's not what want we want. If we focus on something, it should appear dead center on the screen. Let's modify the function and change it so it's like Figure 2.24.

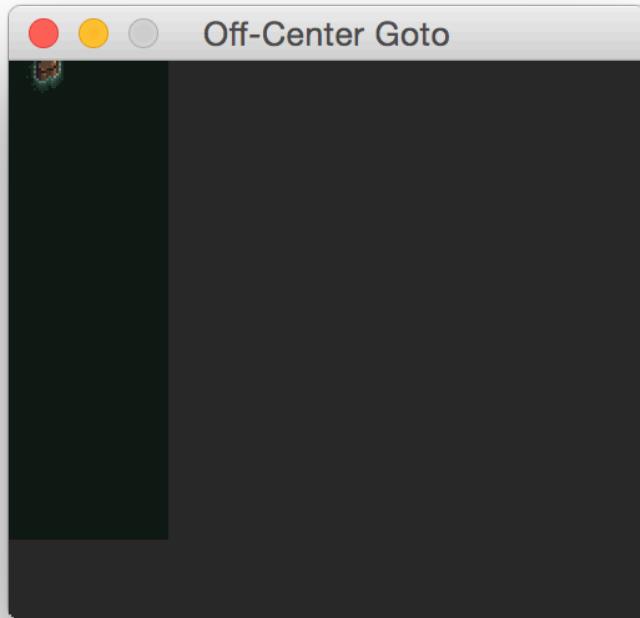


Figure 2.23: A little off focus.

To correct the camera we need to move the window by half its width and height to center it on the point of interest. The updated Goto function is shown in Listing 2.34. Example tilemap-20 uses this updated function and as you can see in Figure 2.24 the rock is now centered.

```
function Map:Goto(x, y)
    self.mCamX = x - System.ScreenWidth()/2
    self.mCamY = -y + System.ScreenHeight()/2
end
```

Listing 2.34: Simple code to focus the camera on a point. In *Map.lua*.

Soon we'll add items more interesting than rocks! Before leaving the map class, let's add one last function, *GotoTile*, a nice shortcut for focusing on a specific tile. We'll

calculate the pixel position for the tile, then call our Goto function. To get the pixel position we multiply the X and Y coordinates by the width and height of the tile, then add an extra half of the tile width and height to get it dead center. You can see the code in Listing 2.35.

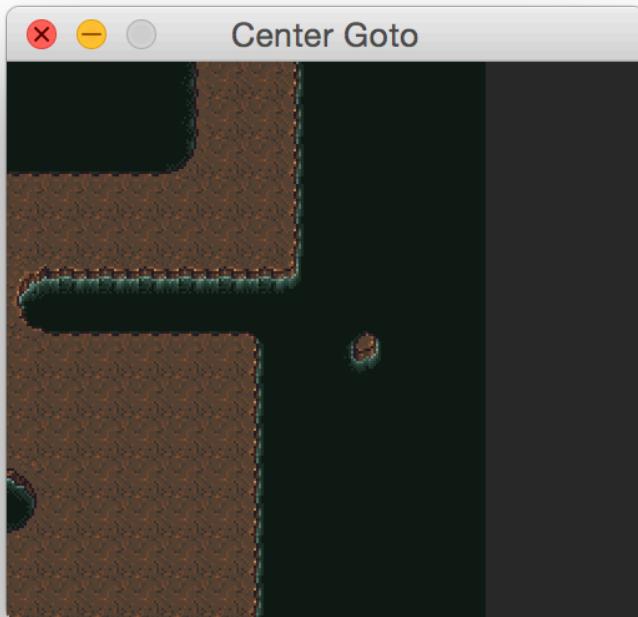


Figure 2.24: The Goto function centers the camera on the target.

All the code we've written in this chapter is available in example tilemap-21. We've been using a small window about the size of the old SNES screen; on a modern computer it's really quite tiny! Therefore tilemap-21 doubles the window resolution which makes things bigger and demonstrates that our code is robust.

```
function Map:GotoTile(x, y)
    self:Goto((x * self.mTileWidth) + self.mTileWidth/2,
              (y * self.mTileHeight) + self.mTileHeight/2)
end
```

Listing 2.35: Simple code to focus the camera on a point. In Map.lua.

Review

We're well on our way to making a game!

With the code we've written so far, we can load maps made in an editor, efficiently render them, and navigate around using basic keyboard input or coordinates. These are essential tasks we'll use to build up our game.

Already we could start constructing haunted castles, secret underground bases or deep underground caverns. You should always feel free to play around with the code and if you suddenly want to try something out, go and do it! The book will be here, waiting where you left off. For instance, is there any way you think you could extend the map now? How do you think characters and NPCs will be added? Never worry about breaking anything. Breaking code helps you learn faster.

Next we'll add a Player character, a walk cycle, basic collision detection, and interaction triggers. Once we're controlling a character and moving around, it will start to really feel like a game.

From Maps to Worlds

All the maps we've created so far have been unmoving and dead. In this chapter we'll start to breathe life into our maps. We'll add a player character who can move around and explore the map. Then we'll add extra data to the tiles to determine if the player can walk on them. If the tile is wall, for instance, we'll say it's *blocking* and in the code we'll stop the player from walking on it.

We'll upgrade our maps to support additional layers allowing characters to be hidden behind arches and foliage. Layers give maps depth and make the world seem more real. Finally we'll add the concept of an *action*, a small piece of code that interacts with the game world. Example actions might include: transporting the player to a new map, healing the player, removing a blockage and so on. Actions are fired by triggers attached to tiles. We can use triggers and actions to allow a character to move from one map to another which really starts to open up our RPG world!

Enter the Hero

What's an RPG without a hero? Let's add one!

Let's begin with the easiest possible step; loading a character sprite and displaying it on a map tile. Example character-1 uses our Map to display a small map. Run it and you'll see a small room like in Figure 2.25.

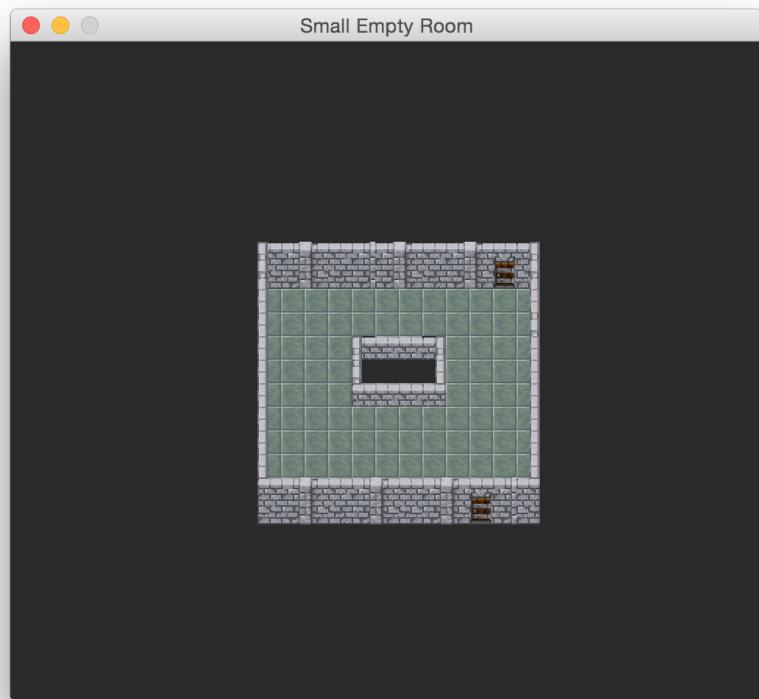


Figure 2.25: A small empty room, waiting for our hero!

In the same folder is a file called `walk_cycle.png` containing an image of the hero and all his animations. (Actually it contains animations for several characters but we'll only be using the hero on the first row). The image file is already included in the manifest, ready to use. We'll ignore animation to begin with and just display a single frame of the hero, standing, facing the camera. We'll start by adding code to the `main.lua` then wrap this in a class when we get it working, much in the same way as we created the `Map` class. Copy the code in Listing 2.36 to get the hero into the map.

```
-- Setup code omitted

gHeroTexture = Texture.Find("walk_cycle.png")
local heroWidth = 16 -- pixels
local heroHeight = 24
gHeroUVs = GenerateUVs(heroWidth,
```

```

        heroHeight,
        gHeroTexture)
gHeroSprite = Sprite>Create()
gHeroSprite:SetTexture(gHeroTexture)
gHeroSprite:SetUVs(unpack(gHeroUVs[9]))

function update()
    gRenderer:Translate(-gMap.mCamX, -gMap.mCamY)
    gMap:Render(gRenderer)
    gRenderer:DrawSprite(gHeroSprite)
end

```

Listing 2.36: Loading the hero. In main.lua.

In Listing 2.36, we create a sprite, set the texture to “walk_cycle.png”, calculate some UVs and set the sprite’s UVs to the 9th frame. This frame is the hero facing the camera, hands to his side.

We use GenerateUVs to split the texture into UV tables, each defining a frame. The hero character, unlike the map tiles, is an odd shape. He has a width of 16 and height of 24 pixels. Fortunately GenerateUVs handles non-square rectangles without any problem.

In the update loop we position the camera so the map is center of the screen. After rendering the map we tell the renderer to draw the hero sprite, which results in Figure 2.26. Rendering the map first and the hero second puts the hero on top of the map. The hero’s position is 0, 0 which is usually the center of the screen, but because we’ve moved the camera it’s now at the bottom right of the screen. Example character-1-solution demonstrates this code working.

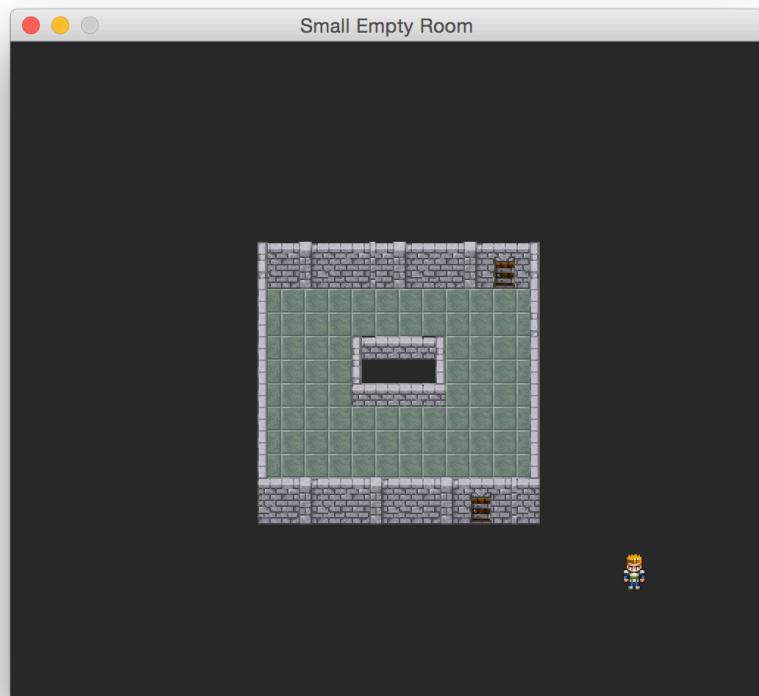


Figure 2.26: The hero appears.

The hero sprite's X and Y pixel position is 0, 0. This puts our character outside the map room. He's floating in black space, so we need to move him. Rather than directly set the hero's pixel position, it would be better if we could tell him to stand on a certain tile.

To move the hero character into the room we'll give him a tile position of X:10, Y:2. This position is in front of the door near the top of the map. To place a character on a tile we must align the bottom of the character sprite with the bottom of the tile.

Let's write some code, starting with variables to store the player's tile position. Copy the code from Listing 2.37 into your main.lua.

```
gHeroTileX = 10  
gHeroTileY = 2
```

Listing 2.37: Give the hero a location on the map. In main.lua.

Setting the hero sprite position equal to the tile pixel position causes the hero sprite to look offset, as shown in Figure 2.27. To correct the alignment, we first need to get the bottom center pixel position of the tile. We'll call the bottom center position of a tile its *foot*. We can then align the bottom of the hero sprite with the tile's foot and he'll appear to be standing on the tile.

Update your Map code with Listing 2.38.

```
function Map:GetTileFoot(x, y)
    return self.mX + (x * self.mTileWidth),
           self.mY - (y * self.mTileHeight) - self.mTileHeight / 2
end
```

Listing 2.38: Getting the foot position of a tile. In Map.lua.

The GetTileFoot function takes the pixel position of the top left corner of the map and offsets it by the tile location to get the center of the tile. Then half the tile's height is subtracted from this position to give the foot of the tile.



Figure 2.27: Aligning the hero with tile's foot.

Code to place the hero in front of the door shown in Listing 2.39. Example character-2 shows the code so far in action.

```
-- code omitted
gHeroTexture = Texture.Find("walk_cycle.png")
local heroWidth = 16 -- pixels
local heroHeight = 24
gHeroUVs = GenerateUVs(heroWidth,
                       heroHeight,
                       gHeroTexture)

gHeroSprite = Sprite>Create()
gHeroSprite:SetTexture(gHeroTexture)
-- 9 is the hero facing forward
```

```

gHeroSprite:SetUVs(unpack(gHeroUVs[9]))
-- 10, 2 is the tile in front of the door
gHeroTileX = 10
gHeroTileY = 2
local x, y = gMap:GetTileFoot(gHeroTileX, gHeroTileY)
gHeroSprite:SetPosition(x,
                        y + heroHeight / 2)

function update()
    gRenderer:Translate(-gMap.mCamX, -gMap.mCamY)
    gMap:Render(gRenderer)
    gRenderer:DrawSprite(gHeroSprite)
end

```

Listing 2.39: Placing the hero in front of the door. In main.lua.

In Listing 2.39 we find the hero texture, generate the UVs, and set the hero sprite position as before. Then we get the foot of tile 10, 2 and position the hero sprite so its horizontal center is aligned with the tile's center and its bottom is aligned with the tile's bottom. Run example character-2 to see the hero correctly positioned as shown in Figure 2.28.

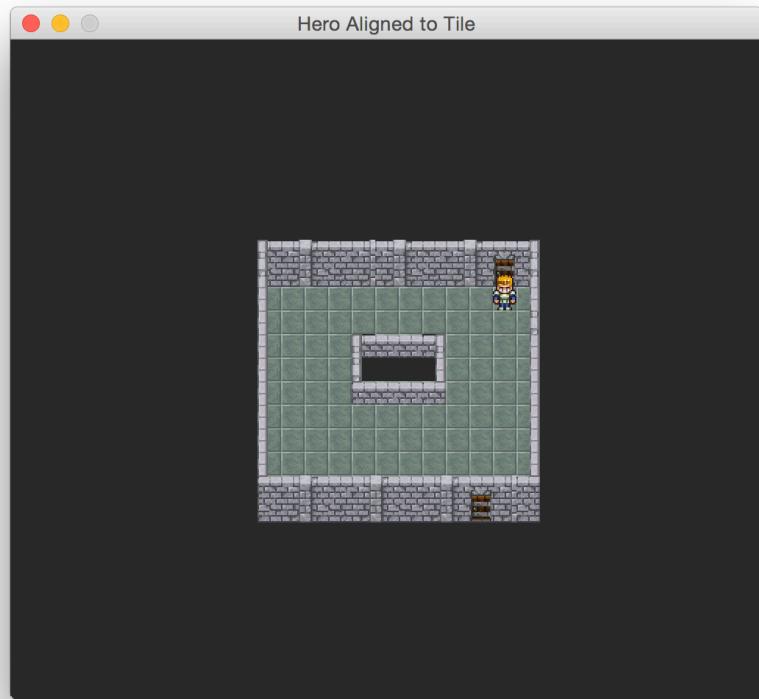


Figure 2.28: Setting the hero's position with tile coordinates.

Experiment by changing the tile coordinates passed into `GetTileFoot` to see the hero move around the map. We'll be adding this kind of simple tile-based movement with the arrow keys next.

A Word About Movement

In tile-based RPG games, like the one we're writing, movement is handled in one of two ways, which I'm going to call *fixed* and *free*.

Fixed movement means that when the character stops they're always aligned to a tile. This is how the early Final Fantasy games work. There's no being half on a tile or half off. Free movement means the character can move anywhere, ignoring the underlying grid. This is commonly used in action RPGs. Our game uses a fixed movement system.

Naive Movement

Let's add simple tile-based movement. When an arrow key is pressed we want the character to jump one tile in that direction. Continue with the code you're writing or use example character-3 as a starting point.

Add a new function to main.lua, called Teleport, as shown in Listing 2.40. This function moves the hero to a specified tile. It calls GetTileFoot to align the hero's sprite position to the tile. The globals gHeroX and gHeroY track the tile the hero is currently on.

```
function Teleport(tileX, tileY, map)
    local x, y = map:GetTileFoot(tileX, tileY)
    gHeroSprite:SetPosition(x,
                           y + heroHeight / 2)
end
-- 10, 2 is the tile in front of the door
gHeroX = 10
gHeroY = 2
Teleport(gHeroX, gHeroY, gMap)
```

Listing 2.40: Adding a simple teleport function. In main.lua.

Every time the player presses an arrow key, we'll call Teleport to move the character 1 tile in that direction. Copy the update loop from Listing 2.41. In the code we use the JustPressed keyboard function, not Held. Try changing the code to use Held instead of JustPressed and you'll quickly discover why! For this type of basic jerky movement, tapping an arrow key works much better than holding it.

```
function update()

    gRenderer:Translate(-gMap.mCamX, -gMap.mCamY)
    gMap:Render(gRenderer)
    gRenderer:DrawSprite(gHeroSprite)

    if Keyboard.JustPressed(KEY_LEFT) then
        gHeroX = gHeroX - 1
        Teleport(gHeroX, gHeroY, gMap)
    elseif Keyboard.JustPressed(KEY_RIGHT) then
        gHeroX = gHeroX + 1
        Teleport(gHeroX, gHeroY, gMap)
    end

    if Keyboard.JustPressed(KEY_UP) then
        gHeroY = gHeroY - 1
        Teleport(gHeroX, gHeroY, gMap)
```

```

elseif Keyboard.JustPressed(KEY_DOWN) then
    gHeroY = gHeroY + 1
    Teleport(gHeroX, gHeroY, gMap)
end

end

```

Listing 2.41: Adding character movement. In main.lua.

Run the code and press the arrow keys to walk the hero around the room. Ok, he can walk through walls and doesn't animate but it's a start! See example character-3-solution for the full working code.

Entity class

Now that we can move the hero around it's time to wrap the movement code into a class. Let's define a class called Entity to represent any kind of map object from a treasure chest to an NPC⁴. A base project is available as example character-4 or you can continue to use your own code.

Create a new lua file called Entity.lua and add it to the manifest. Add the code shown in Listing 2.42.

```

Entity = {}
Entity.__index = Entity
function Entity:Create(def)
    local this =
    {
        mSprite = Sprite.Create(),
        mTexture = Texture.Find(def.texture),
        mHeight = def.height,
        mWidth = def.width,
        mTileX = def.tileX,
        mTileY = def.tileY,
        mStartFrame = def.startFrame,
    }

    this.mSprite:SetTexture(this.mTexture)
    this.mUVs = GenerateUVs(this.mWidth, this.mHeight, this.mTexture)
    setmetatable(this, self)
    this:SetFrame(this.mStartFrame)
    return this

```

⁴NPC, non-playing character. Usually this means any character in the world apart from the player or the members of the party.

```

end

function Entity:SetFrame(frame)
    self.mSprite:SetUVs(unpack(self.mUVs[frame]))
end

```

Listing 2.42: An entity class. In Entity.lua.

Like the Map class, the Entity constructor takes in a def table parameter. These definition tables are great places to store lots of hard coded, magic number data that describes how to make the object.

In the constructor we use the def table to create a sprite, assign the relevant textures, store the width and height of the entity, calculate UVs, and set the sprite's starting frame. The code in the constructor is basically the hero setup code that was in main.lua. The Entity has a new helper function SetFrame which uses the UV data to set the sprite to a certain frame.

Let's use the Entity class to tidy up our main.lua file. See Listing 2.43.

```

Asset.Run("Entity.lua")

-- Setup code omitted

local heroDef =
{
    texture     = "walk_cycle.png",
    width       = 16,
    height      = 24,
    startFrame  = 9,
    tileX       = 10,
    tileY       = 2
}

gHero = Entity>Create(heroDef)

function Teleport(entity, map)
    local x, y = map:GetTileFoot(entity.mTileX, entity.mTileY)
    entity.mSprite:SetPosition(x, y + entity.mHeight / 2)
end
Teleport(gHero, gMap)

```

Listing 2.43: The definition of a hero. In main.lua.

In Listing 2.43 we create a definition table for the hero called heroDef containing all the important data about the hero.

If you're wondering "Why go to the trouble of defining definition tables like this?", well, it makes it easier to add new entities. A new entity can be created by just defining a second def table. For example, orcDef might describe an orc but we don't have to write any additional code; we can just reuse the Entity class.

We use the hero definition table to create an Entity which we store as gHero. All the information about our hero is now packaged up in one place!

The Teleport function has been updated to take in an entity parameter. This means we can teleport any entity, not just the hero. After the Teleport function definition, we call it, passing in the hero to place him on the map.

Next let's update the movement code to use the new Teleport function as shown in Listing 2.44.

```
function update()

    gRenderer:Translate(-gMap.mCamX, -gMap.mCamY)
    gMap:Render(gRenderer)
    gRenderer:DrawSprite(gHero.mSprite)

    if Keyboard.JustPressed(KEY_LEFT) then
        gHero.mTileX = gHero.mTileX - 1
        Teleport(gHero, gMap)
    elseif Keyboard.JustPressed(KEY_RIGHT) then
        gHero.mTileX = gHero.mTileX + 1
        Teleport(gHero, gMap)
    end

    if Keyboard.JustPressed(KEY_UP) then
        gHero.mTileY = gHero.mTileY - 1
        Teleport(gHero, gMap)
    elseif Keyboard.JustPressed(KEY_DOWN) then
        gHero.mTileY = gHero.mTileY + 1
        Teleport(gHero, gMap)
    end

end
```

Listing 2.44: Entity moving code. In Entity.lua.

Run the code and you'll be able to move the character around as before (but the code is a lot tidier and reusable). See character-4-solution for the finished example.

Smooth Movement

When we move the character he jumps from tile to tile. It would be nicer if he smoothly moved from one tile to the next.

To implement smooth movement we'll use two Lua helper classes that were introduced at the start of the book, Tween and StateMachine. The state machine is made of two states, "moving" and "waiting," as shown in Figure 2.29. Tween controls the smooth movement from one tile to the next. Input from the user is only checked during the "waiting" state.

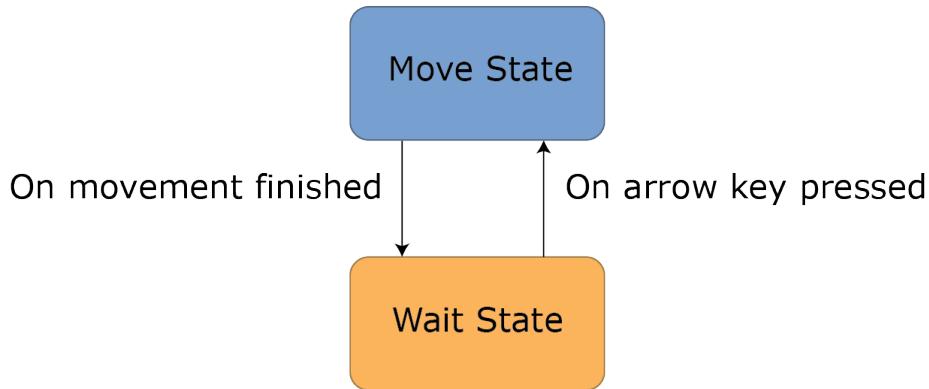


Figure 2.29: A state machine to control entity movement.

Example character-5 contains what we've done so far and the two helper classes Tween.lua and StateMachine.lua (these have also been added to the manifest). To continue using your own code, copy over these files and update your manifest.

Call Asset.Run on both the Tween and StateMachine files at the top of the main.lua file. This is shown below in Listing 2.45.

```
-- code omitted
LoadLibrary("Keyboard")

Asset.Run("StateMachine.lua")
Asset.Run("Tween.lua")

Asset.Run("Map.lua")
-- code omitted
```

Listing 2.45: Running the code for the statemachine and tween files. In main.lua.

To use the state machine we'll create a number of new classes, but don't worry, they tend to be short and help keep the code nicely segregated.

Let's begin to add smooth movement by defining a new gHero definition. This gHero is made from an Entity and StateMachine, shown in Listing 2.46. The code won't compile right now as it's referencing classes we haven't made, but it shows how the overall system is going to work. Update your main file so it appears as in Listing 2.46.

```
-- gHero = Entity:Create(heroDef) <- remove this
local gHero
gHero =
{
    mEntity = Entity:Create(heroDef),
    mController = StateMachine:Create
    {
        ['wait'] = function() return WaitState:Create(gHero, gMap) end,
        ['move'] = function() return MoveState:Create(gHero, gMap) end,
    },
}
gHero.mController:Change("wait")
```

Listing 2.46: Adding states to the hero. In main.lua.

The mEntity represents the hero on the map. The mController controls what the hero is doing.

The mController is a state machine with a WaitState and a MoveState. The WaitState listens for user input and the MoveState acts on that input as shown in Figure 2.29. It's easy to imagine swapping out the state machine for a different one that doesn't listen for the user controls but acts on its own - randomly moving around for instance.

The state machine uses two state classes, WaitState and MoveState. We need to implement both of these classes but let's start with the WaitState. Create a new file called WaitState.lua and add it to the manifest. The StateMachine class requires each state to have four functions: Enter, Exit, Render and Update. Copy the code from Listing 2.47 to the WaitState file.

```
WaitState = { mName = "wait" }
WaitState.__index = WaitState
function WaitState:Create(character, map)
    local this =
    {
        mCharacter = character,
        mMap = map,
        mEntity = character.mEntity,
        mController = character.mController
    }

    setmetatable(this, self)
```

```

        return this
    end

    function WaitState:Enter(data)
        -- Reset to default frame
        self.mEntity:SetFrame(self.mEntity.mStartFrame)
    end

    function WaitState:Render(renderer) end
    function WaitState:Exit() end

    function WaitState:Update(dt)
        if Keyboard.Held(KEY_LEFT) then
            self.mController:Change("move", {x = -1, y = 0})
        elseif Keyboard.Held(KEY_RIGHT) then
            self.mController:Change("move", {x = 1, y = 0})
        elseif Keyboard.Held(KEY_UP) then
            self.mController:Change("move", {x = 0, y = -1})
        elseif Keyboard.Held(KEY_DOWN) then
            self.mController:Change("move", {x = 0, y = 1})
        end
    end

```

Listing 2.47: The wait state. In WaitState.lua.

The WaitState, shown in Listing 2.47, really does only one thing; it waits until an arrow key has been pressed and then changes the state to the MoveState, passing along the direction the player wants to move.

The OnEnter function for the WaitState resets the entity frame back to its original starting frame. This means if we tell a character to wait and they're mid-run, they don't stay stuck mid-run; instead they revert to the default standing frame.

In WaitState:Update a table with an x and y field is passed into the Change function when the state changes. This data tells the MoveState which direction we want the player to move. A figure showing the movement offsets can be seen in Figure 2.30.

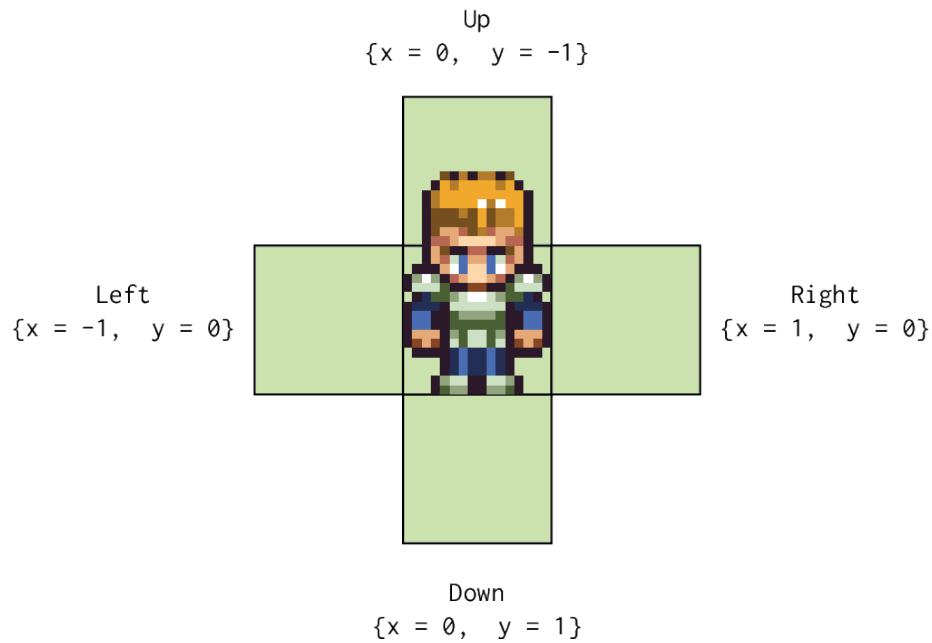


Figure 2.30: Movement offsets and directions for a one tile move.

The Keyboard input uses the Held function again. This lets us hold a key down to keep moving the character in one direction.

On very first line of the WaitState file there's a field mName with a value of "wait". This gives the class a name to use as an id. All objects created from WaitState have mName set to "wait". This value is used to determine the type of a state and helps when debugging.

Let's implement the MoveState and get a little closer to silky smooth movement. Copy Listing 2.48 into your MoveState.lua file.

```
MoveState = { mName = "move" }
MoveState.__index = MoveState
function MoveState:Create(character, map)
    local this =
    {
        mCharacter = character,
        mMap = map,
        mTileWidth = map.mTileWidth,
        mEntity = character.mEntity,
        mController = character.mController,
        mMoveX = 0,
```

```

        mMoveY = 0,
        mTween = Tween:Create(0, 0, 1),
        mMoveSpeed = 0.3
    }

    setmetatable(this, self)
    return this
end

function MoveState:Enter(data)
    self.mMoveX = data.x
    self.mMoveY = data.y
    local pixelPos = self.mEntity.mSprite:GetPosition()
    self.mPixelX = pixelPos:X()
    self.mPixelY = pixelPos:Y()
    self.mTween = Tween:Create(0, self.mTileWidth, self.mMoveSpeed)
end

function MoveState:Exit()

    self.mEntity.mTileX = self.mEntity.mTileX + self.mMoveX
    self.mEntity.mTileY = self.mEntity.mTileY + self.mMoveY
    Teleport(self.mEntity, self.mMap)

end
function MoveState:Render(renderer) end

function MoveState:Update(dt)
    self.mTween:Update(dt)

    local value = self.mTween:Value()
    local x = self.mPixelX + (value * self.mMoveX)
    local y = self.mPixelY - (value * self.mMoveY)
    self.mEntity.mX = x
    self.mEntity.mY = y
    self.mEntity.mSprite:SetPosition(x, y)

    if self.mTween:IsFinished() then
        self.mController:Change("wait")
    end
end

```

Listing 2.48: The move state. In MoveState.lua.

The MoveState has a bit more going on than the WaitState. As before, the state name

is added to the class definition on the first line.

The constructor stores useful data needed to move entities from tile to tile. The `mMoveSpeed` field is the time in seconds to perform the move. This is set to 0.3 seconds; we don't want the player waiting too long. The `mMoveX` and `mMoveY` variables store the direction to *move* the entity. Their values are only ever 0 or 1 and only one is ever set a time; we don't support diagonal movement.

The important setup code happens in the `OnEnter` function where the `mTween` is set to go from 0 to `mTileWidth` in `mMoveSpeed` seconds.

In the `Update` function the `mTween` is updated and the current value of the tween is used to position the entity. Once the `mTween` is finished we change state back to the `WaitState`.

Moving from the `MoveState` to the `WaitState` causes the state machine to call the `MoveState:Exit` function. In the exit function we update the entity tile position and teleport to that position.

The tween is the driving force behind the smooth movement. We know how wide each tile is, so to move left all we need to do is add on 16 pixels while to move right we subtract 16 pixels. It's the same for moving up and down but we add and subtract on the Y axis.

Over a few frames the tween goes from 0 to 16, the width (or height) of our tile in pixels. The entity has its X, Y pixel position calculated by multiplying the tween's current value by `mMoveX` for the X position and `mMoveY` for the Y position. If we're moving left or right `mMoveY` is 0 so the pixel position for the up and down is zero. If `mMoveX` is -1 the tween's value becomes negative, causing the entity to move left. If it's +1 then it moves the entity right. (If this is a little unclear try building your own small program based on this one to see what's going on!)

Note that `SetPosition` is called without `math.floor`, so there's some sub-pixel movement occurring. Sub-pixel movement is fine for the entity because it isn't part of a tilemap and won't cause artifacts.

Let's pull everything together in the `main.lua` file to get a working program. Make your `main.lua` file look like Listing 2.49. Remember, you need to include your `MoveState` and `WaitState` in the manifest file.

```
Asset.Run("StateMachine.lua")
Asset.Run('MoveState.lua') -- Remember to add
Asset.Run('WaitState.lua') -- these!
Asset.Run("Tween.lua")

-- Setup code omitted

local heroDef =
{
    texture = "walk_cycle.png",
```

```

        width = 16,
        height = 24,
        startFrame = 9,
        tileX = 10,
        tileY = 2
    }

local gHero
gHero =
{
    mEntity = Entity:Create(heroDef),
    Init =
    function(self)
        self.mController = StateMachine:Create
        {
            [ 'wait' ] = function() return self.mWaitState end,
            [ 'move' ] = function() return self.mMoveState end,
        }
        self.mWaitState = WaitState:Create(self, gMap)
        self.mMoveState = MoveState:Create(self, gMap)
        self.mController:Change("wait")
    end
}
--gHero.mController:Change("wait") <- remove this
gHero:Init()

function Teleport(entity, map)
    local x, y = map:GetTileFoot(entity.mTileX, entity.mTileY)
    entity.mSprite:SetPosition(x, y + entity.mHeight / 2)
end
Teleport(gHero.mEntity, gMap)

function update()

    local dt = GetDeltaTime()

    gRenderer:Translate(-gMap.mCamX, -gMap.mCamY)
    gMap:Render(gRenderer)
    gRenderer:DrawSprite(gHero.mEntity.mSprite)

    gHero.mController:Update(dt)
end

```

Listing 2.49: Using the Move and Wait states for smooth movement. In main.lua.

Listing 2.49 shows the state machine and smooth movement in action. A lot of this code we've already seen. The update loop is much smaller now because the meat of the update is handled by the hero's controller. Run the code and you'll see the character slide nicely from one tile to the next. The full code is provided in [character-5-solution](#).

There's one more quick change we can add to make it feel more like a game, and that's camera tracking. Just below the `GetDeltaTime` lines, in `main.lua`, add the code in *Listing 2.50*. It uses `math.floor` because we only ever want to move the camera whole pixels.

```
local playerPos = gHero.mEntity.mSprite:GetPosition()
gMap.mCamX = math.floor(playerPos:X())
gMap.mCamY = math.floor(playerPos:Y())
```

Listing 2.50: Set the camera to follow the player. In main.lua.

Run this code and you'll see the camera follow the player around!

Hopefully getting smooth movement wasn't too scary! Next we're going to extend the state machine to add animation.

Animation

We animate the character by changing the sprite's frame over time, a little like a flip book. Example `character-6` contains the code so far. To make the character walk around we need four sets of walking animations: up, down, left and right. We store each animation set as a table of frames indices. *Listing 2.51* shows the animation data for the hero character. For example, `mAnimUp` is made of four numbers (1, 2, 3, 4) which index the UV data for the first, second, third and fourth frames. Run these frames as a sequence and the character will appear to be walking north.

Note that in *Listing 2.51* we've modified how the states are created and entered. The states are now stored in the `gHero` table as `mWaitState` and `mMoveState` and the state machine references these fields.

```
-- code omitted
gHero =
{
    mAnimUp = {1, 2, 3, 4},
    mAnimRight = {5, 6, 7, 8},
    mAnimDown = {9, 10, 11, 12},
    mAnimLeft = {13, 14, 15, 16},
    mEntity = Entity>Create(heroDef),
    Init =
        function(self)
            self.mController = StateMachine>Create
            {
```

```

        [ 'wait' ] = function() return self.mWaitState end,
        [ 'move' ] = function() return self.mMoveState end,
    }
    self.mWaitState = WaitState:Create(self, gMap)
    self.mMoveState = MoveState:Create(self, gMap)
    self.mController:Change("wait")
end
}
gHero:Init()

```

Listing 2.51: The frames to animate the player walking. In main.lua.

The MoveState moves the character, so this is where we need to trigger the relevant walk animation.

To make handling sprite animation easier let's write an Animation class. Create a new file called Animation.lua, add it to the manifest, and run it at the top of the main.lua (See Listing 2.52).

```

-- code omitted
Asset.Run("Animation.lua")
Asset.Run("Map.lua")
Asset.Run("Util.lua")
-- code omitted

```

Listing 2.52: Adding the Animation class code. In main.lua.

Add the following, in Listing 2.53, to Animation.lua.

```

Animation = {}
Animation.__index = Animation
function Animation:Create(frames, loop, spf)

    if loop == nil then
        loop = true
    end

    local this =
    {
        mFrames = frames or { 1 },
        mIndex = 1,
        mSPF = spf or 0.12,
        mTime = 0,
        mLoop = loop
    }
    setmetatable(this, Animation)
    return this
end

```

```

}

setmetatable(this, self)
return this
end

function Animation:Update(dt)
self.mTime = self.mTime + dt

if self.mTime >= self.mSPF then

    self.mIndex = self.mIndex + 1
    self.mTime = 0

    if self.mIndex > #self.mFrames then

        if self.mLoop then
            self.mIndex = 1
        else
            self.mIndex = #self.mFrames
        end

    end
end
end

function Animation:SetFrames(frames)
    self.mFrames = frames
    self.mIndex = math.min(self.mIndex, #self.mFrames)
end

function Animation:Frame()
    return self.mFrames[self.mIndex]
end

function Animation:IsFinished()
    return self.mLoop == false
        and self.mIndex == #self.mFrames
end

```

Listing 2.53: An animation class for moving through frames over time. In `Animation.lua`.

The Animation constructor takes in three parameters: frames, loop and spf. The frames is a table of numbers, each indexing a frame. The loop is a boolean flag indicating whether the animation should loop or not.

ing if the animation should repeat forever. If no loop flag is passed in, it's set to true by default. The spf is the seconds per frame. It's used to advance the frame index after spf seconds have passed.

The frames are stored in `mFrames` in the `this` table. If no frames are passed in, a list with a single entry of 1 is used. The next field, `mIndex`, is an index into `mFrames` indicating the current frame to display. The `mIndex` starts at 1 which points to the first frame of the animation. The `spf` is assigned to `mSPF` or defaults to 0.12 if nil. This means that, by default, each 0.12 seconds that pass, a new frame is displayed. Finally the `mTime` field counts up the passing of time to compare against `mSPF`.

The `Animation:Update` function increments `mTime` by the delta time. If `mTime` is greater than or equal to the seconds per frame, the `mTime` is reset to 0 and the frame index, `mIndex`, is increased. This is the piece of code that makes our animation play.

At some point the frame index, `mIndex`, reaches the end of the frame list, `mFrames`. If the animation is looping, the index resets to 1, otherwise the index remains pointing to the last frame.

The `SetFrame` is a helper function to change the frame list for an animation. This means we can swap the animation being played without recreating the object. The internal `mFrames` table is replaced and the `mIndex` is clamped, so it isn't greater than the new length of `mFrames`.

The `Animation:Frame` function returns the current frame of animation. The `IsFinished` function returns true only if the animation is *not* looping and the index is pointing to the last frame.

With `Animation` defined, let's use it in `MoveState`! Update your `MoveState` code to look like Listing 2.54.

```
-- code omitted
function MoveState:Create(character, map)
    local this =
    {
        -- code omitted
    }
    this.mAnim = Animation>Create({ this.mEntity.mStartFrame })

    setmetatable(this, self)
    return this
end

function MoveState:Enter(data)

    local frames = nil

    if data.x == -1 then
        frames = self.mCharacter.mAnimLeft
```

```

elseif data.x == 1 then
    frames = self.mCharacter.mAnimRight
elseif data.y == -1 then
    frames = self.mCharacter.mAnimUp
elseif data.y == 1 then
    frames = self.mCharacter.mAnimDown
end

self.mAnim:SetFrames(frames)

-- code omitted
end

-- code omitted

function MoveState:Update(dt)

    self.mAnim:Update(dt)
    self.mEntity:SetFrame(self.mAnim:Frame())
    -- code omitted
end

```

Listing 2.54: Adding animation to the movestate. In MoveState.lua.

We've added an Animation object to our MoveState class. By default it's initialized with a single frame; the default frame. This means if we don't have animation data the default frame continues to display while the entity moves.

The MoveState:Enter function checks the direction the character is moving and correctly sets the animation frames. The character only ever moves one tile in one direction. We use a chain of if-statements to work out the direction the character is moving.

The Update loop updates the animation, asks the animation for the current frame, and sets the entity to that frame.

We're nearly ready to run this code, but first the WaitState needs a little modification. Leaving the MoveState sometimes leaves the character on a random animation frame, which looks odd, so in WaitState we'll return to the default frame after a short pause. Update WaitState to look like Listing 2.55.

```

function WaitState:Create(character, map)
    local this =
    {
        -- omitted code
        mController = character.mController,
        mFrameResetSpeed = 0.05,
    }

```

```

        mFrameCount = 0
    }
    -- omitted code
end

-- omitted code

function WaitState:Enter(data)
    self.mFrameCount = 0
end

function WaitState:Update(dt)

    -- If we're in the wait state for a few frames, reset the frame to
    -- the starting frame.
    if self.mFrameCount ~= -1 then
        self.mFrameCount = self.mFrameCount + dt
        if self.mFrameCount >= self.mFrameResetSpeed then
            self.mFrameCount = -1
            self.mEntity:SetFrame(self.mEntity.mStartFrame)
        end
    end
    -- omitted code
end

```

Listing 2.55: Updating the WaitState to an idle frame. In WaitState.lua.

We've added two new variables: `mFrameResetSpeed` which is how long the character is in `WaitState` before setting the sprite to the default frame, and `mFrameCount` which is used to count up the time and trigger the reset.

In `WaitState:Update` the `if`-statement checks if `mFrameResetSpeed` seconds have passed and then sets the entity sprite to the default frame and sets the `mFrameCount` to `-1` which stops the check. All this has the effect that when moving our hero character around, if we stop pressing the arrow keys he'll face the camera. The `mFrameResetSpeed` is a pretty small number because I want the reset to be almost instantaneous. In the `Enter` function the `mFrameCount` is reset to `0` ensuring the code works on subsequent re-entries into the `WaitState`.

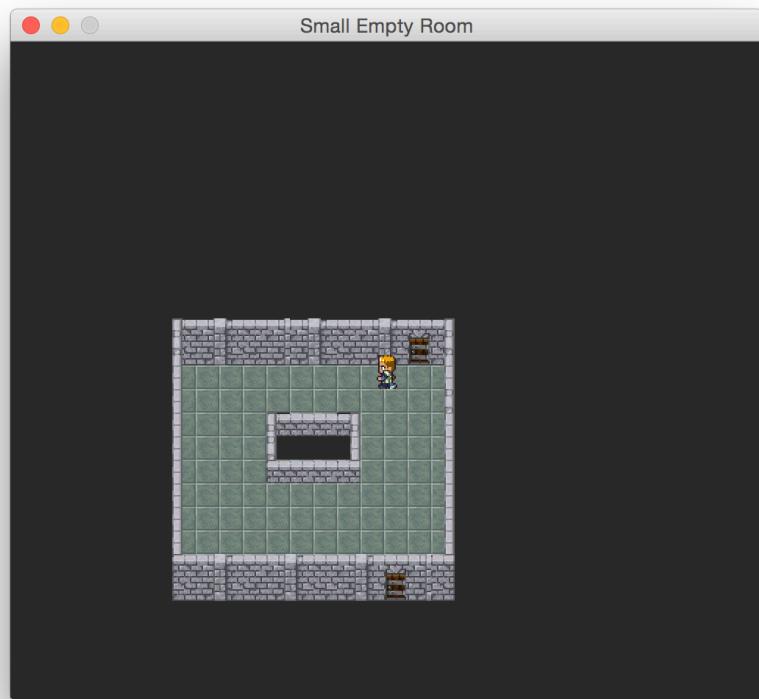


Figure 2.31: The character animating as it moves from one tile to the next.

Run the code and you'll see the hero character move smoothly from tile to tile, playing the appropriate walk animation as shown in Figure 2.31. Example character-6-solution contains the code so far.

The map navigation looks good but the character still doesn't feel part of the world. We'll start fixing that by adding simple collision detection.

Simple Collision Detection

The maps we've created so far have no collision data. To add collision information, we're going to revisit the Tiled editor. Then we'll extend the Map class to store collision data and update MoveState to do collision checks. With this done, our character will respect the walls in our world!

Throughout this section we've been using a map called `small_room`. In example character-7 there's a new `small_room.tmx` file. Open this file in Tiled.

Let's add a new tile layer called "collision". Go to the menu bar and select Layer > Add Tile Layer, as in Figure 2.32.

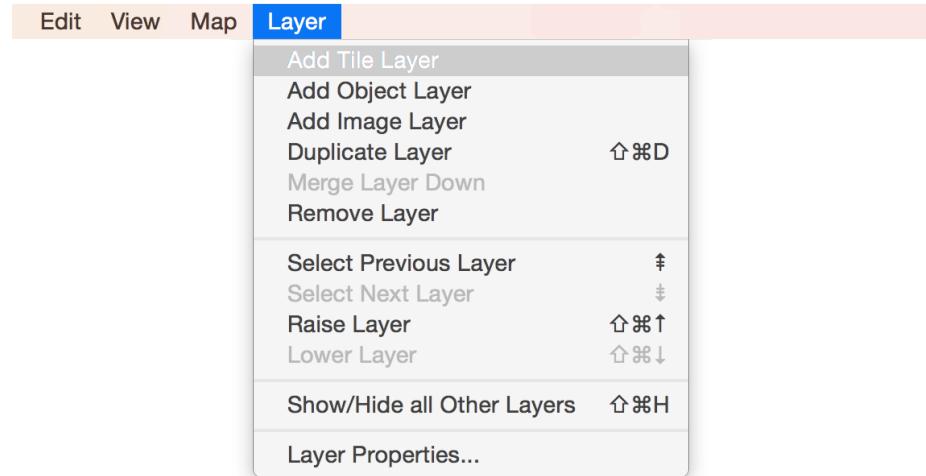


Figure 2.32: Adding a new Tile Layer in Tiled.

Name the layer "collision". The character-7 folder contains an image called `collision_graphic.png`. We'll use this image as a tileset for marking collision areas. Its first tile is red. The rest are white. The first red tile indicates a *non-walkable* area in our tiled map.

Select Map > New Tile Set, then choose the `collision_graphic.png` in the dialog box. Set the tile size to 16 by 16 and press ok, as shown in Figure 2.33. The tileset's window now has two tabs, one called `rpg_indoor` that the map was built from and one called `collision_graphic` that we've just added.

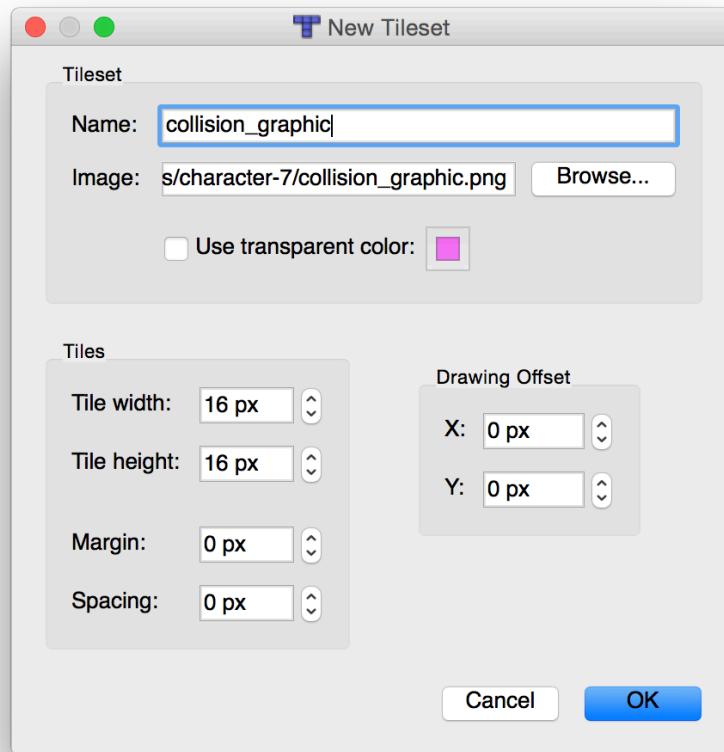


Figure 2.33: Adding a tileset to represent collision information.

In the Layers window select the layer called “collision”. At the very top of the Layers window there’s an opacity slider. Slide the opacity to 50%. This makes the collision layer semi-transparent so it overlays the layer below and makes placing collision easier. In the Tilesets window select the collision_graphic tab and choose the red tile, as shown in Figure 2.34.

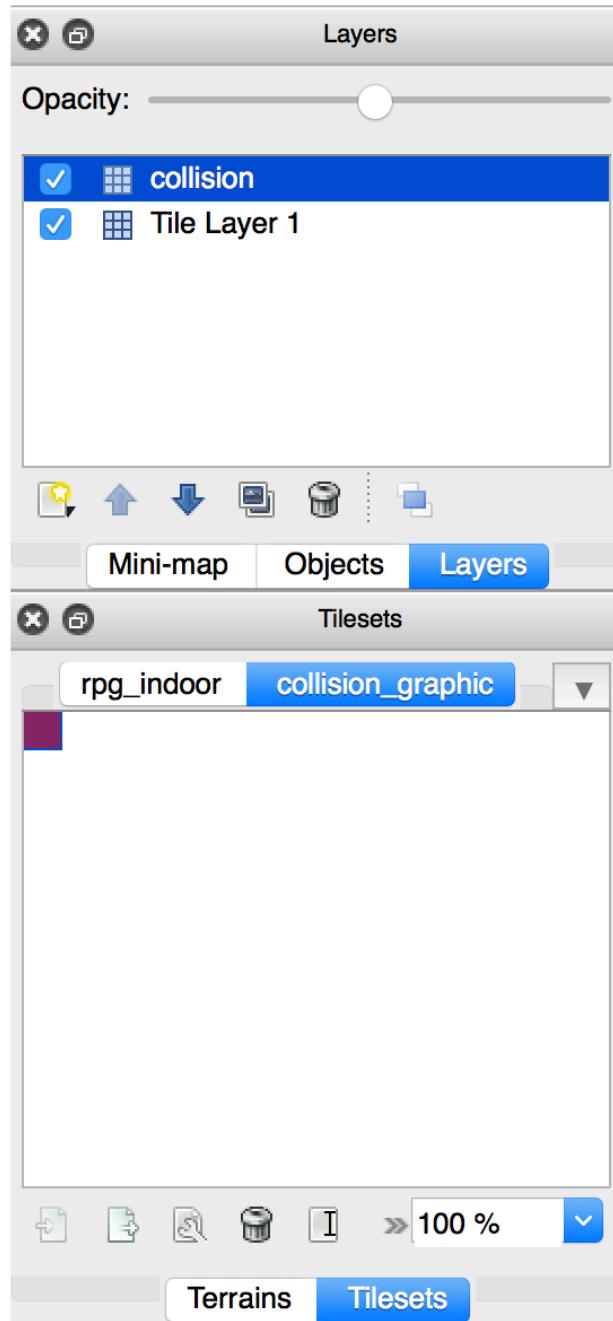


Figure 2.34: Selecting the collision tile from the collision tileset in Tiled. Notice the opacity for the collision layer is set to 50%.

Paint the red tile over all tiles that block the player. Once done the collision layer ends up looking like Figure 2.35. Save your tmx file.

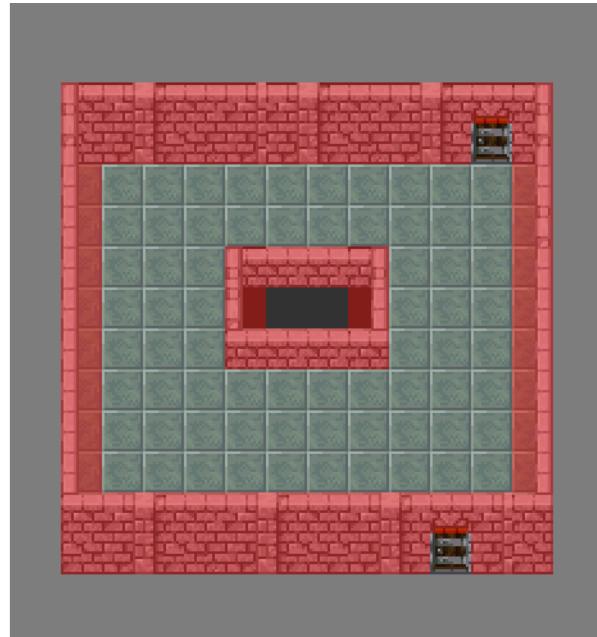


Figure 2.35: The map and collision layer shown together in the Tiled Editor. Red tiles are areas the player cannot walk.

Export the map to lua and overwrite the file `small_room.lua` (either in your project or in example character-7). Open the lua file and wrap the map def in a function called `CreateMap1`, as shown in Listing 2.56.

```
function CreateMap1()
    return {
        version = "1.1",
        -- omitted code
    }
end
```

Listing 2.56: Wrapping the exported Lua map in a function.

Run the code to make sure everything works. Look inside `small_room.lua` and you'll see there's new data about the collision tileset. Most of this data we can ignore. In the layers subtable there's an entry for the collision layer; we'll use this to block the character's movement.

The Map needs to know that the red tiles are blocking tiles. We should be able to ask the map “Is this tile blocking?” and get a true or false reply. The first layer of our map is purely graphical and the second layer describes the collision. To make things simple we assume that all layers in our Map are formed from a graphics layer followed by the collision layer.

To do collision checks, the Map needs to store the id of the red blocking tile. If you look at the map definition in `small_room.lua` you’ll see that all tilesets used in a map are named and that each stores the id of its very first tile. We’ll search the `tilesets` table for one named `collision_graphic` and store the id of its first tile. This id is the red blocking tile. We use this id to check if a tile is blocking. Copy the code shown in Listing 2.57 to find the blocking tile. As we edit the map, the layer order and tile ids may change; therefore it’s better to search for the blocking tile id than have to worry about it becoming invalid each time we change the map.

```
Map = {}
Map.__index = Map
function Map:Create(mapDef)

    -- code omitted
    this.mUVs = GenerateUVs(mapDef.tilesets[1].tilewidth,
                           mapDef.tilesets[1].tileheight,
                           this.mTextureAtlas)

    -- Assign blocking tile id
    for _, v in ipairs(mapDef.tilesets) do
        if v.name == "collision_graphic" then
            this.mBlockingTile = v.firstgid
        end
    end
    assert(this.mBlockingTile)

    setmetatable(this, self)

    return this
end
```

Listing 2.57: Searching for the blocking tile id. In `Map.lua`.

In Listing 2.57 there’s an assert to make sure we always find the blocking tile id. An assert is an instruction to the program saying “Make sure this is always true.” If it’s not true the program is stopped and the user is told which assert failed. Players should never see an assert, but they’re useful during development to catch stupid mistakes early on.

Next let’s add a function that, given an X, Y tile position, tells us if that position is blocked or not. Ideally we’d use the `GetTile` function but it only returns data from the

graphics layer, not the collision layer. Let's extend the GetTile function to take in an extra layer parameter that controls which layer's data it returns. Update your code so it matches Listing 2.58.

```
function Map:GetTile(x, y, layer)
    local layer = layer or 1
    local tiles = self.mMapDef.layers[layer].data
    x = x + 1 -- change from 1 -> rowsize
               -- to          0 -> rowsize - 1
    return tiles[x + y * self.mWidth]
end

function Map:IsBlocked(layer, tileX, tileY)
    -- Collision layer should always be 1 above the official layer
    local tile = self:GetTile(tileX, tileY, layer + 1)
    return tile == self.mBlockingTile
end
```

Listing 2.58: Adding a test for blocking tiles. In Map.lua.

Listing 2.58 shows the GetTile function taking in the extra, optional parameter, layer. It defaults to 1 if nothing is passed in. Optional parameters in Lua are commonly implemented with the syntax `a = a or default_value`. The first line of the function means, *if layer is nil, set it to 1*. If GetTile is called without specifying the layer parameter then it works as it did before.

In Tiled there are two layers, the base graphics layer and the collision layer. In our game we consider the base layer and collision to be two sections of a single map layer. We want to be able to say "On layer 1 is tile x:1 y:2 blocking?" without having to calculate the offsets manually. We'll use the Map:IsBlocked function to ask if a tile is blocking on a certain layer and have it deal with checking the collision section.

The Map:IsBlocked function takes in a layer and tile coordinates. It uses GetTile to get collision data at the coordinates by indexing the collision section. The collision section is one above the base layer so to access it we +1 to the base layer. It returns true if the tile at that coordinate equals the `mBlockingTile` id; otherwise it returns false.

Let's use the Map:IsBlocked function to stop the player from walking on blocking tiles in the MoveState class. Make the changes at the end of MoveState:Enter as shown in Listing 2.59.

```
function MoveState:Enter(data)

    -- code omitted

    local targetX = self.mEntity.mTileX + data.x
```

```

local targetY = self.mEntity.mTileY + data.y
if self.mMap:IsBlocked(1, targetX, targetY) then
    self.mMoveX = 0
    self.mMoveY = 0
    self.mEntity:SetFrame(self.mAnim:Frame())
    self.mController:Change("wait")
end
end

```

Listing 2.59: Adding a test for blocking tiles. In MoveState.lua.

The code in Listing 2.59 checks whether the tile the player wants to move to is blocked. If it is a blocking tile then the state is changed back to the WaitState and the player sprite is updated to face the direction it tried to move. The `mMoveX` and `mMoveY` fields are reset to 0 to stop `MoveState:Exit` teleporting the player onto the blocking tile.

This finishes our collision code. Run the code and test out the collision detection. You'll find the hero can no longer walk through walls! We've built a nice pipeline where we can make a map, add collisions, and quickly test them in the game.

The collision layer we've introduced also has other uses that we haven't explored. The collision layer could contain other tile types such as a water tile you can only pass if you have flippers, or a flying character that can go through air tiles. (In fact we'll adding some custom collision tiles later in the book to help us deal with monster encounters!)

Example character-7-solution has the complete collision detection code. Next we'll start using maps made of multiple layers.

Layers

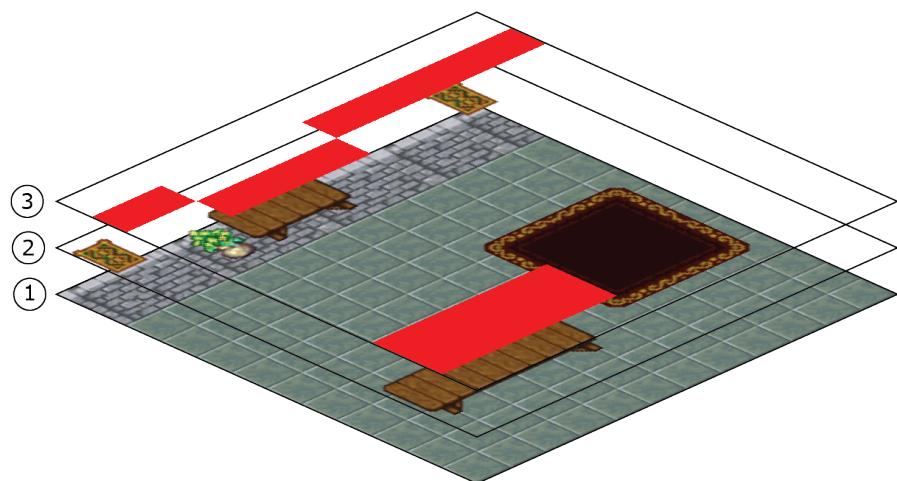
Each layer of our map is made up of two sections, the base graphics and the collision. To add an additional map layer, we just add another graphics and collision layer. Layer 1 is rendered first, then layer 2, then maybe the character, then 3 and so on. Layers help give the scene a sense of depth.

Decorating Our Layer

Before we start stacking up layers, we're going to add an extra section to our map layer definition.

Each layer will have three parts: the base tiles, decoration items, and then the collision. Together these three sections make a single *map layer*. The obvious first question is "What are decoration items?". Well, imagine things like carpet, or a banner on a wall. These items make the map more interesting but they don't form a new layer; they just decorate the base tiles beneath.

Look at the tile palette for the `small_room.tmx` and you'll see things like books, banners, potted plants and so on. These items have a transparent background making it easy to place them on top of the base graphics layer to add detail to a map. You can see how the map layer fits together in Figure 2.36.



Combined Layer

Figure 2.36: The three components of a layer.

As the game creator we decide our game conventions. To simplify our maps we've created the convention that each layer is made from three sections: the base tiles, decoration, and collision. Any of these sections may be blank but they must be present in the correct order. All our maps are expected to be in laid out in this manner. This is our final map layer format!

Open character-8 and you'll see it has a new small_room.tmx map with one layer of three sections. The new map is shown rendered out in Figure 2.37. The decoration section includes some tapestries, a small section of carpet, and a jar. This Tiled map has been exported as small_room.lua and is ready to use. Use example character-8 as a base for this section or copy small_room.lua into your current code.

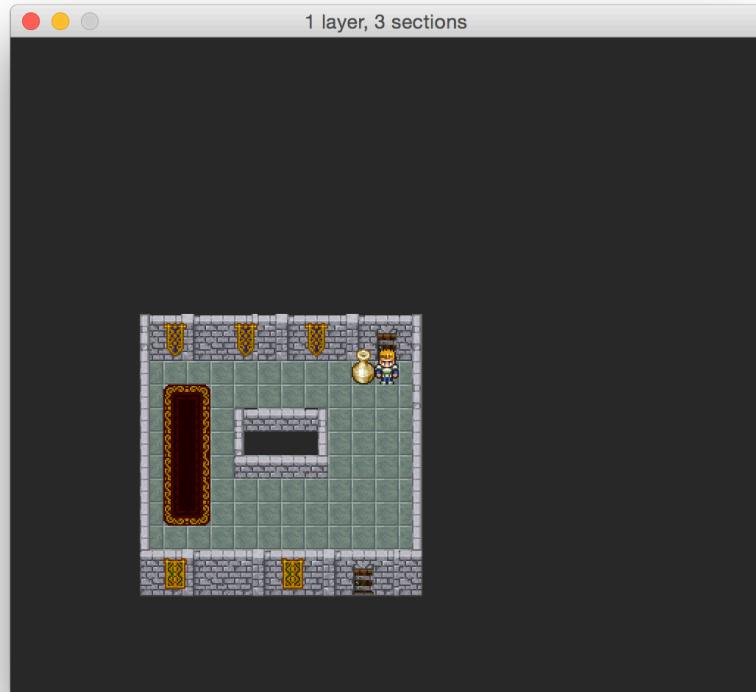


Figure 2.37: The map from example character-8, one layer made from three sections.

Open the Tiled .tmx file and you'll see the collision layer is above the decoration layer. This means we need to update the code to handle the new collision layer offset. Adjust the collision offset in the Map:IsBlocked function from 1 to 2 as shown in Listing 2.60.

```
function Map:IsBlocked(layer, tileX, tileY)
    local tile = self:GetTile(tileX, tileY, layer + 2)
    return tile == self.mBlockingTile
end
```

Listing 2.60: Updating the offset for the collision section. In Map.lua.

Listing 2.60 fixes our collision checks to work with new map layer format.

Next let's modify our render code to draw out the decoration layer. After drawing a base tile, we check if there's a decoration tile and draw that on top. Make your Map:Render function look like Listing 2.61.

```
function Map:Render(renderer)

    -- Code omitted
    for j = tileTop, tileBottom do
        for i = tileLeft, tileRight do

            -- Code omitted

            renderer:DrawSprite(self.mTileSprite)

            -- The second section of layer is always the decoration.
            tile = self:GetTile(i, j, 2)

            -- If the decoration tile exists
            if tile > 0 then
                uvs = self.mUVs[tile]
                self.mTileSprite:SetUVs(unpack(uvs))
                renderer:DrawSprite(self.mTileSprite)
            end

        end
    end
end
```

Listing 2.61: Drawing out tiles with decorations. In Map.lua.

In Listing 2.61 after the base tile is drawn we get the id of the decoration tile at the same position. If there's no tile on the decoration layer it equals 0. If there's a tile id of

0 we draw nothing. If the id is greater than 0 then we look up the decoration tile and draw it on top of the base tile.

Run the code to see the decorated map in action, as shown in Figure 2.37. You can also verify collision is still working as expected. The code so far is available in example character-8-solution.

Now that we've defined exactly what a layer contains and written the code to render one, we can begin to introduce *multiple* layers.

Multiple Layers

Multiple layers allow for more interesting maps and can help collision appear to be more precise.

You may wonder why each layer needs collision information; imagine a bridge over an alleyway. The alleyway is layer 1 and the bridge above it is layer 2. If the player is in the alley they don't care about collision with items on the bridge directly above. If the player's on the bridge then they only care about collision on the bridge layer.

With a little up-front planning we're able to build complex multi-layered maps. All the example maps we'll create are simple but the flexibility exists in the map format if you want to try something more ambitious!

Example character-9 contains a new `small_room.tmx` file. It's similar to the previous example but it's slightly larger and has an extra layer. Figure 2.38 shows the first and second layer side by side and then combined. Only the first layer has collision information. This collision data lets the player walk right up to the edge of the walls which feels far more natural than in previous examples.

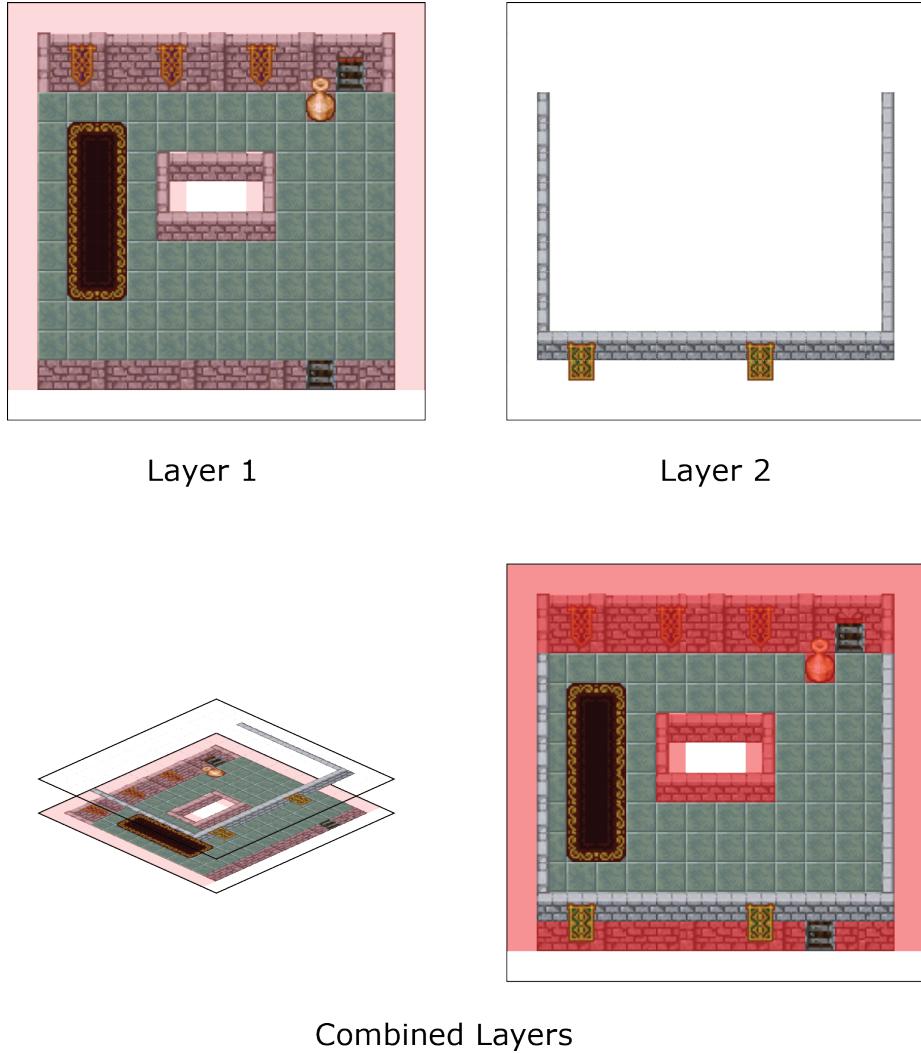


Figure 2.38: The two layers combined to make a single map. The collision section for the second layer is blank.

Having two layers lets the character move behind scenery, giving the map a sense of depth that we couldn't convey with just one layer. The hero is standing on the first layer. To render this map we need to draw the first layer, *then* the hero and then the second layer. This render order lets the hero walk behind the walls at the bottom of the map that were previously blocked off.

Figure 2.39 shows the structure of a two-layer map. Each of our layers is made of three sections: base tiles, decoration, and collision. This means that when we render the

second layer we need to start at index 4, which is the base tile section of our second layer.

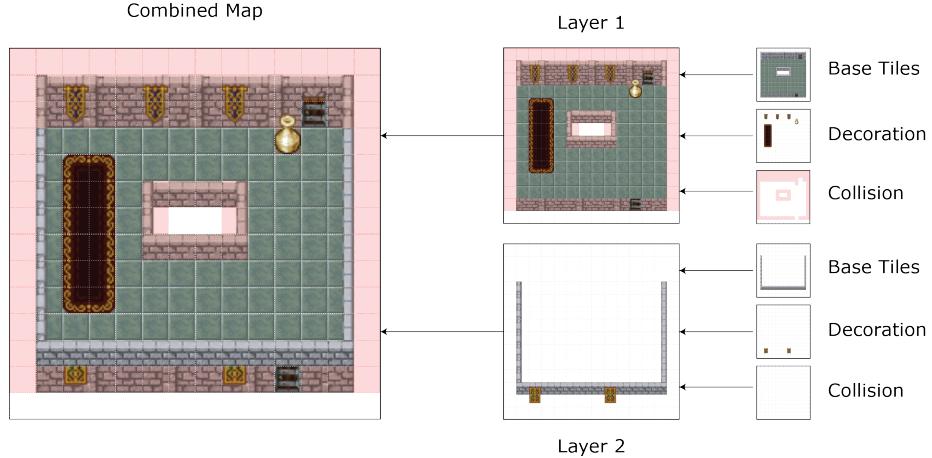


Figure 2.39: The structure of a two-layer map.

Let's update the map code to handle drawing multiple layers. We already have code to draw one layer that we can reuse to draw multiple layers. Rename `Map:Render` to `Map:RenderLayer`. This function is responsible for rendering a single layer. `RenderLayer` assumes we're only rendering the first layer. Let's change that. Add an extra parameter `layer` to the `RenderLayer` function. The `layer` parameter is 1 for the first map layer, 2 for the second layer and so on. Make your code look like Listing 2.62.

```
function Map:Render(renderer)
    self:RenderLayer(renderer, 1)
end

function Map:RenderLayer(renderer, layer)

    -- Our map layers are made of 3 sections
    -- We want the index to point to the base section of a given layer
    local layerIndex = (layer * 3) - 2

    local tileLeft, tileBottom =
        self:PointToTile(self.mCamX - System.ScreenWidth() / 2,
                        self.mCamY - System.ScreenHeight() / 2)

    local tileRight, tileTop =
        self:PointToTile(self.mCamX + System.ScreenWidth() / 2,
                        self.mCamY + System.ScreenHeight() / 2)
```

```

for j = tileTop, tileBottom do
    for i = tileLeft, tileRight do

        local tile = self:GetTile(i, j, layerIndex)
        local uvs = {}

        self.mTileSprite:SetPosition(self.mX + i * self.mTileWidth,
                                      self.mY - j * self.mTileHeight)

        -- Base layer
        if tile > 0 then
            uvs = self.mUVs[tile]
            self.mTileSprite:SetUVs(unpack(uvs))
            renderer:DrawSprite(self.mTileSprite)
        end

        -- Decoration layer
        tile = self:GetTile(i, j, layerIndex + 1)

        -- If the decoration tile exists
        if tile > 0 then
            uvs = self.mUVs[tile]
            self.mTileSprite:SetUVs(unpack(uvs))
            renderer:DrawSprite(self.mTileSprite)
        end

    end
end
end

```

Listing 2.62: Updating Map:Render to Map:RenderLayer. In Map.lua.

In Listing 2.62 Map:Render just calls Map:RenderLayer telling it to render layer 1. The main.lua file already calls Map:Render every frame; therefore our changes in Map:RenderLayer are shown every frame too.

In RenderLayer we calculate the layerIndex that points to the base tile section. To get the base tile section for layer 1 the layerIndex is equal to 1. To get the base tile section for layer 2 the layerIndex is equal to 4, to get the base tile section for 3 the layerIndex is equal to 7, and so on.

As before we iterate through the map and draw all tiles in the viewport. We draw base tiles by first calling GetTile with layerIndex to get the tile id from the layer we're rendering. Any base tile with id 0 is skipped. After drawing the base tile we check for a decoration tile in the section above and, if it exists we draw it.

Run the code and you'll get something like Figure 2.40 that renders the first layer only. Try drawing the second layer (on its own) by changing the 1 to a 2 in the Map:Render function.

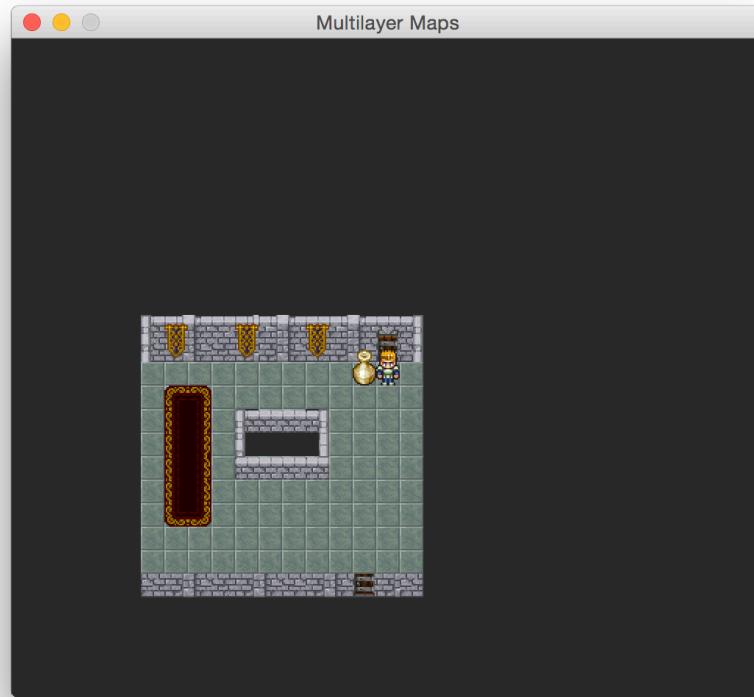


Figure 2.40: Using Map:RenderLayer to render just the first layer.

We need to render all the layers and any characters on those layers. To help achieve this let's add a new function, Map:LayerCount, as shown in Listing 2.63. The Map:LayerCount function returns the number of layers on a map.

```
function Map:LayerCount()
    -- Number of layers should be a factor of 3
    assert(#self.mMapDef.layers % 3 == 0)
    return #self.mMapDef.layers / 3
end
```

Listing 2.63: Function to get the numbers of layers in a map. In Map.lua.

Listing 2.63 calculates the number of layers by dividing the total layer count by the number of sections. There's an assert to make sure there's no remainder; layers should always come in multiples of three!

Let's switch our focus back to `main.lua`. Currently if you run the game the hero is stuck inside a wall! In *Listing 2.64* we've changed the `tileX` and `tileY` coordinates to place the hero in front of the room's door. We've also added a `layer` field set to 1, meaning the hero will render on top of the first layer.

```
local heroDef =
{
    texture = "walk_cycle.png",
    width = 16,
    height = 24,
    startFrame = 9,
    tileX = 11,
    tileY = 3,
    layer = 1
}
```

Listing 2.64: Move the hero out of the wall!. In Main.lua.

The def table is used by the Entity class and it needs code to handle the new layer field. In the constructor of Entity add the line shown in *Listing 2.65*.

```
function Entity:Create(def)
    local this =
    {
        -- code omitted

        mTileX = def.tileX,
        mTileY = def.tileY,
        mLayer = def.layer, -- New line

        -- code omitted
    }

    -- code omitted
end
```

Listing 2.65: Adding a layer field to the entity class. In Entity.lua.

Now that entities have `x`, `y` and `layer` coordinates, we can place them at different depths in the scene.

We have everything we need to render maps with multiple layers. We're going to manually render each layer in the main.lua update function to begin with, so we can easily position the hero. We'll test each layer to see if it's the one the hero is on and render his sprite after the matching layer. Update your update function so it looks like Listing 2.66.

```
function update()

    local dt = GetDeltaTime()

    local playerPos = gHero.mEntity.mSprite:GetPosition()
    gMap.mCamX = math.floor(playerPos:X())
    gMap.mCamY = math.floor(playerPos:Y())

    gRenderer:Translate(-gMap.mCamX, -gMap.mCamY)

    local layerCount = gMap:LayerCount()

    for i = 1, layerCount do
        gMap:RenderLayer(gRenderer, i)
        if i == gHero.mEntity.mLayer then
            gRenderer:DrawSprite(gHero.mEntity.mSprite)
        end
    end

    gHero.mController:Update(dt)
end
```

Listing 2.66: Render all the layers of a map and the hero. In main.lua.

Run the updated code and you'll see that all the layers and the hero are rendering nicely. Move the hero far enough to right and you'll see him partially obscured by the wall as shown in Figure 2.41. This gives a nice sense of depth and the small room feels more real and comfortable to explore. The example code so far can be found in character-9-solution.

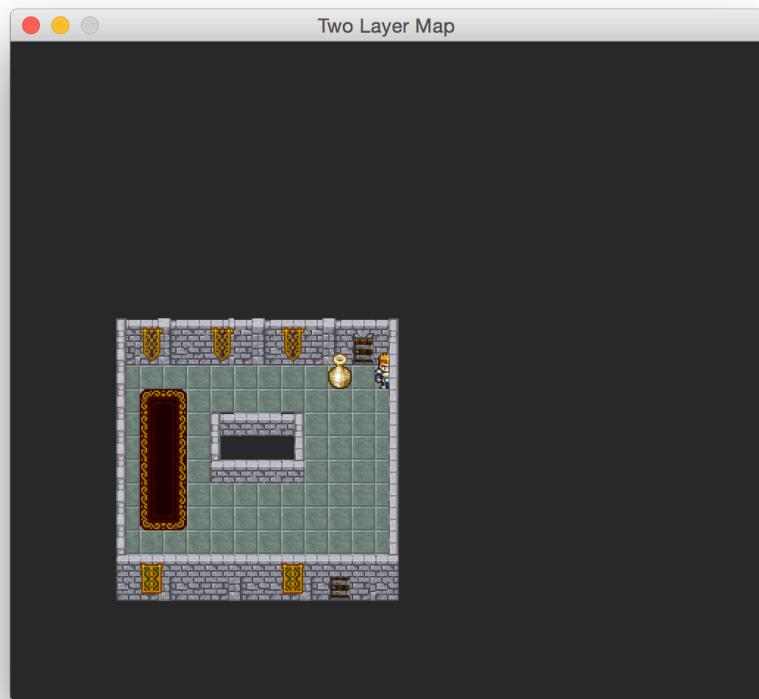


Figure 2.41: Rendering layers to give the level a better sense of depth.

That's it for layer maps. We're now able to create richly detailed maps and explore them in a game. We'll continue to tweak exactly how the map is rendered as we introduce NPCs in the next section.

All tile-based RPGs use a similar method to build their levels. You can look at classic RPG maps with a fresh eye and make a good guess at how they made their worlds and even try to reverse engineer a few maps!

Interaction Zones

Interaction zones are areas on the map that do something special. For instance, when the character walks onto a door tile we might teleport them to a new map.

Our interaction zones are made up two parts: actions and triggers. Actions are special functions that cause something to happen in the world. Triggers are invisible objects

placed on the map and can be activated by walking on them or using them via a button press. Triggers call actions when activated.

Actions

Actions are small functions that alter the game world in some way. They're easy to reuse and provide a diverse array of effects.

The first action we'll add is a simple teleport action. To test the action we'll trigger it when we hit the spacebar.

When designing a new system it's best to start with a little pseudo-code describing how you'd ideally like it to work. The pseudo code can then be used as a model to implement the real thing. Let's begin with pseudo code describing how the teleport action might be used. See Listing 2.67.

```
local action = TeleportAction(10, 15)
local trigger = nil
action(trigger, gHero)
```

Listing 2.67: An ideal teleport action.

In Listing 2.67 we create an action by calling `TeleportAction`. The action is a function and we store it in the variable `action`. This action function teleports any entity to X:10, Y:15 when called. We trigger the action by calling it with two parameters. The first parameter is the trigger that gave rise to the action and second is the entity the action is applied to. In our case the trigger is `nil` because we manually fire the action when the player presses the spacebar.

Let's implement our pseudo-code. Example character-10 has the code so far, or you can continue to use your own code. Start by creating a new Lua file called `Actions.lua`. This stores all our actions. Add it to the manifest and call `Asset.Run` on it in the `main.lua`. Here's the implementation in Listing 2.68.

```
Actions =
{
    -- Teleport an entity from the current position to the given position
    Teleport = function(map, tileX, tileY)
        return function(trigger, entity)
            entity.mTileX = tileX
            entity.mTileY = tileY
            Teleport(entity, map)
        end
    end
}
```

Listing 2.68: An actual teleport implementation. In Actions.lua.

The Teleport action, in Listing 2.68, uses the Teleport function we defined previously to move an entity to a given position on the map.

Our previously defined Teleport function is quite useful and we'll use it a number of places in the code, so let's move it into the Util.lua file. Once that's done, let's try out our teleport action in main.lua as shown in Listing 2.69.

```
-- Code omitted
gHero:Init()

gUpDoorTeleport = Actions.Teleport(gMap, 11, 3)
gUpDoorTeleport(nil, gHero.mEntity) -- replaces the old teleport call
```

Listing 2.69: Creating a teleport action that teleports the player to the tile in front of the door. In main.lua.

In Listing 2.69 we create a teleport action after the hero is created, then call it immediately to place the hero in front of the door. This works for a quick test, but for a better demonstration of the action we're going to hook it up to the spacebar key. Then we can fire the action at any time!

Copy the code from Listing 2.70 into your main.lua file.

```
function update()

    -- code omitted

    if Keyboard.JustPressed(KEY_SPACE) then
        gUpDoorTeleport(nil, gHero.mEntity)
    end
end
```

Listing 2.70: Pressing space to teleport. In main.lua.

Run the code, move around, and press the spacebar to fire the teleport action. This returns the hero character to the front of the door.

We can reuse the Actions.Teleport function to create an action that teleports the character to different locations. Let's make one more example teleport action. In Listing 2.71 we create a teleporter that teleports the player to the door at the bottom of the map.

```
-- code omitted
gUpDoorTeleport = Actions.Teleport(gMap, 11, 3)
gDownDoorTeleport = Actions.Teleport(gMap, 10, 11)
gUpDoorTeleport(nil, gHero.mEntity) -- replaces the old teleport call
```

Listing 2.71: Creating a teleport action that teleports the player to the tile in front of the door. In main.lua.

Now try calling the new action, gDownDoorTeleport, or changing which action is called when the spacebar is pressed! The code so far is available in character-10-solution.

It's easy to imagine the teleport having a particle effect, and fading the character out and then back in. This can all be built in but for now we're keeping it simple.

The teleport action is just a taste of what we can use actions for and a preview of their power. We can create actions that heal the player, spawn and despawn items on the map, or edit the map collision or tile data. There's a lot of scope.

To tie our actions into the world we need to write the Trigger class.

Trigger

A trigger calls an action when a certain event happens. The most common event is when a character enters a tile. We're going to create two triggers for each door in the map. If the character steps on a door tile, they'll teleport to the opposite door. We'll also implement code to activate a trigger if the user presses the 'use' button while facing it. This use button lets the player interact with levers, read signs, talk to people, and just generally interact with the world. For our project the use button will be the spacebar.

Example character-11 has the code we've written so far, or you can continue to use your own code.

Let's make a Trigger class. The Trigger constructor takes in a def table with three optional callback functions: OnEnter, OnExit, and OnUse. The action is called via these callbacks when the trigger is activated. OnEnter is called when the player finishes arriving on a tile. OnExit is called when a player finishes exiting a tile. OnUse is called when the player presses the use button facing a tile.

Create a new file called Trigger.lua, add it to the manifest, and in main.lua call Asset.Run on it. Add the Trigger code as shown in Listing 2.72.

```
Trigger = {}
Trigger.__index = Trigger
function Trigger:Create(def)

    local EmptyFunc = function() end
    local this =
```

```

{
    OnEnter = def.OnEnter or EmptyFunc,
    OnExit = def.OnExit or EmptyFunc,
    OnUse = def.OnUse or EmptyFunc,
}

setmetatable(this, self)
return this
end

```

Listing 2.72: A simple trigger class. In Trigger.lua.

The Trigger class, as shown in Listing 2.72, is extremely simple; it's just three callbacks in a table. If the callback doesn't exist then it's assigned an empty function. This makes all the callbacks valid functions but only those defined in the def table actually do anything.

There's nothing else to add to the Trigger class, so let's put it into use and create a trigger for each door in our example map. Add the code in Listing 2.73 to your main.lua.

```

-- code omitted

gTriggerTop = Trigger:Create
{
    OnEnter = gDownDoorTeleport
}

gTriggerBot = Trigger:Create
{
    OnEnter = gUpDoorTeleport
}

```

Listing 2.73: Creating two triggers. In main.lua.

In Listing 2.73 we define two triggers that call the teleport actions when the character stands on them. To make them work we need to update the map code so it can have triggers added.

Let's start by breaking up the GetTile function into two separate functions, GetTile and CoordToIndex, as shown in Listing 2.74. Remember that a layer of tile data is just one long table of tile ids. There's no information about which row or column a tile belongs to; that's calculated using the map width. The CoordToIndex function takes in an X, Y coordinate for a map and returns the index to a particular tile.

```

function Map:GetTile(x, y, layer)
    local layer = layer or 1
    local tiles = self.mMapDef.layers[layer].data

    return tiles[self:CoordToIndex(x, y)]
end

function Map:CoordToIndex(x, y)
    x = x + 1 -- change from 1 -> rowsize
    -- to          0 -> rowsize - 1
    return x + y * self.mWidth
end

```

Listing 2.74: Adding the coord to the index helper function. In Map.lua.

Update your Map.lua code to match Listing 2.74. We'll use CoordToIndex when adding triggers to the map. Add a triggers table, `mTriggers`, to the constructor as shown in Listing 2.75. By default the trigger table is empty. Each entry in the triggers table represents a layer on the map. Each entry in a layer table has an integer key representing the tile, and a Trigger value representing the trigger on that tile.

```

function Map:Create(mapDef)
    local layer = mapDef.layers[1]
    local this =
    {
        -- code omitted
        mTileWidth = mapDef.tilesets[1].tilewidth,
        mTileHeight = mapDef.tilesets[1].tileheight,
        mTriggers = {}
    }

    -- code omitted
end

```

Listing 2.75: An empty trigger table. In Map.lua.

To determine when to fire off a trigger, we need code in the map that tracks what the characters are doing. As a first step, we need to be able to query the map and ask if a trigger exists at a certain location. Add a new function called GetTrigger to the map, as shown in Listing 2.76.

GetTrigger takes in a layer and x, y coordinates and returns the trigger at that position or nil.

```

function Map:GetTrigger(layer, x, y)
    -- Get the triggers on the same layer as the entity
    local triggers = self.mTriggers[layer]

    if not triggers then
        return
    end

    local index = self:CoordToIndex(x, y)
    return triggers[index]
end

```

Listing 2.76: Helper function to get a trigger. In Map.lua.

In Listing 2.76 we use the layer variable to get the table of Trigger objects we'll be dealing with. If no table of Trigger objects exists for the given layer, we return nil. If there are triggers on this layer we call CoordToIndex with the x and y coordinates to get the tile index. Then we check the tile index in the triggers table and return either a valid Trigger or nil.

In the main.lua, let's manually add our two door triggers. Copy the code from Listing 2.77. The bottom door's X, Y coordinates are 10, 12 and the top door's coordinates are 11, 2. We use the CoordToIndex helper function to convert these two sets of coordinates to an index. The index CoordToIndex matches the index of door tiles in the gMap.mLayer.data table.

```

-- code omitted

gTriggerBot = Trigger:Create
{
    OnEnter = gUpDoorTeleport,
}

gMap.mTriggers =
{
    -- Layer 1
    {
        [gMap:CoordToIndex(10, 12)] = gTriggerBot,
        [gMap:CoordToIndex(11, 2)] = gTriggerTop,
    }
}

-- code omitted

```

Listing 2.77: Hard coding some triggers. In main.lua.

Listing 2.77 adds triggers to the map but we need code to activate the triggers when an appropriate event occurs.

We'll implement code to fire a trigger's OnEnter event first, as it's the most immediately useful. OnEnter is called when a character finishes entering a tile. This occurs when the character exits the MoveState. Update the MoveState:Exit to match Listing 2.78.

```
function MoveState:Exit()
    local trigger = self.mMap:GetTrigger(self.mEntity.mLayer,
                                         self.mEntity.mTileX,
                                         self.mEntity.mTileY)
    if trigger then
        trigger:OnEnter(self.mEntity)
    end
end
```

Listing 2.78: Check for triggers when exiting. In MoveState.lua

In the MoveState:Exit function we ask the map if there's a trigger on the current tile. If there is then we call the trigger's OnEnter; if not, nothing happens. In our code we have a trigger on each door tile. When the player moves onto the trigger, its OnEnter callback is fired and this calls the teleport action which moves the player to the opposite door.

Run the code and check out the working teleporters! Example character-11-solution has the code so far. Walking onto one door tile causes you to appear at the opposite door.

OnExit and OnUse Trigger callbacks

We've added the OnEnter callback for triggers but we also need to add OnUse and OnExit. We'll start with OnExit as it complements OnEnter and deals with the same code.

Update your MoveState code to match Listing 2.79.

```
function MoveState:Exit()

    if self.mMoveX ~= 0 or self.mMoveY ~= 0 then
        local trigger = self.mMap:GetTrigger(self.mEntity.mLayer,
                                             self.mEntity.mTileX,
                                             self.mEntity.mTileY)
        if trigger then
            trigger:OnExit(self.mEntity)
        end
    end
end
```

```

        self.mEntity.mTileX = self.mEntity.mTileX + self.mMoveX
        self.mEntity.mTileY = self.mEntity.mTileY + self.mMoveY
        Teleport(self.mEntity, self.mMap)

        local trigger = self.mMap:GetTrigger(self.mEntity.mLayer,
                                              self.mEntity.mTileX,
                                              self.mEntity.mTileY)
        if trigger then
            trigger:OnEnter(self.mEntity)
        end
    end
end

```

Listing 2.79: MoveState:Exit that triggers OnExit and OnEnter. In MoveState.lua

The OnExit callback is called when character leaves a tile. In the MoveState:OnExit function, we check the tile we've just left for a trigger and if one exists we call the OnExit callback.

The final trigger callback we want to support is OnUse. The *use* key is the spacebar. When the spacebar is pressed, the tile that the character is facing is checked for triggers and the OnUse callback is called. This callback differs in that it's not on the same tile as the character but one tile in front. If there's a lever in the world, the player doesn't stand on it to pull it, they stand on the tile in front of the lever and press the use button.

To implement OnUse we need to know which way the player is facing, and to do that we're going to add a field mFacing to the hero def. Update your main.lua file to add the new field as shown in Listing 2.80.

```

local gHero
gHero =
{
    -- code omitted
    mAnimLeft = {13, 14, 15, 16},
    mFacing = "down",
    -- code omitted
}

```

Listing 2.80: Adding facing information to the character. In main.lua.

The mFacing field is initially set to "down" which is the direction the character is facing when first loaded. Character facing is changed in two places, the MoveState and the WaitState. Modify both states so the code is the same as in Listing 2.81.

```

-- WaitState.lua
function WaitState:Update(dt)

    -- code omitted
    self.mFrameCount = -1
    self.mEntity:SetFrame(self.mEntity.mStartFrame)
    self.mCharacter.mFacing = "down"
    -- code omitted
end

-- MoveState.lua
function MoveState:Enter(data)

    local frames = nil

    if data.x == 1 then
        frames = self.mCharacter.mAnimRight
        self.mCharacter.mFacing = "right"
    elseif data.x == -1 then
        frames = self.mCharacter.mAnimLeft
        self.mCharacter.mFacing = "left"
    elseif data.y == -1 then
        frames = self.mCharacter.mAnimUp
        self.mCharacter.mFacing = "up"
    elseif data.y == 1 then
        frames = self.mCharacter.mAnimDown
        self.mCharacter.mFacing = "down"
    end

    self.mAnim:SetFrames(frames)

```

Listing 2.81: Updating the facing value. In MoveState.lua and WaitState.lua.

In Listing 2.81 we modified the code to update the `mFacing` field when the character changes direction.

Let's create a helper function `GetFacedTileCoords` to help find the tile in front of a character. Add the code as shown in Listing 2.82 to your `main.lua` file.

```

function GetFacedTileCoords(character)

    -- Change the facing information into a tile offset
    local xInc = 0
    local yInc = 0

```

```

if character.mFacing == "left" then
    xInc = -1
elseif character.mFacing == "right" then
    xInc = 1
elseif character.mFacing == "up" then
    yInc = -1
elseif character.mFacing == "down" then
    yInc = 1
end

local x = character.mEntity.mTileX + xInc
local y = character.mEntity.mTileY + yInc

return x, y
end

```

Listing 2.82: Helper function to get the tile the player is facing. In main.lua.

In Listing 2.82 we translate the player's facing direction to a tile offset. Add the offset to the player position and return those coordinates.

To implement the OnUse event update your main file to match Listing 2.83. We use GetFacedTileCoords when we press spacebar to activate any trigger in front of the character.

```

function update()

    -- code omitted

    if Keyboard.JustPressed(KEY_SPACE) then
        -- which way is the player facing?
        local x, y = GetFacedTileCoords(gHero)
        local trigger = gMap:GetTrigger(gHero.mEntity.mLayer, x, y)
        if trigger then
            trigger:OnUse(gHero)
        end
    end
end

```

Listing 2.83: Adding the OnUse callback. In main.lua.

Example character-12 demonstrates all the trigger callbacks we've implemented. Be sure to check it out!

Next Steps

That concludes our map code. There's always more that can be done but we're stopping here and will move on to NPCs in the next section.

A Living World

In the last section we learned how to build multi-layer worlds with collision detection and triggers. We're extremely close to something that looks like a basic game. In this chapter we're going to refactor a lot of our code and make it more general and more data driven⁵. A data driven approach is more suited to managing the complexity of larger games like RPGs. Generalizing the player character code will let us easily add NPCs. We'll be extending the map definition so characters and triggers are loaded with maps, rather than added ad hoc in the main file, like we're doing presently.

This chapter doesn't cover much new ground. We're consolidating and refactoring code to make it more general. These kind of refactoring techniques are important to know; the more you can reduce complexity in your codebase the easier it is to reason about and understand.

Character Class

In our most recent examples the player can explore a map by using the arrow keys to move the hero character around. We're going to refine this code by making a Character class to use for both the hero and NPCs. The Character class brings together a lot of code that's currently spread out in the main.lua file. Example npc-1 has the code so far, or you can continue using your own codebase. Once we make the Character class we can edit a character's controller (the state machine that controls what the character does) to add simple AI⁶. The Character class also helps us better integrate NPCs with the map.

In main.lua there is a table, heroDef, that defines the hero entity; what texture it uses, how many animation frames there are, etc. To add more entities, we need to add more definitions tables, which would quickly make the main.lua file messy. To make things cleaner let's move heroDef into a new file where we'll store *all* our entity definitions. Create a file called EntityDefs.lua. Remove the heroDef from main.lua and make your EntityDef.lua file look like Listing 2.84.

Add EntityDefs.lua to the manifest and call Asset.Run on it in the main.lua file. Add the Asset.Run call *after* Asset.Run(WaitState) and Asset.Run(MoveState) because it uses these classes when run.

⁵The term data driven means we want to extract and isolate the data in the code that the logic acts upon. This makes it easier to tweak the data to refine our game and add more data to add more content.

⁶Artificial Intelligence. In computer games AI is a broad term most often referring to code that controls entities in the game world.

```

-- WaitState, MoveState must already be loaded.
assert(WaitState)
assert(MoveState)

-- Text Id -> Controller State
gCharacterStates =
{
    wait = WaitState,
    move = MoveState
}

gEntities =
{
    hero =
    {
        texture = "walk_cycle.png",
        width = 16,
        height = 24,
        startFrame = 9,
        tileX = 11,
        tileY = 3,
        layer = 1
    }
}

gCharacters =
{
    hero =
    {
        entity = "hero",
        anims =
        {
            up      = {1, 2, 3, 4},
            right   = {5, 6, 7, 8},
            down   = {9, 10, 11, 12},
            left    = {13, 14, 15, 16},
        },
        facing = "down",
        controller = { "wait", "move" },
        state = "wait"
    }
}

```

Listing 2.84: EntityDefs.lua, a place to store our definitions. In EntityDefs.lua.

Definition tables are data, not code. Data tables are made from simple types: strings, numbers, booleans, and tables, but no functions or references to other tables. The use of definition tables is a way to make the code more data driven. We use definition tables to help create new game objects.

This EntityDef.lua file contains all our definitions for entities in the game. Think of it as a big book of ingredients and a list of recipes to combine those ingredients. These ingredients include characters, monsters, AI states and animations. As we create our game we add more and more ingredients, and combining these gives a giant repertoire of content. At this stage it may seem over-engineered but as we continue to add more assets and content to the game you'll quickly see how this type of structure keeps complexity manageable.

The definition table is a kind of programming pattern, and it's applicable to all types of games, not just RPGs. (Sometimes the term *blueprint* is used to mean something quite close to a definition table.) If you ever tried to take on big project and failed because it became too complex and buggy, then pay special attention to how this is being set up and what the results are a few chapters into the book.

We only have the hero entity definition at the moment but we'll be adding more. Near the top of the file there's a table, gControllerStateMap. The gControllerStateMap table associates string ids with state classes. We use this to build state machines from text descriptions.

Look at the hero table in gCharacters. It has a field called controller. The controller table is just a list of strings, each the name of a state. When we make a character we transform this table of strings into a state machine with the help of the gControllerStateMap.

The gCharacters table, near the bottom of the file, stores character definitions. A character is an entity with extra state and animation data; it's usually an NPC or the hero character.

Character definitions contains information about how the character should look and animate. The first entry, entity, is the id of the entity used to represent the character. For the hero the value is "hero" which we defined in the gEntities table above. If we wanted to change the character's appearance we could swap this out for a different entity. The next entry is a table of animations for walking in each of the four directions. The initialFacing field tells the character which direction to face when created. The controller describes a state machine of two states with ids of move and wait. We also define the initial state by setting the state entry to wait. This a complete character definition for the hero. When our hero is created he'll be facing to the south and be in wait state, awaiting user input.

Create a brand new file, Character.lua, and copy the code from Listing 2.85. This class creates characters based on the definition files in EntityDefs.lua.

```
Character = {}  
Character.__index = Character
```

```

function Character:Create(def, map)

    -- Look up the entity
    local entityDef = gEntities[def.entity]
    assert(entityDef) -- The entity def should always exist!

    local this =
    {
        mEntity = Entity:Create(entityDef),
        mAnims = def.anims,
        mFacing = def.facing,
    }

    setmetatable(this, self)

    -- Create the controller states from the def
    local states = {}
    -- Make the controller state machine from the states
    this.mController = StateMachine:Create(states)

    for _, name in ipairs(def.controller) do
        local state = gCharacterStates[name]
        assert(state)
        assert(states[state mName] == nil) -- State already in use!
        local instance = state:Create(this, map)
        states[state mName] = function() return instance end
    end

    this.mController.states = states

    -- Change the statemachine to the initial state
    -- as defined in the def
    this.mController:Change(def.state)

    return this
end

```

Listing 2.85: The Character class. In Character.lua.

The character constructor in Listing 2.85 is one of the more complicated constructors we've created so far, so let's go through it and make sure every line makes sense.

The constructor takes in a character definition table and the current map which is used by the controller. The definition table contains an entity field which stores the string id of an entity definition. We use the id to look up the entity def in the gEntities table

in EntityDefs.lua. All characters need a valid entity id so we use an assert to ensure entityDef exists. If entityDef doesn't exist that may mean the character entity field has a typo.

After retrieving the entityDef we use it to construct an Entity which we store in the Character's this table with the animations and initial facing direction. The only task remaining for the constructor is to create the controller state machine.

The def.controller describes the controller using simple strings. These strings need transforming into a working state machine. In the Character constructor we create a states table to store the controller's states. Then we create an empty state machine mController. For each entry in def.controller we look up the state class using the id, create an instance of the state, and store it in the states table. TheStateMachine controller expects a table of functions, therefore each entry in the states table is a function that returns a state. After the for-loop our table states is full of functions that return state objects when given the associated state id. The states table is then assigned to the state machine and it's ready to use.

In our heroDef character definition the controller has two strings, "wait" and "move", which refer to the classes WaitState and MoveState in the gCharacterStates table. In the Character constructor these string ids are looked up using gCharacterStates to retrieve the classes they refer to.

The code is a little tricky because there's a dependency loop. The WaitState and MoveState constructors require the state machine and the state machine constructor requires all the states! To sidestep this loop we create an empty state machine, create the states, and then fill up the state machine with the created states. To make this explicit after the for-loop we assign the states to the mController.mStates table.

A Character object constructed using the hero definition has a state table that looks like Listing 2.86.

```
{  
    ['wait'] = function() return waitState end,  
    ['move'] = function() return moveState end,  
}
```

Listing 2.86: A state table for the controller of the hero.

Listing 2.85 is very similar to what we were hard coding previously. Instead of hand building the character controllers, we now define them in the def and the code builds them for us. This makes developing and tweaking characters much faster and less error prone.

In main.lua remove the gHeroDef and gHero tables and copy in the code from Listing 2.87.

```

-- code omitted
Asset.Run("Trigger.lua")
Asset.Run("EntityDefs.lua")
Asset.Run("Character.lua")
Asset.Run("small_room.lua")

local gMap = Map:Create(CreateMap1())
gRenderer = Renderer:Create()

gMap:GotoTile(5, 5)

gHero = Character:Create(gCharacters.hero, gMap)
-- code omitted

```

Listing 2.87: Creating a character from a definition. In main.lua.

If you run the program now, it works ... until you try to move. Then it crashes. The Character class changes how the animation frames are stored and MoveState expects the old animation locations. We need to update MoveState. Copy the code from Listing 2.88. These changes ensure self.mAnim is assigned correctly.

```

function MoveState:Enter(data)

    local frames = nil

    if data.x == 1 then
        frames = self.mCharacter.mAnims.right
        self.mCharacter.mFacing = "right"
    elseif data.x == -1 then
        frames = self.mCharacter.mAnims.left
        self.mCharacter.mFacing = "left"
    elseif data.y == -1 then
        frames = self.mCharacter.mAnims.up
        self.mCharacter.mFacing = "up"
    elseif data.y == 1 then
        frames = self.mCharacter.mAnims.down
        self.mCharacter.mFacing = "down"
    end

    self.mAnim:SetFrame(frames)

```

Listing 2.88: Fixing up the animations. In MoveState.lua.

Run the code again and movement is fixed. The latest version of the code is in `npc-1-solution`. Next we'll use the Character class to add NPCs.

A New Controller

An RPG world can seem a little lifeless without NPCs wandering around. With our new Character class, adding some NPCs is pretty simple. Example npc-2 has the code so far, or you can continue to use your own codebase. Let's start very simply by getting a stationary NPC standing in the world. If we're going to add a character to the world, we need a suitable character definition, so head over to EntityDefs.lua and add a new definition to the gCharacters table as shown in Listing 2.89.

```
gCharacters =  
{  
    standing_npc =  
    {  
        entity = "hero",  
        anims = {},  
        facing = "down",  
        controller = {"npc_stand"},  
        state = "npc_stand"  
    },  
  
    -- code omitted
```

Listing 2.89: Fixing up the animations. In EntityDefs.lua.

In Listing 2.89 there's a new character def standing_npc that uses the hero entity but has no animations. This def creates a character that just stands on the map. It only has one state npc_stand. Before implementing npc_stand let's add it to the gCharacterStates map as shown in Listing 2.90.

```
gCharacterStates =  
{  
    wait = WaitState,  
    move = MoveState,  
    npc_stand = NPCStandState,  
}
```

Listing 2.90: Adding a new standing state. In EntityDefs.lua.

NPCStandState is the class we'll use for our standing state. Create a new file NPCStandState.lua, add it to the manifest, and call Asset.Run on it in the main.lua file. This class is a state to be used in a state machine, therefore it implements all the expected state functions, even if they're just empty. Copy the stand state code from Listing 2.91.

```

NPCStandState = { mName = "npc_stand" }
NPCStandState.__index = NPCStandState
function NPCStandState:Create(character, map)
    local this =
    {
        mCharacter = character,
        mMap = map,
        mEntity = character.mEntity,
        mController = character.mController,
    }

    setmetatable(this, self)
    return this
end

function NPCStandState:Enter() end
function NPCStandState:Exit() end
function NPCStandState:Update(dt) end
function NPCStandState:Render(renderer) end

```

Listing 2.91: A simple standing state. In `NPCStandState.lua`.

The `NPCStandState` code is pretty sparse. It just maintains the character's initial settings. We've already added it the `gCharacterStates` table and referenced it in `standing_npc` so it's ready to use! Let's make an NPC character, add it into the world, and render it. Update your `main.lua` file with the code from Listing 2.92.

```

gHero = Character>Create(gCharacters.hero, gMap)
gNPC = Character>Create(gCharacters.standing_npc, gMap)
Actions.Teleport(gMap, 11, 5)(nil, gNPC.mEntity)

-- code omitted
function update()

    -- code omitted

    for i = 1, layerCount do
        gMap:RenderLayer(gRenderer, i)
        if i == gHero.mEntity.mLayer then
            gRenderer:DrawSprite(gHero.mEntity.mSprite)
        end
        if i == gNPC.mEntity.mLayer then
            gRenderer:DrawSprite(gNPC.mEntity.mSprite)
        end
    end
end

```

```

gHero.mController:Update(dt)
gNPC.mController:Update(dt)

-- code omitted
end

```

Listing 2.92: A simple standing state. In main.lua.

In Listing 2.92 we create an NPC, as gNPC, then use a Teleport action to place him on tile X:11, Y:5. To render the NPC we check his layer in the map render loop and render him after that layer has been drawn. Run the code and you'll see a new NPC character a couple of tiles down from the player. They're using the same art assets so they look the same. Example npc-2-solution has the code so far.

We've successfully added an NPC but there are couple of issues; the code doesn't scale well when adding multiple NPCs, and you can walk through the NPC we just added. These are issues we're going to solve, but first we're going to add more functionality to the NPC.

A standing NPC is good but a moving NPC is better. Let's add a new NPC character def that uses a new state, PlanStrollState, to make him move around. Copy the code below, Listing 2.93, into EntityDefs.lua.

```

gCharacterStates =
{
    wait = WaitState,
    move = MoveState,
    npc_stand = NPCStandState,
    plan_stroll = PlanStrollState,
}

-- code omitted

gCharacters =
{
    strolling_npc =
    {
        entity = "hero",
        anims =
        {
            up      = {1, 2, 3, 4},
            right   = {5, 6, 7, 8},
            down   = {9, 10, 11, 12},
            left   = {13, 14, 15, 16},
        },
    },
}

```

```

facing = "down",
controller = {"plan_stroll", "move"},
state = "plan_stroll"
},
-- code omitted

```

Listing 2.93: Creating a strolling NPC def. In EntityDefs.lua.

In Listing 2.93 we add PlanStrollState to the gCharacterStates table with the id "plan_stroll". Then we add a new character def, strolling_npc. The strolling_npc def has a controller with two states: plan_stroll, the new state, and move, the state we're already using to move the hero character around. The default state is set to plan_stroll.

PlanStrollState is the state that's going to make our NPC wander around the map. It works by waiting for a random number of seconds then picking a random direction to move in and changing to the MoveState. Create a new file PlanStrollState.lua, add it to the manifest, and call Asset.Run on it in the main.lua file. Add the code from Listing 2.94.

```

PlanStrollState = { mName = "plan_stroll" }
PlanStrollState.__index = PlanStrollState
function PlanStrollState:Create(character, map)
    local this =
    {
        mCharacter = character,
        mMap = map,
        mEntity = character.mEntity,
        mController = character.mController,

        mFrameResetSpeed = 0.05,
        mFrameCount = 0,

        mCountDown = math.random(0, 3)
    }

    setmetatable(this, self)
    return this
end

function PlanStrollState:Enter()
    self.mFrameCount = 0
    self.mCountDown = math.random(0, 3)
end
function PlanStrollState:Exit() end

```

```

function PlanStrollState:Update(dt)

    self.mCountDown = self.mCountDown - dt
    if self.mCountDown <= 0 then
        -- Choose a random direction and try to move that way.
        local choice = math.random(4)
        if choice == 1 then
            self.mController:Change("move", {x = -1, y = 0})
        elseif choice == 2 then
            self.mController:Change("move", {x = 1, y = 0})
        elseif choice == 3 then
            self.mController:Change("move", {x = 0, y = -1})
        elseif choice == 4 then
            self.mController:Change("move", {x = 0, y = 1})
        end
    end

    -- If we're in the stroll state for a few frames, reset the frame to
    -- the starting frame.
    if self.mFrameCount ~= -1 then
        self.mFrameCount = self.mFrameCount + dt
        if self.mFrameCount >= self.mFrameResetSpeed then
            self.mFrameCount = -1
            self.mEntity:SetFrame(self.mEntity.mStartFrame)
            self.mCharacter.mFacing = "down"
        end
    end

end

function PlanStrollState:Render(renderer) end

```

Listing 2.94: A strolling state. In PlanStrollState.lua.

The PlanStrollState, in Listing 2.94, is similar to the WaitState but instead of using keyboard input to decide the move direction, it chooses one at random. In the constructor we set mCountDown to a random number between 0 and 3; this is the number of seconds the state waits before choosing a random direction to stroll in. Notice in the Enter function the mCountDown number is chosen randomly again. Each time we enter the state, the wait time is a little different. This makes our character movement appear more natural and less uniform.

In the Update function mCountDown is reduced by the frame time, dt, then it's tested to see if it's equal to or below 0. When the mCountDown runs out it's time to move the character. To choose the move direction we generate a random number between

1 and 4. Each number represents a different direction to move. We change to the MoveState passing in our randomly chosen direction. The MoveState state then handles the business of moving the character to the next tile. At the end of the Update function there's code to reset the facing direction to down if the character hasn't moved for a while.

PlanStrollState works well enough, but when it finishes, it returns to the WaitState not the PlanStrollState! Our strolling npc doesn't have a WaitState! Let's make MoveState return to the character's initial state after finishing. To do this we record the initial state in the Character class as shown in Listing 2.95.

```
function Character:Create(def, map)

    -- code omitted

    local this =
    {
        -- code omitted
        mDefaultState = def.state,
    }
```

Listing 2.95: Recording the initial state. In Character.lua.

Update the MoveState to return to the initial state when finished by copying the code from Listing 2.96.

```
function MoveState:Enter(data)

    -- code omitted
    if self.mMap:IsBlocked(1, targetX, targetY) then
        self.mMoveX = 0
        self.mMoveY = 0
        self.mEntity:SetFrame(self.mAnim:Frame())
        self.mController:Change(self.mCharacter.mDefaultState)
        return
    end
    -- code omitted

end

function MoveState:Update(dt)

    -- code omitted

    if self.mTween:IsFinished() then
```

```

        self.mController:Change(self.mCharacter.mDefaultState)
    end
end

```

Listing 2.96: Recording the initial state. In MoveState.lua.

This code is straightforward. We store the initial state in the character as `mDefaultState` and return to this state in `MoveState:Update` when the move finishes.

Return to `main.lua` and copy the code from Listing 2.97 to replace the `standing_npc` with a `strolling_npc`. Run the code and the NPC will stroll around the map!

```

gNPC = Character>Create(gCharacters.strolling_npc, gMap)

```

Listing 2.97: A strolling NPC. In main.lua.

The NPC strolls at random but all the animations, smooth movement, collision detection, and triggers work the same as with the character controlled player. This is because we're carefully reusing the `MoveState` in the NPC controller. Example `npc-3-solution` contains the code so far.

Adding NPCs to the Map

Our game world is a bit more lively now that we have NPCs wandering around! But what if we wanted to add ten NPCs? The way we render and update characters on the map isn't scalable. To properly integrate NPCs with the map let's add two tables to the `Map` class, a list of characters and an entity table. These two tables represent all the entities and characters that exist on a map. Example `npc-4` contains the code so far, or you can continue to use your own code.

Let's extend the `mapDef` by adding NPC information. We'll start by directly editing our `mapDef` in `main.lua`. Copy the code from Listing 2.98 to add two NPCs, a strolling NPC and a standing one.

```

local mapDef = CreateMap1()
mapDef.on_wake =
{
    {
        id = "AddNPC",
        params = {{ def = "strolling_npc", x = 11, y = 5 }},
    },
    {
        id = "AddNPC",
        params = {{ def = "standing_npc", x = 4, y = 5 }},
    }
}

```

```

        }
    }
local gMap = Map:Create(mapDef)

```

Listing 2.98: Extending the map def with NPCs. In main.lua.

In the map definition above we've added a table called `on_wake`. The `on_wake` table is a list of action defs that run when the map is first loaded. Our `on_wake` table contains two `AddNPC` actions that add an NPC to the map when run. We'll define the `AddNPC` action soon, but first let's look at how it's used. The `id` is the name of the action to run. The `params` table is a list of parameters to pass to the action when it's executed. The `AddNPC` action constructor takes in a single table; this is why `params` is a table of one table.

For the `AddNPC` action, the `params` table contains three fields: a `def` which references an entry in the `gCharacters` table, and a set of X and Y tile coordinates to place the NPC on map. Before writing the `AddNPC` action, we need to change how the map deals with NPCs.

Copy the code from Listing 2.99 into your `map.lua` file. We're adding two fields to the `Map`: `mNPCs`, a table to store characters on the map, and `mEntities`, a table to store entities.

```

function Map:Create(mapDef)
    local layer = mapDef.layers[1]
    local this =
    {
        -- code omitted
        mTriggers = {},
        mEntities = {},
        mNPCs = {}
    }

    -- code omitted

```

Listing 2.99: Adding an NPCs and Entities table to the map. In Map.lua.

The `mEntities` tables is structured in a similar way to the `mTriggers` table. Each entry is a layer and each layer contains tile-index keys pointing to entities. This structure is visualized in Figure 2.42. When rendering a tile we can use `mEntities` to quickly check if there's an entity at that location. The `mNPCs` table is simpler. It's just a list of all the NPCs on the map. The `mNPCs` table makes it easy to call the `Update` function for each character and makes it easy to search for a certain character.

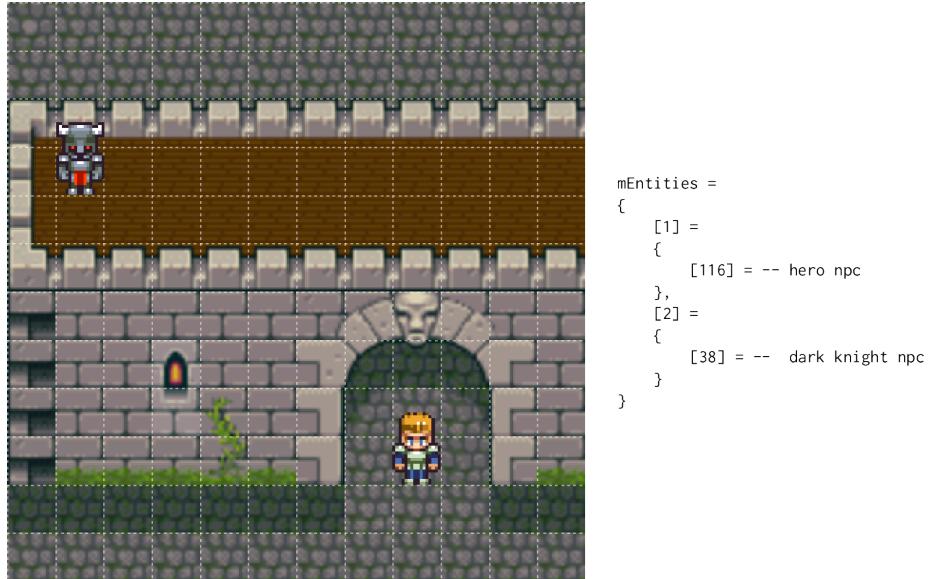


Figure 2.42: How the Map.mEntities table describes locations on the map.

The `mEntities` and `mNPCs` tables both start out empty and are populated when running through the `on_wake` action list supplied by `mapDef`. Copy the code to parse and execute the `on_wake` table from Listing 2.100.

```

setmetatable(this, self)

for _, v in ipairs(mapDef.on_wake or {}) do
    local action = Actions[v.id]
    action(this, unpack(v.params))()
end

return this
end

```

Listing 2.100: Using an action to add NPCs. In Map.lua.

The `on_wake` table in Listing 2.100 contains a list of action defs. We're going to construct actions from these defs. The action constructor takes in the Map as the first parameter and the actions expect to be able to use the map's functions. Therefore we process the `on_wake` table *after* the Map's `setmetatable` call, so its functions are linked up.

We iterate through the `on_wake` table from the `mapDef`. If no `on_wake` table exists it defaults to the empty table. For each entry we use the `id` to look up the associated

action function, then call the function, passing in the map as the first parameter and the contents of the params table for the remaining parameters. This returns an action that we execute immediately. Each action we run modifies the map in some way.

Open Actions.lua and let's create the AddNPC action that places an NPC on a map. This action takes two parameters, the map and an NPC def. Copy the code from Listing 2.101.

```
-- code omitted
AddNPC = function(map, npc)
    return function(trigger, entity)

        local charDef = gCharacters[npc.def]
        assert(charDef) -- Character should always exist!
        local char = Character:Create(charDef, map)

        -- Use npc def location by default
        -- Drop back to entities locations if missing
        local x = npc.x or char.mEntity.mTileX
        local y = npc.y or char.mEntity.mTileY
        local layer = npc.layer or char.mEntity.mLayer

        char.mEntity:SetTilePos(x, y, layer, map)

        table.insert(map.mNPCs, char)
    end
end
}
```

Listing 2.101: AddNPC Action. In Actions.lua.

In Listing 2.101 we define AddNPC. The AddNPC function looks up the character definition and uses it to create a new character to represent the NPC. There's an assert to make sure the character definition we intend to use actually exists. We get character position from the npc table passed in or default to the current entity values. We call a new function SetTilePos using the x, y, layer position, and this adds the NPC's entity to the map's mEntities table. The final line adds the newly created character to the map's mNPCs list.

Maps now track the location of the entities. To keep the map and entity location in sync we're going to add a SetTilePos function to the entity. Instead of directly setting entity.mTileX we'll use Entity:SetTilePos. Copy the code from Listing 2.102.

```
function Entity:SetTilePos(x, y, layer, map)

    -- Remove from current tile
```

```

if map:GetEntity(self.mTileX, self.mTileY, self.mLayer) == self then
    map:RemoveEntity(self)
end

-- Check target tile
if map:GetEntity(x, y, layer, map) ~= nil then
    assert(false) -- there's something in the target position!
end

self.mTileX = x or self.mTileX
self.mTileY = y or self.mTileY
self.mLayer = layer or self.mLayer

map:AddEntity(self)
local x, y = map:GetTileFoot(self.mTileX, self.mTileY)
self.mSprite:SetPosition(x, y + self.mHeight / 2)

end

```

Listing 2.102: Entity set tile position function. In Entity.lua.

The code in Listing 2.102 uses a few new map functions we'll define shortly but it should be pretty readable. The Entity.SetTilePos function moves the entity to a certain tile on a map. It keeps the Map's mEntities table up to date and correctly positions the entity sprite, meaning we no longer need to call the Teleport function.

Let's check out SetTilePos in a little more detail. First it uses the current entity position to find its reference on the map and then calls RemoveEntity (this doesn't cause an error if the entity doesn't exist). It then updates the entity position and calls AddEntity on the map passed in, using the new position. Finally it updates the sprite position to align it to the foot of the new tile. Trying to move to a position already occupied by another entity is an error so in that case we assert.

The code uses three new Map methods, GetEntity, AddEntity, and RemoveEntity, defined below. Copy the code from Listing 2.103 into your Map.lua file.

```

function Map:GetEntity(x, y, layer)
    if not self.mEntities[layer] then
        return nil
    end
    local index = self:CoordToIndex(x, y)
    return self.mEntities[layer][index]
end

function Map:AddEntity(entity)

```

```

-- Add the layer if it doesn't exist
if not self.mEntities[entity.mLayer] then
    self.mEntities[entity.mLayer] = {}
end

local layer = self.mEntities[entity.mLayer]
local index = self:CoordToIndex(entity.mTileX, entity.mTileY)

assert(layer[index] == entity or layer[index] == nil)
layer[index] = entity
end

function Map:RemoveEntity(entity)
    -- The layer should exist!
    assert(self.mEntities[entity.mLayer])
    local layer = self.mEntities[entity.mLayer]
    local index = self:CoordToIndex(entity.mTileX, entity.mTileY)
    -- The entity should be at the position
    assert(entity == layer[index])
    layer[index] = nil
end

```

Listing 2.103: Helpers functions to deal with entities on the map. In Map.lua.

The first function GetEntity takes an X, Y coordinate and layer and returns the entity, if any, at that position. It checks whether the layer exists, and if not then there's no entity so it returns nil. It gets the tile index by using the CoordsToIndex function and returns whatever is in the mEntity table at that index. There's nothing we haven't seen before in this function.

The second function is the AddEntity function. It takes an Entity and uses its location to insert it into the map's mEntities table. It checks if the layer the entity wants to be added to exists or not. If the layer doesn't exist it's created. Then the index is calculated from the coordinates using the CoordToIndex function and it's added to the layer table at the index position. There's an assert that checks that the target position is either empty or contains the entity we're positioning. If we add the entity to a place on the map where it already is, that's fine nothing changes and we can return as normal. If there's an entity there that isn't the one we're dealing with then we assert. We don't allow more than one entity on a tile.

The final function is RemoveEntity and as you'd imagine this removes an entity from the Map's mEntities table. Asserts ensure the entity exists at the remove position. It's removed from the mEntities table by setting the value at the target position to nil.

This finishes our updates to the map.

Before testing we must update our codebase to use the Entity:SetTilePos function instead of Teleport or directly setting the entity tile position. Start in main.lua by copying the code from Listing 2.104.

```
--gUpDoorTeleport(nil, gHero.mEntity) -- Remove this  
gHero.mEntity:SetTilePos(11, 3, 1, gMap) -- Add this
```

Listing 2.104: Updating how the player is positioned. In main.lua.

Next let's update MoveState:Exit. Copy the code from Listing 2.105.

```
function MoveState:Exit()  
  
    if self.mMoveX ~= 0 or self.mMoveY ~= 0 then  
        local trigger = self.mMap:GetTrigger(  
            self.mEntity.mLayer,  
            self.mEntity.mTileX,  
            self.mEntity.mTileY)  
  
        if trigger then  
            trigger:OnEnter(self.mEntity, x, y, layer)  
        end  
    end  
  
    self.mEntity:SetTilePos(  
        self.mEntity.mTileX + self.mMoveX,  
        self.mEntity.mTileY + self.mMoveY,  
        self.mEntity.mLayer,  
        self.mMap)
```

Listing 2.105: Update to use SetTilePos. In MoveState.lua.

The final update is for the Teleport trigger. Copy the code from Listing 2.106.

```
Teleport = function(map, tileX, tileY, layer)  
    layer = layer or 1  
    return function(trigger, entity)  
        entity:SetTilePos(tileX, tileY, layer, map)  
    end  
end,
```

Listing 2.106: Updating the Teleport action to use SetTilePos. In Actions.lua.

NPCs are loaded from the mapDef and added to the map but there's currently no code to update or render them. Let's fix this by updating the code to look like Listing 2.107. Listing 2.107 uses an updated RenderLayer function that renders the NPCs for that layer. The new function also optionally takes in a hero character and renders this when appropriate. The NPCs are updated at the end of the update function.

```
function update()

    -- code omitted

    for i = 1, layerCount do

        local heroEntity = nil
        if i == gHero.mEntity.mLayer then
            heroEntity = gHero.mEntity
        end

        gMap:RenderLayer(gRenderer, i, heroEntity)

    end

    gHero.mController:Update(dt)
    --gNPC.mController:Update(dt)
    for k, v in ipairs(gMap.mNPCs) do
        v.mController:Update(dt)
    end
```

Listing 2.107: Updating the NPCs. In main.lua.

Let's implement this new Map:RenderLayer function. Copy the code from Listing 2.108.

```
function Map:RenderLayer(renderer, layer, hero)

    -- code omitted

    for j = tileTop, tileBottom do
        for i = tileLeft, tileRight do

            -- code omitted

            local entLayer = self.mEntities[layer] or {}
            local drawList = { hero }

            for _, j in pairs(entLayer) do
                table.insert(drawList, j)
```

```
    end

    table.sort(drawList, function(a, b) return a.mTileY < b.mTileY end)
    for _, j in ipairs(drawList) do
        renderer:DrawSprite(j.mSprite)
    end
end
end
```

Listing 2.108: Rendering the entities. In Map.lua.

In Listing 2.108 we add all the entities to a list called `drawList` which we sort by the Y coordinate. Then we draw everything added to this list. Why do we sort it? Well, our entities are often taller than a tile. In a tile-based game the row of tiles nearest the bottom of the screen is nearer than the row of tiles at the top of the screen. Therefore entities nearer the bottom should be drawn in front of the entities nearer the top. This is what the sort does. Sorting the entities allows the player to walk in front of and behind other entities and NPCs so everything looks natural. Without the sort it's random chance if the entities are drawn correctly. Render order is shown in Figure 2.43. Try removing the sort code or reversing the check and you'll see the problematic behavior.

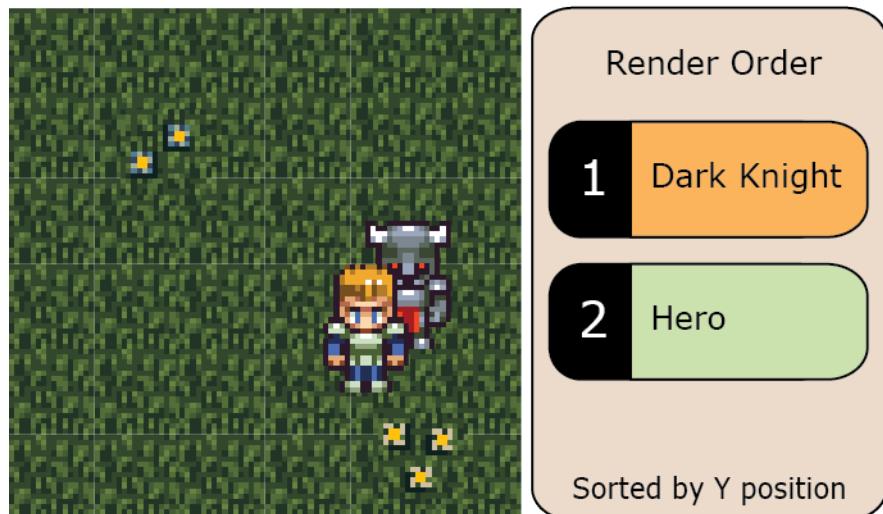
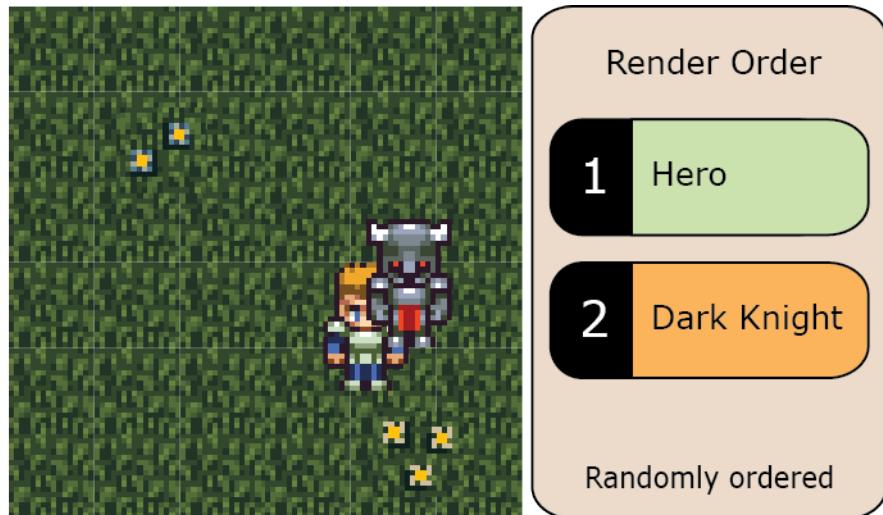


Figure 2.43: The different render orders for two entities. Sorting by Y gives the correct order.

Run the code and you'll see that there are now two NPCs, one standing and one walking around. Play the game for a short period and you'll notice that if an NPC tries to walk on top of another the game will crash. Two entities can't be in the same place at the same time. We're going to fix that next as we add collision detection for entities.

Example npc-4-solution has the code so far. Change by change we're moving our project to a more data driven codebase which makes adding content easier.

Collision Detection for Entities

Our collision detection works for tiles on the map but not for entities. Let's update the Map:IsBlocked function to rectify this. Copy the code from Listing 2.109.

```
function Map:IsBlocked(layer, tileX, tileY)
    -- Collision layer should always be 2 above the official layer
    local tile = self:GetTile(tileX, tileY, layer + 2)
    local entity = self:GetEntity(tileX, tileY, layer)

    return tile == self.mBlockingTile or entity ~= nil
end
```

Listing 2.109: Updating the IsBlocked function to work for entities. In Map.lua.

In Listing 2.109 we've added an extra check for an entity at the passed-in location. Run the code and everything *seems* to work fine, but there is a subtle issue here. If your two entities try to move to the same tile in the same frame, there's nothing to say they can't and they'll end up occupying the same tile! This is shown in Figure 2.44.

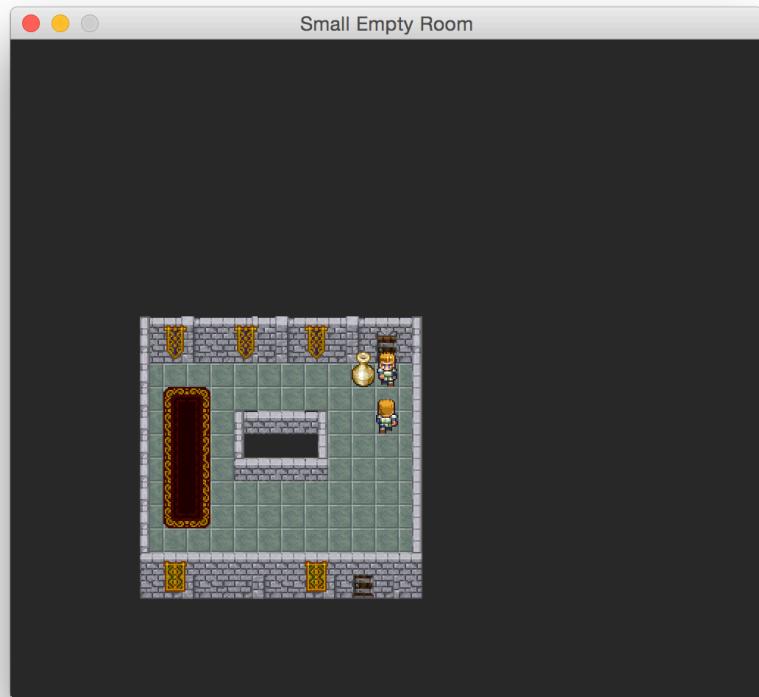


Figure 2.44: Two characters see an empty tile and move toward it!

Obviously we don't want collisions to occur. Luckily the solution is quite simple. As soon as a character decides they're going to move, we mark its entity as occupying the destination tile. Then when any other character checks the tile they'll see it's occupied and they can't move there. Any character deciding to move to a tile that another character is departing will arrive at exactly the same time the occupying character leaves so there will be no collision.

Update the MoveState, Enter and Exit functions as shown in Listing 2.110.

```
function MoveState:Enter(data)

    -- code omitted

    if self.mMoveX ~= 0 or self.mMoveY ~= 0 then
        local trigger = self.mMap:GetTrigger(self.mEntity.mLayer,
```

```

        self.mEntity.mTileX,
        self.mEntity.mTileY)

    if trigger then
        trigger:OnExit(self.mEntity)
    end
end

self.mEntity:SetTilePos(self.mEntity.mTileX + self.mMoveX,
                       self.mEntity.mTileY + self.mMoveY,
                       self.mEntity.mLayer,
                       self mMap)
self.mEntity.mSprite:SetPosition(pixelPos)
end

function MoveState:Exit()

    local trigger = self mMap:GetTrigger(self.mEntity.mLayer,
                                         self.mEntity.mTileX,
                                         self.mEntity.mTileY)

    if trigger then
        trigger:OnEnter(self.mEntity, x, y, layer)
    end

end

```

Listing 2.110: Updating the MoveState to avoid collisions. In MoveState.lua.

Listing 2.110 moves the exit trigger check from the MoveState:Exit function to the MoveState:Enter function. As soon as an entity starts to move to a tile, we call SetTilePos, which marks the tile it's occupying as free and the tile it's moving to as occupied. To stop this from affecting the smooth movement between tiles, we reset the sprite position and prevent the entity from suddenly jumping forward a tile.

There's one final issue that we're *not* going to fix (though I encourage you to take a crack at it.⁷). If an entity goes through a doorway, a trigger fires and they're teleported across the map. The teleport doesn't check the destination tile, so there can be a collision! The game we're making doesn't have free-roaming NPCs so this won't be a problem, therefore we're not going to add code to handle this case.

Example npc-5 has the code so far, with the latest collision checks. Example npc-6 is a test program with a lot of strolling NPCs. We're nearing the end of this chapter. The last issue we're going to touch on is making the triggers data driven.

⁷If you want to have a go at fixing this, a good approach might be to have the trigger check its teleport target before allowing an entity to enter its trigger area. You may need to add an additional trigger callback something like OnTryToEnter.

Data Driven Triggers

Triggers are hard coded at the top of the main file but this doesn't scale well when we want to make multiple maps. Let's add the triggers to the map definition table and then generate the triggers when the map gets created. This process is similar to how we handle NPCs and entities.

We're going to have three different definition tables: one for the actions, one for the trigger types, and one for the trigger placements. Why separate out the trigger definition from the placement? Well, imagine you have a big castle gate, several tiles wide, and you want each tile of the gate passage to be a trigger to a new map. The easiest way to do this is to create *one* trigger and place it on every tile of the game passage instead of creating a new trigger for each tile.

In main.lua we manually add the trigger tables to the mapDef but later we'll move this data into map files directly. Copy the definition tables from Listing 2.111.

```
-- code omitted
Asset.Run("small_room.lua")

local mapDef = CreateMap1()
mapDef.on_wake =
{
    {
        id = "AddNPC",
        params = {{ def = "strolling_npc", x = 11, y = 5 }},
    },
    {
        id = "AddNPC",
        params = {{ def = "standing_npc", x = 4, y = 5 }},
    }
}
mapDef.actions =
{
    tele_south = { id = "Teleport", params = {11, 3} },
    tele_north = { id = "Teleport", params = {10, 11} }
}
mapDef.trigger_types =
{
    north_door_trigger = { OnEnter = "tele_south" },
    south_door_trigger = { OnEnter = "tele_north" }
}
mapDef.triggers =
{
    { trigger = "north_door_trigger", x = 11, y = 2},
    { trigger = "south_door_trigger", x = 10, y = 12},
}
```

```
local gMap = Map:Create(mapDef)
-- code omitted
```

Listing 2.111: Adding triggers to the map definition. In main.lua.

In Listing 2.111 we define two triggers, each using a different action, the same as we had previously. One trigger on the north door teleports a character to the south and vice versa. There are three tables, actions, trigger_types and triggers. We use these definitions to construct the actions and place the trigger objects when the map is created.

The map constructor is in charge of parsing these three new tables. Copy the code from Listing 2.112 into your map.lua file. This code looks pretty hairy but we'll go through it line by line and see what's going on.

```
function Map:Create(mapDef)

    -- code omitted

    --
    -- Create the Actions from the def
    --

    this.mActions = {}
    for name, def in pairs(mapDef.actions or {}) do

        -- Look up the action and create the action-function
        -- The action takes in the map as the first param
        assert(Actions[def.id])
        local action = Actions[def.id](this, unpack(def.params))
        this.mActions[name] = action
    end

    --
    -- Create the Trigger types from the def
    --

    this.mTriggerTypes = {}
    for k, v in pairs(mapDef.trigger_types or {}) do
        local triggerParams = {}
        for callback, action in pairs(v) do
            print(callback, action)
            triggerParams[callback] = this.mActions[action]
            assert(triggerParams[callback])
        end
        this.mTriggerTypes[k] = Trigger:Create(triggerParams)
    end
```

```

setmetatable(this, self)

-- Place any triggers on the map
--

this.mTriggers = {}
for k, v in ipairs(mapDef.triggers) do
    local x = v.x
    local y = v.y
    local layer = v.layer or 1

    if not this.mTriggers[layer] then
        this.mTriggers[layer] = {}
    end

    local targetLayer = this.mTriggers[layer]
    local trigger = this.mTriggerTypes[v.trigger]
    assert(trigger)
    targetLayer[this:CoordToIndex(x, y)] = trigger
end

-- code omitted

end

```

Listing 2.112: Creating triggers from the map definition. In Map.lua.

In the Map constructor we start unpacking the data by looping over each of the newly added tables.

Each entry in the actions table has this sort of format `tele_south = { id = "Teleport", params = {11, 3} }`. We use the key value `tele_south` as the action's name, we use `id` to look up the action type, and we use the `params` table to create the action object. A map isn't required to have any actions therefore we have an `or {}` which means, use an empty table if no actions are defined in the def. In the for-loop we use an assert to ensure the action `id` refers to a real action.

The action is added to the `mActions` table using its name, e.g. `tele_south`, as the key. This name key becomes important later when we build the triggers.

After creating the actions we create the trigger types. First we create an `mTriggerTypes` table to store the trigger types in the map. The map isn't required to have a `trigger_types` table so we use `or {}` to make the table optional. As with the `mActions` table, the keys are the names of the trigger types. Each entry in the `trigger_types` table uses the format `north_door_trigger = { OnEnter = "tele_north" }`, which is

a string key pointing to a table of callbacks. We loop through each trigger definition, create the trigger, and store it in this.mTriggerTypes.

To create the trigger object we need to link up the callbacks to actual actions. The trigger def has callback strings that we use to look up the relevant action in the mActions table. All our callback information is stored in the triggerParams table, and once we've filled this we use it to create a Trigger. We store the trigger by name in the mTriggerTypes table in the Map.

After filling the mActions and mTriggerTypes tables we're ready to place the triggers on the map. Trigger location information is stored in mapDef.triggers. Before parsing this last table we make sure the Map's metatable has been set because we need to use the Map's functions. Each entry in the mapDef.triggers table is made of an id and an x, y position on the map. For each entry we make sure the layer where we're placing the trigger exists, then we transform the x, y coord to an tile index and use that to place the trigger. We get the trigger object from the mTriggerTypes table.

Once all the triggers are placed, that's the setup code done. All of our trigger information is now data driven. The code in main.lua as shown in Listing 2.113 shows the current setup of the map. Run this code and the triggers are now generated and placed using the mapDef.

```
-- code omitted
Asset.Run("small_room.lua")

local mapDef = CreateMap1()
mapDef.on_wake =
{
    {
        id = "AddNPC",
        params = {{ def = "strolling_npc", x = 11, y = 5 }},
    },
    {
        id = "AddNPC",
        params = {{ def = "standing_npc", x = 4, y = 5 }},
    }
}
mapDef.actions =
{
    tele_south = { id = "Teleport", params = {11, 3} },
    tele_north = { id = "Teleport", params = {10, 11} }
}
mapDef.trigger_types =
{
    north_door_trigger = { OnEnter = "tele_north" },
    south_door_trigger = { OnEnter = "tele_south" }
}
```

```

mapDef.triggers =
{
    { trigger = "north_door_trigger", x = 11, y = 2},
    { trigger = "south_door_trigger", x = 10, y = 12},
}
local gMap = Map:Create(mapDef)
gRenderer = Renderer:Create()
gMap:GotoTile(5, 5)
gHero = Character:Create(gCharacters.hero, gMap)
gHero.mEntity:SetTilePos(11, 3, 1, gMap)

function GetFacedTileCoords(character)
-- code omitted

```

Listing 2.113: Extending the map definition with triggers. In Main.lua.

Example npc-7 contains the latest code. If you run the program you'll see that everything is working as before. Maps can now be easily extended by adding all sorts of triggers into the mapDef directly. In the next chapter we'll move on to the user interface code.

User Interface

RPGs have a surprising amount of menus and a large user interface. Luckily for us, most of the user interface is built up from simple reusable UI elements. In this chapter we'll build a toolbox of common UI elements; from the ubiquitous textbox, to the progress bar and basic selection list. Armed with these elements we'll be prepared for our UI needs in the coming chapters.

Panel

Panels are rectangles that often have a decorative border. A typical example panel can be seen in Figure 2.45. The panel is one of the foundational UI elements in RPGs; they're used in menus, conversations, and dialog boxes. Nearly all UI code is built on top of this little box.

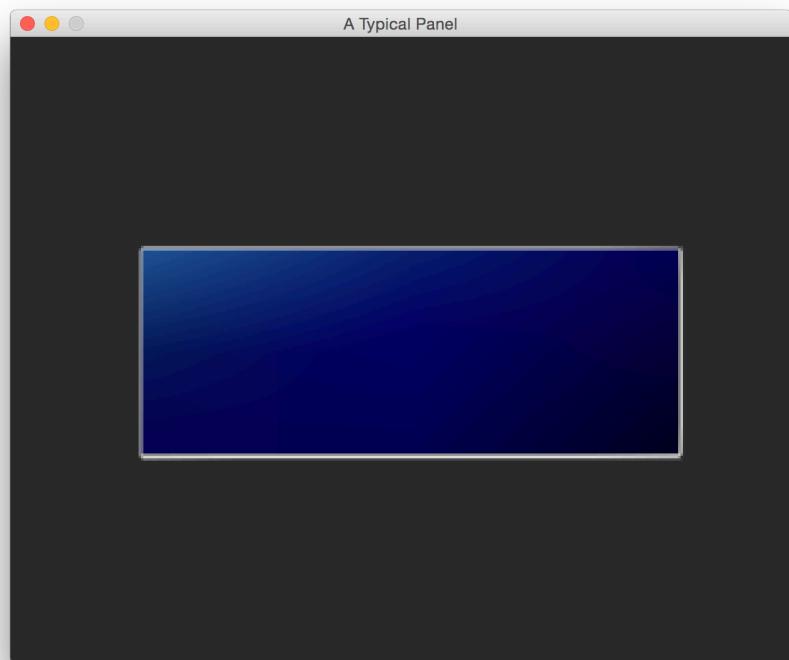


Figure 2.45: A common RPG panel.

Panels are used to frame menus and separate out different areas by use. The equipment menu shown in Figure 2.46 is common, in some form, to most RPGs. Notice how the panels are used to build up the structure of the screen. On the top there are details about the selected party member, in the middle a box for text about selected equipment, the bottom left has stats, and the bottom right a list of inventory items. Each area of the menu is marked out using a panel.



Figure 2.46: An equip menu screen made out of panels.

A good question to ask, when designing a UI element, is “How will we use this and what should it be able to do?” Looking at examples is a good way to get the answers to this question. Examples, such as Figure 2.46, show the range of uses for a panel element. At the very least we should be able to give a top left pixel position and a bottom right pixel position and have a panel fit exactly in that area. This way of constructing panels makes it easier to construct menu screens.

A panel class should have simple support for decorative edging. A good way to handle decorative edging is by constructing the panel out of nine sprites: four corner sprites, one central backing panel sprite, and four border sprites. The corner sprites are a fixed size but the sprites that make up the border are stretched vertically or horizontally depending on which side of the panel they appear. The center sprite is also scaled to the size of the panel minus the borders. Figure 2.47 shows how a texture is used to construct one of these boxes.

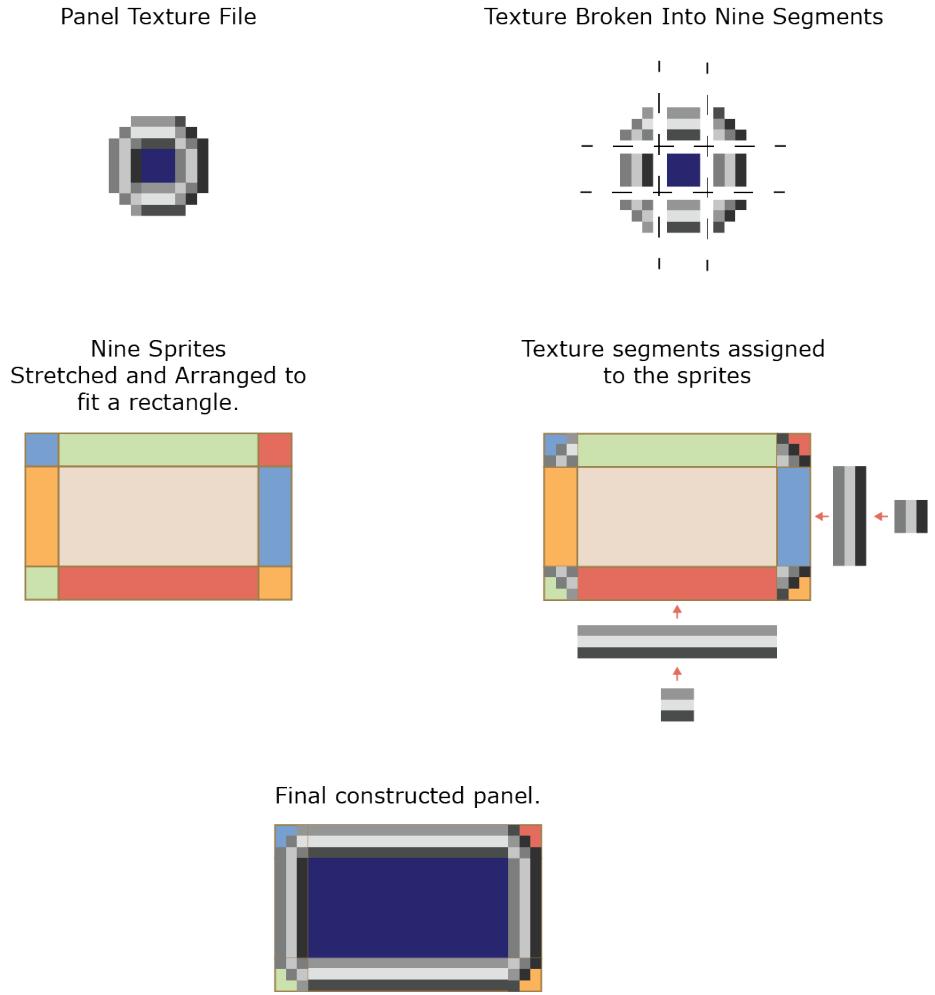


Figure 2.47: Using 9 sprites to construct a resizable panel from a simple texture.

Now that we have an idea of how we want our panel to work, let's create a project and start implementing it. Example gui-1 is an empty project containing a texture file called `simple_panel.png`. You can see it in Figure 2.48 with some dotted lines to show how it's broken up. It's only 9 by 9 pixels in size, so it gives a very slim textbox border.

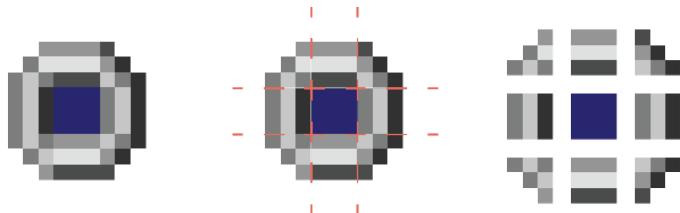


Figure 2.48: The border texture we'll use to make a textbox.

The textbox code scales up parts of texture. Graphics cards have multiple ways to scale textures. Because we're dealing with pixel art we want to keep the scaling crisp with no interpolation. In Dinodeck this is done with a `pixelart` flag in the manifest as shown in Listing 2.114. Try removing the flag in some of the examples and observe the changes to scaled textures.

```
['simple_panel.png'] =
{
    path = "simple_panel.png",
    scale = "pixelart"
}
```

Listing 2.114: Setting the scale to 'pixelart' to stop images become blurry when scaled. In manifest.lua.

Example gui-1 contains the texture shown in Figure 2.48, its manifest entry, and the Util.lua file. The Util.lua file contains code for cutting up textures using UV coordinates. Finally there's an empty file Panel.lua where we'll write the panel code.

The first order of business is to create a simple constructor that tells the panel which texture to use and how to correctly break the texture up. The constructor takes in a params table. Listing 2.115 shows the basic class and constructor. In the constructor we refer to each individual piece of the textbox as a tile.

```
Panel = {}
Panel.__index = Panel
function Panel:Create(params)

    local this =
    {
        mTexture = params.texture,
        mUVs = GenerateUVs(params.size, params.size, params.texture),
        mTileSize = params.size,
        mTiles = {}, -- the sprites representing the border.
    }

    -- Create a sprite for each tile of the panel
    -- 1. top left      2. top          3. top right
    -- 4. left          5. middle       6. right
    -- 7. bottom left   8. bottom      9. bottom right
    for k, v in ipairs(this.mUVs) do
```

```

local sprite = Sprite:Create()
sprite:SetTexture(this.mTexture)
sprite:SetUVs(unpack(v))
this.mTiles[k] = sprite
end

setmetatable(this, self)
return this
end

```

Listing 2.115: The panel constructor creates 9 sprites from the texture. In Panel.lua.

Listing 2.115 shows the panel constructor. It uses two parameters from the params table, texture and size. The texture is the texture used to decorate the box. The size is the pixel size of each tile and it's used to divide the texture into nine sprites. The nine sprites are stored in the mTiles table.

To actually use the panel we need a function, Position, to define its position and size on screen. Copy this function from Listing 2.116.

```

function Panel:Position(left, top, right, bottom)

    -- Reset scales
    for _, v in ipairs(self.mTiles) do
        v:SetScale(1, 1)
    end

    local hSize = self.mTileSize / 2

    -- Align the corner tiles
    self.mTiles[1]:SetPosition(left + hSize, top - hSize)
    self.mTiles[3]:SetPosition(right - hSize, top - hSize)
    self.mTiles[7]:SetPosition(left + hSize, bottom + hSize)
    self.mTiles[9]:SetPosition(right - hSize, bottom + hSize)

    -- Calculate how much to scale the side tiles
    local widthScale = (math.abs(right - left) - (2 * self.mTileSize))
        / self.mTileSize
    local centerX = (right + left) / 2

    self.mTiles[2]:SetPosition(centerX, top - hSize)
    self.mTiles[2]:SetScale(widthScale, 1)

    self.mTiles[8]:SetPosition(centerX, bottom + hSize)
    self.mTiles[8]:SetScale(widthScale, 1)

```

```

local heightScale = (math.abs(bottom - top) - (2 * self.mTileSize))
/ self.mTileSize
local centerY = (top + bottom) / 2

self.mTiles[4]:SetScale(1, heightScale)
self.mTiles[4]:SetPosition(left + hSize, centerY)

self.mTiles[6]:SetScale(1, heightScale)
self.mTiles[6]:SetPosition(right - hSize, centerY)

-- Scale the middle backing panel
self.mTiles[5]:SetScale(widthScale, heightScale)
self.mTiles[5]:SetPosition(centerX, centerY)

-- Hide corner tiles when scale is equal to zero
if left - right == 0 or top - bottom == 0 then
    for _, v in ipairs(self.mTiles) do
        v:SetScale(0, 0)
    end
end
end

```

Listing 2.116: Placing the panel and all the tiles it's made up from. In Panel.lua.

The Position function takes in four parameters: left, top, right, and bottom, that form two points, the top left and bottom right. These points describe a rectangular space for the panel to occupy.

In the constructor we filled the mTiles table with nine tile sprites. The Position function resets all the tiles so we know there's no scale applied. Then it takes the corner tiles and aligns them to the corners of the rectangle. All sprites are drawn from their center, so to perfectly align them they're adjusted by half their width and height. Once this is done, four of the tiles are correctly placed.

After placing the corners, we place and scale the border tiles. The amount of scaling required is calculated for the horizontal sides by working out the width of the rectangle and then subtracting the corner tiles' width. The tile width is subtracted so we don't overlap the corner tiles. Calculating the vertical scale factor is done in the same as way as the horizontal but uses the rectangle's height. Each border tile is positioned centrally between two of the corner tiles. Once this is done only the central tile remains to be placed.

The middle tile is scaled to fill all remaining empty space in the center of the panel. The position and scale numbers we use for the side tiles are reused here for the backing

panel - its vertical scale is the same as the left (or right) tile and its horizontal scale is the same as the top (or bottom) tile.

Before leaving the function, there's a loop that checks if the current textbox size is 0. If it is then the loop scales all the tiles to zero. If this wasn't here the corner tiles would still get drawn, creating an odd visual artifact on the screen.

Next let's add a function to render the tiles to the screen. Copy the code from Listing 2.117. All it does is loop through the tiles and draws each one.

```
function Panel:Render(renderer)
    for k, v in ipairs(self.mTiles) do
        renderer:DrawSprite(v)
    end
end
```

Listing 2.117: Rendering the tiles of a panel. In Panel.lua.

Ok, our panel is now ready for testing. Let's write some test code in main.lua. Listing 2.118 shows how to use and render the panel.

```
-- code omitted
gRenderer = Renderer.Create()

local gPanel = Panel:Create
{
    texture = Texture.Find("simple_panel.png"),
    size = 3,
}
local left = -100
local top = 0
local right = 100
local bottom = -100
gPanel:Position(left, top, right, bottom)

function update()
    gPanel:Render(gRenderer)
end
```

Listing 2.118: Using the panel class in main.lua.

In Listing 2.118 we create a panel, using a texture and size parameter, and store it as gPanel. The size parameter determines how the texture is cut up to make the nine tiles of the panel. Then we define the left, top, right, and bottom positions to

describe a rectangle where the panel will be rendered. Panel.Position is called using the position information which scales and places all the panel's tiles where we want them. In the update loop we call the render function and it renders all the panel's tiles.

Example gui-1-solution contains the code so far. If you run the program you'll get a result as shown in Figure 2.49. Our first panel - great!

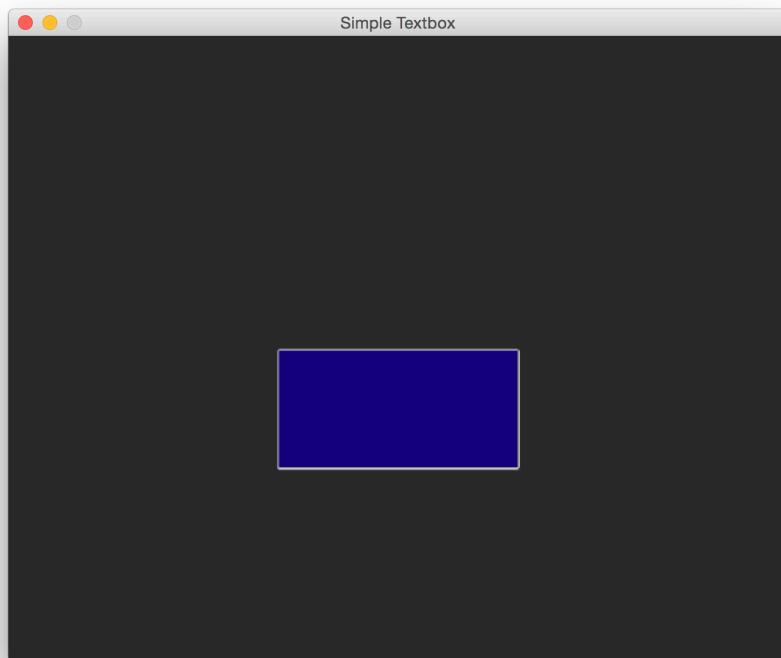


Figure 2.49: Our first panel.

Positioning a Panel From the Center

The panel has a Position function which works well when making panels for fullscreen menus but which isn't so nice to use when we want a dialog box. Let's add a new CenterPosition function that makes a panel centered on an X, Y coordinate and lets us specify a width and a height. Copy the code from Listing 2.119.

```
function Panel:CenterPosition(x, y, width, height)
    local hWidth = width / 2
    local hHeight = height / 2
```

```
    return self:Position(x - hWidth, y + hHeight,
                          x + hWidth, y - hHeight)
end
```

Listing 2.119: Drawing a panel from the center. In Panel.lua.

Listing 2.119 shows a function that positions a panel around a center point. If we want to pop up a textbox it's much easier to use CenterPosition rather than the Position function.

Update main.lua as shown in Listing 2.120 so we can test out this new code. Listing 2.120 draws a panel in the center of the screen and we've also added some "Hello World" text to give a taste of what's to come.

```
local gPanel = Panel>Create
{
    texture = Texture.Find("simple_text_box.png"),
    size = 3,
}
gPanel:CenterPosition(0, 0, 128, 32)

function update()
    gPanel:Render(gRenderer)

    gRenderer:AlignText("center", "center")
    gRenderer:DrawText2d(0,0, "Hello World", Vector.Create(1,1,1,1))
end
```

Listing 2.120: A Hello World textbox. In main.lua.

Example gui-2 contains the code so far. Run the program and you'll see a panel with "Hello World" written in it as shown in Figure 2.50.

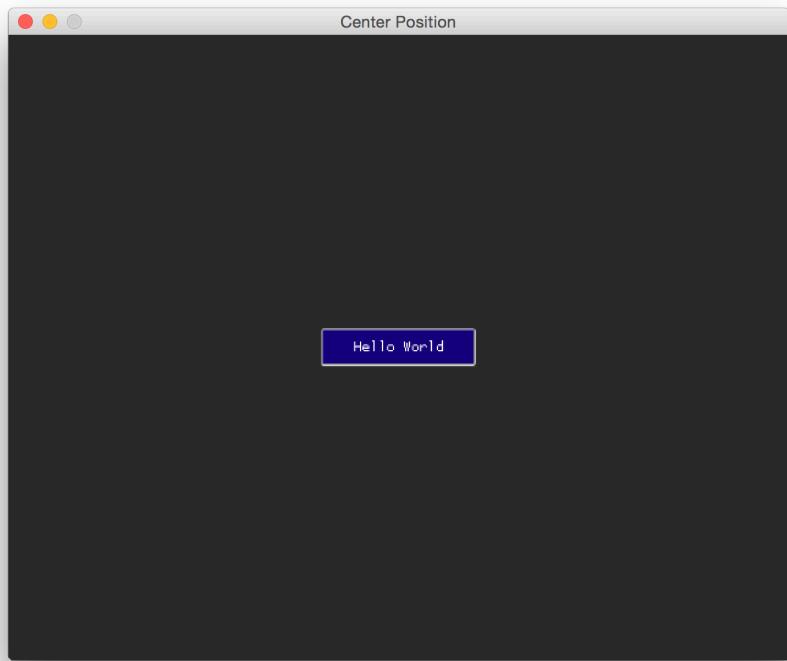


Figure 2.50: A Hello World textbox demonstrating positioning textboxes from their center.

Adding a Gradient to the Panel

The middle section of the panel we've been rendering is plain blue. It would be nice if the background was a little more interesting and had a gradient. Open the example `gui-3` and you'll find a version of the panel project using a different texture, "gradient_panel.png". View the texture and you'll see that the center 3x3 tile isn't all one color. Instead, each pixel is a slightly different shade of blue and if you zoom out it looks a little like a gradient. This texture is included in the manifest but it does not set the scale flag to `pixelart`. We want this texture to be interpolated so our pixels blend into a smooth gradient. There's one problem with this. Run the code and you'll get an image like in Figure 2.51.

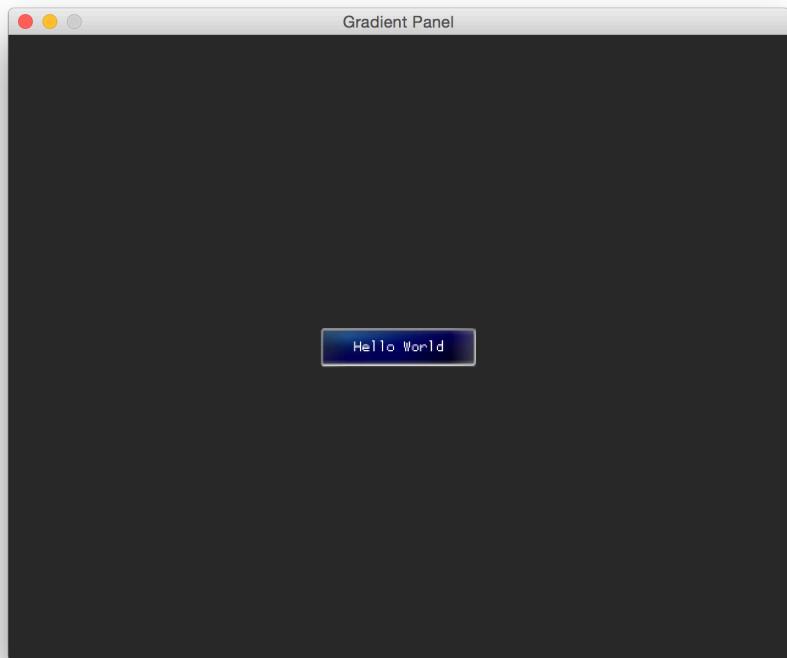


Figure 2.51: Interpolation occurring on the panel between the border and middle tiles.

The panel looks kind of cool but the white from the border is being blended into the center panel. What we really want is a crisp clean gradient in the center panel and no blending in with the border decoration. To make this happen we're going to use our underlying knowledge of how the graphics pipeline works. This bit may seem a bit black magic if you haven't had much exposure to graphics programming, and you're free to skip over it if it seems to be digging into the details too much.

We're going to move the U,V coordinates for the middle tile in by half a pixel, or texel⁸ to use the correct term. Moving the center tile's UVs stops them sampling the border pixels and gives us a crisp gradient for the middle panel.

The center tiles' UVs on the left side and right sides are reduced by half a pixel. We do the same for the top and bottom too; in total we reduce the middle panels U,V coordinates by 1, 1. Once that's done our gradient works as expected.

⁸Texel comes from the phrase *texture element*. A texel is a pixel in a texture image. It uses the word texel instead of the word pixel because after a texture is applied to some mesh and rendered to the screen, there's no 1:1 mapping of texels to pixels.

Let's consider an example. Say you have a texture of 1 pixel by 1 pixel and it's red. Then you draw a 3d cube in the middle of your scene, using the red texture. In the final rendering the cube takes up a lot of pixels on the screen, but it's all created from one texel of texture data.

We change the middle panel's UVs in the `TextBox:Create` constructor as shown in Listing 2.121.

```
function Panel:Create(params)

    local this =
    {
        -- code omitted
    }

    -- Fix up center U,Vs by moving them 0.5 texels in.
    local center = this.mUVs[5]
    local pixelToTexelX = 1 / this.mTexture:GetWidth()
    local pixelToTexelY = 1 / this.mTexture:GetHeight()
    center[1] = center[1] + (pixelToTexelX / 2)
    center[2] = center[2] + (pixelToTexelY / 2)
    center[3] = center[3] - (pixelToTexelX / 2)
    center[4] = center[4] - (pixelToTexelY / 2)

    -- The center sprite is going to be 1 pixel smaller on the X, Y
    -- So we need a variable that will account for that when scaling.
    this.mCenterScale = this.mTileSize / (this.mTileSize - 1)

    -- code omitted
}

function Panel:Position(left, top, right, bottom)

    -- code omitted

    -- Scale the middle backing panel
    self.mTiles[5]:SetScale(widthScale * self.mCenterScale,
                           heightScale * self.mCenterScale)
    self.mTiles[5]:SetPosition(centerX, centerY)
end

-- code omitted
```

Listing 2.121: Reducing the size of the middle panel's UV rectangle by 1 pixel. In `Panel.lua`.

In this snippet we create a scale factor for the middle panel. Dinodeck sets a sprite's size using its UVs, and reducing the middle panel's UVs means it's now smaller than the other tiles. To correct the center tile's scale we call SetScale using mCenterScale to adjust for differences in the UVs.

This updated code is available in example gui-3-solution. Run it now and you'll see a textbox as pictured in Figure 2.52.

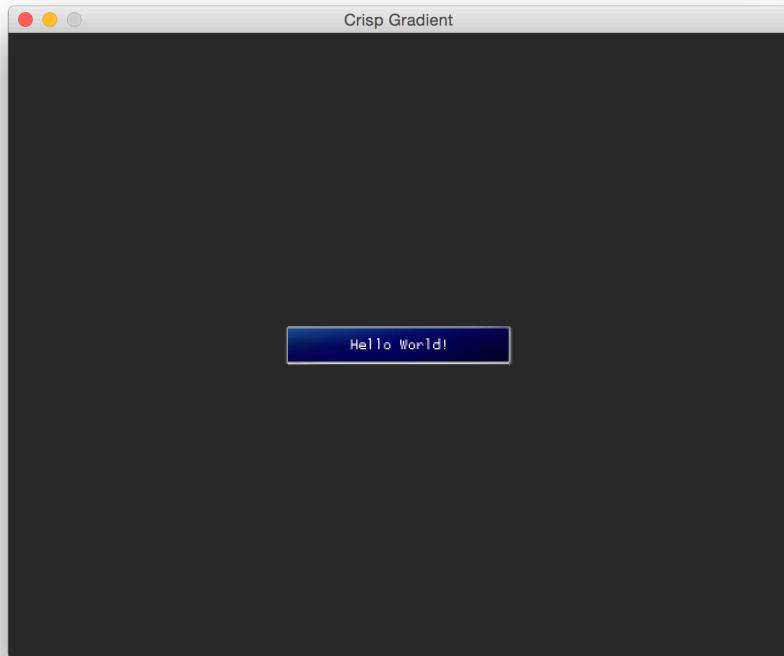


Figure 2.52: A panel with a gradient.

Textboxes

Textboxes are panels that contain text. They're used to represent speech⁹ from a character in the world or to send some game message to the user.

Textboxes may include options for the player to choose from (such as Yes / No), images and a title. What we want to do is create a simple flexible textbox that we can use for many different jobs in our RPG. At a high level a textbox is made from a rectangle

⁹Textboxes used to represent character speech are also referred to as *Dialog Boxes*.

describing the absolute size of the box and an inner rectangle denoting where the text is rendered. Check out Figure 2.54 for the basic structure of the textbox.

The text area prevents any text drawing over the box's decorative edging and later makes it easier to bolt on portraits and that type of thing. We're going to start off looking at a simple textbox that sizes itself according to the text it contains. Then we'll look at altering the textbounds to add a picture and title to the textbox. Finally we'll add textboxes of a fixed height and width that automatically break up long pieces of text between several boxes. This makes life easier when displaying lengthy dialog text.

Absolute Size

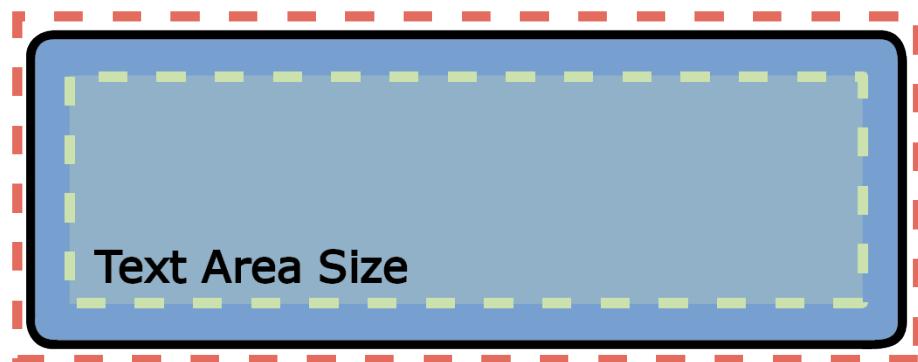


Figure 2.53: A textbox made up from a size rectangle and text area using a panel.

In Figure 2.54 you can see there are two boxes. The outer absolute bounds is filled by a panel, then the text area is a rectangle inside that panel. The green dotted line represents the textbounds, and this is where text is drawn. The textbounds are made up of four variables, left, right, top, and bottom, and these are offsets from the absolute bounds. Therefore if we wanted a text area that had 10 pixels of padding we'd set the text area bounds to +10, -10, -10, +10 to position the text nicely in the middle of the panel.

Let's write some code and make our first textbox. Example gui-4 contains the panel code we wrote in the previous section as well as an empty lua file called Textbox.lua. Textbox.lua has been added to the project in the manifest and loaded in main.lua.

We're going to start with a little pseudo-code, shown in Listing 2.122 imagining how we'd like to use a textbox.

```
local textbox = Textbox:Create
{
    text = "hello",
    textSize = 2,
    size =
```

```

{
    left = -100,
    right = 100,
    top = 32,
    bottom = -32,
},
textbounds =
{
    left = 10,
    right = -10,
    top = -10,
    bottom = 10
},
panelArgs =
{
    texture = Texture.Find("gradient_panel.png"),
    size = 3
}
}

function update()
    local dt = GetDeltaTime()
    textbox:Render(gRenderer)
end

```

Listing 2.122: How we might use the textbox. In main.lua.

Listing 2.122 contains a lot of code just to set up a textbox! Don't worry, we'll be wrapping this code up into simpler functions.

Let's go through it. Textbox is a class, and its constructor expects a table as the only parameter. The setup table has five fields.

text The text to write in the textbounds of the box.

textsclae An optional argument that determines size of the font. By default it's 1 but we're setting it to 2 here.

size The absolute size of the textbox. Nothing is drawn outside of this rectangle and the backing panel fills the entire area.

textbounds The left, right, top and bottom offsets from the size rectangle. The textbounds is where the text is written. Characters are never drawn outside of this box. In the example we can see that we've added a 10-pixel border of padding to stop the text being written right up against the borders.

panelArgs Describes the backing panel appearance and uses the same parameters we saw in the previous section.

After the textbox setup code there's the update loop that renders it out. Figure 2.54 shows the type of textbox this code generates with some guidelines showing the size and textbounds.

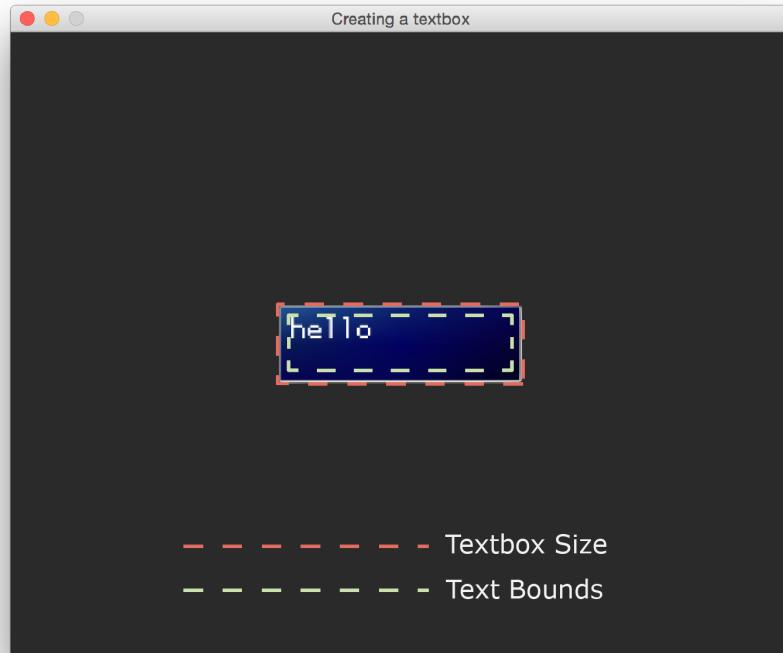


Figure 2.54: An example textbox our pseudo-code might create.

Now that we have an idea of what we want, it's time to get coding! Copy the code from Listing 2.123. Listing 2.123 contains the basic constructor for the textbox.

```
Textbox = {}
Textbox.__index = Textbox
function Textbox:Create(params)

    params = params or {}

    local this =
```

```

{
    mText = params.text,
    mTextScale = params.textScale or 1,
    mPanel = Panel>Create(params.panelArgs),
    mSize = params.size,
    mBounds = params.textbounds,
}

-- Calculate center point from mSize
-- We can use this to scale.
this.mX = (this.mSize.right + this.mSize.left) / 2
this.mY = (this.mSize.top + this.mSize.bottom) / 2
this.mWidth = this.mSize.right - this.mSize.left
this.mHeight = this.mSize.top - this.mSize.bottom

setmetatable(this, self)
return this
end

```

Listing 2.123: The textbox constructor. In Textbox.lua.

The constructor adds the fields from the params table into its this table. A backing panel, mPanel, is constructed using the Panel class. The center of the panel is calculated along with the width and height. We work out the width and height because later we're going to add a small special effect to the textbox and have it scale up from nothing. In order to add this special effect we need these values.

Next let's write the render function. Copy the code from Listing 2.124.

```

function Textbox:Render(renderer)
    local scale = 1

    renderer:ScaleText(self.mTextScale * scale)
    renderer:AlignText("left", "top")
    -- Draw the scale panel
    self.mPanel:CenterPosition(
        self.mX,
        self.mY,
        self.mWidth * scale,
        self.mHeight * scale)

    self.mPanel:Render(renderer)

    local left = self.mX - (self.mWidth/2 * scale)
    local textLeft = left + (self.mBounds.left * scale)

```

```

local top = self.mY + (self.mHeight/2 * scale)
local textTop = top + (self.mBounds.top * scale)

renderer:DrawText2d(
    textLeft,
    textTop,
    self.mText,
    Vector.Create(1,1,1,1))
end

```

Listing 2.124: The textbox render function. In Textbox.lua.

The Textbox:Render function centers and renders the backing panel, then it renders the text using the textbounds. The top left of the text is aligned to the top left of the textbounds.

Later we're going to scale the textbox but for now, in the Render function, the scale is hard coded to 1. We scale the text size using the scale factor and set its alignment to the "top left". Then we arrange the backing panel around the textbox position, scaling the width and height by the scale factor.

In order to draw the text from the top left corner of the textbounds we calculate the top, left of the panel and store the results in the variables top and left. Then we apply the textbounds offsets to get the point to draw the text and store these values in textLeft and textTop. These positions apply the scaling factor so the text position is correct even when we scale the box.

With the Render function defined we can try it out. Example gui-4-solution contains the code so far. Run it and you'll see the textbox with "Hello" written in it. It doesn't look too amazing at the moment but we've laid the groundwork.

Textbox Transitions

The textbox we've created is quite plain so let's add a simple transition to make it appear a little more flashy. Example gui-5 has the code so far, or you can continue on with your own codebase. The example folder contains the Tween.lua file. The Tween class was covered back in the introduction, and we'll use it to make the box scale up from nothing and scale back down to nothing when dismissed. Like last time, let's first consider how we'd like to use it in the main function, then dive into the implementation code. See Listing 2.125.

```

gStartText = false
function update()

    if Keyboard.JustPressed(KEY_SPACE) then

```

```

        gStart = true
    end

    if not gStart then
        gRenderer:AlignText("center", "center")
        gRenderer:DrawText2d(0, 0, "Press space.")
        return
    end

    if not textbox:IsDead() then
        local dt = GetDeltaTime()
        textbox:Update(dt)
        textbox:Render(gRenderer)
    end

    if Keyboard.JustPressed(KEY_SPACE) then
        textbox:OnClick()
    end
end

```

Listing 2.125: Creating a textbox to scale. In main.lua.

In Listing 2.125 there's code to pop up a textbox when the space key is pressed. This code contains two new textbox functions, Update and IsDead, which we'll define shortly. At the start of the code there's a check to see if the space key has been pressed. If not we display a "Press space." message and exit. When space is pressed the gStart flag is set to true and the textbox begins to get rendered and updated. If the user presses space again then the OnClick function is called and the textbox is dismissed. Once the dismiss animation finishes, the IsDead function returns true and the textbox no longer renders.

Let's update the Textbox class to make the code in Listing 2.125 work. Copy the updated code from Listing 2.126.

```

function Textbox>Create(params)

    params = params or {}

    local this =
    {
        -- code omitted
        mAppearTween = Tween:Create(0, 1, 0.4, Tween.EaseOutCirc),
    }
    -- code omitted
end

```

```

function Textbox:Update(dt)
    self.mAppearTween:Update(dt)
end

function Textbox:OnClick()
    --
    -- If the dialog is appearing or disappearing
    -- ignore interaction
    --
    if not (self.mAppearTween:IsFinished()
        and self.mAppearTween:Value() == 1) then
        return
    end
    self.mAppearTween = Tween:Create(1, 0, 0.2, Tween.EaseInCirc)
end

function Textbox:IsDead()
    return self.mAppearTween:IsFinished()
        and self.mAppearTween:Value() == 0
end

function Textbox:Render(renderer)
    local scale = self.mAppearTween:Value()
    -- code omitted
end

```

Listing 2.126: Adding a simple in-out transition to the textbox. In Textbox.lua.

In Listing 2.126 we add a tween, `mAppearTween`, to handle the in and out transitions (when the textbox appears on the screen and when it disappears from the screen). For our textbox the in-transition scales it up from 0 to 1 and the out-transition scales it down from 1 to 0. The third argument passed to the `mAppearTween` controls the duration of the transition and it's set to 0.4 seconds. We're using the `EaseOutCirc` function to slow the scale rate a little as it finishes to make the transition a little bit nicer.

The new `Textbox:Update` function just updates the `mAppearTween` tween. The `mAppearTween` value is used in the `Render` function to display the transition effect.

The `OnClick` function represents an interaction with the textbox such as a button press or mouse click. Our `OnClick` function dismisses the textbox. It reverses the `mAppearTween` to make the textbox shrink down to nothing. The `OnClick` is exited immediately if the textbox is already transitioning. The `IsDead` function returns true if the tween has finished and the dialog has been scaled to 0, otherwise it returns false.

In the `Render` function the value of `scale` is set from the `mAppearTween`'s value.

Example gui-5-solution has the code so far. Run the program to see the textbox transitions in action. Press space to see the textbox appear from nothing and press space again to make it shrink down and disappear. Transitions make a textbox more eye-catching and less jarring than if it suddenly appeared on screen.

A Fixed Textbox

RPGs use many different types of textbox. The one we'll implement next is a *fixed* textbox. A fixed textbox is created from a position and a width and height. The textbox size doesn't change; *the width and height remain fixed*. Currently creating a single textbox is a long and pretty unfriendly process but we can make creating a fixed textbox a simple one-liner as shown in Listing 2.127. Example gui-6 contains the code so far, or you can continue to use your own codebase.

```
local textbox = CreateFixed(gRenderer, 100, -100, 70, 35, "Hello")
```

Listing 2.127: Creating a textbox of a fixed size and position. In main.lua.

Let's write the CreateFixed function in the main.lua file for now and then later we'll move it somewhere more appropriate. To create a textbox of fixed size and position we simply wrap up the existing Textbox:Create function. The width and height are translated directly into the absolute size of the box, and padding information is used to set the textbounds. See Listing 2.128 for the code.

```
function CreateFixed(renderer, x, y, width, height, text)

    local padding = 10
    local textScale = 1.5
    local panelTileSize = 3

    return Textbox:Create
    {
        text = text,
        textScale = textScale,
        size =
        {
            left     = x - width / 2,
            right    = x + width / 2,
            top      = y + height / 2,
            bottom   = y - height / 2
        },
        textbounds =
        {
            left = padding,
```

```

        right = -padding,
        top = -padding,
        bottom = padding
    }
    panelArgs =
    {
        texture = Texture.Find("gradient_panel.png"),
        size = panelTileSize,
    }
}
end

```

Listing 2.128: Creating a textbox of a certain size. In main.lua.

The majority of Listing 2.128 and the CreateFixed function is taken up by calling Textbox:Create. We first set up some variables we're going to need; how much padding we want around the text, what the text scale is going to be, and how big the texture tiles of the backing panel are in pixels. In the textbox constructor we work out the values we need to fit the absolute size of the box. We use the pad data to assign the offsets for the textbounds. Then the function is ready to use!

Copy the code from Listing 2.129 to try out the CreateFixed function. It's similar to our previous textbox example but shorter and easier to use.

```

local textbox = CreateFixed(gRenderer, 100, -100, 70, 35, "Hello")

function update()
    textbox:Update(GetDeltaTime())
    textbox:Render(gRenderer)
end

```

Listing 2.129: The main.lua, demonstrating the fixed textbox. In main.lua.

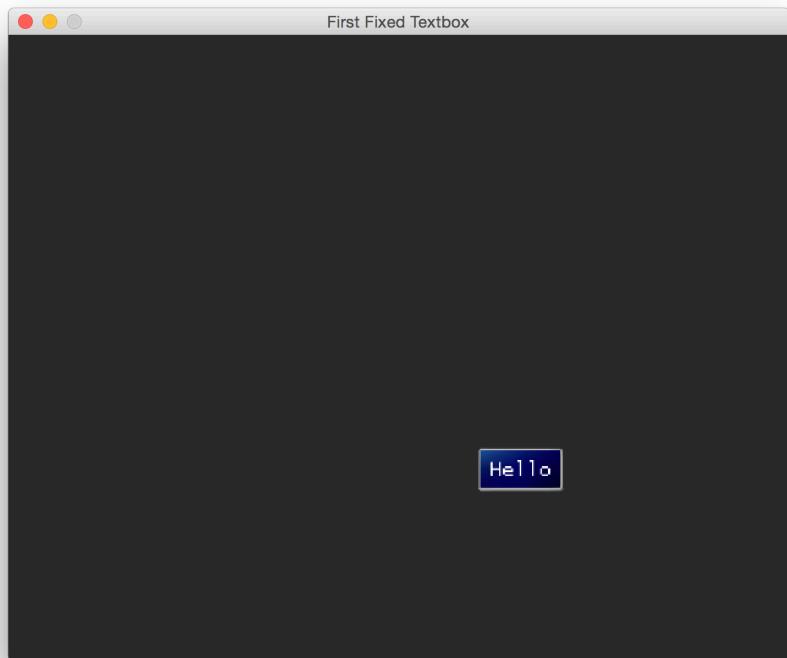


Figure 2.55: A fixed textbox with the word "Hello".

Example gui-6-solution contains the code so far. If you run the program, you'll see the text "Hello" inside a textbox as shown in Figure 2.55.

Wrapping Text

If you play around with the fitted textbox, you'll probably notice that it's very easy to create a string of text so long that it runs outside the box! Figure 2.56 shows an example of this. To fix this we need to define a maximum text length for the textbox and make the text start a new line when it's too long. The term for breaking text into lines this way is called *wrapping*. Dinodeck has pretty good text support so we can make it perform the wrapping for us with only a few code changes. Example gui-7 contains the code so far, or you can continue on with your own codebase.

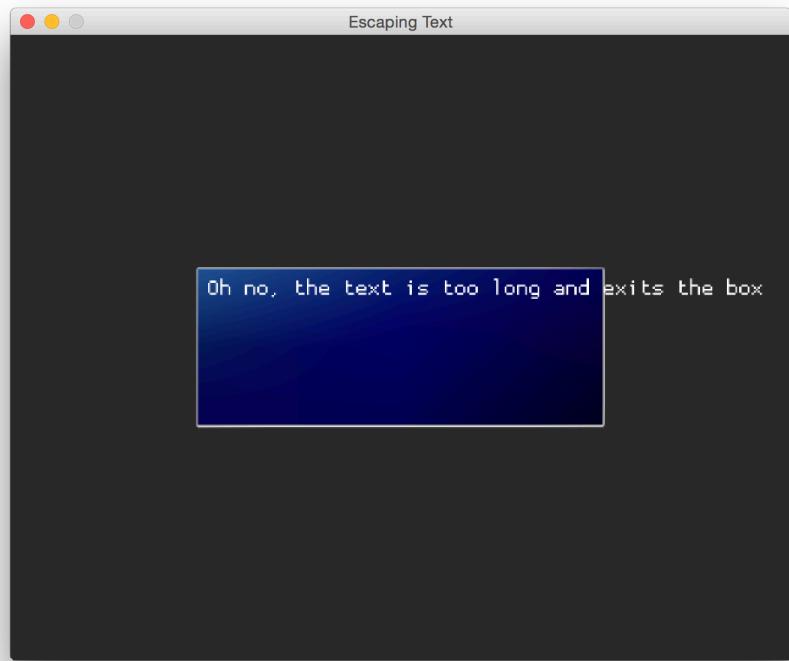


Figure 2.56: Text that's too long breaches the right side of the textbox.

To add wrapping we're going to modify the Textbox constructor to take in an extra wrap field in the params table. The wrap field is the maximum width (in pixels) a line of text can be before a new line is started. Copy the code from Listing 2.130.

```
function Textbox:Create(params)

    params = params or {}

    local this =
    {
        -- code omitted
        mWrap = params.wrap or -1
    }

    -- code omitted
end
```

```
-- code omitted

function Textbox:Render(renderer)

    -- code omitted

    renderer:DrawText2d(
        textLeft,
        textTop,
        self.mText,
        Vector.Create(1,1,1,1),
        self.mWrap * scale)
end
```

Listing 2.130: Adding a wrap parameter to the textbox. In Textbox.lua.

The wrap parameter is assigned to mWrap in the constructor. It's an optional parameter and if it's not present it defaults to -1, meaning wrapping is ignored. In the Render function the mWrap value is passed into DrawText2d and Dinodeck automatically wraps the text for us. The wrap value is multiplied by the scale so that the wrapping works even when scaling the textbox. Next let's update the CreateFixed function in main.lua to take in a wrap parameter too. Copy the code from Listing 2.131.

```
function CreateFixed(renderer, x, y, width, height, text)

    -- code omitted

    local wrap = width - padding * 2

    return Textbox>Create
    {
        wrap = wrap,
        -- code omitted
    }

end
```

Listing 2.131: Altering the CreateFixed function to support wrapping. In main.lua.

The CreateFixed function now explicitly defines the textbox width because we use the width value to work out the wrap parameter. We'll wrap at the end of the textbounds width. We calculate the textbounds width by subtracting the padding from the absolute width of the box. The padding is multiplied by two because we're subtracting the padding

on the left of the box *and* the right. After that calculation is done the wrap variable is equal to the width of the textbounds and it's passed on to the Textbox constructor.

We can test the wrapping out with a little example text. Copy the code from Listing 2.132.

```
--code omitted
local text = "'Twas brillig, and the slithy toves"
local textbox = CreateFixed(gRenderer, 100, -100, 250, 64, text)

function update()
    textbox:Update(GetDeltaTime())
    textbox:Render(gRenderer)
end
```

Listing 2.132: Testing the wrapping. In main.lua.

Example gui-7-solution contains the code so far. Run the program to see the wrapping in action.



Figure 2.57: Example of the textbox using wrapped text.

Figure 2.57 shows the wrapping working. Try changing the width of the box and see how the text attempts to fit it. Text can still overflow out of bottom of the textbox but we'll be fixing this too, later on.

Adding Children

Textboxes are often used to show dialog. When showing dialog it's common to display the name and a portrait of the speaker in the textbox. An example dialog box is shown in Figure 2.58. The portrait picture and the title are like add-ons to the textbox. Depending on where these elements are placed they alter the dimensions of the textbounds.

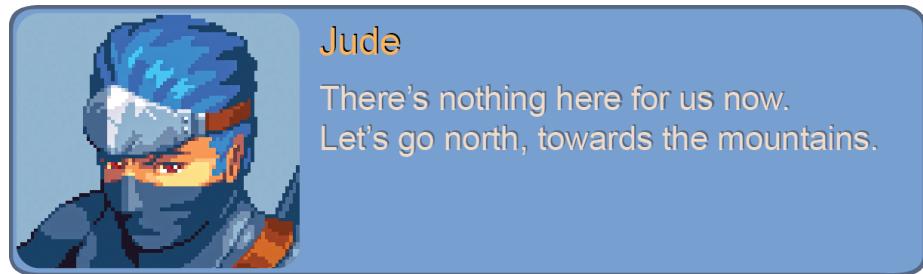


Figure 2.58: Example textbox with portrait and title.

RPGs have a lot of variation in exactly what makes up a dialog box. Some RPGs include a portrait, some don't, some align the portrait to the left inside the box, some have the portrait flush to the top of the box... nearly every game does something a little bit different. Therefore we'll keep the code somewhat generic but code for the specific case where the portrait appears to the left of the text and a title appears at the top.

Let's modify the `Textbox` class to take in additional sprites and text. These elements are added as *children* of the textbox. Each *child* has its own offset from the top left of the textbox. Example `gui-8` contains the code so far as well as an example portrait picture, "avatar.png". If you're going to extend your own codebase, copy the portrait sprite and add it to your manifest.

The textbox children are either sprites or pieces of text and they're represented using simple tables as shown in Listing 2.133.

```
{ type = "text", text = "NPC:", x = 0, y = 0 }
{ type = "sprite", sprite = SomeSprite, x = 0, y = 0 }
```

Listing 2.133: The two children we're going to add to the textbox.

The child tables contain an offset stored as x, y and a type field to identify if they're text or a sprite. The data is stored in the table under either text or sprite depending on the type.

The children give our UI a little flexibility but not so much that we end up creating a fully generic GUI system! Let's modify the Textbox class to store and render children. Copy the code from Listing 2.134.

```
function Textbox:Create(params)

    params = params or {}

    local this =
    {
        -- code omitted
        mWrap = params.wrap or -1,
        mChildren = params.children or {},
    }

    -- code omitted

end

-- code omitted

function Textbox:Render(renderer)

    -- code omitted

    for k, v in ipairs(self.mChildren) do
        if v.type == "text" then
            renderer:DrawText2d(
                textLeft + (v.x * scale),
                textTop + (v.y * scale),
                v.text,
                Vector.Create(1,1,1,1)
            )
        elseif v.type == "sprite" then
            v.sprite:SetPosition(
                left + (v.x * scale),
                top + (v.y * scale))
            v.sprite:SetScale(scale, scale)
            renderer:DrawSprite(v.sprite)
        end
    end
end
```

Listing 2.134: Changing up the constructor to add support for children. In Textbox.lua.

In Listing 2.134 we modified the constructor to check for an optional children table. The `this.mChildren` variable is set to the children field of the params table, or if it doesn't exist it's set to the empty table.

The children are all drawn at the end of the Render function which means they're drawn last and on top of everything else. The child drawing code is a for-loop that iterates through the children and checks their type. If the child is a text type then it's rendered from the top and left of the textbounds area using its offsets. Child text is scaled by the scale factor of the box. If the child is a sprite type then the sprite is positioned from the top left of the entire textbox. The sprite is scaled according to the scale factor and rendered. These code changes let us add sprites or text decoration to textboxes and have them automatically rendered them out.

To see the new textbox and children code in action, we need to add some children. Modify the `CreateFixed` function as shown in Listing 2.135.

```
function CreateFixed(renderer, x, y, width, height, text, params)

    params = params or {}
    local avatar = params.avatar
    local title = params.title

    local padding = 10
    local textScale = 1.5
    local panelTileSize = 3

    local wrap = width - padding
    local boundsTop = padding
    local boundsLeft = padding

    local children = {}

    if avatar then
        boundsLeft = avatar:GetWidth() + padding * 2
        wrap = width - (boundsLeft) - padding
        local sprite = Sprite.Create()
        sprite:SetTexture(avatar)
        table.insert(children,
        {
            type = "sprite",
            sprite = sprite,
            x = avatar:GetWidth() / 2 + padding,
            y = -avatar:GetHeight() / 2
        })
    end
end
```

```

    end

    if title then
        -- adjust the top
        local size = renderer:MeasureText(title, wrap)
        boundsTop = size:Y() + padding * 2

        table.insert(children,
        {
            type = "text",
            text = title,
            x = 0,
            y = size:Y() + padding
        })
    end

    return Textbox:Create
{
    text = text,
    textScale = textScale,
    size =
    {
        left     = x - width / 2,
        right    = x + width / 2,
        top      = y + height / 2,
        bottom   = y - height / 2
    },
    textbounds =
    {
        left = boundsLeft,
        right = -padding,
        top = -boundsTop,
        bottom = padding
    },
    panelArgs =
    {
        texture = Texture.Find("gradient_panel.png"),
        size = panelTileSize,
    },
    children = children,
    wrap = wrap
}
end

```

Listing 2.135: Extending create fitted to support children. In main.lua.

In Listing 2.135 we've added an extra parameter to the constructor, a params table. The params table is optional and it's used to add the title and portrait. The first few lines of the constructor assign the avatar and title entries from the params table to local variables. If no avatar or title is passed in then the local avatar and title variables are set to nil.

We draw the avatar on the left of the textbox and move the textbounds so they don't overlap it. Adding a portrait forces the textbounds to shrink a little and move over to the right. If the title exists we draw it at the top of the textbox and once again alter the textbounds to prevent any overlap. Near the start of the function we introduce three new variables: children, boundsLeft, and boundsTop. The children variable is a list of sprites and text that we're adding to the textbox. The boundsLeft and boundsTop variables define the top left point of the textbounds. How the top left position of the textbounds changes is shown in Figure 2.59. The default values of boundsLeft and boundsTop are the top left of textbox with the padding added.



Figure 2.59: How the portrait and title change the top left textbounds location.

If there's a portrait we measure the texture to get its width, and reduce the textbound's width by that amount plus a little padding. Altering the textbound's width means we must recalculate the wrapping parameter, so we do that next. Finally we add the sprite as a child, aligning it to the left edge and top of the textbox, with a little padding, so that it's not right on the border.

If there's a title then we measure its height in pixels and reduce the textbound's height by that amount, plus a little padding. Then, much like the avatar, we add it as a child of the textbox. The child table we're adding is of the type "text". The text is the title text that's been passed in, the X is 0, the very left edge of the textbounds area and the Y is size:Y() + padding which moves the title by its height upwards with a little padding. This positions the title to the left and above the textbounds areas.

After the avatar and title text have been added, we come to the Textbox constructor which is nearly unchanged. There's an extra argument for the children table. The textbounds top and left fields use the boundsTop and boundsLeft values. Let's test the code. Copy the code from Listing 2.136 into your main.lua file.

```
local text = "It's dangerous to go alone! Take this."
local textbox = CreateFixed(gRenderer, 100, -100, 320, 100, text,
{
    title = "Charles:",
    avatar = Texture.Find("avatar.png")
})

function update()
    textbox:Update(GetDeltaTime())
    textbox:Render(gRenderer)
end
```

Listing 2.136: Creating and using a dialog box with portrait and title. In main.lua.

Example gui-8-solution contains the completed project. Run the code and you'll see something like Figure 2.60. Try removing the title argument or the avatar to see how the layout changes.



Figure 2.60: A textbox with a portrait and title.

In this section we've created a standard dialog textbox with a portrait to the left and a title above the text, but in your own game feel free to mix it up! There are a lot of different textboxes out there with many different transitions. The best textboxes help reinforce and reflect the game themes and story.

Chunking Text

When text is too long it spills out the bottom of the textbox; it's time to fix this! We'll write code that breaks overly long text into textbox-sized chunks. To do this we'll modify the textbox code again. We'll add a continue caret that's shown when the textbox is displaying the first in series of chunks. The continue caret is a bouncing arrow near the bottom of the textbox of the type shown in Figure 2.61.

At this point, it's probably best to switch the project to example gui-9. This example drops the avatar and title from before so we can just concentrate on the chunking. (If you're feeling brave, by all means continue with your current codebase! It's probably best to refer to the gui-9 main.lua file to see how it's setup). Example gui-9 has

text that's too long for the textbox and a continue caret graphic, "continue_caret.png", already added to the manifest.

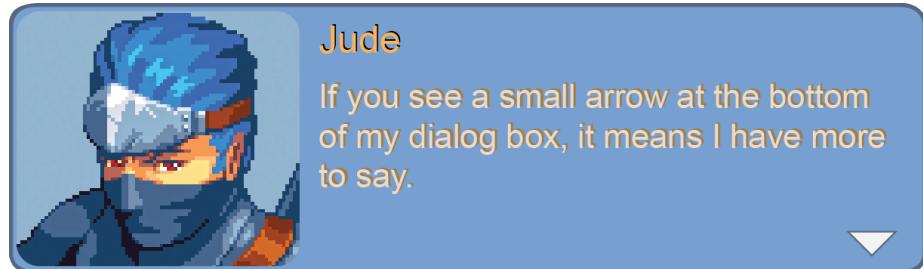


Figure 2.61: A dialog box with a continue caret.

In example gui-9 the text variable is set to a quote from Cicero as written below.

A nation can survive its fools, and even the ambitious. But it cannot survive treason from within. An enemy at the gates is less formidable, for he is known and carries his banner openly. But the traitor moves amongst those within the gate freely, his sly whispers rustling through all the alleys, heard in the very halls of government itself. For the traitor appears not a traitor; he speaks in accents familiar to his victims, and he wears their face and their arguments, he appeals to the baseness that lies deep in the hearts of all men. He rots the soul of a nation, he works secretly and unknown in the night to undermine the pillars of the city, he infects the body politic so that it can no longer resist. A murderer is less to fear. The traitor is the plague.

The textbox in gui-9 is aligned to the bottom of the screen and the width is set to nearly equal the screen width. Run the code. You'll see that the text fills the entire textbox and runs off the bottom as shown in Figure 2.62.

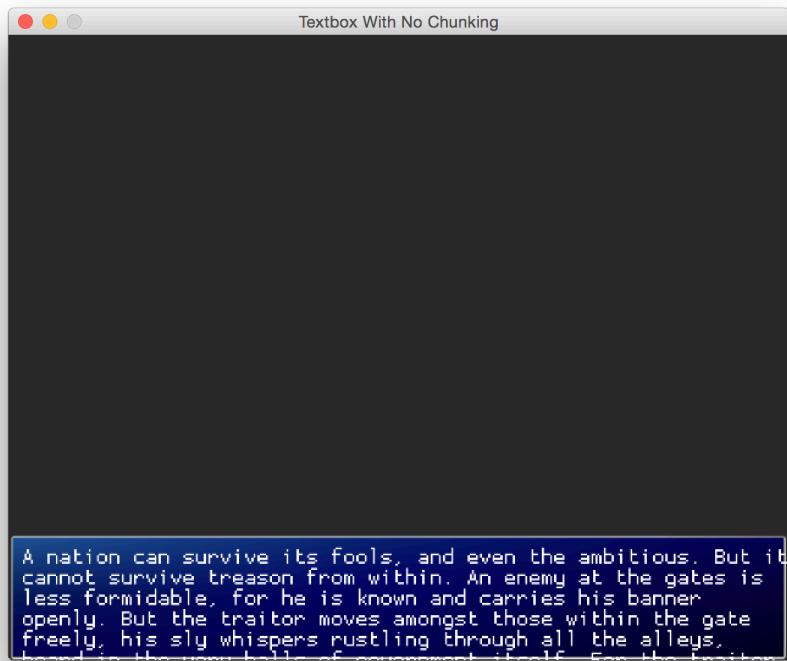


Figure 2.62: Large amounts of text will eventually spill out of the bottom of a textbox.

We need to measure the text and split it into chunks. Then we'll pass the list of the chunks to the textbox and let it handle displaying them. Initially the textbox displays the first chunk. When its OnClick function is called it advances to the next chunk and when no more chunks are left it closes the textbox.

Chunking Algorithm

To break the text into chunks we use a Renderer function called NextLine. Given some text and a wrap width NextLine returns the number of characters of text before it's needs to wrap. Figure 2.63 helps show how the NextLine function works.

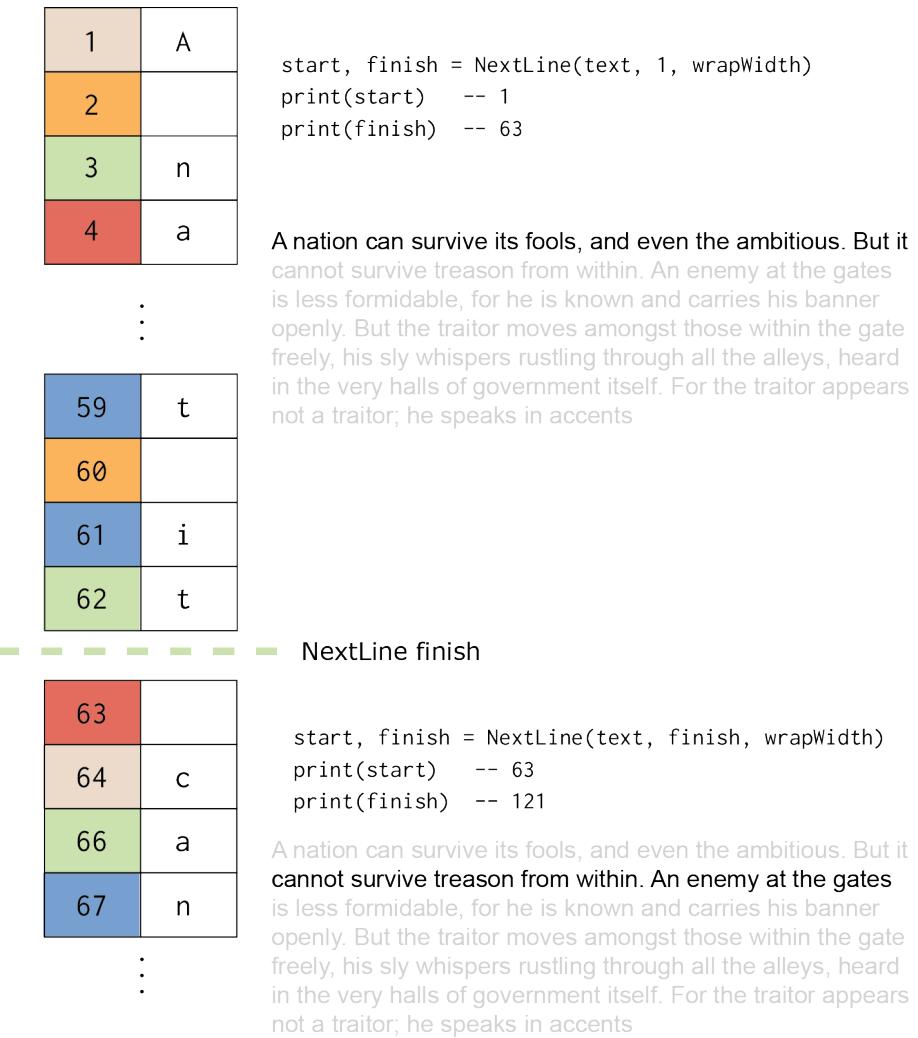


Figure 2.63: How `NextLine` is used to break text into chunks.

Copy the code from Listing 2.137 to add chunking code to the `CreateFixed` function.

```

-- code omitted

renderer:ScaleText(textScale)
--
-- Section text into box-sized chunks.
--

local faceHeight = math.ceil(renderer:MeasureText(text):Y())

```

```

local start, finish = gRenderer:NextLine(text, 1, wrap)

local boundsHeight = height - (boundsTop + boundsBottom)
local currentHeight = faceHeight

local chunks = {{string.sub(text, start, finish)}}
while finish < #text do
    start, finish = gRenderer:NextLine(text, finish, wrap)

    -- If we're going to overflow
    if (currentHeight + faceHeight) > boundsHeight then
        -- make a new entry
        currentHeight = 0
        table.insert(chunks, {string.sub(text, start, finish)})
    else
        table.insert(chunks[#chunks], string.sub(text, start, finish))
    end
    currentHeight = currentHeight + faceHeight
end

-- Make each textbox be represented by one string.
for k, v in ipairs(chunks) do
    chunks[k] = table.concat(v)
end

return Textbox>Create
{

-- code omitted

```

Listing 2.137: Breaking the text into a table of chunks in CreateFixed function. In main.lua.

In Listing 2.137, we begin by storing some variables we need for the chunking. The first variable we need is faceHeight. To get the faceHeight we call MeasureText but don't pass in a wrap amount; this ensures the height value is for a single line of text. We use faceHeight to work out how many lines can fit in the textbounds area.

Next we make use of NextLine. It takes in a string of text, an index into the text, and a wrap width. It returns two positions in the text, a start and finish, denoting the length of text that can be rendered before we need to start a new line.

The start and finish variables returned by NextLine mark out the very first line of text to be rendered. We create a variable boundsHeight that represents the height of the textbounds rectangle. We use boundsHeight to find out how many lines of text we can

fit in the textbounds. The textbounds height is calculated by subtracting the padding from the height of the textbox.

The variable `currentHeight` is set to `faceHeight` and we use this to track the height of the current chunk and make sure we're not writing text off the bottom of the box.

The final variable, `chunks`, is a table that holds all the chunks. Each entry in the `chunks` table is a textbox chunk. You can see text broken into chunks in Figure 2.64. Each chunk entry is a table of strings, each a line of text.

To get the first line of text we use the `substring` command and the start and finish values returned from `NextLine`. When creating the chunk table we've already called `NextLine` and so we add this first line as the first entry.

A nation can survive its fools, and even the ambitious. But it cannot survive treason from within. An enemy at the gates is less formidable, for he is known and carries his banner openly. But the traitor moves amongst those within the gate freely, his sly whispers rustling through all the alleys, heard in the very halls of government itself. For the traitor appears not a traitor; he speaks in accents familiar to his victims, and he wears their face and their arguments, he appeals to the baseness that lies deep in the hearts of all men. He rots the soul of a nation, he works secretly and unknown in the night to undermine the pillars of the city, he infects the body politic so that it can no longer resist. A murderer is less to fear. The traitor is the plague.

A nation can survive its fools, and even the ambitious. But it cannot survive treason from within. An enemy at the gates is less formidable, for he is known and carries his banner openly. But the traitor moves amongst those within the gate freely, his sly whispers rustling through all the alleys, heard

in the very halls of government itself. For the traitor appears not a traitor; he speaks in accents familiar to his victims, and he wears their face and their arguments, he appeals to the baseness that lies deep in the hearts of all men. He rots the soul of a nation, he works secretly and unknown in the night

to undermine the pillars of the city, he infects the body politic so that it can no longer resist. A murderer is less to fear. The traitor is the plague.

Figure 2.64: How large amounts of text are broken into textbox-sized chunks.

The code that actually does the chunking is the while loop. It breaks the text into lines and fills in the `chunks` table. The while loop continues looping until we reach the end

of the text. The finish variable represents the end of the current line. If the finish position is bigger than the size of the string, it means we've processed all the text.

The chunking code uses NextLine to grab the next line of text. If adding the line to the current chunk means it's too big for the textbounds, then we start a new chunk, reset the currentHeight to zero, and add the new line. If we can add the line and it still fits in the textbounds, we go ahead and add the line to the current chunk. After checking the line we increase the currentHeight by the faceHeight.

After the loop has chunked the text, we have a table of chunks and each chunk is a table of lines. The chunks table might look something like Listing 2.138.

```
chunks =
{
    {
        "A nation can survive its fools, and even the ambitious. But it",
        " cannot survive treason from within. An enemy at the gates",
        "is less formidable, for he is known and carries his banner",
        "openly. But the traitor moves amongst those within the gate",
        "freely, his sly whispers rustling through all the alleys, heard"
    },
    {
        "in the very halls of government itself. For the traitor appears",
        "not a traitor; he speaks in accents familiar to his victims, and",
        " he wears their face and their arguments, he appeals to the",
        "baseness that lies deep in the hearts of all men. He rots the",
        "soul of a nation, he works secretly and unknown in the night"
    },
    {
        "to undermine the pillars of the city, he infects the body politic",
        "so that it can no longer resist. A murderer is less to fear. The",
        "traitor is the plague."
    }
}
```

Listing 2.138: An example of how some chunked text might appear.

Each entry in Listing 2.138 is a list of strings. To merge the strings together we use a final loop that calls table.concat on each table of strings. The Lua function table.concat efficiently transforms a table of strings into a single string by adding them all together, a process which is also known as *concatenation*.

Modifying the Textbox to Work With Chunks

Let's begin by adding the continue arrow to the textbox to signify there's more text to read than is currently being shown.

The chunking code has left us with a table of strings called chunks. Each chunk is piece of text that fits the textbox perfectly. We're going to pass the chunk table straight into the Textbox constructor and let it deal with how to display the text. Copy the Textbox changes changes from Listing 2.139.

```
-- code omitted

return Textbox:Create
{
    text = chunks,
    textScale = textScale,
-- code omitted
```

Listing 2.139: Passing chunks of text to the Textbox constructor in the CreateFixed function. In main.lua.

Instead of the textbox receiving a single string of text, it now receives a table of strings, so we need to add support for that. Alter the code as shown in Listing 2.140. This code lets us pass in either a single string or a table of strings.

```
function Textbox:Create(params)

    params = params or {}

    if type(params.text) == "string" then
        params.text = {params.text}
    end

    local this =
    {
        -- mText = params.text, Removing this
        mChunks = params.text,
        mChunkIndex = 1,
        mContinueMark = Sprite.Create(),
        mTime = 0,
        -- code omitted
    }
    this.mContinueMark:SetTexture(Texture.Find("continue_caret.png"))

-- code omitted
```

Listing 2.140: Adding chunk support to the Textbox. In Textbox.lua.

In Listing 2.140 we use an if-statement to check if the text type is a string. If so, we turn it into a table with a single entry of that string. This way we can treat a single string or table chunks in the same way.

In the constructor we've replaced the old `mText` variable with `mChunks`. We've also added a new variable, `chunkIndex`. The `chunkIndex` tells us which chunk is currently being displayed in the textbox.

To display and animate the continue arrow we've added `mContinueMark` and `mTime`. The `mContinueMark` field is the sprite representing the arrow and the `mTime` is a counter that helps animate the arrow.

Copy the code from Listing 2.141.

```
function Textbox:Update(dt)
    self.mTime = self.mTime + dt
    self.mAppearTween:Update(dt)
end

function Textbox:Render(renderer)

    -- code omitted

    local textTop = top + (self.mBounds.top * scale)
    local bottom = self.mY - (self.mHeight/2 * scale)

    renderer:DrawText2d(
        textLeft,
        textTop,
        self.mChunks[self.mChunkIndex],
        Vector.Create(1,1,1,1),
        self.mWrap * scale)

    if self.mChunkIndex < #self.mChunks then
        -- There are more chunks to come.
        local offset = 12 + math.floor(math.sin(self.mTime*10)) * scale
        self.mContinueMark:SetScale(scale, scale)
        self.mContinueMark:SetPosition(self.mX, bottom + offset)
        renderer:DrawSprite(self.mContinueMark)
    end

    for k, v in ipairs(self.mChildren) do
        -- code omitted
    end
end
```

Listing 2.141: Adding support to the Textbox for chunks. In Textbox.lua.

We've updated the Render function to work with our new chunk data. The DrawText2d call now takes in self.mChunks[self.mChunkIndex]. We draw the continue caret if the chunkIndex is less than the number of chunks. This tells the user there's more text to be displayed.

The Render function draws the continue arrow and handles its animation. To make the continue caret bounce up and down we use the math.sin function. The math.sin function takes in mTime to move the arrow up and down according to a sine wave. We multiply mTime by 10 to speed up the animation. The math.sin function returns a number between -1 and 1. The continue arrow is going to be centered at the 12 pixels from the bottom. Our sin animation only moves the offset a few pixels but this gives a nice bounce. We scale the continue caret, like everything else, then set the position and render it.

The user needs a way to advance to the next chunk of text when viewing the textbox. Listing 2.142 shows code to advance to the next chunk using the OnClick function.

```
function Textbox:OnClick()

    if self.mChunkIndex >= #self.mChunks then
        --
        -- If the dialog is appearing or disappearing
        -- ignore interaction
        --
        if not (self.mAppearTween:IsFinished()
            and self.mAppearTween:Value() == 1) then
            return
        end
        self.mAppearTween = Tween:Create(1, 0, 0.2, Tween.EaseInCirc)
    else
        self.mChunkIndex = self.mChunkIndex + 1
    end
end
```

Listing 2.142: Moving from chunk to chunk in OnClick. In Textbox.lua.

If the chunkIndex is greater than or equal to the number of chunks then OnClick works as before; it tries to dismiss the textbox. If the chunkIndex is less than the number of chunks, then we increment the chunkIndex. When the Render function is next called it renders the text using the incremented index and so renders the next chunk of the text.

That's the end of the modifications for textbox. Copy the code from Listing 2.143 into your main.lua and let's try it out.

```

local width = System.ScreenWidth() - 4
local height = 102 -- a nice height
local x = 0
local y = -System.ScreenHeight()/2 + height / 2 -- bottom of the screen

local text = [[A nation can survive its fools, and even the ambitious.
But it cannot survive treason from within. An enemy at the gates is less
formidable, for he is known and carries his banner openly. But the
traitor moves amongst those within the gate freely, his sly whispers
rustling through all the alleys, heard in the very halls of government
itself. For the traitor appears not a traitor; he speaks in accents
familiar to his victims, and he wears their face and their arguments, he
appeals to the baseness that lies deep in the hearts of all men. He
rots the soul of a nation, he works secretly and unknown in the night
to undermine the pillars of the city, he infects the body politic so
that it can no longer resist. A murderer is less to fear."]]

local title = "NPC:"
local avatar = Texture.Find("avatar.png")
local textbox = CreateFixed(gRenderer, x, y,
                           width, height, text,
                           title, avatar)

function update()
    if not textbox:IsDead() then
        textbox:Update(GetDeltaTime())
        textbox:Render(gRenderer)
    end

    if Keyboard.JustPressed(KEY_SPACE) then
        textbox:OnClick()
    end
end

```

Listing 2.143: Update loop for handling text that's longer than one box. In main.lua.

Example gui-9-solution contains the code so far. Run the program and you'll see a fixed textbox appear at the bottom of the screen as shown in Figure 2.65, full of text with a little bobbing arrow at the bottom of the box.

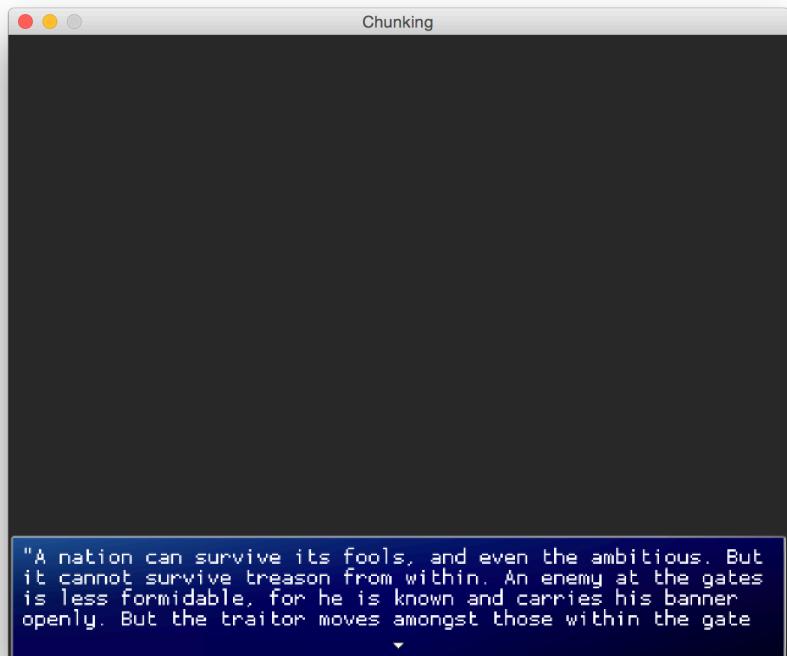


Figure 2.65: A fitted textbox using chunks.

Press space and the next chunk of text will be shown. Keep pressing space until there are no more chunks to display, and then the textbox will close.

Choices

Need to ask the player if they want to save? Or ask them to save a rabbit from a rabbit-crushing machine? You need a textbox that lets the player choose from some options. That's what we're going to add next.

In order to add this type of selection menu, we need a cursor graphic to indicate which choice is selected. A cursor usually appears to the left of the selected item and points to the right. The Final Fantasy games use a gloved hand for a cursor but we'll be using a simple triangle.

Example gui-10 is a new project containing the code so far and a cursor image called "cursor.png" which has already been added to the manifest. There's also an empty file called "Selection.lua", where we'll create the new selection menu class.

Once we create the selection class, we'll integrate it into the Textbox class and finish by displaying an example difficult choice for our hero to face. Figure 2.66 shows a simple selection menu. There are two choices "Yes" and "No" in a single column. This is the most basic use case.

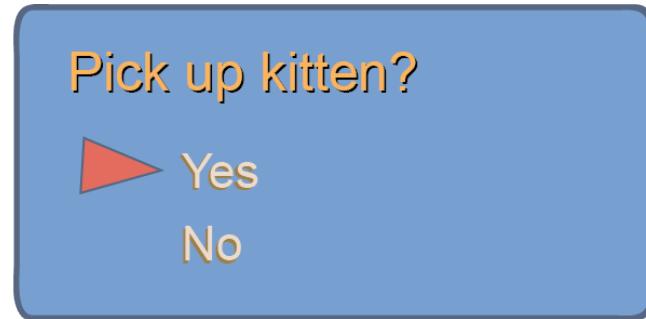


Figure 2.66: A simple list of two items.

The selection code isn't just used for textboxes. Browsing the inventory and picking out an item can all be handled by our Selection class. A more complicated list is shown in Figure 2.67. To avoid code repetition, the Selection class supports lists made from multiple columns such as the one shown in Figure 2.67. Lists such as these usually have many items, more than can be displayed on screen at once. If the list is very long it displays only a portion of the list and you can scroll the items.

Choose spell.		
Fire	► Fire II	Ice Cut
Blades	--	--
--	--	--
--	--	--

Figure 2.67: A multi-column list displaying spells.

Different lists display their items in different ways. Inventory items are drawn with a small icon, shop items are drawn with a price, etc. Therefore the Selection class item

render code can be overridden. When a Selection menu is created, a custom Render function can be passed in to override the default item render code. This makes the Selection class a real workhorse we can use for a lot of our game's menus.

Copy Listing 2.144 into your Selection.lua file.

```
Selection = {}
Selection.__index = Selection
function Selection:Create(params)

    local this =
    {
        mX = 0,
        mY = 0,
        mDataSource = params.data,
        mColumns = params.columns or 1,
        mFocusX = 1,
        mFocusY = 1,
        mSpacingY = params.spacingY or 24,
        mSpacingX = params.spacingX or 128,
        mCursor = Sprite.Create(),
        mShowCursor = true,
        mMaxRows = params.rows or #params.data,
        mDisplayStart = 1,
        mScale = 1,
        OnSelection = params.OnSelection or function() end
    }

    this.mDisplayRows = params.displayRows or this.mMaxRows

    local cursorTex = Texture.Find(params.cursor or "cursor.png")
    this.mCursor:SetTexture(cursorTex)
    this.mCursorWidth = cursorTex:GetWidth()

    setmetatable(this, self)

    this.RenderItem = params.RenderItem or this.RenderItem
    this.mWidth = this:CalcWidth(gRenderer)
    this.mHeight = this:CalcHeight()

    return this
end
```

Listing 2.144: Selection class constructor. In Selection.lua.

The Selection constructor takes in a params table that describes how the menu is constructed. Most parameters are optional but generally it requires a table of items

called the datasource to populate the menu and an OnSelection function which is called when a menu item is selected. This is a pretty big class so let's begin by covering what each class field does.

mX, mY Define the menu position from the top left corner.

mDataSource The list of items to be displayed. It can't be empty or nil. In most cases **mDataSources** is a table of strings but it doesn't have to be.

mColumns The number of columns the menu has. This defaults to 1. When displaying large amounts of items, like an inventory, more columns means more items can be shown on screen.

mFocusX, mFocusY Indicates which item in the list is currently selected. **mFocusX** tells us which column is selected and **mFocusY** which element in that column. These are both set to 1 which means the first element in the first column starts selected.

mSpacingY The vertical space in pixels for each row of the selection list. If we want more spacing between elements, this value can be increased. If we want less space, it can be decreased.

mSpacingX The horizontal space between columns. If the **mColumns** field is set to 1 this field doesn't do anything.

mCursor A sprite which represents the cursor and is used to point to the currently focused item.

mShowCursor There are certain times when the cursor shouldn't be visible, for instance when there are multiple selection menus on screen and the user is only interacting with one of them. For that reason the **mShowCursor** flag is used to determine if the cursor should be rendered or not.

mMaxRows Represents how many rows the selection menu can hold. In most cases this is equal to the number of items. For cases like an inventory the max number of rows might be 30 but only 5 are displayed at once.

mDisplayRows The number of rows to display at once.

mDisplayStart Used when we can only show a portion of the items that are in the list. The **mDisplayStart** tells us which row is the first row that's rendered out. Increasing and decreasing this variable lets us scroll through the list. See Figure 2.68.



Figure 2.68: How `mDisplayStart` and `mDisplayRows` are used to display a scrollable list.

`mScale` : Scales the menu.

`OnSelection` : Called when an item in the listbox is selected. If we give the user a choice of "Yes" or "No" this is the function that's called when one of the options is selected.

After the this table is created, the `mDisplayRows` is set, to determine how many rows are shown on screen at once. For most cases this will equal the maximum number of rows. For cases where there are a lot of items it will be significantly less. The cursor sprite is assigned a texture, and the width of the cursor is stored in `mCursorWidth`. We store the cursor's width so later we can align it to the right of a selected item.

The `RenderItem` function is optionally overridden with a function from the `params` table and finally the width and height of the selection menu are assigned.

That's a lot to take in at once, but as we write the rest of the code we'll see all these fields in action.

Let's implement the `Render` function. Copy the code from Listing 2.145.

```

function Selection:Render(renderer)

    local displayStart = self.mDisplayStart
    local displayEnd = displayStart + self.mDisplayRows - 1

    local x = self.mX
    local y = self.mY

    local cursorWidth = self.mCursorWidth * self.mScale
    local cursorHalfWidth = cursorWidth / 2

```

```

local spacingX = (self.mSpacingX * self.mScale)
local rowHeight = (self.mSpacingY * self.mScale)

self.mCursor:SetScale(self.mScale, self.mScale)

local itemIndex = ((displayStart - 1) * self.mColumns) + 1
for i = displayStart, displayEnd do

    for j = 1, self.mColumns do

        if i == self.mFocusY and j == self.mFocusX
            and self.mShowCursor then

            self.mCursor:SetPosition(x + cursorHalfWidth, y)
            renderer:DrawSprite(self.mCursor)
        end

        local item = self.mDataSource[itemIndex]
        self:RenderItem(renderer, x + cursorWidth, y, item)

        x = x + spacingX
        itemIndex = itemIndex + 1
    end

    y = y - rowHeight
    x = self.mX
end
end

```

Listing 2.145: Selector class Render function. In Selection.lua.

The first lines of the render function determine which rows of items to display. When `mDisplayRows` is smaller than `mMaxRows` we only display a subset of items. The `displayStart` variable determines the first row and `displayEnd` determines the last row to render. The `displayEnd` variable is calculated by adding the `mDisplayRows` value to `displayStart`. (We subtract one from the `mDisplayRows` to prevent drawing one more item than we should. If you look carefully at Figure 2.68 you'll be able to see why.)

The `x` and `y` variables are used to keep track of where the current item is being rendered. All variables involved with the selection menu's size are scaled by the `mScale` variable.

To draw each row of items we use a nested pair of loops. The outer loop draws the rows and the inner loop draws the columns. Before entering the loop, we calculate the `itemIndex`. This is an index into the `mDataSource` array. We use `itemIndex` to get the first element we want to draw. Normally `itemIndex` is 1, but if we've scrolled down the

list it changes. To calculate the index we multiply `mDisplayStart`, which tells us how many rows we've scrolled down, by `mColumn`, the number of the elements in a row.

We loop through the rows and columns, drawing the items to the screen. There's a test in the inner loop to check if the current item being drawn is in focus. If an item is in focus and the menu is set to show the cursor, then we draw the cursor so it points at that item. Items are taken from the `mDataSource` table and the drawing is handled by the `RenderItem` function. At the end of the inner loop the `x` position is increased to move us to the next column. The `itemIndex` is also increased to index the next item in the `mDataSource` table. The first element is drawn in the top left, the next element to the right of it on the same row, and so on. You can see how the layout changes by looking at Figure 2.69 and comparing it with Figure 2.68.

At the end of the outer for-loop the `x` and `y` variables are prepared for drawing the next row of elements. The `x` position is reset to the start of the row and the `y` is decreased by the `rowHeight`.

```

mDataSource =
{
    "Knife",    -- 1
    "Dart",     -- 2
    "Grenade",   -- 3
    "Shuriken",  -- 4
    "Axe",       -- 5
    "Card",      -- 6
    "Bola",      -- 7
    "Spear",     -- 8
},
mDisplayRows = 2,
mColumns = 2,
mDisplayStart = 3

```

Figure 2.69: Changing the Selection params to get a two-column layout.

Let's add the `RenderItem` function next, as it's responsible for drawing each item in the selection menu. Copy the default `RenderItem` implementation from Listing 2.146.

```

function Selection:RenderItem(renderer, x, y, item)
    if not item then
        renderer:DrawText2d(x, y, "--")
    else
        renderer:DrawText2d(x, y, item)
    end
end

```

Listing 2.146: The default `RenderItem` function. In `Selection.lua`.

The `RenderItem` function is pretty short. It draws the item at the given `x` and `y`. If the item doesn't exist, it instead draws “-” to indicate an empty slot.

That's the all rendering code for the Selection menu. Let's deal with input next. Copy the code from Listing 2.147 to implement menu navigation.

```
function Selection:HandleInput()
    if Keyboard.JustPressed(KEY_UP) then
        self:MoveUp()
    elseif Keyboard.JustPressed(KEY_DOWN) then
        self:MoveDown()
    elseif Keyboard.JustPressed(KEY_LEFT) then
        self:MoveLeft()
    elseif Keyboard.JustPressed(KEY_RIGHT) then
        self:MoveRight()
    elseif Keyboard.JustPressed(KEY_SPACE) then
        self:OnClick()
    end
end

function Selection:MoveUp()
    self.mFocusY = math.max(self.mFocusY - 1, 1)
    if self.mFocusY < self.mDisplayStart then
        self:MoveDisplayUp()
    end
end

function Selection:MoveDown()
    self.mFocusY = math.min(self.mFocusY + 1, self.mMaxRows)
    if self.mFocusY >= self.mDisplayStart + self.mDisplayRows then
        self:MoveDisplayDown()
    end
end

function Selection:MoveLeft()
    self.mFocusX = math.max(self.mFocusX - 1, 1)
end

function Selection:MoveRight()
    self.mFocusX = math.min(self.mFocusX + 1, self.mColumns)
end

function Selection:OnClick()
    local index = self:GetIndex()
    self.OnSelection(index, self.mDataSource[index])
end
```

```

function Selection:MoveDisplayUp()
    self.mDisplayStart = self.mDisplayStart - 1
end

function Selection:MoveDisplayDown()
    self.mDisplayStart = self.mDisplayStart + 1
end

function Selection:GetIndex()
    return self.mFocusX + ((self.mFocusY - 1) * self.mColumns)
end

```

Listing 2.147: Functions to handle input and scrolling the list.

The list menu responds to five types of input: *up*, *down*, *left*, *right*, and *select*. When *up* is pressed the focus moves up a row by changing the *self.mFocusY* variable. The *math.max* function makes sure the *mFocusY* cannot be less than 1. After *self.mFocusY* is updated it's compared with *self.mDisplayStart*. The list needs scrolling up if *mFocusY* is greater than *displayStart*. This is done by calling *MoveDisplayUp*. Pressing *down* runs similar code but in the opposite direction.

Pressing *left* and *right* moves the *mFocusX* back and forth along the columns. Pressing *select* calls *OnClick* which calls the *OnSelection* function with the selected index and the currently focused item.

The *MoveDisplayUp* and *MoveDisplayDown* functions decrement and increment the *mDisplayStart* variable scrolling the selection menu up and down. The *GetIndex* function uses *mFocusX* and *mFocusY* to calculate an index into the *mDataSource* array.

The remaining code for the Selection menu consists mainly of support and helper functions that make it easier to use. Copy the code from Listing 2.148.

```

function Selection:GetWidth()
    return self.mWidth * self.mScale
end

function Selection:GetHeight()
    return self.mHeight * self.mScale
end

-- If the RenderItem function is overwritten
-- This won't give the correct result.
function Selection:CalcWidth(renderer)
    if self.mColumns == 1 then
        local maxEntryWidth = 0
        for k, v in ipairs(self.mDataSource) do

```

```

        local width = renderer:MeasureText(tostring(v)):X()
        maxEntryWidth = math.max(width, maxEntryWidth)
    end
    return maxEntryWidth + self.mCursorWidth
else
    return self.mSpacingX * self.mColumns
end
end

function Selection:CalcHeight()
    local height = self.mDisplayRows * self.mSpacingY
    return height - self.mSpacingY / 2
end

function Selection>ShowCursor()
    self.mShowCursor = true
end

function Selection:HideCursor()
    self.mShowCursor = false
end

function Selection:SetPosition(x, y)
    self.mX = x
    self.mY = y
end

function Selection:PercentageShown()
    return self.mDisplayRows / self.mMaxRows
end

function Selection:PercentageScrolled()
    local onePercent = 1 / self.mMaxRows
    local currentPercent = self.mFocusY / self.mMaxRows

    -- Allows a 0 value to be returned.
    if currentPercent <= onePercent then
        currentPercent = 0
    end
    return currentPercent
end

function Selection:SelectedItem()
    return self.mDataSource[self:GetIndex()]
end

```

Listing 2.148: Helper functions for Selection class. In Selection.lua.

To correctly position the selection menu we need to know its width and height. We use the CalcWidth and CalcHeight functions defined in Listing 2.148 in the constructor to work out the width and height. The CalcWidth function works in one of two ways. If there are multiple columns then the width is calculated by multiplying the number of columns by mSpacingX which represents the column width. If there's a single column we iterate through each row and find the longest, then add on spacing for the size of the cursor. The CalcHeight function multiplies the number of rows by the face height plus spacing. Half a mSpacingY is subtracted from the end result because spacing after the last element doesn't count and we use the other half for padding.

The GetHeight and GetWidth functions are simple helper functions to return the scaled width and height of the menu.

The ShowCursor and HideCursor functions toggle the cursor's visibility. SetPosition positions the selection menu on the screen.

PercentageShown returns a number between 0 and 1 indicating how far the user has scrolled through the list. We use PercentShown for displaying scrollbars or continue carets. If the percent scrolled is equal to or below 1%, it's reduced to 0; otherwise 0 is never returned. This is done to make scrollbars work nicely.

The final helper function is SelectedItem which returns the currently focused item.

That was a lot of code! But now we're ready to see a selection menu in action. Copy the code from Listing 2.149 into your main.lua file. If you're using your own codebase, remember to include the Selection.lua class in the manifest and to call Asset.Run on it in the main.lua.

```
gRenderer:ScaleText(1.5, 1.5)
gRenderer:AlignText("left", "center")
gLastSelection = "?"
gChoice = Selection>Create
{
    data =
    {
        "Yes",
        "No"
    },
    cursor = "cursor.png",
    OnSelection = function(selectIndex)
        gLastSelection = selectIndex
    end
}

function update()
    gRenderer:DrawText2d(0, -50, "Last selection: "
```

```
.. tostring(gLastSelection))

gChoice:Render(gRenderer)
gChoice:HandleInput()
end
```

Listing 2.149: Using the Selection class. In main.lua.

The code in Listing 2.149 creates a simple selection menu with two options, “Yes” and “No”. The OnSelection callback sets gLastSelection to equal the selection index. In the update function we draw the gLastSelection value on screen and we also draw the selection menu itself. The input for the selection menu is handled by calling its HandleInput function.

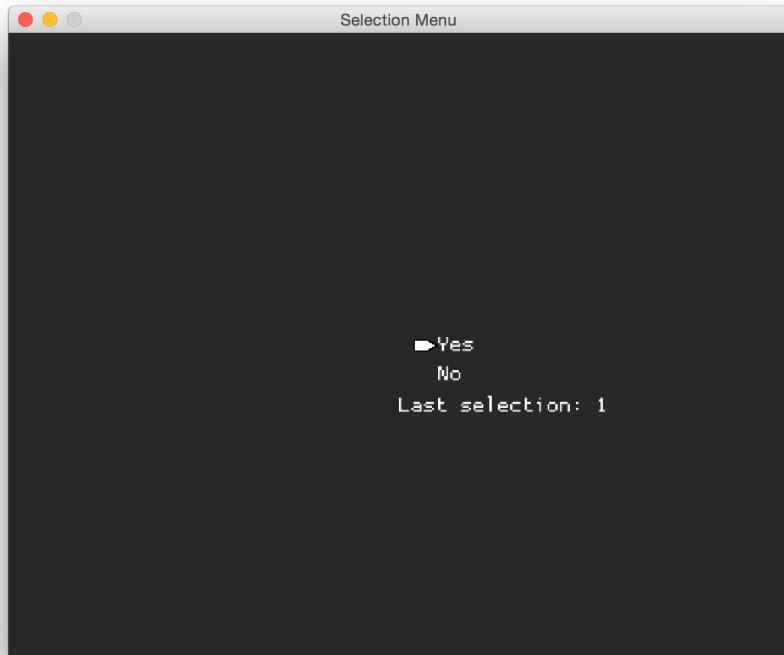


Figure 2.70: The selection menu up and running.

Run the program and you’ll see a screen similar to Figure 2.70. You can press up and down to scroll through the selection menu, and pressing space will select one of

the options. Try changing the params table, number of columns, spacing, etc. to get a feel for how it's put together. An up to date version of the code can be found in gui-10-solution.

Adding Choices to the Textbox

When presenting the player with an interesting choice we use a textbox. The textbox displays a choice to be made and a selection menu supplies the options. To present choices like this we need to be able to put a selection menu inside a textbox.

Let's start by updating the CreateFixed function as shown in Listing 2.150 to support selection menus. Example gui-11 has a new project with the code so far.

```
function CreateFixed(renderer, x, y, width, height, text, params)

    params = params or {}
    local avatar = params.avatar
    local title = params.title
    local choices = params.choices

    -- code omitted

    local boundsTop = padding
    local boundsLeft = padding
    local boundsBottom = padding

    local children = {}

    if avatar then
        -- code omitted
    end

    local selectionMenu = nil
    if choices then
        -- options and callback
        selectionMenu = Selection>Create
        {
            data = choices.options,
            OnSelection = choices.OnSelection,
        }
        boundsBottom = boundsBottom - padding*0.5
    end

    -- code omitted
```

```

    return Textbox>Create
    {

        -- code omitted

        wrap = wrap,
        selectionMenu = selectionMenu
    }
end

```

Listing 2.150: Updating the Fixed menu to use a selection menu. In main.lua.

The params table now accepts an optional new field, choices. If the choices table exists, we use it to add a selection menu to the textbox. If not, we just generate a normal textbox with no options.

In the if-statement the selection menu is created using the values from the choices table. The textbox height is enlarged to fit the selection menu and the boundsBottom is increased to give some more space around the menu. The selection menu is then passed through to the constructor and it's ready to use.

Next we need to update the Textbox class to support the selection menu. Copy the updated code from Listing 2.151.

```

function Textbox>Create(params)

    -- code omitted

    local this =
    {

        -- code omitted

        mSelectionMenu = params.selectionMenu
    }

    -- code omitted
end

```

Listing 2.151: Added options menu to the Fitted textbox creator. In Textbox.lua.

In Listing 2.151 we've modified the textbox constructor to assign any selectionMenu in the params table to the this.mSelectionMenu field. This selectionMenu is an optional parameter, and if it doesn't exist, no selection menu is rendered. Next let's add code to allow the user to interact with it. Copy the code from Listing 2.152.

```

function Textbox:HandleInput()

    if self.mSelectionMenu then
        self.mSelectionMenu:HandleInput()
    end

    if Keyboard.JustPressed(KEY_SPACE) then
        self:OnClick()
    end
end

```

Listing 2.152: New input functions for the Textbox. In Textbox.lua.

In Listing 2.152 we've added a HandleInput function to the textbox which passes input on to the selection menu if it exists. We're also checking for the space key to trigger the OnClick code. Let's add rendering next. Copy the code from Listing 2.153. We're rendering the menu directly after any text is drawn.

```

function Textbox:Render(renderer)

    -- code omitted

    renderer:DrawText2d(
        textLeft,
        textTop,
        self.mChunks[self.mChunkIndex],
        Vector.Create(1,1,1,1),
        self.mWrap * scale)

    if self.mSelectionMenu then
        renderer:AlignText("left", "center")
        local menuX = textLeft
        local menuY = bottom + self.mSelectionMenu:GetHeight()
        menuY = menuY + self.mBounds.bottom
        self.mSelectionMenu.mX = menuX
        self.mSelectionMenu.mY = menuY
        self.mSelectionMenu.mScale = scale
        self.mSelectionMenu:Render(renderer)
    end

```

Listing 2.153: Add SelectionMenu rendering to the Textbox. In Textbox.lua.

The selection menu is positioned so its left side matches the left side of the textbounds. It's top side is aligned with the bottom of the textbounds. This means it appears under

any text and is aligned to the text's left side. In the code we set the scale so it scales correctly with the rest of the textbox. Those are only changes required in Textbox.lua. We can now go back to the main.lua code and test it out.

Copy the code from Listing 2.154 to create a textbox with an embedded selection menu.

```
local width = System.ScreenWidth() - 4
local height = 102 -- a nice height
local x = 0
local y = -System.ScreenHeight()/2 + height / 2 -- bottom of the screen
local text = 'Should I join your party?'
local title = "NPC:"
local avatar = Texture.Find("avatar.png")

local textbox = CreateFixed(gRenderer, x, y, width, height, text,
{
    title = title,
    avatar = avatar,
    choices =
    {
        options = {"Yes", "No"},
        OnSelection = function(i) print('selected', i) end
    }
})

function update()
    local dt = GetDeltaTime()
    if not textbox:IsDead() then
        textbox:Update(GetDeltaTime())
        textbox:Render(gRenderer)
        textbox:HandleInput()
    end
end
```

Listing 2.154: Added a Fixed textbox with a selection menu. In main.lua.

The main menu code creates a textbox with two options, "Yes" and "No". The update loop tells the textbox to handle any input by calling its HandleInput function. Example gui-11-solution has the code so far. Run the code and you'll see something like Figure 2.71.

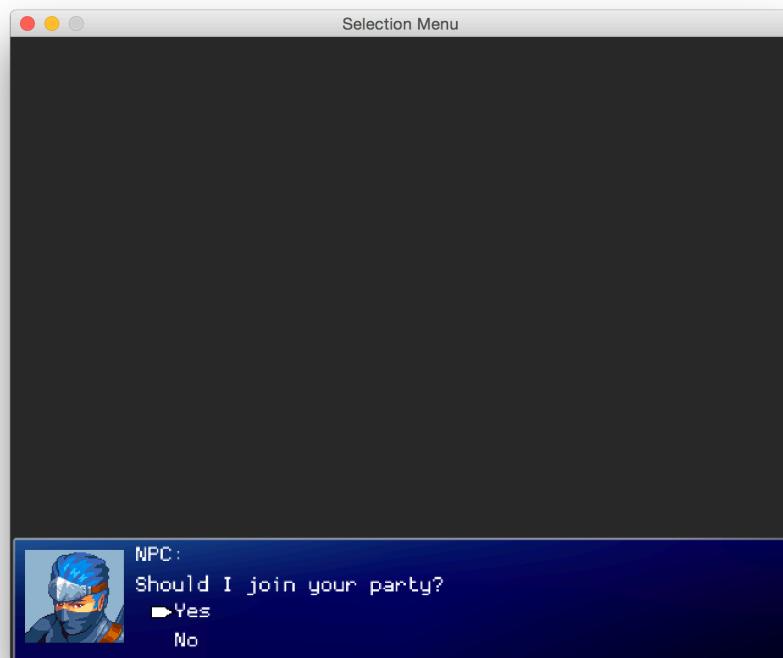


Figure 2.71: Presenting the user with a choice using a fixed textbox and menu.

Let's finish up our textbox class by considering a type of textbox that automatically sizes itself to its content; the fitted textbox.

Fitted Textbox

Fixed textboxes are great when we want to display a message without worrying about the textbox size. With a fitted textbox you just say "Hey I want some text here and put a textbox around it." With a fixed textbox you're saying "Hey I've got a textbox here and I want to fit this text inside it." Both approaches are useful depending on the situation. Figure 2.72 shows an example of each.



Figure 2.72: Fitted and Fixed comparison.

Example gui-12 has the code so far.

We don't need to write a lot of new code to make a fitted textbox because we can reuse our CreateFixed function. The CreateFitted function takes in some text and optional extras such as choices, avatar and the title. It calculates the area the contents take up, adds some padding, and calls CreateFitted.

In the main.lua copy the code from Listing 2.155 to add a new CreateFitted function.

```
function CreateFitted(renderer, x, y, text, wrap, params)

    local params = params or {}
    local choices = params.choices
    local title = params.title
    local avatar = params.avatar

    local padding = 10
    local panelTileSize = 3
    local textScale = 1.5

    renderer:ScaleText(textScale, textScale)

    local size = renderer:MeasureText(text, wrap)
    local width = size:X() + padding * 2
    local height = size:Y() + padding * 2

    if choices then
        -- options and callback
        local selectionMenu = Selection>Create
        {
            data = choices.options,
            displayRows = #choices.options,
            columns = 1,
        }
        height = height + selectionMenu:GetHeight() + padding * 4
        width = math.max(width, selectionMenu:GetWidth() + padding * 2)
    end

    if title then
        local size = renderer:MeasureText(title, wrap)
        height = height + size:Y() + padding
        width = math.max(width, size:X() + padding * 2)
    end
```

```

if avatar then
    local avatarWidth = avatar:GetWidth()
    local avatarHeight = avatar:GetHeight()
    width = width + avatarWidth + padding
    height = math.max(height, avatarHeight + padding)
end

return CreateFixed(renderer, x, y, width, height, text, params)
end

```

Listing 2.155: CreateFitted function.

Let's go through the CreateFitted function to see how it works. The first part is similar to CreateFixed but it differs at the call to MeasureText. MeasureText uses the current text scale of the renderer and the wrap width to work out how wide and tall the rendered text is in pixels. Then padding is added for the left, right, top and bottom borders. If there's no params table, the width and height are sent on to the CreateFixed function and a textbox is created that snugly fits our text.

If the choices parameter has been passed in then we need to work out the selection menu size. We do this by creating the menu and asking for its dimensions. The height is increased by the selection menu's height, with a little padding. The width is set to whichever is wider, the current text or the selection menu.

If the title parameter exists, its height is added to the height of the textbox, with a little padding. Once again the width is set to whichever is wider, the current text or the title. This is to ensure that when someone with a long name like "Old Farmer Fred" says something short like "Ok.", the textbox is wide enough for the title to fit.

Finally we check for the avatar and increase the width and height if it's present. The avatar reduces the text's width, whereas the title and selection menu affect the text vertically.

Once the height and width calculations are finished, they're passed along to the CreateFixed function to create the textbox. Listing 2.156 demonstrates the new fitted function. The code so far is available in example gui-12-solution.

```

local width = System.ScreenWidth() - 4
local height = 102 -- a nice height
local x = 0
local y = 0 -- bottom of the screen
local text = 'Should I join your party?'
local title = "NPC:"
local avatar = Texture.Find("avatar.png")

local textbox = CreateFitted(gRenderer, 0, 0, text, 300,
{

```

```
    title = title,
    avatar = avatar,
})

function update()
    -- code omitted
end
```

Listing 2.156: Using the CreateFitted function. In main.lua.

The fitted function is quite flexible. Get to know how it works by playing around and creating different types of textbox. It makes asking the player a quick question very simple.

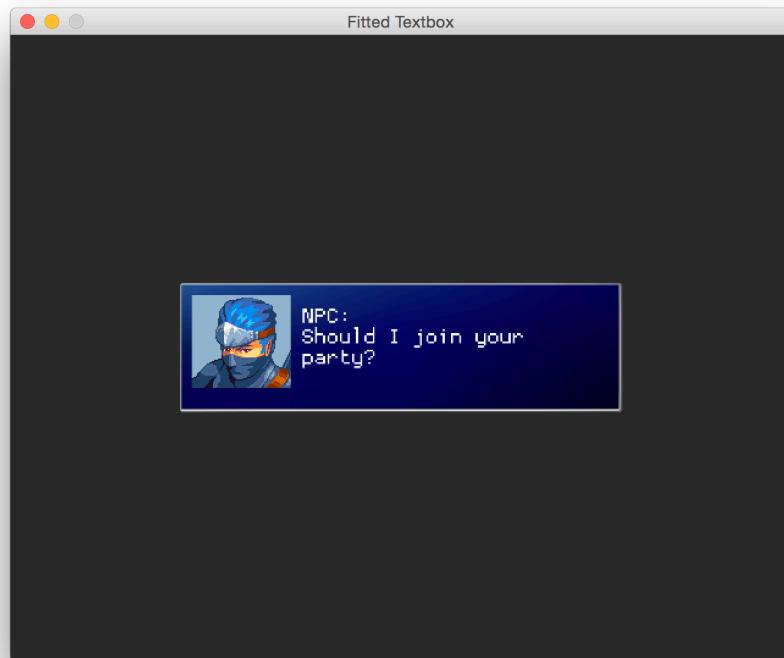


Figure 2.73: The Fitted Textbox in action.

Figure 2.73 shows what the program looks like when you run it.

Progress Bar

Progress bars are a UI element that display percentage information and can be filled from 0 to 100%. They're used in RPGs to represent player and enemy health, progress towards the next level, the amount of mana, and progress when loading a level or saving a game. You can see some progress bars in action in Figure 2.74. They're pretty useful!



Figure 2.74: Example uses of a progress bar.

A progress bar is created from two images the exact same size. One image depicts the bar empty and the other shows it totally full. We layer the images, on top of each other, by drawing the empty bar first and the full bar second. This arrangement can be seen in Figure 2.75. We'll refer to the empty bar as the background image and the full bar as the foreground.

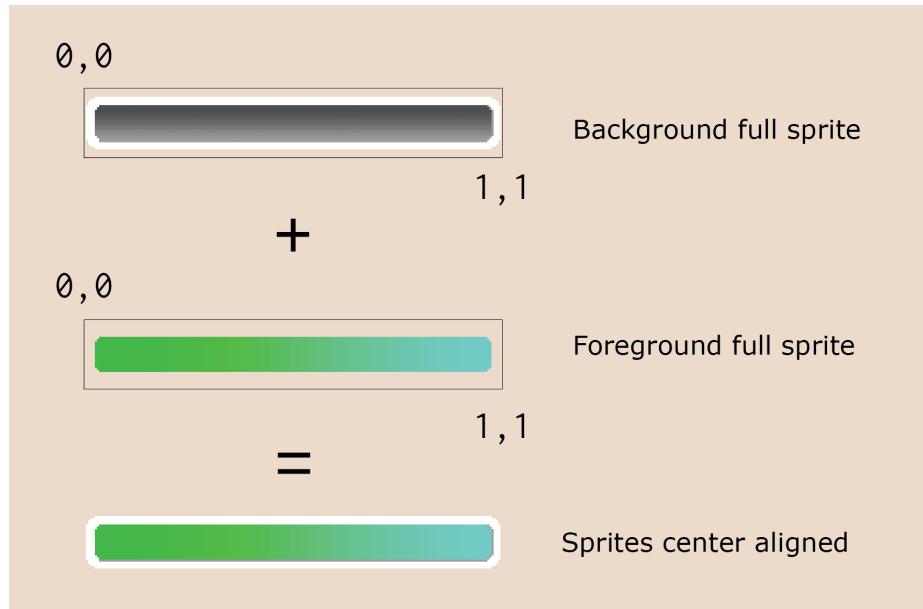


Figure 2.75: Layering the foreground and background images of a progress bar.

To make the bar appear filled, empty, or somewhere in between, we alter U, V information of the foreground sprite. If we cut the foreground sprite at 50% then the bar appears 50% full. We can make this cut anywhere we want to simulate the progress from 0 to 100% by changing the U value from 0 to 1. To make the effect complete we align the background and foreground sprites along their left edge. You can see how this effect is constructed in Figure 2.76.

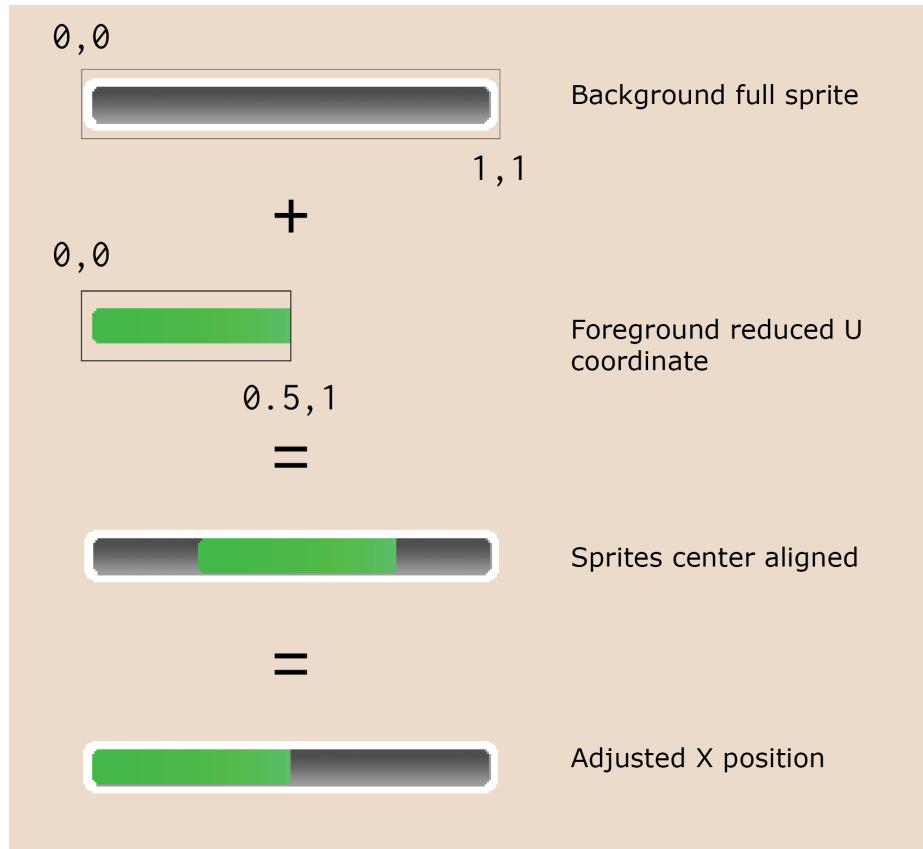


Figure 2.76: Changing the UV simulates different fills of the progress bar.

Let's make a class for the progress bar. Example gui-13 has all the code so far and contains an empty `ProgressBar.lua` file that's been added to the manifest and has had `Asset.Run` called on it in `main.lua`. The example project also contains two textures, "background.png" and "foreground.png", that we'll use to display the progress bar.

First let's consider how we'd ideally like to use the `ProgressBar`. A use case is presented in Listing 2.157 below.

```
local bar = ProgressBar:Create
{
    x = 0,
    y = 0,
    foreground = Texture.Find("foreground.png"),
    background = Texture.Find("background.png"),
    maximum = 100,
```

```

}

bar:SetValue(50)

function update()
    bar:Render(gRenderer)
end

```

Listing 2.157: How we might like to use a ProgressBar.

In the above code we create a progress bar called bar. We give it an x, y position and this is the position the progress bar is centered on. The foreground and background textures are exactly the same size. One shows the bar full and one shows it empty. You can see the two textures below in Figure 2.77.



Figure 2.77: The two textures we can use to make a progress bar.

The final parameter, maximum, represents the maximum value that the progress bar can hold. If a player has 100 HP then this value would be 100. It's that simple! The maximum number is optional, and it has a default value of 1.

Once the bar is created we call SetValue which takes a number from 0 to the maximum amount. In this case the value is set to 50 which is half the full amount and sets the progress bar to be half full.

Finally we Render the bar and that's it! There's no update function. To change the fill amount we just call SetValue again.

Copy the code from Listing 2.158 to start implementing the ProgressBar class.

```

ProgressBar = {}
ProgressBar.__index = ProgressBar
function ProgressBar:Create(params)
    params = params or {}

    local this =
    {
        mX           = params.x or 0,
        mY           = params.y or 0,
        mBackground  = Sprite.Create(),
        mForeground  = Sprite.Create(),

```

```

        mValue      = params.value or 0,
        mMaximum    = params.maximum or 1,
    }

    this.mBackground:SetTexture(params.background)
    this.mForeground:SetTexture(params.foreground)

    -- Get UV positions in texture atlas
    -- A table with name fields: left, top, right, bottom
    this.mHalfWidth = params.foreground:GetWidth() / 2

    setmetatable(this, self)
    this:SetValue(this.mValue)
    return this
end

function ProgressBar:SetValue(value, max)
    self.mMaximum = max or self.mMaximum
    self:SetNormalValue(value / self.mMaximum)
end

function ProgressBar:SetNormalValue(value)

    self.mForeground:SetUVs(
        0,                      -- left
        1,                      -- top
        value,                  -- right
        0)                     -- bottom

    local position = Vector.Create(
        self.mX - (self.mHalfWidth * (1 - value)),
        self.mY)

    self.mForeground:SetPosition(position)
end

```

Listing 2.158: The first pieces of the ProgressBar class. In ProgressBar.lua.

The constructor, from Listing 2.158, is pretty straightforward. It takes in a single params table, and all the params fields are added to the this table or a default value is used. Then the foreground and background sprites are set up. We store a mHalfWidth value that represents half the texture width in pixels; this is used to center align the progress bar. Finally we call SetValue to set the initial value.

After the constructor we define the SetValue function. SetValue transforms the passed in value into the range 0 - 1 by dividing the value by the mMaximum amount. In our

earlier example the maximum was 100 and the value was 50, so $50 / 100 = 0.5$. The SetValue function then calls SetNormalValue passing through the value in the 0 - 1 range.

SetNormalValue uses the 0 - 1 value to update the display of the progress bar. A 0 value displays a fully empty bar and a 1 value displays a fully filled bar. The value is used to change the UVs for the mForeground sprite. It keeps the top, bottom and left UV coordinates the same but sets the right to equal the passed in value. The value amount sets the point where the foreground image is cut off. In the case of a 0.5 value, the right is set to 0.5 and this cuts the sprite off at the halfway point. This can be seen Figure 2.76.

In Dinodeck, changing a sprite's U coordinate changes its width. Cutting the sprite U in half makes the sprite half as wide. Sprites are rendered from their center points. The center point changes when the sprite's width is reduced. This causes problems with alignment.

We need to make it so that the foreground has its left edge aligned with the left edge of the background. To do this we move the foreground sprite right according to how much of it has been removed. In the code this is calculated with the expression $-(self.mHalfWidth * (1 - value))$. The $1 - value$ gives the length of the part that has been removed. For example if the value is 0.2, then $1 - 0.2$ gives 0.8, the value of the amount that's been removed from the foreground. We use this number multiplied by half the width to place the bar in the correct position.

Once we've worked out the new position we set it using mForeground:SetPosition.

Copy the code from Listing 2.159 to add render and position functions to the Progress-Bar class.

```
function ProgressBar:SetPosition(x, y)
    self.mX = x
    self.mY = y
    local position = Vector.Create(self.mX, self.mY)
    self.mBackground:SetPosition(position)
    self.mForeground:SetPosition(position)
    -- Make sure the foreground position is set correctly.
    self:SetValue(self.mValue)
end

function ProgressBar:GetPosition()
    return Vector.Create(self.mX, self.mY)
end

function ProgressBar:Render(renderer)
    renderer:DrawSprite(self.mBackground)
    renderer:DrawSprite(self.mForeground)
end
```

Listing 2.159: The rendering and position functions. In ProgressBar.lua.

The SetPosition code sets the internal mX and mY values and then updates the background and foreground sprite positions. It also updates the value to correctly position the foreground sprite. The GetPosition and Render functions are both straightforward so let's try the new ProgressBar class out!

To test out the progress bar, update your main.lua to match Listing 2.160.

```
local bar = ProgressBar:Create
{
    x = 0,
    y = 0,
    foreground = Texture.Find("foreground.png"),
    background = Texture.Find("background.png"),
}

local tween = Tween:Create(1, 0, 1)

function update()
    local dt = GetDeltaTime()
    tween:Update(dt)
    local v = tween:Value()
    bar:SetValue(v)

    if tween:IsFinished() then
        tween = Tween:Create(v, math.abs(v - 1), 1)
    end

    bar:Render(gRenderer)
end
```

Listing 2.160: Testing the ProgressBar. In main.lua.

This code continually fills and empties a ProgressBar object. The latest version of the code is available as gui-13-solution. Run the code and you'll see something like Figure 2.78.

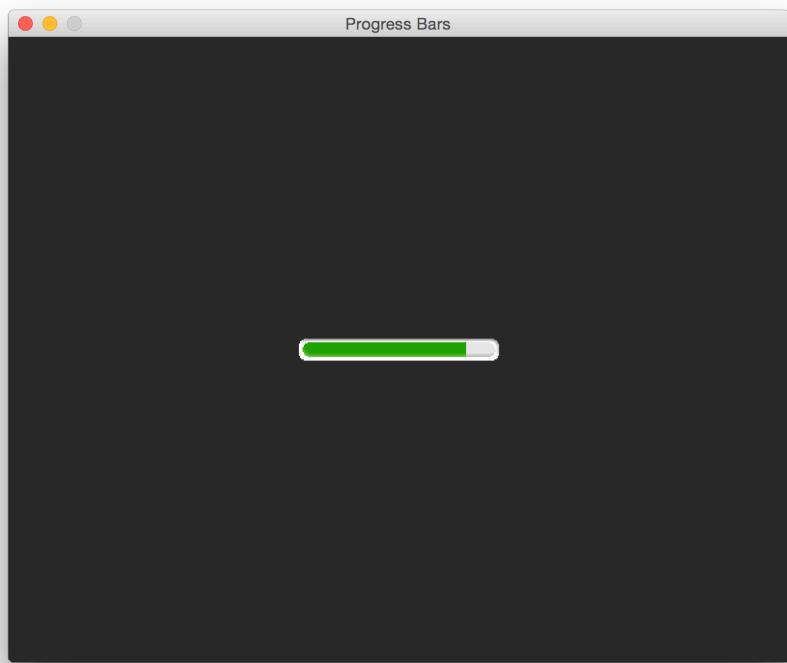


Figure 2.78: The progress bar as working in an example program.

Scrollbar

Scrollbars are a common UI element used to indicate what portion of information is currently being displayed. If you're viewing a computer screen you might even be able to see a scrollbar right now! Figure 2.79 shows an image of a scrollbar. The scrollbar pictured is vertical, and this is the only type we need for our RPG. The length of the scrollbar represents the total amount of information, and the caret (or thumb) represents the portion of information currently being displayed.



Figure 2.79: An example of a scrollbar.

When there's not enough room to display a large amount information to the user at once, we use a scrollbar.

Think of an inventory in a RPG; it could easily contain 1000 items. If each item requires only 32 pixels height and there are 3 items per row that would still require over 10,000 pixels to display everything! Higher resolutions help but only to a certain point. Even if we can display 1000 items on one screen we'd choose not to because it would be unreadable. As a solution we might code our inventory to display only 10 items at a time, but now we have a new problem! Which 10 items are being shown? This is the problem the scrollbar solves. With a quick glance we can look at scrollbar and see where we are in the list. If we're 50% through a list, the caret appears in the center of the scrollbar.

Scrollbars display our position in a list but they may signal more information by altering the size of the caret. If we're looking at a list of 100 items with 10 on screen at once, then at any time we're seeing 10% of the list. The scrollbar can represent the displayed portion of the list by scaling the caret to 10% of the scrollbar height. The user then has a rough idea of how long it's going to take to scroll through the list.

Example gui-14 has an empty project with a scrollbar.png texture, an empty Scrollbar.lua file, and our Utils.lua file. From these files we'll build up the code for the scrollbar.

The scrollbar.png texture is shown below at Figure 2.80.



Figure 2.80: The scrollbar texture.

Figure 2.80 shows that there are four sections to the scrollbar texture; the up and down arrows at the top, the background, and the caret. We're going to break this texture into four sprites and then stretch the background according to how tall we want the scrollbar.

Copy the constructor code from Listing 2.161.

```
Scrollbar = {}  
Scrollbar.__index = Scrollbar  
function Scrollbar:Create(texture, height)  
    local this =  
    {  
        mX = 0,  
        mY = 0,  
        mHeight = height or 300,  
        mTexture = texture,  
        mValue = 0,  
  
        mUpSprite = Sprite.Create(),  
        mDownSprite = Sprite.Create(),  
        mBackgroundSprite = Sprite.Create(),  
        mCaretSprite = Sprite.Create(),  
        mCaretSize = 1,  
    }  
  
    local texWidth = texture:GetWidth()  
    local texHeight = texture:GetHeight()  
  
    this.mUpSprite:SetTexture(texture)  
    this.mDownSprite:SetTexture(texture)  
    this.mBackgroundSprite:SetTexture(texture)  
    this.mCaretSprite:SetTexture(texture)  
    -- There are expected to be 4 equally sized pieces  
    -- that make up a scrollbar.  
    this.mTileHeight = texHeight/4  
    this.mUVs = GenerateUVs(texWidth, this.mTileHeight, texture)  
    this.mUpSprite:SetUVs(unpack(this.mUVs[1]))  
    this.mCaretSprite:SetUVs(unpack(this.mUVs[2]))
```

```

this.mBackgroundSprite:SetUVs(unpack(this.mUVs[3]))
this.mDownSprite:SetUVs(unpack(this.mUVs[4]))

-- Height ignore the up and down arrows
this.mLineHeight = this.mHeight - (this.mTileHeight * 2)

setmetatable(this, self)
this:SetPoint(0, 0)
return this
end

```

Listing 2.161: Scrollbar constructor. In Scrollbar.lua.

The scrollbar constructor sets up four sprites to represent the different pieces of the scrollbar. It takes a texture parameter, which it breaks into 4 pieces, and a height parameter that determines the height of the scrollbar. If no height is passed in, it's set to 300 pixels and stored in the this table under `mHeight`.

The texture is stored in `mTexture`. The `mValue` variable represents the position of the caret on the scrollbar. The `mValue` field should always be in the range 0 to 1 and it's set to 0 by default.

After setting up the `this` table we set the texture for each of the sprites. The `GenerateUVs` function, from `Util.lua`, is used to create four sets of UVs for each piece of the scrollbar. These UVs are then set for each sprite.

Near the end of the constructor we set the `mLineHeight`. This is the height of the scrollbar, ignoring the up and down arrows. You can see this visualized in Figure 2.81. We use the `mLineHeight` when positioning the caret.

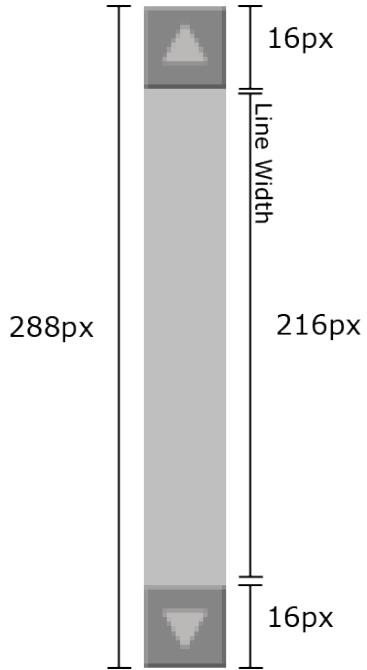


Figure 2.81: The line height of the scrollbar.

At the end of the constructor the metatable is set and SetPosition is called, which positions and scales the four sprites that make up the scrollbar.

The SetPosition code is show below in Listing 2.162.

```
function Scrollbar:SetPosition(x, y)
    self.mX = x
    self.mY = y

    local top = y + self.mHeight / 2
    local bottom = y - self.mHeight / 2
    local halfTileHeight = self.mTileHeight / 2

    self.mUpSprite:SetPosition(x, top - halfTileHeight)
    self.mDownSprite:SetPosition(x, bottom + halfTileHeight)

    self.mBackgroundSprite:SetScale(1, self.mLineHeight / self.mTileHeight)
    self.mBackgroundSprite:SetPosition(self.mX, self.mY)
    self:SetNormalValue(self.mValue)
end
```

```

function Scrollbar:SetNormalValue(v)
    self.mValue = v

    self.mCaretSprite:SetScale(1, self.mCaretSize)
    -- caret 0 is the top of the scrollbar
    local caretHeight = self.mTileHeight * self.mCaretSize
    local halfCaretHeight = caretHeight / 2
    self.mStart = self.mY + (self.mLineHeight / 2) - halfCaretHeight

    -- Subtracting caret, to take into account the first -halfcaret
    -- and the one at the other end
    self.mStart = self.mStart -
        ((self.mLineHeight - caretHeight) * self.mValue)

    self.mCaretSprite:SetPosition(self.mX, self.mStart)
end

```

Listing 2.162: Setting the position of the scrollbar. In Scrollbar.lua.

SetPosition sets where the scrollbar is drawn and scales the background sprite and caret according to mValue and the height of the bar. The top and bottom arrows sprites are positioned at the top and bottom of the scrollbar. Sprites are drawn from the center, so we adjust the positions by half the height to get them flush with the boundary of the scrollbar.

The background sprite is positioned at the center of the bar and scaled to fill in the space between the two arrow sprites. This is done by dividing the line height by the height of the background image, which gives the number of times the background image needs to be scaled to fully fit the background area.

The SetPosition function ends by calling SetNormalValue to position the caret.

SetNormalValue takes in a value from 0 to 1 and assigns it to mValue. The caret is scaled on the Y axis according to mCaretSize. The pixel height of the caret is calculated by multiplying the original caret height by its current scale.

A new variable is created, mStart, that represents the start of the scroll area. If the caret's Y position is set to mStart, the caret is rendered at the top of the bar perfectly flush with the top arrow. We use this start position with an offset to scroll the caret down the bar. The mStart position is calculated by adding half the mLineHeight and subtracting half the caret height.

Once we've calculated mStart, we modify it using mValue to get the actual position where we want to render the caret. We get the Y position for the caret by multiplying the mLineHeight by the mValue and subtracting half the caret height.

Let's implement Render and SetScrollCaretScale next. Copy the code below from Listing 2.163.

```

function Scrollbar:Render(renderer)
    renderer:DrawSprite(self.mUpSprite)
    renderer:DrawSprite(self.mBackgroundSprite)
    renderer:DrawSprite(self.mDownSprite)
    renderer:DrawSprite(self.mCaretSprite)
end

function Scrollbar:SetScrollCaretScale(normalValue)
    self.mCaretSize = (self.mLineHeight * normalValue)
        / self.mTileHeight

    -- Don't let it go below 1
    self.mCaretSize = math.max(1, self.mCaretSize)
end

```

Listing 2.163: The Scrollbar Render and SetScrollCaretScale functions. In Scrollbar.lua.

All the sprite positions are already in the correct location so the Render function just renders them out.

The SetScrollCaretScale function is a little more interesting. It takes in a value from 0 to 1 to determine how large the caret appears as a percentage of the scrollbar height. Calling the function with a 1 makes the caret fill up the entire area, 0.5 half the area, 0.3 a third, and so on. By default the caret is set to match its pixel size in the texture.

The scrollbar code is now complete! Let's test it out in main.lua.

In the code below Listing 2.164 we create 3 scrollbars of different sizes, values and caret scales. Play with the numbers and get a feel for how it all works.

```

local bar1 = Scrollbar>Create(Texture.Find("scrollbar.png"), 100)
local bar2 = Scrollbar>Create(Texture.Find("scrollbar.png"), 200)
local bar3 = Scrollbar>Create(Texture.Find("scrollbar.png"), 75)

bar1:SetScrollCaretScale(0.5)
bar1:SetNormalValue(0.5)
bar1:SetPoint(-50, 10)

bar2:SetScrollCaretScale(0.3)
bar2:SetNormalValue(0)
bar2:SetPoint(0, 0)

bar3:SetScrollCaretScale(0.1)
bar3:SetNormalValue(1)
bar3:SetPoint(50, -10)

```

```
function update()
    bar1:Render(gRenderer)
    bar2:Render(gRenderer)
    bar3:Render(gRenderer)
end
```

Listing 2.164: Using the scrollbar. In main.lua.

This code is available in gui-14-solution. The screenshot below in Figure 2.82 shows what you'll see when you run it.

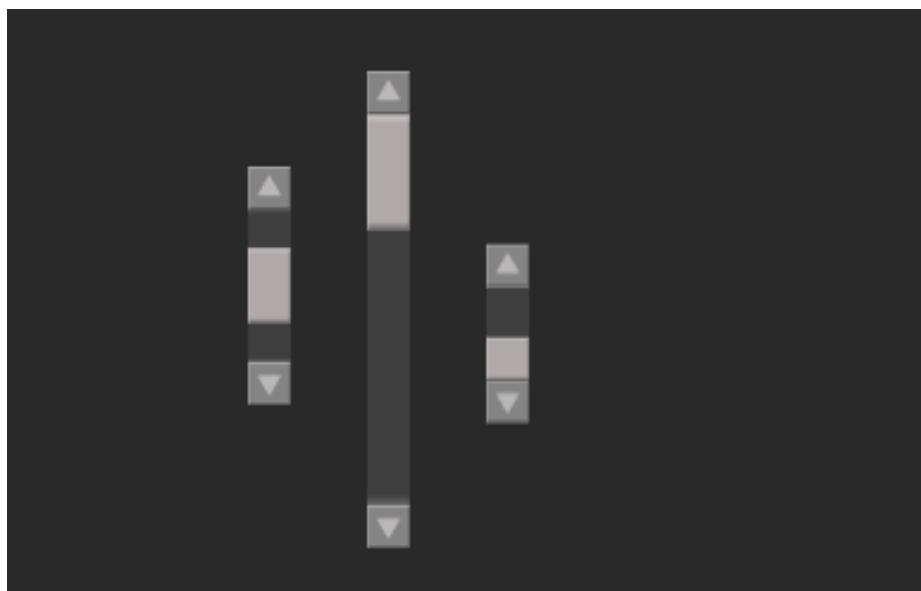


Figure 2.82: Three scrollbars with different settings.

That's the last UI element we need to create in this chapter. With this set of elements we should be able to make 90% of all menu code we'll need in an RPG.

Stacking Everything Together

Before ending, let's write a little code demonstrating how to display multiple dialog boxes.

Create a new class StateStack in a new StateStack.lua file. This will manage all our textboxes. There's a project called gui-15 that contains this new file, and it's already been added to the manifest and run in the main.lua.

This stack code allows multiple textboxes to be shown at once but only the top textbox gets input. Listing 2.165 shows the main body of the code.

```
StateStack = {}
StateStack.__index = StateStack
function StateStack:Create()
    local this =
    {
        mStates = {}
    }

    setmetatable(this, self)
    return this
end

function StateStack:Update(dt)
    -- update them and check input
    for k, v in ipairs(self.mStates) do
        v:Update(dt)
    end

    local top = self.mStates[#self.mStates]

    if not top then
        return
    end

    if top:IsDead() then
        table.remove(self.mStates)
        return
    end

    top:HandleInput()
end

function StateStack:Render(renderer)
    for _, v in ipairs(self.mStates) do
        v:Render(renderer)
    end
end
```

Listing 2.165: The code for the StateStack class. In StateStack.lua.

In Listing 2.165 the constructor is basically empty. We just create an empty table called mStates and store it in the this table. The mStates table is where we'll put our

textboxes. The Update function iterates through all the entries in the `mStates` table calling Update on each one. After updating the states, we take the top state from the stack and if it's `dead` we remove it and return. If the top isn't dead we call HandleInput on it.

The Render function renders each of the states in order. For our example each state is a textbox. The boxes near the top of the stack are rendered last and on top of everything else. This can be seen in Figure 2.83.

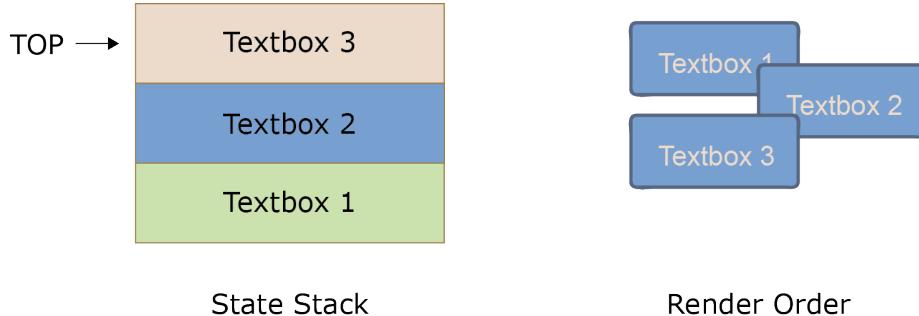


Figure 2.83: The relationship between the stack and render order.

We're testing textboxes, so we're going to add functions to make it easy to create them on the stack. Let's move our `CreateFitted` and `CreateFixed` functions from `main.lua` to the `StateStack` class in `StateStack.lua`. Rename them to `AddFitted` and `AddFixed` respectively. At the end of the functions, instead of returning a textbox we add it to the `mStates` table. Listing 2.166 shows the overview of these changes and the full changes are also available in `gui-15-solution`.

```
function StateStack:AddFixed(renderer, x, y, width, height, text, params)
    -- code omitted

    local textbox = Textbox:Create
    {
        -- code omitted
    }

    table.insert(self.mStates, textbox)
end

function StateStack:AddFitted(renderer, x, y, text, wrap, params)
    -- code omitted

```

```
    return self:AddFixed(renderer, x, y, width, height, text, params)
end
```

Listing 2.166: Add the textbox creation functions to the StateStack. In StateStack.lua.

That's it we're ready to use the stack! Here's the code for main.lua in Listing 2.167.

```
gRenderer = Renderer.Create()

stack = StateStack>Create()

stack:AddFitted(gRenderer, 0, 0, "Hello!")
stack:AddFitted(gRenderer, -25, -25, "World!")
stack:AddFitted(gRenderer, -50, -50, "Lots")
stack:AddFitted(gRenderer, -75, -75, "of")
stack:AddFitted(gRenderer, -100, -100, "boxes!")

function update()
    stack:Update(GetDeltaTime())
    stack:Render(gRenderer)
end
```

Listing 2.167: Using the StateStack. In main.lua.

If you run this code you'll see an image like the one shown in Figure 2.84.

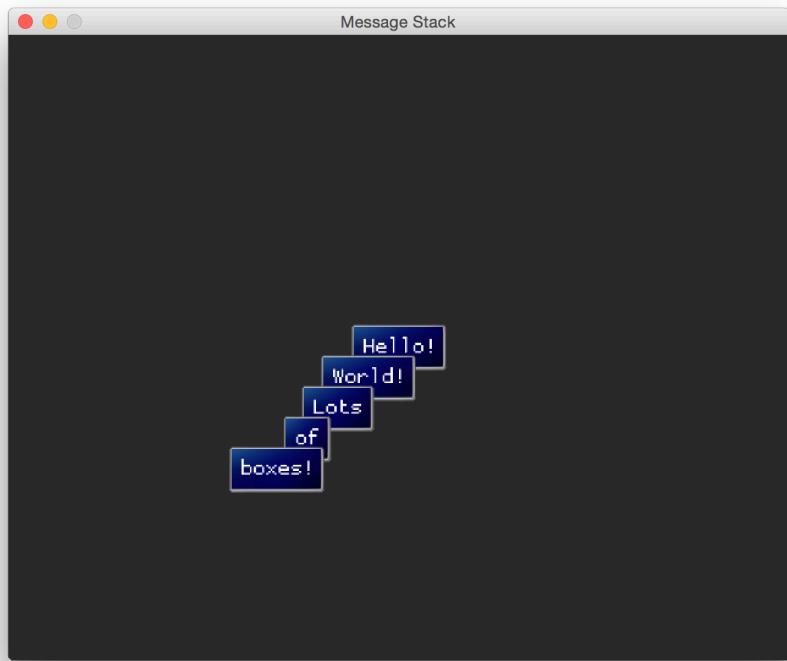


Figure 2.84: Multiple textboxes displayed using the stack.

The state stack helps us easily show and manage multiple textboxes. We're going to end the chapter here. In this chapter we've made panels we can use to create menus and textboxes. We've created two types of textbox that we can use for many different purposes. The textboxes support titles, avatar portraits, and the ability to select from a list of choices. We covered progress bars for displaying player health, magic, and experience points. Finally we built the state stack. These UI elements give us an excellent base for what we'll need to create a great game.

Menus

In the previous chapter on user interface we built up a useful collection of tools; a panel, textbox, progress bar, scrollbar, a selection menu and a state stack to manage them. In this chapter we're going to use these tools to build the start of an in-game RPG menu. This menu displays the party members, allows you to look at character status information, browse the inventory and that type of thing. We're not going to implement

it all now but we are going to lay down the framework that we'll flesh out during the rest of the book.

In the same way that all RPGs are different, so too are their menu systems. You should never try to write generalized menu code. Doing so is a trap. Concentrate on the game you're making, not the hypothetical games you might make in the future. It's best to create the menu specifically and only for the game you are creating. In this chapter we'll create a menu for a typical RPG which should make an excellent base for you to customize and use for your own creations.

By the end of this chapter we'll have a simple in-game menu that opens from the map exploration state.

Game Flow

The in-game menu is the menu you can open while wandering around a tilemap. The menu has a number of choices such as

- Items
- Magic
- Status
- Equipment
- Save Game

Apart from "Save Game" each of these choices loads a different submenu. As a player you'll bring the menu up when you want to tweak or upgrade your characters or when you want to save the game. An example in-game menu is shown in Figure 2.85.



Figure 2.85: An example of an in game menu.

RPGs contain a lot of code and this code can quickly become overwhelming unless we're proactive about managing it. Consider how the in-game menu is used in the flow of the game. The menu is activated from the tilemap, which means on the code side we're in the *exploration game state*. Opening the menu pauses the exploration state and transfers control to the menu state. The menu state has several submenus which are again represented by separate states. At any time the player can back out of the menu and return to the tilemap.

The basic game flow is illustrated by Figure 2.86.

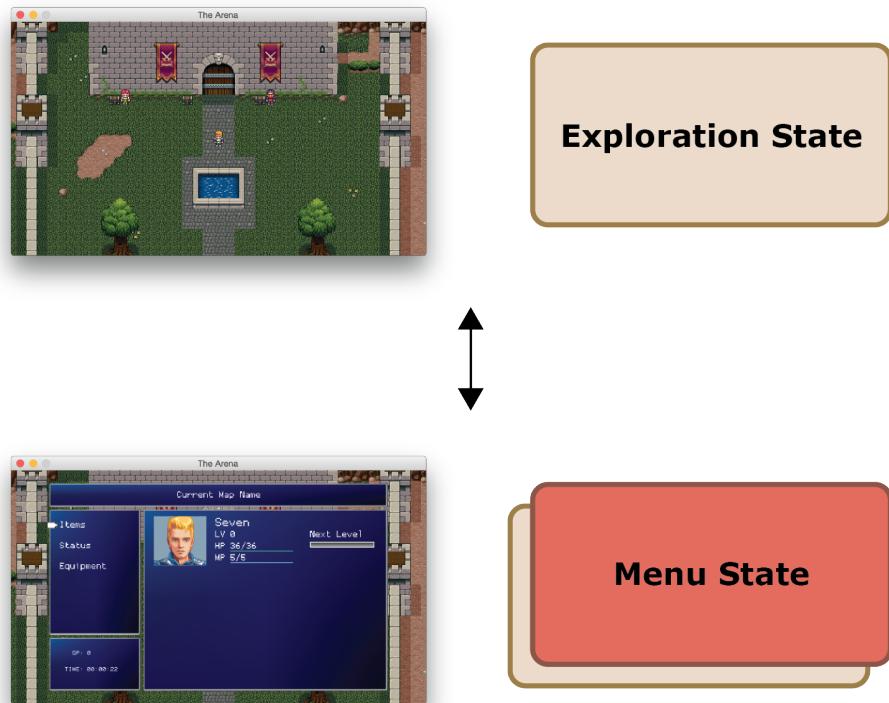


Figure 2.86: The basic game flow from map to menu.

To manage the transitions between the exploration and menu states we could use a StateMachine but we're not going to. The menu is drawn on top of the exploration screen. We want to keep the explore state live and partially visible in the background. The StateStack is better suited for the flow we want.

In the menu state itself we'll use a StateMachine to represent the submenus. The flow between the menu states is shown in Figure 2.87.

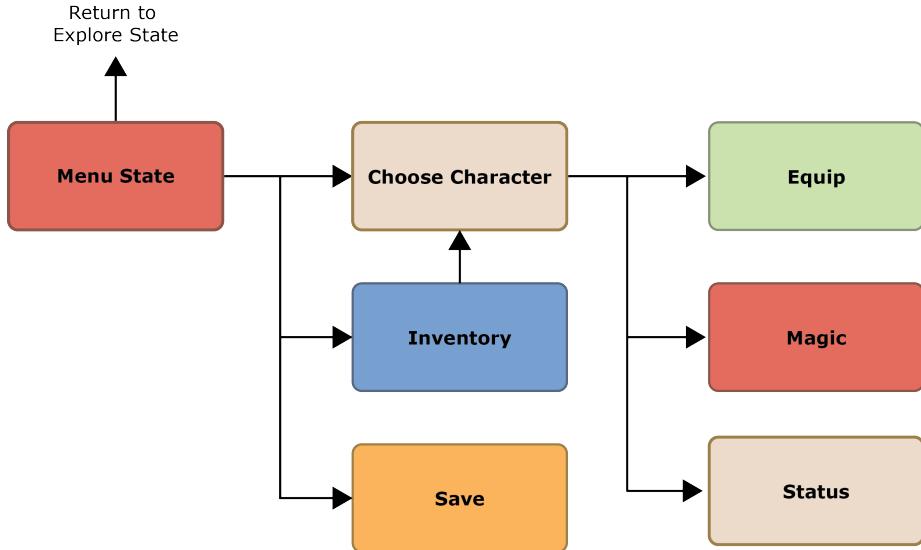


Figure 2.87: The states and flow between the menu's substates.

Figure 2.87 is only an approximation of the menu flow. It's not made explicit in the diagram but the user is able to return to the main menu from the equipment menu etc.

If the player chooses the *Save* option, we save and return to the main menu, but if the player chooses *Status* we need to know which character's status to display. The *ChooseCharacter* state lets us ask the player which character they want to pick.

Menu flow differs depending on the exact nature of the RPG. Is there a vehicle state, a materia state, a skill board or a pet breeding state? All RPGs have their own needs. If we architecture the menu using a state machine then flow is clear and it's easy to add and remove states, as well as alter the flow between them. Starting with a state diagram like Figure 2.87 makes things much easier when the time comes to write the code.

Let's write some code! We'll start by updating the state stack so we can use it to organize the game structure.

Stacking States

Example menu-1 combines the map rendering and UI element code that we developed previously. Look in the example folder and you'll notice all the image and code files are now sorted into three subdirectories called:

- “original_assets”
- “art”
- “code”

These folders help keep things a little more organized. The “original_assets” folder stores files that aren’t used directly by the project but help generate code or art files, such as Tiled’s .tmx files or original Photoshop files. The art and code directories are hopefully pretty self-explanatory! If you refer to the manifest.lua you’ll see that the path variables have been changed to point to these subdirectories.

Run the code and you’ll see a small map you can walk around. We’re going to use this as a basis for creating two states, the explore state which handles tilemaps and an empty menu state. We’ll then add a state stack to manage the states and introduce a transition from the explore state to the menu state and back again. Finally we’ll build up the menu state into something useful.

A state stack is, simply, a stack of states. Every time you add a new state it goes on top of the stack and is rendered last. Stacks have two main operations, push and pop. Push is for adding a new state on the top of the stack and pop is for removing a state from the top.

For our stack we’ll optionally render and update all the states but only handle input for the top state. Rendering and updating each state means states can be seen beneath the top state as shown in Figure 2.88.

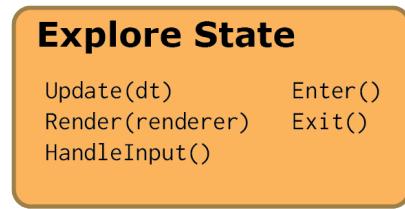


Figure 2.88: Stacking States.

Figure 2.88 shows the explore state and in-game menu state drawn separately, then drawn together as a stack.

Cleaning Up the Code

Look at main.lua and check out the number of Asset.Run commands. There are a lot! This list of Asset.Runs at the top of the main file is ugly. Let's move them into a separate file called Dependencies.lua. Remember to add it to your manifest! Replace all the LoadLibrary and Asset.Run calls at the top of main.lua with the two lines shown in Listing 2.168.

```
-- Replacing all previous LoadLibrary and Asset.Run calls
LoadLibrary('Asset')
Asset.Run('Dependencies.lua')

-- code omitted
```

Listing 2.168: Using Dependencies.lua to reduce the amount of setup code. In main.lua.

Let's move the LoadLibrary and Asset.Run commands to Dependencies.lua. We're not just going to move them, we're going to trim them down a little. No more repeated function calls! To do this we're going to use a function that takes in a list and a function and applies that function to every element in the list. We're going to call this function Apply¹⁰. It's defined below in Listing 2.169.

```
function Apply(list, f, iter)
    iter = iter or ipairs
    for k, v in iter(list) do
        f(v, k)
    end
end
```

Listing 2.169: Apply calls a function on every element of a list. In Dependencies.lua.

The code is straightforward. Pass in a table, a function and an optional iterator. By default the Apply function uses ipairs to iterate over the list. If you need to iterate over a table with non-integer keys then you can just pass in pairs as the third argument. The code calls the function, f, on every table element.

Listing 2.170 shows how we use the apply function in Dependencies.lua.

```
function Apply(list, f, iter)
    iter = iter or ipairs
    for k, v in iter(list) do
        f(v, k)
    end
end

Apply({
    "Renderer",
    -- code omitted
```

¹⁰In functional programming languages you might see this function called Map but I think Apply is more descriptive for how we'll be using it.

```

    "Vector",
    "Keyboard",
},
function(v) LoadLibrary(v) end)

Apply({
    "Animation.lua",
    "Map.lua",

    -- code omitted

    "Selection.lua",
    "Textbox.lua"
},
function(v) Asset.Run(v) end)

```

Listing 2.170: Reducing repeated code with the Apply function. In Dependencies.lua.

That's removed some of the clutter from the main.lua file. From now on, when we want to add a script or library, we'll add it in Dependencies.lua. The Apply function is useful for writing terse code but in the book I'll tend not to use it in order to be explicit about what the code is doing.

Explore State

The main.lua is now cleaner but there's still an awful lot of code and global variables in there. Most of the code is related to exploring the map. We'll bring all the map exploring code together in a state called ExploreState.

Create a new file called ExploreState.lua. Add it to the manifest and the Dependencies.lua file. The ExploreState displays a map and lets us wander around it. The state requires Enter, Exit, Render and Update functions. Since we're using it with a StateStack, and want to restrict the input, we'll need a HandleInput function too.

Listing 2.171 shows the code moved into the ExploreState.lua state.

```

ExploreState = {}
ExploreState.__index = ExploreState
function ExploreState:Create(stack, mapDef, startPos)
    local this =
    {
        mStack = stack,
        mMapDef = mapDef,
    }

```

```

this.mMap = Map:Create(this.mMapDef)
this.mHero = Character:Create(gCharacters.hero, this.mMap)
this.mHero.mEntity:SetTilePos(
    startPos:X(),
    startPos:Y(),
    startPos:Z(), this.mMap)
this.mMap:GotoTile(startPos:X(), startPos:Y())

setmetatable(this, self)
return this
end

function ExploreState:Enter() end
function ExploreState:Exit() end

function ExploreState:Update(dt)

local hero = self.mHero
local map = self.mMap

-- Update the camera according to player position
local playerPos = hero.mEntity.mSprite:GetPosition()
map.mCamX = math.floor(playerPos:X())
map.mCamY = math.floor(playerPos:Y())

hero.mController:Update(dt)
for k, v in ipairs(map.mNPCs) do
    v.mController:Update(dt)
end
end

function ExploreState:Render(renderer)

local hero = self.mHero
local map = self.mMap

renderer:Translate(-map.mCamX, -map.mCamY)
local layerCount = map:LayerCount()

for i = 1, layerCount do

    local heroEntity = nil
    if i == hero.mEntity.mLayer then
        heroEntity = hero.mEntity
    end

```

```

        map:RenderLayer(gRenderer, i, heroEntity)

    end

    renderer:Translate(0, 0)
end

function ExploreState:HandleInput()

    if Keyboard.JustPressed(KEY_SPACE) then
        -- which way is the player facing?
        local x, y = self.mHero:GetFacedTileCoords()
        local layer = self.mHero.mEntity.mLayer
        local trigger = self mMap:GetTrigger(layer, x, y)
        if trigger then
            trigger:OnUse(self.mHero)
        end
    end

end

```

Listing 2.171: The ExploreState created with code from the main.lua. In ExploreState.lua.

There's not much new in this code. It's just been rearranged and divided between the Update, Render and HandleInput functions. At the end of the Render function there's a call to Renderer.Translate to reset the renderer's translation to zero. This reset stops the map's offset affecting the UI's positions. The GetFaceTileCoords function is now a member of the Character class instead of a global function. It's shown below in Listing 2.172.

```

function Character:GetFacedTileCoords()

    -- Change the facing information into a tile offset
    local xInc = 0
    local yInc = 0

    if self.mFacing == "left" then
        xInc = -1
    elseif self.mFacing == "right" then
        xInc = 1
    elseif self.mFacing == "up" then
        yInc = -1
    elseif self.mFacing == "down" then

```

```

        yInc = 1
    end

    local x = self.mEntity.mTileX + xInc
    local y = self.mEntity.mTileY + yInc

    return x, y
end

```

Listing 2.172: The ExploreState created with code from the main.lua. In Character.lua.

Let's create the ExploreState in main.lua and run it from the update loop. Copy the changes to main.lua from Listing 2.173. This code doesn't use a state stack yet so we pass in a nil as the first parameter to the ExploreState constructor.

```

LoadLibrary('Asset')
Asset.Run('Dependencies.lua')

gRenderer = Renderer.Create()

local mapDef = CreateMap1()
mapDef.on_wake = {}
mapDef.actions = {}
mapDef.trigger_types = {}
mapDef.triggers = {}

-- 11, 3, 1 == x, y, layer
local state = ExploreState>Create(nil, mapDef, Vector.Create(11, 3, 1))

function update()
    local dt = GetDeltaTime()
    state:Update(dt)
    state:HandleInput()
    state:Render(gRenderer)
end

```

Listing 2.173: The main.lua code. In main.lua.

Running this code displays a map the player can explore. This is nothing we haven't seen before, but now the flow is going through a state stack. The complete code for this section is available as menu-1-solution.

This section has mainly been shuffling code around but it needed doing to make room for what's to come.

StateStack Code

In the UI chapter we created a state stack to display and manage multiple dialog boxes. The current stack code is great for dialog boxes, but to handle real game states we need to make it more robust. Example menu-2 contains the code so far.

To begin with, our stack doesn't even have a Push or Pop function! These are the *fundamental* stack functions, so let's add them first. We'll also add a Top function to return the top state. The code is below in Listing 2.174.

```
function StateStack:Push(state)
    table.insert(self.mStates, state)
    state:Enter()
end

function StateStack:Pop()
    local top = self.mStates[#self.mStates]
    table.remove(self.mStates)
    top:Exit()
    return top
end

function StateStack:Top()
    return self.mStates[#self.mStates]
end
```

Listing 2.174: The Pop, Push and Top functions for the stack. In StackStack.lua.

The current StateStack update loop is custom made to handle dialog boxes; we're going to generalize it. These changes mean we'll be tinkering with the Textbox code, one last time. Let's update the StateStack first. Copy the updated Update function from Listing 2.175.

```
function StateStack:Update(dt)
    -- update them and check input
    for k = #self.mStates, 1, -1 do
        local v = self.mStates[k]
        local continue = v:Update(dt)
        if not continue then
            break
        end
    end

    local top = self.mStates[#self.mStates]
```

```

if not top then
    return
end

top:HandleInput()
end

```

Listing 2.175: A slim, Update function to use states. In StateStack.lua.

The revised Update function, in Listing 2.175, is a little slimmer because input handling is performed by the top state rather than the stack.

The Update loop is a little funky. First off, it's iterating through the `mState` table backwards. That's because the most important state is at the top and it needs updating first. Each state can return a value, stored in the `continue` variable. If `continue` is false then the loop breaks and no subsequent states are updated. This lets states decide if they want states beneath them to update or not. For example, let's say you're running a cutscene, it's fullscreen, and you might not want to update the map state below because that cuts into your processing time. In this case after updating the cutscene, you'd return false and all states below would be ignored.

The final section of the update function is similar to what existed before; the top of the stack is told to handle any input.

Update the `main.lua` to what's shown in Listing 2.176 and check out the stack in action!

```

local stack = StateStack>Create()
local state = ExploreState>Create(stack, mapDef, Vector.Create(11, 3, 1))
stack:Push(state)

function update()
    local dt = GetDeltaTime()
    stack:Update(dt)
    stack:Render(gRenderer)
end

```

Listing 2.176: Using the stack in main. In main.lua.

Run this program and you'll see it works the same as before but now the state is being managed by the state stack. To show off the state stack we need at least two states. Let's edit the `Textbox` class so it too can work as a state in the modified `StateStack`. Copy the updated `Textbox` code from Listing 2.177.

```
function Textbox>Create(params)
```

```

-- code omitted
local this =
{
    mStack = params.stack,
    mDoClickCallback = false,

    -- code omitted
}

function Textbox:OnClick()

    if self.mSelectionMenu then
        self.mDoClickCallback = true
    end

    -- code omitted
end

function Textbox:HandleInput()

    if self.mSelectionMenu then
        self.mSelectionMenu:HandleInput()
    end

    if Keyboard.JustPressed(KEY_SPACE) then
        self:onClick()
    end
end

function Textbox:Enter() end
function Textbox:Exit()
    if self.mDoClickCallback then
        self.mSelectionMenu:onClick()
    end
end

function Update(dt)
    self.mTime = self.mTime + dt
    self.mAppearTween:Update(dt)
    if self:IsDead() then
        self.mStack:Pop()
    end
    return true
end

```

Listing 2.177: Making the textbox into a state. In Textbox.lua.

The code in figure Listing 2.177 makes several changes to the existing Textbox functions.

The Create function adds an extra field for the stack, `mStack`, and a new field called `mDoClickCallback`. The Textbox can have an embedded selection menu and this needs its `OnClick` callback calling when appropriate. Look at the `Textbox:OnClick` function and you'll see the `mDoClickCallback` is set to true if there's a selection menu. Then we deal with the callback in the `Exit` function when the textbox is dismissed.

The `HandleInput` function checks for key presses and calls the relevant functions.

The `Enter` function does nothing but is required as it's part of the expected interface for a state.

The `Update` function now returns true to indicate states below can be rendered and updated.

Let's get this updated Textbox on to our stack. We'll rename `StackState.AddFitted` and `StackState.AddFixed` to `StateStack.PushFit` and `StateStack.PushFix` to match how they're used in the stack. The `PushFix` function now also passes the stack into the `Textbox` constructor. Update the code to match Listing 2.178.

```
function StateStack:PushFix(renderer, x, y, width, height, text, params)

    -- code omitted
    local textbox = Textbox>Create
    {
        -- code omitted
        selectionMenu = selectionMenu,
        stack = self
    }
    table.insert(self.mStates, textbox)
end

function StateStack:PushFit(renderer, x, y, text, wrap, params)

    -- code omitted

    return self:PushFix(renderer, x, y, width, height, text, params)
end
```

Listing 2.178: Passing the stack through to the textbox. In StackStack.lua.

With these changes we're ready to use the `Textbox` and `ExploreState` together! Try it out with the code shown in Listing 2.179.

```
stack:Push(state)
stack:PushFit(gRenderer, 0, 0, "You're trapped in a small room.")
```

Listing 2.179: Adding an extra state. In main.lua.

Run the game and you'll get something like Figure 2.89. The up to date code is available in menu-2-solution.



Figure 2.89: A textbox state stacked on top of the ExploreState.

Fade In State

Often when starting a game the screen starts black and slowly fades into the game. This type of fade effect is easy to add now that we have a state stack. First we create a new fade state rendering a black rectangle covering the entire screen and then we

use a tween to make the rectangle more and more transparent over time. When the rectangle is totally transparent the fade state is popped off the stack and the player can interact with the game.

Example menu-3 contains the code so far and an empty file called FadeState.lua that's been added to the manifest and dependencies. Add the code shown in Listing 2.180 to this file so it can be used to fade in and out.

```
FadeState = {}
FadeState.__index = FadeState
function FadeState:Create(stack, params)
    params = params or {}
    local this =
    {
        mStack = stack,
        mAlphaStart = params.start or 1,
        mAlphaFinish = params.finish or 0,
        mDuration = params.time or 1
    }
    this.mColor = params.color or Vector.Create(0,0,0, this.mAlphaStart)
    this.mTween = Tween:Create(this.mAlphaStart,
                               this.mAlphaFinish,
                               this.mDuration)

    setmetatable(this, self)
    return this
end

function FadeState:Enter() end
function FadeState:Exit() end
function FadeState:HandleInput() end

function FadeState:Update(dt)
    self.mTween:Update(dt)

    local alpha = self.mTween:Value()
    self.mColor:SetW(alpha)

    if self.mTween:IsFinished() then
        self.mStack:Pop()
    end

    return true
end

function FadeState:Render(renderer)
```

```

    renderer:DrawRect2d(
        -System.ScreenWidth()/2,
        System.ScreenHeight()/2,
        System.ScreenWidth()/2,
        -System.ScreenHeight()/2,
        self.mColor)
end

```

Listing 2.180: Adding an extra state. In main.lua.

The FadeState constructor takes in two parameters, the stack and a params table. It requires the stack, so it can pop itself off the top once the fade has finished. The params table is optional and contains data to control the fade. If no params table is passed in, then by default the state fades from black to transparent over one second.

The params table contains up to four parameters: start, finish, time, and color. The color parameter controls the color of the fade. This is usually black but it can be set to any color. The start and finish parameters determine what the alpha value is at the start and end of the fade. To *fade in* we set start to 1 and finish we set it to 0. To *fade out* we reverse this; start is 0 and finish is 1. The time value represents how long the fade takes in seconds.

In the Update function the tween is updated and the alpha component of the color is set to the tween value. At the end of the Update function, there's a check to see if the tween has finished, and if it has it pops the fade state off the stack.

The Render function draws a rectangle the size of the screen using the current color. The remaining state functions are empty.

The state can now be used in the main.lua. Copy the code from Listing 2.181. Pressing the F key causes the screen to fade in from black.

```

function update()
    local dt = GetDeltaTime()
    stack:Update(dt)
    stack:Render(gRenderer)

    if Keyboard.JustPressed(KEY_F) then
        local fade = FadeState>Create(stack, {start = 1, finish = 0})
        stack:Push(fade)
    end
end

```

Listing 2.181: Hooking up the fade in state. In main.lua.

Run the program and press F to see the screen fade for yourself! Try switching up the fade state parameters to get different effects. The full code is available in example menu-3-solution.

Waking Up

Before leaving this section, let's extend menu-3-solution with a few more states and get the briefest taste of a bit of narrative. We'll start with a black screen, then show a textbox saying "Uh...", followed by "My head hurts!" and then "Where am I?" and fade into the game. We can do this by cleverly using the state stack.

Let's begin with an obvious approach as shown in Listing 2.182.

```
stack:Push(FadeState>Create(stack))
stack:PushFit(gRenderer, 0, 0, "Where am I?")
stack:PushFit(gRenderer, -50, 50, "My head hurts!")
stack:PushFit(gRenderer, -100, 100, "Uh...")
```

Listing 2.182: First go implementing Wake Up. In main.lua.

If you run this code everything happens at once, the fade starts, all the textboxes appear, and we end up with the mess shown in Figure 2.90.



Figure 2.90: All our states occurring at once!

To make the sequence work, each state needs to run after the previous state has finished. To do this we're going to add one more state that sits in the stack, blocking all updates below until it reaches the top. Once it reaches the top it pops itself off the stack and lets the next state update. This isn't the approach we'll take when coding a full game but it's a good example of using the stack. The code below shows the updated main.lua in Listing 2.183.

```
local CreateBlock = function(stack)
    return
{
    Enter = function() end,
    Exit = function() end,
    HandleInput = function(self)
        stack:Pop()
    end,
    Render = function() end,
    Update = function(self)
        return false
    end
}
end

local stack = StateStack>Create()
local state = ExploreState>Create(stack, mapDef, Vector.Create(11, 3, 1))
stack:Push(state)
stack:Push(FadeState>Create(stack))
stack:Push(CreateBlock(stack))
stack:PushFit(gRenderer, 0, 0, "Where am I?")
stack:Push(CreateBlock(stack))
stack:PushFit(gRenderer, -50, 50, "My head hurts!")
stack:Push(CreateBlock(stack))
stack:PushFit(gRenderer, -100, 100, "Uh...")

function update()
    local dt = GetDeltaTime()
    stack:Update(dt)
    stack:Render(gRenderer)
end
```

Listing 2.183: Using a blocking state to control the flow between states.

The code above starts with a function called CreateBlock, which returns a table with all the functions required to be a state. Only two of the methods are used: Update, which

returns false to stop updates of any states below, and HandleInput, which immediately calls stack:Pop. HandleInput is only called on the top state. This means if we put the pop here, the block is removed when it reaches the top of the stack. The purpose of the Block state is to stop any states below it being updated until it reaches the top and removes itself. We can then position these blocks on the stack to create a sequence of actions.

We add Block states to the stack in between the textboxes and fade, to make the states trigger one after the other. Example menu-3-wakeup contains the full code. If you run the code you'll see everything working as we first intended. Figure 2.91 shows the program running.

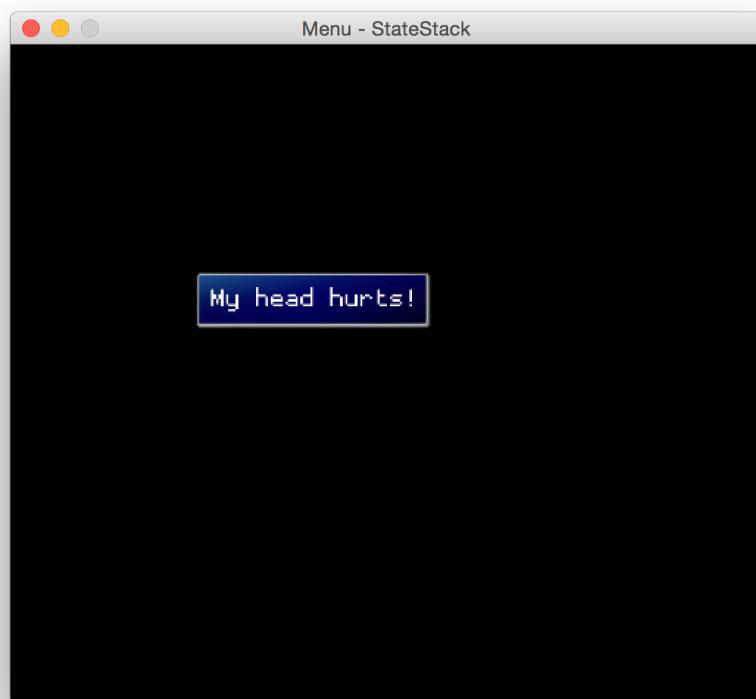


Figure 2.91: The sequence of states to show a character waking up.

With Figure 2.91 we've reached a new milestone; the code base is now mature enough to start telling stories! This is only the first taste. With each section we'll build and refine our tools until we're actually making full games.

An Abstraction to Help Make Menus

RPG games have a lot of menus, and these menus are broken up into many panels. When you come across a programming task that looks like it's going to be repetitive or tedious, it's almost always a wise move to build a tool or try to add an abstraction to make things easier. To help build the menu panels we're going to use an abstraction.

Specifying the exact X, Y coordinates to create all the panels for all the menus sounds like a pain! Luckily as programmers if we find something that's repetitive we can ask the computer to do the work for us. One way to describe the panel layout for a menu is to imagine starting with one big panel and cutting it up into smaller panels. This feels like a more natural way to talk about how menus are constructed.

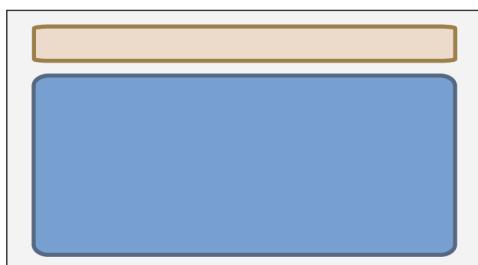
To help us easily make menu panels we're going to write a class called Layout. Figure 2.92 shows how we want to talk about making panels for menus.



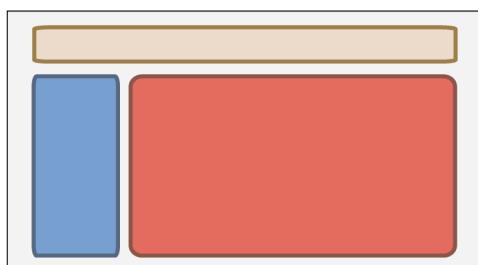
Begin with a
full screen
rectangle



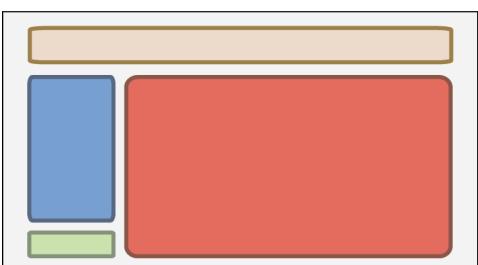
Contract 10%



Split horizontal
at 15%



Bottom box
Split vertical
at 20%



Left box
Split horizontal
at 80%

Figure 2.92: Creating a menu using contract and split operations.

In the Figure 2.92 we make the front menu using only four commands, which is far easier than manually specifying the x, y and width and height for each panel. Here are the operations starting with a panel the size of the display area:

1. Contract the panel.
2. Horizontally cut the panel 15% from the top.
3. Cut the lower panel vertically at 20% from the left.
4. Cut the left panel horizontally at 80% from the top.

We're going to take that English language description and turn it into code to show how we'd like our Layout class to work. Lua is a great language for doing this kind of translation.

1. `layout:Contract('screen', 118, 40)`
2. `layout:SplitHorz('screen', 'upper', 'lower', 0.15)`
3. `layout:SplitVert('lower', 'left', 'right', 0.20)`
4. `layout:SplitHorz('left', 'left_upper', 'left_lower', 0.8)`

This list demonstrates how we'd use the Layout class to construct a menu. Notice that it's quite similar to the English description. The first command `Contract` takes in a string "screen" which is the name of a preexisting panel the size of the screen. The second and third parameters are pixel values telling us how much to subtract from the panel's width and height to shrink it down.

The `SplitHorz` command breaks a panel into two pieces. The first parameter is the name of the panel to split, and in this case we're going to split the "screen" panel. Splitting a panel results in two new smaller panels. The second and third parameters are the names of these panels. The command tells Layout to take the "screen" panel and break it into two pieces called "upper" and "lower". The final argument is where to do the split, and 0.15 indicates 15% from the top. The `SplitVert` command works in the same way as `SplitHorz` but splits the panel vertically.

We now have a good idea of how we'd like to use the Layout class, so let's write the code! In the actual layout code our `Split` functions take an extra argument indicating the *thickness* of the split. The split thickness describes how much spacing is added between child panels.

Example menu-4 contains the code so far and an empty `Layout.lua` file that's been added to the dependencies and manifest files. You can use this or continue with your existing codebase. Copy the constructor code from Listing 2.184 in your `Layout.lua` file.

```

Layout = {}
Layout.__index = Layout
function Layout:Create()
    local this =
    {
        mPanels = {},
        mPanelDef =
        {
            texture = Texture.Find("gradient_panel.png"),
            size = 3,
        }
    }

    -- First panel is the full screen
    this.mPanels['screen'] =
    {
        x = 0,
        y = 0,
        width = System.ScreenWidth(),
        height = System.ScreenHeight(),
    }

    setmetatable(this, self)
    return this
end

```

Listing 2.184: The Layout constructor. In Layout.lua.

The layout constructor, as shown in Listing 2.184, doesn't take in any parameters.

The this table contains mPanels, which is a list of panels, and a definition table, mPanelDef, which is used to create panels. By default we add a single panel, called "screen", the size of the entire display, to the mPanels table.

Entries in the mPanels table contain an x, y, width field, and height field. These variables are easy to manipulate, and once we're happy with them, we use them to create real Panel objects using the function CreatePanel as shown in Listing 2.185.

```

function Layout>CreatePanel(name)
    local layout = self.mPanels[name]
    local panel = Panel>Create(self.mPanelDef)
    panel:CenterPosition(layout.x, layout.y,
                         layout.width, layout.height)
    return panel
end

```

```

function Layout:DebugRender(renderer)

    for k, v in pairs(self.mPanels) do
        local panel = self>CreatePanel(k)
        panel:Render(renderer)
    end
end

```

Listing 2.185: Panel Creation and Debug rendering. In Layout.lua.

The CreatePanel function takes in the name of the panel in `mPanels` and returns a real Panel object. The panel is created using `mPanelDef`, and it's sized and positioned according to its data stored in `mPanels`.

`DebugRender` is a debug function we can use when constructing a layout. It iterates through the layouts, creates a panel for each one, and then renders that panel.

Next let's add the contract and splitting functions as shown in Listing 2.186.

```

function Layout:Contract(name, horz, vert)
    horz = horz or 0
    vert = vert or 0
    local panel = self.mPanels[name]
    assert(panel)

    panel.width = panel.width - horz
    panel.height = panel.height - vert
end

function Layout:SplitHorz(name, tname, bname, x, splitSize)

    local parent = self.mPanels[name]
    self.mPanels[name] = nil

    local p1Height = parent.height * x
    local p2Height = parent.height * (1 - x)
    self.mPanels[tname] =
    {
        x = parent.x,
        y = parent.y + parent.height/2 - p1Height/2 + splitSize/2,
        width = parent.width,
        height = p1Height - splitSize,
    }

    self.mPanels[bname] =
    {

```

```

        x = parent.x,
        y = parent.y - parent.height/2 + p2Height/2 - splitSize/2,
        width = parent.width,
        height = p2Height - splitSize,
    }
end

function Layout:SplitVert(name, lname, rname, y, splitSize)
    local parent = self.mPanels[name]
    self.mPanels[name] = nil

    local p1Width = parent.width * y
    local p2Width = parent.width * (1 - y)
    self.mPanels[rname] =
    {
        x = parent.x + parent.width/2 - p1Width/2 + splitSize/2,
        y = parent.y,
        width = p1Width - splitSize,
        height = parent.height,
    }
    self.mPanels[lname] =
    {
        x = parent.x - parent.width/2 + p2Width/2 - splitSize/2,
        y = parent.y,
        width = p2Width - splitSize,
        height = parent.height,
    }
end

```

Listing 2.186: Functions to contract and split panels. In Layout.lua.

The Contract function takes a horz and a vert argument, which it subtracts from the width and height of the named panel.

The SplitHorz function is a little more involved. Splitting a panel destroys the original panel and creates two new ones. To destroy the parent panel we set its entry in the mPanel table to nil. The split is horizontal so the height is adjusted by the x variable which is a number from 0 to 1. To split a panel in half, the x value would be 0.5 (or 50%), and each panel's height would be reduced by half. The function calculates the new heights for each child. The child panels are added into the mPanels table using tname and bname for the names. The child panels then have their Y coordinates adjusted so that the bottom panel is below the top panel.

The final parameter is splitSize, which represents how much space should separate the two child panels. The splitSize is subtracted from the child panel's height.

When the function finishes, the parent panel has been removed from the `mPanels` table and the two children have been added, correctly sized and positioned.

The `SplitVert` function works in the same way as the `SplitHorz` function but on Y axis instead of the X axis. Let's switch back to `main.lua` and start using some of this code in Listing 2.187.

```
local layout = Layout>Create()

layout:Contract('screen', 118, 40)
layout:SplitHorz('screen', "top", "bottom", 0.12, 2)
layout:SplitVert('bottom', "left", "party", 0.726, 2)
layout:SplitHorz('left', "menu", "gold", 0.7, 2)

function update()
    local dt = GetDeltaTime()
    layout:DebugRender(gRenderer)
end
```

Listing 2.187: Using the layout to create backing panels for a menu. In main.lua.

Run this code and you'll have something like Figure 2.93 which shows all the panels being rendered. Try commenting out the layout commands and then uncommenting them one by one to see how the menu is constructed. Also try changing the positions of the cuts or the order of the commands. The layout code is quite flexible.

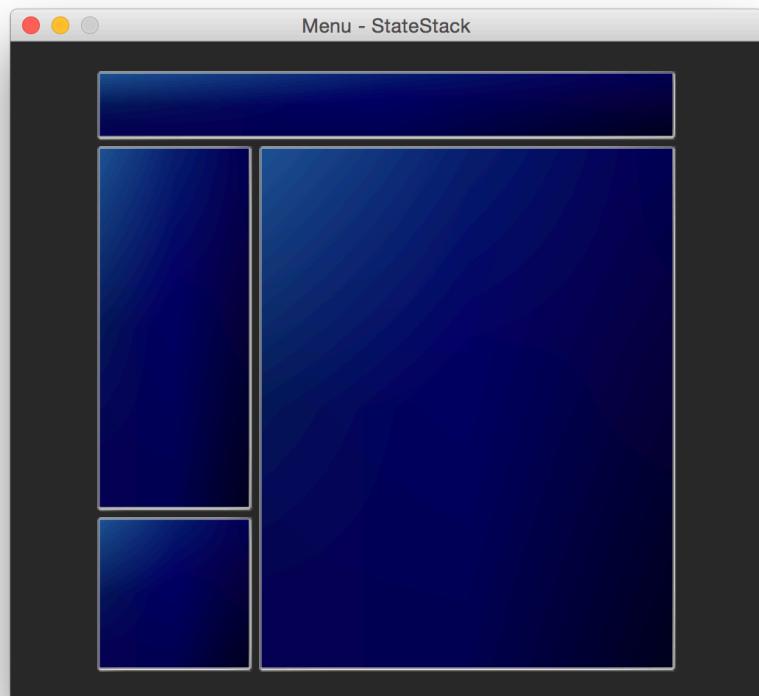


Figure 2.93: A panel cut up into several subpanels.

We're nearly ready to move on from the layout, but there's one last batch of functions to add. These helper functions help us lay out text and images inside the panels. Copy the code from below in Listing 2.188.

```
function Layout:Top(name)
    local panel = self.mPanels[name]
    return panel.y + panel.height / 2
end

function Layout:Bottom(name)
    local panel = self.mPanels[name]
    return panel.y - panel.height / 2
end
```

```

function Layout:Left(name)
    local panel = self.mPanels[name]
    return panel.x - panel.width / 2
end

function Layout:Right(name)
    local panel = self.mPanels[name]
    return panel.x + panel.width / 2
end

function Layout:MidX(name)
    local panel = self.mPanels[name]
    return panel.x
end

function Layout:MidY(name)
    local panel = self.mPanels[name]
    return panel.y
end

```

Listing 2.188: Layout functions for alignment. In Layout.lua.

These functions can be used to get various points on the panel, like the top left or center. Check out Figure 2.94 to see the positions visualized.

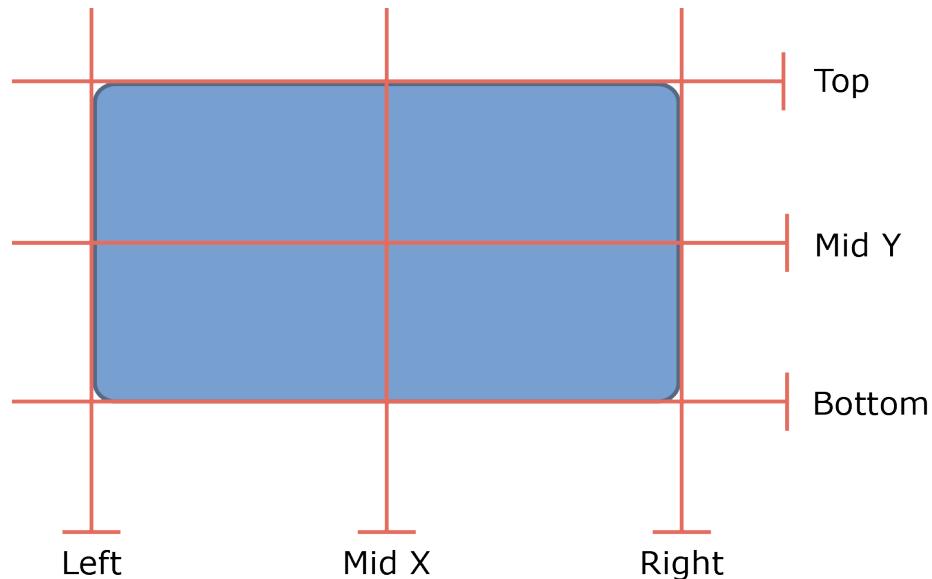


Figure 2.94: Position data on a panel.

That's the final addition to Layout.lua. You can find the code so far in menu-4-solution.

Simple Menu State

We're now ready to create the *in-game menu*. This menu works as a quick heads-up to show pertinent information: amount of gold, play time, brief party status info, and as a directory to dig down into more detailed information.

Example menu-5 contains the code so far. You'll notice that the width and height of the window has been increased in the settings file so its aspect ratio is more regular.

Create a new file called InGameState.lua (example menu-5 already has this included). When the player opens the menu, we'll push InGameState onto the stack, and when they close it we'll pop it off. The InGameState displays some information and allows the player to navigate submenus. We make use of a StateMachine to help handle the submenus.

Copy the code from Listing 2.189.

```
InGameState = {}
InGameState.__index = InGameState
function InGameState:Create(stack)
    local this =
    {
        mStack = stack
    }

    this.mStateMachine = StateMachine:Create
    {
        ["frontmenu"] =
        function()
            return FrontMenuState:Create(this)
        end,
        ["items"] =
        function()
            --return ItemMenuState:Create(this)
            return this.mStateMachine.mEmpty
        end,
        ["magic"] =
        function()
            --return MagicMenuState:Create(this)
            return this.mStateMachine.mEmpty
        end,
        ["equip"] =
        function()
            --return EquipMenuState:Create(this)
        end
    }
end
```

```

        return this.mStateMachine.mEmpty
    end,
    ['status'] =
    function()
        --return StatusMenuState>Create(this)
        return this.mStateMachine.mEmpty
    end
}
this.mStateMachine:Change("frontmenu")

setmetatable(this, self)
return this
end

function InGameMenuState:Update(dt)
    if self.mStack:Top() == self then
        self.mStateMachine:Update(dt)
    end
end

function InGameMenuState:Render(renderer)
    self.mStateMachine:Render(renderer)
end

function InGameMenuState:Enter() end
function InGameMenuState:Exit() end
function InGameMenuState:HandleInput() end

```

Listing 2.189: The InGameMenuState. In InGameMenuState.lua.

Most of the code in Listing 2.189 is concerned with creating and updating a StateMachine object. The start state is set to FrontMenu. The FrontMenu displays a list of submenus and important game information. We're not going to fill out all the menu states at this time and currently most of the states just return the empty state.

In the update loop you can see that the mStateMachine is only updated if the InGameMenuState is at the top of the stack. This stops the menu handling input when it shouldn't.

The Default State of the In-Game Menu

Let's write the first submenu state, FrontMenuState. A finished front menu state might look something like Figure 2.95.



Figure 2.95: A complete front menu picture.

We're only going to create part of the state pictured in Figure 2.95. The menu only contains the "Items" option and we don't show any party members yet. We'll use the Layout class to match the panel layout shown in Figure 2.95.

Create a new file called FrontMenuState.lua and add it to the manifest and dependencies files. Copy the code from Listing 2.190.

```
FrontMenuState = {}
FrontMenuState.__index = FrontMenuState
function FrontMenuState:Create(parent, world)

    local layout = Layout:Create()
    layout:Contract('screen', 118, 40)
    layout:SplitHorz('screen', "top", "bottom", 0.12, 2)
    layout:SplitVert('bottom', "left", "party", 0.726, 2)
    layout:SplitHorz('left', "menu", "gold", 0.7, 2)

    local this
    this =
    {
        mParent = parent,
        mStack = parent.mStack,
        mStateMachine = parent.mStateMachine,
```

```

mLayout = layout,

mSelections = Selection:Create
{
    spacingY = 32,
    data =
    {
        "Items",
        --"Magic",
        --"Equipment",
        --"Status",
        --"Save"
    },
    OnSelection = function(...) this:OnMenuClick(...) end
},

mPanels =
{
    layout>CreatePanel("gold"),
    layout>CreatePanel("top"),
    layout>CreatePanel("party"),
    layout>CreatePanel("menu")
},
mTopBarText = "Current Map Name",
}

setmetatable(this, self)
return this
end

```

Listing 2.190: The constructor for the first screen of the in-game menu. In `FrontMenuState.lua`.

The constructor creates a Layout object and uses it to divide the screen into four panels. One section displays the party's gold and the time played, the top section displays the current map name, the largest section contains a quick summary of each party member, and a sidebar contains the menu options.

The this table stores a reference to some useful values including the parent state, the stack, state machine and layout. The this table also contains a selection menu with only one entry, "Items". The "Items" option lets the player check the inventory. The menu's selection function calls OnMenuClick, a function we haven't yet defined.

The mPanels table stores the backing panels. We use the Layout class to arrange panels and then call CreatePanel to add each backing panel to the mPanels list. The

`mTopBarText` field holds the name of the current map and is displayed at the top of the menu.

Next let's add the Update and Render function as shown in Listing 2.191.

```
function FrontMenuState:Update(dt)
    self.mSelections:HandleInput()

    if Keyboard.JustPressed(KEY_BACKSPACE) or
        Keyboard.JustPressed(KEY_ESCAPE) then
        self.mStack:Pop()
    end
end

function FrontMenuState:Render(renderer)
    for k, v in ipairs(self.mPanels) do
        v:Render(renderer)
    end

    renderer:ScaleText(1.5, 1.5)
    renderer:AlignText("left", "center")
    local menuX = self.mLayout:Left("menu") - 16
    local menuY = self.mLayout:Top("menu") - 24
    self.mSelections:SetPosition(menuX, menuY)
    self.mSelections:Render(renderer)

    local nameX = self.mLayout:MidX("top")
    local nameY = self.mLayout:MidY("top")
    renderer:AlignText("center", "center")
    renderer:DrawText2d(nameX, nameY, self.mTopBarText)

    local goldX = self.mLayout:MidX("gold") - 22
    local goldY = self.mLayout:MidY("gold") + 22

    renderer:ScaleText(1.22, 1.22)
    renderer:AlignText("right", "top")
    renderer:DrawText2d(goldX, goldY, "GP:")
    renderer:DrawText2d(goldX, goldY - 25, "TIME:")
    renderer:AlignText("left", "top")
    renderer:DrawText2d(goldX + 10, goldY, "0")
    renderer:DrawText2d(goldX + 10, goldY - 25, "0")

end
```

Listing 2.191: The Update and Render functions for the FrontMenuState. In `FrontMenuState.lua`.

In Listing 2.191 the Update function handles the input for the menu and checks if the backspace or escape key has been pressed. If one of the keys is pressed, it pops itself off the stack and returns the player to the map exploration state.

The Render function iterates through the list of panels and renders them all out. This forms the backdrop for the menu.

The first UI element rendered is the menu, `mSelections`. The `x` and `y` position are calculated by adding a little padding to the top left of the menu panel. The menu is then set to this position and rendered.

Next the `mTopBarText` is rendered, then the gold and time in the gold panel. The gold and time both draw a zero for their values as we aren't currently tracking this information. If you read the code carefully you'll see the "gold" and "time" labels are right aligned and the values are left aligned. Aligning the text in this way keeps the labels nicely lined up even if we change the gold text, say to another language, or the gold value, as more gold is collected while playing the game.

In Listing 2.192 we add the empty Enter and Exit functions as well as the callback for the menu selection, `OnMenuClick`.

```
function FrontMenuState:OnMenuClick(index)

    local ITEMS = 1

    if index == ITEMS then
        return self.mStateMachine:Change("items")
    end
end

function FrontMenuState:Enter() end
function FrontMenuState:Exit() end
```

Listing 2.192: The selection menu callback for the `FrontMenuState` and the Enter and Exit functions. In `FrontMenuState.lua`.

The `OnMenuClick` callback changes to the "items" state when the first entry in the selection menu is chosen. The item state doesn't exist yet, so if the player chooses it the program crashes! Before fixing it let's try the code out! Update your main file to match the code in Listing 2.192.

```
local mapDef = CreateMap1()
mapDef.on_wake = {}
mapDef.actions = {}
mapDef.trigger_types = {}
mapDef.triggers = {}
local stack = StateStack>Create()
```

```

local explore = ExploreState:Create(stack, mapDef, Vector.Create(11, 3, 1))
local menu = InGameMenuState:Create(stack)

stack:Push(explore)
stack:Push(menu)

function update()
    local dt = GetDeltaTime()
    stack:Update(dt)
    stack:Render(gRenderer)
end

```

Listing 2.193: Rendering the in game menu. In main.lua.

In Listing 2.192 we create the map, load up the ExploreState, push it onto the stack, and then create the InGameMenuState and push it onto the stack too. The update loop renders and updates the stack.

Run the code and you'll see something like Figure 2.96. Because the menu state is on the top of the stack, it's the last thing rendered when running the game.

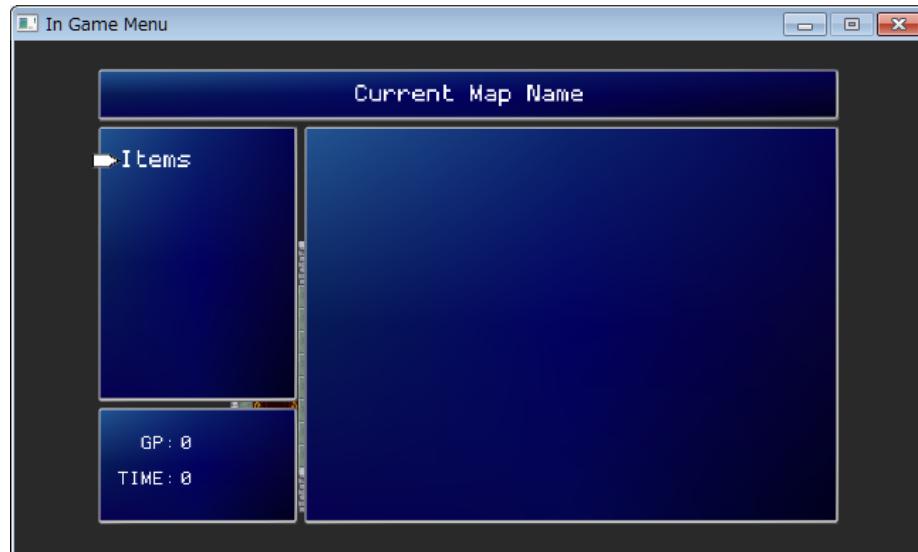


Figure 2.96: A basic in game menu.

Run this code and you'll see the menu is already on the top of the stack. Press Backspace to remove menu and return to the ExploreState. But if we try to get back to the menu, there's currently no way to do it! We need to modify the ExploreState to allow the

player to bring up the menu at any time. The code below in Listing 2.194 shows how this can be done.

```
function ExploreState:HandleInput()

    -- code omitted

    if Keyboard.JustPressed(KEY_LALT) then
        local menu = InGameMenuState:Create(self.mStack)
        return self.mStack:Push(menu)
    end

end
```

Listing 2.194: Adding code to go from the ExploreState to the InGameMenuState. In ExploreState.lua.

Run the updated code. You can now exit the menu by pressing the backspace key, and return to it by pressing the left alt key.

Example menu-5-solution contains the complete code from this section.

Creating the World

In the menu we display gold and time information but currently we're not tracking it! The values are set to zero. In this section we'll start tracking gold, time, and the party inventory.

Before we can make an inventory we need items! We're going store all the item definitions in one database that's easy to extend. Example menu-6 has the code so far and two new files, World.lua and ItemDB.lua. These have been added to the manifest and dependencies files.

Items

Items are anything in the world the player can pick up. Examples include keys, swords and health potions. All items have a few things in common. They have a name and a description. For example, a key might be called "Rusty Key" and the description might be "A small rusted key. What could it open?" For our purposes items fall into the following categories:

- **Key Item** - a special item, usually required to progress past a certain part of the game.
- **Weapon** - an item that can be equipped and has properties related to inflicting damage.

- **Armor** - an item that can be equipped and has properties related to reducing damage.
- **Accessory** - an item that can be equipped and has special properties.
- **Usable** - an item that can be used during combat or on the world map.

All items are represented as Lua tables. Each table has a name, description and type variable. Certain items may then contain additional data depending on their type. At this point we're going to consider some basic stats that armor, weapons and accessories can have.

Here are the player stats that an item might alter.

- **Strength** - amount this item contributes to the character's strength.
- **Speed** - amount this item contributes to the character's speed.
- **Intelligence** - amount this item contributes to the character's intelligence.
- **Attack** - contributes to damage inflicted.
- **Defense** - amount of damage deflected.
- **Magic** - amount magic attacks are bolstered.
- **Resist** - amount of magic resisted.

These are stats I have chosen. You can choose more, or less. Equipping items may modify these stat numbers.

ItemDB.lua shows how we're going to define the items in Listing 2.195.

```
ItemDB =
{
  [-1] =
  {
    name = "",
    description = "",
    special = "",
    stats =
    {
      strength = 0,
      speed = 0,
      intelligence = 0,
      attack = 0,
      defense = 0,
      magic = 0,
      resist = 0
    }
  },
  {
    name = "Mysterious Torque",
    description = "A mysterious torque that grants the wielder increased strength and agility.",
    special = "When activated, the torque grants temporary bonuses to Strength and Speed for a short duration.",
    stats =
    {
      strength = 10,
      speed = 5,
      intelligence = 0,
      attack = 0,
      defense = 0,
      magic = 0,
      resist = 0
    }
  }
}
```

```

        type = "accessory",
        description = "A golden torque that glitters.",
        stats =
        {
            strength = 10,
            speed = 10
        }
    },
{
    name = "Heal Potion",
    type = "useable",
    description = "Heals a little HP."
},
{
    name = "Bronze Sword",
    type = "weapon",
    description = "A short sword with dull blade.",
    stats =
    {
        attack = 10
    }
},
{
    name = "Old bone",
    type = "key",
    description = "A calcified human femur"
},
}

```

Listing 2.195: The Item Database. In ItemDB.lua.

The item database shown in Listing 2.195 has five items. The first item is probably the most interesting. It's an empty item at position -1 in the table. The *empty item* is a *nil item* used to represent the absence of an item. The empty item is used often so let's make a global reference to it, `EmptyItem`. It's defined beneath the database code, as shown in Listing 2.196.

```
EmptyItem = ItemDB[-1]
```

Listing 2.196: Creating a global EmptyItem variable. In ItemDB.lua.

After the empty item, there's an entry in the database for a "Mysterious Torque" an accessory. When the torque is equipped it adds 10 to both strength and speed.

This torque item is inserted into the table directly, which means it has an index of 1. These index numbers uniquely identify each item in the game.

The next entry is a useable item, then a weapon, and finally a key item. You may notice that the stats table for the sword and torque doesn't define every stat; this makes it easier to write the item definitions. However, the code needs know all the stats for each item. For instance, the code must know that the resist stat for both the torque and sword is zero. After the table is defined let's add code that fills in any missing stats with their default values. Copy the code from Listing 2.197 and add it to the bottom of the file.

```
local function DoesItemHaveStats(item)
    return item.type == "weapon" or
           item.type == "accessory" or
           item.type == "armor"
end

-- 
-- If any stat is missing add it and set it to
-- the value in EmptyItem
--

for k, v in ipairs(ItemDB) do
    if DoesItemHaveStats(v) then
        v.stats = v.stats or {}
        local stats = v.stats
        for k, v in ipairs(EmptyItem) do
            stats[key] = stats[key] or v.stats
        end
    end
end
```

Listing 2.197: Adding any missing stats to equippable items. In ItemDB.lua.

The code in Listing 2.197 iterates through all the items in the database and checks if each item has stats. We assume all equippable items have a stats table with a value for each stat. For each equippable item we iterate through the EmptyItem stats, check the current item, and add in any missing stats using the EmptyItem value. The great thing about doing it this way is that if we want to add a new stat, we just need to add it to the EmptyItem and it will get automatically filled in for every other item.

The World

The world object tracks all the items the party picks up, the amount of gold they have, and the amount of time that has passed. Later on it will also track quests and current party members, and be involved with loading and saving.

Let's start with the world constructor in Listing 2.198.

```
World = {}
World.__index = World
function World:Create()
    local this =
    {
        mTime = 0,
        mGold = 0,
        mItems =
        {
            { id = 3, count = 1 },
        },
        mKeyItems = {},
    }
    setmetatable(this, self)
    return this
end
```

Listing 2.198: The World constructor. In World.lua.

The constructor contains an `mTime` field to record the play time in seconds. The `mGold` field records the number of gold coins the party has found. The `mKeyItems` table contains items related to quests and the `mItems` table contains all other items. The items table has a single item in it made up of two fields, an `id` and a `count`. The `id` is an index into the ItemDB and in this case it's the *Bronze Sword*. The `count` field is the number of Bronze Swords we have, which is 1.

The world keeps track of time so it needs a update function. Let's add that next, as well as two helper functions for translating the gold and time into strings. Copy the code from Listing 2.199.

```
function World:Update(dt)
    self.mTime = self.mTime + dt
end

function World:TimeAsString()

    local time = self.mTime
    local hours = math.floor(time / 3600)
    local minutes = math.floor((time % 3600) / 60)
    local seconds = time % 60

    return string.format("%02d:%02d:%02d", hours, minutes, seconds)
end
```

```

function World:GoldAsString()
    return string.format("%d", self.mGold)
end

```

Listing 2.199: Update, TimeAsString and GoldAsString functions. In World.lua.

In Listing 2.199 the Update function keeps track of the play time by incrementing `mTime` by the time taken to render the last frame.

The `TimeAsString` returns the play time in a nicely formatted string. It uses the time in seconds, stored in the `mTime` variable, to calculate the number of hours, minutes and seconds that have passed. It formats these numbers and returns them as a string.

The `GoldAsString` function does much the same thing. It formats the gold value as a string. If you're going to print numbers that get very big, it's a good idea to escape them through `string.format` to avoid outputting the number in scientific notation as shown in Listing 2.200.

```

local largeNumber = 999999999999999
print(largeNumber) -- 1e+15
print(string.format("\%d", largeNumber)) -- 999999999999999

```

Listing 2.200: Formatting numbers for display on screen.

It's unlikely we'll allow the amount of gold to ever get this high, but if you're adding a score or similar large value to a game, it's worth knowing about.

Each item in the party inventory has a count field. When we pick up a new item we check if there's already an item of the same type in the inventory. If the item is already present we increase the count but if not then we add it and set the count to 1. Obviously this is a non-trivial operation and therefore it's a good idea to write some functions to help us. These helpers are shown in Listing 2.201.

```

function World:AddItem(itemId, count)

    count = count or 1

    assert(ItemDB[itemId].type ~= "key")

    -- 1. Does it already exist?
    for k, v in ipairs(self.mItems) do
        if v.id == itemId then
            -- 2. Yes, it does. Increment and exit.
            v.count = v.count + count
            return
        end
    end

```

```

        end
    end

    -- 3. No it does not exist.
    --     Add it as a new item.
    table.insert(self.mItems,
    {
        id = itemId,
        count = count
    })
end

function World:RemoveItem(itemId, amount)

    assert(ItemDB[itemId].type ~= "key")
    amount = amount or 1

    for i = #self.mItems, 1, -1 do
        local v = self.mItems[i]
        if v.id == itemId then
            v.count = v.count - amount
            assert(v.count >= 0) -- this should never happen
            if v.count == 0 then
                table.remove(self.mItems, i)
            end
            return
        end
    end

    assert(false) -- shouldn't ever get here!
end

```

Listing 2.201: Add and remove functions for the inventory.

These functions in Listing 2.201 work for normal, everyday items but the key items have their own functions. You can never have more than one type of key item.

The `AddItem` function iterates through the inventory searching for the item we're trying to add. If it finds the item, then it increases the count and exits. If the item cannot be found, then a new item entry is added with the passed in count. If no count is passed in it defaults to 1.

The `RemoveItem` function takes in an item id and optional amount to remove. By default the amount to remove is 1. It iterates through the table until it finds the item and reduces the count by the amount to remove. If the count equals 0 then the item is totally removed from the `mItems` table and the function returns.

Let's add similar functions to handle key items. The `mKeyItems` list is an array of tables with an id but no count information.

```
function World:HasKey(itemId)
    for k, v in ipairs(self.mKeyItems) do
        if v.id == itemId then
            return true
        end
    end
    return false
end

function World:AddKey(itemId)
    assert(not self:HasKey(itemId))
    table.insert(self.mKeyItems, {id = itemId})
end

function World:RemoveKey(itemId)
    for i = #self.mKeyItems, 1, -1 do
        local v = self.mKeyItems[i]

        if v.id == itemId then
            table.remove(self.mKeyItems, i)
            return
        end
    end
    assert(false) -- should never get here.
end
```

Listing 2.202: Adding, Removing and Checking Key Items. In World.lua.

During the game there are times when we want to check if the player has a key item. For example, say there's rubble blocking a mountain pass. If you have dynamite you can pass but otherwise you're out of luck. The `HasKey` does this check. It loops through the key items and returns true if it finds an item of the same id.

The `AddKey` function inserts the new item directly into the table. The `RemoveKey` function searches the list and removes the item when found. With those additions we're ready to test the inventory out!

Testing the Inventory

The `World` class now tracks gold, time, items and key items. We're going to update the `main.lua` to display this information and allow basic interaction with the inventory. To do this we'll make good use of the `Selection` class.

We'll start by creating the world, then create selection menus for the key item and regular items. The selection menu uses the item and key item tables directly. This means no data is copied and everything stays in sync.

The item tables entries are tables containing an id. We need to transform the ids into names for drawing to the screen. To do this we'll write a custom RenderItem function for each menu. Copy the code from Listing 2.203 into your main file.

```
LoadLibrary('Asset')
Asset.Run('Dependencies.lua')

gRenderer = Renderer.Create()
gWorld = World>Create()

local itemList = Selection>Create
{
    data = gWorld.mItems,
    spacingY = 32,
    rows = 5,
    renderItem = function(menu, renderer, x, y, item)
        if item then
            local itemDef = ItemDB[item.id]
            local label = string.format("\%s (\%d)",
                                         itemDef.name,
                                         item.count)
            renderer:DrawText2d(x, y, label)
        else
            renderer:DrawText2d(x, y, "--")
        end
    end
}

local keyItemList = Selection>Create
{
    data = gWorld.mKeyItems,
    spacingY = 32,
    rows = 5,
    renderItem = function(menu, renderer, x, y, item)
        if item then
            local itemDef = ItemDB[item.id]
            renderer:DrawText2d(x, y, itemDef.name )
        else
            renderer:DrawText2d(x, y, "--")
        end
    end
}
```

```
keyItemList:HideCursor()
```

Listing 2.203: Creating a list to show the items. In main.lua.

In Listing 2.203 we define both selection menus for each inventory. Each of the selection menus has a custom item render function that's defined inline. The regular item render function draws the name of the item and a number showing how many of those items you own. The key item render function just renders the name as there's no count information for key items.

If either RenderItem is passed a nil value it renders a double dash “–”. We can only use one menu at once, so we hide the cursor for the keyItemMenuList and we'll only send input to the itemList selection.

Copy the update loop from Listing 2.204 to render the two selection menus.

```
function update()
    local dt = GetDeltaTime()
    gWorld:Update(dt)

    local x = -200
    local y = 50
    gRenderer:AlignText("center", "center")
    gRenderer:DrawText2d(x + itemList:GetWidth()/2, y + 32, "ITEMS")
    gRenderer:AlignText("left", "center")
    itemList:SetPosition(x, y)
    itemList:Render(gRenderer)
    itemList:HandleInput()

    x = 100

    gRenderer:AlignText("center", "center")
    gRenderer:DrawText2d(x + itemList:GetWidth()/2, y + 32, "KEY ITEMS")
    keyItemList:SetPosition(x, y)
    keyItemList:Render(gRenderer)

    local timeText = string.format("TIME:\%s", gWorld:TimeAsString())
    local goldText = string.format("GOLD:\%s", gWorld:GoldAsString())
    gRenderer:DrawText2d(0, 150, timeText)
    gRenderer:DrawText2d(0, 120, goldText)

    if Keyboard.JustPressed(KEY_A) then
        gWorld:AddItem(1)
    end
```

```

if Keyboard.JustPressed(KEY_R) then
    local item = itemList:SelectedItem()
    if item then
        gWorld:RemoveItem(item.id)
    end
end

if Keyboard.JustPressed(KEY_K) then
    if not gWorld:HasKey(4) then
        gWorld:AddKey(4)
    end
end

if Keyboard.JustPressed(KEY_U) then
    if gWorld:HasKey(4) then
        gWorld:RemoveKey(4)
    end
end

if Keyboard.JustPressed(KEY_G) then
    gWorld.mGold = gWorld.mGold + math.random(100)
end

local tip = "A - Add Item, R - Remove Item, " ..
            "K - Add Key, U - Use Key, G - Add Gold"
gRenderer:DrawText2d(0, -150, tip, 300)
end

```

Listing 2.204: The Update loop demoing the new World features. In main.lua.

The code in Listing 2.204 renders the `itemList` on the left of the screen and the `keyItemList` on the right. Then the gold and time values are both rendered at the top of the screen. When you run the program you'll be able to see the time tick up.

It isn't a good test program unless we can interact with the item lists. At the end of the update function we for check keyboard input and use this to interact with the list. The A key adds the "Torque" item to the inventory (try adding more than one). The R key removes whatever item is currently selected in the selection menu. The K key adds item 4 which is the "Old Bone" key item. The U key emulates using the bone key item and then removes it.

The code has guards to prevent a key item from being added twice or removed when it doesn't exist. The G key adds a random amount between 0 and 100 coins to the gold amount. At the bottom of the screen the instructions for the keys that can be pressed are listed.

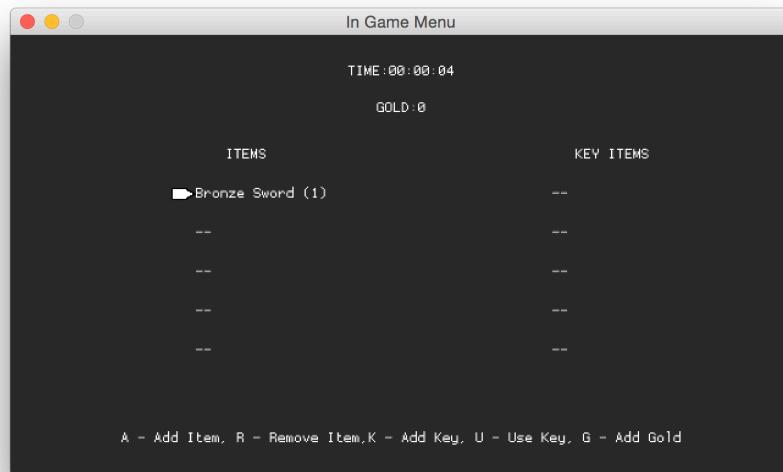


Figure 2.97: Screenshot of the new World features being demonstrated.

This code is available as menu-6-solution. Run it. You'll be presented with something like Figure 2.97.

The Item Menu

Now that the inventory data is stored in the World class we're ready to make the inventory menu state. This state displays the inventory and allows the user to browse the items. The menu state looks like the image shown below in Figure 2.98.

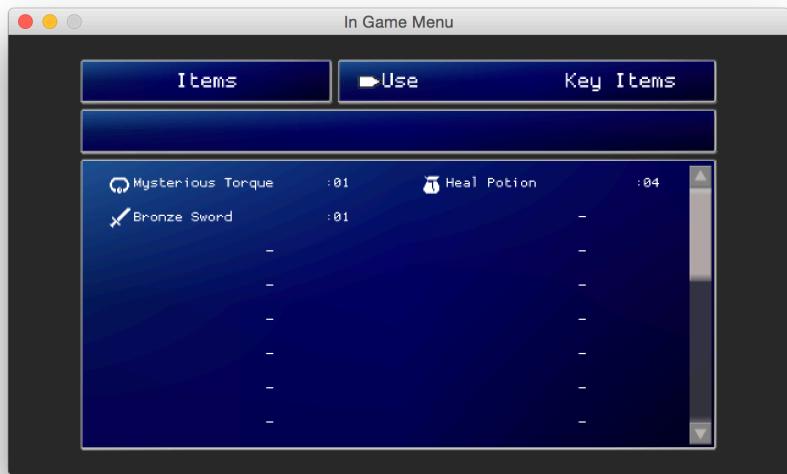


Figure 2.98: The finished item menu.

The top left panel contains the menu title, "Items". To the left of the title are the words "Use" and "Key Items". This is a selection menu that lets the player switch between the normal item inventory and the key items one. The panel below the title and selection menu contains description text. When an item is selected, its description is written on this panel.

The largest panel at the bottom lists the inventory items. By default this panel displays all the regular items but if we flick to the "Key Items" menu then the key items are displayed.

The screenshot in Figure 2.98 uses a scrollbar to show the progress through the inventory list.

The selection menu for the screenshot is using a custom render function. Each entry in the menu has a small icon representing the item type, the item's name, and the number of items in the inventory. Small icons are used extensively in RPGs to indicate item information, status effects, levelling systems, UI arrows and so on. We'll build an Icons class to store and draw icons.

Icons

Example menu-7 contains the latest version of the code and two new Lua files, Icons.lua and ItemMenuState.lua. It also contains a new texture, inventory_icons.png, shown in Figure 2.99, which contains all our icons.



Figure 2.99: The texture storing all the inventory icons.

The icon texture is 180x180 pixels and each icon is 18x18 pixels. Therefore the texture sheet can contain up to 100 icons. That's more than enough for our needs! What we want to do is take this texture, split it up into 18 x 18 pixels chunks, and create a sprite for each icon we want to use. Each icon is given a simple id. The code is shown below in Listing 2.205.

```
Icons = {}
Icons.__index = Icons
function Icons:Create(texture)
    local this =
    {
        mTexture = texture,
        mUVs = {},
        mSprites = {},
        mIconDefs =
        {
            useable = 1,
            accessory = 2,
            weapon = 3,
            armor = 4,
            uparrow = 5,
            downarrow = 6
        }
    }
    this.mUVs = GenerateUVs(18, 18, this.mTexture)
```

```

for k, v in pairs(this.mIconDefs) do
    local sprite = Sprite.Create()
    sprite:SetTexture(this.mTexture)
    sprite:SetUVs(unpack(this.mUVs[v]))
    this.mSprites[k] = sprite
end

setmetatable(this, self)
return this
end

function Icons:Get(id)
    return self.mSprites[id]
end

```

Listing 2.205: A helpful class to store icons. In Icons.lua.

The interesting part of the Icon class is the `mIconDefs` table. It's a list of name keys and number values. The numbers refer to the position of the sprite in the texture: 1 is the first icon in the top left corner of the texture, id 2 is the one to the right of that, and so on.

After the `this` table is defined, we loop through the `mIconDefs` table and create a sprite for each icon. These sprites can be retrieved later by using the `Get` function.

We'll use the icon sprites to help create the `ItemMenuState`.

The Item Menu State

Update `InGameMenuState.lua` file as shown in Listing 2.206. These changes will let the user navigate to the `ItemMenuState` from the `FrontMenuState`.

```

["items"] =
function()
    return ItemMenuState:Create(this) -- add this transition.
end,

```

Listing 2.206: Updating the statemachine to move to the item menu. In InGameMenuState.lua.

Now we can write the `ItemMenuState` class. We'll begin with the constructor. Copy the code from Listing 2.207. In the constructor we'll start by laying out the backing panels.

```

ItemMenuState = {}

ItemMenuState.__index = ItemMenuState
function ItemMenuState:Create(parent)

    local layout = Layout>Create()
    layout:Contract('screen', 118, 40)
    layout:SplitHorz('screen', "top", "bottom", 0.12, 2)
    layout:SplitVert('top', "title", "category", 0.6, 2)
    layout:SplitHorz('bottom', "mid", "inv", 0.14, 2)

    local this
    this =
    {
        mParent = parent,
        mStack = parent.mStack,
        mStateMachine = parent.mStateMachine,

        mLayout = layout,
        mPanels =
        {
            layout>CreatePanel("title"),
            layout>CreatePanel("category"),
            layout>CreatePanel("mid"),
            layout>CreatePanel("inv"),
        },
        mScrollbar = Scrollbar>Create(Texture.Find("scrollbar.png"), 228),
    }
end

```

Listing 2.207: The start of ItemMenuState. In ItemMenuState.lua.

The layout code is similar to the FrontMenuState but customized for the item menu's needs. Four panels are created: a title panel, a panel for the categories of items, a panel to display the item description, and finally the largest panel to allow the items to be browsed. If you look at the layout calls you can see how this is constructed step by step.

In the this table the usual variables are stored: the parent state, stack, state machine, and the layout we just created. The mPanels table is filled with the panels we created using Layout class. At the end of this snippet we create a scrollbar to display progress through the item list.

Let's finish off the constructor. Copy the code from Listing 2.208. We're going to add three selection menus: one for each inventory and one to switch between the inventories.

```

mItemMenus =
{
    Selection:Create
    {
        data = gWorld.mItems,
        spacingX = 256,
        columns = 2,
        displayRows = 8,
        spacingY = 28,
        rows = 20,
        RenderItem = function(self, renderer, x, y, item)
            gWorld:DrawItem(self, renderer, x, y, item)
        end
    },
    Selection:Create
    {
        data = gWorld.mKeyItems,
        spacingX = 256,
        columns = 2,
        displayRows = 8,
        spacingY = 28,
        rows = 20,
        RenderItem = function(self, renderer, x, y, item)
            gWorld:DrawKey(self, renderer, x, y, item)
        end
    },
},
mCategoryMenu = Selection:Create
{
    data = {"Use", "Key Items"},
    OnSelection = function(...) this:OnCategorySelect(...) end,
    spacingX = 150,
    columns = 2,
    rows = 1,
},
mInCategoryMenu = true
}

for k, v in ipairs(this.mItemMenus) do
    v:HideCursor()
end

setmetatable(this, self)
return this

```

```
end
```

Listing 2.208: The end of the ItemMenuState constructor. In ItemMenuState.lua.

The code above in Listing 2.208 introduces a table called `mItemMenus` that contains two selection menus. The first selection menu is for the normal items and the second is for the key items. Both of these selection menus have the similar parameters. The data sources are `gWorld.mItems` and `gWorld.mKeyItems` respectively. The `RenderItem` function differs for each menu. The code responsible for rendering inventory items is delegated to the `World` class.

We can't use items from this menu because neither the key item menu or normal item selection menu implements an `OnSelection` function.

The next entry in the `this` table is `mCategoryMenu`. It's a little different from the selection menus that we've seen so far. It's a *vertical* selection menu, rather than a horizontal one. Vertical menus are created by setting the number of columns to equal the number of items in the data table and setting the `rows` field to 1.

The `this` table contains an entry named `mInCategoryMenu`. This is a boolean that controls which selection menu the player interacts with. Changing the flag moves the focus from the category menu at the top to the item menus at the bottom.

After defining the `this` table there's a for loop that hides the cursor for the two bottom selection menus, and that's the end of the `ItemMenuState` constructor.

At this point we'll quickly jump back into `World.lua` and add the `DrawItem` and `DrawKey` functions used by the selection menus. We're defining these in the `World` class because they may be used by several different menus. The code is shown in Listing 2.209.

```
function World:DrawKey(menu, renderer, x, y, item)
    if item then
        local itemDef = ItemDB[item.id]
        renderer:AlignText("left", "center")
        renderer:DrawText2d(x, y, itemDef.name)
    else
        renderer:AlignText("center", "center")
        renderer:DrawText2d(x + menu.mSpacingX/2, y, " - ")
    end
end

function World:DrawItem(menu, renderer, x, y, item)
    if item then
        local itemDef = ItemDB[item.id]
        local iconSprite = gIcons:Get(itemDef.type)
        if iconSprite then
            iconSprite:SetPosition(x + 6, y)
```

```

        renderer:DrawSprite(iconSprite)
    end
    renderer:AlignText("left", "center")
    renderer:DrawText2d(x + 18, y, itemDef.name)
    local right = x + menu.mSpacingX - 64
    renderer:AlignText("right", "center")
    renderer:DrawText2d(right, y, string.format(":%02d", item.count))
else
    renderer:AlignText("center", "center")
    renderer:DrawText2d(x + menu.mSpacingX/2, y, " - ")
end
end

```

Listing 2.209: Functions to draw item entries. In World.lua.

The DrawKey function is responsible for rendering out a single key item. It can be passed a nil, so it first checks if the item exists and then looks up the name in the ItemDB table before drawing it out. If the item doesn't exist, it draws a double dash in the center of the column.

The DrawItem function is a little more involved. If an item exists, it gets the name, the amount carried, and an icon based on the item type. These are all rendered on a single line, the icon, followed by the name and then at the end of the column the amount carried. Like the DrawKey function, if there's no item it draws a double dash in the center of the column.

Let's return to the ItemMenuState class and finish it off. We'll add the Enter and Exit functions, which are empty, and OnCategorySelect, which isn't. If you look at the mCategoryMenu constructor, you'll see its OnSelection callback calls OnCategorySelect. This function gets called when the user chooses a category. Copy the callback function from Listing 2.210.

```

function ItemMenuState:Enter() end
function ItemMenuState:Exit() end

function ItemMenuState:OnCategorySelect(index, value)
    self.mCategoryMenu:HideCursor()
    self.mInCategoryMenu = false
    local menu = self.mItemMenus[index]
    menu:ShowCursor()
end

```

Listing 2.210: The callback for selecting a category in the Item State. In ItemMenuState.lua.

OnCategorySelect takes two parameters, index and value. The index is the index of the item selected. The value is the value of the item at that index; in this case it can only be "Use" or "Key Items". We'll ignore the value and use the index to decide what to do.

An index of 1 is "Use" and 2 is "Key Items". Look at the mItemMenus table and you'll notice that the first entry is the *items list* and the second entry is the *key items list*. The index maps directly to one of the two item menus. The index of the mCategoryMenu determines which of the two item menus gets rendered in the bottom panel.

In the OnCategorySelect function we hide the category menu's cursor because the category menu is losing focus and the menu in the bottom panel is gaining focus. We set the mInCategoryMenu flag to false to indicate that the item menus have focus. The mInCategoryMenu determines which of the menus has its input handled and what the backspace key does. Finally we look up the active menu and set its cursor to visible.

It seems like there's a lot going on in this callback. If we take a look at the Render function in Listing 2.211 things should become clearer.

```
function ItemMenuState:Render(renderer)
    for k,v in ipairs(self.mPanels) do
        v:Render(renderer)
    end

    local titleX = self.mLayout:MidX("title")
    local titleY = self.mLayout:MidY("title")
    renderer:ScaleText(1.5, 1.5)
    renderer:AlignText("center", "center")
    renderer:DrawText2d(titleX, titleY, "Items")

    renderer:AlignText("left", "center")
    local categoryX = self.mLayout:Left("category") + 5
    local categoryY = self.mLayout:MidY("category")
    renderer:ScaleText(1.5, 1.5)
    self.mCategoryMenu:Render(renderer)
    self.mCategoryMenu:SetPosition(categoryX, categoryY)

    local descX = self.mLayout:Left("mid") + 10
    local descY = self.mLayout:MidY("mid")
    renderer:ScaleText(1, 1)

    local menu = self.mItemMenus[self.mCategoryMenu:GetIndex()]

    if not self.mInCategoryMenu then
        local description = ""
        local selectedItem = menu:SelectedItem()
        if selectedItem then
```

```

        local itemDef = ItemDB[selectedItem.id]
        description = itemDef.description
    end
    renderer:DrawText2d(descX, descY, description)
end

local itemX = self.mLayout:Left("inv") - 6
local itemY = self.mLayout:Top("inv") - 20

menu:SetPosition(itemX, itemY)
menu:Render(renderer)

local scrollX = self.mLayout:Right("inv") - 14
local scrollY = self.mLayout:MidY("inv")
self.mScrollbar:SetPosition(scrollX, scrollY)
self.mScrollbar:Render(renderer)
end

```

Listing 2.211: The render function for the ItemMenuState. In ItemMenuState.lua.

The Render function draws out all the backing panels. Then it draws the title centered in the title box. The category menu is drawn from the left of the category panel with a little padding.

The next lines get the position for the description; the center left of the mid panel. At this point things become more interesting. In order to render the description text we need to know which menu has focus, the normal items or the key items? This is done by indexing the self.mItemMenus table with the category menu's index. After we have the menu we can work out which item is selected and get its description by looking up its id in the ItemDB table.

After the description, we draw the inventory menu from the top left of the inv panel with a little padding. Finally the scrollbar is positioned to the left of this same panel and rendered.

This leaves us with the last two functions, Update and FocusOnCategoryMenu. FocusOnCategoryMenu moves the focus from the item menus at the bottom to the category menu at the top. The code is shown below in Listing 2.212.

```

function ItemMenuState:Update(dt)

    local menu = self.mItemMenus[self.mCategoryMenu:GetIndex()]

    if self.mInCategoryMenu then
        if Keyboard.JustReleased(KEY_BACKSPACE) or
            Keyboard.JustReleased(KEY_ESCAPE) then

```

```

        self.mStateMachine:Change("frontmenu")
    end
    self.mCategoryMenu:HandleInput()
else

    if Keyboard.JustReleased(KEY_BACKSPACE) or
    Keyboard.JustReleased(KEY_ESCAPE) then
        self:FocusOnCategoryMenu()
    end

    menu:HandleInput()
end

local scrolled = menu:PercentageScrolled()
self.mScrollbar:SetScrollCaretScale(menu:PercentageShown())
self.mScrollbar:SetNormalValue(scrolled)
end

function ItemMenuState:FocusOnCategoryMenu()
    self.mInCategoryMenu = true
    local menu = self.mItemMenus[self.mCategoryMenu:GetIndex()]
    menu:HideCursor()
    self.mCategoryMenu>ShowCursor()
end

```

Listing 2.212: The Update and FocusOnCategoryMenu functions. In ItemMenuState.lua.

The Update function in Listing 2.212 checks which of the three menus we're in. If we're in the category menu then input is passed to the category selection menu. Otherwise we look up the active item menu and tell it to handle its input.

Pressing backspace from the category menu returns us to the FrontMenuState. Pressing backspace when we're in one of the item menus calls FocusOnCategoryMenu and moves focus back to the category menu.

At the end of the of the Update function the scrollbar is updated and positioned according to the active menu.

The FocusOnCategoryMenu function flips the mInCategoryMenu boolean to true, hides the item menu cursor, and shows the cursor for the category menu.

That's it for the ItemMenuState. The item menu can now be explored. Copy the code from Listing 2.213 into the main.lua.

```

LoadLibrary('Asset')
Asset.Run('Dependencies.lua')

gRenderer = Renderer.Create()

local mapDef = CreateMap1()
mapDef.on_wake = {}
mapDef.actions = {}
mapDef.trigger_types = {}
mapDef.triggers = {}
local stack = StateStack>Create()
local explore = ExploreState>Create(stack, mapDef, Vector.Create(11, 3, 1))
local menu = InGameMenuState>Create(stack)

gWorld = World>Create()
gIcons = Icons>Create(Texture.Find("inventory_icons.png"))
stack:Push(explore)
stack:Push(menu)

function update()
    local dt = GetDeltaTime()
    stack:Update(dt)
    stack:Render(gRenderer)
    gWorld:Update(dt)
end

```

Listing 2.213: Demonstrating the ItemMenuState. In main.lua.

In Listing 2.213 gWorld and gIcons are created as global variables. Both of these variables are used by the menu states to display information about the inventory. Run the project and you'll be able to navigate to the ItemMenuState and browse the items (there aren't many).

Try adding new items to the mItem and mKeyItem tables in the World class and check out your changes. At this point you can even create your own items by adding or altering the ItemDB table!

The World class tracks the gold and time, so we should update the FrontMenuState to render the actual values. Copy the changes from Listing 2.214.

```

function FrontMenuState:Render(renderer)

    -- code omitted

    renderer:DrawText2d(goldX + 10, goldY, gWorld:GoldAsString())

```

```
    renderer:DrawText2d(goldX + 10, goldY - 25, gWorld:TimeAsString())
end
```

Listing 2.214: Hooking the gold and time values up to the FrontMenuState. In FrontStateMenu.lua.

Example menu-7-solution contains the completed code. There's still a lot to add to the menu but we've now come to the end of this part of the book. We'll add new states to the menu as they're needed.

In the next section we'll take everything we've created so far and build a small game based around exploration!

An Exploration Game

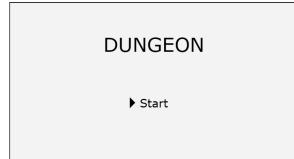
This is the final section of the Exploration part of the book. We're going to make a small game using the codebase we've built up so far. Even at this stage we have all the tools required to make a small RPG-like game that's a meaningful, interesting experience. We don't have combat at this point or a quest infrastructure, but many interesting games don't use these elements at all.

The game we'll make is called Dungeon, and it's about escaping a dungeon. It's not going to be very long but it does have some interaction and hopefully leaves the player wanting to know more.

The Introduction Storyboard

When planning a game, storyboards are a great tool to summarize an ideal play-through or non-interactive sequence. The minigame we'll make is the classic JRPG mix of cutscenes, exploration, puzzle solving and NPC interaction.

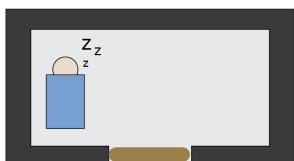
To get a clear idea of how we're going introduce the game, check out the storyboard below Figure 2.100. This storyboard is the introduction that sets the scene and leads into the playable game. It attempts to set up a little intrigue and hopefully hook in the player.



Menu screen.



Fade to black
Show message
Fade out test



Fade into game
Player sleeping in bed
Fade to black



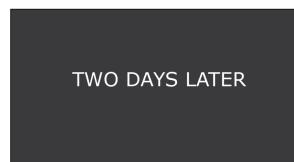
Crash sound



Fade into game
Door is open
Guard in room
Guard shouts "Take him!"



Fade to black
Bells and rain sounds
Rain fades
Silence



Creaking and wind
sounds
Sounds fade out
Text fades in



Fade in prison cell
Hero stands up
"Where am I?"

Figure 2.100: The introduction storyboard, showing the hero kidnapped during the night.

The storyboard illustrates a rough outline of the game introduction. In this case we've started right at the start and included the first menu screen. The user will select "New Game" and trigger the opening cutscene to run. The screen starts black and the words "Sontos Village, Midnight" fade in, introducing the first location to the player. Then the black screen fades to a tile map showing the player sleeping in bed. The screen fades out again and a crash sound plays. Then we fade back to the map to see a soldier standing over the player saying "Get him!" We fade out one more time and use text to announce two days have passed. Finally we see the player waking up in a prison cell.

The storyboard helps identify which assets need creating. There are two maps, the house and the prison. At least 2 character sprites, the hero and the guard. There are also a number of sound effects: wind, rain, crashing and creaking.

Now that we've got a good idea of what we want to happen, we need to write the code!

Creating A Cutscene

A storyboard could be modeled using a StateStack by pushing and popping states for each part of the cutscene. That's a good start but a higher-level approach would be better. We want a nice abstraction that lets us describe what should happen in a way that's closer to the storyboard diagram. That way we can iterate on our ideas and quickly see the changes.

Let's come at the problem from the point of view of "What would be the nicest, simplest way to encode the introduction storyboard?"¹¹. A table that describes each event would be a good start and that table might look something like Listing 2.215 below.

```
introduction =
{
    --
    -- Titles
    --
    scene { name = "house", map = houseMap },
    black_screen,
    play_sound "rain",
    caption { style = title, "Sontos Village" },
    caption { style = subtitle, "Midnight" },
    black_screen fade_out,
```

¹¹I often find it useful to work this way. First I consider how I'd like to write something at a high level and then, later, think about how to make that work. Lua is well suited to this kind of development.

```

--  

-- Sleeping NPC  

--  

--  

wait 3,  

--  

-- Enter Guard  

--  

black_screen fade_in,  

wait 1,  

play_sound "crash",  

addnpc "mysterious_guard" guardef, 0, 0,  

black_screen fade_out,  

say "mysterious_guard" "You're coming with me!"  

--  

-- Kidnap  

--  

black_screen fade_in,  

play "church_bell"  

fade_out_sound "church_bell",  

fade_out_sound "rain"  

--  

-- Travel  

--  

replace_scene "house" { name = "jail", map = jailMap },  

fade_in_sound "creaking"  

fade_in_sound "wind"  

caption { style = title, "Two days later..."},  

fade_out_sound all,  

black_screen fade_out,  

wait 0.1 ,  

textbox "Where am I?"  

}

```

Listing 2.215: An idealized way that we might encode the storyboard.

The above storyboard-as-table isn't valid Lua but it's not too far off and it's pretty readable. Let's use it as a model for how we'd like to create cutscenes in the game.

Now comes the tricky part! Each event in the table needs to be run in order. Some events run at the same time but other events must prevent any events that follow from being updated. How are we going to manage this?

We'll start by working on a simpler storyboard table and then build our way up, event by event, to something approaching the introduction table above.

Example dungeon-1 contains all our code so far and two new files, Storyboard.lua and StoryboardEvents.lua. The Storyboard class runs the cutscene tables, and the StoryboardEvents.lua file contains the events and operations the cutscenes are made up from.

Each entry in the cutscene table is an operation that creates an event that occurs over a small period of time. We'll start with the simplest operation and event pair, Wait.

A One-Operation Cutscene

We'll consider the list of events first and worry about the Storyboard implementation later. Let's start simple with a storyboard with only *one* type of event, the wait event. The cutscene will wait for a few seconds, then finish.

Check out Listing 2.216 where we set everything up in main.lua. Note how the Storyboard class is a state that can be pushed onto the StateStack.

```
LoadLibrary('Asset')
Asset.Run('Dependencies.lua')

gRenderer = Renderer.Create()
gStack = StateStack>Create()

local intro =
{
    Wait(5),
    Wait(2)
}
local storyboard = Storyboard>Create(gStack, intro)
gStack:Push(storyboard)

function update()
    local dt = GetDeltaTime()
    gStack:Update(dt)
    gStack:Render(gRenderer)
end
```

Listing 2.216: The main code showing how we'll use the Storyboard and events. In main.lua.

The main.lua code shows a short, two-instruction storyboard called intro. A Storyboard state is created and pushed onto the top of the stack. The storyboard waits for

5 seconds, then it waits for 2 seconds. After 7 seconds both events have finished and the Storyboard state pops itself off the stack.

The intro table has two entries, Wait(5) and Wait(2). These are both functions. Both functions get executed when the file is first loaded. When we call the Wait function with a number of seconds, what gets returned?

A storyboard table only ever contains two types, functions or event objects. When we first declare a storyboard table, every entry is a function, resembling something like Listing 2.217.

```
expanded_storyboard =
{
    function() --[[ code ]] end,
    function() --[[ code ]] end,
    ...
}
```

Listing 2.217: The makeup of a storyboard table.

Each frame the Storyboard class loops over its cutscene table, like the one pictured in Listing 2.217. If it comes to a function it calls it. Every function returns an Event object and this object replaces the function in the table. As the Storyboard runs, more and more of the functions are run and turned into Event objects. Don't worry if it seems a little confusing now, as it will become clearer as we add the code bit by bit.

We call the functions in the Storyboard table *operations* and the objects *events*. Each *operation* when run returns an *event* object.

The example in Listing 2.218 shows how our wait table changes after it's defined and once the Storyboard class updates it.

```
-- how the code looks in the source file
intro =
{
    Wait(5),
    Wait(2)
}

-- main.lua is loaded
intro =
{
    function() return WaitEvent:Create(5) end,
    function() return WaitEvent:Create(2) end
}

-- Intro Table After Storyboard class is Updated
intro =
```

```
{
    table: 0x2273ec0, -- the WaitEvent object
    function() return WaitEvent:Create(2) end
}
```

Listing 2.218: How the storyboard table changes.

When `Wait(5)` is called, it returns a function that sits in the event table. When the Storyboard runs, this function is called and it returns an Event object.

Why not just return an Event object immediately? Well, timing is important. Sometimes we'll only want events to trigger when the Storyboard adds them - such as playing a sound, fading in a caption, etc.

Each frame the `Storyboard:Update` function is called and it updates all its active event objects. The `Wait` event object will tick down the time until all its seconds are used up. Then it will say it's finished and Storyboard will move on to the next event.

WaitEvent

We're going to create a lot of events. Each event only contains a few lines of code, so rather than have a separate file for each event we'll put them all in `StoryboardEvents.lua`.

Like the states, each Event must have certain functions. Listing 2.219 shows the functions required.

```
GEvent =
{
    function Create() ... end,
    function Update(dt) ... end,
    function IsBlocking() ... end,
    function IsFinished() ... end
}
```

Listing 2.219: The functions all events are expected to have.

Let's look at the `Wait` event so we have an example of how these functions are implemented. Copy the code from Listing 2.220 into `StoryboardEvents.lua`.

```
WaitEvent = {}
WaitEvent.__index = WaitEvent
function WaitEvent:Create(seconds)
    local this =
```

```

{
    mSeconds = seconds,
}
setmetatable(this, self)
return this
end

function WaitEvent:Update(dt)
    self.mSeconds = self.mSeconds - dt
end

function WaitEvent:IsBlocking()
    return true
end

function WaitEvent:IsFinished()
    return self.mSeconds <= 0
end

function Wait(seconds)
    return function(storyboard)
        return WaitEvent:Create(seconds)
    end
end

```

Listing 2.220: Create a Wait function and WaitEvent class. In StoryboardEvents.lua.

It's best to read the Wait operation near the bottom before looking at the rest of the code. The Wait function is used when setting up our storyboard. It takes in one parameter, the number of seconds to wait. It returns a function that takes in a Storyboard and returns a WaitEvent object.

The Wait function returns an inner function. The outer function is called when it's declared in a storyboard, and the inner function is called when the storyboard runs it. All operations take in the Storyboard as a parameter.

Let's look at the WaitEvent next. The constructor stores the number of seconds to wait. The Update function reduces the number of seconds by the delta time of each frame.

The IsFinished function reports whether or not the event is active. If the event object isn't active, it's not updated. The IsFinished function returns true if no seconds remain.

The IsBlocking function lets the Storyboard know if it should process the next event in the list. If IsBlocking returns true, no other events are processed until this one finishes. Wait, by definition, should make the Storyboard stop and wait, so it returns true.

Now that we've seen how the WaitEvent is constructed, we can build the Storyboard class. With each step we get a little closer to running the idealized storyboard defined in Listing 2.217.

To use an event we also need to define the Storyboard class itself. A Storyboard is a state that can be pushed onto the statestack. It contains a list of events and manages which ones are executed.

Copy the code from Listing 2.221.

```
Storyboard = {}
Storyboard.__index = Storyboard
function Storyboard:Create(stack, events)
    local this =
    {
        mStack = stack,
        mEvents = events,
    }

    setmetatable(this, self)
    return this
end

function Storyboard:Enter() end
function Storyboard:Exit() end
function Storyboard:HandleInput() end
function Storyboard:CleanUp() end

function Storyboard:Update(dt)

    if #self.mEvents == 0 then
        self.mStack:Pop()
    end

    local deleteIndex = nil
    for k, v in ipairs(self.mEvents) do

        if type(v) == "function" then
            self.mEvents[k] = v(self)
            v = self.mEvents[k]
        end

        v:Update(dt, self)
        if v:IsFinished() then
            deleteIndex = k
            break
        end
    end
end
```

```

        if v:IsBlocking() then
            break
        end
    end

    if deleteIndex then
        table.remove(self.mEvents, deleteIndex)
    end
end

function Storyboard:Render(renderer)
    local debugText = string.format("Events Stack: %d", #self.mEvents)
    renderer:DrawText2d(0, 0, debugText)
end

```

Listing 2.221: A first implementation of the Storyboard class. In Storyboard.lua.

The constructor for the Storyboard adds the stack and events into its this table. Most of the state functions are empty. The two functions containing code are Update and Render.

The Update function checks the number of events in the storyboard. If there are no entries it pops itself off the stack. If there are entries we check if the next one is a function. If the entry is a function it means this is the first time we've ever seen this operation. When the Storyboard reaches a function it calls it, takes the returned event object, and replaces the operation in the table. In our case a WaitEvent is returned.

Once the function check is done, the event object is updated. If the event has finished, the variable deleteIndex is set to the current loop index and we break out of the loop. Outside the loop we remove the deleted event from the events table. If the current event hasn't finished, then there's a check to see if it's blocking. If the event blocks, no more events are updated.

In our example there are two Wait instructions. The first is expanded into a WaitEvent and blocks the storyboard from running any more events. After 5 seconds, the first WaitEvent finishes. It's removed, and the storyboard moves on to the second Wait instruction.

The Render loop contains debug code to show that something is happening when we run the example.

Figure 2.101 shows how the mEvents table in the Storyboard state changes as time passes and events are processed.

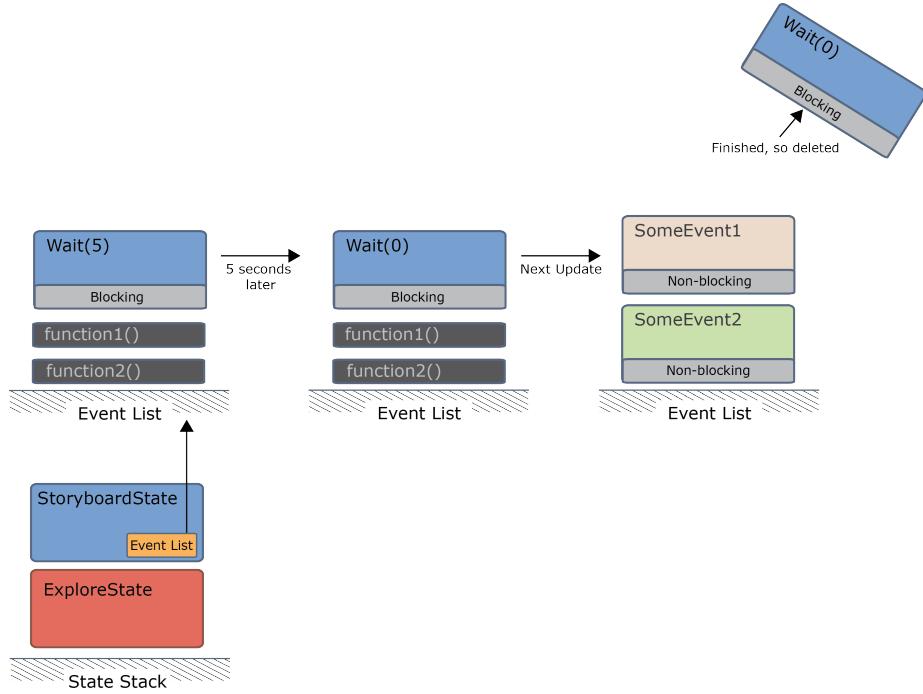


Figure 2.101: How the Storyboard changes as it's updated over a number of frames.

As you can see in Figure 2.101 the Storyboard is using a Wait event that blocks. This means the other operations on the stack haven't been touched and are still functions. When the Wait function finishes, it's removed from the event list. The operations are run and replaced by their event objects. These events don't block so both are updated. There's a lot going on, but as we work through some more events and storyboards it will become clearer.

Example dungeon-1-solution contains the full code. Run it. You'll see the text "Events Stack: 2", then after 5 seconds the first Wait event finishes and is removed. The text then says "Events Stack: 1". After another 2 seconds, the last event is removed and the Storyboard itself is popped off the stack. With the Storyboard gone, the render function stops getting called, so there's no text shown on screen. Try it out! You'll see something like Figure 2.102.



Figure 2.102: The output from example dungeon-1-solution.

The Storyboard Render Stack

Now that we have a simple proof of concept for the Storyboard, let's make it more powerful! We'll allow events to change what's rendered on screen by adding an internal stack into the storyboard. Then we'll add several new operations that can manipulate the states on its internal stack.

The Screen Operation

We'll start with a Screen event. This is going to be a simple state that sits on the top of the stack and draws a colored rectangle over the entire screen. The code we've written so far is available in `dungeon-2` and it contains three new files: `ScreenState.lua`, `CaptionState.lua`, and `CaptionStyles.lua`.

Our Dungeon minigame uses custom fonts. In the art directory there are fonts called "junction.ttf" and "contra_italic.ttf". Open `manifest.lua` in `dungeon-2` and you'll see a fonts table like in Listing 2.222.

```
['fonts'] =
{
    ["default"] =
```

```

{
    path = "art/junction.ttf",
},
["title"] =
{
    path = "art/contra_italic.ttf",
}
}

```

Listing 2.222: Adding a default font. In manifest.lua.

If you set a font's id to "default" in the manifest, then all text is drawn using that font unless told otherwise.

Let's begin by extending the intro table as shown in Listing 2.223 in main.lua.

```

local intro =
{
    BlackScreen(),
    Wait(2),
    Wait(5),
}

```

Listing 2.223: Black screen for the introduction. In main.lua.

The BlackScreen operation affects what's rendered on screen. This means it needs a render function. If we have multiple screen events, then we need to know the correct order to render them. What's the best way to handle this kind of rendering? A stack!

The Storyboard object itself is on the global stack. We could add the screen states on top of the storyboard but, if we do this, things become complicated. If we want to delete the Storyboard before it's finished, we need to know which events belong to the Storyboard and delete them too!

Everything becomes simpler if add an internal stack to the Storyboard. All storyboard events are pushed and popped off the internal stack. When the Storyboard.Exit function is called, we clean up this internal stack automatically and don't have to worry about leaking states.

Each state we push onto the storyboard internal stack also has an id. These ids let the events interact with states. Copy the code changes from Listing 2.224.

```

function Storyboard:Create(stack, events)
    local this =
    {
        mStack = stack,

```

```

        mEvents = events,
        mStates = {},
        mSubStack = StateStack>Create()
    }
    -- code omitted
end

function Storyboard:Update(dt)
    self.mSubStack:Update(dt)

    -- code omitted
end

function Storyboard:Render(renderer)
    self.mSubStack:Render(renderer)
end

function Storyboard:PushState(id, state)
    -- push a State on the stack but keep a reference here.
    assert(self.mStates[id] == nil)
    self.mStates[id] = state
    self.mSubStack:Push(state)
end

```

Listing 2.224: Adding a StackState to the Storyboard. In Storyboard.lua.

Listing 2.224 adds a new StateStack called `mSubStack` to the Storyboard. The Storyboard Update and Render functions are updated to use this new internal stack.

In the constructor, the `mStates` table stores all the states that are pushed onto the internal storyboard stack. The states are the values and the keys are string ids. A new function, `PushState`, loads states onto the storyboard stack. It pushes states onto the `mSubStack` and records the state in `mStates` table using the id.

We're now ready to create the BlackScreen operation. It adds a ScreenState to the Storyboard using the new PushState function. The code for the ScreenState is quite simple. Copy it from Listing 2.225.

```

ScreenState = {}
ScreenState.__index = ScreenState
function ScreenState>Create(color)
    params = params or {}
    local this =
    {
        mColor = color or Vector.Create(0, 0, 0, 1),
    }

```

```

    setmetatable(this, self)
    return this
end

function ScreenState:Enter() end
function ScreenState:Exit() end
function ScreenState:HandleInput() end

function ScreenState:Update(dt)
    return true
end

function ScreenState:Render(renderer)
    renderer:DrawRect2d(
        -System.ScreenWidth()/2,
        System.ScreenHeight()/2,
        System.ScreenWidth()/2,
        -System.ScreenHeight()/2,
        self.mColor)
end

```

Listing 2.225: A state that renders a rectangle over the entire screen. In ScreenState.lua.

The ScreenState in Listing 2.225 renders a rectangle that covers the entire screen.

Our example cutscene has an operation called BlackScreen. We could have used a more general operation like Screen(Vector.Create(0,0,0,1)), but black screens are so common that it's worth making a custom operation. The storyboard table becomes less cluttered and easier to read if operations don't take many arguments.

The BlackScreen function is defined in Listing 2.226.

```

EmptyEvent = WaitEvent>Create(0)

function BlackScreen(id, alpha)

    id = id or "blackscreen"
    local black = Vector.Create(0, 0, 0, alpha or 1)

    return function(storyboard)
        local screen = ScreenState>Create(black)
        storyboard:PushState(id, screen)
        return EmptyEvent
    end
end

```

Listing 2.226: Creating a BlackScreen instruction. In StoryboardEvents.lua.

Check out the first line of Listing 2.226. We've defined a global event called EmptyEvent that waits for 0 seconds. Why would we make such an odd event?

Well, in the case of an operation like BlackScreen, all it does is push a black screen state onto the internal stack. That happens instantly. It doesn't need to hang around in the event list. But all cutscene operations *must* return an Event object, so operations that occur instantly return an EmptyEvent.

The BlackScreen instruction optionally takes in an id parameter. The id identifies the state after it's been added to the storyboard. The id defaults to "blackscreen". The inner function, called when the event runs, creates a black colored ScreenState and pushes it onto the storyboard's internal stack. There's an optional alpha parameter that controls whether the screen starts opaque, transparent, or somewhere in between. Finally it returns the EmptyEvent as it has nothing else to do.

Run the code and you'll see a black screen. After 7 seconds the storyboard finishes and pops itself off the stack. Once the storyboard finishes, the default background color of the program is rendered.

Fade Operations

A black screen on its own isn't very interesting, so let's add some fade instructions. A fade event alters the screen alpha over time, often from 0 to 1 or 1 to 0. This sounds like a good job for a tween!

Tweens are pretty versatile, so instead of a FadeEvent we'll create a TweenEvent that can do the job of the FadeEvent and many more! The TweenEvent takes three parameters:

- tween - the Tween object.
- target - the target object the tween's value is applied to.
- ApplyFunc - a function that controls how the tween value is applied to the target.

Here's the code for the TweenEvent in Listing 2.227.

```
TweenEvent = {}
TweenEvent.__index = TweenEvent
function TweenEvent:Create(tween, target, ApplyFunc)
    local this =
    {
        mTween = tween,
        mTarget = target,
        mApplyFunc = ApplyFunc
    }
    setmetatable(this, self)
```

```

        return this
end

function TweenEvent:Update(dt, storyboard)
    self.mTween:Update(dt)
    self.mApplyFunc(self.mTarget, self.mTween:Value())
end

function TweenEvent:Render() end

function TweenEvent:IsFinished()
    return self.mTween:IsFinished()
end

function TweenEvent:IsBlocking()
    return true
end

```

Listing 2.227: An event that performs a tween. In StoryboardEvents.lua.

The TweenEvent takes in the tween, target and ApplyFunc function and stores them in its this table. Each time its Update function is called, it updates the tween and calls the ApplyFunc with the target object and the tween's current value. This let us use tween events for all kinds of things: scaling, fading, moving, etc. The TweenEvent blocks until the tween finishes.

With the TweenEvent created, adding FadeScreenIn and FadeScreenOut instructions becomes a lot easier.

Let's create a function called FadeScreen that FadeScreenIn and FadeScreenOut both use. The three functions are defined in Listing 2.228.

```

function FadeScreen(id, duration, start, finish)

    local duration = duration or 3

    return function(storyboard)

        local target = storyboard.mSubStack:Top()
        if id then
            target = storyboard.mStates[id]
        end
        assert(target and target.mColor)

        return TweenEvent>Create(
            Tween>Create(start, finish, duration),

```

```

        target,
        function(target, value)
            target.mColor:SetW(value)
        end
    end

    function FadeInScreen(id, duration)
        return FadeScreen(id, duration, 0, 1)
    end

    function FadeOutScreen(id, duration)
        return FadeScreen(id, duration, 1, 0)
    end

```

Listing 2.228: Fading Instructions. In StoryboardEvents.lua.

FadeInScreen and FadeOutScreen use the FadeScreen function directly. They only differ in the last two parameters, which represent the start and end alpha for the screen. The fade in goes from alpha 0 to alpha 1 and the fade out from alpha 1 to alpha 0.

FadeScreen takes in four parameters: the id of the screen to fade, the duration for the fade to take, and finally the start and finish alpha values. Duration is an optional parameter. If no duration is given, it's set to three seconds. These parameters are used to create a TweenEvent.

The FadeScreen inner function is called when the operation is executed. To get the screen to fade we use the id to look it up, or if no id exists we get the screen from the top of the stack. There's an assert to make sure that the target exists and has an mColor member. With the target identified, we're ready to create the TweenEvent. We create a tween using the target, start, finish and duration values. The ApplyFunc is defined inline and sets the target's color alpha to whatever the tween value is.

While the TweenEvent runs, it alters the transparency of the ScreenState color.

Let's use the operation. Copy the code below Listing 2.229 into your main.lua file.

```

local intro =
{
    BlackScreen(),
    FadeOutScreen(),
    Wait(2),
    FadeInScreen(),
    Wait(5),
}

```

Listing 2.229: Trying out the Fade instructions. In main.lua.

Let's examine the intro storyboard line by line.

BlackScreen pushes a black screen onto the stack and returns an EmptyEvent.

FadeOutScreen gets the state on the top of the stack, which is the black screen. It then fades the screen from alpha 1 to alpha 0 over 3 seconds.

After the 3 seconds have passed, the WaitEvent is created and the storyboard waits for 2 more seconds.

FadeInScreen fades the black screen back in over a period of 3 seconds. After a final 5-second wait the storyboard finishes and pops itself off the stack.

Run this code and you'll start to get a taste of how powerful this abstraction is and how it makes writing these storyboards so much easier than hard coding everything. At this point it's worth playing around with the operation order and perhaps even modifying some of the operation or adding your own.

Caption Operation

The Caption operation starts to make the storyboards really useful. It communicates directly with the user by displaying text on the screen.

Plain text is boring! Therefore we'll write code to make styling the caption text easy. Style information describes the font, size, color etc. In CaptionStyles.lua, add the two classes shown in Listing 2.230.

```
local DefaultRenderer =
function(self, renderer, text)
    renderer:SetFont(self.font)
    renderer:ScaleText(self.scale, self.scale)
    renderer:AlignText(self.alignX, self.alignY)
    renderer:DrawText2d(self.x, self.y, text, self.color, self.width)

    -- Reset renderer
    local default = CaptionStyles["default"]
    renderer:SetFont(default.font)
    renderer:ScaleText(default.scale, default.scale)
end

local FadeApply =
function(target, value)
    target.color:SetW(value)
end

CaptionStyles =
{
    ["default"] =
```

```

{
    font = "default",
    scale = 1,
    alignX = "center",
    alignY = "center",
    x = 0,
    y = 0,
    color = Vector.Create(1, 1, 1, 1),
    width = -1,
    Render = DefaultRenderer,
    ApplyFunc = function() end,
    duration = 3,
},
["title"] =
{
    font = "title",
    scale = 3,
    y = 75,
    ApplyFunc = FadeApply,
},
["subtitle"] =
{
    scale = 1,
    y = -5,
    color = Vector.Create(0.4, 0.38, 0.39, 1),
    ApplyFunc = FadeApply,
    duration = 1
}
}

```

Listing 2.230: Two style classes to store how to render our text. In CaptionStyles.lua.

CaptionStyles.lua contains a CaptionStyles table with three entries:

- default - a style that determines the default style values.
- title - a style for titles.
- subtitle - a style for text beneath the title.

The fields in the style definitions describe how text is displayed, the font, scale, alignment, position, color and wrapping width.

Each style contains two functions, Render and ApplyFunc. The Render function renders the text. The ApplyFunc transitions the text. The ApplyFunc can fade text in, move it in from offscreen, or if you want to get complicated you can fade in each letter one after

the other etc. In our case, both the styles we've defined use the default renderer and FadeApply function, which means we'll be able to fade them in and out. There's a final duration field to determine how long the ApplyFunc takes.

The default style render function is set to DefaultRenderer. This function is defined at the top of Listing 2.230. The DefaultRenderer sets up the Render object with the style information for the text, renders it out, and then reverts the style information back to default.

The FadeApply function alters the alpha channel of the style's color field. Like with ItemDB, not all fields need to be defined. Missing fields are filled in automatically at the end of the file. This is shown in Listing 2.231.

```
--  
-- Set caption defaults  
--  
  
for name, style in pairs(CaptionStyles) do  
    if name ~= "default" then  
        for k, v in pairs(CaptionStyles.default) do  
            style[k] = style[k] or v  
        end  
    end  
end
```

Listing 2.231: Adding the default values to the styles. In CaptionStyles.lua.

Let's add the CaptionState next. It's simple, as most of the work is done in the style. The code in Listing 2.232 goes into CaptionState.lua.

```
CaptionState = {}  
CaptionState.__index = CaptionState  
function CaptionState:Create(style, text)  
    params = params or {}  
    local this =  
    {  
        mStyle = style,  
        mText = text  
    }  
  
    setmetatable(this, self)  
    return this  
end  
  
function CaptionState:Enter() end  
function CaptionState:Exit() end  
function CaptionState:HandleInput() end
```

```

function CaptionState:Update(dt)
    return true
end

function CaptionState:Render(renderer)
    self.mStyle:Render(renderer, self.mText)
end

```

Listing 2.232: A Caption State for using the stack to display text. In CaptionState.lua.

The CaptionState takes in a style and text parameter. In the Render function it uses the style to render the text.

We're nearly ready to write the Caption operation, but first there's a utility function we need to add to Util.lua. Here's the code Listing 2.233.

```

function ShallowClone(t)
    local clone = {}
    for k, v in pairs(t) do
        clone[k] = v
    end
    return clone
end

```

Listing 2.233: A shallow clone function. In Util.lua.

This ShallowClone function makes a copy of a Lua table. It's called a shallow clone because it doesn't clone nested tables. The code below in Listing 2.234 shows how the ShallowClone works.

```

-- b is a reference to a
a = { label = "dog" }
print(a.label) -- dog
b = a
print(b.label) -- dog
b.label = "cat"
print(a.label, b.label) -- cat, cat

-- b is a shallow clone of a
b = ShallowClone(a)
b.label = "monkey"
print(a.label, b.label) -- cat, monkey

```

```
-- Not a deep clone
a = { nested = {label = "nest" } }
b = ShallowClone(a)
b.nested.label = "hello"
print(a.nested.label, b.nested.label) -- hello, hello
```

Listing 2.234: How the shallow clone works.

Now that we understand ShallowClone, let's see why it's needed! Here's the Caption instruction, defined in StoryboardEvents.lua Listing 2.235.

```
function Caption(id, style, text)

    return function(storyboard)
        local style = ShallowClone(CaptionStyles[style])
        local caption = CaptionState>Create(style, text)
        storyboard:PushState(id, caption)

        return TweenEvent>Create(
            Tween>Create(0, 1, style.duration),
            style,
            style.ApplyFunc)
    end
end
```

Listing 2.235: Defining the Caption instruction. In StoryBoardEvents.lua.

The Caption operation takes in style and text parameters. When the operation is executed it clones the style and creates a CaptionState object using the cloned style. Cloning the style means we can alter the values stored in the style and not worry about affecting other pieces of text. If we didn't clone the style and changed its color, we'd change the color for all the text using that style! Cloning the style means it's unique for our caption.

We push the CaptionState on to the storyboard's internal stack so it can render the caption. Then we create a TweenEvent using data from the style to handle the in-transition of the text.

Let's create a FadeOutCaption instruction in a similar way. See the code below in Listing 2.236.

```

function FadeOutCaption(id, duration)
    return function(storyboard)
        local target = storyboard.mSubStack:Top()
        if id then
            target = storyboard.mStates[id]
        end
        local style = target.mStyle
        duration = duration or style.duration

        return TweenEvent:Create(
            Tween:Create(1, 0, duration),
            style,
            style.ApplyFunc
        )
    end
end

```

Listing 2.236: Fade out caption instruction. In StoryboardEvents.lua.

The FadeOutCaption operation is similar to FadeOutScreen but it works on the style of the CaptionState.

Let's update the intro to make use of the new caption operations. Copy the code from Listing 2.237 into your main.lua file.

```

local intro =
{
    BlackScreen(),
    Caption("place", "title", "Village of Sontos"),
    Caption("time", "subtitle", "MIDNIGHT"),
    FadeOutCaption("place"),
    FadeOutCaption("time"),
    Wait(10),
}

```

Listing 2.237: Fading captions in and out. In main.lua.

Run the code and the large title will fade in, followed by the subtitle. The large title then fades out, followed by the subtitle. Figure 2.103 shows the caption just as it's starting to fade.



Figure 2.103: A caption instruction displayed using a storyboard.

The intro cutscene looks quite nice, but it would better if both titles faded together. This would happen if FadeOutCaption("place") didn't block the storyboard.

NoBlock Operation

Let's add a special function to stop an instruction from blocking. Open the Storyboard-Events.lua file and copy the code from Listing 2.238.

```
function NoBlock(f)
    return function(...)
        local event = f(...)
        event.IsBlocking = function()
            return false
        end
        return event
    end
end
```

Listing 2.238: The NoBlock function. In StoryboardEvents.lua.

The NoBlock function replaces an event's IsBlocking command, so it always returns false. Let's update the intro table and try it out. Copy the code from Listing 2.239 into your main.lua function.

```
local intro =
{
    BlackScreen(),
    Caption("place", "title", "Village of Sontos"),
    Caption("time", "subtitle", "MIDNIGHT"),
    Wait(1),
    NoBlock(
        FadeOutCaption("place", 3)
    ),
    FadeOutCaption("time", 3),
    Wait(10),
}
```

Listing 2.239: A more pleasing title fade out. In main.lua.

Run the code and the titles both fade at the same time!

In Listing 2.239 we set the fade out to 3 seconds. If we didn't pass in a duration parameter it would use the style's default value.

We've added a cool caption and faded it in and out, but the CaptionState is still on the stack! We need another operation to remove it. Each state on the stack has an id, so we can use that to remove it. Copy the code from Listing 2.240.

```
function Storyboard:RemoveState(id)
    local state = self.mStates[id]
    self.mStates[id] = nil
    for i = #self.mSubStack.mStates, 1, -1 do
        local v = self.mSubStack.mStates[i]
        if v == state then
            table.remove(self.mSubStack.mStates, i)
        end
    end
end
```

Listing 2.240: Remove State function. In Storyboard.lua.

The RemoveState removes a state from the mStates and mSubStack tables that matches the passed-in id.

To use the RemoveState we need an new operation, KillState. Copy the code from Listing 2.241 into StoryboardEvents.lua.

```

function KillState(id)
    return function(storyboard)
        storyboard:RemoveState(id)
        return EmptyEvent
    end
end

```

Listing 2.241: Kill State instruction. In StoryboardEvents.lua.

With the KillState operation defined, we can create the first part of the introduction cutscene! Copy the code from Listing 2.242.

```

local intro =
{
    BlackScreen(),
    Caption("place", "title", "Village of Sontos"),
    Caption("time", "subtitle", "MIDNIGHT"),
    Wait(2),
    NoBlock(
        FadeOutCaption("place", 3)
    ),
    FadeOutCaption("time", 3),
    KillState("place"),
    KillState("time"),
    FadeOutScreen(),
}

```

Listing 2.242: A screen with appearing and disappearing captions.

Example dungeon-2-solution contains the complete code. Run the code and you'll see the title and subtitle fade in, one after another, then they'll stay for a little while before fading out together. The black screen then fades out and the storyboard finishes.

That's it for this section. Next we're going to add operations to play sounds and a special operation to load a scene in the background.

Sound Events

Our introduction cutscene includes the sound of rain and church bells. To play these we use a new type of event.

The code so far is available in dungeon-3. You'll notice that it has an extra subdirectory called "sound", and that there's a file in there called rain.wav. The manifest file has been modified to reference this file as below in Listing 2.243. You can either use the example code or update your own codebase to match it.

```

[ 'sounds' ] =
{
    [ 'rain' ] =
    {
        path = "sound/rain.wav"
    }
}

```

Listing 2.243: Add a sound asset to the manifest file. In manifest.lua.

To use sounds we need to include the sound library. Open Dependencies.lua. Add "Sound" just after "Keyboard" in the library list as shown in Listing 2.244.

```

    "Keyboard",
    "Sound",
},
function(v) LoadLibrary(v) end)

```

Listing 2.244: Adding the sound library. In Dependencies.lua.

To handle sounds, we need operations to play, stop and fade sounds. Let's start by adding Play and Stop operations. As with the states, we need to store a reference to any sound played so future operations can alter them. Listing 2.245 shows the instruction code.

```

function Play(soundName, name, volume)
    name = name or soundName
    volume = volume or 1
    return function(storyboard)
        local id = Sound.Play(soundName)
        Sound.SetVolume(id, volume)
        storyboard:AddSound(name, id)
        return EmptyEvent
    end
end

function Stop(name)
    return function(storyboard)
        storyboard:StopSound(name)
        return EmptyEvent
    end
end

```

Listing 2.245: Play and Stop operations for sounds. In StoryboardEvents.lua.

The Play operation takes in the name of a sound, an optional name to identify the instance of the sound being played, and an optional volume. We may want to play a single sound multiple times - a footstep sound, for instance. If the same sound is being played at the same time, you can't depend on the sound name to uniquely identify it; that's why there's also an optional name parameter. The name parameter might be "footstep1" or "footstep2" in the case of a footstep sound. If a name isn't supplied, it's set to the soundName value. If the volume isn't supplied it's set to 1, which is 100% of the default volume.

The inner function calls Sound.Play, which returns a reference to the sound being played. We store the sound reference as id. The id can be used to stop the sound later if we want. Next we set the volume using id and add the sound being played to the storyboard.

Stop just calls StopSound on the Storyboard using the name parameter to identify which sound to stop.

We need to add the AddSound and StopSound functions to the Storyboard. We also need to update the constructor with a table to track playing sounds. When we exit the Storyboard we go through the sound table and stop each sound from playing. Copy the code from Listing 2.246.

```
function Storyboard:Create(stack, events)
    local this =
    {
        -- code omitted
        mPlayingSounds = {}
    }

    setmetatable(this, self)
    return this
end

function Storyboard:Exit()
    for k, v in pairs(self.mPlayingSounds) do
        Sound.Stop(v)
    end
end

function Storyboard:AddSound(name, id)
    assert(self.mPlayingSounds[name] == nil)
    self.mPlayingSounds[name] = id
end

function Storyboard:StopSound(name)
    local id = self.mPlayingSounds[name]
    self.mPlayingSounds[name] = nil
```

```
    Sound.Stop(id)
end
```

Listing 2.246: Functions to add sounds and remove them. In Storyboard.lua.

The AddSound function adds a sound id to the `mPlaySounds` table using the name as the key. There's an assert to ensure that a sound with the same name isn't already stored in the table.

`StopSound` looks up the sound using the name, removes it from the `mPlaySounds` table, and then stops it playing using the sound library.

Let's try the operations out. Copy the updated intro table from Listing 2.247.

```
local intro =
{
    BlackScreen(),
    Play("rain"),
    Caption("place", "title", "Village of Sontos"),
    Caption("time", "subtitle", "MIDNIGHT"),
    Wait(2),
    NoBlock(
        FadeOutCaption("place", 3)
    ),
    FadeOutCaption("time", 3),
    Stop("rain"),
    KillState("place"),
    KillState("time"),
    FadeOutScreen(),
}
```

Listing 2.247: An introduction cutscene.

Run the code and you'll hear the rain sound play. Shortly after the titles fade out the sound stops. The stop is quite sudden. We can definitely do better and fade it out in a more gradual way.

To handle the sound fade we'll make a `FadeSound` operation. This operation changes the volume over a period of time. Copy the code from Listing 2.248.

```
function FadeSound(name, start, finish, duration)
    return function(storyboard)

        local id = storyboard.mPlayingSounds[name]
        return TweenEvent:Create(
```

```

        Tween:Create(start, finish, duration),
        id,
        function(target, value)
            Sound.SetVolume(target, value)
        end
    end
end

```

Listing 2.248: The FadeSound operation. In StoryboardEvents.lua.

The FadeSound operation reuses the TweenEvent to change a sound's volume. We can now modify our intro to slowly fade in and fade out the rain sound. The updated code is in Listing 2.249.

```

BlackScreen(),
Wait(4),
Play("rain"),
NoBlock(
    FadeSound("rain", 0, 1, 3) -- 0 -> 1 in 3 secs
),
Caption("place", "title", "Village of Sontos"),
Caption("time", "subtitle", "MIDNIGHT"),
Wait(2),
NoBlock(
    FadeOutCaption("place", 3)
),
FadeOutCaption("time", 3),
FadeSound("rain", 1, 0, 1), -- 1 -> 0 in 1 sec
KillState("place"),
KillState("time"),
FadeOutScreen(),
Stop("rain")

```

Listing 2.249: Adding the rain sound effect. In main.lua.

Run the storyboard and the sound now nicely fades.

At the end of the cutscene we explicitly stop the rain sound but this isn't necessary. Once the Storyboard is popped off the stack it stops all the sounds.

That's it for the sound operations. Next we'll add map operations.

Map Events

Cutscenes often manipulate maps and the characters in them. Sometimes a cutscene needs to load a new map, while at other times it uses the map that is already loaded.

For instance, a cutscene might show an evil wizard plotting in his secret tower. To show this we need to load the tower map.

In the intro storyboard we're using two maps, a house and a dungeon. Neither map is loaded when the cutscene begins. We're going to write the code that lets us load and manipulate these maps. The map for the house is in the dungeon-3 folder as "map_sontos_house.lua" with its tileset "tileset_sontos_house.png". If you peek inside the map_sontos_house file you'll see its fully finished map. The map has its actions, trigger_type and triggers tables all set up.

To make maps easier to manage, we store them all in a central location, MapDB. MapDB is a table with string keys and function values. Each function creates and returns a map, like CreateHouseMap. Each key is a simple unique id for the map.

Create a file called MapDB.lua and add it to the manifest and Dependencies.lua files. Copy the code from Listing 2.250 to create the MapDB table and our first map entry.

```
MapDB =
{
    ["player_house"] = CreateHouseMap
}
```

Listing 2.250: Associate text names with map creation functions. In MapDB.lua.

Remember the ExploreState class? We wrote it to draw the map and allow the player to control the hero. In cutscenes we want to use the ExploreState but we don't want to see the hero, so we need a way to hide him.

We're going to hide the hero by pushing him underground to the -1 z level. Let's create a HideHero and a complementary ShowHero function to the ExploreState.

Generally players shouldn't move around during cutscenes. Therefore we'll move the player-handling code from Update to HandleInput. Storyboard doesn't call a state's HandleInput, so if we move the hero controls to that function, it stops the player from being able to move around during cutscenes. Listing 2.251 shows the updated code.

```
function ExploreState:HideHero()
    self.mHero.mEntity:SetTilePos(
        self.mHero.mEntity.mTileX,
        self.mHero.mEntity.mTileY,
        -1,
        self.mMap
    )
end

function ExploreState>ShowHero(layer)
    self.mHero.mEntity:SetTilePos(
```

```

        self.mHero.mEntity.mTileX,
        self.mHero.mEntity.mTileY,
        layer or 1,
        self.mMap
    )

function ExploreState:HandleInput()

    self.mHero.mController:Update(GetDeltaTime()) -- removed from Update
    -- code omitted
end

```

Listing 2.251: Hiding and Showing the Hero. In ExploreState.lua.

With these changes applied, we're ready to add some new storyboard operations. The first operation we'll create is called Scene.

Scene Operation

The Scene operation creates an ExploreState and pushes it onto the storyboard stack. Maps are complicated objects and therefore the Scene operation takes in a table of parameters.

Copy the Scene operation from Listing 2.252 into your StoryboardEvents.lua file.

```

function Scene(params)
    return function(storyboard)
        local id = params.name or params.map
        local map = MapDB[params.map]()
        local focus = Vector.Create(params.focusX or 1,
                                    params.focusY or 1,
                                    params.focusZ or 1)
        local state = ExploreState:Create(nil, map, focus)
        if params.hideHero then
            state:HideHero()
        end
        storyboard:PushState(id, state)

        -- Allows the following operation to run
        -- on the same frame.
        return NoBlock(Wait(0))()
    end
end

```

Listing 2.252: An instruction to load a scene. In StoryboardEvents.lua.

The Scene operation takes a params table with the following fields:

- name - the name to uniquely identify this scene. Optional. Defaults to the map name.
- map - a key to the MapDB. It's used to create a map.
- focusX, focusY, focusZ - where the camera focuses on the map.
- hideHero - if true, it hides the hero character.

The Scene operation uses the map id to get the map, creates it, adds it to an ExploreState, and pushes that onto the Storyboard stack. A Wait instruction wrapped in a NoBlock command is returned. The NoBlock ensures that the following instruction executes in the same frame.

Let's update the intro storyboard to use this new operation. Copy the code from Listing 2.253 into main.lua.

```
local intro =
{
    Scene
    {
        map = "player_house",
        focusX = 14,
        focusY = 20,
        hideHero = true,
    },
    BlackScreen(),
    Play("rain"),
    NoBlock(
        FadeSound("rain", 0, 1, 3)
    ),
    Caption("place", "title", "Village of Sontos"),
    Caption("time", "subtitle", "MIDNIGHT"),
    Wait(2),
    NoBlock(
        FadeOutCaption("place", 3)
    ),
    FadeOutCaption("time", 3),
    FadeSound("rain", 1, 0, 1),
    KillState("place"),
    KillState("time"),
    FadeOutScreen(),
    Wait(2),
    Stop("rain")
}
```

Listing 2.253: Updating the introduction storyboard. In main.lua.

Run the code. You'll see the titles fade in, hear the rain sound start, and see a fade into an empty room, as shown in Figure 2.104.



Figure 2.104: The intro storyboard fading into a map.

The code thus far is available in `dungeon-3-solution`.

NPCs In Cutscenes

We're doing pretty well; most of the intro cutscene is implemented! The next set of operations to add is concerned with NPCs.

The opening cutscene has an NPC in bed and a guard NPC that bursts into the room. We already have an action called `AddNPC`, and it seems a good idea to make our Storyboard able to run these actions. Then as the number of actions increases, so does the power of our storyboard! We'll use the `AddNPC` action to add the sleeping NPC into the bed. To communicate to the player that the NPC is sleeping we'll also add a "ZZZ" animation.

Let's deal with the "ZZZ" animation first.

There's no code for adding effects to an entity. It would be great to create things like exclamation marks over an NPC's head, a glittering chest, smoke from a fire, etc. To do this we're going to extend the `Entity` class.

Example `dungeon-4` contains the code so far as well as two new files, `sleeping.png` and `SleepState.lua`. The manifest and dependencies files have been updated too.

Entity Children

In `Map.lua` entity sprites are rendered using the `Map.Render` function. We're going to change this so entities are responsible for rendering themselves. Copy the code from Listing 2.254.

```
-- From this code in Map.RenderLayer
for _, j in ipairs(drawList) do
    renderer:DrawSprite(j.mSprite)
end

-- To this code
for _, j in ipairs(drawList) do
    j:Render(renderer)
end
```

Listing 2.254: Changing entity rendering. In Map.lua.

Let's add the Render function to the entity and support for attaching and rendering children. Copy the code from Listing 2.255.

```
function Entity:Create(def)
    local this =
    {
        -- omitted code
        mX = def.x or 0,
        mY = def.y or 0,
        mChildren = {}
    }

    -- omitted code

    return this
end

function Entity:AddChild(id, entity)
    assert(self.mChildren[id] == nil)
    self.mChildren[id] = entity
end

function Entity:RemoveChild(id)
    self.mChildren[id] = nil
end

function Entity:SetTilePos(x, y, layer, map)

    -- omitted code
    self.mX = x
    self.mY = y
end
```

```

function Entity:Render(renderer)
    renderer:DrawSprite(self.mSprite)

    for k, v in pairs(self.mChildren) do
        local sprite = v.mSprite
        print(self.mX + v.mX, self.mY + v.mY)
        sprite:SetPosition(self.mX + v.mX, self.mY + v.mY)
        renderer:DrawSprite(sprite)
    end

end

```

Listing 2.255: Adding Render and child support to the entity. In Entity.lua.

Listing 2.255 shows an updated Entity class with a few functions and a modified the constructor.

The updated Entity class stores an X, Y pixel position that controls where it's rendered. It contains a new mChildren table containing entities that are rendered on top of it with some offset.

The AddEntity and RemoveEntity functions use an id to add and remove children from entities. For instance, we might want a chest to sparkle but after the player interacts with it, we want to turn the sparkle off. In this case we'd use RemoveEntity.

SetTilePos updates the entity tile position and updates the X, Y pixel position to that tile.

The Render function draws the entity sprite and then draws all of its children using the X, Y positions as an offset.

SleepState

Now that we can support children, let's make the ZZZ entity. The ZZZ image can be seen in Figure 2.105. It's a short animation to represent sleep.

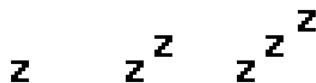


Figure 2.105: The three frames of ZZZ animation.

In EntityDefs.lua we're going to add a new definition as shown in Listing 2.256.

```

gEntities =
{
    -- code omitted
    sleep =
    {
        texture = "sleeping.png",
        width = 32,
        height = 32,
        startFrame = 1,
        x = 18,
        y = 32
    }
}

```

Listing 2.256: Adding a definition of the sleep entity, In EntityDefs.lua.

In Listing 2.256 we define the ZZZ entity def. We use this def to create an entity and add it as a child to an NPC. First we need to define the NPC. Here's the definition in Listing 2.257.

Note that the sleeper character is using a new sleep controller.

```

gCharacterStates =
{
    -- code omitted

    sleep = SleepState,
}

gCharacters =
{
    -- code omitted
    sleeper =
    {
        entity = "hero",
        anims =
        {
            left = {13},
            facing = "left",
            controller = {"sleep"},
            state = "sleep"
        }
    }
}

```

Listing 2.257: Create a sleeper controller that makes a Hero NPC snore. In EntityDefs.lua.

We need to define a SleepState to act as the controller. The SleepState file has already been added to the project, so we only need to create the class. Copy the code from Listing 2.258.

```
SleepState = {mName = "sleep"}  
SleepState.__index = SleepState  
function SleepState:Create(character, map)  
    local this =  
    {  
        mCharacter = character,  
        mMap = map,  
        mEntity = character.mEntity,  
        mController = character.mController,  
        mAnim = Animation>Create({1,2,3,4}, true, 0.6),  
    }  
  
    this.mSleepEntity = Entity>Create(gEntities.sleep),  
    -- set to default facing  
    this.mEntity:SetFrame(character.mAnims[character.mFacing][1])  
  
    setmetatable(this, self)  
    return this  
end  
  
function SleepState:Enter(data)  
    self.mEntity:AddChild("snore", self.mSleepEntity)  
end  
  
function SleepState:Render(renderer) end  
  
function SleepState:Exit()  
    self.mEntity:RemoveChild("snore")  
end  
  
function SleepState:Update(dt)  
    self.mAnim:Update(dt)  
    self.mSleepEntity:SetFrame(self.mAnim:Frame())  
end
```

Listing 2.258: The SleepState class. In SleepState.lua.

The SleepState is similar to the WaitState controller but with additional code for the ZZZ effect.

In the constructor we create a SleepEntity from the entity def we added earlier. The SleepEntity creates a sprite using the "sleeping.png" texture and breaks it up into frames. To animate the ZZZ we use an Animation object stored as mAnim in the this table. The Update loop updates the animation and sets the current SleepEntity frame. When the SleepState is entered we add the SleepEntity as a child to the character entity. When the SleepState is exited, we remove the SleepEntity.

A RunAction Operation

To add the sleeping NPC to the room we'll use an action via a new operation, RunAction. The sleeper NPC has its controller set to a SleepState object. The sleep state makes the character appear to be sleeping.

Using an action from the storyboard is a little tricky. Many actions require a reference to the current map, but when the actions are defined in the storyboard table the map may not exist! To get around this we'll allow our operation to take in a map id rather than a map object.

The RunAction instruction takes three parameters.

- Name of the action
- Table of action parameters
- Table of patch up functions

The first two parameters are straightforward. The third parameter is more interesting. When it comes time to execute the instruction, the patch up table functions are run on the params table entries. That's a bit abstract so let's look at an example, as shown in Listing 2.259.

```
Scene
{
    map = "player_house",
    focusX = 20,
    focusY = 19,
    hideHero = true,
},
RunAction("AddNPC",
    {"player_house", { def = "sleeper", x = 14, y = 19}},
    {GetMapRef}),
```

Listing 2.259: Tying the map into the storyboard actions.

The first parameter for RunAction is the action name, "AddNPC".

The second parameter is a table of parameters to be used when running the action. In this case the first parameter is the map and the second is a table with the NPC definition and tile position. Note that the first parameter is just an id to a map, not a map object.

The final parameter is a list of functions that are applied to the action parameters table. The GetMapRef function takes in a id and returns a matching map object from the storyboard. Copy the GetMapRef function into the StoryboardEvents.lua file as shown in Listing 2.260.

```
function GetMapRef(storyboard, stateId)
    local exploreState = storyboard.mStates[stateId]
    assert(exploreState and exploreState.mMap)
    return exploreState.mMap
end
```

Listing 2.260: The GetMapRef function. In StoryboardEvents.lua.

GetMapRef converts the id “player_house” from a string id to a reference to the player house map. It takes in the storyboard and a stateId. It uses the stateId to find an ExploreState in the storyboard. There’s an assert to make sure the ExploreState exists. Then it returns the map from the explore state.

The final piece missing to make all this work is the RunAction instruction itself. Copy the code from Listing 2.261 into your StoryboardEvents.lua file.

```
function RunAction(actionId, actionParams, paramOps)

    local action = Actions[actionId]
    assert(action)

    return function(storyboard)

        --
        -- Look up references required by action.
        --

        paramOps = paramOps or {}

        for k, op in pairs(paramOps) do
            if op then
                actionParams[k] = op(storyboard, actionParams[k])
            end
        end

        local actionFunc = action(unpack(actionParams))
        actionFunc()
    end
end
```

```

        return EmptyEvent
    end
end

```

Listing 2.261: The RunAction operation. In StoryboardEvents.lua.

RunAction uses the actionId to get the action from the Actions table. There's an assert to ensure the action exists. Then in the inner function the paramOps table functions are applied to the actionParams table. The results replace the values in the actionParams table. This process allows us to use the GetMapRef function we discussed earlier.

Once that action params table has been processed, the action is run. Actions occur instantly, so the EmptyEvent is returned.

We're now able to use actions directly in our cutscenes. We'll use the RunAction next to add NPCs to our map.

Adding Sleep To The Cutscene

Let's add the snoring NPC into our storyboard. Copy the code from Listing 2.262.

```

local intro =
{
    Scene
    {
        map = "player_house",
        focusX = 20,
        focusY = 19,
        hideHero = true,
    },
    BlackScreen(),
    RunAction("AddNPC",
        {"player_house", { def = "sleeper", x = 14, y = 19}},
        {GetMapRef}),
    Play("rain"),
    NoBlock(
        FadeSound("rain", 0, 1, 3)
    ),
    Caption("place", "title", "Village of Sontos"),
    Caption("time", "subtitle", "MIDNIGHT"),
    Wait(2),
    NoBlock(
        FadeOutCaption("place", 3)
    ),
    FadeOutCaption("time", 3),
}

```

```

        FadeSound("rain", 1, 0, 1),
        KillState("place"),
        KillState("time"),
        FadeOutScreen(),
        Wait(2),
        FadeInScreen(),
        Wait(0.3),
        FadeOutScreen(),
        Wait(1),
        Stop("rain")
    }
}

```

Listing 2.262: Adding a snoring NPC, using the RunAction instruction. In main.lua.

Run this code and you'll see the title text fade into our map as before, but this time there's a snoring NPC in bed! The code so far is available in dungeon-4 solution.

NPC Movement

In the introduction we see the player sleeping in bed, then there's a crash and a guard bursts into the house. To make this happen we need operations to move NPCs around the map.

To move NPCs we're going to create a new type of controller state, a way to reference NPCs by id, and a new type of action.

Example dungeon-5 contains the code so far as well as the FollowPathState.lua file.

The guard is a new character but uses the existing "walk_cycle.png" texture. Let's add his definition into EntityDefs.lua shown in Listing 2.263.

```

gCharacterStates =
{
    -- code omitted
    follow_path = FollowPathState
}

gEntities =
{
    -- code omitted
    guard =
    {
        texture = "walk_cycle.png",
        width = 16,
        height = 24,
}
}

```

```

        startFrame = 89,
        tileX = 1,
        tileY = 1,
        layer = 1
    },
}

gCharacters =
{
    -- code omitted
    guard =
    {
        entity = "guard",
        anims =
        {
            up = {81, 82, 83, 84},
            right = {85, 86, 87, 88},
            down = {89, 90, 91, 92},
            left = {93, 94, 95, 96},
        },
        facing = "up",
        controller = {"npc_stand", "follow_path", "move"},
        state = "npc_stand"
    },
}

```

Listing 2.263: Adding a definition for the Guard character. In EntityDefs.lua.

In Listing 2.263 we add the guard entity and character. There's also a new "follow_path" controller that we haven't written yet. The "follow_path" controller makes a character follow a set path. The guard texture map is laid out in the same way as the hero texture map, just offset by 80 frames.

Let's create the FollowPathState next. Copy the code from Listing 2.264 in FollowPathState.lua.

```

FollowPathState = { mName = "follow_path" }
FollowPathState.__index = FollowPathState
function FollowPathState:Create(character, map)
    local this =
    {
        mCharacter = character,
        mMap = map,
        mEntity = character.mEntity,
        mController = character.mController,
    }

```

```

}

setmetatable(this, self)
return this
end

function FollowPathState:Enter()

local char = self.mCharacter
local controller = self.mController

if char.mPathIndex == nil
or char.mPath == nil
or char.mPathIndex > #char.mPath then
char.mDefaultState = char.mPrevDefaultState or char.mDefaultState
return controller:Change(char.mDefaultState)
end

local direction = char.mPath[char.mPathIndex]
if direction == "left" then
return controller:Change("move", {x = -1, y = 0})
elseif direction == "right" then
return controller:Change("move", {x = 1, y = 0})
elseif direction == "up" then
return controller:Change("move", {x = 0, y = -1})
elseif direction == "down" then
return controller:Change("move", {x = 0, y = 1})
end

-- If we get here, there's an incorrect direction in the path.
assert(false)
end

function FollowPathState:Exit()
self.mCharacter.mPathIndex = self.mCharacter.mPathIndex + 1
end

function FollowPathState:Update(dt) end
function FollowPathState:Render(renderer) end

```

Listing 2.264: A FollowPathState. In FollowPathState.lua.

The FollowPathState expects the character to contain an mPath table and an mPathIndex. The mPath table contains a list of directions and the mPathIndex tracks the progress

through the list. The `mPath` table contains string directions "up", "down", "left" and "right". These strings tell the character which way to move.

The `FollowState` goes through the directions table switching to the `MoveState` to handle the movement for each direction. This is a crude path follower. If something interrupts the path, like a wall, then it skips to the next direction in the path. For our own cutscene needs, this kind of dumb path following is fine. Every time the `FollowPathState` exits, it increments the `mPathIndex` counter.

After the `MoveState` finishes it uses `mDefaultState` to decide which state to change to. The `mDefaultState` is usually the `WaitState`, but when we're following a path, the `MoveState` needs to switch back to the `FollowPathState`. When we tell a character to follow a path, we'll overwrite its `mDefaultState` to the `FollowPathState`. (You can see where we do this in Listing 2.265.)

When we enter the `FollowPathState` it checks if the character has finished following the path. If it has, it sets the `mDefaultState` value back to the `mPrevDefaultState`.

To handle path following, let's add a new function to `Character.lua` called `FollowPath`, as shown in Listing 2.265.

```
function Character:FollowPath(path)
    self.mPathIndex = 1
    self.mPath = path
    self.mPrevDefaultState = self.mDefaultState
    self.mDefaultState = "follow_path"
    self.mController:Change("follow_path")
end
```

Listing 2.265: Adding a follow path command. In Character.lua.

`FollowPath` assigns the path and path index to the character, sets the default state to "follow_path", and stores the previous value in `mPrevDefaultState`, then changes the state to "follow_path".

NPC IDs

Next let's add code to allow NPCs to be uniquely identified. When we create an NPC we'll give it an id. NPCs are stored in the map by id for easy look up. First make a change in `Map.lua` as in Listing 2.266.

```
function Map>Create(mapDef)
    local layer = mapDef.layers[1]
    local this =
    {
        -- code omitted
```

```

        mNPCs = {},
        mNPCbyId = {}
    }
}

```

Listing 2.266: Adding an NPC look-up table in the map. In Map.lua.

Now that the table exists in the Map, we need a way to fill it up with NPCs. Listing 2.267 shows an updated action to add an NPC to a map.

```

AddNPC = function(map, npc)
    return function(trigger, entity)

        -- code omitted

        assert(map.mNPCbyId[npc.id] == nil)
        char.mId = npc.id
        map.mNPCbyId[npc.id] = char
    end
end

```

Listing 2.267: Updating AddNPC. In Actions.lua.

The AddNPC function sets the character `mId` field. The `mId` is used as a key to add the npc to the map `mNPCbyId` table.

In Listing 2.268 we add a new operation, `MoveNPC`, and a new state, `BlockUntilEvent`.

```

BlockUntilEvent = {}
BlockUntilEvent.__index = BlockUntilEvent
function BlockUntilEvent:Create(UntilFunc)
    local this =
    {
        mUntilFunc = UntilFunc,
    }
    setmetatable(this, self)
    return this
end

function BlockUntilEvent:Update(dt) end
function BlockUntilEvent:Render() end

function BlockUntilEvent:IsBlocking()
    return not self.mUntilFunc()
end

```

```

function BlockUntilEvent:IsFinished()
    return not self:IsBlocking()
end

function MoveNPC(id, mapId, path)
    return function(storyboard)
        local map = GetMapRef(storyboard, mapId)
        local npc = map.mNPCbyId[id]
        npc:FollowPath(path)
        return BlockUntilEvent>Create(
            function()
                return npc.mPathIndex > #npc.mPath
            end)
    end
end

```

Listing 2.268: Creating a MoveNPC event. In StoryboardEvents.lua.

You might be able to guess what the BlockUntilEvent does from the name! It blocks the storyboard progress until its passed-in function returns true. Once BlockUntilEvent stops blocking, it's removed from the storyboard.

The MoveNPC operation needs to block the storyboard until the NPC has finished moving. To achieve this it pushes a BlockUntilEvent onto the storyboard's internal stack.

MoveNPC takes 3 parameters: the NPC id, the map id, and the path for the NPC to follow. When the instruction is executed, it looks up the map and npc and tells the npc to follow the path. It creates a BlockUntilEvent with a function that checks whether the NPC has finished following the path. We know the NPC has followed the path when its pathIndex is greater than the size of the path.

Let's add the MoveNPC operation to the end of our storyboard as shown in Listing 2.269.

```

local intro =
{
    -- code omitted
    FadeOutScreen(),
    Wait(2),
    FadeInScreen(),
    RunAction("AddNPC",
        {"player_house", { def = "guard", id = "guard1", x = 19, y = 22}},
        {GetMapRef}),
    MoveNPC("guard1", "player_house",
    {
        "up", "up", "up",
        "left", "left", "left",

```

```

        }),
        Wait(0.3),
        FadeOutScreen(),
        Wait(2),
        FadeSound("rain", 1, 0, 1),
        Stop("rain")
    }
}

```

Listing 2.269: Adding movement to the intro cutscene. In main.lua.

Run the code and you'll see the intro storyboard play as before but with some new events near the end. When the screen fades to black, a guard NPC is added called "guard1", and then it walks from somewhere near the door towards the bed.

NPC Speech

We've added some movement to our cutscenes. Next let's add speech. When the guard gets to the bed he should say something. To make him speak we'll use a new Say operation. Speech is handled using a dialog box.

Our storyboard isn't interactive, therefore the player has no way to dismiss a dialog box. To deal with this we'll create and use a TimedTextboxEvent. The TimedTextboxEvent displays a dialog box for a certain amount of time and then removes it. Check out the code in Listing 2.270.

```

TimedTextboxEvent = {}
TimedTextboxEvent.__index = TimedTextboxEvent
function TimedTextboxEvent:Create(box, time)
    local this =
    {
        mTextbox = box,
        mCountDown = time
    }
    setmetatable(this, self)
    return this
end

function TimedTextboxEvent:Update(dt, storyboard)
    self.mCountDown = self.mCountDown - dt
    if self.mCountDown <= 0 then
        self.mTextbox:OnClick()
    end
end
function TimedTextboxEvent:Render() end

```

```

function TimedTextboxEvent:IsBlocking()
    return self.mCountDown > 0
end

function TimedTextboxEvent:IsFinished()
    return not self:IsBlocking()
end

function Say(mapId, npcId, text, time, params)

    time = time or 1
    params = params or {textScale = 0.8}

    return function(storyboard)
        local map = GetMapRef(storyboard, mapId)
        local npc = map.mNPCById[npcId]
        local pos = npc.mEntity.mSprite:GetPosition()
        storyboard.mStack:PushFit(
            gRenderer,
            -map.mCamX + pos:X(), -map.mCamY + pos:Y() + 32,
            text, -1, params)
        local box = storyboard.mStack:Top()
        return TimedTextboxEvent>Create(box, time)
    end
end

```

Listing 2.270: Adding TimedTextboxEvent and Say instruction. In StoryBoardEvents.lua.

The TimedTextboxEvent takes in a textbox and a time in seconds. It displays the textbox for that number of seconds and then closes it. This way, textboxes can be used in cutscenes and dismissed automatically.

The Say operation takes in a mapId, npcId, text and time. It uses the mapId and npcId to get the npc from the map. The textbox is positioned 32 pixels above the NPC. The map's mCamX and mCamY variables are used to make sure the textbox aligns correctly with the current map translation. The time parameter is optional and defaults to 1 second. The text is the text displayed in the box, and time controls how long it's shown.

The Say operation takes in one last optional parameter, a params table. This table is passed through to the textbox constructor. If no params table exists, a default one is used which sets the textbox font scale to 0.8. This scale works nicely for the font we're using.

Once the Say operation creates the textbox, it uses it to create and return a TimedTextboxEvent. This event blocks the storyboard while the textbox is displayed.

Let's update the intro cutscene with some dialog! The latest storyboard table is shown in Listing 2.271.

```
local intro =
{
    Scene
    {
        map = "player_house",
        focusX = 14,
        focusY = 19,
        hideHero = true,
    },
    BlackScreen(),
    RunAction("AddNPC",
        { "player_house",
            { def = "sleeper", id="sleeper", x = 14, y = 19},
            {GetMapRef}),
        Play("rain"),
        NoBlock(
            FadeSound("rain", 0, 1, 3)
        ),
        Caption("place", "title", "Village of Sontos"),
        Caption("time", "subtitle", "MIDNIGHT"),
        Wait(2),
        NoBlock(
            FadeOutCaption("place", 3)
        ),
        FadeOutCaption("time", 3),

        KillState("place"),
        KillState("time"),
        FadeOutScreen(),
        Wait(2),
        FadeInScreen(),
        RunAction("AddNPC",
            {"player_house", { def = "guard", id = "guard1", x = 19, y = 22}},
            {GetMapRef}),
        NoBlock(FadeOutScreen()),
        MoveNPC("guard1", "player_house",
            {
                "up", "up", "up",
                "left", "left", "left",
            }),
        Wait(0.3),
        Say("player_house", "guard1", "Take Him!"),
        FadeInScreen(),
    }
}
```

```

        FadeSound("rain", 1, 0, 1),
        Stop("rain")
    }

```

Listing 2.271: The storyboard making use of NPC movement and textboxes. In main.lua.

Run the code and you'll see the guard appear and kidnap the player! Example dungeon-5-solution contains the code so far.

Swapping Maps

For the final part of the cutscene we switch the map from the player's house to a jail cell. Example dungeon-6 contains the code so far, and new assets map_jail.lua, dungeon_tiles.png, door_break.wav, wagon.wav, wind.wav, and bell.wav. We'll use the sound files during the cutscene.

Before we can change to a new map, it needs an entry in the MapDB table. Copy the code from Listing 2.272.

```

MapDB =
{
    ["player_house"] = CreateHouseMap,
    ["jail"] = CreateJailMap
}

```

Listing 2.272: Adding the jail map to the MapDB. In MapDB.lua.

Potentially loading a map can take a long time. Triggers fire off, NPCs are loaded, etc., but because our maps are all quite small, we're not going to worry about this. The ReplaceScene operation finds the ExploreState, loads the new map, and replaces the existing one.

Copy the code from Listing 2.273.

```

function ReplaceScene(name, params)
    return function(storyboard)
        local state = storyboard.mStates[name]

        -- Give the state an updated name
        local id = params.name or params.map
        storyboard.mStates[name] = nil
        storyboard.mStates[id] = state
    end
end

```

```

local mapDef = MapDB[params.map]()
state.mMap = Map:Create(mapDef)

state.mMap:GotoTile(params.focusX, params.focusY)
state.mHero = Character>Create(gCharacters.hero, state.mMap)
state.mHero.mEntity:SetTilePos(
    params.focusX,
    params.focusY,
    params.focusZ or 1,
    state.mMap)

if params.hideHero then
    state:HideHero()
else
    state>ShowHero()
end

return NoBlock(Wait(0))()
end
end

```

Listing 2.273: ReplaceScene instruction. In Storyboard events. In StoryboardEvents.lua.

ReplaceScene is similar to the Scene operation but it doesn't create an ExploreState. Instead it finds the existing ExploreState and replaces its map object.

When the ReplaceScene operation executes, it finds the ExploreState matching the name parameter. The storyboard keeps track of all the states by id. Once we find the ExploreState we remove it from the Storyboard and put it back using an updated id. The new scene name is generated in the same way as in the Scene instruction; if there's a name field then that's used, if not then the name of the map to be loaded is used.

The operation creates a new map using the mapDef from the MapDB table. The new map replaces the existing map in the ExploreState.

Next the camera and hero position are set. The quickest way to add the hero to a map is to recreate him, so that's what we do. If the hideHero flag is true the hero is hidden. Finally a non-blocking wait event is returned much like the Scene instruction.

Let's try this operation out with a simplified storyboard. Copy the code from Listing 2.274 into your main.lua.

```

intro =
{
    Scene
    {

```

```

        map = "player_house",
        focusX = 14,
        focusY = 19,
        hideHero = true,
    },
    Wait(3),
    ReplaceScene(
        "player_house",
    {
        map = "jail",
        focusX = 56,
        focusY = 11,
        hideHero = false
    }),
    Wait(3)
}

```

Listing 2.274: Testing the ReplaceScene instruction is a simple storyboard. In main.lua.

The storyboard, in Listing 2.274, loads the first scene, waits 3 seconds, replaces it with the second scene, waits 3 more seconds, and ends.

Run the code and you'll see the scene change from the house to the jail.

Creating a cutscene with good flow and timing is quite hard, and the more time you put in, the better the result. Before moving on to finish the intro cutscene, we're going to take a little time to flesh out the story.

Listing 2.275 shows our rough and ready cutscene described at the start of this section. There have been a few more Wait operations added and more sound effects. It's also using a modified Say instruction.

```

local intro =
{
    Scene
    {
        map = "player_house",
        focusX = 14,
        focusY = 19,
        hideHero = true,
    },
    BlackScreen(),
    RunAction("AddNPC",
        {"player_house",
        { def = "sleeper", id="sleeper", x = 14, y = 19}},

```

```

        {GetMapRef}),
Play("rain"),
NoBlock(
    FadeSound("rain", 0, 1, 3)
),
Caption("place", "title", "Village of Sontos"),
Caption("time", "subtitle", "MIDNIGHT"),
Wait(2),
NoBlock(
    FadeOutCaption("place", 3)
),
FadeOutCaption("time", 3),

KillState("place"),
KillState("time"),
FadeOutScreen(),
Wait(2),
FadeInScreen(),
RunAction("AddNPC",
    {"player_house", { def = "guard", id = "guard1", x = 19, y = 22}},
    {GetMapRef}),
Wait(1),
Play("door_break"),
NoBlock(FadeOutScreen()),
MoveNPC("guard1", "player_house",
{
    "up", "up", "up",
    "left", "left", "left",
}),
Wait(1),
Say("player_house", "guard1", "Found you!", 2.5),
Wait(1),
Say("player_house", "guard1", "You're coming with me.", 3),
FadeInScreen(),

-- Kidnap
NoBlock(Play("bell")),
Wait(2.5),
NoBlock(Play("bell", "bell2")),
FadeSound("bell", 1, 0, 0.2),
FadeSound("rain", 1, 0, 1),
Play("wagon"),
NoBlock(
    FadeSound("wagon", 0, 1, 2)
),

```

```

Play("wind"),
NoBlock(
    FadeSound("wind", 0, 1, 2)
),
Wait(3),
Caption("time_passes", "title", "Two days later..."),
Wait(1),
FadeOutCaption("time_passes", 3),
KillState("time_passes"),
NoBlock(
    FadeSound("wind", 1, 0, 1)
),
NoBlock(
    FadeSound("wagon", 1, 0, 1)
),
Wait(2),
Caption("place", "title", "Unknown Dungeon"),
Wait(2),
FadeOutCaption("place", 3),
KillState("place"),

ReplaceScene(
    "player_house",
    {
        map = "jail",
        focusX = 56,
        focusY = 11,
        hideHero = false
    }),
FadeOutScreen(),
Wait(0.5),
Say("jail", "hero", "Where am I?", 3),
Wait(3),
}

```

Listing 2.275: A fleshed out storyboard. In main.lua.

The second to last instruction has the hero speaking but the current Say code only works for NPCs. The current Say operation looks up an NPC in the map's NPC table so it can position the textbox. The hero isn't in that table and therefore needs a special case. Copy the updated code from Listing 2.276.

```

function Say(mapId, npcId, text, time, params)

    time = time or 1

```

```

params = params or {textScale = 0.8}

return function(storyboard)
    local map = GetMapRef(storyboard, mapId)
    local npc = map.mNPCbyId[npcId]
    if npcId == "hero" then
        npc = storyboard.mStates[mapId].mHero
    end

```

Listing 2.276: Letting the hero speak. In StoryboardEvents.lua.

In Listing 2.276 the updated Say function checks if the npcId equals “hero”, and if so we return the hero from the current map. Otherwise the Say operation works as before.

Potentially there’s now a problem if we call an NPC “hero” so we’ll add a check for that in the AddNPC instruction. Here’s the change Listing 2.277.

```

AddNPC = function(map, npc)
    assert(npc.id ~= "hero") -- reserved npc name

```

Listing 2.277: Safety check when adding an NPC. In Actions.lua.

Run the code and you’ll see a cutscene that’s looking pretty good! Example dungeon-solution-6 contains a complete version of the code.

To continue on with the game we need to stop playing the cutscene and return control to the player.

Returning Control

When the cutscene ends, control should return to the player.

At the end of our cutscene, the hero is in jail. The ExploreState contains the jail map. It’s sitting on the Storyboard internal stack. When the cutscene ends, we need to pull the ExploreState out of the Storyboard and push it onto the main stack.

A final instruction called HandOff moves the ExploreState out of the storyboard and onto the main stack.

Example dungeon-7 contains the code so far. Copy the code from Listing 2.278 into the StoryboardEvents.lua file.

```

function HandOff(mapId)
    return function(storyboard)
        local exploreState = storyboard.mStates[mapId]

```

```

    -- remove storyboard from the top of the stack
    storyboard.mStack:Pop()
    storyboard.mStack:Push(exploreState)
    exploreState.mStack = gStack
    return EmptyEvent
end
end

```

Listing 2.278: The HandOff instruction, ends the cutscene and pushes an ExploreState from the cutscene= on top of the main statestack. In StoryboardEvents.lua.

In Listing 2.278 the HandOff operation pops the storyboard off the stack, takes the ExploreState, and pushes it onto the stack. We also update the ExploreState's mStack reference to point to the global stack where it now lives.

Add this instruction to the bottom of cutscene as shown in Listing 2.279.

```

local intro =
{
    -- code omitted

    HandOff("jail")
}

```

Listing 2.279: Adding Handoff. In main.lua.

Example dungeon-solution-7 contains this code. Run the code. After the cutscene plays and control is returned to the player, the character can be moved around to start exploring the jail.

The storyboard we've built is a powerful abstraction, but it can always be made better! We're going to make one last set of changes to the storyboard to help organize the code.

Storyboard Namespace

While making this first cutscene, we've created a *lot* of operations. The Storyboard class is powerful but we're also skirting close to some dangerous territory.

All the operations are global and they use *very common names* like "Play" and "Stop". These are variables names with a high potential to clash. For instance, in the future we might be doing some animation and make a Play or Stop function; suddenly everything breaks because two things are trying to use the same name!

Therefore we're going to put all our instructions into a table called SOP, short for *storyboard operation*. Usually I try to avoid abbreviating names but in this case we'll be using it so much the length really affects readability.

Example dungeon-8 contains all the code we've written so far, but with an empty table `SOP = {}` defined at the top of `StoryboardEvents.lua`. All the operations are defined in this table as demonstrated in Listing 2.280.

```
-- Definition
SOP.MoveNPC = ...

-- Use
local intro =
{
    SOP.Scene ...
    SOP.BlackScreen ...
```

Listing 2.280: SOP Namespace example. In StoryboardEvents.lua and main.lua.

This makes the story scripts a little noisier but helps avoid potentially irritating bugs in the future!

Action

We're ready explore the prison map in Figure 2.106. The prison cell is quite small but there are two main places of interest, the grating near the top and the crumbling wall over on the right.

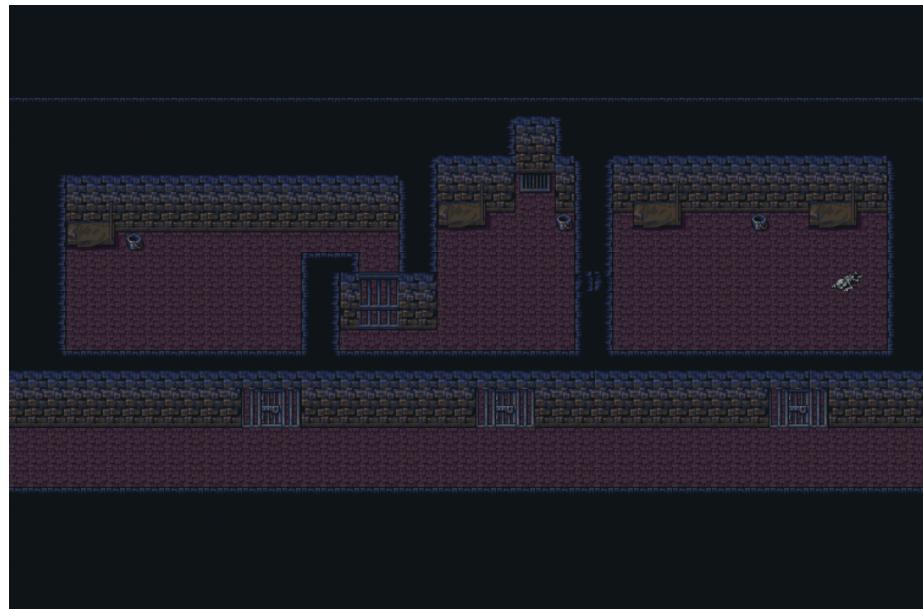


Figure 2.106: The jail map.

Breaking Through The Wall

Example dungeon-9 contains the code so far and the sound file crumble.wav. We're going to let the player break through the prison wall into the next cell. We'll play the crumble sound when the wall is destroyed.

To allow the player to push through the wall to the right we'll use a trigger. This trigger runs a custom Lua function when the player tries to use it.

Our game is small, so we don't need a generic break-wall action. For the most part we'll directly write the Lua code we need and call it via a special RunScript action. Copy the definition from Listing 2.281 into your Actions.lua file.

```
Actions =
{
    -- code omitted
    RunScript = function(map, Func)
        return function(trigger, entity)
            Func(map, trigger, entity)
        end
    end
}
```

Listing 2.281: A RunScript action. In Actions.lua.

This is a generic action to fire off some bespoke Lua code. This type of action isn't very reusable but it gets the job done.

On the jail map, the tile containing the weak wall is at position 35, 22. This is where we'll add the first trigger.

When the user tries to interact with the wall, we're going to display a dialog box that says "The wall is crumbling. Do you want to push it?" If the player says "No", the dialog disappears and nothing changes. If the player says "Yes", we pop up a new dialog saying "The wall crumbles", play the crumble sound effect, change the wall tile, and make it walkable. We also remove the trigger, as we don't ever want to trigger it again.

To make this work we need to extend the map, so we can change tiles and remove triggers. Copy these changes from Listing 2.282.

```
function Map:WriteTile(params)
    local layer = params.layer or 1
    local detail = params.detail or 0

    -- Each layer is of 3 parts
    layer = ((layer - 1) * 3) + 1

    local x = params.x
    local y = params.y
    local tile = params.tile
    local collision = self.mBlockingTile
    if not params.collision then
        collision = 0
    end

    local index = self:CoordToIndex(x, y)
    local tiles = self.mMapDef.layers[layer].data
    tiles[index] = tile

    -- Detail
    tiles = self.mMapDef.layers[layer + 1].data
    tiles[index] = detail

    -- Collision
    tiles = self.mMapDef.layers[layer + 2].data
    tiles[index] = collision

end
```

```

function Map:RemoveTrigger(x, y, layer)
    layer = layer or 1
    assert(self.mTriggers[layer])
    local triggers = self.mTriggers[layer]
    local index = self:CoordToIndex(x, y)
    assert(triggers[index])
    triggers[index] = nil
end

```

Listing 2.282: WriteTile alters the map by changing a tile and RemoveTrigger adds support for removing triggers from the map. In Map.lua.

The WriteTile function takes in a params table that tells it which tile to overwrite. The params table may contain the following fields:

- *x, y* - the coordinates for the tile to overwrite.
- *layer* - the layer to overwrite the tile. Optional. Defaults to layer 1.
- *tile* - the tile id to write.
- *collision* - collision flag for the tile. Either true or false. Optional. Defaults to false.
- *detail* - tile id to write to the detail layer. Optional. Defaults to 0 (no detail information).

These fields are used to write a new tile to the map. The tile field is a number indicating the new tile graphic. The WriteTile function isn't used for making massive changes to the map but it's useful for revealing hidden doors, blowing up walls, and that type of thing.

The RemoveTrigger function takes in an *x, y* and a *layer* parameter. It then removes any trigger at that location. The code is simple. It finds the layer, then finds the trigger at the *x, y* position on the layer and sets the value to nil.

Ok. Now we're ready to create a trigger to crumble the dungeon wall. Copy the code from Listing 2.283.

```

function CreateJailMap()

    local CrumbleScript =
        function(map, trigger, entity, x, y, layer)

            local OnPush
            local dialogParams =
            {
                textScale = 1,
                choices =

```

```

{
    options =
    {
        "Push the wall",
        "Leave it alone"
    },
    OnSelection = function(index)
        if index == 1 then
            OnPush(map)
        end
    end
},
}

gStack:PushFit(gRenderer,
    0, 0,
    "The wall here is crumbling. Push it?",
    255,
    dialogParams)

-- 2. OnYes erase wall
OnPush = function(map)
    -- The player's pushing the wall.
    gStack:PushFit(gRenderer, 0, 0,
        "The wall crumbles.",
        255)
    Sound.Play("crumble")

    map:RemoveTrigger(x, y, layer)
    map:WriteTile
    {
        x = x,
        y = y,
        layer = layer,
        tile = 134,
        collision = false
    }
end
end

```

Listing 2.283: The CrumbleScript which makes a wall crumble in the jail. In map_jail.lua.

The Listing 2.283 shows the first part of the CreateJailMap function where the CrumbleScript is defined. Let's take it line by line and see what's happening.

The CrumbleScript takes in the map, trigger, entity activating the trigger, and a set of coordinates for which tile to change. The first line is local OnPush. It's not assigned to anything. It's just telling Lua that we're going to be using this variable. It's assigned a value at the bottom of the function, but we reference it earlier than that in the callback.

Next we create a table called dialogParams describing a textbox. We use dialogParams when creating the textboxes to tell the player that the wall looks like it's crumbling and to ask them if they want to push it. The dialogParams sets the text size to 1. It also contains two choices to display to the player:

1. "Push the wall"
2. "Leave it alone"

There's an OnSelection callback, called when the player makes a choice. It checks whether the index is 1, "Push the wall", and if it is, it calls OnPush, passing through the map.

With dialogParams defined, we use it to create a fitted textbox and push that onto the stack. The textbox is positioned at 0, 0, which is the center of the screen. The text is set to "The wall here is crumbling. Push it?" and the text width is set to 255 pixels.

After the user pushes the wall, what should happen? Well, the wall should fall down and let the player pass through. The OnPush function makes this happen.

When OnPush is called it creates a new textbox with the text "The wall crumbles.", plays the crumble sound, and removes the trigger so we can't interact with the wall once it's been destroyed. Finally the crumbling wall tile id is set to 134 (the floor tile index) and it's set to walkable.

That's the CrumbleScript defined. Let's hook it into the map with a trigger. Copy the code from Listing 2.284 to update the jail map definition.

```
return
{
    -- code omitted
    actions =
    {
        break_wall_script =
        {
            id = "RunScript",
            params = { CrumbleScript }
        }
    },
    trigger_types =
    {
        cracked_stone = { OnUse = "break_wall_script" }
    },
    triggers =
```

```

{
    { trigger = "cracked_stone", x = 60, y = 11},
},

```

Listing 2.284: Adding a trigger to let the player push through a crumbling wall. In map_jail.lua.

In the map def's action table we add a new action called "break_wall_script". It's a RunScript action that calls our newly defined CrumbleScript function. In the trigger_types table we add a new type of trigger, cracked_stone, that fires the "break_wall_script" action when the player uses the trigger. Finally the trigger is placed in the scene on the cracked wall tile at X: 60, Y: 11 as shown in Figure 2.107.

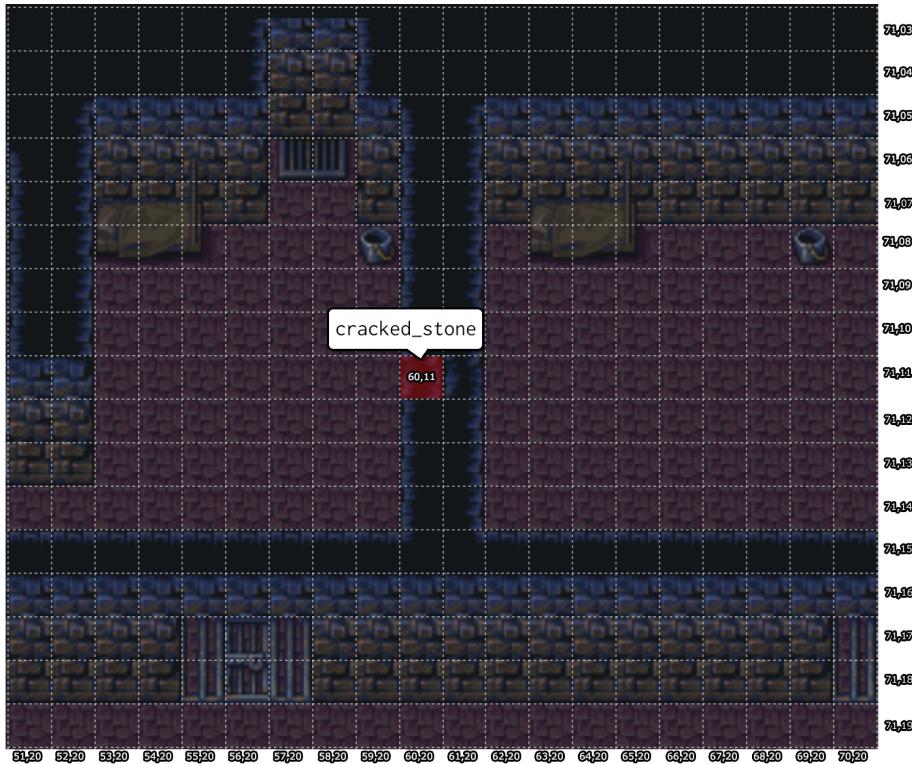


Figure 2.107: The cracked_stone trigger on the jail map.

Example dungeon-9-solution contains the complete code. Run it and the cutscene will play, then you're given control. Go up to the wall, face it, and press space. It will run

the code we've just written. You'll be able to push through the wall and enter the next cell or leave it alone. You can see what it should look like in Figure 2.108.

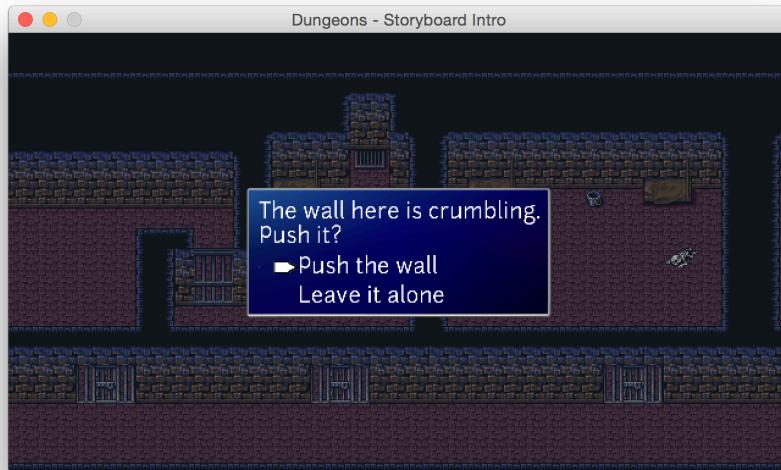


Figure 2.108: The CrumbleScript in action on the Jail map.

Picking Up A Bone

Once the player pushes down the wall, they're free to explore the neighboring cell. There's a skeleton in the cell, and the player can take a bone from it. We will add this interaction with a RunAction trigger that calls some specifically written code.

Example dungeon-10 contains the code so far and two new sound effects, skeleton_destroy.wav and key_item.wav. The bone from the skeleton is a *key item*. To use key items we need to set up the world object as shown in Listing 2.285.

```
gRenderer = Renderer.Create()
gStack = StateStack.Create()
gWorld = World.Create()
gIcons = Icons.Create(Texture.Find("inventory_icons.png"))

-- code omitted

function update()

    -- code omitted
```

```
    gWorld:Update(dt)
end
```

Listing 2.285: Creating the World object to enable use of the inventory. In main.lua.

Run the game and you'll be able to bring up the menu and browse the inventory by pressing alt.

When the user inspects the skeleton, we'll add a bone key item to the inventory. We already have an entry in the ItemDB called "Old bone", and by default the user already has it in their inventory. We'll need to change that! Update your world class to match Listing 2.286.

```
World = {}
World.__index = World
function World:Create()
    local this =
    {
        mTime = 0,
        mGold = 0,
        mItems = {},
        mKeyItems = {},
    }
    setmetatable(this, self)
    return this
end
```

Listing 2.286: Clearing the inventory. In World.lua.

In Listing 2.286 the `mItems` and `mKeyItems` tables have both been set to the empty table. This means the player has no items when starting.

The trigger on the skeleton displays a dialog box that says "The skeleton crumbles into dust", follow by the text "You found a bone". With the current code, it's hard to chain up dialog boxes like this.

To make chaining dialog boxes easier we'll add an "OnFinish" callback to the textbox params table. This function gets called when the textbox state is exited. Copy the code from Listing 2.287.

```
function StateStack:PushFix(renderer, x, y, width, height, text, params)
    -- code omitted

    local textbox = Textbox>Create
    {
```

```

    -- code omitted
    OnFinish = params.OnFinish,
    stack = self,
}
table.insert(self.mStates, textbox)
end

```

Listing 2.287: Adding an extra callback when creating a textbox. In StateStack.lua.

Next let's make textbox report when it's been closed, as in Listing 2.288.

```

function Textbox>Create(params)

    -- code omitted

    local this =
    {
        -- code omitted

        mOnFinish = params.OnFinish or function() end
    }

    -- code omitted
end

function Textbox:Exit()
    if self.mDoClickCallback then
        self.mSelectionMenu:OnClick()
    end
    if self.mOnFinish then
        self.mOnFinish()
    end
end

```

Listing 2.288: Modify the textbox to notify us when it dies. In Textbox.lua.

Now we're ready to set up the skeleton trigger. The skeleton is already part of the map, taking up two tiles, X:73 Y:11 and X:74 Y:11. We'll add the trigger to both these tiles.

When the player interacts with the skeleton trigger, we'll call the function defined in Listing 2.289.

```

function CreateJailMap()

    local BoneItemId = 4

    local BoneScript =
        function(map, trigger, entity, x, y, layer)

            local GiveBone = function()
                gStack:PushFit(gRenderer, 0, 0,
                               'Found key item: "Calcified bone"',
                               255,
                               {textScale=1})
                gWorld:AddKey(BoneItemId)
                Sound.Play("key_item")
            end

            -- 1. The skeleton collapsed into dust
            gStack:PushFit(gRenderer, 0, 0, "The skeleton collapsed into dust.",
                           255, {textScale=1, OnFinish=GiveBone})

            Sound.Play("skeleton_destroy")
            map:RemoveTrigger(73, 11, layer)
            map:RemoveTrigger(74, 11, layer)
            map:WriteTile
            {
                x = 74,
                y = 11,
                layer = layer,
                tile = 136,
                collision = true
            }
        end
    end

```

Listing 2.289: BoneScript, the code called when the player uses the skeleton. In map_jail.lua.

The BoneScript function in Listing 2.289 is added at the top of CreateJailMap.

Before the function definition we set a variable, BoneItemId, to 4. This is the id of the bone key item. We use the id in a few callbacks so it's defined outside of the BoneScript function.

The first part of the BoneScript function defines a callback called GiveBone. GiveBone is called after the first textbox is dismissed. The GiveBone function pushes a new textbox

onto the stack to tell the player they've found an item. It also plays the "found a key item sound" and adds the item to the player inventory.

After defining the callback, a textbox is pushed onto the stack that says "The skeleton collapsed into dust". The params table for this textbox contains an OnFinish callback that's set to the GiveBone function. You can see the callback visualized in Figure 2.109. When the player closes the textbox, the GiveBone function runs.

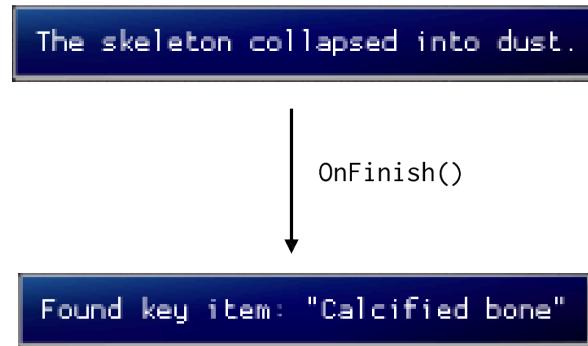


Figure 2.109: Using the OnFinish to trigger a second callback.

After pushing the first textbox onto the stack, the sound of the skeleton being destroyed is played. Any triggers on the skeleton tiles are removed and the tile with the skeleton's leg is replaced so it only shows a single leg. That's everything we need from the skeleton interaction. Copy the code from Listing 2.290 to hook the BoneScript into the map.

```
return
{
    -- code omitted

    actions =
    {
        -- code omitted
        bone_script =
        {
            id = "RunScript",

```

```

        params = { BoneScript }
    }
},
trigger_types =
{
    cracked_stone = { OnUse = "break_wall_script" },
    skeleton = { OnUse = "bone_script" }
},
triggers =
{
    { trigger = "cracked_stone", x = 35, y = 22 },
    { trigger = "skeleton", x = 73, y = 11 },
    { trigger = "skeleton", x = 74, y = 11 },
},

```

Listing 2.290: Code to add the skeleton trigger and actions into the map. In map_jail.lua.

In Listing 2.290, triggers are added to both tiles of the skeleton. This lets the player activate the trigger from the front or back part of the skeleton; it doesn't matter.

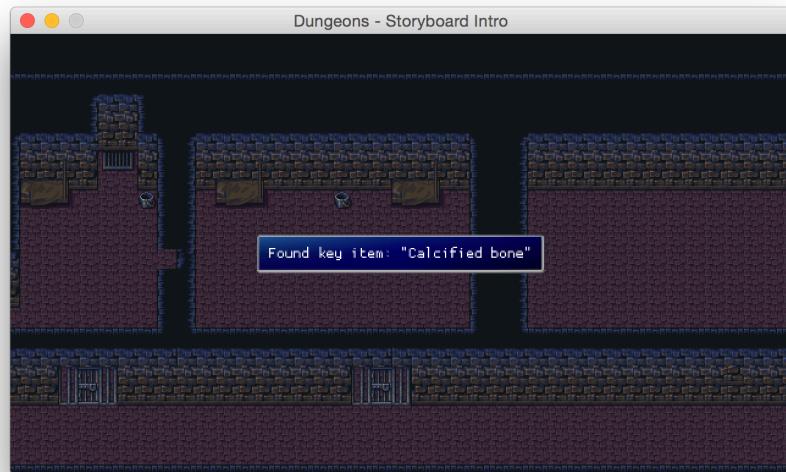


Figure 2.110: The player finds a useful bone from an old skeleton.

Example dungeon-10-solution contains the code so far. Run it. After the cutscene that you're probably sick of (you can comment a lot of it out to get into the game quicker),

you'll be able to break through the wall, destroy the skeleton, and get the bone. Figure 2.110 shows the player finding the bone. Also you'll be able to check the bone out in your inventory!

An Interested Prisoner

When the player returns to his cell with the bone, the prisoner in the next cell over will move to an adjoining door, close enough to interact with.

The code so far is available in dungeon-11.

To add another prisoner, we need a new NPC definition in the EntityDefs.lua file. The prisoner definition is shown below in Listing 2.291.

```
gEntities =
{
    -- code omitted
    prisoner =
    {
        texture = "walk_cycle.png",
        width = 16,
        height = 24,
        startFrame = 58,
        tileX = 1,
        tileY = 1,
        layer = 1
    },
}

gCharacters =
{
    -- code omitted
    prisoner =
    {
        entity = "prisoner",
        anims =
        {
            up = {49, 50, 51, 52},
            right = {53, 54, 55, 56},
            down = {57, 58, 59, 60},
            left = {61, 62, 63, 64},
        },
        facing = "down",
        controller = {"npc_stand", "follow_path", "move"},
        state = "npc_stand"
    }
}
```

```
    },
}
```

Listing 2.291: Adding the prisoner. In EntityDefs.lua.

The prisoner is very much like the other characters we previously defined. Note the prisoner's states. He can stand, follow a path, and move.

Let's add the prisoner to the map. This requires opening jail_map.lua, finding the NPC table, and adding an entry like the one shown in Listing 2.292.

```
function CreateJailMap()

-- code omitted

return
{
    -- code omitted
    on_wake =
    {
        {
            id = "AddNPC",
            params = {{ def = "prisoner", id = "gregor", x = 44, y = 12 }},
        }
    },
}
```

Listing 2.292: Adding a prisoner to the neighboring cell. In map_jail.lua.

Run the game and the prisoner will appear next door. Tantalizing but inaccessible, as shown in Figure 2.111.

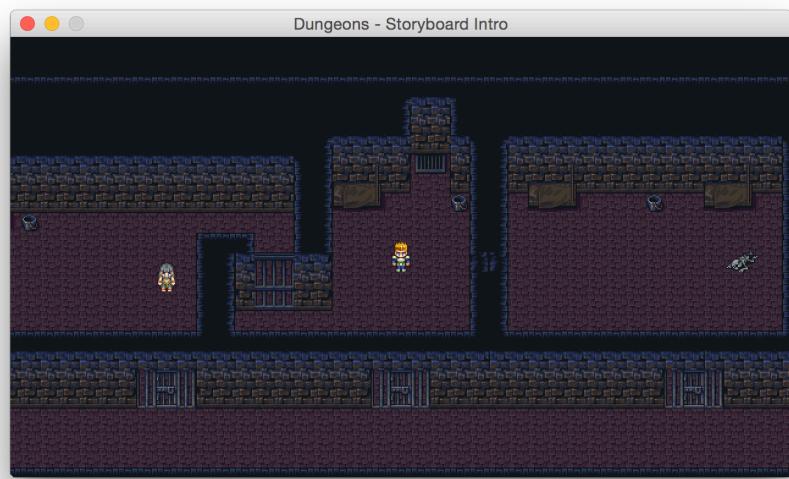


Figure 2.111: Adding a prisoner to the next cell.

The definition shown in Listing 2.292 gives the prisoner the unique id of “gregor”. We use this name when telling him to move.

When the player carries the bone back through the broken wall, we want Gregor to move over to the prison bars. The movement is shown in Figure 2.112. We want it to happen at this point, so the player can see that Gregor is moving and hopefully become intrigued and go over to talk to him.

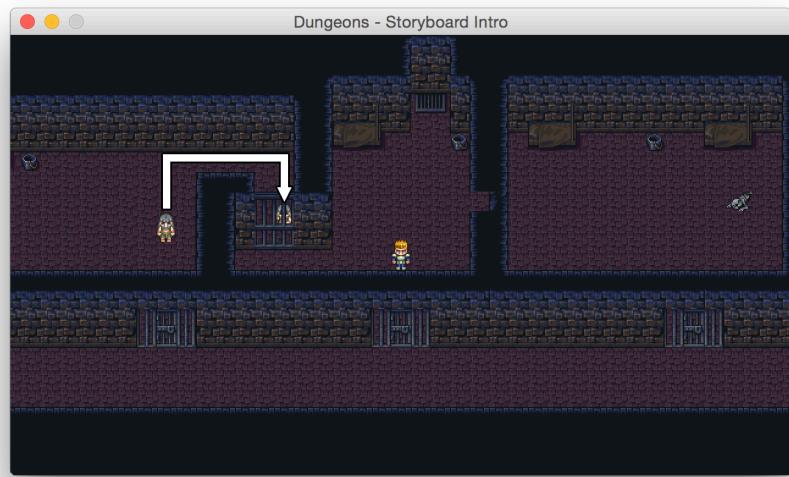


Figure 2.112: Moving the prisoner closer to the player.

Let's add a trigger before the wall that fires when its OnExit trigger is activated. The code only executes if the player is carrying the key. When the trigger is fired it calls a function, MoveGregor, which is shown in Listing 2.293.

```
function CreateJailMap()

    local BoneItemId = 4

    local MoveGregor =
        function(map, trigger, entity, x, y, layer)

            if gWorld:HasKey(BoneItemId) then
                local gregor = map.mNPCbyId["gregor"]
                assert(gregor) -- should always exist!
                gregor:FollowPath
                {
                    "up",
                    "up",
                    "up",
                    "right",
                    "right",
                    "right",
                    "right",
                    "right"
                }
            end
        end
    end
```

```

        "right",
        "right",
        "right",
        "down",
        "down",
        "down",
    }
    map:RemoveTrigger(x, y, layer)
end
end

-- code omitted

```

Listing 2.293: The MoveGregor script tells Gregor to move to the bars. In map_jail.lua.

If the player doesn't have the bone, the function in Listing 2.293 does nothing. If the player does have the bone item, the script tells Gregor to follow a path and then removes itself from the map, so it can't be triggered again.

Let's add the trigger to the map definition, as shown in Listing 2.294.

```

actions =
{
    -- code omitted
    move_gregor =
    {
        id = "RunScript",
        params = { MoveGregor }
    }
},
trigger_types =
{
    -- code omitted
    gregor_trigger = { OnExit = "move_gregor" }
},
triggers =
{
    -- code omitted
    { trigger = "gregor_trigger", x = 59, y = 11 },
}

```

Listing 2.294: Adding the move gregor trigger to the dungeon map. In map_jail.lua.

The trigger uses `OnExit` instead of the `OnUse` callback, only firing when the user exits the tile in front of the collapsed wall. Run the code and you'll see Gregor move to the prison bars when you bring the bone back to your cell.

To let the player talk to Gregor, we need another trigger. The bars stop the player from getting right next to Gregor, but we can put a trigger on the tile in front of him. Copy the TalkGregor script from Listing 2.295 into the CreateJailMap function.

```
local TalkGregor =
function(map, trigger, entity, x, y, layer)
    local gregor = map.mNPCbyId["gregor"]
    if gregor.mEntity.mTileX == x and
        gregor.mEntity.mTileY == y - 1 then

        gregor.mTalkIndex = gregor.mTalkIndex or 1

        local speech =
        {
            "You're another black blood aren't you?",
            "Come the morning, they'll kill you, just like the others.",
            "If I was you, I'd try to escape.",
            "Pry the drain open, with that big bone you're holding."
        }

        local text = speech[gregor.mTalkIndex]

        local height = 102
        local width = 500
        local x = 0
        local y = -System.ScreenHeight()/2 + height / 2
        gStack:PushFix(gRenderer, x, y, width, height, text,
        {
            textScale = 1,
            title = "Prisoner:"
        })
        gregor.mTalkIndex = gregor.mTalkIndex + 1
        if gregor.mTalkIndex > #speech then
            gregor.mTalkIndex = 1
        end
    end
end
```

Listing 2.295: The GregorTalk script. In map_jail.lua.

The TalkGregor code in Listing 2.295 checks where the NPC is standing. We don't want to fire the trigger unless Gregor is behind the bars. If Gregor isn't one tile north of the trigger, then the function exits.

When Gregor is in the correct position, we try to get the `mTalkIndex` value from the NPC object. If the `mTalkIndex` field is nil then it's set to 1. NPCs don't normally have an `mTalkIndex` variable so the first time the code runs it's created and set to 1.

There's a table called `speech` containing all the text that Gregor might say. He'll only ever say one of these lines at a time. The `mTalkIndex` indexes the speech table to decide which line Gregor says next. Then there's a little code to position a fixed size textbox at the bottom of the screen. At the end of the function the textbox is created and displays the selected speech text.

After the textbox is shown, the `mTalkIndex` is increased by 1. The next time the player talks to Gregor, Gregor will say the next phrase. If the `mTalkIndex` becomes greater than the number of entries in the speech table it resets to 1. So Gregor starts repeating himself eventually.

With the script written, let's add it to the map using a trigger. Copy the code from Listing 2.296 into your `map_jail.lua` file.

```
return
{
    -- code omitted
    on_wake =
    {
        {
            id = "AddNPC",
            params = {{ def = "prisoner", id = "gregor", x = 44, y = 12 }},
        }
    },
    actions =
    {
        -- code omitted
        talk_gregor =
        {
            id = "RunScript",
            params = { TalkGregor }
        }
    },
    trigger_types =
    {
        -- code omitted
        gregor_talk_trigger = { OnUse = "talk_gregor" },
    },
    triggers =
    {
        -- code omitted
        { trigger = "gregor_talk_trigger", x = 50, y = 13 }
    },
}
```

Listing 2.296: Adding the TalkGregor script into the map. In map_jail.lua.

Run the code and you'll be able to have a chat with Gregor! See Figure 2.113.



Figure 2.113: Chatting with the prisoner next door.

Prising The Grate

Our player can break through a wall, desecrate a corpse, and chat to a prisoner, but he cannot yet escape. We need a trigger to let the player force open the grate with the bone. Once the grate is open, we'll change the grate tile to represent that it's broken and add a new trigger to allow the player to escape.

Example dungeon-12 contains the code so far but with two extra sounds, *grate.wav* and *reveal.wav*. These sounds are played when moving the grate and when the tunnel is revealed.

We're going to add two functions, *UseGrateScript* and an empty *EnterGrateScript*. Unlike the previous examples, we'll hook both the scripts into the map first and then implement the *UseGrateScript* code. The code is shown in Listing 2.297.

```
return
{
    -- code omitted
    actions =
```

```

{
    -- code omitted
    use_grate =
    {
        id = "RunScript",
        params = { UseGrate }
    },
    enter_grate =
    {
        id = "RunScript",
        params = { EnterGrate }
    }
},
trigger_types =
{
    -- code omitted
    grate_close = { OnUse = "use_grate" },
    grate_open = { OnEnter = "enter_grate" },
},
triggers =
{
    -- code omitted
    { trigger = "grate_close", x = 57, y = 6 },
    { trigger = "grate_close", x = 58, y = 6 },
}
,

```

Listing 2.297: Adding the triggers and actions for the open and closed grate. In map_jail.lua.

The grate has two triggers, one for when it's closed and one for when it's open. At the start of the map, the only trigger is the closed one. We switch the triggers to grate_open using code when the player prizes it open.

Before writing the scripts, let's add an AddTrigger function to the map. This allows triggers to be added to the map at runtime. Copy the changes from Listing 2.298 in map.lua.

```

function Map:Create(mapDef)

    -- code omitted

    setmetatable(this, self)

    --
    -- Place any triggers on the map

```

```

--  

this.mTriggers = {}  

for k, v in ipairs(mapDef.triggers) do  

    this:AddTrigger(v)
end

-- code omitted
end

function Map:AddTrigger(def)
    local x = def.x
    local y = def.y
    local layer = def.layer or 1

    if not self.mTriggers[layer] then
        self.mTriggers[layer] = {}
    end

    local targetLayer = self.mTriggers[layer]
    local trigger = self.mTriggerTypes[def.trigger]
    assert(trigger)
    targetLayer[self:CoordToIndex(x, y)] = trigger
end

```

Listing 2.298: AddTrigger function, used to add triggers to the map. In Map.lua.

In Listing 2.298, code to handle adding triggers on the map is moved from the constructor into the AddTrigger function.

Next let's add the UseGrate function as shown in Listing 2.299.

```

local EnterGrate = function() end

local UseGrate =
function(map, trigger, entity, x, y, layer)
    if gWorld:HasKey(BoneItemId) then
        local OnOpen
        local dialogParams =
        {
            textSizeScale = 1,
            choices =
            {
                options =
                {
                    "Prize open the grate",

```

```

        "Leave it alone"
    },
    OnSelection = function(index)
        if index == 1 then
            OnOpen(map)
        end
    end
},
}

gStack:PushFit(gRenderer,
0, 0,
"There's a tunnel behind the grate. " ..
"Prize it open using the bone?",
300,
dialogParams)

OnOpen = function()
    Sound.Play("grate")
    map:WriteTile
    {
        x = 57,
        y = 6,
        layer = layer,
        tile = 151,
        collision = false
    }
    map:WriteTile
    {
        x = 58,
        y = 6,
        layer = layer,
        tile = 152,
        collision = false
    }

    map:AddTrigger
    {
        x = 57,
        y = 6,
        layer = layer,
        trigger = "grate_open",
    }
    map:AddTrigger

```

```

    {
        x = 58,
        y = 6,
        layer = layer,
        trigger = "grate_open",
    }
end
else
    gStack:PushFit(gRenderer, 0, 0,
        "There's a tunnel behind the grate. " ..
        "But you can't move the grate with your bare hands",
        300,
        {textScale=1})
end
end

```

Listing 2.299: Implementing the UseGrate script. In map_jail.lua.

The UseGrate function first checks if the bone is in the inventory. If the player doesn't have the bone a dialog box is created, telling the player they can see a tunnel behind the grate but can't open it.

If UseGrate is called when the player has the bone, then we use a dialog box to ask the player if they want to try to pry the grate open. The callback for the *pry the grate* reply is named OnOpen and predeclared near the top of the function.

Then we define a table for the textbox parameters. The textscale is set to 1 as normal and the two choices for interacting with the grate are set to:

1. "Open it"
2. "Leave it alone"

The function to handle the choice is also defined here. If the user selects "Leave it alone" then the dialog box closes and nothing happens. Choosing "Open it" calls the empty OnOpen function. After the dialogParams table definition, the dialog box is displayed and the OnOpen function is defined.

The OnOpen function plays a grate sound and then removes the trigger to stop the player from being able to open the grate more than once. The tile id is changed to show an open grate and a new trigger is added on the tile. The new trigger is grate_open and it calls the EnterGrate script when the player walks on it.

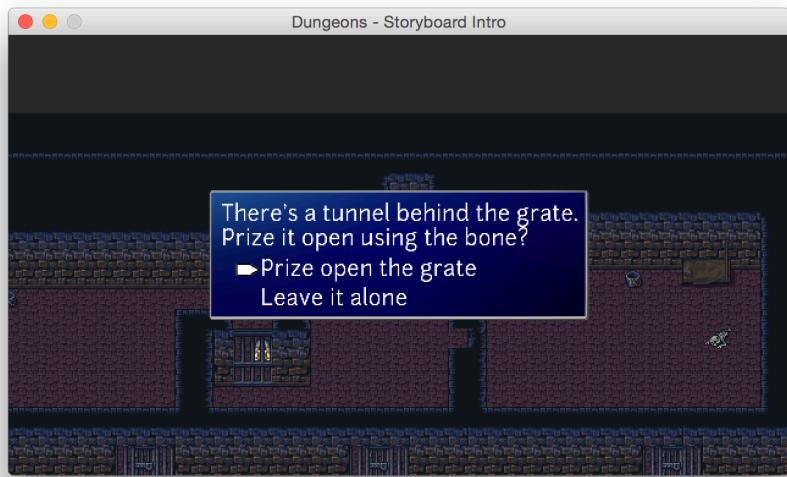


Figure 2.114: Using a bone to open the grate.

The EnterGrate function is currently empty so nothing happens when it runs.

Example dungeon-12-solution contains all the code so far. Run it. You'll be able to open the grate but the player still can't escape quite yet. The interaction dialog is shown in Figure 2.114.

Intrigue Cutscene

In this section we'll implement the EnterGrate function. This function makes the player disappear and then plays a cutscene that tries to give this micro-rpg a sense of intrigue.

After the player exits, the camera pans left a little and a new NPC walks on screen. The new NPC unlocks the adjoining jail cell and talks to the prisoner who was your neighbor. They'll talk cryptically of a plan, then the camera changes to a new map and control returns to the player.

Example dungeon-13 contains the code so far.

Handing A State To The Storyboard

When we run the Storyboard at the start of the game, we're using it fresh. There are no other states on the global stack state. The storyboard we run when the player escapes

is played on a stack that already has an ExploreState. Previously we used the HandOff operation to move the jail from the storyboard stack to the main stack. Now we do the opposite and move an ExploreState from the main stack onto the Storyboard stack so we can control it. Copy the code from Listing 2.300.

```
function Storyboard:Create(stack, events, handIn)
    local this =
    {
        mStack = stack,
        mEvents = events,
        mStates = {},
        mSubStack = StateStack>Create(),
        mPlayingSounds = {}
    }

    setmetatable(this, self)

    if handIn then
        local state = this.mStack:Pop()
        this:PushState("handin", state)
    end

    return this
end
```

*Listing 2.300: Extending the Storyboard class to pull in the top state from the stack.
In Storyboard.lua.*

In the Storyboard:Create we've added a third optional parameter called handIn. If handIn is set to true, it pops the state off the top of the main stack and pushes it onto the storyboard internal stack.

When the player leaves the dungeon, we want to pan the camera to the left, but the ExploreState forces it to *always* follow the player. We need code to help move the camera around more freely.

Free Camera Movement

Let's extend ExploreState and add an mFollowCam toggle to control how the camera behaves. Copy the changes from Listing 2.301.

```
function ExploreState:Create(stack, mapDef, startPos)
    local this =
    {
        mStack = stack,
```

```

mMapDef = mapDef,

mFollowCam = true,
mFollowChar = nil,
mManualCamX = 0,
mManualCamY = 0,
}

-- code omitted

this.mFollowChar = this.mHero
setmetatable(this, self)
return this
end

```

Listing 2.301: Adding camera options to the ExploreState. In ExploreState.lua.

Four fields have been added:

- `mFollowCam` - determines if the camera tracks the player or is moved manually.
- `mFollowChar` - the character the camera should follow.
- `mManualCamX` and `mManualCamY` - coordinates used to position the camera when manually setting its position.

By default the ExploreState follows the hero, so at the end of the constructor the `mFollowChar` is set to the hero.

Now let's add some helper functions and modify the update loop a little. Copy the code from Listing 2.302.

```

function ExploreState:UpdateCamera(map)

    if self.mFollowCam then
        local pos = self.mHero.mEntity.mSprite:GetPosition()
        map.mCamX = math.floor(pos:X())
        map.mCamY = math.floor(pos:Y())
    else
        map.mCamX = math.floor(self.mManualCamX)
        map.mCamY = math.floor(self.mManualCamY)
    end

end

function ExploreState:SetFollowCam(flag, character)

```

```

self.mFollowChar = character or self.mFollowChar
self.mFollowCam = flag
if not self.mFollowCam then
    local pos = self.mFollowChar.mEntity.mSprite:GetPosition()
    self.mManualCamX = pos:X()
    self.mManualCamY = pos:Y()
end
end

function ExploreState:Update(dt)

    local map = self.mMap

    self:UpdateCamera(map)

    for k, v in ipairs(map.mNPCs) do
        v.mController:Update(dt)
    end
end

```

Listing 2.302: Helper functions and Update loop modifications. In ExploreState.lua.

Listing 2.302 contains two new functions, `UpdateCamera` and `SetFollowCam`. `UpdateCamera` gathers together some code that was previously in the `Update` function. It checks the `mFollowCam` flag. If the camera is following a character, it uses the character's sprite position to set the camera position. If the camera isn't following anyone, it uses the `mManualCamX` and `mManualCamY` coordinates to position the camera. This gives the `ExploreState` two different ways of positioning the camera.

The `SetFollowCam` function allows character tracking to be turned on or off. If you switch from following a character to the manual camera, the position of the manual camera will be set to the last position of the character. This prevents the camera from suddenly jumping to 0, 0 when the follow camera is turned off.

Finally, the `Update` loop now has less code because the camera code has been offloaded to the `UpdateCamera` function.

These changes require updates to the `ReplaceScene` operation. When a scene change occurs, we need to reset the camera position in the `ExploreState`. The code below in Listing 2.303 resets the camera so that if the scene is changed the camera goes back to following the player.

```

function SOP.ReplaceScene(name, params)
    return function(storyboard)

        -- code omitted
    end
end

```

```

        state.mHero.mEntity:SetTilePos(
            params.focusX,
            params.focusY,
            params.focusZ or 1,
            state.mMap)
        state:SetFollowCam(true, state.mHero)

        -- code omitted
    end
end

```

Listing 2.303: When the ReplaceScene operation is called the camera should reset. In StoryboardEvents.lua.

Now we can manually move the camera in the ExploreState or have it follow a character.

As you create your own cutscenes you'll find yourself adding new operations as you go. The more operations, the more power and flexibility you can draw on when creating future cutscenes. Let's increase our power a little by adding three new operations: FadeOutChar, WriteTile, and MoveCamToTile.

```

function SOP.FadeOutChar(mapId, npcId, duration)

    duration = duration or 1

    return function(storyboard)

        local map = GetMapRef(storyboard, mapId)
        local npc = map.mNPCbyId[npcId]
        if npcId == "hero" then
            npc = storyboard.mStates[mapId].mHero
        end

        return TweenEvent:Create(
            Tween:Create(1, 0, duration),
            npc.mEntity.mSprite,
            function(target, value)
                local c = target:GetColor()
                c:SetW(value)
                target:SetColor(c)
            end)
        end
    end
end

```

Listing 2.304: The FadeOutChar operation. In StoryboardEvents.lua.

The operation in Listing 2.304 fades a character out using a tween. We use this operation to disappear the hero into the grate. The duration of the fade defaults to 1 second if not specified. When the operation is executed, we get the map using the mapId and get the NPC to fade using the npcId.

If the npcId is set equal to "hero", then the fade is applied to the hero character. The fade uses a TweenEvent to alter the alpha channel of the character sprite.

```
function SOP.WriteTile(mapId, writeParams)
    return function(storyboard)
        local map = GetMapRef(storyboard, mapId)
        map:WriteTile(writeParams)
        return EmptyEvent
    end
end
```

Listing 2.305: WriteTile instruction. In StoryboardEvents.lua.

The operation in Listing 2.305 is simpler. It uses the mapId to look up the map and writes new tile information into that map. This is all code we've covered before but wrapped up as an operation.

```
function SOP.MoveCamToTile(stateId, tileX, tileY, duration)

    duration = duration or 1

    return function(storyboard)

        local state = storyboard.mStates[stateId]
        state:SetFollowCam(false)
        local startX = state.mManualCamX
        local startY = state.mManualCamY
        local endX, endY = state.mMap:GetTileFoot(tileX, tileY)
        local xDistance = endX - startX
        local yDistance = endY - startY

        return TweenEvent:Create(
            Tween:Create(0, 1, duration, Tween.EaseOutQuad),
            state,
            function(target, value)
                local dX = startX + (xDistance * value)
                local dY = startY + (yDistance * value)
            end
        )
    end
end
```

```

        state.mManualCamX = dX
        state.mManualCamY = dY
    end)
end
end

```

Listing 2.306: Move Camera To Tile instruction. In StoryboardEvents.lua.

The operation in Listing 2.306 moves the camera from its current position to focus on a given tile. It's powered by a TweenEvent. An optional duration parameter controls how fast the camera moves. The duration defaults to 1 second if no value is supplied.

When the MoveCamToTile operation is executed it looks up the ExploreState and turns off the FollowCam. This stops the ExploreState from following the hero around and allows us to manually set the camera position. The start and end positions are then calculated, as is the distance between them. We use these values to smoothly move the camera with the tween.

The TweenEvent callback function updates the camera position. It tweens between the start and end positions by adding the distance multiplied by the tween to the start position.

With these three new operations added, we're ready to implement the EnterGrate function. Copy it from Listing 2.307.

```

local EnterGrate =
    function(map, trigger, entity, x, y, layer)
        Sound.Play("reveal")
        map:RemoveTrigger(57, 6, layer)
        map:RemoveTrigger(58, 6, layer)
        local cutscene =
        {
            --[[ We'll add this shortly --]]
        }
        local storyboard = Storyboard.Create(gStack, cutscene, true)
        gStack:Push(storyboard)
    end

```

Listing 2.307: The EnterGrate function. In map_jail.lua.

For clarity I've removed the cutscene content from Listing 2.307. We'll fill this in shortly, but first let's consider the other parts of the EnterGrate function.

The EnterGrate function plays the reveal sound effect to notify the player that they've solved a problem. Then there's a cutscene table, which we'll flesh out next. A storyboard is created with the third parameter set to true. The third parameter tells the storyboard

to remove whatever's on top of the global stack and add it to its internal stack. In practice this means the current map can be referenced by the storyboard using the "handin" id. Finally, the storyboard is pushed on top of the stack and starts to run.

Let's fill in the storyboard data! Hopefully this gives the player a taste of intrigue. It's a pretty big cutscene so we'll tackle it in two pieces. The first piece is shown in Listing 2.308.

```
local cutscene =
{
    SOP.BlackScreen("blackscreen", 0),
    SOP.NoBlock(
        SOP.FadeOutChar("handin", "hero")
    ),
    SOP.RunAction("AddNPC",
        {"handin", { def = "guard", id="guard1", x = 12, y = 28}},
        {GetMapRef}),
    SOP.Wait(2),
    SOP.NoBlock(
        SOP.MoveNPC("gregor", "handin",
            {
                "left",
                "left",
                "left",
                "left",
                "left",
                "left",
                "left",
                "down",
            })
    ),
    SOP.Wait(1),
    SOP.NoBlock(
        SOP.MoveCamToTile("handin", 19, 26, 3)
    ),
    SOP.Wait(1),
    SOP.MoveNPC("guard1", "handin",
        {
            "right",
            "right",
            "right",
            "right",
            "right",
            "right",
            "right",
            "up"
        })
}
```

```

}),
SOP.Wait(1),
SOP.Play("unlock"),
SOP.WriteTile("handin", {x = 19, y = 26, tile = 0}),

```

Listing 2.308: The cutscene that plays after the player enters the grate. In map_jail.lua.

Most of the operations in the cutscene are ones we've used before, but some are new. Let's go through what's happening.

First we add a black screen that we'll use for fading in and out. We begin by setting its alpha to 0. Then we fade out the hero character as he travels through the tunnel behind the grate.

As the hero is exiting the jail, we add a new NPC in the corridor. This NPC is the same as the one that kidnapped the player from his house. After the guard NPC is added, there's a short wait to give the user time to realize that something else is going to happen in the jail.

Gregor, the prisoner in the next cell, is given a MoveNPC instruction. This path moves him back so that he's inline with the door. It's a non-blocking instruction, so the following operations execute while Gregor starts to move.

After a short wait the camera moves to a tile just in front of Gregor's cell door. This movement takes 3 seconds and doesn't block the cutscene. The guard NPC starts walking to the same tile the camera is moving to. When the guard arrives, he waits for 1 second, then an unlock sound is played and the door to Gregor's cell is replaced with an open pathway.

That's the first part of the cutscene. Gregor and the guard are now ready to speak. The second part of the cutscene continues below in Listing 2.309.

```

SOP.NoBlock(
    SOP.MoveNPC("guard1", "handin",
    {
        "up",
        "up",
        "up",
        "up",
        "up",
    })
),
SOP.Wait(2),
SOP.Say("handin", "guard1", "Has the black blood gone?", 3),
SOP.Wait(1),
SOP.Say("handin", "gregor", "Yeh.", 1),
SOP.Wait(1),

```

```

SOP.Say("handin", "guard1", "Good.", 1),
SOP.Wait(0.25),
SOP.Say("handin", "guard1",
        "Marmil will want to see you in the tower.", 3.5),
SOP.Wait(1),
SOP.Say("handin", "gregor", "Let's go.", 1.5),
SOP.Wait(1),
SOP.NoBlock(
    SOP.MoveNPC("gregor", "handin",
    {
        "down",
        "down",
        "down",
        "down",
        "down",
        "down",
        "down",
    })
),
SOP.NoBlock(
    SOP.MoveNPC("guard1", "handin",
    {
        "down",
        "down",
        "down",
        "down",
        "down",
        "down",
        "left",
    })
),
SOP.FadeInScreen(),
-- Need to change scene here
}

```

Listing 2.309: The second part of the jail cutscene. In StoryboardEvents.lua.

In Listing 2.309 the MoveNPC operation moves the guard into Gregor’s cell. Then we use the Say operation to make Gregor and the guard have a chat. The dialog is controlled by timed text boxes, and the timing is set according to the length of the text. After the conversation ends, both characters have MoveNPC operations run on them and the screen fades out.

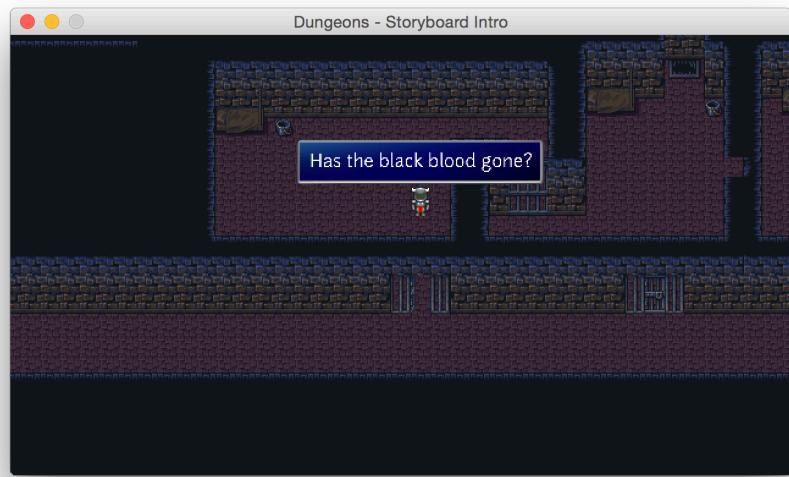


Figure 2.115: The post jail cutscene.

Example dungeon-13-solution contains the code so far. Run the it and you'll be able to see the cutscene in action. Part of the cutscene is displayed in Figure 2.115.

Once the cutscene ends, the game ends. The storyboard has the jail on its internal stack. When the storyboard finishes, it pops itself off the stack and there's nothing left. In the next section we add a few more instructions to the cutscene to transition the player to the sewers under the jail.

Map Swap And Escape

The final part of our small demo places the player into a sewer, a classic RPG locale. If the player follows the sewer to the end, a trigger is activated to end the game. Example dungeon-14 contains the code so far. This project contains a new map map_sewer.lua, GameOverState.lua, and a new tileset, tileset_sewer.png.

We're going to start by adding the sewer to MapDB.lua. Copy the code from Listing 2.310.

```
MapDB =
{
    ["player_house"] = CreateHouseMap,
    ["jail"] = CreateJailMap,
```

```
    [ "sewer" ] = CreateSewerMap,  
}
```

Listing 2.310: Add the sewer map to the map database. In MapDB.lua.

Next we add a few operations to the end of the jail cutscene to move control over to the sewer map. Copy the code from Listing 2.311.

```
SOP.FadeInScreen(),  
SOP.ReplaceScene(  
    "handin",  
{  
    map = "sewer",  
    focusX = 35,  
    focusY = 15,  
    hideHero = false  
}),  
SOP.FadeOutScreen(),  
SOP.HandOff("sewer")  
}
```

Listing 2.311: Moving into the sewers. In map_jail.lua.

In this final part of the jail cutscene, the screen fades to black, the scene changes to the sewer map, and finally the screen fades into the sewer. The storyboard moves the sewer to the global stack and then the storyboard is removed.

Our hero can now escape into the sewers. The final task is to break free entirely and thus bring our mini-rpg to an end. We'll add a row of triggers near the end of the sewer that trigger the end of the game.

When the player leaves the sewer, we switch to a GameOverState. The GameOverState displays text telling the user that the game has ended. This is going to be a very simple class! Copy the code from Listing 2.312.

```
GameOverState = {}  
GameOverState.__index = GameOverState  
function GameOverState:Create(stack)  
    local this =  
    {  
        mStack = stack  
    }  
  
    setmetatable(this, self)  
    return this
```

```

end

function GameOverState:Enter()
    CaptionStyles["title"].color:SetW(1)
    CaptionStyles["subtitle"].color:SetW(1)
end

function GameOverState:Exit() end
function GameOverState:Update(dt) end
function GameOverState:HandleInput() end

function GameOverState:Render(renderer)
    CaptionStyles["title"]:Render(renderer,
        "Game Over")

    CaptionStyles["subtitle"]:Render(renderer,
        "Want to find out what happens next? Write it!")
end

```

*Listing 2.312: The GameOverState to be displayed when the player finishes the game.
In GameOverState.lua.*

There's not much to the GameOverState class. The Render function reuses the caption styles to display a nice game over message at the end. The enter function sets the label's alpha to 1 to ensure the text is visible.

With the GameOverState created, we can transition to it by implementing a new trigger. The triggers use a RunScript action to call the SewerExit function. Copy the code from Listing 2.313.

```

function CreateSewerMap()

    local SewerExit =
        function(map, trigger, entity, x, y, layer)
            gStack:Pop() -- pop off the ExploreState
            gStack:Push(GameOverState>Create(gStack))
        end

```

Listing 2.313: The SewerExit function. In map_sewer.lua.

The SewerExit script pops the ExploreState off the stack and pushes on the GameOverState, causing an instant transition. Let's add the triggers to the map as in Listing 2.314.

```

return
{
    -- code omitted
    actions =
    {
        exit_sewer_script =
        {
            id = "RunScript",
            params = { SewerExit }
        },
    },
    trigger_types =
    {
        exit_trigger = { OnEnter = "exit_sewer_script" }
    },
    triggers =
    {
        { trigger = "exit_trigger", x = 52, y = 15},
        { trigger = "exit_trigger", x = 52, y = 16},
        { trigger = "exit_trigger", x = 52, y = 17},
        { trigger = "exit_trigger", x = 52, y = 18},
        { trigger = "exit_trigger", x = 52, y = 18},
        { trigger = "exit_trigger", x = 52, y = 20},
    },
}

```

Listing 2.314: Placing the exit triggers. In map_sewer.lua.

In Listing 2.314 we add a row of triggers at the end of the sewer. The sewer is a long thin corridor. After the player walks a distance down the corridor, the trigger fires and the game ends. The line of triggers is width of the corridor, so the user is forced to cross one.

The code for this section is available in `dungeon-14-solution`. Run it and you'll be able to play through the game and see the game over screen on completing it.

The Main Menu

How can we call the game finished if it doesn't have a title screen menu? Let's add one!

Example `dungeon-15` contains the latest version of the code as well as the files `TitleScreenState.lua` and `title_screen.png`. You can see a finished title screen in Figure 2.116.



Figure 2.116: The implemented title screen.

The title screen displays a sprite of the game logo, a subtitle rendered in text, and a menu asking the user if they would like to play the game or quit. Copy the constructor from Listing 2.315 into your TitleScreenState.lua file.

```
TitleScreenState = {}  
TitleScreenState.__index = TitleScreenState  
function TitleScreenState:Create(stack, storyboard)  
    local this  
    this =  
    {  
        mTitleBanner = Sprite.Create(),  
        mStack = stack,  
        mStoryboard = storyboard,  
        mMenu = Selection:Create  
        {  
            data = {"Play", "Exit"},  
            spacingY = 32,  
            OnSelection = function(index)  
                this:OnSelection(index)  
            end  
        }  
    }  
end
```

```

this.mTitleBanner:SetTexture(Texture.Find("title_screen.png"))
this.mTitleBanner:SetPosition(0, 100)

-- Code to center the menu
this.mMenu.mCursorWidth = 50
this.mMenu.mX = -this.mMenu:GetWidth()/2 - this.mMenu.mCursorWidth
this.mMenu.mY = -50
setmetatable(this, self)
return this
end

```

Listing 2.315: The TitleScreenState constructor. In TitleScreenState.lua.

The TitleScreenState constructor takes in two parameters, the global stack and a storyboard state. The title screen uses the stack and storyboard to run the opening cutscene when the player selects “New Game”.

The game logo is stored in the this table as a sprite called mTitleBanner. The this table also contains references to the stack, storyboard, and an mMenu. The mMenu is a selection menu that gives the player two options, play the game or quit. The mMenu’s OnSelection callback calls the TitleScreenState.OnSelection function and we handle the selection logic there.

After the this table is set up, we set texture for the logo and position it near the top of the screen. The mMenu is positioned so it’s centered horizontally and 50 pixels down from the vertical center. The menu cursor is adjusted so it’s not so close to the text.

Let’s add the remaining functions for TitleScreenState from Listing 2.316.

```

function TitleScreenState:Enter() end
function TitleScreenState:Exit() end
function TitleScreenState:Update(dt) end

function TitleScreenState:OnSelection(index)
    if index == 1 then
        self.mStack:Pop()
        self.mStack:Push(self.mStoryboard)
    elseif index == 2 then
        System.Exit()
    end
end

function TitleScreenState:Render(renderer)

    renderer:DrawRect2d(
        -System.ScreenWidth()/2,

```

```

        -System.ScreenHeight()/2,
        SystemScreenWidth()/2,
        System.ScreenHeight()/2,
        Vector.Create(0, 0, 0, 1)
    )

    renderer:DrawSprite(self.mTitleBanner)
    renderer:AlignText("center", "center")
    renderer:ScaleText(0.8, 0.8)
    renderer:DrawText2d(0, 60, "A mini-rpg adventure.")
    renderer:ScaleText(1, 1)
    renderer:AlignText("left", "center")
    self.mMenu:Render(renderer)
end

function TitleScreenState:HandleInput()
    self.mMenu:HandleInput()
end

```

Listing 2.316: Title screen functions. In TitleScreenState.lua.

In Listing 2.316 the first function with code is `OnSelection`. This is the callback from the selection menu. If the selection index equals 1, the first option, “Play”, has been selected. If it equals 2, “Quit” has been selected. When “Play” is chosen we pop the `TitleMenuState` off the stack and push on the `mStoryboard` to start the introduction cutscene. If the player chooses “Quit” then we exit the game.

The `Render` function draws a black rectangle covering the entire screen. It then draws the title banner. The subtitle is scaled to 0.8 and is drawn below the title logo. The text alignment is changed to center, left and the selection menu is drawn.

Let’s use the `TitleScreenState` in the game. Update the `main.lua` file as shown in Listing 2.317.

```

-- code omitted

local storyboard = Storyboard:Create(gStack, intro)
local titleState = TitleScreenState:Create(gStack, storyboard)
gStack:Push(titleState)

-- code omitted

```

Listing 2.317: Making the Title Screen the first state when running the program. In main.lua.

Example `dungeon-15-solution` contains all the code. Run it and you’ll see a nice title screen that you can use to enter the game.

Fix Up

At this point the game is finished and it looks good, but we can make it look better!

The mini-rpg uses custom fonts. Using custom fonts means the text is a slightly different size and therefore some of the textboxes and in-game menus look a little off. This usually can't be solved 100% in code. Fonts contain their own spacing and flourishes, and differ in legibility. For that reason it's hard to predict the exact spacing, scale and position to make an appealing interface.

Open the in-game menu. You'll notice that the text overflows some of the panels. This can be seen in Figure 2.117.



Figure 2.117: Using a different font means we need to adjust the menus. The text is too close to the menu edging.

Similarly there's not quite enough spacing between the title of the speaker and the body text in the dialog boxes.

We need to adjust the font scale, and it's better if we can do this globally. Therefore we'll store scale values in the InGameMenuState as shown in Listing 2.318.

```
function InGameMenuState:Create(stack)
    local this =
    {
        mStack = stack,
```

```
mTitleSize = 1.2,  
mLabelSize = 0.88,  
mTextSize = 1,  
}
```

Listing 2.318: Putting the text sizes all in one place. In InGameMenuState.lua.

In Listing 2.318 we've added `mTitleSize`, `mLabelSize` and `mTextSize`. We'll use these values to globally adjust text scales. The scale values have already been tweaked for the font we're using.

Let's update the `FrontMenuState` to make use of these font scales. Copy the code from Listing 2.319.

```
function FrontMenuState:Render(renderer)  
    -- code omitted  
    renderer:ScaleText(self.mParent.mTitleSize, self.mParent.mTitleSize)  
    -- code omitted  
    self.mSelections:Render(renderer)  
  
    -- code omitted  
    renderer:DrawText2d(nameX, nameY, self.mTopBarText)  
  
    -- code omitted  
  
    renderer:ScaleText(self.mParent.mLabelSize, self.mParent.mLabelSize)  
    renderer:AlignText("right", "top")  
    renderer:DrawText2d(goldX, goldY, "GP:")  
    -- code omitted  
end
```

Listing 2.319: Updating the text size in the front menu. In FrontMenuState.lua.

In Listing 2.319 all hardcoded text scales are replaced with values from the parent state. Next let's do the same for the `ItemMenuState` as shown in Listing 2.320.

```
function ItemMenuState:Render(renderer)  
    -- code omitted  
  
    renderer:ScaleText(self.mParent.mTitleSize, self.mParent.mTitleSize)  
    renderer:AlignText("center", "center")  
    renderer:DrawText2d(titleX, titleY, "Items")  
  
    -- code omitted
```

```

renderer:ScaleText(self.mParent.mTitleSize, self. mParent.mTitleSize)
self.mCategoryMenu:Render(renderer)

-- code omitted

renderer:ScaleText(self.mParent.mTextSize, self. mParent.mTextSize)

-- code omitted
end

```

Listing 2.320: Updating the text size in the item menu state. In ItemMenuState.lua.

Open the in-game menu now and you'll see that there's no overflow, as shown in Figure 2.118.

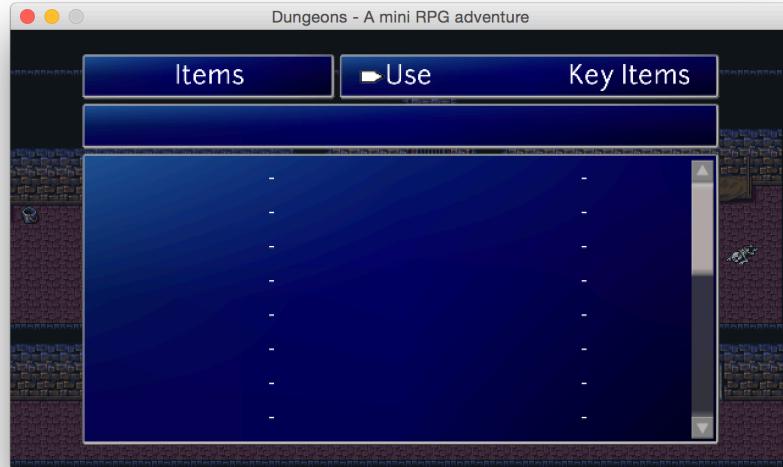


Figure 2.118: The ItemMenuState with the overflow removed.

If we change the font again, it's easy to change the scales because they're all in one central place.

The change in font also affects the textboxes. The spacing between the title of the textbox and the body text needs increasing. We make this change by adding a new optional parameter, `titlePadY`.

```

function StateStack:PushFix(renderer, x, y, width, height, text, params)

    -- code omitted

    local padding = 10
    local titlePadY = params.titlePadY or 10

    -- code omitted

    if title then
        -- adjust the top
        local size = renderer:MeasureText(title, wrap)
        boundsTop = size:Y() + padding * 2 + titlePadY

        table.insert(children,
        {
            type = "text",
            text = title,
            x = 0,
            y = size:Y() + padding + titlePadY
        })
    end

    -- code omitted

end

function StateStack:PushFit(renderer, x, y, text, wrap, params)

    -- code omitted

    local padding = 10
    local titlePadY = params.titlePadY or 10

    -- code omitted

    if title then
        local size = renderer:MeasureText(title, wrap)
        height = height + size:Y() + titlePadY
        width = math.max(width, size:X() + padding * 2)
    end

```

Listing 2.321: Changing textbox size for new fonts. In StateStack.lua.

That ends the tweaks for the menu and textboxes! There are many other improvements

we *could* add, such as

- Adding new operations for making the cutscenes easier to script
- Fades and transitions for the menu states and in-game menu
- Better direction for the cutscenes
- More sound effects

There's always more to polish but we're stopping here to move on to combat.

Conclusion

The game you've made in this section is quite an achievement. Be proud of it!

You now have the skills to make a game like this, or one of much larger scope and depth. Pause and reflect on that. You can make story-driven RPGs. You can create worlds with the tools we built up in these last few chapters! If you have an idea for a game, give it a go. You don't know where you might end up! Don't have any ideas right now? Well, do you think you could recreate a scene from an RPG you enjoyed? That would be a good learning experience and a source of inspiration. Or possibly you want to build on the demo we've just created?

Once you're ready let's move on to the next part: combat!

Chapter 3

Combat

All adventures are defined by struggle and overcoming adversity. In the JRPG this struggle is usually represented by combat. This section covers all the elements required to build a robust JRPG battle system. We'll cover stats, levelling, equipment, UI and then on to the combat simulation itself. We'll also cover how to tie these systems into our existing codebase. This part of the book ends with an arena game demonstrating everything we'll learn.

Stats

"Nothing Exists Per Se Except Atoms and the Void."

- On the Nature of Things, Lucretius, ~100 B.C

What is a Stat?

A statistic, or stat, is a number describing an aspect of a game entity. A game entity might be a monster, character, weapon or spell. Stats help describe entities in the game.

This table shows game items described using different stats.

Gold Bar	Sword	Potion
Weight	Damage	HP Restore Amount
Cost	Souls Eaten	MP Restore Amount

Table 3.1: Example stats for items.

In JRPGs, stats are used exclusively to simulate combat. When you attack a monster, the computer needs a way to decide if you manage to hit it, how much damage is inflicted, and if the blow kills the creature. To make these decisions, the computer runs combat simulation code using stats.

To get a better idea of what a stat is, let's take a look at the stats commonly used to describe characters and monsters in JRPGs.

- **Health Points (HP)** - The amount of damage a character can take before dying or being knocked out.
- **Magic Points (MP)** - The amount of magical power a character has. The higher the power, the more spells can be cast.
- **Strength** - Represents the character's physical strength. Determines how damaging attacks are.
- **Speed** - How fast the character moves. Determines frequency of attacks and chance to dodge attacks.
- **Intelligence** - How clever the character is. Determines power of spells and ability to resist magic attacks.

These examples will be familiar if you've played a lot of JRPGs, though the exact effect of each stat differs from game to game. In Western RPGs, there tend to be a lot more stats because Western RPGs use stats outside of combat. For instance, a charisma stat, describing how charming the character is, might open up special conversation options or lower the prices of goods at shops.

Stats aren't only used for characters. Consider the stats for a weapon.

- **Attack type** - Is the attack physical, magical, elemental or something else?
- **Attack power** - The damage this weapon can inflict.
- **Defense power** - The amount of damage this weapon can block.
- **Attack rate** - How quickly the weapon can be used to attack. A small knife is quicker than a two-handed hammer.

In JRPGs, stats describe armor, magic spells, special abilities and so on. By now I'm sure you have a good idea of how stats are used. Stats are the pieces of information the computer uses to simulate combat. Therefore we can come up with a simple definition of what a stat is.

Stats are a description the computer can understand.

Notice how similar the sword stats are to the character stats? Attack power is similar to character strength; both determine the amount of damage inflicted. Attack rate is similar to speed; both determine how often an attack can take place. Stats that seem to have a similar effect are combined in the combat simulation according to a formula to get a single, final number, called a *derived stat*.

Derived stats

The stats we've covered so far are called *base stats*. Base stats represent one pure aspect of an entity. They can't be broken down into smaller parts. Derived stats are calculated from one or more base stats to form a new number that represents them all. This calculation might be as simple as adding them but can be more complicated.

Base stats are rarely used in the combat simulation directly. Instead, they're transformed into derived stats to make the calculations easier. When we're simulating combat, we might want to know the total attack power of a character. This would be a derived stat. The calculation might look like this

```
total_attack_power = char.strength + weapon.power
```

The final attack power is derived from the weapon's power and the character's strength. In this case the calculation is simple and intuitive, but in a finished game it's likely to be more complicated.

Why Use Stats At All?

In the real world there are no stats. As far as we know, reality is made up of particles and the space between them, but modeling reality at that scale and fidelity is beyond our means and desire. In reality a wild boar does not have some speed stat describing how fast it is, it's just a big blob of particles. In our JRPG we want to be able to model a fight between a man and a boar without replicating reality exactly, and we do this using stats.

Computer games use abstraction to make things simpler. Stats are part of that abstraction. How do we decide who wins in a battle between a dragon and a giant squid? We write a simulation, give the dragon and squid stats, run the simulation code, and see the answer.

Stats without the simulation aren't very useful. The stats and simulation together help us answer questions important for an RPG, such as:

- How well does the magic potion heal my character?
- How much damage does my axe strike do to this troll?
- Will this dragon's fireball kill my character?
- Do I have enough magical energy to teleport my party?

The combat simulation and stats aren't modeled closely on reality because that tends to be quite limiting. Dragons aren't real, rabbits rarely carry gold coins, and combatants rarely trade blows in an orderly turn-based manner, but these features are often better for games. Gameplay is the most important consideration when designing a simulation.

In JRPGs, the most important simulation is the combat. Stats exist to make combat interesting and enjoyable.

Which Type of Stats Should a Game Have?

Here are three rules of thumb to help decide when to add a stat.

The Simulation Requires It

The best reason to create a stat is that it's needed to simulate an action in the game world. We might need to decide who attacks first in combat, then we need to know who is fastest, and therefore we need a speed stat. We need to determine how much damage is done and therefore we need a strength stat and an attack-power stat for the weapon, and so on.

Thematic

If the game is based on the Cthulhu mythos¹, and deals with mind-melting gods from other dimensions, a stat to represent sanity may be a good addition. If the game is about vampires, a stat representing the player's need to feed would be a good addition. A post-apocalypse game could have stats to represent hunger or radiation level.

Where possible, make stats complement the game themes.

Define by Difference

A stone giant and a dragon are fighting. Imagine how the battle would play out. The giant is strong but the dragon is fast. Those comparisons indicate an excellent place to introduce two stats to describe that difference. In this case we'd introduce speed and strength stats, and assign them appropriate values. When you want to differentiate between two entities according to some aspect, then it's worth adding a new stat (or depending on the context, a special ability).

Implementing Stats

We're now at a good point to start implementing stats in code. At the most basic level, a stat is a name associated with a number. A dictionary is a good datastructure to store this information. Below is a sketch of an implementation in Lua, Listing 3.1.

```
-- Stat: [id:string] -> [value:int]
base_stats =
{
    ["hp_now"] = 300,
    ["hp_max"] = 300,
    ["mp_now"],
```

¹Works based in the same universe as those of the author H. P. Lovecraft.

```

["mp_max"] = 300,
["strength"] = 10,
["speed"] = 10,
["intelligence"] = 10,
}

```

Listing 3.1: A sketch in Lua of how we might represent stats.

This code snippet defines a table of basic stats describing a player or creature in the game. The HP and MP stats each have been broken into two, hp_now and hp_max. This is because these stats are commonly depleted during combat. The hp_max and mp_max represent the maximum value the HP and MP can be, while the hp_now and mp_now represent the value of the HP and MP at the current point in time. In combat a player might receive a lot of damage. This reduces the hp_now stat. Drinking a potion restores the hp_now stat. When the player is fully healed, the hp_now stat is the same value as hp_max. The hp_max stat rarely changes.

A General Stat Class

A table of names and values is a great starting point, but for a combat simulation we're going to need to add more functionality. To begin with, we'll make a Stat class to store the stats and useful functions. See the Listing 3.2 below. A skeleton project for the code we'll be creating is available in combat/stats-1. Copy the code from Listing 3.2 into your own Stats.lua file.

```

Stats = {}
Stats.__index = Stats
function Stats>Create(stats)
    local this =
    {
        mBase = {},
        mModifiers = {}
    }

    -- Shallow copy
    for k, v in pairs(stats) do
        this.mBase[k] = v
    end

    setmetatable(this, self)
    return this
end

function Stats:GetBase(id)

```

```
    return self.mBase[id]
end
```

Listing 3.2: Building a basic Stats class. In Stats.lua.

This class constructor takes in one argument, a table of base stats (like those in Listing 3.1). It copies the values of this table into `mBase`. Copying the values gives each instance of the `Stats` class its own copy of each stat. If we didn't copy the stats, instances would share the same stat table, so when we reduced the `hp` in one object, we'd reduce the `hp` in all the objects! That's not what we want.

There's an additional empty table called `mModifiers` to handle temporary stat changes. We'll cover `mModifiers` a little later. The class currently has one function, `GetBase`. If you pass in a stat's id, it returns the current value. The use can be seen in this next snippet Listing 3.3.

```
local stats =
Stats>Create
{
    ["hp_now"] = 300,
    ["hp_max"] = 300,
    ["mp_now"] = 300,
    ["mp_max"] = 300,
    ["strength"] = 10,
    ["speed"] = 10,
    ["intelligence"] = 10,
}
print(stats:GetBase("intelligence")) -- 10
print(stats:GetBase("hp_now")) -- 300
```

Listing 3.3: A code snippet demonstrating the basic Stats table in use. In main.lua.

At the moment, there's little advantage to using the `stats` class compared to a simple table, but it's a good structure for us to build on. You can see a working example of this early version of the `stats` table in `combat/1-stats-solution`.

Modifiers

RPGs often require the base stats to be modified in some way. Here are some examples of modifiers:

- A magic sword that adds +5 to the player's strength
- A ring that adds 5% to the maximum HP

- A curse that reduces strength, speed and intelligence by 5
- The character becomes enraged, doubling his strength and halving his intelligence

There are a lot of different things happening in those examples! We need a way of simplifying these types of modifications so we don't end up with a pile of spaghetti code. Example combat/stats-2 contains the code so far.

Look carefully at the examples. We can classify them into two types, modifiers of addition and modifiers of multiplication. Let's reformulate the examples using addition and multiplication explicitly using Lua. See Listing 3.4 below.

```
magic_sword = { add = { ["strength"] = 5 } }
magic_ring = { mult = { ["max_hp"] = 0.05 } }
curse =
{
  add =
  {
    ["strength"] = -5,
    ["speed"] = -5,
    ["intelligence"] = -5
  }
}
rage = { mult = { ["strength"] = 1, ["intelligence"] = -0.5 } }
```

Listing 3.4: Some Lua code describing modifiers using addition and multiplication.

Listing 3.4 translates the English language descriptions into data, which is pretty cool! Halving a value is as simple as multiplying it by -0.5 and then subtracting it from the original value. Why not just multiply by 0.5 instead? We're coming to that, but basically it's going to help us when stacking multiple modifiers.

We're almost ready to extend our Stats class to support modifiers, but first we need to make a decision. This decision requires some context, so let's start with an example. Imagine the player has two items equipped as in Listing 3.5.

```
magic_hat = { add = { ["strength"] = 10 } }
magic_sword = { add = { ["strength"] = 5, ["speed"] = 5 } }
```

Listing 3.5: The data for a magic hat and sword.

In this example, the player has two items and they both affect the str stat. One adds 10 and one adds 5. What happens when the player is wearing both? How should the modifiers stack? It seems obvious the modifiers should be added together, giving the player a +15 str bonus.

Consider the case with multiple mult modifiers, as shown below in Listing 3.6.

```

curse =
{
    mult =
    {
        ["strength"] = -0.5,
        ["speed"] = -0.5,
        ["intelligence"] = -0.5
    }
}
bravery =
{
    mult =
    {
        ["strength"] = 0.1,
        ["speed"] = 0.1,
        ["intelligence"] = 0.1
    }
}

```

Listing 3.6: The data for curse and bravery effect modifiers.

In this example the player is afflicted by a curse spell that halves the basic stats, but he's boosted by a bravery spell that increases all stats by 10%. How do we deal with applying both of these modifiers? Just like the addition modifier we can add them up, giving -0.4, which means stats are reduced by 40% rather than 50%.

This leads to the decision we must make. Do we do the apply the multiply modifiers first or the addition modifiers? The order is important because it changes the result. This is down to preference. The only key rule is that they must be applied as sets; otherwise the order in which each modifier is evaluated changes the bonuses, which would be very unclear.

Doing the multiplies last makes multipliers more powerful, and that's what I've opted for here. This works both ways; bonuses are stronger but so are curses. With this decision made, we can add the modifier code to the Stats class, as shown in Listing 3.7.

```

-- id = used to uniquely identify the modifier
-- modifier =
-- {
--     add = { table of stat increments }
--     mult = { table of stat multipliers }
-- }
function Stats:AddModifier(id, modifier)
    self.mModifiers[id] =
    {

```

```

        add      = modifier.add or {},
        mult    = modifier.mult or {}
    }
end

function Stats:RemoveModifier(id)
    self.mModifiers[id] = nil
end

function Stats:Get(id)
    local total = self.mBase[id] or 0
    local multiplier = 0

    for k, v in pairs(self.mModifiers) do
        total = total + (v.add[id] or 0)
        multiplier = multiplier + (v.mult[id] or 0)
    end

    return total + (total * multiplier)
end

```

Listing 3.7: Modifier code doing additions before multiplication. In Stat.lua.

In order to apply modifiers and remove them, we've added two new functions called `AddModifier` and `RemoveModifier`.

The `AddModifier` function takes two parameters: an id to uniquely identify the modifier, and the modifier table itself. `AddModifier` is called when the character equips a weapon, is affected by a spell, or uses a special skill. The modifier table may contain `add` and `mult` tables but both are optional, and if they don't exist empty tables are used.

It's important that the modifier id is unique. Let's consider an example to illustrate why that's the case. If the player has two rings of +1 strength, the rings' ids need to differ. If each id was "ring_str_plus_one", then the rings' modifiers would overwrite each other. Instead of giving +2 strength, the rings would only give +1. This is because modifiers are stored by id in the `mModifiers` table.

If the ids are totally unique ("000001" and "000004"), then the modifiers don't clash and everything works as expected.

`RemoveModifier` simply looks up the modifier by its id and removes it from the `mModifiers` table. This is used when unequipping a weapon or when a spell expires.

The `Get` function returns a base stat with all modifications applied. In a combat simulation, we'll rarely need a base stat; we'll use the fully modified one. The fully modified stat is calculated by first getting the base number and then applying all modifiers.

Modifiers are combined together using a simple loop. The add modifiers are added to the base stat directly. The multiplier modifiers have their values summed and are applied after the add modifiers.

Let's run through some quick examples to show it working. Copy the code from Listing 3.8 into your main.lua file.

```
local stats =
Stats:Create
{
    ["hp_now"] = 300,
    ["hp_max"] = 300,
    ["mp_now"] = 300,
    ["mp_max"] = 300,
    ["strength"] = 10,
    ["speed"] = 10,
    ["intelligence"] = 10,
}

function PrintStat(id)
    local base = stats:GetBase(id)
    local full = stats:Get(id)
    local str = string.format("%s>%d:%d", id, base, full)
    print(str)
end

PrintStat("strength") -- str>10:10
```

Listing 3.8: Printing out stats with and without modifiers. In main.lua.

In Listing 3.8 we create a Stats object and print out its modified and unmodified value for the strength stat. We've added no modifiers at this point, so both values are the same.

Let's add some modifiers. We'll add this code in main.lua just beneath the last print statement. See the code below in Listing 3.9.

```
magic_hat =
{
    id = 1,
    modifier =
    {
        add =
        {
            ["strength"] = 5
        },
    }
}
```

```

        }
}

stats:AddModifier(magic_hat.id, magic_hat.modifier)
PrintStat("strength") -- str>10:15

```

Listing 3.9: Adding magic hat that modifies the character's strength. In main.lua.

When our character wears the hat of +5 str, the base value remains at 10 but the total goes to 15, as expected. Equipping an additional item in the form of the magic sword increases the str even more, as shown in Listing 3.10.

```

magic_sword =
{
    id = 2,
    modifier =
    {
        add =
        {
            ["strength"] = 5,
        }
    }
}

stats:AddModifier(magic_sword.id, magic_sword.modifier)
PrintStat("strength") -- str>10:20

```

Listing 3.10: Adding a magic sword to boost the strength stat even more. In main.lua.

There are now two add modifiers, each adding +5, which combined with the base 10 gives a total of 20 strength. The stacking works as expected. These examples demonstrate the add modifiers. Let's create some mult modifiers in the next example shown below, Listing 3.11.

```

spell_bravery =
{
    id = "bravery",
    modifier =
    {
        mult =
        {
            ["strength"] = 0.1,
            -- other stats omitted
        }
    }
}

```

```

        }
}

stats:AddModifier(spell_bravery.id, spell_bravery.modifier)
PrintStat("strength") -- str>10:22

```

Listing 3.11: The bravery spell boosts strength by 10%. In main.lua.

In our fictitious example game, I like to think the bravery spell adds 10% to all stats, but here we only care about the strength. Our total strength after the +10 from equipment is 20. 10% of 20 is 2. We add those together to get the final strength value of 22. Our character is getting pretty powerful. It's time to bring him down a notch with a final modifier.

In Listing 3.11 we're not using a unique id. There may be certain cases where a clashing id is desirable. Here, for example, the bravery spell uses a non-unique id, which means it can only ever be applied to the creature once. If the ids of all bravery spells are the same, only one can ever be applied.

Below we're going to add one last modifier in Listing 3.12.

```

spell_curse =
{
    id = "curse",
    modifier =
    {
        mult =
        {
            ["strength"] = -0.5,
            -- other stats omitted
        }
    }
}

stats:AddModifier(spell_curse.id, spell_curse.modifier)
PrintStat("strength") -- strength>10:12

```

Listing 3.12: The curse spell cuts strength in half. In main.lua.

In this example, the curse spell has a non-unique id, which means only one curse spell can afflict the character at a time. The curse spell halves all the stats, but again we're only concerned about strength.

All add modifiers are applied first, giving the strength a value of 20. Then the multiplication modifiers are summed, in this case $0.1 + -0.5 = -0.4$. Instead of the curse

reducing the strength by 50%, the +10% of the bravery spell means the strength is only reduced by 40%. 40% of 20 is 8, and 8 subtracted from 20 gives the final strength total of 12. This is great; everything is working as expected!

The current Stat class allows multiple modifiers such as spells and equipment to be tracked easily and applied in a general, scalable way. We'll use this class as a base for the equipment and item code. You can check out a working version of the code so far in combat/2-stats-solution.

Equipment Stats

Weapons and armor make combat more interesting and tactical. They make the combatant more deadly and better equipped to deflect or absorb damage.

Weapons and armor generally share four basic stats.

- **Attack** - How much damage is inflicted
- **Defense** - How much damage the weapon or armor deflects
- **Magic** - How much magic damage is inflicted
- **Resist** - How much magic damage is resisted

To keep things simple, each actor in our game will only ever have one Stat object. Weapons and armor use the stats above, but they're applied as modifiers on the actor's stat object. Let's go over a quick example to demonstrate this in Listing 3.13.

```
half_sword =
{
    name = "Half Sword",
    id = 5,
    stats =
    {
        add = { attack = 5 }
        -- other stats are omitted as they're 0
    }
    -- omitted code
}

spiked_helm_of_necromancy =
{
    name = "Spiked Helm of Necromancy",
    id = 6,
    stats =
    {
        add =
        {

```

```

        attack = 1,
        defense = 3,
        magic = 1,
        resist = 5
    }
}

-- omitted code

}

print(hero.mStats:Get("attack")) -- 0
print(hero.mStats:Get("defense")) -- 0
print(hero.mStats:Get("magic")) -- 0
print(hero.mStats:Get("resist")) -- 0

hero:Equip('weapon', half_sword)
hero:Equip('armor', spiked_helm_of_necromancy)

print(hero.mStats:Get("attack")) -- 6
print(hero.mStats:Get("defense")) -- 3
print(hero.mStats:Get("magic")) -- 1
print(hero.mStats:Get("resist")) -- 5

```

Listing 3.13: Demonstrating weapons and armor using modifiers. In main.lua.

This code snippet shows that all equipment defs are simple tables with a table of modifiers applied when equipped.

The half sword is in a table with an English language name, an id, and a stats value which acts as a modifier. We can add more data to this table as needed. For instance, we might add a field saying it's a "weapon" type, and if there are special rules we can store them here too. Imagine a sword that drinks blood and restores the wielder's health. That would be a special entry in this table.

The helm is defined in exactly the same way, but with different values for the various fields. A hero with no equipment has no attack, defense, magic or resist stats, so if we try to print these out 0 is returned.

Once we equip the helm and sword, their modifiers are applied to the stats of the hero. When attack, defense, magic or resist stats are printed, they all have values and are ready to use in a combat simulation.

Weapons and armor modifiers don't just affect the stats a combatant already has. They can add brand new ones that are only ever associated with equipment. This makes the combat code later much simpler.

A Stat Set Function

In combat, the hp_now and mp_now stats often change. To change a stat we'll use a Set function. It alters the base value of a stat. Add the code in Listing 3.14.

```
function Stats:Set(id, value)
    self.mBase[id] = value
end
```

Listing 3.14: A stat set function. In Stats.lua.

Pretty simple, and now we have an approved way of modifying stats.

Moving On

The stat code is just one component of a full combat system, and it's time to move on. Next we'll cover levels and how levels affect stats.

Levels

As characters progress through the game, they become stronger. This increase in strength is handled by the leveling system, which in turn is closely tied to the stats.

What is a Level?

A level is a number that represents the overall power of a game entity. Player characters and enemies commonly have levels, but much like stats, levels can be assigned to nearly anything.

A level one warrior might be a youth heading out from a small village ready for his first adventure. A level fifty warrior is a veteran of many battles; she's strong, fast and quick witted. The level fifty warrior has grown powerful through experience to make her physically and mentally tough. The level number represents this accumulated experience.

Level Origins

Like almost everything in JRPGs, levels have been passed down from tabletop roleplaying games. In D&D, levels serve much the same function as in modern JRPGs; they mark out how powerful an entity is and provide a sense of progress for the player.

How Levels Work

Characters usually start the game at level one and increase their level by gaining experience. Experience is represented by experience points, also known as XP. Experience points measure how much experience a character currently has. Once a character's XP reaches a certain threshold, their level is increased, the XP is reset to 0, and the threshold for the next level is raised.

Game entities use three numbers to track their level: current level, XP, and "XP required for next level". An example level data structure can be seen in Listing 3.15.

```
level_data =  
{  
    level = 1,  
    xp = 5,  
    next_level = 100,  
}
```

Listing 3.15: An example level data structure.

The xp number tracks how close the entity is to advancing to the next level. XP is gained by defeating enemies but can also be given out for any worthy accomplishment. Each successive level requires more XP to attain, therefore gaining levels gradually becomes tougher. Level thresholds are calculated using a special formula that we'll be covering shortly. When a character gains a level, the game acknowledges the achievement with some fanfare and rewards the player by increasing stats and possibly giving out special abilities. A level increase usually happens at the end of combat with a special screen to show the player how the character has changed. See Figure 3.1.



Figure 3.1: Levelling up after combat.

The level number may be used by the combat simulation code to determine the outcome of some event, but XP is rarely used in this way; it only tracks the distance to the next level.

Each level bumps the character stats, so as the character's level rises they become stronger. After a level the enemies are suddenly a little easier, but very quickly the player discovers more powerful enemies and the difficulty plateaus again. If combat ever gets too easy, it becomes boring. You can see this illustrated in Figure 3.2.

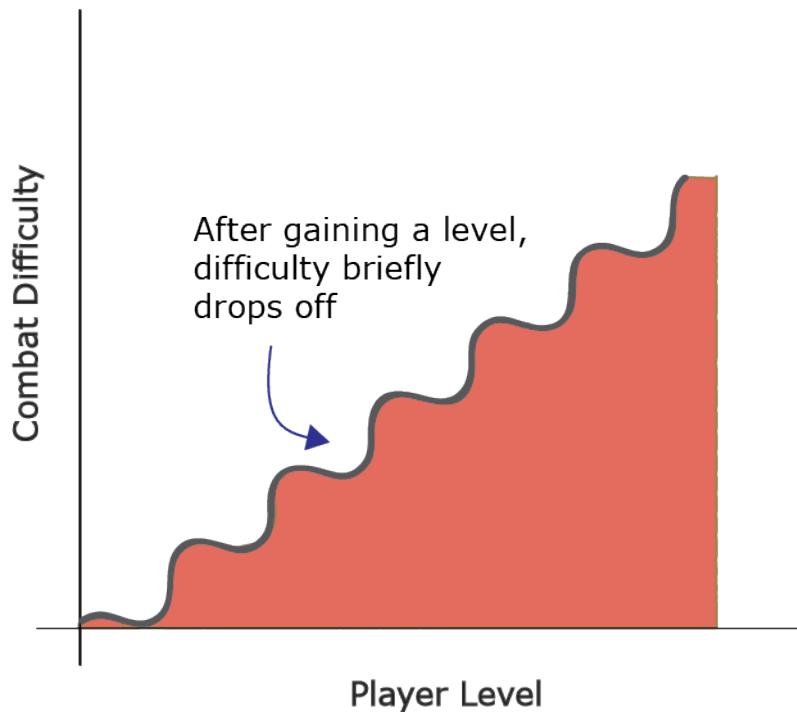


Figure 3.2: The difficulty ramp.

Why Use Levels

Levels and experience points are a popular mechanic with both players and designers for good reason. Here are some features of the mechanic.

Sense of Achievement

Gaining a level gives a sense of achievement and acknowledges the player's effort.

When the level number increases, it gives the player a pat on the back and says "Good job!" The number increasing is a small reward all by itself, but RPGs usually give out additional rewards by increasing the character stats and sometimes unlocking special abilities.

The value in gaining a level is related to how difficult it is to achieve. The distance between levels gives them meaning. In order for a level to have value, the player needs to struggle and overcome adversity. Higher levels require more XP and take longer to attain, making the time invested proportional to the reward.

The Hero's Journey

In a JRPG, the characters are literally on a hero's journey². The level number indicates how far they've come and hints at how far is left to go.

Drip-feed Complex Mechanics

JRPGs have quite complicated battle mechanics. Combat may include status effects, various types of spell, elements, chain attacks, and who knows what else. Combat can be so complicated that it's too much to explain to the player all at once. Instead, as the player slowly levels up, mechanics are introduced in understandable chunks. This drip-feed helps make the game more accessible and shallows out the learning curve.

At level 1, only the most basic mechanics are available, making combat straightforward for a new player. Choices are restricted, allowing the player to quickly try them all out. Gaining levels unlocks new abilities and spells which the player can immediately try out and get comfortable with before the next bit of functionality is introduced.

This drip-feed also applies to enemies. Early in the game, enemies exhibit simple behaviors and few abilities, but as the characters level up and progress to new areas, they meet new monsters with special attacks and more advanced tactics. For the player to encounter these more advanced enemies, they must first defeat simpler ones by mastering the basic combat techniques.

Control Content Flow

Levels help drip-feed mechanics but they also help control how the player navigates the world.

Is there a cave where the monsters are far too hard? Maybe the player can't tackle it yet, but they'll remember the cave. It will become a goal, something to return to when stronger.

Levels also control how quickly a character can dispatch monsters, which controls how fast and easily a character can travel from place to place and how long a play-through of the game takes.

Level Formulae

The meat of a leveling system concerns how the level thresholds are defined. We need a formula that, given a level, tells us how many experience points are required to reach the next one. This kind of function is shown in Listing 3.16.

²The Hero's Journey is a plot structure described in the book *The Hero with a Thousand Faces* by Joseph Campbell. Worth reading if you're into mythology and plot.

```
xp_for_level_2 = next_lvl(1)
```

Listing 3.16: Signature of a function to calculate XP thresholds for levels.

The formula should return an increasing amount of XP per level and be easy to tweak for balancing purposes.

Example Level Formulae

Nearly all RPGs have some leveling function hidden inside them. Let's take a look at some functions used in existing games. (These have been reverse engineered and may not be totally accurate!)

Lua doesn't have a round function, to round a number to the nearest whole number, but we're going to need it so we'll define our own here in Listing 3.17.

```
-- round(0.7) -> 1
-- round(5.32) -> 5
function round(n)
    if n < 0 then
        return math.ceil(n - 0.5)
    else
        return math.floor(n + 0.5)
    end
end
```

Listing 3.17: A round function for Lua.

It's important to get your hands dirty to get a better understanding of these systems. Open Excel or your programming editor and tinker with some of the examples below.

Original D&D

Dungeons and Dragons is the game many early JRPGs used for their inspiration, so it seems only fair to include its level function. Figure 3.3 shows the leveling graph and Listing 3.18 shows the code.

```
function NextLevel(level)
    return 500 * (level ^ 2) - (500 * level)
end
```

Listing 3.18: D&D levelling function.



Figure 3.3: The level ramp for Dungeons and Dragons.

Generation 1 Pokémon

Pokémon actually has two leveling systems. This is the faster of the two. Figure 3.4 shows the leveling graph and Listing 3.19 shows the code.

```
function NextLevel(level)
    return round((4 * (level ^ 3)) / 5)
end
```

Listing 3.19: Pokémon leveling function.

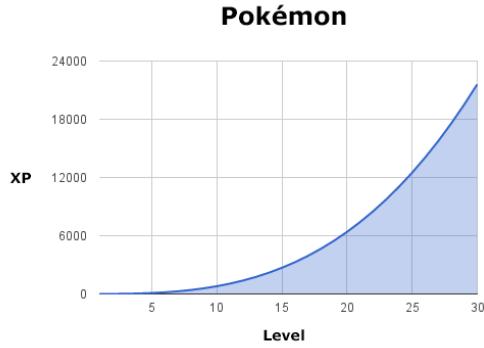


Figure 3.4: The level ramp for Pokemon Generation 1.

Disgea

Disgea applies this formula to the first 99 levels. After that it changes things up. Figure 3.5 shows the leveling graph and Listing 3.20 shows the code.

```
function NextLevel(level)
    return round( 0.04 * (level ^ 3) + 0.8 * (level ^ 2) + 2 * level)
end
```

Listing 3.20: Disgea levelling function.

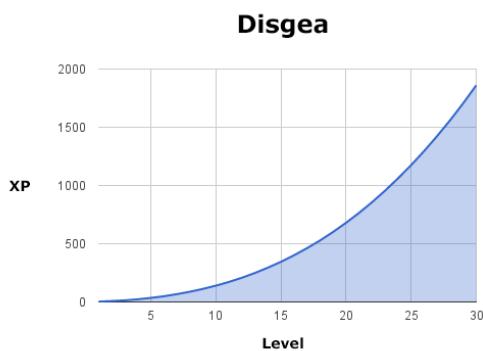


Figure 3.5: The level ramp for Disgea.

Common Themes in Levelling Functions

All the examples use exponential functions to get a steep leveling curve. The steeper the curve, the greater the amount of XP required for each successive level and probably the more time the player has to invest to attain it. Each of the level function is tailored for each game. For example, in Dungeons and Dragons, games progress for weeks and high-level characters are rare, so the level curve is extremely steep. Pokémon has many monsters, each of which can be leveled up, so it has a more modest curve. The shallowest curve of all is Disgea, as nearly every item in the game has some type of leveling applied to it.

A leveling system doesn't exist in isolation. It's affected by the number of monsters encountered and how much XP they give out. This means we may need to revise it when creating the game content, deciding xp for enemies, and determining how quickly we want the player to progress.

Creating a Levelling Function

The level formula we'll use in our game is loosely based on Final Fantasy. The basic formula is shown in Listing 3.21.

```
function NextLevel(level)
    local exponent = 1.5
    local baseXP = 1000
    return math.floor(baseXP * (level ^ exponent))
end
```

Listing 3.21: A simple leveling function.

This formula is pretty simple, but let's go through it.

Assume the exponent is 1. This means that for each level, the amount of XP needed increases by baseXP. If baseXP is 1000, level 1 requires 1000xp, level 2 2000xp, level 3 3000xp, and so on. This is a simple step increase; the levels don't get increasingly hard. If killing ten wolves gets you to level 1, to get to level 2 you only need to kill another 10 wolves; obviously this isn't ideal. We want each level to be more challenging than the last. To achieve this we need to use an exponent.

The exponent represents the increase of difficulty between levels. Let's set it at 1.5. Level 1 now requires 1000xp, level 2 requires 2828xp, level 3 5196xp, etc. This ever-increasing difficulty better suits our needs.

When tweaking our formula, we can alter the baseXP for how much XP we generally want a level to cost and the exponent for how increasingly difficult we want attaining those levels to become.

How Levels Affect Stats

The leveling and stats system are closely linked. Each level increases the character stats but this doesn't happen in a uniform way. Different characters level up in different ways.

The personality of a JRPG character is tied closely to how their stats are distributed. A thief-like character will have better speed increases from a level than a hulking warrior. Different characters have different strategies for stat growth.

Final Fantasy games usually have a slightly random stat growth from 1 to 3 points for the base stats and a larger increase for HP and MP. Our system will follow this convention.

Dice

To handle the stat growth, we'll use dice. We'll use one type of roll for a fast rate and another type of roll for a slow rate. Dice are an intuitive way to think about different

distributions. Rolling 1 die with 6 sides returns a number from 1 to 6, all equally likely, as is shown in Figure 3.6.

1D6 Distribution

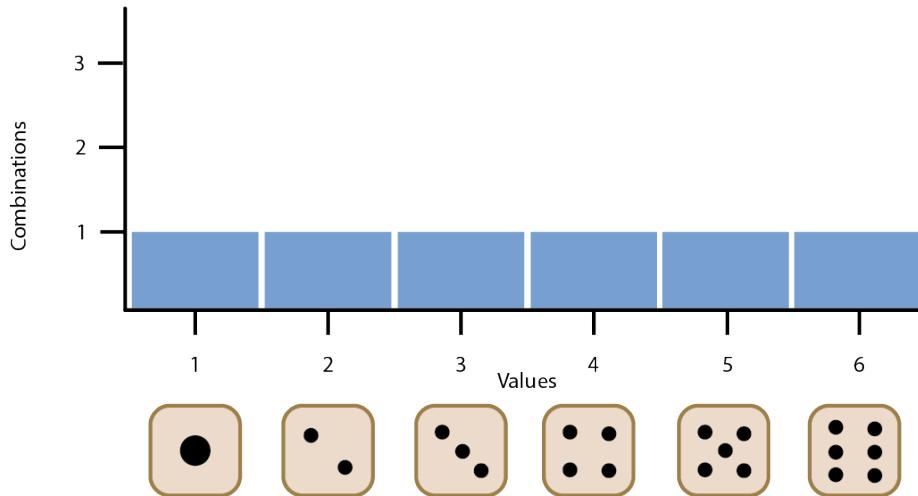


Figure 3.6: The distribution for one die with six sides.

Roll 2 dice with 3 sides, add the results, and you'll get a range from 2 to 6. The distribution is weighted towards the middle. The middle-heavy distribution arises because there are more possible combinations of rolls that make a 4 than a 2 or 6.

Here's a shorthand for writing down dice throws: 1D6 means one throw of a six-sided die. 2D6 would be two rolls of a six-sided die. The distribution for this roll is illustrated in Figure 3.7. This notation is easy to use so we'll make our dice code understand it.

2D6 Distribution

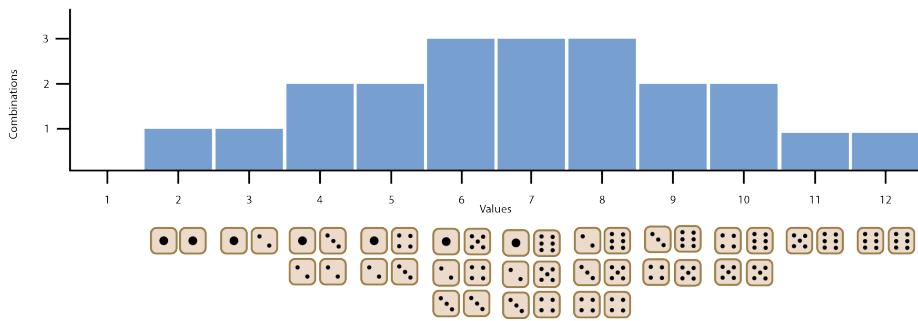


Figure 3.7: The distribution for two dice with six sides.

Example combat/levels-1 contains a project with an empty Dice.lua file.

Below is a definition of the grammar for the dice rolls. Don't worry if it seems a little formal. We'll jump into the code shortly.

```
[roll] = [number]D[number]
[rolls] = [roll] | [roll]+[number]
[throw] = [rolls] | [rolls] " " [throw]
```

This grammar can match the following dice roll examples.

- 1D6 - One roll of a six-sided die.
- 2D6+3 - Two rolls of a six-sided die and add three.
- 1D6 2D8+10 - One roll of a six-sided die plus two rolls of an eight-sided die plus ten.

In the Dice.lua class, add the following class definition and constructor shown in Listing 3.22.

```
Dice = {}
Dice.__index = Dice

function Dice:Create(diceStr)
    local this =
    {
        mDice = {}
    }
    setmetatable(this, self)
    this:Parse(diceStr)
    return this
end

function Dice:Parse(diceStr)
    local len = string.len(diceStr)
    local index = 1
    local allDice = {}

    while index <= len do
        local die
        die, index = self:ParseDie(diceStr, index)
        table.insert(self.mDice, die)
        index = index + 1 -- eat ' '
    end
end
```

Listing 3.22: Constructor and first part of the code we'll need for a Dice class. In Dice.lua.

This code in Listing 3.22 allows a dice object to be created with a string, such as "1d6". The constructor sets up the class and passes the string to a method called Parse. Parse has an index variable for the position in the string and it passes this, with dice string, to a function called ParseDie until the index is at or past the end of the string. Parse, in plain English, just says "keep parsing dice from the string until we reach the end". ParseDie returns a table describing a die and a new index for the string. Each new die is added to the allDice table.

Let's add the final parsing functions and then move on to the rest of the class. Copy the code from Listing 3.23.

```
function Dice:ParseDie(diceStr, i)
    local rolls
    rolls, i = self:ParseNumber(diceStr, i)

    i = i + 1 -- Move past the 'D'

    local sides
    sides, i = self:ParseNumber(diceStr, i)

    if i == string.len(diceStr) or
        string.sub(diceStr, i, i) == ' ' then
        return { rolls, sides, 0 }, i
    end

    if string.sub(diceStr, i, i) == '+' then
        i = i + 1 -- move past the '+'
        local plus
        plus, i = self:ParseNumber(diceStr, i)
        return { rolls, sides, plus }, i
    end

end

function Dice:ParseNumber(str, index)

    local isNum =
    {
        ['0'] = true,
        ['1'] = true,
        ['2'] = true,
        ['3'] = true,
        ['4'] = true,
```

```

[ '5' ] = true,
[ '6' ] = true,
[ '7' ] = true,
[ '8' ] = true,
[ '9' ] = true
}

local len = string.len(str)
local subStr = {}

for i = index, len do

    local char = string.sub(str, i, i)

    if not isNum[char] then
        return tonumber(table.concat(subStr)), i
    end

    table.insert(subStr, char)

end

return tonumber(table.concat(subStr)), len

```

Listing 3.23: The second part of the Dice class. In Dice.lua.

These two methods are a little longer but they're both quite simple. ParseDie parses the rolls number, advances the index past the letter D, and then parses the sides number. It checks if the sides number is at the end of the string or at a space; if not, it advances the index past the + and reads a final number, the modifier. It then returns the die as a table of 3 parts: the rolls, the sides, and the modifier.

ParseNumber reads all characters that are digits and returns the new number and index. It's not as important to go through step by step but if you're interested it should be quite easy to read.

That's the parsing finished. Let's add a little more functionality to the Dice class as shown below in Listing 3.24.

```

-- Notice this uses a '.' not a ':' meaning the function can be called
-- without having a class instance. e.g Dice.RollDie(1, 6) is ok
function Dice.RollDie(rolls, faces, modifier)
    local total = 0

```

```

    for i = 1, rolls do
        total = total + math.random(1, faces)
    end
    return total + (modifier or 0)
end

function Dice:Roll()
    local total = 0

    for _, die in ipairs(self.mDice) do
        total = total + Dice.RollDie(unpack(die))
    end

    return total
end

```

Listing 3.24: Adding functions to the Dice class, to actually roll the dice.

The class has a function called Dice.RollDie which can be called directly as shown in Listing 3.25.

```
Dice.RollDie(1, 6) -- some number 1..6
```

Listing 3.25: A one-liner to roll some dice.

The class can also be instantiated so it can be called multiple times as shown in Listing 3.26.

```

local r1d6 = Dice>Create("1d6")

print(r1d6:Roll())
print(r1d6:Roll())

```

Listing 3.26: Rolling a six-faced die.

More complicated rolls can be constructed too, as shown in Listing 3.27

```

local roll = Dice>Create("1d6 1d8+10")

print(roll:Roll())

```

Listing 3.27: Rolling multiple dice.

That's all we need from the dice rolling code. Next we'll use it to implement our stat growth strategies. The full dice rolling code with examples is available in example combat/levels-1-solution.

Stat Growth

Now that we have a way to roll dice, we can increase character stats when leveling up. Both monsters and characters have levels and stats. We use the term *actor* to refer to any game entity that might have stats or levels. Example combat/levels-2 contains the code so far.

Copy the code from Listing 3.28 into your main.lua. We'll worry about moving this into a more appropriate file later.

```
local Growth =
{
    fast = Dice:Create("3d2"),
    med = Dice:Create("1d3"),
    slow = Dice:Create("1d2")
}
```

Listing 3.28: Creating growth strategies using Dice.

This is just a simple table with 3 different types of dice roll: slow, med, and fast. These rolls are used to increase stats on level ups.

Leveling up can be complicated. For instance, carrying a certain item or having certain abilities selected may influence or control how stats grow. In our game we'll keep things simple. Each level increases stats, and later we'll unlock some abilities too.

Our RPG contains three characters: the main hero, a thief, and a mage. We might define them as below in Listing 3.29.

```
heroDef =
{
    stats = ... -- starting stats
    statGrowth =
    {
        ["hp_max"] = Dice:Create("4d50+100"),
        ["mp_max"] = Dice:Create("2d50+100"),
        ["str"] = Growth.fast,
        ["spd"] = Growth.fast,
        ["int"] = Growth.med,
    },
    -- additional actor definition info
}

thiefDef =
{
    stats ... -- starting stats
```

```

statGrowth =
{
    ["hp_max"] = Dice:Create("4d40+100"),
    ["mp_max"] = Dice:Create("2d25+100"),
    ["str"] = Growth.fast,
    ["spd"] = Growth.fast,
    ["int"] = Growth.slow,
},
-- additional actor definition info
}

mageDef =
{
    stats = ... -- starting stats
    statGrowth =
    {
        ["hp_max"] = Dice:Create("3d40+100"),
        ["mp_max"] = Dice:Create("4d50+100"),
        ["str"] = Growth.med,
        ["spd"] = Growth.med,
        ["int"] = Growth.fast,
    },
    -- additional actor definition info
}

```

Listing 3.29: Different actor definitions with different stat growth strategies.

The hero character has the best stat growth. Strength and speed both grow quickly while intelligence grows at a moderate rate. His HP increases by 4d50+100 and his MP by 2d50+10. The thief is similar, but weaker in both HP and MP and not as quick to increase intelligence. The mage growth in strength and speed is moderate but rapid for intelligence. He has lower HP gains than the thief but higher MP gains than anyone. Laying the actor data out in definition tables as in Listing 3.29 makes it easy to see the big picture and tweak as needed.

To test out these definitions, we're going to create a class called Actor. Copy the code from Listing 3.30 into Actor.lua.

```

Actor = {}
Actor.__index = Actor

function Actor:Create(def)

    local this =
    {

```

```

        mDef = def,
        mStats = Stats>Create(def.stats),
        mStatGrowth = def.statGrowth,
        mXp = 0,
        mLevel = 1,
    }

    this.mNextLevelXP = NextLevel(this.mLevel)

    setmetatable(this, self)
    return this
end

```

Listing 3.30: Adding Stats and growth strategies to the Actor class.

To get the Actor class ready to deal with levels, we need a function to add XP and a way to detect when a level is gained.

At this point it's worth taking a moment to consider when and how we're going to display this information to the user. In a standard JRPG, level and XP information is displayed in a battle summary screen after combat. The battle summary screen displays XP, stats and level numbers from before combat and then counts up the XP gained for each character. If a level is gained, it's shown on this screen. When we make our RPG, we'll be programming this screen. By carefully structuring the code now, we'll make things easier later.

Copy the code from Listing 3.31.

```

function Actor:ReadyToLevelUp()
    return self.mXp >= self.mNextLevelXP
end

function Actor:AddXP(xp)
    self.mXp = self.mXp + xp
    return self:ReadyToLevelUp()
end

```

Listing 3.31: Extending the Actor class to support XP gaining.

The AddXP function increases the actor's XP and returns true if a level has been gained. This function is not responsible for actually applying the level up.

To apply the level up data, we'll first use a function called CreateLevelUp. This returns a table with the leveled up stats, new abilities, and so forth. Copy the code from Listing 3.32.

```

function Actor:CreateLevelUp()

    local levelup =
    {
        xp = - self.mNextLevelXP,
        level = 1,
        stats = {},
    }

    for id, dice in pairs(self.mStatGrowth) do
        levelup.stats[id] = dice:Roll()
    end

    -- Additional level up code
    -- e.g. if you want to apply
    -- a bonus every 4 levels
    -- or heal the players MP/HP

    return levelup
end

```

Listing 3.32: CreateLevelUp function.

This levelup table returned by CreateLevelUp describes the next level for the actor. We can inspect the levelup table and display the values on screen before applying it. Let's say a new ability is unlocked. We'd see a single entry for it in the levelup table. If the new ability was just added directly to the character with all the other abilities, it would be harder to discover and display to the player.

ApplyLevel is the function that actually increases the player level and stats (and decreases the XP). Copy the implementation from Listing 3.33.

```

function Actor:ApplyLevel(levelup)
    self.mXp = self.mXp + levelup.xp
    self.mLevel = self.mLevel + levelup.level
    self.mNextLevelXP = NextLevel(self.mLevel)

    assert(self.mXp >= 0)

    for k, v in pairs(levelup.stats) do
        self.mStats.mBase[k] = self.mStats.mBase[k] + v
    end

    -- Unlock any special abilities etc.
end

```

Listing 3.33: Adding the ApplyLevel function to the Actor class. In Actor.lua.

That's all the leveling up code needed for now. Example combat/levels-2-solution contains the complete code.

Level-Up Test Program

Let's write a small program to demonstrate the leveling code so far. The program adds xp to an actor, and for each level gained it prints out the changes.

Example combat/levels-3 contains the code so far. Let's begin by adding some helper functions.

PrintLevelUp is a helper function that prints the levelup table in a nicely formatted way. There's also an ApplyXP function to add XP to an actor and to call PrintLevelUp for each level gained. Copy it into your main.lua from Listing 3.34.

```
function PrintLevelUp(levelup)

    local stats = levelup.stats

    print(string.format("HP:+%d MP:+%d",
        stats["hp_max"],
        stats["mp_max"]))

    print(string.format("str:+%d spd:+%d int:+%d",
        stats["str"],
        stats["spd"],
        stats["int"]))
    print("")

end

function ApplyXP(actor, xp)
    actor:AddXP(xp)

    while(actor:ReadyToLevelUp()) do

        local levelup = actor>CreateLevelUp()
        local levelNumber = actor.mLevel + levelup.level
        print(string.format("Level Up! (Level %d)", levelNumber))
        PrintLevelUp(levelup)
        actor:ApplyLevel(levelup)

    end
end
```

Listing 3.34: Level up print and ApplyXP functions.

The ApplyXP function adds XP to the actor and then uses a while-loop to count up all the levels gained, if any, printing out each levelup table and applying them to the actor.

Let's write a small script to call the ApplyXP function. See Listing 3.35.

```
hero = Actor>Create(heroDef)
ApplyXP(hero, 10001)

-- Print out the final stats
print("==XP applied==")
print("Level:", hero.mlevel)
print("XP:", hero.mXp)
print("Next Level XP:", hero.mNextLevelXP)

local stats = hero.mStats

print(string.format("HP:%d MP:%d",
    stats:Get("hp_max"),
    stats:Get("mp_max")))

print(string.format("str:%d spd:%d int:%d",
    stats:Get("str"),
    stats:Get("spd"),
    stats:Get("int")))
```

Listing 3.35: An example of applying XP.

The example combat/levels-3-solution contains this script. In this example, the hero is given 10,001 XP points, which are enough to level the hero up to level 4, with a little XP left over. You can see the console output in Figure 3.8. This is a good script to experiment with, changing the stat growth strategies, the amount of XP awarded, and so on, to get a feel for how the leveling system we've built hangs together.

```
[LUASTATE|Manifest] DESTROYED
Reloading project.

Calling main_script main.lua at main.lua
Level Up! (Level 2)
HP:+210 MP:+143
str:+6 spd:+4 int:+2

Level Up! (Level 3)
HP:+227 MP:+132
str:+5 spd:+4 int:+2

Level Up! (Level 4)
HP:+219 MP:+132
str:+5 spd:+4 int:+3

==XP applied==
Level: nil
XP: 977
Next Level XP: 8000
HP:956 MP:707
str:26 spd:22 int:17
Reload success:
```

Figure 3.8: The console output from the level up test program.

Next Steps

We've covered the default level-up system most JRPGs use. It's a great starting point to customize for the needs of your own games! For instance, instead of XP you might have souls that are sucked from an enemy, or some kind of reverse XP system where you remove disadvantages from your player to restore him to super demi-god like state! This is your codebase and you can change it as desired.

The Party

Adventures in the original Dungeons and Dragons often began by gathering a party to complete a fantastical quest. Each member of the party was controlled by a player sitting at a gaming table. The party system survived the move to JRPGs but the multiplayer aspect did not³. In video games, multiple protagonists are mainly used to drive the narrative. In this section we'll add support for a party made up of multiple adventurers.

³At least initially. Secret of Mana on the SNES supports cooperative multiplayer. Prior to that, university mainframes were a hotbed of multiplayer dungeon crawlers, collectively known as MUDs (Multi-User Dungeons).

In the game world the party of adventurers has some common reason for travelling together. They may not always get along – it's often more interesting if they don't – but for better or worse they have a task to complete. The personality and fighting style of each adventurer should be carefully considered. The best games develop the characters over the course of the game. Their past secrets are revealed and they grow as people.

Games commonly begin with a single main character who finds and recruits the rest of the party. The number of party members in play at one time is limited to make the game more manageable. In combat, it's rare to have more than four characters at once, and in our game we have a maximum limit of three.

Implementing the Party

In the Exploration part of the book we built the start of the in-game menu. We'll extend this as we begin the party code. We'll add the notion of a party, give each character in the party some stats, and make a status screen to display them.

In order to start defining characters, we need to state exactly which stats they'll use. The stats are divided into two groups, those associated with the character and those associated with equipment. Here they are listed.

Character Stats

- **HP** - Health Points.
- **MP** - Magic Points, also used for special abilities.
- **Strength** - How hard the player hits.
- **Speed** - How quickly the player hits. Chance to dodge attacks.
- **Intelligence** - How powerful spells are.

Item Stats

- **Attack** - Damage the item does.
- **Defense** - Damage the item deflects.
- **Magic** - The amount of magic power the item gives.
- **Resist** - The amount of magic power the item deflects.

Party Member Definitions

Run example combat/party-1 and have a look around. You'll see a map like Figure 3.9. This is the arena. The example contains new art assets for the arena tiles and character portraits.



Figure 3.9: The area outside the arena.

The arena is small and currently quite empty. The code is a stripped-down version of the Dungeon mini-game. We'll be using it as a base to introduce the combat. As before, pressing the Alt key opens the in-game menu. There's only one option in the menu at the moment, "Inventory".

To form a party, we first need a concept of a party member. Up until now character information has been stored in the EntityDefs.lua class in the gCharacters table. We'll use the gCharacters def for the party member on the map, but once the party member joins the party we'll use its actor def.

Let's create and add a new file StatDefs.lua. This is where we'll store the stat growth definitions.

Copy the code in Listing 3.36 into the file.

```
gStatGrowth =  
{  
    fast = Dice:Create("3d2"),  
    med = Dice:Create("1d3"),  
    slow = Dice:Create("1d2")  
}
```

Listing 3.36: The Stat growth rates. In StatDefs.lua.

Make sure this is included in Dependencies.lua before EntityDefs.lua. You can include StatDefs.lua immediately above EntityDefs.lua as shown below in Listing 3.37.

```
"StatDefs.lua",
"EntityDefs.lua",
```

*Listing 3.37: EntityDefs.lua makes use of StatDefs.lua, so stats must be included first.
In Dependencies.lua.*

We'll use this gStatGrowth table directly in the character definitions.

Next create a file called PartyMemberDefs.lua. Here we define the attributes of all the people who might join the party. When adding PartyMemberDefs.lua to the manifest, add it just under StatsDefs.lua.

We'll make three characters; the hero, the mage, and the thief. I'm going to call the Hero "Seven", the thief "Jude", and the mage "Ermis".

The basic data we need for a party member is an id, starting stats, stat growth, name, starting level, xp, portrait image, and the actions they can do in combat. That's quite a lot! The id uniquely identifies the party member, and the characters on the map will contain this reference id.

Make sure the portrait textures are added to the game. There are three portraits in the art directory: hero_portrait.png, thief_portrait.png, and mage_portrait.png. Add a reference to each portrait in the manifest.lua.

Copy the full definitions for each character from Listing 3.38.

```
gPartyMemberDefs =
{
    hero =
    {
        id = "hero",
        stats =
        {
            ["hp_now"] = 300,
            ["hp_max"] = 300,
            ["mp_now"] = 300,
            ["mp_max"] = 300,
            ["strength"] = 10,
            ["speed"] = 10,
            ["intelligence"] = 10,
        },
        statGrowth =
        {
            ["hp_max"] = Dice:Create("4d50+100"),
```

```

["mp_max"] = Dice:Create("2d50+100"),
["strength"] = gStatGrowth.fast,
["speed"] = gStatGrowth.fast,
["intelligence"] = gStatGrowth.med,
},
portrait = "hero_portrait.png",
name = "Seven",
actions = { "attack", "item" }
},
thief =
{
    id = "thief",
    stats =
    {
        ["hp_now"] = 280,
        ["hp_max"] = 280,
        ["mp_now"] = 150,
        ["mp_max"] = 150,
        ["strength"] = 10,
        ["speed"] = 15,
        ["intelligence"] = 10,
    },
    statGrowth =
    {
        ["hp_max"] = Dice:Create("3d40+100"),
        ["mp_max"] = Dice:Create("4d50+100"),
        ["strength"] = gStatGrowth.med,
        ["speed"] = gStatGrowth.med,
        ["intelligence"] = gStatGrowth.fast,
    },
    portrait = "thief_portrait.png",
    name = "Jude",
    actions = { "attack", "item" }
},
mage =
{
    id = "mage",
    stats =
    {
        ["hp_now"] = 200,
        ["hp_max"] = 200,
        ["mp_now"] = 250,
        ["mp_max"] = 250,
        ["strength"] = 8,
        ["speed"] = 10,
    }
}

```

```

        ["intelligence"] = 20,
    },
    statGrowth =
    {
        ["hp_max"] = Dice:Create("3d40+100"),
        ["mp_max"] = Dice:Create("4d50+100"),
        ["strength"] = gStatGrowth.med,
        ["speed"] = gStatGrowth.med,
        ["intelligence"] = gStatGrowth.fast,
    },
    portrait = "mage_portrait.png",
    name = "Ermis",
    actions = { "attack", "item" }
},
}

```

Listing 3.38: Defining the three party members. In PartyMemberDefs.lua.

The definitions in Listing 3.38 are used to set up the Actor object when a character joins the party. The stats table is used to set up a Stats object and the portrait string is used to create a portrait sprite. The actions table defines what the character can do in combat. A more developed JRPG might include actions like “dragon_magic”, “steal”, “summon”, etc.

Ok, we’ve defined the possible party members. Next we need a way to store who is currently in the party. For that we’ll need to create a Party class.

The Party Class

The party class records the characters in the party.

Create a file called Party.lua and add it to the dependencies and manifest. Copy the code shown below from Listing 3.39.

```

Party = {}
Party.__index = Party
function Party:Create()
    local this =
    {
        mMembers = {}
    }

    setmetatable(this, self)
    return this
end

```

```

function Party:Add(member)
    self.mMembers[member.mId] = member
end

function Party:RemoveById(id)
    self.mMembers[id] = nil
end

function Party:Remove(member)
    self:RemoveById(member.mId)
end

```

Listing 3.39: Party simple class to store the party members. In Party.lua.

There's only ever one party of adventurers, so we only need one instance of the party. We store the party in the World class. Copy the code from Listing 3.40.

```

World = {}
World.__index = World
function World:Create()
    local this =
    {
        -- code omitted
        mParty = Party:Create()
    }

```

Listing 3.40: Adding a party to the world. In World.lua.

At this point the party is empty. The party deals with actors, so next let's define an Actor class to fill it up. The Actor class is based on the one we created in the Levels section.

The Actor Class

An actor is any creature or character that participates in combat and therefore requires stats, equipment, etc.

We'll begin by creating a new file, LevelFunction.lua, and adding it to the project. This file stores our level-up function. The code is the same as we've seen before and you can copy it from Listing 3.41.

```

function NextLevel(level)
    local exponent = 1.5
    local baseXP = 1000
    return math.floor(baseXP * (level ^ exponent))
end

```

Listing 3.41: LevelFunction.lua.

With the NextLevel function defined we can make the Actor class.

Create a new file, Actor.lua, and add it to the manifest and dependencies files. Copy the constructor for the Actor class from Listing 3.42.

```

Actor = {}
Actor.__index = Actor
function Actor:Create(def)

    local growth = def.statGrowth or {}

    local this =
    {
        mName = def.name,
        mId = def.id,
        mActions = def.actions,
        mPortrait = Sprite.Create(),
        mStats = Stats:Create(def.stats),
        mStatGrowth = growth,
        mXP = def.xp or 0,
        mLevel = def.level or 1,
        mEquipment =
        {
            weapon = def.weapon,
            armor = def.armor,
            acces1 = def.acces1,
            acces2 = def.acces2,
        }
    }

    if def.portrait then
        this.mPortraitTexture = Texture.Find(def.portrait)
        this.mPortrait:SetTexture(this.mPortraitTexture)
    end

    this.mNextLevelXP = NextLevel(this.mLevel)

```

```
    setmetatable(this, self)
    return this
end
```

Listing 3.42: The Actor constructor. In Actor.lua.

As we build up the actor class, it's worth remembering that it's used for monsters as well as party members so it needs to be flexible.

The Actor constructor takes in a def table. We'll be using the defs stored in PartyMemberDefs.lua.

The first line in the constructor checks the def for a statsGrowth table and stores it in a local growth variable. Stat growth describes how the stats increase on level-up. Enemies in the game don't level up and don't need the statsGrowth information. If statsGrowth doesn't exist, the growth variable defaults to the empty table.

In the this table, the name and id are set from the def, followed by the combat actions. A sprite is created for the portrait field. In our game, only important actors like the party members have portraits. If an actor doesn't have a portrait then the sprite is left empty. Next a stats object is created using the def and the mStatsGrowth field is set.

The initial XP defaults to 0 and the level to 1. Our monsters don't level up but their level value may be used in the combat calculations as a shorthand to represent overall experience.

The mEquipment field in the this table describes what the actor is wearing and carrying. We haven't designed any weapon definitions yet, so for now this table remains empty.

Near the end of the constructor, we check if the def has a portrait and set the portrait sprite to use that texture. Finally we work out how much XP is required for the next level and store it.

That's the constructor for an actor done. We'll revisit the constructor to extend it as we implement more of the combat code.

Next, copy the leveling functions from Listing 3.43.

```
function Actor:ReadyToLevelUp()
    return self.mXp >= self.mNextLevelXP
end

function Actor:AddXP(xp)
    self.mXp = self.mXp + xp
    return self:ReadyToLevelUp()
end

function Actor>CreateLevelUp()
```

```

local levelup =
{
    xp = - self.mNextLevelXP,
    level = 1,
    stats = {},
}

for id, dice in pairs(self.mStatGrowth) do
    levelup.stats[id] = dice:Roll()
end

-- Additional level up code
-- e.g. if you want to apply
-- a bonus every 4 levels
-- or heal the players MP/HP

return levelup
end

function Actor:ApplyLevel(levelup)
    self.mXp = self.mXp + levelup.xp
    self.mLevel = self.mLevel + levelup.level
    self.mNextLevelXP = NextLevel(self.mLevel)

    assert(self.mXp >= 0)

    for k, v in pairs(levelup.stats) do
        self.mStats.mBase[k] = self.mStats.mBase[k] + v
    end

    -- Unlock any special abilities etc.
end

```

Listing 3.43: Adding the levelling functions to the Actor. In Actor.lua.

We've already covered these functions in the previous section, so there's no need to cover them again. Next let's add a status screen to the in-game menu.

Status Screen Preparation

Our menu currently looks like Figure 3.10. We're going to add an extra option for "Status". The status screen displays the stats, currently equipped weapons, level information, and the actions the character can take in combat.

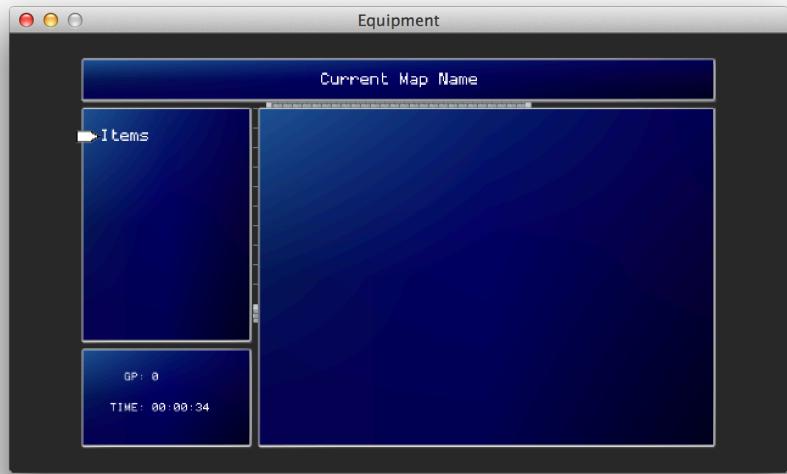


Figure 3.10: The current menu.

When the player selects the “Status” option, we let the player to pick from a list of all the party members and then display the status screen for the chosen member.

Let’s edit main.lua and force the starting party to have three members. The code is shown below, Listing 3.44.

```
-- code omitted
gIcons = Icons>Create(Texture.Find("inventory_icons.png"))

gWorld.mParty:Add(Actor>Create(gPartyMemberDefs.hero))
gWorld.mParty:Add(Actor>Create(gPartyMemberDefs.thief))
gWorld.mParty:Add(Actor>Create(gPartyMemberDefs.mage))

gStack:Push(ExploreState>Create(gStack, CreateArenaMap(),
    Vector.Create(15, 8, 1)))
-- code omitted
```

Listing 3.44: Adding party members directly. In main.lua.

Our party is now has three people: the hero, the thief, and the mage.

Menu options are defined in FrontMenuState.lua. In the FrontMenuState there’s a selection menu called mSelections. We’ll modify this by adding the “Status” option to the data table. Copy the code from Listing 3.45.

```

FrontMenuState = {}
FrontMenuState.__index = FrontMenuState
function FrontMenuState:Create(parent)

    -- code omitted

    this =
    {
        -- code omitted

        mSelections = Selection:Create
        {
            spacingY = 32,
            data =
            {
                "Items",
                "Status",
            }
        }
    }

```

Listing 3.45: Adding the status menu. In FrontMenuState.lua.

Inserting the string “Status” to the data table adds the option to the in-game menu.

The FrontMenuState should display all active party members on the right of the screen. In Figure 3.10 you’ll notice on the right there’s a large empty blue section. This is the perfect place for our character summaries! Each summary displays important information about the character: name, portrait, current level, etc.

Actor Summary UI Element

To display the party summaries on the FrontMenuState we need an ActorSummary class. You can see an actor summary in Figure 3.11.

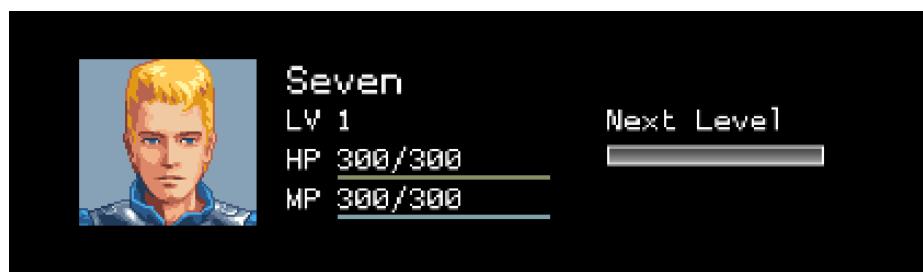


Figure 3.11: What the ActorSummary class should look like.

The summary element looks complicated, but it's made from existing interface elements. The elements include a portrait sprite, some text, and a few progress bars. All we need to do is write the layout code and hook it up to the actor data.

Create a file called ActorSummary.lua and add it to the manifest and dependencies files. Each of the three progress bars needs two textures. These are in the art directory. Add them to your manifest file too. The textures you need are: hpbackground.png, hpforeground.png, mpbackground.png, mpforeground.png, xpbackground.png, and xpforeground.png.

Copy the ActorSummary constructor from Listing 3.46.

```
ActorSummary = {}
ActorSummary.__index = ActorSummary
function ActorSummary:Create(actor, params)
    params = params or {}

    local this =
    {
        mX = 0,
        mY = 0,
        mWidth = 340, -- width of entire box
        mActor = actor,
        mHPBar = ProgressBar:Create
        {
            value = actor.mStats:Get("hp_now"),
            maximum = actor.mStats:Get("hp_max"),
            background = Texture.Find("hpbackground.png"),
            foreground = Texture.Find("hpforeground.png"),
        },
        mMPBar = ProgressBar:Create
        {
            value = actor.mStats:Get("mp_now"),
            maximum = actor.mStats:Get("mp_max"),
            background = Texture.Find("mpbackground.png"),
            foreground = Texture.Find("mpforeground.png"),
        },
        mAvatarTextPad = 14,
        mLabelRightPad = 15,
        mLabelValuePad = 8,
        mVerticalPad = 18,
        mShowXP = params.showXP
    }

    if this.mShowXP then
        this.mXPBar = ProgressBar:Create
        {

```

```

        value = actor.mXP,
        maximum = actor.mNextLevelXP,
        background = Texture.Find("xpbackground.png"),
        foreground = Texture.Find("xpforeground.png"),
    }
end

setmetatable(this, self)
this:SetPosition(this.mX, this.mY)
return this
end

```

Listing 3.46: ActorSummary constructor. In ActorSummary.lua.

The ActorSummary constructor takes in an actor object and a parameter table, params. It uses the actor object to fill out summary details. The params table contains display settings and is optional.

The first line sets the params table to the empty table if the params argument is nil. Then the this table is set up with all the fields.

The mX and mY fields define where the summary is drawn on screen. They represent the top left hand corner of the summary element. The mWidth field defines how wide the summary element is. This is used to align the internal elements. Here the width value is hard coded to 340 pixels.

Next we create progress bars for the current actor HP and MP. The progress bars use the values from the actor's stat object.

The final entries in the this table contains layout data.

- mAvatarTextPad - defines in pixels how far the text should appear to the right of the portrait
- mLabelRightPad - position info for the XP label
- mLabelValuePad - how much space should appear between the label and values
- mVerticalPad - defines the distance between each line of text

When the summary displays "LV: 1" the mLabelValuePad defines how many pixels are between the "LV:" and "1".

After setting up the this table, we check the params for an mShowXP value. If the value exists and is set to true, then we create an XP progress bar.

Then the metatable is set, and before the object is returned, the SetPosition function is called to position the element and arrange all its internal parts.

Let's define the SetPosition function. Copy the code from Listing 3.47.

```

function ActorSummary:SetPosition(x, y)
    self.mX = x
    self.mY = y

    if self.mShowXP then
        self.mXPBar:Render(renderer)

        local boxRight = self.mX + self.mWidth
        local textY = statsStartY
        local left = boxRight - self.mXPBar.mHalfWidth * 2

        renderer:AlignText("left", "top")
        renderer:DrawText2d(left, textY, "Next Level")
    end

    -- HP & MP
    local avatarW = self.mActor.mPortraitTexture:GetWidth()
    local barX = self.mX + avatarW + self.mAvatarTextPad
    barX = barX + self.mLabelRightPad + self.mLabelValuePad
    barX = barX + self.mMPBar.mHalfWidth

    self.mMPBar:SetPosition(barX, self.mY - 72)
    self.mHPBar:SetPosition(barX, self.mY - 54)
end

```

Listing 3.47: The SetPosition function for the actor summary. In ActorSummary.lua.

We can use SetPosition to position the actor summary anywhere on the screen.

The MP and HP bars are aligned to the left of the avatar, one on top of the other. The alignment numbers here are hard coded. Ideally these numbers should be in a central location so they're easy to adjust, but for our needs, hard coded will work.

The actor summaries in the in-game menu need to be selectable. Selection is done with a cursor that points to the summary in focus. To help position the cursor, let's add a function that returns the middle left of the summary. Copy the code from Listing 3.48.

```

function ActorSummary:GetCursorPosition()
    return Vector.Create(self.mX, self.mY - 40)
end

```

Listing 3.48: Function to get the position for a cursor. In ActorSummary.lua.

Now for the most important function, Render. Copy the code from Listing 3.49.

```

function ActorSummary:Render(renderer)

    local actor = self.mActor

    --
    -- Position avatar image from top left
    --
    local avatar = actor.mPortrait
    local avatarW = actor.mPortraitTexture:GetWidth()
    local avatarH = actor.mPortraitTexture:GetHeight()
    local avatarX = self.mX + avatarW / 2
    local avatarY = self.mY - avatarH / 2

    avatar:SetPosition(avatarX, avatarY)
    renderer:DrawSprite(avatar)

    --
    -- Position basic stats to the left of the
    -- avatar
    --
    renderer:AlignText("left", "top")

    local textPadY = 2
    local textX = avatarX + avatarW / 2 + self.mAvatarTextPad
    local textY = self.mY - textPadY
    renderer:ScaleText(1.6, 1.6)
    renderer:DrawText2d(textX, textY, actor.mName)

    --
    -- Draw LVL, HP and MP labels
    --
    renderer:AlignText("right", "top")
    renderer:ScaleText(1.22, 1.22)
    textX = textX + self.mLabelRightPad
    textY = textY - 20
    local statsStartY = textY
    renderer:DrawText2d(textX, textY, "LV")
    textY = textY - self.mVerticalPad
    renderer:DrawText2d(textX, textY, "HP")
    textY = textY - self.mVerticalPad
    renderer:DrawText2d(textX, textY, "MP")
    --
    -- Fill in the values
    --

```

```

local textY = statsStartY
local textX = textX + self.mLabelValuePad
renderer:AlignText("left", "top")
local level = actor.mLevel
local hp = actor.mStats:Get("hp_now")
local maxHP = actor.mStats:Get("hp_max")
local mp = actor.mStats:Get("mp_now")
local maxMP = actor.mStats:Get("mp_max")

local counter = "%d/%d"
local hp = string.format(counter,
                        hp,
                        maxHP)
local mp = string.format(counter,
                        mp,
                        maxMP)

renderer:DrawText2d(textX, textY, level)
textY = textY - self.mVerticalPad
renderer:DrawText2d(textX, textY, hp)
textY = textY - self.mVerticalPad
renderer:DrawText2d(textX, textY, mp)

-- 
-- Next Level area
--
if self.mShowXP then
    renderer:AlignText("right", "top")
    local boxRight = self.mX + self.mWidth
    local textY = statsStartY
    renderer:DrawText2d(boxRight, textY, "Next Level")
    self.mXPBar:Render(renderer)
end

-- 
-- MP & HP bars
--
self.mHPBar:Render(renderer)
self.mMPBar:Render(renderer)
end

```

Listing 3.49: Adding a Render function. In ActorSummary.lua.

This Render function in Listing 3.49 is long but simple. The code positions and renders the elements in the summary box.

First the portrait information is extracted. The width and height of the portrait are used to align it to the top left of the box.

Next the avatar name is drawn near the top and to the left of the portrait. The X coordinate uses the right side of the portrait and the Y coordinate uses a hard coded offset from the top of the box. The name text is important, so text size is set to 1.6.

Below the name we draw the level, HP and MP labels, each below the other. We aligned the text right so they appear lined up when the values are rendered. Values are drawn next to the labels, aligned left. A little padding is added to create a gap between label and value. The value text is taken from the actor and mStats object. Values are formatted to appear like “100/100” so the user can see the current and maximum values at a glance.

If the XP bar is rendered, then the “Next Level” text is also drawn. Finally the HP and MP bars are both rendered. Their positions have already been set in the SetPosition function.

Displaying the Party in the In-Game Menu

Let’s use the new ActorSummary class to add a summary for each party member to the in-game menu. Start by copying the code from Listing 3.50.

```
function FrontMenuState:CreatePartySummaries()

    local partyMembers = gWorld.mParty.mMembers

    local out = {}
    for _, v in pairs(partyMembers) do
        local summary = ActorSummary:Create(v,
            { showXP = true})
        table.insert(out, summary)
    end
    return out
end
```

Listing 3.50: Creating the party actor summaries. In FrontMenuState.lua.

The actor summaries show at a glance the condition of the party. When the player picks a menu option like “Status” or “Equipment” they’ll be prompted to choose a party member from the summary list. Therefore we’ll store the actor summaries using a Selection menu. Copy the code from Listing 3.51 into the FrontMenuState constructor.

```
this.mPartyMenu = Selection>Create
{
    spacingY = 90,
```

```

data = this>CreatePartySummaries(),
columns = 1,
rows = 3,
OnSelection = function(...) this:OnPartyMemberChosen(...) end,
RenderItem = function(menu, renderer, x, y, item)
    if item then
        item:SetPosition(x, y + 35)
        item:Render(renderer)
    end
end
}
this.mPartyMenu:HideCursor()

return this
end

```

Listing 3.51: Adding a Selection object for the ActorSummaries at the end of the constructor. In FrontMenuState.lua.

In Listing 3.51 we create a new field called `mPartyMenu`. It's a Selection menu made up of `ActorSummary` objects, one for each member of the party. The spacing is set to 90 pixels, which is around the height of each `ActorSummary`. The data source is filled using the `CreatePartySummaries` function. There is a single column with three rows, one for each member of the party. When an actor is chosen, the `OnSelection` callback calls `OnPartyMemberChosen`, a function we haven't written yet.

The `RenderItem` function positions the summary and renders it. An item is only rendered if it exists. This lets us have parties of one or two people instead of always having three. Once the selection menu is created, we hide the selection cursor because when the `FrontMenuState` is open we're not interacting with the party members.

To see this in action, modify the render function to match the code in Listing 3.52.

```

function FrontMenuState:Render(renderer)

    -- code omitted

    renderer:DrawText2d(goldX + 10, goldY - 25, gWorld:TimeAsString())

    local partyX = self.mLayout:Left("party") - 16
    local partyY = self.mLayout:Top("party") - 45
    self.mPartyMenu:SetPosition(partyX, partyY)
    self.mPartyMenu:Render(renderer)
end

```

Listing 3.52: Rendering the party selection. In FrontMenuState.lua.

In Listing 3.52 we've modified the Render function to draw the character summaries on the right. Run the code and you'll see the summaries when you open the in-game menu by pressing alt, as shown in Figure 3.12.



Figure 3.12: Displaying the current party on the in game menu.

Example combat/party-1-solution contains the code so far.

Selecting Party Members

When we select the "Status" menu option, focus needs to switch from the in-game menu to the party member menu. Then the player can select a party member and open the status screen to display their details.

Choosing an option from the in-game menu calls `OnMenuClick`. `OnMenuClick` needs to support choosing a specific party member. Copy the updated code from Listing 3.53.

```
function FrontMenuState:OnMenuClick(index)

    local ITEMS = 1

    if index == ITEMS then
        return self.mStateMachine:Change("items")
    end
```

```
    self.mInPartyMenu = true
    self.mSelections:HideCursor()
    self.mPartyMenu>ShowCursor()
    self.mPrevTopBarText = self.mTopBarText
    self.mTopBarText = "Choose a party member"
end
```

Listing 3.53: Allowing party members to be selected. In FrontMenuState.lua.

When “Items” is chosen, we don’t need to select a party member. For any other menu option, we default to choosing a party member. The `mInPartyMenu` flag is set to true. The cursor for the menu options is hidden and the party selection cursor is revealed. The top bar text is changed from what should be the map name to some helper text, asking the player to choose a party member.

The `mInPartyMenu` and `mTopBarText` are new fields that need to be added to the constructor. Copy the code from Listing 3.54.

```
this =
{
    -- code omitted
    mTopBarText = "Empty Room",
    mInPartyMenu = false,
}
```

Listing 3.54: Modifying the constructor. In FrontMenuState.lua.

Run the code now. Open the in-game menu and select “Status”. You’ll see the selection jump over to the party members as shown in the image in Figure 3.13.



Figure 3.13: The party selection menu in action.

The focus now moves to the party members, but we don't have code to handle input! Let's modify the `FrontMenuState:Update` function to handle input when selecting a character. Copy the code from Listing 3.55.

```
function FrontMenuState:Update(dt)

    if self.mInPartyMenu then
        self.mPartyMenu:HandleInput()

        if Keyboard.JustPressed(KEY_BACKSPACE) or
            Keyboard.JustPressed(KEY_ESCAPE) then
            self.mInPartyMenu = false
            self.mTopBarText = self.mPrevTopBarText
            self.mSelections>ShowCursor()
            self.mPartyMenu:HideCursor()
        end

    else
        self.mSelections:HandleInput()

        if Keyboard.JustPressed(KEY_BACKSPACE) or
            Keyboard.JustPressed(KEY_ESCAPE) then
```

```

        self.mStack:Pop()
    end

end
end

```

Listing 3.55: Adding support for party member selection to the Update function. In FrontMenuState.lua.

Run the code, open the menu, choose “Status”, and you’ll be able to navigate up and down the party member selection.

We changed the Update function to update the correct menu depending on the focus. If the `mInPartyMenu` is false, it means the in-game menu has focus and the update code works as it did previously. If `mInPartyMenu` is true, then the party member selection is updated. If backspace is pressed, focus is returned to the in-game options menu. To return focus we set `mInPartyMenu` to false, hide the cursor on the party member selection, and show the menu selection cursor.

Try to select a party member and the game crashes! This is because the `OnPartyMemberChosen` function doesn’t exist. Let’s write it now.

`OnPartyMemberChosen` identifies which party member has been picked and uses the menu selection index to decide which state to move to. Copy the code from Listing 3.56.

```

function FrontMenuState:OnPartyMemberChosen(actorIndex, actorSummary)
    -- Need to move state according to menu selection

    local indexToStateId =
    {
        [2] = "status",
        -- more states can go here.
    }

    local actor = actorSummary.mActor
    local index = self.mSelections:GetIndex()
    local stateId = indexToStateId[index]
    self.mStateMachine:Change(stateId, actor)
end

```

Listing 3.56: Add an OnSelection callback for the mPartyMenu. In FrontMenuState.lua.

The `OnPartyMember` function is written to easily support future menu options. The `indexToStateId` table takes the index of the chosen menu item and returns the name of the state to change to. Our current menu only has two items; index 1 is Items, index

2 is Status. Items doesn't use the party member selection, so it's ignored; only the status state is added.

When we change to a new menu state, we pass in the destination state id and an actor. The actor object is taken from the actorSummary. The state id is retrieved from the indexToStateId table using the options menu index.

Hooking Up the Status Screen

The InGameMenuState uses a state machine to handle all states in the menu. Let's add the status screen to this state machine. Copy the code from Listing 3.57.

```
function InGameMenuState:Create(stack)

    -- code omitted

    this.mStateMachine = StateMachine>Create
    {
        -- code omitted
        ['status'] =
        function()
            return StatusMenuState>Create(this)
        end
    }
}
```

Listing 3.57: Adding a transition to the status menu. In InGameMenuState.lua.

The code in Listing 3.57 means that whenever we change to the status we create and return a StatusMenuState. Next let's define the StatusMenuState class.

Status Menu State

The status screen draws inspiration from the character sheets of Dungeon and Dragons. It gives detailed information about the character: their stats, equipment, and other important details. Generally it's not interactive. It only displays information.

Create a new file called StatusMenuState.lua and add it to the manifest and dependencies files. Open it and add the following constructor Listing 3.58.

```
StatusMenuState = {}
StatusMenuState.__index = StatusMenuState
function StatusMenuState:Create(parent)

    local layout = Layout>Create()
```

```

layout:Contract('screen', 118, 40)
layout:SplitHorz('screen', "title", "bottom", 0.12, 2)

local this =
{
    mParent = parent,
    mStateMachine = parent.mStateMachine,
    mStack = parent.mStack,

    mLayout = layout,
    mPanels =
    {
        layout>CreatePanel("title"),
        layout>CreatePanel("bottom"),
    },
}

setmetatable(this, self)
return this
end

```

Listing 3.58: StatusMenuState Constructor. In StatusMenuState.lua.

The constructor starts by creating panels. The panels here are very simple. We create a panel the size of the screen, shrink it down a little by calling contract, then split it horizontally into a title bar and main body panel.

After the layout, we set up the this table. We add an mParent field which stores the InGameMenuState object. We also store references to the state machine and state stack. Remember, the state machine is controlling the menu screens, and stack holds the game with the menu as the top state. We add the layout data using the mLayout field and all the panels are stored in mPanels. That's all the constructor needs to do.

Next let's add the Enter and Exit functions. Copy the code from Listing 3.59.

```

function StatusMenuState:Enter(actor)
    self.mActor = actor
    self.mActorSummary = ActorSummary>Create(actor, { showXP = true})

    self.mEquipMenu = Selection>Create
    {
        data = self.mActor.mActiveEquipSlots,
        columns = 1,
        rows = #self.mActor.mActiveEquipSlots,
        spacingY = 26,
    }

```

```

        RenderItem = function(...) self.mActor:RenderEquipment(...) end
    }
self.mEquipMenu:HideCursor()

self.mActions = Selection>Create
{
    data = self.mActor.mActions,
    columns = 1,
    rows = #self.mActor.mActions,
    spacingY = 18,
    RenderItem = function(self, renderer, x, y, item)
        local label = Actor.ActionLabels[item]
        Selection.RenderItem(self, renderer, x, y, label)
    end
}
self.mActions:HideCursor()

end

function StatusMenuState:Exit()
end

```

Listing 3.59: The Enter and Exit functions. In StatusMenuState.lua.

The status screen Enter function takes in the actor that the screen is going to describe. The actor is stored in `mActor` and we create and store a summary in `mActorSummary`.

The status screen displays information about the selected character. A lot of this information is lists of labels and values. To make them easy to display, we'll use a Selection menu. These selections aren't interactive. They're only used for layout.

First we set up the character equipment slots. The equipment slots the character has available are stored in `mActiveEquipmentSlots` on the actor. We haven't added this field yet but we will shortly. We're using a selection to display this information. The column number is set to 1 so we only display a single column of equipped items. The number of rows is determined by how many active equipment slots there are. Spacing between the equipment slot labels is hard coded to 26 pixels. The render function is set to `RenderEquipment` which we'll also need to implement. After creating the equipment menu, we hide the cursor so it doesn't look like something you can interact with.

Next we set up the character's combat actions. All party members have some actions, like `attack` and `item`, but some actions are unique, such as `steal` for the thief or `magic` for the mage. The Actor class already has an `mActions` table that we'll use to populate the selection menu. This menu doesn't overload the `RenderItem` callback because it's just simple text. As with the previous menu, the cursor is hidden.

Now that we've created our menus, we extend the Actor class to get them working.

Extending the Actor Class

Let's begin by adding some useful strings to the Actor table. These strings are used by all actors, so it's better to have them in a central place rather than scattered throughout the code. Copy the code from Listing 3.60.

```
Actor =
{
    EquipSlotLabels =
    {
        "Weapon",
        "Armor",
        "Accessory",
        "Accessory"
    },
    EquipSlotId =
    {
        "weapon",
        "armor",
        "acces1",
        "acces2"
    },
    ActorStats =
    {
        "strength",
        "speed",
        "intelligence"
    },
    ItemStats =
    {
        "attack",
        "defense",
        "magic",
        "resist"
    },
    ActorStatLabels =
    {
        "Strength",
        "Speed",
        "Intelligence"
    },
    ItemStatLabels =
    {
        "Attack",
        "Defense",
        "Magic",
        "Resist"
    }
}
```

```

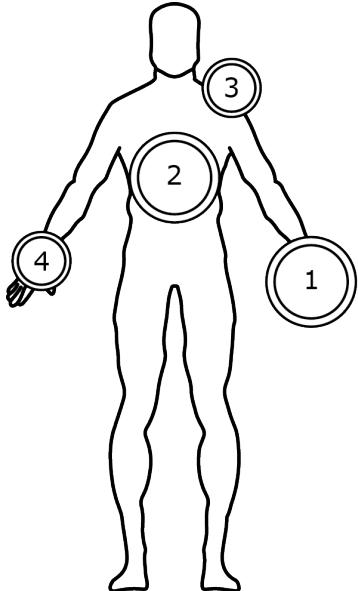
    "Resist"
},
ActionLabels =
{
    ["attack"] = "Attack",
    ["item"] = "Item",
},
}

```

Listing 3.60: Adding description constants to the Actor. In Actor.lua.

We've added seven tables of strings. The strings are used for adding labels to the status screen and other menus. Adding them directly to the Actor table means they aren't duplicated for each Actor instance.

Tables with names ending in "Labels" have their entries printed directly to the screen.



index	id	name
1	weapon	Weapon
2	armor	Armor
3	access1	Accessory
4	access2	Accessory

Figure 3.14: How the equipment slots variables relate to each other. Accessory locations are just for illustration.

The EquipSlotLabels table contains English descriptions for the equipment slots. The EquipSlotId table maps a number to an equipment slot name. The first slot is weapon, the second armor, and so on. These tables help us render out an actor's equipment. You can see the relationship between the various fields in Figure 3.14.

The ItemStats and ActorStats tables list the stats to display for each item and actor. The game may have more stats, but these are the stats we show to the player.

Let's add the mActiveEquipSlots table to the Actor class. Copy the code from Listing 3.61.

```
Actor.__index = Actor
function Actor:Create()
    local this =
    {
        -- code omitted
        mActiveEquipSlots = def.equipslots or { 1, 2, 3 },
    }
    -- code omitted
end
```

Listing 3.61: Adding mActiveEquipSlots to the Actor. In Actor.lua.

Listing 3.61 adds a table mActiveEquipSlots to the actor. It's filled from the def or defaults to the first three slots. This means all characters begin without a second accessory slot. This slot might be unlocked later in the game.

Next add the RenderEquipment function from Listing 3.62.

```
function Actor:RenderEquipment(menu, renderer, x, y, index)

    x = x + 100
    local label = self.EquipSlotLabels[index]
    renderer:AlignText("right", "center")
    renderer:DrawText2d(x, y, label)

    local slotId = self.EquipSlotId[index]
    local text = "none"
    if self.mEquipment[slotId] then
        local itemId = self.mEquipment[slotId]
        local item = ItemDB[itemId]
        text = item.name
    end

    renderer:AlignText("left", "center")
    renderer:DrawText2d(x + 10, y, text)
end
```

Listing 3.62: Adding a function to render out equipment. In Actor.lua.

The RenderEquipment function is used by the status screen (and later the equipment screen) to render out an actor's equipment slots. We use the index parameter to get the equipment slot label and id. The slotId lets us know if there's anything in the slot. If the slot is empty, the value text is set to "none". If a slot is non-null, it contains an item id number. The itemId lets us get the item from the item database and draw its name.

Let's return to the status menu code.

Adding StatusMenuState Render and Update Functions

The Update function for the status screen is pretty simple, so let's add that first. Copy the code from Listing 3.63.

```
function StatusMenuState:Update(dt)
    if Keyboard.JustReleased(KEY_BACKSPACE) or
        Keyboard.JustReleased(KEY_ESCAPE) then
        self.mStateMachine:Change("frontmenu")
    end
end
```

Listing 3.63: The Update function. In StatusMenuScreen.

The code in Listing 3.63 changes the state back to the front menu when backspace or escape is pressed.

Next we need a DrawStat helper function. This gets called from the status screen's Render function. Copy the code from Listing 3.64.

```
function StatusMenuState:DrawStat(renderer, x, y, label, value)
    renderer:AlignText("right", "center")
    renderer:DrawText2d(x -5, y, label)
    renderer:AlignText("left", "center")
    renderer:DrawText2d(x + 5, y, tostring(value))
end
```

Listing 3.64: The DrawStat function. In StatusMenuState.lua.

The code in Listing 3.64 draws the label and value at the passed in position. The label is aligned right and the value is on the left. This nicely aligns the labels and means the value text can be any width without overlapping the label.

Next copy the Render function from Listing 3.65.

```

function StatusMenuState:Render(renderer)

    for k,v in ipairs(self.mPanels) do
        v:Render(renderer)
    end

    local titleX = self.mLayout:MidX("title")
    local titleY = self.mLayout:MidY("title")
    renderer:ScaleText(1.5, 1.5)
    renderer:AlignText("center", "center")
    renderer:DrawText2d(titleX, titleY, "Status")

    local left = self.mLayout:Left("bottom") + 10
    local top = self.mLayout:Top("bottom") - 10
    self.mActorSummary:SetPosition(left, top)
    self.mActorSummary:Render(renderer)

    renderer:AlignText("left", "top")
    local xpStr = string.format("XP: %d/%d",
                                self.mActor.mXP,
                                self.mActor.mNextLevelXP)
    renderer:DrawText2d(left + 240, top - 58, xpStr)

    self.mEquipMenu:SetPosition(-10, -64)
    self.mEquipMenu:Render(renderer)

    local stats = self.mActor.mStats

    local x = left + 106
    local y = 0
    for k, v in ipairs(Actor.ActorStats) do
        local label = Actor.ActorStatLabels[k]
        self:DrawStat(renderer, x, y, label, stats:Get(v))
        y = y - 16
    end
    y = y - 16
    for k, v in ipairs(Actor.ItemStats) do
        local label = Actor.ItemStatLabels[k]
        self:DrawStat(renderer, x, y, label, stats:Get(v))
        y = y - 16
    end
end

```

Listing 3.65: The first part of StatusMenuState.Render function. In StatusMenuState.lua.

The render function starts by rendering all the backing panels. It then renders the title, "Status", in the middle of the title panel at 1.5 scale.

The character summary is rendered at the top left corner of the main panel. The XP details are rendered to the left of the character summary. Next the mEquipMenu is rendered to display the actor's equipment. This is followed by the stats for the character. The stats are made up from the stats associated with the character, then the stats associated with the items.

Next copy the final part of this function from Listing 3.66.

```
renderer:AlignText("left", "top")
local x = 75
local y = 25
local w = 100
local h = 56
-- this should be a panel - duh!
local box = Textbox>Create
{
    text = {"Status", "XP: 100", "Level: 10", "Health: 100", "Gold: 100", "Food: 100", "Experience: 100", "Health Regen: 100", "Food Regen: 100", "Gold Regen: 100", "Equipment: 100", "Character Stats: 100", "Item Stats: 100"}, 
    textScale = 1.5,
    size =
    {
        left = x,
        right = x + w,
        top = y,
        bottom = y - h,
    },
    textbounds =
    {
        left = 10,
        right = -10,
        top = -10,
        bottom = 10
    },
    panelArgs =
    {
        texture = Texture.Find("gradient_panel.png"),
        size = 3,
    },
}
box.mAppearTween = Tween>Create(1,1,0)
box:Render(renderer)
self.mActions:SetPosition(x - 14, y - 10)
self.mActions:Render(renderer)
end
```

Listing 3.66: The first part of StatusMenuState.Render function. In StatusMenuState.lua.

The code in Listing 3.66 displays the combat actions. A textbox is drawn around the action list to segment it off and its tween is altered to stop the in-transition.

Run the code so far and you'll see something like Figure 3.15. The full code for the project thus far is available in combat/party-2-solution.



Figure 3.15: The status screens.

Great! We can now view the party and use the status screen to check out party member details. Next we'll expand our project so the hero can recruit characters.

Joining the Party

A party of one isn't much fun, so let's add a few potential party members to the game world.

Example combat/party-3 contains the code so far. Run this project and you'll see there's only a single party member. The starting party is set up as shown in Listing 3.67.

```
-- code omitted  
gIcons = Icons>Create(Texture.Find("inventory_icons.png"))
```

```

gWorld.mParty:Add(Actor:Create(gPartyMemberDefs.hero))

gStack:Push(ExploreState:Create(gStack, CreateArenaMap(),
    Vector.Create(30, 18, 1)))
-- code omitted

```

Listing 3.67: A party of one. In main.lua.

Our party can have two more members, and we're going to recruit those members from the map. Before we can recruit a character, we need to add their definition tables. Copy the code from Listing 3.68.

```

gEntities =
{
    hero =
    {
        texture = "walk_cycle.png",
        width = 16,
        height = 24,
        startFrame = 9,
        tileX = 11,
        tileY = 3,
        layer = 1
    },
    thief =
    {
        texture = "walk_cycle.png",
        width = 16,
        height = 24,
        startFrame = 41,
        tileX = 11,
        tileY = 3,
        layer = 1
    },
    mage =
    {
        texture = "walk_cycle.png",
        width = 16,
        height = 24,
        startFrame = 25,
        tileX = 11,
        tileY = 3,
        layer = 1
    },
}

```

Listing 3.68: Adding the new party members as entities.

Each party member shares the same texture but with a different starting frame.

The definitions in Listing 3.68 let us add the prospective party members to the map. Next we'll add entries into the gCharacters table with new references to the leveling and experience data stored in the gPartyMemberDefs table.

Start by copying the hero definition from Listing 3.69.

```
gCharacters =  
{  
    hero =  
    {  
        ... previous character def code omitted.  
  
        actorId = "hero",  
        partyDef = gPartyMemberDefs.hero  
    },
```

Listing 3.69: The character definition for the hero. In EntityDefs.lua.

The actorId lets us know which party member this character represents. Next we'll add a def for the thief Listing 3.70 in the same table.

```
thief =  
{  
    actorId = "thief",  
    entity = "thief",  
    anims =  
    {  
        up = {33, 34, 35, 36},  
        right = {37, 38, 39, 40},  
        down = {41, 42, 43, 44},  
        left = {45, 46, 47, 48},  
    },  
    facing = "down",  
    controller = { "npc_stand", "move" },  
    state = "npc_stand",  
},
```

Listing 3.70: The thief, Jude, character definition. In EntityDefs.lua.

The thief def is very similar to that of hero, just with different animation frames. Finally add the mage from Listing 3.71.

```

mage =
{
    actorId = "mage",
    entity = "mage",
    anims =
    {
        up = {17, 18, 19, 20},
        right = {21, 22, 23, 24},
        down = {25, 26, 27, 28},
        left = {29, 30, 31, 32},
    },
    facing = "down",
    controller = { "npc_stand", "move" },
    state = "npc_stand",
}

```

Listing 3.71: The mage, Ermis, character definition.

With the characters defined, we can add them to the map.

There's a Tiled map in your project called arena.tmx. Open it and find a good position to place Ermis, the mage, and Jude, the thief. I'm going to choose either side of the door. Edit the lua map file to match Listing 3.72 but feel free to insert your own map positions for the characters.

```

function CreateArenaMap()
    return
{
    -- code omitted

    on_wake =
    {
        {
            id = "AddNPC",
            params = {{ def = "mage", id = "mage", x = 21, y = 14 }}
        },
        {
            id = "AddNPC",
            params = {{ def = "thief", id = "thief", x = 35, y = 14 }}
        },
    },
}

```

Listing 3.72: Adding the Jude and Ermis to the map. In map_arena.lua.

Load up the game and you'll see both characters in the world as shown in Figure 3.16. They don't do anything at the moment, but we can fix that by adding some trigger code.



Figure 3.16: The party members on the map, in the game world.

To interact with the characters on the map, we're going to need two Map helper functions, GetNPC and RemoveNPC. Open Map.lua and add the following Listing 3.73.

```
function Map:GetNPC(x, y, layer)
    layer = layer or 1

    for _, npc in ipairs(self.mNPCs) do
        if npc.mEntity.mTileX == x
            and npc.mEntity.mTileY == y
            and npc.mEntity.mLayer == layer then
            return npc
        end
    end

    return nil
end

function Map:RemoveNPC(x, y, layer)
    layer = layer or 1

    for i = #self.mNPCs, 1, -1 do
        local npc = self.mNPCs[i]
```

```

        if npc.mEntity.mTileX == x
            and npc.mEntity.mTileY == y
            and npc.mEntity.mLayer == layer then
                table.remove(self.mNPCs, i)
                self:RemoveEntity(npc.mEntity)
                self.mNPCById[npc.mId] = nil
                return true
            end
        end

        return false
    end

```

Listing 3.73: Adding NPC helper functions. In Map.lua.

The GetNPC function takes a position on the map, and if there's an NPC there, it returns the Character object of that NPC. Otherwise it returns nil. To do this, it iterates through all NPCs on the map and returns the first with a matching location. The layer position is optional and defaults to one.

The RemoveNPC function works in a very similar way to GetNPC. Given a position, it removes the NPC at that location and returns true, or it returns false if no such NPC can be found. It iterates through the mNPCs table, and if it finds an NPC it removes it from the table and removes the associated entity from the mEntity table.

These functions will help us create a recruit function.

The GetNPC function returns a character object. Currently there's no information in the character object to tell us exactly which character we're talking to! Therefore we'll modify the Character class to store a reference to its def. Copy the changes from Listing 3.74.

```

function Character:Create(def, map)

    -- code omitted

    local this =
    {
        mDef = def, -- add a reference to def

```

Listing 3.74: Recording the character definition. In Character.lua.

We'll use this def to make sure we recruit the correct person. The def can now be used to store all types of metadata in the future.

Let's return to the map and add the recruit triggers. Copy the code from Listing 3.75.

```

actions =
{
    talk_recruit =
    {
        id = "RunScript",
        params = { RecruitNPC }
    },
},
trigger_types =
{
    recruit = { OnUse = "talk_recruit" },
},
triggers =
{
    { trigger = "recruit", x = 10, y = 5 },
    { trigger = "recruit", x = 20, y = 5 },
},

```

Listing 3.75: Adding triggers for the NPCs to the map. In map_arena.lua.

In Listing 3.75 we've added a new action, talk_recruit, to the action table. This action runs a script called RecruitNPC, which we'll define shortly. We create an OnUse trigger type and add it to the list of triggers. The triggers are positioned where our mage and thief are standing.

Copy the RecruitNPC function code from Listing 3.76.

```

function CreateArenaMap()

    local RecruitNPC =
    function(map, trigger, entity, x, y, layer)

        local npc = map:GetNPC(x, y, layer)
        local actorId = npc.mDef.actorId
        local actorDef = gPartyMemberDefs[actorId]
        local name = actorDef.name

        local OnRecruit
        local dialogParams =
        {
            textScale = 1.2,
            choices =
            {
                options =
                {

```

```

        "Recruit",
        "Leave"
    },
    OnSelection = function(index)
        if index == 1 then
            OnRecruit()
        end
    end
},
}

gStack:PushFit(gRenderer,
0, 0,
string.format("Recruit %s?", name),
300,
dialogParams)

OnRecruit = function()
    gWorld.mParty:Add(Actor>Create(actorDef))
    map:RemoveNPC(x, y, layer)
end

```

Listing 3.76: Adding a function that adds an NPC to the player party. In map_arena.lua.

When the player triggers the recruit action, a dialog box appears asking if they want to recruit this NPC, as shown in Figure 3.17.



Figure 3.17: A dialog box is used to confirm you want to recruit this NPC.

The RecruitNPC function removes an NPC from the map and adds it to the player party. We start by storing some useful variables: the npc that we find at the trigger position, the actorId and actorDef of the npc, and the NPC's name. The actorDef is used to add the character to the party. Then OnRecruit callback is predeclared before the dialog box but implemented near the end of the RecruitNPC function.

The dialogParams table defines the dialog box's appearance and behavior. Textscale is set to 1.2 and the player is given a choice of two options:

1. "Recruit"
2. "Leave"

When the player makes a choice, the OnSelection function is called. If the player chooses the first option, "Recruit", the OnRecruit function is called. If the player chooses the second option, the dialog box closes and nothing else happens.

Once the dialogParams table is set up, it's used to create the dialog box. The dialog box is then pushed onto the stack with the text "Recruit [npc_name]?"

In the last part of the RecruitNPC function, we define OnRecruit. This is where the recruiting code actually lives. It's quite simple. It uses the actorDef to create an actor and adds it to the party. Then it removes the NPC from the map.

Run the code and try recruiting the thief or mage into your party. You should see something like Figure 3.18. Check out the status menu and you'll be able to see the new recruit's details.



Figure 3.18: Adding people to the party.

The recruiting process is quite cool, but we can make it cooler! Currently when recruited the thief and mage suddenly disappear from the map. It would be less jarring if they faded out. The storyboard code we made earlier is perfect for this task, but to use it we need some actions.

Let's define RemoveNPC and AddPartyMember as actions in Actions.lua. Copy the code from Listing 3.77.

```
Actions =
{
    -- code omitted

    RemoveNPC = function(map, id)
        return function (trigger, entity, tX, tY, tLayer)
            local npc = map.mNPCbyId[id].mEntity
            assert(npc)
            map:RemoveNPC(npc.mX, npc.mY, npc.mLayer)
        end
    end,
}
```

```

AddPartyMember = function(actorId)
    return function (trigger, entity, tX, tY, tLayer)
        local actorDef = gPartyMemberDefs[actorId]
        assert(actorDef)
        gWorld.mParty:Add(Actor:Create(actorDef))
    end
end,

```

Listing 3.77: Adding RemoveNPC and AddPartyMember as actions. In Actions.lua.

The RemoveNPC action wraps up our existing remove NPC code in an action. The AddPartyMember action uses an actorId to add a new member to the party. We can run these actions in a storyboard.

Let's use a storyboard to give the OnRecruit function a little more polish. Copy the code from Listing 3.78.

```

OnRecruit = function()

local fadeout =
{
    SOP.FadeOutChar("handin", npc.mId),
    SOP.RunAction("RemoveNPC",
        { "handin", npc.mId },
        { GetMapRef }),
    SOP.RunAction("AddPartyMember", { npc.mDef.actorId } ),
    SOP.HandOff("handin")
}
local storyboard = Storyboard:Create(gStack, fadeout, true)
gStack:Push(storyboard)
end

```

Listing 3.78: Updating the recruit function to add a fade. In map_arena.lua.

In Listing 3.78 you can see we're now controlling the recruit process using a storyboard.

You may recall that when a storyboard is created, it has its own internal stack. If the storyboard constructor has true passed in as the third parameter, it takes the map from the global stack and pulls it onto its own internal stack. The pulled-in map is given the id "handin". We used the "handin" mapid when telling the character to fadeout. The FadeOurChar operation takes in the name for the current map and the id for the npc we're recruiting. It then fades the NPC out over a few frames.

The second operation in the fadeout table runs the RemoveNPC action to remove the NPC from the map. The third operation runs the AddPartyMember action using the

actor id stored in the NPC's character definition. Finally the HandOff operation tells the storyboard to push the map back onto the main stack.

Then we create the storyboard object and push it on top of the stack. The full code can be found in example combat/party-3-solution. Run the project and check it out.

Closing

We've covered a lot in this chapter; we've defined a party, created the status screen, and created a small demo that allows NPCs to be recruited from the map.

The next step is writing code to let us arm our new recruits and add the ability to pick up items from the game world.

Equipment

One Ring to rule them all,
One Ring to find them,
One Ring to bring them all,
And in the darkness bind them.

- The Lord of the Rings, J.R.R. Tolkien

There's something magical about discovering hidden treasure in the bowels of a forgotten dungeon, won from a great magician or pulled from an ancient chest. All JRPGs have treasures scattered through the world: Final Fantasy's Masamune, Phoenix Down⁴, Zelda's Bombs, Master Sword and Boomerang. The same is true of writing: the Vorpal Blade from Lewis Carroll's Jabberwocky, the Ring from Lord of the Rings. Storied items are fun. Finding treasure is fun.

Defining Equipment

Game worlds are littered with all types of equipment; in ancient crypts, guarded by monsters or available from merchants for the right price. Before equipment can appear in the world, it first needs to be defined in code.

Example combat/equipment-1 contains the code we've developed so far. We'll fill out the ItemDB.lua file, from the Dungeon game, by adding some weapons, armor and potions.

An item may have the following fields:

⁴Did you know *down* actually refers to the downy feathers of the phoenix? This didn't come across very well in translation!

- **name** - what to call the item
- **type** - the item type, e.g. a weapon
- **icon** - the icon to represent the item
- **restriction** - which characters can use the item
- **description** - flavor text
- **stats** - a modifier table
- **use** - what happens when it's used
- **use_restriction** - describes when it can be used

The item type can be one of the following:

- **weapon** - an item to equip in the weapon slot
- **armor** - an item to equip in the armor slot
- **accessory** - an item to equip in the accessory slot
- **key** - a key item, usually important for quests
- **useable** - an item that can be used, usually during combat

These are basic types of item we support in the game. Changing the type determines where the item appears in the inventory and if it can be equipped or not.

The icon field is optional. If no icon is set, the icon is determined by the item type. This icon is an index into the icon table. For weapons, we use a different icon for each of the character weapons. The thief uses daggers, the wizard staves, and the hero swords. Each of these weapon types will use a different icon. The same is true of armor: plate, leather and robes each have a different icon.

The restriction table describes who can use the item. Restrictions work on the actorId from the actor def. Listing 3.79 shows an example of restricting a weapon to the thief and hero.

```
restriction = { "hero", "thief" }
```

Listing 3.79: How to use restrictions.

If the restriction table is empty, it's assumed everyone can use it. In a more developed game, this would be a nice place to add extra restrictions such as gender or requiring some in-game event to have occurred.

The description field is a string describing what the item does. We show the description on the equip screen. The stats field is a stats modifier table.

The use field is a table that links a useable object to an action. In Listing 3.80 we show an example use field for a heal potion.

```

use =
{
    action = "small_heal",
    target = "any",
    target_default = "friendly_wounded",
    hint = "Choose target to heal.",
}

```

Listing 3.80: The use table.

In the use table, the action field links to an action def. The target field describes which characters can be selected on the battlefield. The target_default field specifies which characters are selected by default. For instance, think of a health potion. When you use it, it should target your most wounded party member. If it is a resurrect potion, then the default target would be whoever is dead. The hint field is the text that appears when the player chooses the item's target.

The use_restriction field describes *when* an item can be used. Some items may only be used in combat, some only on the world map. If use_restriction is nil, the item can be used anywhere. Here are some examples in Listing 3.81

```

-- Berserk potion
use_restriction = { "combat" }

-- Heal potion
use_restriction = { "combat", "world_map" } -- or just set to nil

-- Tent
use_restriction = { "world_map" }

```

Listing 3.81: Examples of the use restriction field.

To begin adding new items to the game, let's first modify the Icon class.

Equipment Icons

Things have become more complicated since the Dungeon minigame. We have multiple characters now: the hero, the thief and the mage. The mage, thief and hero each use different weapons and armor. The thief uses light clothing and daggers, the mage uses robes and staves, and the hero uses heavy armor and swords.

To represent the new item types, we need a host of new icons. Copy the code from Listing 3.82.

```

Icons = {}
Icons.__index = Icons
function Icons:Create(texture)
    local this =
    {

        -- code omitted

        mIconDefs =
        {
            useable = 1,
            accessory = 2,
            weapon = 3,
            sword = 4,
            dagger = 5,
            stave = 6,
            armor = 7,
            plate = 8,
            leather = 9,
            robe = 10,
            uparrow = 11,
            downarrow = 12
        }
    }

```

Listing 3.82: Adding more icons descriptions. In Icons.lua.

The inventory_icons.png texture in the art folder has been updated with these new icons.

In the previous examples, we used a global gIcon variable. Let's move this into the world object. Copy the code from Listing 3.83.

```

World = {}
World.__index = World
function World:Create()
    local this =
    {
        -- code omitted
        mParty = Party:Create(),
        mIcons = Icons:Create(Texture.Find("inventory_icons.png")),
    }

```

Listing 3.83: Adding the icons to the world object. In World.lua.

Now that we've moved the icon object, we need to update the all code that references it! Luckily only one function touches it, Word:DrawItem. Copy the code from Listing 3.84.

We'll also modify this function so item icons fall back to their type icon if a custom icon isn't defined.

```
function World:DrawItem(menu, renderer, x, y, item)
    if item then
        local itemDef = ItemDB[item.id]
        local iconSprite = self.mIcons:Get(itemDef.icon or itemDef.type)

    -- code omitted
```

Listing 3.84: Extending the draw item function.

That's all the icon changes we need to make. Let's add some equipment!

Designing New Equipment

Let's open the item defs and add some new equipment. We're going to overwrite all the existing content in this file, beginning with the empty item. Copy the code from Listing 3.85.

```
ItemDB =
{
    [-1] =
    {
        name = "",
        type = "",
        icon = nil,
        restriction = nil,
        description = "",
        stats =
        {
        },
        use = nil,
        use_restriction = nil,
    },
    -- We'll add all the new items here
}

EmptyItem = ItemDB[-1]

-- Give all items an id based on their position
-- in the list.
for id, def in pairs(ItemDB) do
```

```
    def.id = id
end
```

Listing 3.85: Starting with a new empty item database. In ItemDB.lua.

There's nothing in the database apart from the empty item. The empty item contains all the fields an item may contain but sets them to nil.

At the end of the database we globally declare the EmptyItem variable. There's also a loop that gives each item def an id based on its position in the ItemDB table.

Let's add the hero equipment next. Copy the items into the ItemDB table from Listing 3.86.

```
{
    name = "Bone Blade",
    type = "weapon",
    icon = "sword",
    restriction = {"hero"},
    description = "A wicked sword made from bone.",
    stats = { add = { attack = 5 } }
},
{
    name = "Bone Armor",
    type = "armor",
    icon = "plate",
    restriction = {"hero"},
    description = "Armor made from plates of blackened bone.",
    stats =
    {
        add =
        {
            defense = 5,
            resist = 1,
        }
    }
},
{
    name = "Ring of Titan",
    type = "accessory",
    description = "Grants the strength of the Titan.",
    stats = { add = { strength = 10 } }
},
```

Listing 3.86: Items for the hero character. In ItemsDB.lua.

In Listing 3.86 the hero has three pieces of equipment; a sword, armor, and a ring. The weapons and armor are restricted to the hero; no one else can equip them. The sword has an attack of 5. The armor has a defense of 5 and a resist of 1. The ring can be used by anyone and gives a 10 strength bonus to the wielder.

The armor and weapon have their icons set to plate and sword respectively. This helps identify them as being restricted to the hero character. The ring doesn't have an icon set, so it defaults to its type, the accessory icon.

Let's add the mage equipment next. Copy the code from Listing 3.87.

```
{
    name = "World Tree Branch",
    type = "weapon",
    icon = "stave",
    restriction = {"mage"},
    description = "A hard wood branch.",
    stats =
    {
        add =
        {
            attack = 2,
            magic = 5
        }
    }
},
{
    name = "Dragon's Cloak",
    type = "armor",
    icon = "robe",
    restriction = {"mage"},
    description = "A cloak of dragon scales.",
    stats =
    {
        add =
        {
            defense = 3,
            resist = 10,
        }
    }
},
{
    name = "Singer's Stone",
    type = "accessory",
    description = "The stone's song resists magical attacks.",
    stats = { add = { resist = 10 } }
}
```

Listing 3.87: Items for the mage character. In ItemDB.lua.

The stave and armor are restricted to the mage. The defense and attack values are lower but there are additional magic and resist stats. The Singer's Stone accessory adds an additional 10 magical resist to the wielder.

Next up are a set of items befitting our thief.

```
{  
    name = "Black Dagger",  
    type = "weapon",  
    icon = "dagger",  
    restriction = {"thief"},  
    description = "A dagger made out of an unknown material.",  
    stats = { add = { attack = 4 } }  
},  
{  
    name = "Footpad Leathers",  
    type = "armor",  
    icon = "leather",  
    restriction = {"thief"},  
    description = "Light armor for silent movement.",  
    stats = { add = { defense = 3 } },  
},  
{  
    name = "Swift Boots",  
    type = "accessory",  
    description = "Increases speed by 25%",  
    stats = { mult = { speed = 0.25 } },  
},
```

Listing 3.88: Items for the thief character. In ItemsDB.lua.

The thief equipment is slightly less powerful. The accessory increases speed by 25%, which is pretty powerful.

Next let's add items to restore health. Copy the code from Listing 3.89.

```
{  
    name = "Heal Potion",  
    type = "useable",  
    description = "Heal a small amount of HP.",  
    use =  
    {  
        action = "small_heal",  
        target = "any",
```

```

        target_default = "friendly_wounded",
        hint = "Choose target to heal.",
    }
},
{
    name = "Life Salve",
    type = "useable",
    description = "Restore a character from the brink of death.",
    use =
    {
        action = "revive",
        target = "any",
        target_default = "friendly_dead",
        hint = "Choose target to revive.",
    }
},

```

Listing 3.89: Adding some usable items to the item defs. In ItemDB.lua.

In Listing 3.89 the Heal Potion restores HP and the Life Salve revives a knocked out character. The actions don't exist currently but we'll be adding them as we delve deeper into combat.

That's all the equipment we need. Example combat/equipment-1-solution contains the complete code. Next we'll add functions to equip and unequip weapons, armor and accessories.

Equip and Unequip functions

Now that we have weapons, armor and accessories, we need a way to use them! Example combat/equipment-2 contains the code so far.

Let's add equip functions to the Actor class. Copy the code from Listing 3.90.

```

function Actor:Equip(slot, item)

    -- 1. Remove any item current in the slot
    --     Put that item back in the inventory
    local prevItem = self.mEquipment[slot]
    self.mEquipment[slot] = nil
    if prevItem then
        -- Remove modifier
        self.mStats:RemoveModifier(slot)
        gWorld:AddItem(prevItem)
    end

```

```

-- 2. If there's a replacement item move it to the slot
if not item then
    return
end
assert(item.count > 0) -- This should never be allowed to happen!
gWorld:RemoveItem(item.id)
self.mEquipment[slot] = item.id
local modifier = ItemDB[item.id].stats or {}
self.mStats:AddModifier(slot, modifier)
end

function Actor:Unequip(slot)
    self:Equip(slot, nil)
end

```

Listing 3.90: Adding equip and unequip functions to the actor. In Actor.lua.

The Equip function takes in a slot and an item def. The slot describes where to equip the item; is it a weapon, piece of armor or accessory? The item def is taken from the ItemDB.lua file. To empty a slot we pass in a nil item def.

The equip function checks the target slot. If an item already occupies the slot, we remove it and add it to the inventory. Stat modifiers are removed when any item is unequipped. If the item removed was a sword with +5 attack, once it's removed that +5 attack is also removed.

If the item parameter is nil, there's nothing else to do. The slot is empty and the function returns.

If the item argument isn't nil, we remove the item from the inventory and place it into the equipment slot. Any modifiers attached to the item are applied to the Actor stats.

The unequip function removes an item from a slot. The code calls Equip with nil as the item argument.

It's a good idea to test code as it's written, so let's use the Equip function now. Update your main.lua file to match Listing 3.91.

```

-- Add to inventory, then equip
_, gHero = next(gWorld.mParty.mMembers)
gBoneBlade = ItemDB[1]
gWorld:AddItem(gBoneBlade.id)
gHero:Equip("weapon", gWorld.mItems[1])

function update()
    local dt = GetDeltaTime()
    -- code omitted
end

```

Listing 3.91: Bruteforce equipping a weapon on the hero. In main.lua.

The code in Listing 3.91 adds a sword to the inventory and equips it into the hero's weapon slot.

Run the code. Open the status screen and you'll see that the hero is holding the weapon. See Figure 3.19. Also note that the attack stat is now 5. Without the weapon, it was 0.



Figure 3.19: The status screen showing the hero with weapon.

Equipping weapons is working. In an RPG, it's useful to compare two pieces of equipment to see which is better. Let's write the item comparison code next.

Comparing Equipment

In order to compare stats, for example for a sword, we need the player's current stats and the player's stats if his weapon was swapped for an alternate one.

We'll add a function to the actor that, given an item and a slot id, returns a table describing how the actor stats change if they were to equip it. Once we have this stat difference table, we can report back to the player which stats increase, decrease or stay the same. When we make the equip screen, we'll use the compare function to display how character stats change for a new piece of equipment.

Let's begin by writing a helper function to return a list of all actor stat ids. We'll call this function `CreateStatNameList`. Copy the code from Listing 3.92.

```

function Actor>CreateStatNameList()

    local list = {}

    for _, v in ipairs(Actor.ActorStats) do
        table.insert(list, v)
    end

    for _, v in ipairs(Actor.ItemStats) do
        table.insert(list, v)
    end

    table.insert(list, "hp_max")
    table.insert(list, "mp_max")

    return list
end

```

Listing 3.92: CreateStatNameList gets the id's of all the stats a character might have. It's in Actor.lua.

The list of stats contains the intrinsic stats of the character plus those added by equipment. We also add the hp_max and mp_max stats. We'll use this list when comparing how two items change a character's stats.

Next let's add a PredictStats function. It answers the question "What would happen if I swapped this weapon for that weapon?" The function takes in an equipment slot and an item. It returns a table storing the stat difference if the player were to equip the item.

Copy the code from Listing 3.93.

```

function Actor>PredictStats(slot, item)

    local statsId = self>CreateStatNameList()

    -- Get values for all the current stats
    local current = {}
    for _, v in ipairs(statsId) do
        current[v] = self.mStats:Get(v)
    end

    -- Replace item
    local prevItemId = self.mEquipment[slot]
    self.mStats:RemoveModifier(slot)

```

```

self.mStats:AddModifier(slot, item.stats)

-- Get values for modified stats
local modified = {}
for _, v in ipairs(statsId) do
    modified[v] = self.mStats:Get(v)
end

-- Get difference
local diff = {}
for _, v in ipairs(statsId) do
    diff[v] = modified[v] - current[v]
end

-- Undo replace item
self.mStats:RemoveModifier(slot)
if prevItemId then
    self.mStats:AddModifier(slot, ItemDB[prevItemId].stats)
end

return diff
end

```

Listing 3.93: Adding a predict stats function. In PredictStats.lua.

PredictStats contains the meat of the comparison code.

Let's go through how the code works. First, a list of all the stat ids we care about is generated. This list contains strength, intelligence, speed, attack, resist, etc. Then we get the current Actor stats and store the value of each stat in a table called current. This table represents the full player stats with all modifiers. In our case, the hero has the bone blade equipped and his stats will be something like Listing 3.94.

```

current =
{
    strength = 10,
    speed = 10,
    intelligence = 10,
    hp_max = 300,
    mp_max = 300,
    attack = 5,
    defense = 0,
    magic = 0,
    resist = 0
}

```

Listing 3.94: An hypothetical table of the hero's stats with modifiers applied.

Next we build a similar table if the user was wielding the alternate item passed into the PredictStats function. To work out these stats, we remove the modifiers applied by the item currently in the slot. Then we get the actor stats and apply the modifiers from the alternate item. We store the stats in a table called modified. Let's say we're swapping in the wizard's "World Tree Branch" instead of the "Bone Blade". Then the modified table might look like Listing 3.95.

```
modified =
{
    strength = 10,
    speed = 10,
    intelligence = 10,
    hp_max = 300,
    mp_max = 300,
    attack = 2,
    defense = 0,
    magic = 5,
    resist = 0
}
```

Listing 3.95: A table representing the hero's stats if a piece of equipment was changed.

Compare how the stats have changed from Listing 3.94 to Listing 3.95. You can refer to the ItemDB to see the stats of items being equipped.

The function returns a table of stat differences. To create the difference table, we subtract the modified values from the current ones. In the case of our sword/staff example, the results would look like Listing 3.96.

```
diff =
{
    attack = -3,
    magic = 5,
    -- everything else is 0
}
```

Listing 3.96: A table showing the difference in stats if a weapon was equipped.

The difference table tells us that wielding the staff instead the sword gives +5 magic but -3 attack. This is the type of information we'll display on the equip screen.

Before returning the difference table, we remove the modifier applied by the test item and restore the modifiers of the original item.

Let's modify the main.lua to test out PredictStats. Copy the code from Listing 3.97.

```

_, gHero = next(gWorld.mParty.mMembers)
gBoneBlade = ItemDB[1]
gWorldStaff = ItemDB[4]

gWorld:AddItem(gBoneBlade.id)
gHero:Equip("weapon", gWorld.mItems[1])

local diff = gHero:PredictStats("weapon", gWorldStaff)
print("Diffs")
for k, v in pairs(diff) do
    print(k, v)
end

```

Listing 3.97: Testing out the PredictStats function. In main.lua.

In the test shown in Listing 3.97 we print out how the hero's stats would change if we swapped his sword for a staff.

Example combat/equipment-2-solution contains the code so far. Run it and you'll see the content of Listing 3.98 printed to the console.

```

Diffs
strength    0
attack     -3
speed      0
magic      5
resist     0
hp_max     0
mp_max     0
intelligence  0
defense    0

```

Listing 3.98: The output from PredictStats.

From Listing 3.98 we can see that PredictStats is working as expected. We only tried out a weapon, but it works with armor and accessories too. Try altering the example to work with different items and slots. Next let's create the equip menu.

Equipment Menu

The equipment menu displays the equipped items for a single character. The player can select any equipment slot and remove or swap an item for any other valid item in the inventory. The final screen layout is shown in Figure 3.20.



Figure 3.20: The layout of the equipment screen.

The top of the screen shows the character's current equipment. The bottom shows the inventory. The focus starts on the first equipment slot. If the player selects an equipment slot, the lower half of the screen displays all inventory items that can be put into the slot.

Slots have restrictions. The weapon slot only accepts weapon items and only those weapons the character can use. The equipment menu automatically filters the inventory to show only items that can be equipped.

Like the status menu, selecting Equip from the in-game menu prompts the player to choose one of the party members. The equip menu then displays the data for the chosen party member.

Example combat/equipment-3 contains all the code we've written so far, or you can continue using your own codebase.

Hooking Up the Equipment Menu

Create EquipMenuState.lua and add it to the dependencies and manifest files. This will just be an empty state for now, as shown in Listing 3.99.

```

EquipMenuState = {}
EquipMenuState.__index = EquipMenuState
function EquipMenuState:Create()
    local this = {}

    setmetatable(this, self)
    return this
end

function EquipMenuState:Enter()
end

function EquipMenuState:Exit()
end

function EquipMenuState:Update(dt)
end

function EquipMenuState:Render(renderer)
end

```

Listing 3.99: A skeleton state for the EquipMenuState class. In EquipMenuState.lua.

Next let's link the equipment menu into the in-game menu flow. Copy Listing 3.100 into your code to add it to the in-game state menu.

```

function InGameMenuState:Create(stack)

    -- code omitted

    this.mStateMachine = StateMachine>Create
    {
        ["frontmenu"] =
        function()
            return FrontMenuState>Create(this)
        end,
        ["items"] =
        function()
            return ItemMenuState>Create(this)
        end,
        ["equip"] =
        function()
            return EquipMenuState>Create(this)
        end,
    }

```

```
-- code omitted
```

Listing 3.100: Adding the EquipMenuState into the game menu state machine. In InGameMenuState.lua.

In Listing 3.100 the equipment menu is stored using the “equip” id string. This means we can create the state using the “equip” id and we’ll be able to load it from the in-game menu.

Let’s add “Equip” to the options menu in the in-game menu screen. Copy the code from Listing 3.101.

```
function FrontMenuState:Create(parent)

    -- code omitted

    local this
    this =
    {

        -- code omitted

        mSelections = Selection:Create
        {
            spacingY = 32,
            data =
            {
                "Items",
                "Status",
                "Equipment"
            }
        }
    }
}
```

Listing 3.101: Adding the equipment option to the FrontMenuState. In FrontMenuState.lua.

The changes in Listing 3.101 add the “Equipment” option to the in-game menu, but we need a little code to link it to the EquipMenuState. Selecting an option prompts you to choose a party member. After you choose a party member, the equip menu is opened. Change the code in OnPartyMemberChosen to match Listing 3.102 which is called when the party member is chosen.

```
function FrontMenuState:OnPartyMemberChosen(actorIndex, actorSummary)

    local indexToStateId =
    {
```

```

[2] = "status",
[3] = "equip",
-- more states can go here.
}

```

Listing 3.102: Linking the equip selection option to the Equipment state. In FrontMenuState.lua.

The “Equipment” option is the third option in the menu. This means OnPartyMemberChosen is called with an index of 3 (the equip index). We change the state using the “equip” state id that we get from the indexToStateId table.

Run this code. Open the in-game menu and you’ll see a new “Equipment” option, as shown in Figure 3.21. Select it and choose a party member, and you’ll enter the EquipMenuState. The game then seems to block as the EquipMenuState doesn’t contain any code yet!



Figure 3.21: A new front menu option for the Equipment.

Let’s fill in the missing code for the EquipMenuState.

The Equipment Menu State Constructor

Here’s how we want the equipment menu to work. When the player opens the equip menu, they’ll be able to browse the actor’s equipment slots. As they move from slot to

slot, the inventory menu changes to show which items can be equipped for the current slot. Selecting a slot moves focus to the inventory menu. While in the inventory menu, the stats panel shows stat differences for the currently focused item.

The equipment menu, like the other menus, starts by creating backing panels.

Let's define the constructor. Copy the code from Listing 3.103.

```
EquipMenuState = {}  
EquipMenuState.__index = EquipMenuState  
function EquipMenuState:Create(parent)  
  
    local this =  
    {  
        mParent = parent,  
        mStack = parent.mStack,  
        mStateMachine = parent.mStateMachine,  
        mScrollbar = Scrollbar:Create(Texture.Find('scrollbar.png'), 135),  
        mBetterSprite = gWorld.mIcons:Get('uparrow'),  
        mWorseSprite = gWorld.mIcons:Get('downarrow'),  
        mFilterMenus = {},  
        mInList = false  
    }  
  
    this.mBetterSprite:SetColor(Vector.Create(0,1,0,1))  
    this.mWorseSprite:SetColor(Vector.Create(1,0,0,1))  
  
    -- Create panel layout  
    local layout = Layout:Create()  
    layout:Contract('screen', 118, 40)  
    layout:SplitHorz('screen', 'top', 'bottom', 0.12, 2)  
    layout:SplitVert('top', 'title', 'category', 0.75, 2)  
    local titlePanel = layout.mPanels['title']  
  
    layout = Layout:Create()  
    layout:Contract('screen', 118, 40)  
    layout:SplitHorz('screen', 'top', 'bottom', 0.42, 2)  
    layout:SplitHorz('bottom', 'desc', 'bottom', 0.2, 2)  
    layout:SplitVert('bottom', 'stats', 'list', 0.6, 2)  
    layout.mPanels['title'] = titlePanel  
  
    self.mPanels =  
    {  
        layout:CreatePanel('top'),  
        layout:CreatePanel('desc'),  
        layout:CreatePanel('stats'),  
        layout:CreatePanel('list'),
```

```

        layout>CreatePanel('title'),
    }
self.mLayout = layout

setmetatable(this, self)
return this
end

```

Listing 3.103: The Equipment Menu constructor. In EquipMenuState.lua.

The EquipMenuState takes a single parameter, parent. The parent is the InGameMenuState that we can use to exit the equip menu.

In the this table, we store useful variables from the parent object. We create a scrollbar with a height of 135 pixels. This scrollbar tracks progress through the inventory menu.

Next we create two icon sprites called mBetterSprite and mWorseSprite. These sprites are taken from the World's Icon object. Open the icon texture. You'll see two white arrow icons. The up and down arrows are used to show if a new piece of equipment has better or worse stats than whatever is currently equipped. After the this table, we set the mBetterSprite color to green and the mWorseSprite color to red.

We create a table called mFilterMenus to store a selection menu for each equipment slot. One menu is for all the weapons, one is for the armor, and one is for the accessories. We switch between the mFilterMenus menus to show the items that can go in each slot.

The EquipMenuState displays two selection menus at a time, the equipment slots and the inventory menu. The mInList flag tells us which selection menu has focus. If mInList is true, the inventory menu is in focus. If it is false, the equipment slot menu is focused. The mInList starts set to false.

The remaining part of the constructor sets up the backing panels.

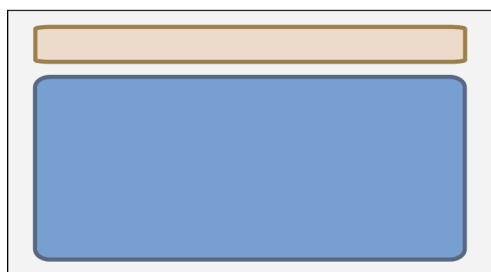
We start with a full screen panel, shrink it a little, then split it horizontally, into a title bar called top and the rest of the screen called bottom. We split the title bar vertically and use the left part for the menu title. The right section we won't render. The title panel is going to be drawn on top of all other backing panels. You can see the steps in Figure 3.22.



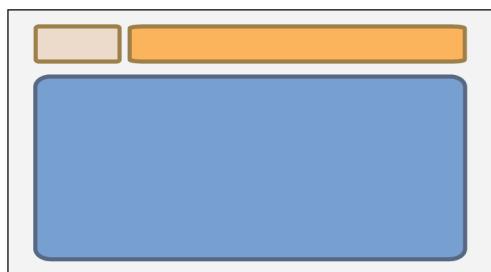
Begin with a
full screen
rectangle



Contract 10%



Split horizontal
at 12%



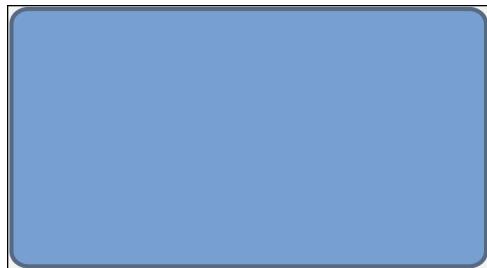
Split vertical
at 75%



Throw away
pieces apart
from the
top left

Figure 3.22: Creating the title panel for the equipment menu.

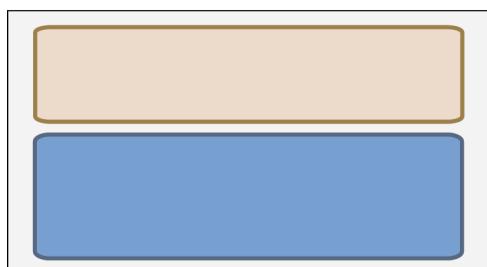
For the second set of panels we start, once again, with a full screen panel. We split the panel horizontally into top and bottom parts. We'll use the bottom to show the inventory and the top to display the character's current equipment. We split the bottom section horizontally again. The upper, slimmer, panel holds the current item description. The lower section is split vertically into stats and inventory panels. The stat panel displays the current Actor stats. The inventory panel displays the inventory. You can see the steps in Figure 3.23.



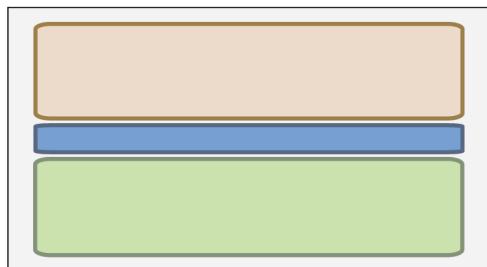
Begin with a
full screen
rectangle



Contract 10%



Split horizontal
at 42%



Split horizontal
at 20%



Split vertically
at 60%

Figure 3.23: Creating the remaining panels for the equipment menu.

After creating the panels we store them in `mPanels`. We also create and store a layout in `mLayout`.

Let's do a quick render test. Copy the code from Listing 3.104.

```
function EquipMenuState:Render(renderer)
    for _, v in ipairs(self.mPanels) do
        v:Render(renderer)
    end
end
```

Listing 3.104: Rendering the panels for the EquipMenuState. In EquipMenuState.lua.

Run the game, enter the in-game menu, select Equipment, and choose a character. As before, this opens the `EquipMenuState` but this time you'll see the backing panels for the menu as shown in Figure 3.24.

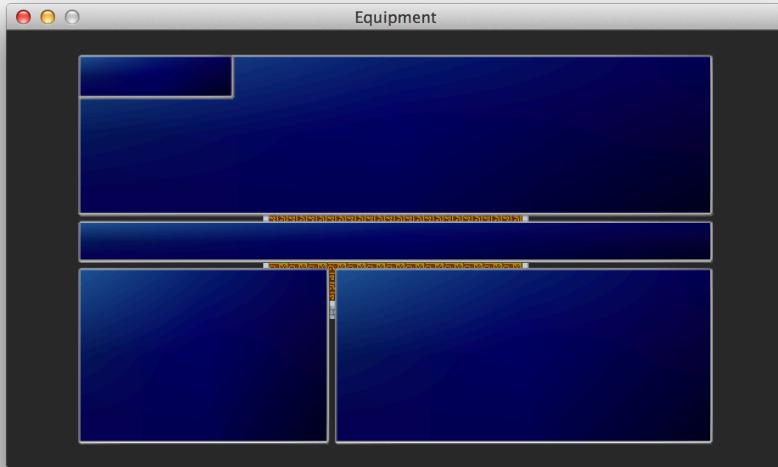


Figure 3.24: Rendering out the panel for the EquipMenuState.

The Equipment Menu State Enter and Exit Functions

The backing panels have been laid out. Let's start filling them with data! A good place to start is the Enter function, where the EquipMenuState is told which Actor to display. Copy the code from Listing 3.105.

```
function EquipMenuState:Enter(actor)
    self.mActor = actor
    self.mActorSummary = ActorSummary>Create(actor)
    self.mEquipment = actor.mEquipment

    self:RefreshFilteredMenus()

    self.mMenuItemIndex = 1
    self.mFilterMenus[self.mMenuItemIndex]:HideCursor()

    self.mSlotMenu = Selection>Create
    {
        data = self.mActor.mActiveEquipSlots,
        OnSelection = function(...) self:OnSelectMenu(...) end,
        columns = 1,
        rows = #self.mActor.mActiveEquipSlots,
        spacingY = 26,
        RenderItem = function(...) self.mActor:RenderEquipment(...) end
    }
end
```

Listing 3.105: The Enter function. In EquipMenuState.lua.

The Enter function is short and uses a few function we haven't defined yet. It takes in one parameter, actor. This is the party member whose equipment we'll display. The actor object is stored as mActor and a summary object is created and stored in mActorSummary. In the equipment menu, the actor summary is drawn near the top of the screen to show whose equipment we're viewing.

The party member's current equipment is stored in mEquipment. The mEquipment table is a list of equipment. The table keys are slot ids and the values are item ids from the ItemDB table.

The inventory panel shows the inventory filtered for the selected equipment slot. When we browse the actor's equipment, only items from the inventory that can replace the equipped item are shown. This can be seen in Figure 3.25.

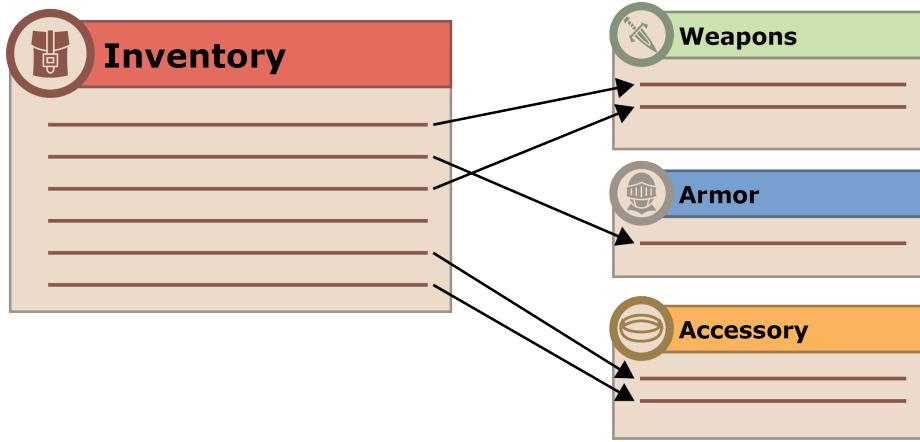


Figure 3.25: Using different selection menu to show the inventory filtered.

We call RefreshFilteredMenus to handle creating the filtered menus. We haven't written this function yet. Each equipment slot has its own inventory menu. We set mMenuIndex to 1, meaning we only show the first of these menus, the weapon menu. The cursor for the filtered menu is hidden as it doesn't have focus.

We list the equipment slot details to the right of the actor summary. The slots are represented by mSlotMenu. The mSlotMenu uses mActiveEquipSlots to gather the equipment data from the actor, the same as in the StatusMenuState.

When the user selects a slot, we call OnSelectMenu which we also haven't written yet. To render slot data we call the actor's RenderEquipment. We've chosen 26 pixels for the spacing between the elements, and we're only going to have a single column.

Before implementing OnSelectMenu and RefreshFilteredMenus we'll define a few helper functions. Copy the code from Listing 3.106.

```
function Actor:Create(def)

    -- code omitted

    local this =
    {
        -- code omitted
        mSlotTypes =
        {
            "weapon",
            "armor",
            "accessory",
            "accessory"
        }
    }

```

```
    }
}
```

Listing 3.106: Adding slot type information to the Actor. In Actor.lua.

Listing 3.106 adds a new `mSlotTypes` table to the actor. This table tells us which equipment slots an actor has. These strings are same as the type of information used by the item defs in the ItemDB.

Not all actors can use every item. We need an extra filter after the type to see if the actor can use the item. To do this filtering, let's add a `CanUse` function to the actor. Copy the code from Listing 3.107.

```
function Actor:CanUse(item)

    if item.restriction == nil then
        return true
    end

    for _, v in pairs(item.restriction) do
        if v == self.mId then
            return true
        end
    end

    return false
end
```

Listing 3.107: Test if an actor can use an item. In Actor.lua.

In Listing 3.107 `CanUse` takes in an item def. If the def has no restriction, we return true. If there's no restriction, it's assumed anyone can use the item. Otherwise we loop through the restriction table and test whether it contains the actor's id. If it does, we return true. If not, we return false.

The RefreshFilteredMenus function

Now let's implement the `RefreshFilteredMenus` function. Copy the code from Listing 3.108.

```
function EquipMenuState:RefreshFilteredMenus()

    -- Get a list of filters by slot type
```

```

-- Items will be sorted into these lists
local filterList = {}
local slotCount = #self.mActor.mActiveEquipSlots

for i = 1, slotCount do
    local slotType = self.mActor.mSlotTypes[i]
    filterList[i] = { type = slotType, list = {} }
end

-- Actually sort inventory items into lists.
for k, v in ipairs(gWorld.mItems) do

    local item = ItemDB[v.id]

    for i, filter in ipairs(filterList) do

        if item.type == filter.type
            and self.mActor:CanUse(item) then
            table.insert(filter.list, v)
        end

    end
end

self.mFilterMenus = {}
for k, v in ipairs(filterList) do
    local menu = Selection>Create
    {
        data = v.list,
        columns = 1,
        spacingX = 256,
        displayRows = 5,
        spacingY = 26,
        rows = 20,
        RenderItem = function(self, renderer, x, y, item)
            gWorld:DrawItem(self, renderer, x, y, item)
        end,
        OnSelection = function(...) self:OnDoEquip(...) end
    }
    table.insert(self.mFilterMenus, menu)
end
end

```

Listing 3.108: The RefreshFilterMenus function that controls the inventory display. In

EquipMenuState.lua.

In RefreshFilteredMenus we first create a list for each of the actor's active equipment slots. The lists are filled with inventory items filtered by slot type and whether the actor can use the item. Then we convert the lists into selection menus. The selection menus are displayed as the user browses the equipment slots.

The code begins by creating a list called filterList. We initialize the filter list by looping through mActiveEquipSlots. For each entry we store the type information and an empty table called list. After the loop finishes, the filter table looks a little like Listing 3.109.

```
filter =
{
    { type = ["weapon"], list = {} },
    { type = ["armor"], list = {} },
    { type = ["accessory"], list = {} }
}
```

Listing 3.109: What an item filter table might look like for inventory slots.

In the second half of the function, we go through the filter table and fill in the list field. We begin by iterating through the gWorld.mItems which is our inventory. Each entry in mItems is a table with an id and a count. The id is used to look up the item def in the ItemDB. The def contains information we can use for filtering. We filter the items by checking the type and if the character can use it. If the item passes the filter, we add it to the list of the current filter table.

Next we take the lists of filtered items and create selection menus. Each selection menu takes its data from an entry in the filter table . The selection menus are created in the same order as the slots. The first slot is the player's weapon, so the first selection menu is for the weapon. The item render function is set to gWorld:DrawItem. Selecting an item calls DoEquip. We'll come back to DoEquip but it will swap the current contents of the player's slot with whatever is selected in the inventory.

The OnSelectMenu function

The OnSelectMenu function is called when an equipment slot is selected. The function switches the focus from the equipment slot to the filtered inventory at the bottom of the screen. Copy the code from Listing 3.110.

```
function EquipMenuState:OnSelectMenu(index, item)
    self.mInList = true
    self.mSlotMenu:HideCursor()
    self.mMenuItemIndex = self.mSlotMenu:GetIndex()
```

```

    self.mFilterMenus[self.mMenuItemIndex]:ShowCursor()
end
```

Listing 3.110: OnSelectMenu callback function. In EquipMenuState.lua.

The OnSelectMenu removes focus from the slots by changing mInList to true and hiding the menu cursor. It sets mMenuItemIndex to the index of the currently selected slot. The mMenuItemIndex controls which inventory menus gets rendered. Then we show the cursor in the selected inventory menu.

We've written quite a lot of code now, and it would be nice to test it. We can do this by writing part of the Render function. Copy the code from Listing 3.111.

```

function EquipMenuState:Render(renderer)
    for _, v in ipairs(self.mPanels) do
        v:Render(renderer)
    end

    -- Title
    renderer:ScaleText(1.5, 1.5)
    renderer:AlignText("center", "center")
    local titleX = self.mLayout:MidX("title")
    local titleY = self.mLayout:MidY("title")
    renderer:DrawText2d(titleX, titleY, "Equip")

    -- Char summary
    local titleHeight = self.mLayout.mPanels["title"].height
    local avatarX = self.mLayout:Left("top")
    local avatarY = self.mLayout:Top("top")
    avatarX = avatarX + 10
    avatarY = avatarY - titleHeight - 10
    self.mActorSummary:SetPosition(avatarX, avatarY)
    self.mActorSummary:Render(renderer)

    -- Slots selection
    local equipX = self.mLayout:MidX("top") - 20
    local equipY = self.mLayout:Top("top") - titleHeight - 10
    self.mSlotMenu:SetPosition(equipX, equipY)
    renderer:ScaleText(1.25, 1.25)
    self.mSlotMenu:Render(renderer)

    -- Inventory list
    local listX = self.mLayout:Left("list") + 6
    local listY = self.mLayout:Top("list") - 20
    local menu = self.mFilterMenus[self.mMenuItemIndex]
```

```

menu:SetPosition(listX, listY)
menu:Render(renderer)

-- Scroll bar
local scrollX = self.mLayout:Right("list") - 14
local scrollY = self.mLayout:MidY("list")
self.mScrollbar:SetPosition(scrollX, scrollY)
self.mScrollbar:Render(renderer)

-- ... there's more to come
end

```

Listing 3.111: Implementing more of the Render function. To display what we've written so far. In EquipMenuState.lua.

In Listing 3.111 we begin by drawing the backing panels. Then we start to fill in some of the panels. To draw the title, we take the title panel and write "Equip" in the center. Next we write out the character summary just beneath the title. The character summary has a little padding to the left.

We draw all the equipment slots to the left of the character summary. The slots are drawn around the center of the "top" panel. The description panel is next, but we'll skip it for now.

We draw the filtered inventory menu in the bottom right panel. The `mFilterMenus` table contains a list of all the inventory menus. We use `mMenuItemIndex` to choose which menu to render. Finally in the same panel the scrollbar is rendered, which reports where we are in the filtered inventory. Run the code now. Enter the `EquipMenuState` and you'll see something like Figure 3.26.



Figure 3.26: Starting to see the progress in the EquipMenuState.

Equipment Menu Utility Functions

The EquipMenuState is progressing well, but there's more to do! Let's dive back in by adding a helper function called `GetSelectedSlot`. `GetSelectedSlot` returns the currently selected slot id. If the player has the first slot selected, it returns 'weapon'. Copy the code from Listing 3.112.

```
function EquipMenuState:GetSelectedSlot()
    local i = self.mSlotMenu:GetIndex()
    return Actor.EquipSlotId[i]
end
```

Listing 3.112: Helper function to get the name of the currently selected slot. In `EquipMenuState.lua`.

Once we know which slot we have selected, we want to know what item is currently in there. Let's write another function `GetSelectedItem` to help answer this question. `GetSelectedItem` returns the item id of the item the player's cursor is currently targeting. It works for the slots menu and the inventory menus. Copy the code from Listing 3.113.

```

function EquipMenuState:GetSelectedItem()
    if self.mInList then
        local menu = self.mFilterMenus[self.mMenuItem]
        local item = menu:SelectedItem() or {id = nil}
        return item.id
    else
        local slot = self:GetSelectedSlot()
        return self.mActor.mEquipment[slot]
    end
end

```

*Listing 3.113: GetSelectedItem returns whichever item is currently selected on screen.
In EquipMenuState.lua.*

If the inventory list has focus, we return the selected item from the filtered inventory. If the equipment menu is in focus, we use GetSelectedSlot to get the slot name and then use the name to get the current item from the actor.

Let's add two more helper functions before getting back into the main code. Copy the code from Listing 3.114.

```

function EquipMenuState:FocusSlotMenu()
    self.mInList = false
    self.mSlotMenu:ShowCursor()
    self.mMenuItem = self.mSlotMenu:GetIndex()
    self.mFilterMenus[self.mMenuItem]:HideCursor()
end

function EquipMenuState:OnEquipMenuChanged()
    self.mMenuItem = self.mSlotMenu:GetIndex()

    -- Equip menu only changes, when list isn't in focus
    self.mFilterMenus[self.mMenuItem]:HideCursor()
end

```

*Listing 3.114: Helper functions for browsing the equipment menu. In
EquipMenuState.lua.*

FocusSlotMenu moves focus from the inventory to the actor's equipment slots. We set mInList to false, show the cursor on the slot menu, and hide it on the currently selected filter menu.

Each time the player selects a different equipment slot, OnEquipMenuChanged is called. OnEquipMenuChanged sets the mMenuItem to the slot index, and this changes which inventory menu is shown. It also hides the cursor for the selected inventory.

The DoEquip Function

With these helper functions implemented, we're ready to write the DoEquip function. DoEquip is called when the player selects an item in the inventory. It's used to remove an item from the inventory and equip it in the current slot. Each of the inventory menus uses this callback. Copy the code from Listing 3.115.

```
function EquipMenuState:OnDoEquip(index, item)
    -- Let the character handle this.
    self.mActor:Equip(self:GetSelectedSlot(), item)
    self:RefreshFilteredMenus()
    self:FocusSlotMenu()
end
```

Listing 3.115: Implement the OnDoEquip function. In EquipMenuState.lua.

OnDoEquip tells the actor to equip the passed-in item. If nil is passed in, the current weapon is removed. It calls the Equip function on the actor. This Equip call may change the inventory, so we call RefreshFilteredMenus to recreate the filter menus. We end the function by restoring focus to the equipment slots.

The Equipment Menu Update Function

Let's start to pull all the equipment menu code together by implementing the Update loop. Copy the code from Listing 3.116.

```
function EquipMenuState:Update(dt)

    local menu = self.mFilterMenus[self.mMenuItemIndex]

    if self.mInList then
        menu:HandleInput()
        if Keyboard.JustReleased(KEY_BACKSPACE) or
            Keyboard.JustReleased(KEY_ESCAPE) then
            self:FocusSlotMenu()
        end
    else
        local prevEquipIndex = self.mSlotMenu:GetIndex()
        self.mSlotMenu:HandleInput()
        if prevEquipIndex ~= self.mSlotMenu:GetIndex() then
            self:OnEquipMenuChanged()
        end
        if Keyboard.JustReleased(KEY_BACKSPACE) or
            Keyboard.JustReleased(KEY_ESCAPE) then
```

```

        self.mStateMachine:Change("frontmenu")
    end
end

local scrolled = menu:PercentageScrolled()
self.mScrollbar:SetNormalValue(scrolled)
end

```

Listing 3.116: The update function for the equipment menu. In EquipMenuState.lua.

The Update function handles interaction with the equip screen. First we get the currently selected filter menu and store it in the variable menu. Then we check if the equipment slots or filter menu has focus. If the filter menu has focus, we update it. If backspace or escape is pressed, we call FocusSlotMenu to return focus to the equipment slots.

If the equipment slots have focus, we get the selected index and store it in prevEquipIndex. Then we update mSlotMenu and check if the selected index has changed. If the index has changed, this means we've selected a new equipment slot and we call OnEquipMenuChanged. This function updates the active filter menu. If backspace or escape is pressed, we change the state back to the front menu screen.

At the end of the Update function we set the scrollbar to match the current progress through the inventory menu.

The EquipMenuState is now pretty functional. Let's add some items to the inventory and test it out. Add the following code in Listing 3.117 to your main.lua above the update loop.

```

for _, v in ipairs(ItemDB) do
    gWorld:AddItem(v.id)
end

```

Listing 3.117: Adding all the items into the inventory. In main.lua.

Listing 3.117 fills the inventory with all the possible items in the game.

Run the code and test out the equip screen! Recruit the other characters and you'll be able to set their equipment too. Note that the filters work correctly. Remember that when you're equipping these items, all the modifiers are being applied to the character; it's a *fully* functioning inventory management system. This is pretty big milestone in making a JRPG, so take a moment to enjoy it!

The Equipment Menu Comparison Functions

The inventory and equipment code is working well, but it's not as friendly as it could be. Currently there's no good way to compare an item in the inventory to the one currently

equipped. That's the final piece of the puzzle we need to implement before we move on from this screen.

On the left of the screen, we're showing the actor's stats. When the player browses the inventory, we want to display another column of stats. The second stat column will show how the stats change if the player equips the selected item. We'll color the stat red if it's worse, or green if it's better. We'll also use the gWorld.mIcons table to draw an up or down arrow.

Drawing comparison stats requires a special function that we're going to imaginatively call DrawStat. Copy the code from Listing 3.118.

```
function EquipMenuState:DrawStat(renderer, x, y, label, stat, diff)
    renderer:AlignText("right", "center")
    renderer:DrawText2d(x, y, label)
    renderer:AlignText("left", "center")

    local current = self.mActor.mStats:Get(stat)
    local changed = current + diff

    renderer:DrawText2d(x + 15, y, string.format("%d", current))

    if diff > 0 then
        renderer:DrawText2d(x + 60, y, string.format("%d", changed),
                           Vector.Create(0,1,0,1))
        self.mBetterSprite:SetPosition(x + 80, y)
        renderer:DrawSprite(self.mBetterSprite)
    elseif diff < 0 then
        renderer:DrawText2d(x + 60, y, string.format("%d", changed),
                           Vector.Create(1,0,0,1))
        self.mWorseSprite:SetPosition(x + 80, y)
        renderer:DrawSprite(self.mWorseSprite)
    end

end
```

Listing 3.118: The draw stat function. In EquipMenuState.lua.

The DrawStat function, in Listing 3.118, draws a player's current stats and the modified ones. It takes in six parameters:

- renderer - used to draw the stats
- x and y - the top left corner position to draw the stats
- label - the name of the stat to draw
- stat - current value of the stat
- diff - how the stat value changes if the current item is equipped

In the function body we store the current stat as `current`, and use a `changed` variable to represent the stat value if the new item was equipped. Then we draw the current stat. If `diff` equals 0, it means there's no difference, so we draw nothing. If `diff` is greater than 0, we draw the changed value next to the current stat in green. We also draw the `mBetterSprite`, which is the up arrow. If `diff` is less than 0, we draw the changed value in red and render `mWorseSprite`, a down arrow.

Let's finish off the end of the `Render` function and make use of the new `DrawStat` function. Copy the code from Listing 3.119.

```
function EquipMenuState:Render(renderer)

    -- code omitted

    local slot = self:GetSelectedSlot()
    local itemId = self:GetSelectedItem() or -1

    local diffs = self.mActor:PredictStats(slot, ItemDB[itemId])
    local x = self.mLayout:MidX("stats") - 10
    local y = self.mLayout:Top("stats") - 14
    renderer:ScaleText(1,1)

    local statList = self.mActor>CreateStatNameList()
    local statLabels = self.mActor>CreateStatLabelList()

    for k, v in ipairs(statList) do
        self:DrawStat(renderer, x, y, statLabels[k], v, diffs[v])
        y = y - 14
    end

    -- Description panel
    local descX = self.mLayout:Left("desc") + 10
    local descY = self.mLayout:MidY("desc")
    renderer:ScaleText(1, 1)
    local item = ItemDB[itemId]
    renderer:DrawText2d(descX, descY, item.description)
end
```

Listing 3.119: Adding item comparisons to the Render function. In `EquipMenuState.lua`.

The `EquipMenuState.Render` function now draws the actor stats.

We get the selected slot name and the id of the selected inventory item. Then we use the `PredictStats` function to get the diff information between the item in the slot and the one selected. We use `CreateStatNameList` to get the list of stat ids. To get the

stat labels, we use a new function, `CreateStatLabelList`. We draw the stats using the `DrawStat` function.

Let's add the `CreateStatLabelList` to the `Actor` class now. Copy the code from Listing 3.120.

```
function Actor:createStatLabelList()
    local list = {}

    for _, v in ipairs(Actor.ActorStatLabels) do
        table.insert(list, v)
    end

    for _, v in ipairs(Actor.ItemStatLabels) do
        table.insert(list, v)
    end

    table.insert(list, "HP:")
    table.insert(list, "MP:")

    return list
end
```

Listing 3.120: Getting a list of stat labels from the Actor. In Actor.lua.

`CreateStatLabelList` returns a list of stat labels to print to the screen.

The final part of the `Render` function prints the description from the `ItemDB` into the description box.

The full code for this section is available in `equipment-3-solution`. Run the code and you'll see something like Figure 3.27.



Figure 3.27: The finished equipscreen with comparisons.

This is a good equipment screen with comparisons and nice UI flow. The next step is to add some items to the game world for the player to find.

Finding Equipment

One of the great pleasures in RPGs is discovering hidden treasure. We're now at the point where we can fill out the world with chests and artifacts for our players to discover. All the code is available in example combat/equipment-4. It's same as the combat/equipment-3-solution example but the player inventory has been emptied.

On the arena map we'll add three chests full of equipment for each of our three characters.

In the art folder there's a file called chests.png. It's an image of an opened and closed treasure chest. We'll use this to create a chest entity. The player will be able to approach and *use* chests on the map.

An Empty Chest

Let's start by making a simple empty chest. Define a chest entity in EntityDefs.lua as shown in Listing 3.121.

```

gEntities =
{
    -- code omitted

    chest =
    {
        texture = "chest.png",
        width = 16,
        height = 16,
        startFrame = 1,
        openFrame = 2
    }
}

```

Listing 3.121: Adding a chest entity to the defs. In EntityDefs.lua.

Every chest has two states, opened and closed. The closed frame is the startFrame. The open frame is the openFrame.

We'll add chests to the world using an action, just like we did for the NPCs. The action creates a chest entity, places it in the world, adds an interact trigger, and associates it with a loot table.

The AddChest action needs a way to add triggers to the map. Copy the AddFullTrigger function from Listing 3.122 into your Map class.

```

function Map:AddFullTrigger(trigger, x, y, layer)

    layer = layer or 1

    -- Create trigger layer if it doesn't exist.
    if not self.mTriggers[layer] then
        self.mTriggers[mLayer] = {}
    end

    local targetLayer = self.mTriggers[layer]
    targetLayer[self:CoordToIndex(x, y)] = trigger

end

```

Listing 3.122: Adding a trigger add function. In Map.lua.

AddFullTrigger takes in a trigger object, a pair of x, y coordinates, and a layer position. The layer parameter is optional and defaults to 1. Triggers are stored by layer. We check the mTriggers table to check if the layer exists, and if it doesn't we create it. The layer entry for a set of triggers only exists if it's needed. We then use the x, y

coordinates to add the entry for the trigger. Any trigger already at this location will be replaced.

Let's use AddFullTrigger to write the AddChest action. Copy the code from Listing 3.123.

```
Actions =
{
    -- code omitted

    AddChest = function(map, entityId, loot, x, y, layer)

        layer = layer or 1

        return function(trigger, entity, tX, tY, tLayer)

            local entityDef = gEntities[entityId]
            assert(entityDef ~= nil)
            local chest = Entity>Create(entityDef)

            chest:SetTilePos(x, y, layer, map)

            local OnOpenChest = function()

                gStack:PushFit(gRenderer, 0, 0,
                               "The chest is empty!", 300)
                map:RemoveTrigger(chest.mTileX, chest.mTileY, chest.mLayer)
                chest:SetFrame(entityDef.openFrame)
            end

            -- Add the user trigger
            local trigger = Trigger>Create( { OnUse = OnOpenChest } )
            map:AddFullTrigger(trigger,
                               chest.mTileX, chest.mTileY, chest.mLayer)
        end
    end,
}
```

Listing 3.123: AddChest action. In Actions.lua.

The AddChest action in Listing 3.123 can be used to add a chest to any map. In fact, it doesn't even have to be a chest! Any object with a closed and open frame will do, so a chest of drawers, bed covers, safe, pot... we could have any of these.

The action takes six parameters:

- map - the map where the chest is added.

- entityId - the id of the entity that represents the chest.
- loot - a table of loot, the contents of the chest.
- tx,tY,tLayer - the map coordinates to add the chest. tLayer is optional and defaults to 1.

The entity is expected to have an openFrame field that's used to get the opened version of the sprite.

The AddChest function is an action, so it returns a function. The returned function adds a chest to the map when run. Let's go over the details. We use entityId to look up the def from the gEntity table. Then we create the entity using the def. By default the entity sets its sprite using the startFrame frame. This means the chest is closed when created.

To add the chest entity to the world, we call its SetTilePos function, passing in its x, y coordinates and its layer position.

To handle the interaction, we define a local function called OnOpenChest. This gets called when the player interacts with the chest. For now the OnOpenChest function is bare-bones; it pushes a dialog box that says "The chest is empty", then it removes the interaction trigger and changes the chest frame to the open version. We'll handle chest loot shortly.

With OnOpenChest defined, we can hook it into the interaction trigger. We create a trigger and assign OnOpenChest as callback for the OnUse command. Then we use our helper method AddFullTrigger to place this trigger over the chest on the map.

Let's test this code out! Copy the changes from Listing 3.124 into your main.lua file.

```
print("Adding chest")
local exploreState = gStack:Top()
local map = exploreState.mMap
local loot = {}

addChestAction = Actions.AddChest(map, "chest", loot, 27, 14)
addChestAction()

function Update()
```

Listing 3.124: Calling the AddChest action. In main.lua.

Run the code and you'll see a chest you can interact with. When used, you'll see something like Figure 3.28.



Figure 3.28: Opening a chest ... but it's empty!

Filling the Chest with Loot

Empty chests are disappointing; let's fill them full of weapons and armor! To do this we need to link loot to the chest.

The player inventory is stored as a list of tables with an item id and item count, like Listing 3.125.

```
inventory =
{
    {id = 1, count = 100 },
    {id = 56, count = 1},
    -- etc
}
```

Listing 3.125: How the player inventory table is stored.

We're going to make our chest loot tables work in the same way as the inventory. Let's edit the main.lua file again and create a more interesting loot table. Copy the code from Listing 3.126.

```

hero_loot =
{
    { id = 1, count = 1 },
    { id = 2, count = 1 },
    { id = 3, count = 1 },
}
addChestAction = Actions.AddChest(map, "chest", hero_loot, 27, 14)
addChestAction()

```

Listing 3.126: Adding loot to the chest. In main.lua.

The loot table contains ids that point to items in the ItemDB. The first three items in the ItemDB table are the Bone Blade, Bone Armor and Ring of Titan. This is the kit for the hero character.

To hook this loot up to the chest, we'll extend the OnOpenChest callback in the AddChest action. Copy the code from Listing 3.127 into the OnOpenChest function.

```

-- code omitted

local OnOpenChest = function()

    if loot == nil or #loot == 0 then

        gStack:PushFit(gRenderer, 0, 0, "The chest is empty!", 300)

    else

        gWorld:AddLoot(loot)

        for _, item in ipairs(loot) do

            local count = item.count or 1
            local name = ItemDB[item.id].name
            local message = string.format("Got %s", name)

            if count > 1 then
                message = message .. string.format(" x%d", count)
            end

            gStack:PushFit(gRenderer, 0, 0, message, 300)
        end
    end

```

```
-- Remove the trigger
map:RemoveTrigger(chest.mTileX, chest.mTileY, chest.mLayer)
chest:SetFrame(entityDef.openFrame)
end
```

Listing 3.127: Extending the OnOpenChest function. In Actions.lua.

The OnOpenChest function tests if the loot table is nil or empty and, if so, displays the message “The chest is empty”. If the loot table exists we call a new function, gWorld:AddLoot. Then we tell the player what they’ve found by looping through the loot table and pushing a message for each item to the screen. If there’s one item, we write “Got [item name]”. If there’s more than one item, we write “Got [item name] x[item count]”. So we might see “Got plate armor” and then “Got Health Potion x3”.

If there are five items in the chest, five dialog boxes will be pushed onto the stack. The player has to click through each dialog box, so we’ll restrict our chest loot to only a few items. Usually chests will only contain a single item.

Copy the AddLoot function from Listing 3.128 into your World.lua file.

```
function World:AddLoot(loot)
    for k, v in ipairs(loot) do
        self:AddItem(v.id, v.count)
    end
end
```

Listing 3.128: Adding an AddLoot function, to deal with lists of item. In World.lua.

Run the game and you’ll be able to open the chest and receive all the hero’s equipment.

Placing Chests in the World

We’re placing the chests directly in the main.lua but it would be tidier to move them into the arena map file.

Let’s begin by defining the loot tables. The map has three chests, one for each member of the party. Each chest contains the equipment needed by one of the party members. Copy the code from Listing 3.129.

```
function CreateArenaMap()

    local loot =
    {
        hero =
```

```

{
    { id = 1 },
    { id = 2 },
    { id = 3 },
},
mage =
{
    { id = 4 },
    { id = 5 },
    { id = 6 },
},
thief =
{
    { id = 7 },
    { id = 8 },
    { id = 9 },
},
}

-- code omitted

```

Listing 3.129: Add loot lists. In map_arena.lua.

In Listing 3.129 we've put the loot tables in the map def.

We'll use the on_wake table in the map to run the AddChest actions. Copy the code from Listing 3.130.

```

return
{
    version = "1.1",
    luaversion = "5.1",
    orientation = "orthogonal",
    width = 30,
    height = 30,
    tilewidth = 16,
    tileheight = 16,
    properties = {},


-- code omitted

on_wake =
{
    -- code omitted
    {

```

```

        id = "AddChest",
        params = { "chest", loot.hero, 27, 14 }
    },
{
    id = "AddChest",
    params = { "chest", loot.mage, 20, 14 }
},
{
    id = "AddChest",
    params = { "chest", loot.thief, 34, 14 }
},
}
,
```

Listing 3.130: Adding the chests to the world via the map. In map_arena.lua.

These three actions in Listing 3.130 add three chests to the map. Before running the code, make sure the main.lua is clear of any of the previous test code. It should appear as below in Listing 3.131.

```

LoadLibrary('Asset')
Asset.Run('Dependencies.lua')

gRenderer = Renderer.Create()
gStack = StateStack>Create()
gWorld = World>Create()

gWorld.mParty:Add(Actor>Create(gPartyMemberDefs.hero))

gStack:Push(ExploreState>Create(gStack, CreateArenaMap(),
    Vector.Create(30, 18, 1)))

function update()
    local dt = GetDeltaTime()
    gStack:Update(dt)
    gStack:Render(gRenderer)
    gWorld:Update(dt)
end

```

Listing 3.131: The clean main code. In main.lua.

The code so far is available in combat/equipment-4-solution. Run the code and you'll see three chests on the map, ready for looting.

Our code base is growing nicely. We can now find and equip items, one of the cornerstones of any RPG. Next we'll look at using the weapons and armor in combat.

Combat Flow

The flow of combat describes what happens during a battle and how it's modelled. RPG combat systems differ widely. We'll implement a traditional system. In our system a battle always contains two teams, the player party and the enemies. The enemies are shown on the left of the battlefield and the player party on the right. You can see this setup in Figure 3.29. Every action that happens during combat is called a combat event. The combat flow is concerned with managing these events.



Figure 3.29: Layout of Actors in a battle.

Our combat is turn-based. Each Actor is asked, in turn, what action they'd like to take. A combat action is an attack, spell, item or any similar action. Turn order is determined by the speed stat.

To model the combat flow, we use a queue of events. The event queue is self-descriptive. It's a queue of events that are due to happen during combat. The event at the top of the queue is going to happen next, the event below that happens after that, and so on. When an event fires it may modify the queue, adding new events or taking old events away.

Imagine a combat instance where there's a mage who is poisoned and only has 1 HP remaining. The event queue might look like Listing 3.132.

```

0: take turn (mage)          0: use item (cure, mage)
1: poison damage (mage) -> 1: poison damage (mage) ->
2: ... other events         2: ... other events

```

Listing 3.132: An Example Event Queue.

In Listing 3.132 we can see that it's the mage's turn. If the mage drinks a cure-poison potion, that event is added to the top of the queue as a new use-item event. When the use-item event runs, it removes the poison effect from the mage and also removes all poison events from the event queue. If the mage chose a different action, the next event would be poison-damage and she'd die!

Each event is associated with a number of *time points*. The event with the lowest number of time points is executed next. Some events like using an item are instant, while others like casting a spell may take several seconds. All actions an actor can take are run using the event queue.

The event queue can model all sorts of things, from the order of character turns to tracking environment effects such a bomb ticking down and exploding.

Combat ends when one side flees or the health of all actors is reduced to zero.

An Event Queue

Let's start with a simple text-based system that uses the event queue. Once that's working, we'll build the structure required for a graphic combat scene and tie it into the current code base.

We'll create two simple events; one to prompt an actor to take a turn and one for a melee attack. All events must implement the fields and functions shown in Listing 3.133.

```

Event =
{
    CountDown = -1,
    function Execute() end,
    function Update() end,
    function IsFinished() end,
    Owner, -- optional
}

```

Listing 3.133: A bare bones event definition.

Listing 3.133 shows the basic event structure.

The CountDown number represents how soon the event is due to execute. The event with the lowest CountDown value is the one executed next. When an event is executed it's removed from the queue, and all other events have their CountDown reduced by one.

The Execute, Update and IsFinished functions are used to run the event until it ends. Imagine an attack. This involves the actor running over to the enemy, playing an attack animation, and returning to their original position. All these movements take time. The attack event continues running over multiple frames until it completes.

Only one event runs at a time. While the top combat event executes, all other events are blocked. No other events are run. Execute is called when the event is first run. Update is called each frame to run the event for as long as it needs. The IsFinished function is used to check when the event has finished.

The Owner field is a tag used to remove all events associated with a certain owner. If an actor dies, all events related to that actor need removing from the queue.

Now that we have a feel for how the event objects work, let's start building up a simple system to process them. The EventQueue is the most important part of the system, so we'll start there.

Example combat/event-queue-1 contains a new minimal project we're using just for the EventQueue. Once we have it working, we'll pull it back into our main codebase.

Make a new file EventQueue.lua. Add it to the manifest.lua. For this small example we won't bother with dependencies lua file. We'll load the lua files directly. Copy the code from Listing 3.134 into your main.lua file.

```
LoadLibrary("System")
LoadLibrary("Renderer")
LoadLibrary("Asset")

Asset.Run("EventQueue.lua")

eventQueue = EventQueue>Create()

eventQueue:Add({ mName = "Msg: Welcome to the Arena"}, -1)
eventQueue:Add({ mName = "Take Turn Goblin" }, 5)
eventQueue:Add({ mName = "Take Turn Hero" }, 4)

eventQueue:Print()

function update()
end
```

Listing 3.134: Setting up a very simple event queue. In main.lua.

In Listing 3.134 we demonstrate how we'd like to use the EventQueue using a simple

test program. We create the EventQueue and then add three stripped-down events. Finally we print the queue to show the event order.

In Listing 3.134 we queue up three events. The first event is a message welcoming the player to the arena. The next two events are a hero and a goblin each taking their turn. Each event is added with a number of time points to represent the speed of the event. The higher the number of time points, the longer the event takes to execute. The Hero with 4 time points should take his turn before the Goblin with 5 time points. The third event is the welcome message. It has -1 time points, meaning it occurs instantly. The EventQueue handles the details of ordering the events. When we print the EventQueue we'll expect to see the message event on top, followed by the hero and then the goblin.

Roughly based on our example, let's implement the EventQueue code. Copy the code Listing 3.135.

```
EventQueue = {}
EventQueue.__index = EventQueue
function EventQueue:Create()
    local this =
    {
        mQueue = {},
        mCurrentEvent = nil
    }

    setmetatable(this, self)
    return this
end

function EventQueue:SpeedToTimePoints(speed)
    local maxSpeed = 255
    speed = math.min(speed, 255)
    local points = maxSpeed - speed
    return math.floor(points)
end

function EventQueue:Add(event, timePoints)
    local queue = self.mQueue

    -- Instant event
    if timePoints == -1 then
        event.mCountDown = -1
        table.insert(queue, 1, event)
    else
        event.mCountDown = timePoints

        -- loop through events
        for i = 1, #queue do
```

```

local count = queue[i].mCountDown

if count > event.mCountDown then
    table.insert(queue, i, event)
    return
end

end

table.insert(queue, event)
end
end

```

Listing 3.135: EventQueue implementation. In EventQueue.lua.

The EventQueue constructor creates a this table with two fields, `mQueue` and `mCurrentEvent`. The `mQueue` field is the table that stores all the events. The front of the queue is `mQueue[1]` and the back is `mQueue[#mQueue]`. Events at the front of the queue are executed first. The `mCurrentEvent` is where the currently running event is stored. If no event is running `mCurrentEvent` is nil.

Events are ordered by their CountDown value. The CountDown value is set when an event is added to the queue. The two main factors influencing the CountDown number are the event itself and the character's speed stat. We consider the CountDown to contain a count of *time points*. Time points represent how long an event takes to occur. To calculate the CountDown the EventQueue contains a helper function called `SpeedToTimePoints`.

In the `SpeedToTimePoints` function `maxSpeed` represents the maximum speed. No creature will ever have a greater speed value than this. The passed-in speed is subtracted from the `maxSpeed` giving the inverse which we can use for the CountDown. The table shows how this is worked out.

Actor	Speed	Inverse Speed
Hero	5	250
Goblin	4	251

Table 3.2: Change from Speed to Inverse Speed.

In this case the hero has a speed of 5 and the goblin has a speed of 4. The hero is faster, so his events should occur first in the EventQueue. Therefore the hero's event should have a *lower* CountDown value.

The `Add` function takes in an event and a number of time points. Time points are stored in the `mCountDown` field. The countdown number is used to insert the event at the correct

position in the EventQueue.

Sometimes an event must occur instantly. In EventQueue terms instant, means on the next available turn. Instant events include things like intro cutscenes, heal potions, etc. If an event is instant, it has a special countdown value of -1 and is inserted right at the front (index 1) of the queue.

If the event isn't instant, it's inserted according to its time point value. A new event may occur before existing events. To add an event, we iterate through the queue until we find an event with a higher countdown and add it there. If no event has a higher countdown, the iteration ends and the event is inserted at the back of the queue.

Let's run through an example in Figure 3.30.

Combat Actions

Goblin: Speed 4, Take Turn

Hero: Speed 5, Take Turn

Message: Instant, "Welcome to the Arena!"

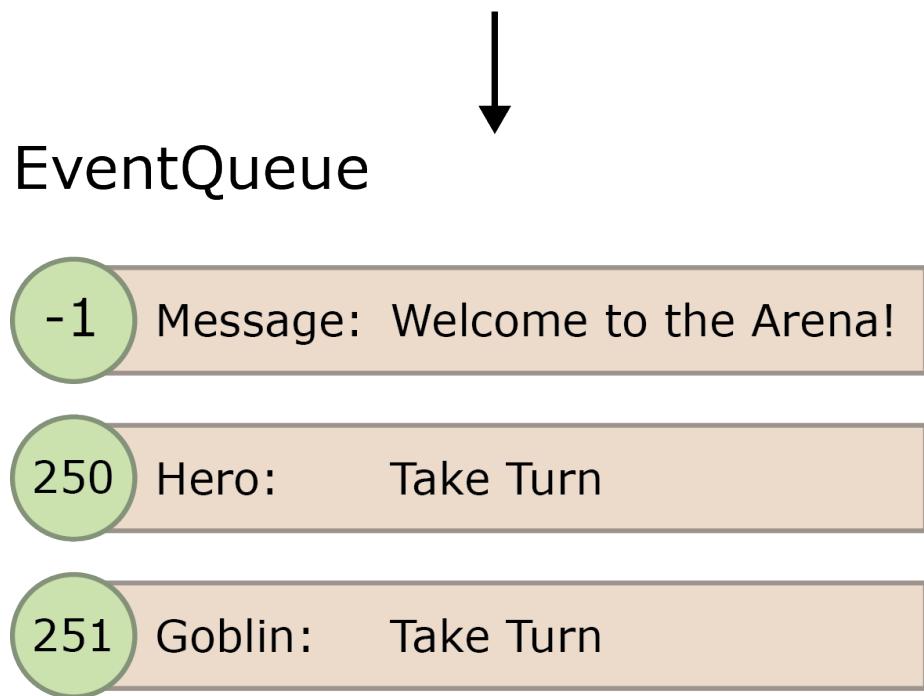


Figure 3.30: How the EventQueue orders combat actions.

In Figure 3.30 you can see how the combat events are arranged by descending number of time points. The instant message goes first because it has a negative time point value. Next the hero will decide what to do for his turn, and then the goblin gets to decide what to do for its turn. Note that events in the queue are executed one after another. The CountDown is only used to determine the order, not how long the player has to wait to see a combat action occur.

Let's add a function called ActorHasEvent to check if a given actor has an event in the queue. Later we can use this function to keep the battle progressing by adding TakeTurn events when actors have no actions. Copy the code from Listing 3.136.

```
function EventQueue:ActorHasEvent(actor)

    local current = self.mCurrentEvent or {}

    if current.mOwner == actor then
        return true
    end

    for k, v in ipairs(self.mQueue) do
        if v.mOwner == actor then
            return true
        end
    end

    return false
end
```

Listing 3.136: ActorHasEvent tests if there's an event in the queue. In EventQueue.lua.

To test if an actor has an event, we check the current event and the events in the queue. The only argument passed in is actor, to test if it has a event.

The current variable is set to the mCurrentEvent, or if nil it's set to an empty table. The mOwner field of the current event is then compared to the actor. If the actor owns the event, we return true. Then we check the rest of the events. If one of the events is owned by the actor, we return true. Otherwise we return false.

Next let's add the RemoveEventsOwnedBy function. This function is called when an actor dies. It removes all events owned by an actor. Copy the code from Listing 3.137.

```
function EventQueue:RemoveEventsOwnedBy(actor)

    for i = #self.mQueue, 1, -1 do
        local v = self.mQueue[i]
        if v.mOwner == actor then
```

```

        table.remove(self.mQueue, i)
    end
end

end

```

Listing 3.137: RemoveEventsOwnedBy function to clear EventQueue of events owned by a certain action. In EventQueue.lua.

The RemoveEventsOwnedBy function loops through the events. If it finds an event owned by the actor, it removes it.

Let's also add a Clear function that removes *all* events from the queue. Copy the code from Listing 3.138.

```

function EventQueue:Clear()
    self.mQueue = {}
    self.mCurrentEvent = nil
end

```

Listing 3.138: Clearing the event queue. In EventQueue.lua.

The Clear function is self-explanatory. Let's move on and add two more functions. IsEmpty tells us if the queue is empty, and Print prints out the queue. The Print function is used for debugging. Copy the code from Listing 3.139.

```

function EventQueue:IsEmpty()
    return self.mQueue == nil or next(self.mQueue) == nil
end

function EventQueue:Print()

    local queue = self.mQueue

    if self:IsEmpty() then
        print("Event Queue is empty.")
        return
    end

    print("Event Queue:")

    local current = self.mCurrentEvent or {}

    print("Current Event: ", current.mName)

```

```

for k, v in ipairs(queue) do
    local out = string.format("[%d] Event: [%d][%s]",
                               k, v.mCountDown, v.mName)
    print(out)
end
end

```

Listing 3.139: The EventQueue Print implementation. In EventQueue.lua.

The IsEmpty function checks if the mQueue table has any entries or is nil. If the queue is empty, the Print function prints “Event Queue is empty.” If the event queue has events, then it prints each event with its position, countdown and name. The front of the queue appears at the top, the back of the queue at the bottom.

We’ve written enough of EventQueue to start testing it. Create a main.lua and add it to the manifest. The test adds placeholder events to the queue and then prints them out. Copy the code from Listing 3.140.

```

LoadLibrary("System")
LoadLibrary("Renderer")
LoadLibrary("Asset")

Asset.Run("EventQueue.lua")

eventQueue = EventQueue:create()

eventQueue:Add({ mName = "Msg: Welcome to the Arena"}, -1)
eventQueue:Add({ mName = "Take Turn Goblin" }, 5)
eventQueue:Add({ mName = "Take Turn Hero" }, 4)

eventQueue:Print()

function update()
end

```

Listing 3.140: Testing the EventQueue with some mock events. In main.lua.

In Listing 3.140 we create the EventQueue and then add three mock events. The first event is an instant event of the kind that might show a message before combat. Then there’s an event to tell the goblin to decide what to do and a similar one for the hero. Each event has a different speed. These events can be added in any order because the EventQueue.Add function sorts them. To see the final order, we print the queue.

Run the code and you'll see how the EventQueue sorts the events. Try mixing up the events until you're satisfied that the EventQueue is working correctly. Listing 3.141 shows the output from the Print function.

```
Event Queue:  
Current Event: nil  
[1] Event: [-1][Msg: Welcome to the Arena]  
[2] Event: [4][Take Turn Hero]  
[3] Event: [5][Take Turn Goblin]
```

Listing 3.141: Output of the EventQueue test.

Listing 3.141 demonstrates how EventQueue correctly orders the events. The EventQueue hasn't had its Update function called yet, so the current event is nil.

Example combat/event-queue-1-solution contains the full EventQueue code.

EventQueue Update Function

The Update function is used to pump the queue and run the events. Use example combat/event-queue-2 or extend your own codebase. Add the Update function from Listing 3.142.

```
function EventQueue:Update()  
  
    if self.mCurrentEvent ~= nil then  
  
        self.mCurrentEvent:Update()  
  
        if self.mCurrentEvent:IsFinished() then  
            self.mCurrentEvent = nil  
        else  
            return  
        end  
  
    elseif self:IsEmpty() then  
  
        return  
  
    else  
  
        -- Need to chose an event  
        local front = table.remove(self.mQueue, 1)  
        front:Execute(self)
```

```

        self.mCurrentEvent = front

    end

    for _, v in ipairs(self.mQueue) do
        v.mCountDown = math.max(0, v.mCountDown - 1)
    end

end

```

Listing 3.142: The EventQueue.Update function that runs the events. In EventQueue.lua.

The Update loop controls when events are executed. Only one event is executed at a time, and it's called `mCurrentEvent`. If `mCurrentEvent` exists, it gets updated. If the `mCurrentEvent` is finished, we set it to `nil`; if not, we do nothing. The `mCurrentEvent` might be an attack animation, using a potion, etc. It can be any combat action. The `EventQueue` doesn't care about the details.

If there isn't a `mCurrentEvent` then we check if there are any other events in the queue. If there are no events in the queue, there's nothing to do and the `Update` function ends. If there are more events, then the event at the front is removed from the queue and set as the `mCurrentEvent`. This new event has its `Execute` function called, to let it know it's now the executing event.

When an event is executed and made current, all the events in the queue have their countdown reduced by one. This means when new events are inserted, they will often go behind older events even if they're a quicker action. We use `math.max` to ensure the countdown doesn't drop below 0.

That's all the `EventQueue` needs to do, process the combat events. The actual interaction with the combat scene is done internally by each event.

To test the `Update` function, we need better event mockups with `Execute`, `IsFinished` and `Update` functions implemented.

Defining Some Events

To test the `EventQueue` we need events. Let's start with two really useful events, take turn and attack. The take turn event tells an actor to decide what to do next. The attack event tells an actor to attack another actor. We'll begin with simple versions of these actions so it's clear what's happening.

CETurn

During combat, many events can occur. Each event that can happen is defined by a class. This means we're going to end up with a lot of combat event classes. To make things clear, we'll start all event class names with "CE" which stands for "Combat Event".

Let's create the take turn event first. We'll name it CETurn. Create a new CETurn.lua file and add it to the manifest, then copy the code from Listing 3.143.

```
CETurn = {}
CETurn.__index = CETurn
function CETurn:Create(scene, owner)
    local this =
    {
        mScene = scene,
        mOwner = owner,
        mName = string.format("CETurn(_, %s)", owner.mName)
    }

    setmetatable(this, self)
    return this
end

function CETurn:TimePoints(queue)
    local speed = self.mOwner.mSpeed
    return queue:SpeedToTimePoints(speed)
end

function CETurn:Execute(queue)

    -- Choose a random enemy target.
    local target = self.mScene:GetTarget(self.mOwner)

    local msg = string.format("%s decides to attack %s",
        self.mOwner.mName,
        target.mName)
    print(msg)
    local event = CEAttack:Create(self.mScene,
        self.mOwner,
        target)
    local tp = event:TimePoints(queue)
    queue:Add(event, tp)
end

function CETurn:Update()
    -- Nothing
```

```

end

function CETurn:IsFinished()
    return true
end

```

Listing 3.143: Creating the TakeTurn Combat Event. In CETurn.lua.

The CETurn class in Listing 3.143 asks the actor “What action do you want to take?” In the enemies’ case, the AI decides. In the player’s case, the combat UI appears and the player chooses an action. For our test, the take turn event only has a single action to choose from – attack. Therefore it immediately adds an attack event to the queue.

The constructor takes in a scene and an owner. We haven’t defined the scene yet, but it describes the combat state and all the actors involved. The owner is the actor who wants to take the turn. The constructor assigns these to fields in the this table.

The CETurn class defines a TimePoints function that returns the number of time points the action takes. It uses the EventQueue to call the SpeedToTimePoints helper function.

In our simple example, the CETurn occurs instantly. Everything important is done in the Execute function. The event only runs for one frame. Therefore the Update function is empty and the IsFinished function returns true.

The Execute function uses two classes we haven’t written yet, scene and CEAttack. How we’ve used them in Listing 3.143 gives some hints about how we’ll implement them.

CETurn chooses a random enemy, creates an attack event, and adds it to the queue. We define a variable called target for the actor we want to attack. If the event owner is a party member, we want to attack an enemy. If the event owner is an enemy, we want to attack a party member. The GetTarget function randomly chooses an appropriate target. At the end of the Execute function, we calculate the time points for the attack event and insert it into the EventQueue.

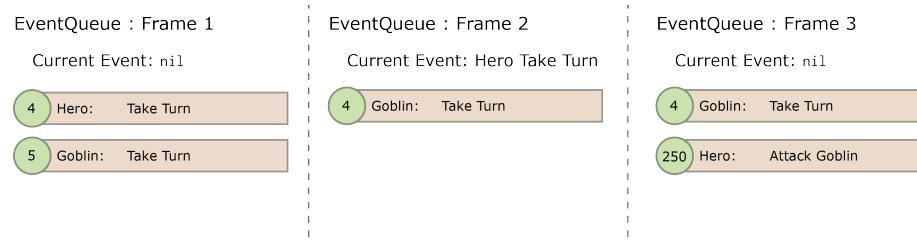


Figure 3.31: Executing a take turn action.

In Figure 3.31 you can see what happens to the event queue during three frames of the game. The EventQueue begins with two CETurn events in the queue. The hero has the

lowest countdown value, so his event becomes the current event. The rest of the events have their countdown reduced by one. In the next frame, the CETurn is executed and the hero adds a CEAttack event to the queue. By the third frame the current event is nil again, so we need to execute the next event. This is the basic pumping action of the queue that keeps the combat flowing. Next let's add the CEAttack.

CEAttack

The only thing an actor can do currently is attack. Attacking deducts the attacker's attack score from the target's health. If the target's health points drop below zero, they're killed and removed from combat. Party members are knocked out instead of being killed.

Create a new CEAttack.lua file and copy in the code from Listing 3.144.

```
CEAttack = {}
CEAttack.__index = CEAttack
function CEAttack:Create(scene, owner, target)
    local this =
    {
        mScene = scene,
        mOwner = owner,
        mTarget = target,
        mName = string.format("CEAttack(_, %s, %s)",
                               owner.mName,
                               target.mName)
    }

    setmetatable(this, self)
    return this
end

function CEAttack:TimePoints(queue)
    local speed = self.mOwner.mSpeed
    return queue:SpeedToTimePoints(speed)
end

function CEAttack:Execute(queue)

    local target = self.mTarget

    -- Already killed!
    if target.mHP <= 0 then

        -- Get a new random target
```

```

        target = self.mScene:GetTarget(self.mOwner)
    end

    local damage = self.mOwner.mAttack
    target.mHP = target.mHP - damage

    local msg = string.format("%s hit for %d damage",
                               target.mName,
                               damage)
    print(msg)

    if target.mHP < 0 then

        local msg = string.format("%s is killed.", target.mName)
        print(msg)

        self.mScene:OnDead(target)
    end

end

function CEAttack:Update()
    -- Once again this is instant
end

function CEAttack:IsFinished()
    return true
end

```

Listing 3.144: The Attack CombatEvent. In CEAttack.lua.

In our simple example, the CEAttack occurs instantly. The constructor is the same as CETurn but with an additional target parameter which is another actor.

The TimePoints function again calculates the time points for the event based on the actor's speed. In a more advanced RPG game, we might take into account the weapon being used and adjust the time points according to that, but let's keep things simple!

The Execute function is where the meat is. First there's a check to see if the target is dead or not. In the early Final Fantasy games, if you attacked a creature but it died before your attack, you'd still attack the corpse! Not fun. In our game, if the target has been defeated we just choose another random target.

The event owner's attack power is stored in a damage variable and subtracted from the target's hp. Then we print out a message reporting what's happened and check if the

target died. If the target dies, we print out another message and tell the scene to handle it.

That's all there is to the attack. To test our events out, let's implement the combat scene.

CombatScene

The combat scene stores two lists of actors and uses an EventQueue to control the combat flow. It also implements the OnDeath and GetTarget functions that we've already used. We'll go through the class in a few sections. Create a CombatScene.lua file and copy in the code from Listing 3.145.

```
CombatScene = {}
CombatScene.__index = CombatScene
function CombatScene:Create(party, enemies)
    local this =
    {
        mPartyActors = party or {},
        mEnemyActors = enemies or {},
        mEventQueue = EventQueue>Create()
    }

    setmetatable(this, self)
    this:AddTurns(this.mPartyActors)
    this:AddTurns(this.mEnemyActors)
    return this
end

function CombatScene:Update()

    self.mEventQueue:Update()

    if self:IsPartyDefeated() or self:IsEnemyDefeated() then
        -- END GAME DETECTED
        self.mEventQueue.mQueue = {}
    else
        -- keep the queue pumping
        self:AddTurns(self.mPartyActors)
        self:AddTurns(self.mEnemyActors)
    end
end

function CombatScene:AddTurns(actorList)
```

```

        for _, v in ipairs(actorList) do

            if not self.mEventQueue:ActorHasEvent(v) then
                local event = CETurn:Create(self, v)
                local tp = event:TimePoints(self.mEventQueue)
                self.mEventQueue:Add(event, tp)
            end

        end
    end

```

Listing 3.145: Combat Scene constructor and update loop. In CombatScene.lua.

Listing 3.145 contains a simple CombatScene object of the same type we'll use in the full game. The constructor takes in two lists of actors, party and enemies. The party list contains the player-controlled actors. The enemies list contains the AI-controlled actors. In the constructor, these two actor lists are assigned and the EventQueue is created.

The Update function pumps the EventQueue. Each update, we check if the combat is finished. If one side is defeated, we empty the queue of events to stop combat. If combat hasn't ended, AddTurns is called for the party and enemies. AddTurns checks if each actor has an event in the queue. If not, it adds a CETurn event. The AddTurns function is also used in the constructor to set up the combat.

The Update function uses two scene functions we've yet to write, IsPartyDefeated and IsEnemyDefeated. Let's write those next. Copy the code from Listing 3.146.

```

function CombatScene:IsPartyDefeated()

    for _, actor in ipairs(self.mPartyActors) do
        if not actor:IsK0ed() then
            return false
        end
    end

    return true
end

function CombatScene:IsEnemyDefeated()
    return #self.mEnemyActors == 0
end

```

Listing 3.146: Checking functions to see if combat is over. In CombatScene.lua.

JRPGs always favor the player party in combat. Enemies die but party members merely get knocked out. Our combat example ends in one of two ways: the player party

gets knocked out or all the enemies are killed. Listing 3.146 contains two checks. IsPartyDefeated loops through the player party and checks if everyone is KO'ed. IsEnemyDefeated checks if the enemy list is empty or not. We use these functions to determine who wins the battle.

Next let's implement GetTarget. This function is used by CEAttack. Copy the code from Listing 3.147.

```
function CombatScene:GetTarget(actor)

    local targetList = nil

    if actor:IsPlayer() then
        targetList = self.mEnemyActors
    else
        targetList = self:GetLivePartyActors()
    end

    return targetList[#targetList]

end

function CombatScene:GetLivePartyActors()

    local live = {}

    for _, actor in ipairs(self.mPartyActors) do
        if not actor:IsKOed() then
            table.insert(live, actor)
        end
    end

    return live
end
```

Listing 3.147: GetTarget function for acquiring a target in combat. In CombatScene.lua.

In Listing 3.147 we define GetTarget. GetTarget takes an actor and returns a target for that actor to attack. The returned target is a random actor from the opposing side. First the list of possible targets is created. If the attacker is a player, the list is the mEnemyActor list. If the attacker is an enemy, we use GetLivePartyActors to get the list of non-KO'ed party members. Once we have the targetList we pick a random actor and return it.

The helper function GetLivePartyActors makes a list of all party members that aren't

KO'ed. It loops through the `mPartyActors` and filters out the live actors into a new list which it returns.

As is traditional, let's end with death. Copy the `OnDead` function from Listing 3.148 into your `CombatScene.lua` file.

```
function CombatScene:OnDead(actor)
    if actor:IsPlayer() then
        actor:KO()
    else
        for i = #self.mEnemyActors, 1, -1 do
            local enemy = self.mEnemyActors[i]
            if actor == enemy then
                table.remove(self.mEnemyActors, i)
            end
        end
    end

    -- Remove owned events
    self.mEventQueue:RemoveEventsOwnedBy(actor)

    if self:IsPartyDefeated() then
        print("Party loses")
    elseif self:IsEnemyDefeated() then
        print("Party wins")
    end
end
```

Listing 3.148: Handling death in combat. In `CombatScene.lua`.

When a party member dies, they are knocked out and the `KO` function is called. When an enemy actor dies, they're removed from the enemy list. Dead actors have all their events removed from the `EventQueue`. The function ends by testing if the player won or lost, then displays the appropriate message.

We're now ready to test this code in the `main.lua` file.

Testing Combat

To test out the combat, we first create the `CombatScene` in `main.lua`. The code simulates a battle between the character and a goblin. We haven't fully defined all the `Actor` fields and functions yet. To make the code work, we'll directly create the hero and goblin actor tables.

Create a `main.lua`, hook it up, and copy the code from Listing 3.149.

```

LoadLibrary("System")
LoadLibrary("Renderer")
LoadLibrary("Asset")

Asset.Run("CombatScene.lua")
Asset.Run("EventQueue.lua")
Asset.Run("CTurn.lua")
Asset.Run("CEAttack.lua")
Asset.Run("EventQueue.lua")

gCombatScene = CombatScene>Create(
{
    -- our hero
    {
        mName = "hero",
        mSpeed = 3,
        mAttack = 2,
        mHP = 5,
        IsPlayer = function() return true end,
        IsKOed = function(self) return self.mHP <= 0 end,
    },
},
{
    -- enemies goblin
    {
        mName = "goblin",
        mSpeed = 2,
        mAttack = 2,
        mHP = 5,
        IsPlayer = function() return false end,
        IsKOed = function(self) return self.mHP <= 0 end,
    },
})

print("---start---")

function update()
    gRenderer:AlignText("center", "center")
    gRenderer:DrawText2d(0, 0, "Testing Combat")
    gCombatScene:Update()
end

```

Listing 3.149: Adding a CombatScene game. In main.lua.

Listing 3.149 sets up the combat scene with two lists of combatants. Each list contains

a single actor. On the party team there's the hero, and on the monster team there's a goblin. The stats for both hero and goblin are almost the same, except that the hero is 1 point quicker.

Combat is simulated by calling `CombatScene:Update` in the update loop. As it executes, it prints updates to the console.

The full code is available in `combat/event-queue-2-solution`. Run the program and you'll see output like Listing 3.150.

```
--start--  
hero decides to attack goblin  
goblin hit for 2 damage  
goblin decides to attack hero  
hero decides to attack goblin  
hero hit for 2 damage  
goblin hit for 2 damage  
goblin decides to attack hero  
hero decides to attack goblin  
hero hit for 2 damage  
goblin hit for 2 damage  
goblin is killed.  
Party wins
```

Listing 3.150: Output of the combat simulation.

The player wins because the hero is faster. Try changing the stats and see how the combat plays out. This kind of automated combat is useful for balancing. Next we'll head into the more exciting graphical version of the combat.

Combat State

In the last chapter we created a small test battle and built the `EventQueue`. In this chapter we'll lay the groundwork to let us pull the `EventQueue` back into the main codebase. We'll create a new `CombatState` class to handle the combat simulation. Then we'll get our party combat-ready with new animations and states, and add a sprinkle of monsters to the game. We'll also create a new UI to show the current combat status.

Getting Ready for Combat

In JRPGs, combat usually occurs in a special, separate scene. This scene has a background to show where the combat is taking place. Monsters usually line up on the right, the player character party on the left.

The basic combat scene is shown in Figure 3.32.

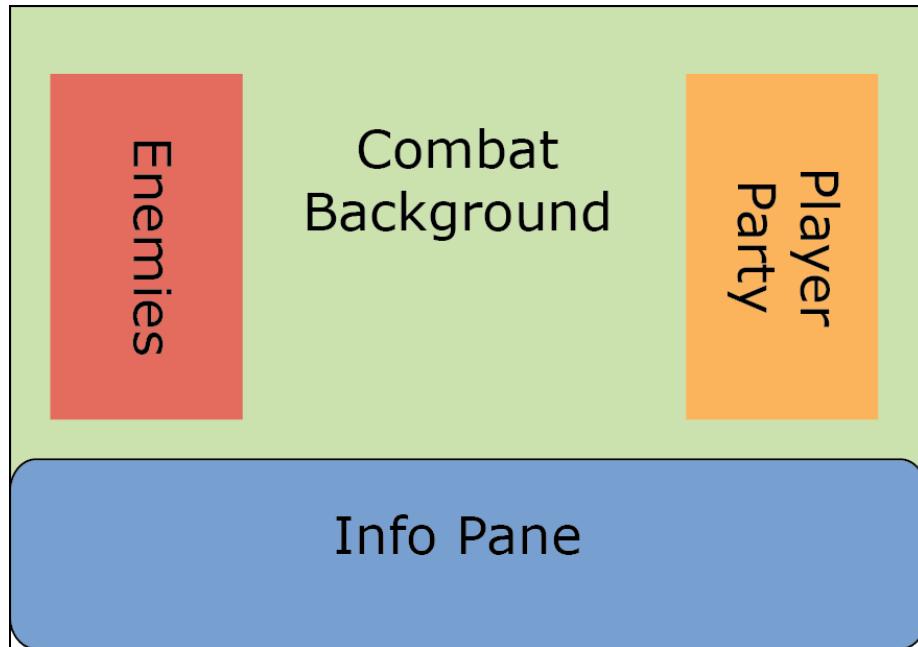


Figure 3.32: A breakdown of the main combat screen elements.

An up-to-date version of the codebase is available in example/combat-1. The example contains

- **arena_background.png** - the backing image that shows the setting for the combat
- **combat_hero.png** - sprite used to represent the hero during combat
- **combat_thief.png** - sprite used to represent the thief during combat
- **combat_mage.png** - sprite used to represent the mage during combat

These files have already been added to the manifest. The combat background is the same size as our screen size, 640 x 360 pixels.

Let's begin by creating a skeleton CombatState class. Create CombatState.lua and add it to the dependencies and manifest files. Then copy in the code from below Listing 3.151.

```
CombatState = {}  
CombatState.__index = CombatState  
function CombatState:Create()
```

```

local this = {}

setmetatable(this, self)
return this
end

function CombatState:Enter()
end

function CombatState:Exit()
end

function CombatState:Update(dt)
end

function CombatState:HandleInput()
end

function CombatState:Render(renderer)

renderer:DrawRect2d(
    -System.ScreenWidth() / 2,
    System.ScreenHeight() / 2,
    System.ScreenWidth() / 2,
    -System.ScreenHeight() / 2,
    Vector.Create(0, 0, 0, 1))

renderer:AlignText("center", "center")
renderer:ScaleText(1.5, 1.5)
renderer:DrawText2d(0, 0, "Combat State")

end

```

Listing 3.151: A state to control combat. In CombatState.lua.

The skeleton combat state in Listing 3.151, when run, renders the words “Combat State” on a black background. All the combat simulation will occur in this state. To make testing easier, let’s edit the main file to automatically push the combat state on the top of the stack.

Copy the code from Listing 3.152.

```

gCombatDef =
{
```

```

        background = "arena_background.png"
    }

gStack:Push(ExploreState>Create(gStack, CreateArenaMap(),
    Vector.Create(30, 18, 1)))
gStack:Push(CombatState>Create(gStack, gCombatDef))

function update()
    local dt = GetDeltaTime()
    gStack:Update(dt)
    gStack:Render(gRenderer)
    gWorld:Update(dt)
end

```

Listing 3.152: Entering the combat state on game load. In main.lua.

The party sprites for combat are bigger than the sprites for exploring the map. Instead of 16x24 they're 64x64 pixels. Combat sprites need more space for swinging swords and casting spells. The combat sprites have separate entity definitions. Add the new definitions by copying the code from Listing 3.153.

```

gEntities =
{
    -- code omitted
    combat_hero =
    {
        texture = "combat_hero.png",
        width = 64,
        height = 64,
        startFrame = 37,
    },
    combat_thief =
    {
        texture = "combat_thief.png",
        width = 64,
        height = 64,
        startFrame = 37,
    },
    combat_mage =
    {
        texture = "combat_mage.png",
        width = 64,
        height = 64,
        startFrame = 37,
    },
}

```

Listing 3.153: Adding party member combat entities. In EntityDefs.lua.

The startFrame in the definitions has the character face right, towards the enemy.

Next let's update the character definitions so they store a combat entity reference. Copy the code from Listing 3.154.

```
gCharacters =
{
    hero =
    {
        -- code omitted
        combatEntity = "combat_hero",
        anims =
        {
            standby = {36, 37, 38, 30},
            -- code omitted
        },
        thief =
        {
            -- code omitted
            combatEntity = "combat_thief",
            anims =
            {
                standby = {36, 37, 38, 30},
                -- code omitted
            },
            mage =
            {
                -- code omitted
                combatEntity = "combat_mage",
                anims =
                {
                    standby = {36, 37, 38, 39},
                    -- code omitted
                }
            }
        }
    }
}
```

Listing 3.154: Adding combat entity references to the party member definitions. In EntityDefs.lua.

In Listing 3.154 we've added a reference to the combat entity to each character definition. We've also added a new animation set called standby which is the default idle animation that plays during combat. The hero's standby animation is shown in Figure 3.33. When we create a combat state in the game, we'll use these new combat entities and animations.



Figure 3.33: The hero's combat standby loop.

In the main file, we created a CombatState object and pushed it onto the state stack. Our current CombatState code is a simple skeleton. Let's start to fill it in! The CombatState constructor takes in the stack and a definition table. We've already defined a definition table in the main file. Currently the combat definition table contains only one field, an image to display as the combat background. Let's update the CombatState to draw the background image. Copy the code from Listing 3.155.

```
function CombatState:Create(stack, def)
    local this =
    {
        mGameStack = stack,
        mStack = StateStack>Create(),
        mDef = def,
        mBackground = Sprite.Create(),
    }

    this.mBackground:SetTexture(Texture.Find(def.background))

    setmetatable(this, self)
    return this
end

-- code omitted

function CombatState:Render(renderer)
    renderer:DrawSprite(self.mBackground)
end
```

Listing 3.155: Adding the background to the combat state. In CombatState.lua.

In Listing 3.155 the main stack is stored as `mGameStack`. Then we create a new stack called `mStack` for states used in combat, such as a state to pick a target to attack,

messages boxes, etc. Creating a separate `mStack` gives the `Combatstate` greater control over what is displayed in each frame. The `mGameStack` will usually contain the `ExploreState` with the `CombatState` on top.

The background sprite is created in the `this table` under the name `mBackground`. The texture of the sprite is set using the `def`.

The render function draws the background. Run the code. You'll get something like Figure 3.34.

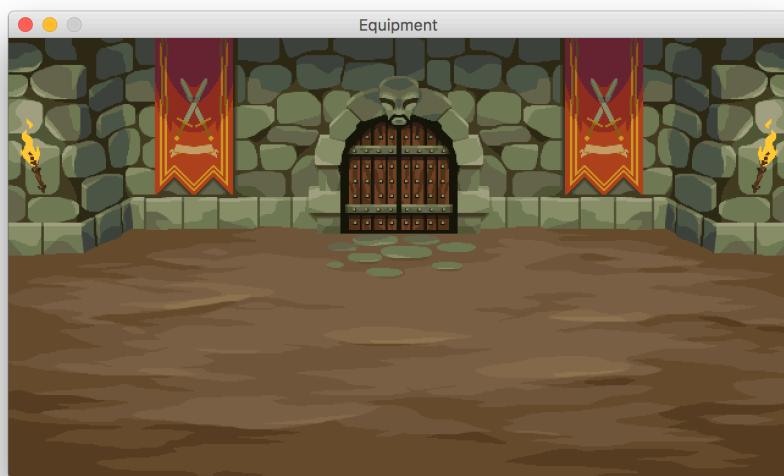


Figure 3.34: The combat background.

Combat is pretty boring without any actors on the field, so let's add the party members!

Combat Layout Definitions

On a traditional battlefield, position is everything. On our simulated battlefield, position is only cosmetic. The party members and monsters change their layout according to how many units there are. The illustration in Figure 3.35 shows some possible layouts.

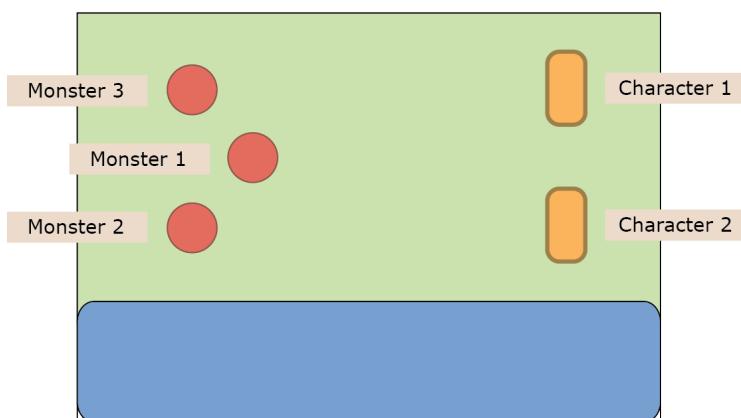
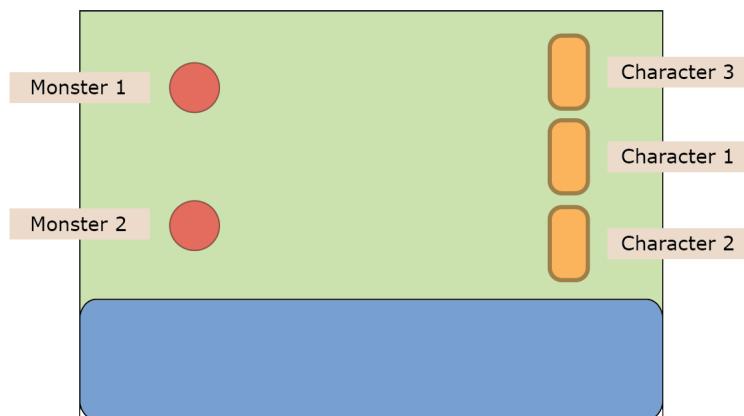
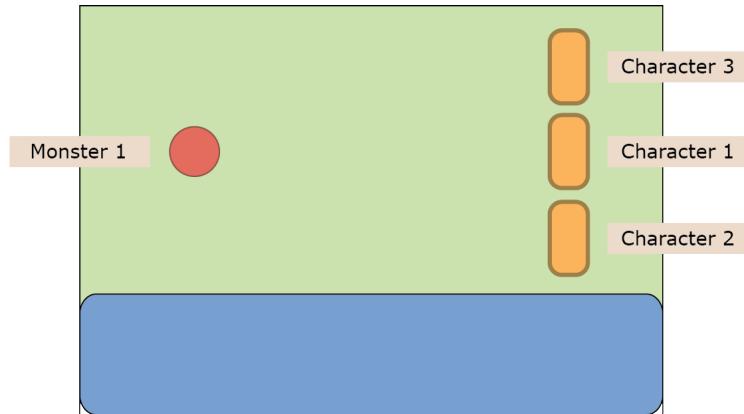


Figure 3.35: Different combat layouts.

We'll transform these different layouts into Lua code and store them in the CombatState. When the combat state is entered, we'll choose the most appropriate layout for the battle.

The bottom quarter of the screen is reserved for the combat info panel. Therefore our layout positions live in the top three quarters of the screen. We're going to have a maximum of three party members and a minimum of one. For monsters, we'll have minimum of one and a maximum of six. Layout data needs to work for one to six monsters against one to three party members.

Layouts are tables of positions for each enemy and party member. Check out Listing 3.156, and add it to your CombatState.lua file.

```
CombatState =  
{  
    Layout =  
    {  
        ["party"] =  
        {  
            {  
                Vector.Create(0.25, -0.056),  
            },  
            {  
                Vector.Create(0.23, 0.024),  
                Vector.Create(0.27, -0.136),  
            },  
            {  
                Vector.Create(0.23, 0.024),  
                Vector.Create(0.25, -0.056),  
                Vector.Create(0.27, -0.136),  
            },  
        },  
        ["enemy"] =  
        {  
            {  
                Vector.Create(-0.25, -0.056),  
            },  
            {  
                Vector.Create(-0.23, 0.024),  
                Vector.Create(-0.27, -0.136),  
            },  
            {  
                Vector.Create(-0.21, -0.056),  
                Vector.Create(-0.23, 0.024),  
            },  
        },  
    },  
}
```

```
        Vector.Create(-0.27, -0.136),  
    },  
    {  
        Vector.Create(-0.18, -0.056),  
        Vector.Create(-0.23, 0.056),  
        Vector.Create(-0.25, -0.056),  
        Vector.Create(-0.27, -0.168),  
    },  
    {  
        Vector.Create(-0.28, 0.032),  
        Vector.Create(-0.3, -0.056),  
        Vector.Create(-0.32, -0.144),  
        Vector.Create(-0.2, 0.004),  
        Vector.Create(-0.24, -0.116),  
    },  
    {  
        Vector.Create(-0.28, 0.032),  
        Vector.Create(-0.3, -0.056),  
        Vector.Create(-0.32, -0.144),  
        Vector.Create(-0.16, 0.032),  
        Vector.Create(-0.205, -0.056),  
        Vector.Create(-0.225, -0.144),  
    },  
},  
}  
}
```

Listing 3.156: Add Layout positions. In CombatState.lua.

All positions are in the range 0 - 1. Each number is a percentage of screen width and height offset from the center of the screen.

These layouts handle all situations we'll deal with in the book. In your own game, you might find this table needs extending or modifying. For instance, if you have multiple large enemies on screen at the same time.

Party Member Layout

Now that we have layout data, let's create the characters and render them to the screen. The Actor class stores skills and stats, and the Character class visually controls the actor on screen. This is the true for both the party members and the enemies.

Let's update the combat def in the main.lua file to include lists of the party and the enemy combatants. We haven't defined any enemies yet, so we'll start with the party members. Copy the code from Listing 3.157.

```

gWorld.mParty:Add(Actor>Create(gPartyMemberDefs.hero))

gCombatDef =
{
    background = "arena_background.png",
    actors =
    {
        party = gWorld.mParty:ToArray(),
        enemy = {}
    }
}

```

Listing 3.157: Creating a combat definition. In main.lua.

We're using a new helper function called `ToArray`. Add it by copying the code from Listing 3.158.

```

function Party:ToArray()
    local array = {}

    for k, v in pairs(self.mMembers) do
        table.insert(array, v)
    end

    return array
end

```

Listing 3.158: Adding `ToArray` to the `Party` class. In `Party.lua`.

The `to ToArray` function transform the `mMembers` dictionary into a simple list of actors. For combat, it's easier to work with a simple array of actors.

When we enter combat, we'll create new character objects for each actor taking part. A new function called `CreateCombatCharacters` will handle creating the characters for combat, but first we'll add code to use it in the constructor. Copy the code from Listing 3.159.

```

function CombatState:Create(stack, def)

    local this =
    {
        -- code omitted
        mActors =
        {

```

```

        party = def.actors.party,
        enemy = def.actors.enemy,
    },
    mCharacters =
    {
        party = {},
        enemy = {}
    },
    mSelectedActor = nil,
    mActorCharMap = {},
}

this.mBackground:SetTexture(Texture.Find(def.background))

setmetatable(this, self)

this>CreateCombatCharacters('party')
this>CreateCombatCharacters('enemy')

return this
end

```

Listing 3.159: Setting up Actor and Character lists in the constructor. In CombatState.lua.

The CombatState constructor expects a list of party and enemy actors in the def. We assign these lists to an mActors table. Then we create a mCharacters table with entries for the enemy and party characters. Near the end of the constructor, CreateCombatCharacters is called to fill in the mCharacters table for the party and enemy.

The mSelectedActor field is used to highlight the selected actor during combat. The mActorCharMap is a table with actors as keys and characters as values. It connects each actor with its associated character.

Copy the code for the CreateCombatCharacters function from Listing 3.160.

```

function CombatState>CreateCombatCharacters(side)

    local actorList = self.mActors[side]
    local characterList = self.mCharacters[side]
    local layout = CombatState.Layout[side][#actorList]

    -- Create a character for each actor
    for k, v in ipairs(actorList) do

```

```

local charDef = ShallowClone(gCharacters[v.mId])

if charDef.combatEntity then
    charDef.entity = charDef.combatEntity
end

local char = Character>Create(charDef, self)
table.insert(characterList, char)
self.mActorCharMap[v] = char

local pos = layout[k]

-- Combat positions are 0 - 1
-- Need scaling to the screen size.
local x = pos:X() * System.ScreenWidth()
local y = pos:Y() * System.ScreenHeight()
char.mEntity.mSprite:SetPosition(x, y)
char.mEntity.mX = x
char.mEntity.mY = y

end

end

```

Listing 3.160: Create combat characters table. In CombatState.lua.

CreateCombatCharacters creates a list of characters from the actor list and stores it in mCharacters. It also fills out the mActorCharMap so we can quickly retrieve the character associated with an actor. It takes in one parameter, side, which is either “party” or “enemy”.

Look at CombatState and you’ll see the actor and character lists both have “party” and “enemy” keys. We use the side variable to choose which side of the battlefield we’re dealing with, the party or the enemies. The side is also important when considering the layout. We use the number of actors a side has to retrieve the correct layout.

Once we have the actorList, we loop through it and use each actor’s mId field to look up the character def. Then we create a character using the def. The character represents the actor on screen during combat. Note that we pass in the CombatState to the Character constructor where previously we’d passed in the ExploreState. This is because these characters exist in combat state rather than on a tile map. Every character is added to the mActorCharMap using the actor as a key and the character as the value. We also add all characters to the characterList. Before leaving the loop, the character position is set using the layout. This sets up all the data we’ll need to show the actors on the battlefield and update them during the combat simulation.

The characters now exist but we can't see them! Let's update the render function. Copy the code from Listing 3.161.

```
function CombatState:Render(renderer)

    renderer:DrawSprite(self.mBackground)

    for k, v in ipairs(self.mCharacters['party']) do
        v.mEntity:Render(renderer)
    end

    for k, v in ipairs(self.mCharacters['enemy']) do
        v.mEntity:Render(renderer)
    end

end
```

Listing 3.161: Rendering the characters out. In CombatState.lua.

The Render function draws each character at their assigned layout position. Run the code and you'll see something like Figure 3.36. The code so far is available as combat-1-solution.

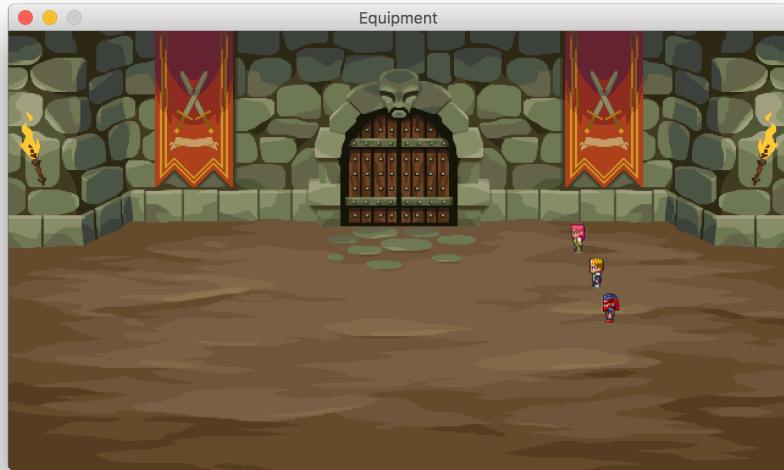


Figure 3.36: The combat background.

Figure 3.36 shows we're getting there! Let's add the enemies next.

The First Enemy

Project example/combat-2 contains everything we've written so far as well as two new files, goblin.png and EnemyDefs.lua. These files have been added to the manifest and dependencies files. You can either use combat-2 or copy the files over to your current codebase.

Goblin.png is a single texture containing one enemy, the goblin, that our party will fight. The EnemyDefs file is where we'll store the actor definitions for the enemies.

To add a new enemy into the game we'll add an actor, entity and character definition. Start by opening the EnemyDefs.lua file and copying in the code from Listing 3.162.

```
gEnemyDefs =  
{  
    goblin =  
    {  
        id = "goblin",  
        stats =  
        {  
            ["hp_now"] = 87,  
            ["hp_max"] = 87,  
            ["mp_now"] = 0,  
            ["mp_max"] = 0,  
            ["strength"] = 8,  
            ["speed"] = 8,  
            ["intelligence"] = 2,  
        },  
        name = "Arena Goblin",  
        actions = { "attack" }  
    }  
}
```

Listing 3.162: Adding our first enemy actor def. In EnemyDefs.lua.

The gEnemyDefs table lists all our enemy actor definitions. This is where we can tweak the enemy stats. This goblin enemy doesn't have equipment. Instead, its raw strength determines how much damage it inflicts. Note that it has one action, "attack". This is the only combat action it can perform. More complicated enemies may have several different actions controlled by a simple AI, but in our game the enemies will mostly just attack.

Next let's add a character def to control the enemy during combat. We'll also add an entity def to tell us which texture files to use. Copy the code into EntityDefs.lua from Listing 3.163.

```

gEntities =
{
    -- code omitted
    goblin =
    {
        texture = "goblin.png",
        startFrame = 1,
        width = 32,
        height = 32,
    },
}

gCharacters =
{
    -- code omitted
    goblin =
    {
        entity = "goblin",
        controller = { "npc_stand" },
        state = "npc_stand",
    }
}

```

Listing 3.163: Adding entity and character definitions. In EntityDef.lua.

The entity def references the enemy sprite and uses the full size of the image. There's only one frame, so the initial frame is set to that.

The goblin character definition references the goblin entity and creates a simple controller with a stand action. The controller is a placeholder. When we add a little life to the enemy, we'll add more interesting states.

With the enemy defined, add him into the game by editing the main.lua file. Copy the code from Listing 3.164.

```

gCombatDef =
{
    background = "arena_background.png",
    actors =
    {
        party = gWorld.mParty:ToArray(),
        enemy =
        {
            Actor>Create(gEnemyDefs.goblin)
        }
    }
}

```

```
    }
}
```

Listing 3.164: Filling out the enemy side of the combat def. In main.lua.

In Listing 3.164 we've added a goblin to the combat def. The goblin is created as an actor using the def from the gEnemyDefs table. Run the code now and you'll get an image like in Figure 3.37.



Figure 3.37: A goblin added to the combat state.

Figure 3.37 shows the party and enemy actors standing off against each other.

Combat States

During combat, each actor is controlled by a state machine. The states include such things as waiting to be told what to do, attacking an enemy, and so on. Each state contains a small amount of code.

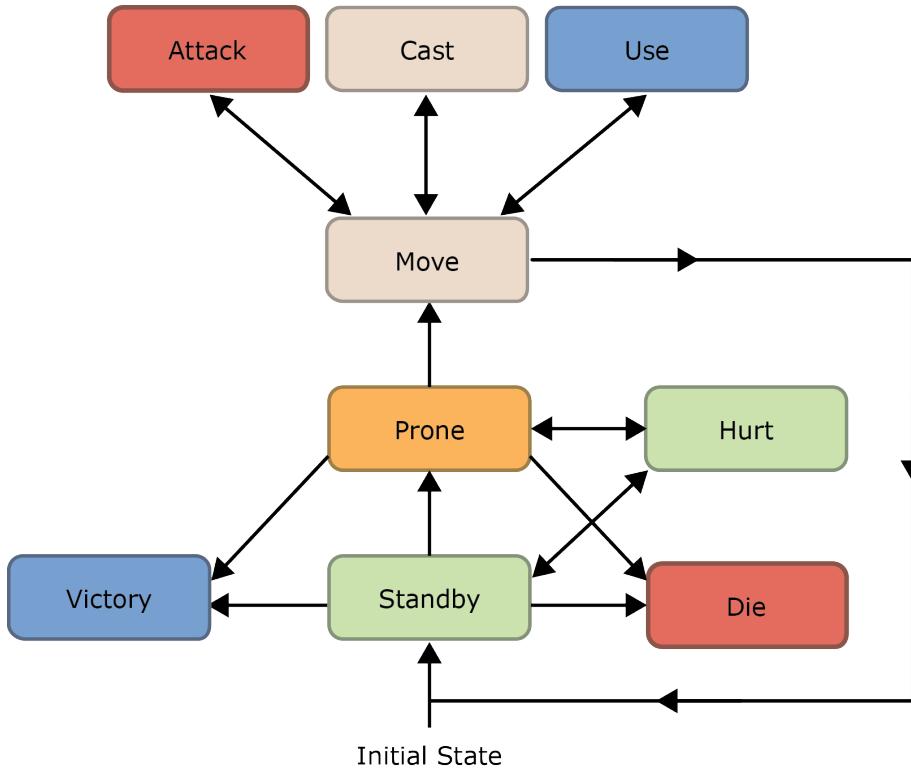


Figure 3.38: The combat states.

Figure 3.38 shows some of the basic states that player characters use in combat. This state graph is easy to extend later if we want to add special skills or spells. States like attack, cast, and use item block the combat simulation until they finish. This is why there are no state transitions from these states to die, victory, or hurt.

Here is a list of the states we'll use in our combat simulation.

- **Standby** - The character is waiting to be told what action to do by the player or AI.
- **Prone** - The character is waiting and ready to perform a given combat action.
- **Attack** - The character will run an attack animation and attack an enemy.
- **Cast** - The character will run a cast-spell animation and a special effect will play.
- **Use** - The character uses some item with a use-item animation.
- **Hurt** - The character takes some damage. Animation and numbers.
- **Die** - The character dies and the sprite is changed to the death sprite.
- **Move** - The character moves toward or away from the enemy, in order to perform an action.

- **Victory** - The character dances around and combat ends.

Most of these states simple play an animation such as Prone and Victory while Move and Hurt are more complicated. States that only run an animation use almost exactly the same code, so rather than repeating ourselves, we'll create a special RunAnim state.

Enemy actors and player actors share states where possible, but when it makes sense each will have separate states. In our game, the enemies are static with no animations while the party members have many special animations.

Creating the Combat States

Let's add the skeleton definitions for *all* the states at once! To make clear which states are combat states, we'll prefix 'CS' to each state. Create four files in /code/combat_states/.

- CSRunAnim.lua - This will use the state id "cs_run_anim".
- CSHurt.lua - This will use the state id "cs_hurt".
- CSMove.lua - This will use the state id "cs_move".
- CSStandby.lua - This will use the state id "cs_standby".

For each file, copy in the skeleton state code from Listing 3.165, changing the ids and class names as appropriate.

```
CSStandby = { mName = "cs_standby" }
CSStandby.__index = CSStandby
function CSStandby:Create(character, context)
    local this =
    {
        mCharacter = character,
        mCombatScene = context,
    }

    setmetatable(this, self)
    return this
end

function CSStandby:Enter()
end

function CSStandby:Exit()
end

function CSStandby:Update(dt)
```

```
end

function CSStandby:Render(renderer)
end
```

Listing 3.165: Skeleton for the CSStandby state. In CSStandby.lua.

Add all the states to the manifest and dependencies file. Remember, these states are in a subdirectory so the manifest should look like Listing 3.166.

```
['CSRunAnim.lua'] =
{
    path = "code/combat_states/CSRunAnim.lua"
},
-- etc.
```

Listing 3.166: Example of adding a script asset in subdirectory. In manifest.lua.

In the dependencies file, add the states *before* EntityDefs.lua, as the states are used by EntityDefs when it's first run. Listing 3.167 shows the correct place to insert them.

```
Apply({
    -- code omitted
    "MoveState.lua",
    "WaitState.lua",
    "CSRunAnim.lua",
    "CSHurt.lua",
    "CSMove.lua",
    "CSStandby.lua",
    "NPCStandState.lua",
    -- code omitted
```

Listing 3.167: Correctly add the combat states. In Dependencies.lua.

Each state in Listing 3.167 plays new combat animations. The party sprite sheets already contain the animation data; we just need to add the frames to the character definitions. This is going to be a lot of data, so get ready for it! Copy all the animation data from Listing 3.168 into the gCharacters table.

```
-- code omitted
gCharacters =
{
```

```

hero =
{
    entity = "hero",
    actorId = "hero",
    anims =
    {
        up      = {1, 2, 3, 4},
        right   = {5, 6, 7, 8},
        down    = {9, 10, 11, 12},
        left    = {13, 14, 15, 16},
        prone   = {19, 20},
        attack  = {5, 4, 3, 2, 1},
        use     = {46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57},
        hurt    = {21, 22, 23, 24},
        standby = {36, 37, 38, 39},
        advance = {36, 37, 38, 39},
        retreat = {61, 62, 63, 64},
        death   = {26, 27, 28, 29},
        victory = {6, 7, 8, 9}
    },
    -- code omitted
},
thief =
{
    entity = "thief",
    actorId = "thief",
    anims =
    {
        up      = {33, 34, 35, 36},
        right   = {37, 38, 39, 40},
        down    = {41, 42, 43, 44},
        left    = {45, 46, 47, 48},
        prone   = {9, 10},
        attack  = {1, 2, 3, 4, 5, 6, 7, 8},
        use     = {11, 12, 13, 14, 15, 16, 17, 18, 19, 20},
        hurt    = {21, 22, 23, 24, 25, 33, 34},
        standby = {36, 37, 38, 39},
        advance = {36, 37, 38, 39},
        retreat = {61, 62, 63, 64},
        death   = {26, 27, 28, 29, 30, 31, 32},
        victory = {56, 57, 58, 59, 60, 40}
    },
    -- code omitted
},
mage =

```

```

{
    entity = "mage",
    actorId = "mage",
    anims =
    {
        up = {17, 18, 19, 20},
        right = {21, 22, 23, 24},
        down = {25, 26, 27, 28},
        left = {29, 30, 31, 32},
        prone = {51, 52},
        attack = {1, 2, 3, 4, 5, 6, 7},
        use = {41, 42, 43, 44, 45, 46, 47, 48},
        hurt = {8, 9, 10, 21, 22, 23},
        standby = {36, 37, 38, 39},
        advance = {36, 37, 38, 39},
        retreat = {61, 62, 63, 64},
        death = {26, 27, 28, 29, 30, 31, 32, 33, 34},
        victory = {56, 57, 58, 59, 60, 53, 54, 55, 49, 50, 40, 35},
    },
    -- code omitted
}
}

```

Listing 3.168: Adding the additional animations. In EntityDefs.lua.

In Listing 3.168 you can see that each party member has unique animations and that these are often of different lengths. This updated animation data is used when running combat animations from the combat states.

The Standby State

The standby state is the default combat state. It's active when the actor is waiting for a command. This state makes the character face their opponent and runs a standby animation, if one exists.

To get the standby animation, let's add a new function, `GetCombatAnim`, in `Character.lua`. For a given animation id, it returns a list of animation frames. Copy the code from Listing 3.169.

```

function Character:GetCombatAnim(id)

    if self.mAnims and self.mAnims[id] then
        return self.mAnims[id]
    else

```

```

        return { self.mEntity.mStartFrame }
    end
end

```

Listing 3.169: Gets the combat animation or returns a default still frame. In Character.lua.

In Listing 3.169 we check if the character has a `mAnims` table and if it contains an entry for the animation id. If there's no animation information, a list containing the entity start frame is returned. If the animation exists, then the animation set is returned. The enemy character has no animation data, so they always use their start frame for the standby animation.

Implementing CSStandby

With the Character.lua file modified, we're ready to fill in our first state. Open your CSStandby state and add the code from Listing 3.170.

```

CSStandby = { mName = "cs_standby" }
CSStandby.__index = CSStandby
function CSStandby:Create(character, context)
    local this =
    {
        mCharacter = character,
        mCombatScene = context,
        mEntity = character.mEntity,
        mAnim = nil,
    }

    setmetatable(this, self)
    return this
end

function CSStandby:Enter()
    local frames = self.mCharacter:GetCombatAnim('standby')
    self.mAnim = Animation>Create(frames)
end

function CSStandby:Exit()
end

function CSStandby:Update(dt)
    self.mAnim:Update(dt)
    self.mEntity:SetFrame(self.mAnim:Frame())

```

```

end

function CSStandby:Render(renderer)
    -- The combat state will do the render for us
end

```

Listing 3.170: Implementing the CSStandby state. In CSStandby.lua.

In CSStandby constructor, we've added an `mEntity` field to store the entity and an `mAnim` field set to nil. We'll play the standby animation on the stored entity.

The `Enter` function calls `Character.GetCombatAnim` to get the animation frames. These frames are used to initialize an `Animation` object, and this is stored in `self.mAnim`.

The `Update` function updates the animation and sets the entity to the current frame. The `Render` function does nothing, as rendering is handled by the `CombatState`.

Implementing CSRunAnim

`CSStandby` is the default state all characters start in. It just runs an animation, but because it's the default state it exists as a separate class. For other simple animated states, we'll use `CSRunAnim`. Let's implement it now. Copy the code from Listing 3.171.

```

CSRunAnim = { mName = "cs_run_anim" }
CSRunAnim.__index = CSRunAnim
function CSRunAnim>Create(character, context)
    local this =
    {
        mCharacter = character,
        mCombatScene = context,
        mEntity = character.mEntity,
    }

    setmetatable(this, self)
    return this
end

function CSRunAnim:Enter(params)
    local anim, loop, spf = unpack(params)
    self.mAnimId = anim
    local frames = self.mCharacter:GetCombatAnim(anim)
    self.mAnim = Animation>Create(frames, loop, spf)
end

function CSRunAnim:Exit()

```

```

end

function CSRunAnim:Update(dt)
    self.mAnim:Update(dt)
    self.mEntity:SetFrame(self.mAnim:Frame())
end

function CSRunAnim:Render(renderer)
end

function CSRunAnim:IsFinished()
    return self.mAnim:IsFinished()
end

```

Listing 3.171: Implementing the Run Animation state. In CSRunAnim.lua.

The CSRunAnim is similar to the CSStandby but the CSRunAnim state takes a params table in its Enter function. The params table describes the animation to be run.

Registering the CombatStates

Let's add the combat states to the gCharacterStates table. This lets us easily associate the states with the combat characters. Copy the code from Listing 3.172.

```

gCharacterStates =
{
    -- code omitted

    follow_path = FollowPathState,
    cs_run_anim = CSRunAnim,
    cs_hurt = CSHurt,
    cs_move = CSMove,
    cs_standby = CSStandby,
}

```

Listing 3.172: All the combat states added to gCharacterStates. In EntityDefs.lua.

All characters in combat use these states. Copy the code from Listing 3.173 to link the states to the characters.

```

gCharacters =
{
    hero =

```

```
{  
    -- code omitted  
    controller =  
    {  
        "wait",  
        "move",  
        "cs_run_anim",  
        "cs_hurt",  
        "cs_move",  
        "cs_standby",  
    },  
    state = "wait",  
},  
thief =  
{  
    -- code omitted  
    controller =  
    {  
        "npc_stand",  
        "cs_run_anim",  
        "cs_hurt",  
        "cs_move",  
        "cs_standby",  
    },  
    state = "npc_stand",  
},  
mage =  
{  
    -- code omitted  
    controller =  
    {  
        "npc_stand",  
        "cs_run_anim",  
        "cs_hurt",  
        "cs_move",  
        "cs_standby",  
    },  
    state = "npc_stand",  
},  
goblin =  
{  
    entity = "goblin",  
    controller =  
    {  
        "cs_move",  
    },  
}
```

```

    "cs_run_anim",
    "cs_hurt",
    "cs_standby",
},
state = "cs_standby",
}
}
}

```

Listing 3.173: Adding the combat states to the character definitions. In EntityDefs.lua.

The code to add the states is pretty much just data entry. The goblin is missing CSMove because enemies don't move on the battlefield.

By default, all party members start in the npc_stand state. On entering combat, we'll change to the standby state. To do this, we'll update the CreateCombatCharacters function as shown in Listing 3.174.

```

function CombatState:CreateCombatCharacters(side)

    -- code omitted

    for k, v in ipairs(actorList) do

        -- code omitted

        char.mEntity.mX = x
        char.mEntity.mY = y

        -- Change to standby because it's combat time
        char.mController:Change(CSStandby.mName)

    end

end

```

Listing 3.174: Changing to standby when combat characters are created. In CombatState.lua.

In Listing 3.174 each combat character has its state set to standby when created. Next let's update the CombatState:Update function, to update each character during combat. Copy the code from Listing 3.175.

```

function CombatState:Update(dt)
    for k, v in ipairs(self.mCharacters['party']) do

```

```

    v.mController:Update(dt)
end

for k, v in ipairs(self.mCharacters[ 'enemy' ]) do
    v.mController:Update(dt)
end
end

```

Listing 3.175: Update the character state machines in the combat state. In CombatState.lua.

The Update function loops through the characters and updates their state machines. Run the code, and you'll see the characters facing the correct way and running on the spot as shown in Figure 3.39. The characters running demonstrates that the standby animation and state is working correctly.

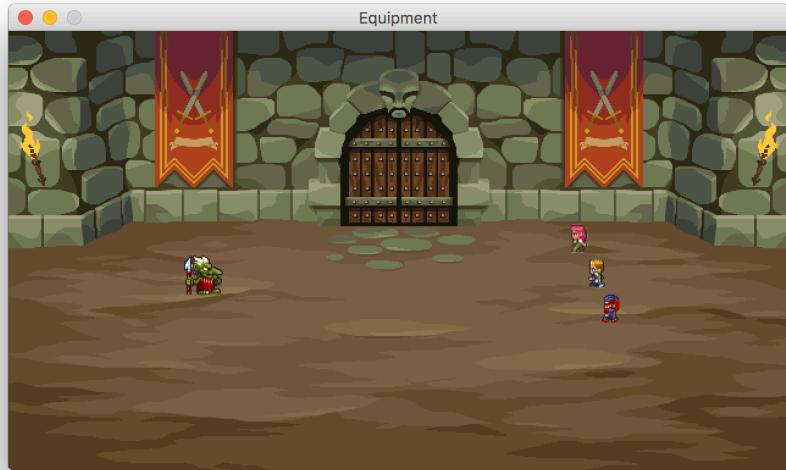


Figure 3.39: Monsters and party members on the battlefield.

All the code so far is available in combat-2-solution.

Screen Layout

The combat is becoming more and more fleshed out, but to help the player read how the battle is going we need some UI. Example combat-3 contains the code so far, or

you can continue to use your own codebase.

The main UI element is a panel that sits at the bottom of the screen. This panel shows HP and MP details for the party. Slightly above the panel, there's a tooltip bar that tells the player what's happening during combat. When not in use, the tooltip bar is hidden. HP and MP info for the monster are not displayed; these are secret.

Let's add the new UI directly into the CombatState class. We'll begin by creating the panels and adding the party member names. Once we have that working, we'll add the health and magic point progress bars. Copy the code from Listing 3.176 into the CombatState constructor.

```
CombatState.__index = CombatState
function CombatState:Create(stack, def)

    local this =
    {
        -- code omitted

        mSelectedActor = nil,
        mActorCharMap = {},

        mPanels = {},
        mTipPanel = nil,
        mNoticePanel = nil,
        mPanelTitles = {},
        mPartyList = nil,
        mStatsYCol = 208,
    }

    -- Setup layout panel
    local layout = Layout>Create()
    layout:SplitHorz('screen', 'top', 'bottom', 0.72, 0)
    layout:SplitHorz('top', 'notice', 'top', 0.25, 0)
    layout:Contract('notice', 75, 25)
    layout:SplitHorz('bottom', 'tip', 'bottom', 0.24, 0)
    layout:SplitVert('bottom', 'left', 'right', 0.67, 0)
    this.mPanels =
    {
        layout>CreatePanel('left'),
        layout>CreatePanel('right'),
    }
    this.mTipPanel = layout>CreatePanel('tip')
    this.mNoticePanel = layout>CreatePanel('notice')

    this.mBackground:SetTexture(Texture.Find(def.background))
```

Listing 3.176: Extending the CombatState constructor. In CombatState.lua.

We've added an `mPanels` table to the `this` table in the `CombatState` constructor. The `mPanels` table stores all the backing panels for the combat UI. The panel layout is shown in Figure 3.40.

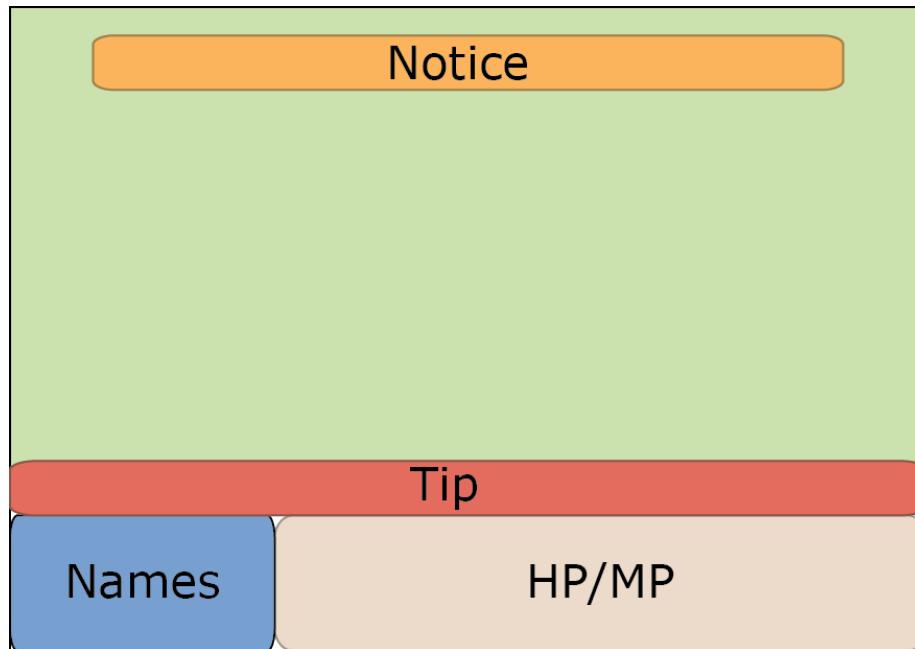


Figure 3.40: The combat panels.

The tip and notice panels are normally hidden and stored in separate fields to make turning them on and off easy. The tip panel displays tooltips to the user. The notice panel announces spells or special actions during combat. The notice panel sits near the top of the screen, center aligned. The left panel displays the party member names. The right panel shows health and mana.

The `mPanelTitles` field contains the title text for the names, the HP, the MP, and the locations to render the titles. The titles are rendered at the top of the left and right panels in capital letters. The `mPartyList` field is a Selection menu containing the party member names.

The `mStatsYCol` holds the distance between the HP and MP columns in the right panel.

Now that the fields are defined, we're ready to initialize them. The panels are created using the `Layout` class. We define the bottom 27% of the screen as where we'll render the UI. Then we break that into two pieces: the top part is the tip panel, usually hidden,

and the bottom part is for the names and stats. The bottom panel is broken into left and right pieces, named "left" and "right".

Once the layout is defined, the panels are created, the left and right panels are added to the `mPanels` table, and the tip panel is assigned to the `mTipPanel` field.

Next we'll create the selection menu to display the party member names. Copy the code from Listing 3.177 into the constructor.

```
-- code omitted

setmetatable(this, self)

-- Need to change actors in to characters
this>CreateCombatCharacters('party')
this>CreateCombatCharacters('enemy')

-- Set up player list
this.mPartyList = Selection>Create
{
    data = this.mActors.party,
    columns = 1,
    spacingX = 0,
    spacingY = 19,
    rows = #this.mActors.party,
    renderItem =
        function(self, renderer, x, y, item)
            this:RenderPartyNames(renderer, x, y, item)
        end,
    OnSelection = this.OnPartyMemberSelect,
}

local x = -System.ScreenWidth() / 2
local y = layout:Top('left')

this.mPanelTitles =
{
    {
        text = 'NAME',
        x = x + 32,
        y = y - 9
    },
    {
        text = 'HP',
        x = layout:Left('right') + 24,
```

```

        y = y - 9
    },
{
    text = 'MP',
    x = layout:Left('right') + 24 + this.mStatsYCol,
    y = y - 9
}
}

y = y - 25 -- padding
this.mPartyList:SetPosition(x, y)
this.mPartyList:HideCursor()

```

Listing 3.177: Continuing to extend the CombatState constructor. In CombatState.lua.

In Listing 3.177 we create a selection menu and store it in `mPartyList`. The data source for the menu is the list of party actors. The menu uses the `RenderPartyNames` and `OnPartyMemberSelect` functions (which we've yet to write) to draw the names and to handle selecting one of the party members. The `RenderPartyNames` function is wrapped in another function so we can easily access the `this` table.

At this stage in the constructor, we calculate the positions for some of the UI elements. The top left position of the panel is calculated and stored in variables `x` and `y`. We use the `x` and `y` to help position all `mPanelTitles` and the `mPartyList` menu. The `NAME` title is positioned above the `mPartyList` selection. The `HP` and `MP` titles are positioned at the top of the right panel. Finally the `mPartyList` cursor is hidden, making the names a simple list.

Next let's implement `RenderPartyNames`. Copy the code from Listing 3.178. We'll also add an empty `OnPartyMemberSelect` function so the code won't crash.

```

function CombatState:RenderPartyNames(renderer, x, y, item)

    local nameColor = Vector.Create(1,1,1,1)

    if self.mSelectedActor == item then
        nameColor = Vector.Create(1,1,0,1)
    end

    renderer:DrawText2d(x, y, item.mName, nameColor)
end

function CombatState:OnPartyMemberSelect(index, data)
end

```

Listing 3.178: Rendering out the names of the party members. In CombatState.lua.

The RenderPartyNames function renders each actor name, coloring it yellow if it's the currently selected actor. Now we can finish implementing the Render to draw out the panels and actor names. Copy the code from Listing 3.179!

```
function CombatState:Render(renderer)

    -- code omitted

    for k, v in ipairs(self.mPanels) do
        v:Render(renderer)
    end
    --self.mTipPanel:Render(renderer)
    --self.mNoticePanel:Render(renderer)

    renderer:ScaleText(0.88, 0.88)
    renderer:AlignText("left", "center")
    for k, v in ipairs(self.mPanelTitles) do
        renderer:DrawText2d(v.x, v.y, v.text)
    end

    renderer:ScaleText(1.25, 1.25)
    self.mPartyList:Render(renderer)

end
```

Listing 3.179: Rendering out the panels and party names. In CombatState.lua.

At the end of the Render function, we draw all the panels except for the tip and notice panels. Then we render the titles with the font scaled down a little. For the names, the font is scaled up and aligned to the left, and mPartyList menu is drawn in the left panel.

Run the code and you'll see something like Figure 3.41.



Figure 3.41: Rendering the party names.

To finish the CombatState UI, we need to render the HP and MP values in the right panel. We'll use progress bars to display this data. Let's revisit the constructor and add a few more fields. Copy the code from Listing 3.180.

```
CombatState.__index = CombatState
function CombatState:Create(stack, def)

    local this =
    {
        -- code omitted

        mBars = {},
        mStatList = nil
    }

    -- code omitted

    this.mPartyList:SetPosition(x, y)
    this.mPartyList:HideCursor()

    for k, v in ipairs(this.mActors.party) do
```

```

local stats = v.mStats

local hpBar = ProgressBar:Create
{
    background = Texture.Find('hpbackground.png'),
    foreground = Texture.Find('hpforeground.png'),
    value = stats:Get('hp_now'),
    maximum = stats:Get('hp_max')
}

local mpBar = ProgressBar:Create
{
    background = Texture.Find('mpbackground.png'),
    foreground = Texture.Find('mpforeground.png'),
    value = stats:Get('mp_now'),
    maximum = stats:Get('mp_max')
}

this.mBars[v] =
{
    mHP = hpBar,
    mMP = mpBar
}

end

this.mStatList = Selection:Create
{
    data = this.mActors.party,
    columns = 1,
    spacingX = 0,
    spacingY = 19,
    rows = #this.mActors.party,
    RenderItem =
        function(self, renderer, x, y, item)
            this.RenderPartyStats(this, renderer, x, y, item)
        end,
    OnSelection = this.OnPartyMemberSelect,
}
x = layout:Left('right') - 8
this.mStatList:SetPosition(x, y)
this.mStatList:HideCursor()

return this
end

```

Listing 3.180: Adding more data to the combat state constructor. In CombatState.lua.

In the right panel, we draw a progress bar for each party member's HP and MP. The HP and MP values are also rendered in text. Two fields have been added to the constructor, `mBars` and `mStatList`. The `mBars` list stores the actors as keys, and it stores the HP and MP progress bars as values. The `mStatList` is a selection menu that renders out the HP and MP values as text.

In the constructor body, we fill up the `mBars` table and create `mStatList` menu. For each actor we create two progress bars, one for the HP and one for the MP. The bars are stored in the `mBars` table under `mHP` and `mMP` using the actor as the key. The progress bars use the textures we previously added to the project.

Once the `mBars` setup is done, we create the `mStatList` menu. It uses a new function, `RenderPartyStats`. This function is wrapped in another so that the `this` table can be passed in as the first parameter. The menu's `x` position is adjusted to add a little padding, and the cursor is hidden.

Let's define the `RenderPartyStats` function so we can draw the `mStatList` menu. Copy the code from Listing 3.181.

```
function CombatState:DrawHP(renderer, x, y, hp, max)

    local hpColor = Vector.Create(1,1,1,1)

    local percent = hp / max

    if percent < 0.2 then
        hpColor = Vector.Create(1,0,0,1)
    elseif percent < 0.45 then
        hpColor = Vector.Create(1,1,0,1)
    end

    local xPos = x
    local hp = string.format('%d', hp)
    renderer:DrawText2d(xPos, y, hp, hpColor)
    local size = renderer:MeasureText(hp)
    xPos = xPos + size:X() + 3
    renderer:DrawText2d(xPos, y, '/')
    size = renderer:MeasureText('/')
    xPos = xPos + size:X() - 1
    renderer:DrawText2d(xPos, y, max)

end

function CombatState:RenderPartyStats(renderer, x, y, item)
    local stats = item.mStats
```

```

local barOffset = 130

local bars = self.mBars[item]

self:DrawHP(renderer, x, y,
            stats:Get('hp_now'),
            stats:Get('hp_max'))

bars.mHP:SetPosition(x + barOffset, y)
bars.mHP:SetValue(stats:Get('hp_now'))
bars.mHP:Render(renderer)

x = x + self.mStatsYCol

local mpStr = string.format('%d', stats:Get('mp_now'))
renderer:DrawText2d(x, y, mpStr)
bars.mMP:SetPosition(x + barOffset * 0.7, y)
bars.mMP:SetValue(stats:Get('mp_now'))
bars.mMP:Render(renderer)
end

```

Listing 3.181: Rendering out the UI for the combat state. In CombatState.lua.

In Listing 3.181 before implementing RenderPartyStats we first define the helper function DrawHP. DrawHP draws out the HP values for an actor as text, e.g. 100/100. It's a separate function because it optionally colors first part of the text. If the player is under 20% health, the text is colored red; under 45% it's colored yellow; otherwise it's white. To do the coloring, we break the text into three pieces: the number before the text, the forward slash, and the number after the text. The text is drawn from left to right.

Next we define RenderPartyStats, the callback that draws the mStatList menu entries. The item parameter is an Actor and we store its stats value in a stats variable. We use stats to draw the text values, followed by the progress bar. The barOffset variable describes the distance between the start of the text and the start of the progress bar.

We use the item as a key to get the progress bars from the mBars table. DrawHP is used to draw the HP text by passing in hp_now and hp_max values from the stats. Then we position the HP progress bar to the right of the text, set its value, and draw it out. On the same row we draw out the MP text and progress bar. The MP text is a little different. Instead of 50/100 we'll just write 50, as the maximum MP information isn't as important.

With those changes made, we're ready to see our new combat UI in action. Run the code and you should see an image like Figure 3.42. The code so far is available in combat-3-solution.



Figure 3.42: The in game combat UI.

Everything is looking good. Time to add some actions! In the next section, we'll pull the EventQueue back into the main codebase and add events for basic combat actions.

Combat Actions

It's high time to get the combatants hitting each other! In this section we pull the EventQueue into the CombatState, create some actions, and start on the combat simulation.

The code so far is available in action-1. The EventQueue has been added and linked into the manifest and dependencies.

EventQueue

The EventQueue drives the combat. Let's begin by adding the EventQueue to the CombatState. Copy the code from Listing 3.182.

```
function CombatState:Create(stack, def)
    local this =
```

```

{
    mGameStack = stack,
    mStack = StateStack>Create(),
    -- code omitted

    mEventQueue = EventQueue>Create()
}

```

Listing 3.182: Adding the EventQueue to the CombatState constructor. In CombatState.lua.

In the CombatState constructor we create an EventQueue in the this table.

We update the event queue by calling its Update function at the end of CombatState.Update. Copy the code from Listing 3.183 to modify CombatState.Update.

```

function CombatState:Update(dt)

    -- code omitted

    if self.mStack:Top() ~= nil then

        self.mStack:Update(dt)

    else
        self.mEventQueue:Update()

        self:AddTurns(self.mActors.enemy)
        self:AddTurns(self.mActors.party)

        if self:PartyWins() then
            self.mEventQueue:Clear()
            -- deal with win
        elseif self:EnemyWins() then
            self.mEventQueue:Clear()
            -- deal with lost
        end
    end

end

```

Listing 3.183: Updating the EventQueue. In CombatState.lua.

The `mEventQueue` is only updated when `mStack` is empty. The stack contains any sub-states being run by the combat state, such as: character animations, spells, the player browsing a menu, and that type of thing.

The event queue controls and orders all events for the characters on the battlefield. Every character **must** have an event in the queue. If a character doesn't have an event in the queue, we give them a *take turn* event that tells them to decide what to do. After the `EventQueue` is updated, the `AddTurns` function searches out characters without an event and gives them a `CETurn` event which is pushed into the queue.

Before the end of the function, we check if the party or enemy side has won and handle that. Listing 3.183 uses many functions we've yet to define: `PartyWins`, `EnemyWins`, `AddTurns`, and the `CETurn` class. We need to implement all of these functions before the code can run.

Before adding the missing functions, let's add a render call for the combat state's internal stack. This lets us render combat substates. Copy the code from Listing 3.184.

```
function CombatState:Render(renderer)

    -- code omitted

    renderer:ScaleText(1.25, 1.25)
    renderer:AlignText("left", "center")
    self.mPartyList:Render(renderer)
    self.mStatList:Render(renderer)

    self.mStack:Render(renderer)
end
```

Listing 3.184: Rendering the stack from the CombatState. In CombatState.lua.

Listing 3.184 calls `render` on the combat state's internal stack.

Copy the code from Listing 3.185 to implement the missing `AddTurns` function. This function is nearly unchanged from our previous project. We've added an HP check. This handles the case when a party member is knocked out and at zero health, but still exists in the actor list. Knocked-out characters shouldn't perform combat actions and don't need a `CETurn` event added to the queue.

```
function CombatState:AddTurns(actorList)
    for _, v in ipairs(actorList) do

        local alive = v.mStats:Get("hp_now") > 0

        if alive and not self.mEventQueue:ActorHasEvent(v) then
```

```

        local event = CETurn>Create(self, v)
        local tp = event:TimePoints(self.mEventQueue)
        self.mEventQueue:Add(event)
    end

end
end

```

Listing 3.185: AddTurn function for a list of actors. In CombatState.lua.

The next two missing functions to implement are PartyWins and EnemyWins. Copy the code from Listing 3.186.

```

function CombatState:HasLiveActors(actorList)

    for _, actor in ipairs(actorList) do
        local stats = actor.mStats
        if stats:Get("hp_now") > 0 then
            return true
        end
    end

    return false
end

function CombatState:EnemyWins()

    return not self:HasLiveActors(self.mActors.party)
end

function CombatState:PartyWins()

    return not self:HasLiveActors(self.mActors.enemy)
end

```

Listing 3.186: Utility functions to see if combat has been won or lost. In CombatState.lua.

In Listing 3.186 we've added an additional helper function, HasLiveActors. It takes a list of actors and returns true if one or more actors are alive. Otherwise it returns false. HasLiveActors also returns false when passed an empty list. We use HasLiveActors to power win checks for both the enemy and the party. EnemyWins is true when all party actors are dead and PartyWins is true when all enemy actors are dead.

So far so good, but our code doesn't run yet. We need to define the CETurn class.

CETurn

We're getting to the point where it's useful to add structure to the project folder. In the code folder, create a subfolder called **combat_events**. Then in this folder add a new file called CETurn.lua. Add the file to the dependencies and manifest files. In the manifest, remember to point the path to the new subdirectory.

Copy the CETurn code from Listing 3.187.

```
CETurn = {}
CETurn.__index = CETurn
function CETurn:Create(state, owner)
    local this =
    {
        mState = state,
        mOwner = owner,
        mIsFinished = false
    }

    this.mName = string.format("Turn for \\"..this.mOwner.mName..") 

    setmetatable(this, self)
    return this
end

function CETurn:TimePoints(queue)
    local speed = self.mOwner.mStats:Get("speed")
    return queue:SpeedToTimePoints(speed)
end

function CETurn:Execute(queue)

    if self.mState:IsPartyMember(self.mOwner) then
        local state = CombatChoiceState:Create(self.mState, self.mOwner)
        self.mState.mStack:Push(state)
        self.mIsFinished = true
        return
    else
        -- 2. Am I an enemy
        -- Skip turn, we'll add AI later
        self.mIsFinished = true
        return
    end
end
```

```

end

function CETurn:Update()
end

function CETurn:IsFinished()
    return self.mIsFinished
end

```

Listing 3.187: Creating the turn combat event. In CETurn.lua.

The majority of the CETurn code is the same as in the previous section. The TimePoints function gets the speed information using the actor stats. There's an mIsFinished flag to report if the event is finished. The mName field is used to make debugging easier. We can use the name when printing out all the events in the queue.

The behavior of the Execute function changes depending on whether the actor is from the party or enemy side. All the party members are assumed to be player controlled. We deal with the enemy behavior later. For now enemies skip their turn to keep the combat flowing. Once we have the combat loop functional, we'll add basic AI.

In Listing 3.187 we used a new function, IsPartyMember. Copy the implementation from Listing 3.188.

```

function CombatState:IsPartyMember(actor)
    for _, v in ipairs(self.mActors.party) do
        if actor == v then
            return true
        end
    end
    return false
end

```

Listing 3.188: Adding a party member check to the Actor. In CombatState.lua.

The function in Listing 3.188 checks if the given actor is in the party. It returns true if it finds an actor and false otherwise.

Let's return to the CETurn:Execute function in Listing 3.187. Once we know we have a party member, we create a CombatChoiceState and push it onto the stack. The CombatChoiceState asks the player what they'd like the character to do. If you run the code, it executes but crashes on the first actor's turn because CombatChoiceState doesn't exist yet. Let's implement this state next.

CombatChoiceState

The combat choice state displays a dialog box of actions for the player to choose from. It's important for the player to know which character they're directing. When we enter the state, we'll highlight the current character name using yellow text. At the same time we use a Selection list to display all the actions the character can perform. The list displays up to three actions at a time, and supports scrolling if there are more. We won't use a scrollbar because space is quite tight. Instead we use small up and down arrows. The arrows only appear when the player can scroll.

While the CombatChoiceState is on the stack, the EventQueue isn't updated. No other combat events occur while the player is choosing an action⁵.

Let's begin by adding two utility functions to the Selection class. Copy the code from Listing 3.189.

```
function Selection:CanScrollUp()
    return self.mDisplayStart > 1
end

function Selection:CanScrollDown()
    return self.mDisplayStart <= (self.mMaxRows - self.mDisplayRows)
end
```

Listing 3.189: Adding Scroll checks to the Selection. In Selection.lua.

CanScrollUp and CanScrollDown return true if the Selection box contains more items than are currently shown. These functions tell us when to render the up and down arrows to indicate that there are more commands than the box can show.

Create a file called CombatChoiceState.lua and add it to the manifest and dependencies files. Copy the CombatChoiceState constructor from Listing 3.190.

```
CombatChoiceState = {}
CombatChoiceState.__index = CombatChoiceState
function CombatChoiceState:Create(context, actor)
    local this =
    {
        mStack = context.mStack,
        mCombatState = context,
        mActor = actor,
        mCharacter = context.mActorCharMap[actor]
        mUpArrow = gWorld.mIcons:Get('uparrow'),
```

⁵In the Final Fantasy series, the term *active time battle* describes a combat system where the player *can* be interrupted in the CombatChoiceState by another combat event. The system we're implementing here is more turn-based.

```

        mDownArrow = gWorld.mIcons:Get('downarrow'),
    }

setmetatable(this, self)

this.mSelection = Selection>Create
{
    data = this.mActor.mActions,
    columns = 1,
    displayRows = 3,
    spacingX = 0,
    spacingY = 19,
    OnSelection = function(...) this.OnSelect(...) end,
    RenderItem = this.RenderAction,
}

this>CreateChoiceDialog()

return this
end

function CombatChoiceState:SetArrowPosition()
    local x = self.mTextbox.mSize.left
    local y = self.mTextbox.mSize.top
    local width = self.mTextbox.mWidth
    local height = self.mTextbox.mHeight

    local arrowPad = 9
    local arrowX = x + width - arrowPad
    self.mUpArrow:SetPosition(arrowX, y - arrowPad)
    self.mDownArrow:SetPosition(arrowX, y - height + arrowPad)
end

function CombatChoiceState>CreateChoiceDialog()
    local x = -System.ScreenWidth()/2
    local y = -System.ScreenHeight()/2

    local height = this.mSelection:GetHeight() + 18
    local width = this.mSelection:GetWidth() + 16

    y = y + height + 16
    x = x + 100

    this.mTextbox = Textbox>Create
{

```

```

textScale = 1.2,
text = "",
size =
{
    left = x,
    right = x + width,
    top = y,
    bottom = y - height
},
textbounds =
{
    left = -20,
    right = -0,
    top = 0,
    bottom = pad/2
},
panelArgs =
{
    texture = Texture.Find("gradient_panel.png"),
    size = 3,
},
selectionMenu = this.mSelection,
stack = this.mStack,
}
end

```

Listing 3.190: Setting up the panel and selection menu for the CombatChoiceState. In CombatChoiceState.lua.

The constructor takes in two parameters: context, which is the CombatState, and actor, which is the party member we're instructing to take an action.

In the this table, we store a reference to the stack as mStack so we can pop and push states as needed. We hold the combat state in mCombatState, the selected actor in mActor, and character object associated with the actor in mCharacter. We look up the character object in the CombatState's mActorCharMap using the actor as the key. The character represents the actor on the screen.

Finally two sprites are created, an up arrow and a down arrow. These are used to help navigate the Selection list.

After the this table we set up the metatable, giving us access to the methods associated with the CombatChoiceState class.

The next chunk of constructor code creates the mSelection object to display the party member actions. The data comes from the mActor.mActions table. There is only one column of data, and only three rows are displayed at once. There's no spacing on the x

axis because there's only one column. The spacing on the y axis is set to 19 pixels. The OnSelection and RenderItem callbacks call OnSelect and RenderAction, which we'll implement soon. OnSelect is wrapped in another function to ensure that the self in the callback refers to the CombatChoiceState object.

At the end of the constructor we call CreateChoiceDialog. This creates the textbox that holds the action choices. To position the choice box, we first get the x and y of the bottom right of the screen. Then we work out the width and height of the Selection list and add some padding to those values. The textbox is positioned from the top left, so we add the height of the Selection list to the y plus a little padding. The x is moved 100 pixels to the right so it's not obscuring the party member names. The image in Figure 3.43 shows how the box is rendered.



Figure 3.43: Adding the Action Selection list with backing panel.

The SetArrowPositions function positions the arrow sprites at the top and bottom of the right side of the panel. The arrows positions are set each frame so they're in the correct position even if the panel changes size during its in-transition.

We store the action list in mTextbox.

CombatChoiceState Methods

Next let's fill out the remaining CombatChoiceState functions to get it working. Copy the code from Listing 3.191 into your project. These methods are all quite brief.

```

function CombatChoiceState:RenderAction(renderer, x, y, item)
    local text = Actor.ActionLabels[item] or ""
    renderer:DrawText2d(x, y, text)
end

function CombatChoiceState:OnSelect(index, data)
    -- Needs implementing.
end

function CombatChoiceState:Enter()
    self.mCombatState.mSelectedActor = self.mActor
end

function CombatChoiceState:Exit()
    self.mCombatState.mSelectedActor = nil
end

function CombatChoiceState:Update(dt)
    self.mTextbox:Update(dt)
end

function CombatChoiceState:Render(renderer)
    self.mTextbox:Render(renderer)

    self:SetArrowPosition()
    if self.mSelection:CanScrollUp() then
        renderer:DrawSprite(self.mUpArrow)
    end
    if self.mSelection:CanScrollDown() then
        renderer:DrawSprite(self.mDownArrow)
    end
end

function CombatChoiceState:HandleInput()
    self.mSelection:HandleInput()
end

```

Listing 3.191: Adding methods for the CombatChoiceState. In CombatChoiceState.lua.

First up is the `RenderAction` callback for the selection list. This function is responsible for drawing each choice the player can pick from. The `item` parameter is an action id. We transform the id into text using the `Actor.ActionLabels` table. If the action id or text is empty, we draw an empty string. Text is drawn to the screen at the `x, y` position passed in.

OnSelect is called when the player chooses an item from the choice list. This function is empty for now.

The Enter and Exit functions assign and clear the mSelectedActor field in the CombatState. When we enter a CombatChoiceState we're telling one of the party members what to do. This mSelectedActor holds the actor we're ordering about. The CombatState highlights the mSelectedActor name in the UI and unhighlights it once we've picked an action. Look at Figure 3.43 and you can see Ermis is currently selected and her name is highlighted in yellow.

Update tells the textbox update so it can run its enter and exit transitions.

Render draws the textbox, which also draws the Selection list. The up and down arrows are drawn if the selection list can be scrolled up or down respectively.

Finally the HandleInput function tells the mSelection list to handle any input. This lets the player scroll through the actions.

Run the code and the action menu now appears and is browseable. To test scrolling and the up and down arrows, you need to make a temporary change in the constructor as shown in Listing 3.192.

```
data = this.mActor.mActions,  
to  
data = {"attack", "item", "attack", "item", "attack"},
```

Listing 3.192: Adding extra actions. In the constructor of CombatChoiceState.lua.

Make these changes and run the code again, and you'll see the up and down arrows working as in Figure 3.44. Change the code back once you've finished testing.



Figure 3.44: Testing scrolling through the action choices.

Selection and Target Markers

The selected party member has a yellow name in the UI. To make it even clearer which character is selected, we'll add a marker above the character's head. We use the character sprite to determine the top of the character's head. Sprites may differ in size, so let's add a helper function in the Entity class to tell us where to place the marker. Copy the code from Listing 3.193.

```
function Entity:GetSelectPosition()

    local pos = self.mSprite:GetPosition()
    local height = self.mHeight

    local x = pos:X()
    local y = pos:Y() + height / 2

    local yPad = 16

    y = y + yPad

    return Vector.Create(x, y)
```

```
end
```

Listing 3.193: Adding a GetSelectedPosition function to the Entity. In Entity.lua.

GetSelectedPosition helps indicate which actor is currently selected.

The selected actor usually needs to target other actors on the battlefield. They might attack an enemy or use a health potion on a friend. When this happens, we need a way to highlight the targets of an action. We'll mark the targets with an arrow to the side of the sprite. This way it's clear which actor is performing the action and which actor is a target. This also allows the actor to target itself (for instance if they wanted to use a heal potion on themselves). Copy the helper function from Listing 3.194 to the target position for an entity.

```
function Entity:GetTargetPosition()

    local pos = self.mSprite:GetPosition()
    local width = self.mWidth

    local x = pos:X() + width / 2
    local y = pos:Y()

    return Vector.Create(x, y)

end
```

Listing 3.194: Adding a function to get the target marker position for an entity. In Entity.lua.

Let's use the GetSelectPosition function in CombatChoiceState to help highlight the active actor. Displaying the active actor is important information, so we'll bounce the marker up and down. Begin by updating the constructor to match the code in Listing 3.195.

```
CombatChoiceState = {}
CombatChoiceState.__index = CombatChoiceState
function CombatChoiceState:Create(context, actor)
    local this =
    {
        -- code omitted
        mDownArrow = gWorld.mIcons:Get('downarrow'),
        mMarker = Sprite.Create(),
    }
}
```

```
this.mMarker:SetTexture(Texture.Find('continue_caret.png'))
this.mMarkPos = this.mCharacter.mEntity:GetSelectPosition()
this.mTime = 0
```

Listing 3.195: Adding a marker to the CombatChoiceState constructor. In CombatChoiceState.lua.

In Listing 3.195 we create a sprite, `mMarker`, to represent the selected character marker. We'll reuse the "continue_caret" texture for the marker sprite; this a small arrow with the point facing down. We use the new `GetSelectPosition` function to store the marker position in `mMarkPos`. This position is just over the head of the character. The `mTime` variable is used to animate the arrow by recording how much time has passed. Copy the animation code that uses `mTime` from Listing 3.196.

```
function CombatChoiceState:Update(dt)
    -- code omitted

    self.mTime = self.mTime + dt
    local bounce = self.mMarkPos + Vector.Create(0, math.sin(self.mTime * 5))
    self.mMarker:SetPosition(bounce)

end

function CombatChoiceState:Render(renderer)
    -- code omitted

    renderer:DrawSprite(self.mMarker)
end
```

Listing 3.196: Drawing the selection arrow and making it bounce. In CombatChoiceState.lua.

In Listing 3.196 the Update loop increases `mTime` each frame by the delta time. We use `mTime` with `math.sin` to get a nice bouncy sine wave over time. The sine wave is used to move the marker sprite up and down 5 pixels; a simple bounce effect. The Render function renders the marker to the screen.

Run the code and you'll see the bouncing arrow. The complete code is available in [action-1-solution](#).

Next we'll add support for targeting actors on the battlefield using a new state, `CombatTargetState`.

CombatTargetState

The CombatTargetState state asks the player “Who do you want to attack?” A default target is selected. The player can change the selection using the arrows keys. The CombatChoiceState can be cancelled by pressing Backspace. The CombatTargetState is also used for spells, special abilities, and items. It has to support selecting multiple targets at a time.

The CombatTargetState can use three different modes.

- **One** - Targets a single character on the battlefield. A simple attack would use this.
- **Side** - Targets all characters on the enemy side or all characters on the player side. A heal-all spell would use this.
- **All** - Targets all characters on the battlefield. Summon meteors would use this.

The CombatTargetState also supports restrictions on targeting so *only* enemies may be targeted, or *only* party members.

For now, let’s concentrate on getting an attack action working. We can deal with more complicated targeting later.

There’s a project with the code so far available in action-2.

Modify CombatChoiceState to push CombatTargetState onto the stack when the player chooses the attack action, as shown in Listing 3.197.

```
function CombatChoiceState:OnSelect(index, data)

    if data == "attack" then
        self.mSelection:HideCursor()
        local state = CombatTargetState:Create(
            self.mCombatState,
            self,
            {
                onSelect = function(targets)
                    self:TakeAction(data, targets)
                end,
                onExit = function()
                    self.mSelection>ShowCursor()
                end
            })
        self.mStack:Push(state)
    end
end
```

Listing 3.197: Pushing on a select state when the actor is told to attack. In CombatChoiceState.lua.

The CombatChoiceState:OnSelect function pushes the CombatTargetState onto the stack when the “attack” action is selected. At this point we hide the selection cursor, so only the CombatTargetState cursor is on screen. The CombatTargetState constructor takes in the CombatState, CombatChoiceState, and a params table to let us adjust the targeting behavior.

Our attack action targets a single enemy. By default CombatTargetState selects the weakest enemy. In the params table there’s a callback for onSelect. This is called when the player confirms the target to attack. The onExit callback is called when the CombatTargetState exits and it’s safe to show the CombatChoiceState selection cursor again. When the target is selected we call a new function, TakeAction, which we’ve yet to implement. TakeAction adds the attack action to the EventQueue.

Create a CombatTargetState.lua file and add it to the manifest and dependencies files. Copy the code from Listing 3.198 into the file. This code contains some definitions and helper functions we’ll use when writing the class.

```
CombatSelector =
{
    WeakestEnemy = function(state)

        local enemyList = state.mActors["enemy"]
        local target = nil
        local health = 99999

        for k, v in ipairs(enemyList) do
            local hp = v.mStats:Get("hp_now")
            if hp < health then
                health = hp
                target = v
            end
        end

        return { target }
    end,

    SideEnemy = function(state)
        return state.mActors["enemy"]
    end,

    SelectAll = function(state)
        local targets = {}

        for k, v in ipairs(state.mActors["enemy"]) do
            table.insert(targets, v)
        end
    end
}
```

```

        for k, v in ipairs(state.mActors["party"]) do
            table.insert(targets, v)
        end

        return targets
    end
}

CombatTargetType =
{
    One = "One",
    Side = "Side",
    All = "All",
}

```

Listing 3.198: Definitions to help target characters. In CombatTargetState.lua.

We use CombatSelectorTable, in Listing 3.198, to decide which characters on the battlefield to target by default. Selectors are functions that take in the CombatState and return a table of actors. We use a default selector to guess who the player wants to target by default.

After these function definitions, there's the CombatTargetType table that defines three different types of selection. It's a very simple table where the key and value are the same.

- **WeakestEnemy** - targets the single weakest enemy on the enemy side. This returns a table of one actor.
- **SideEnemy** - targets all enemy actors on the enemy side.
- **SelectAll** - returns all the actors on the battlefield.

With the initial targeting defined, let's implement the CombatTargetState class. Copy the constructor code from Listing 3.199.

```

CombatTargetState = {}
CombatTargetState.__index = CombatTargetState
function CombatTargetState:Create(context, params)

    if params.switchSides == nil then
        params.switchSides = true
    end

    local this =

```

```

{
    mCombatState = context,
    mStack = context.mStack,
    mDefaultSelector = params.defaultSelector,
    mCanSwitchSide = params.switchSides,
    mTargetType = params.targetType or CombatTargetType.One,
    mOnSelect = params.onSelect,
    mOnExit = params.onExit or function() end,
    mTargets = {},
    mMarker = Sprite.Create(),
    mEnemy = {},
    mParty = {},
}

if this.mDefaultSelector == nil then

    if this.mTargetType == CombatTargetType.One then
        this.mDefaultSelector = CombatSelector.WeakestEnemy
    elseif this.mTargetType == CombatTargetType.Side then
        this.mDefaultSelector = CombatSelector.SideEnemy
    elseif this.mTargetType == CombatTargetType.All then
        this.mDefaultSelector = CombatSelector.SelectAll
    end

end

local markTexture = Texture.Find('cursor.png')
this.mMarkerWidth = markTexture:GetWidth()
this.mMarker:SetTexture(markTexture)
this.mMarker:SetUVs(1,1,0,0)

setmetatable(this, self)

return this
end

```

Listing 3.199: Creating the CombatTargetState class. In CombatTargetState.lua.

The constructor takes in two parameters: the context which is the CombatState and a params table. The params table contains an optional boolean field, switchSides, which defaults to true. If true, the player can switch the selection between the enemy and party sides. In our case, this means we'll be able to attack our own party and hit a teammate.

The this table contains the following fields.

- **mCombatState** - The CombatState object.
- **mStack** - The stack of states from the CombatState object.
- **mDefaultSelector** - The function that chooses which characters are targeted when the state begins.
- **mCanSwitchSide** - Flag that permits the player to target either the enemy or party members.
- **mTargetType** - The selection type: One, Side or All.
- **mOnSelect** - Callback when the player confirms the selection.
- **mOnExit** - Callback when the target state is exited. Defaults to an empty function.
- **mTargets** - The characters selected as targets.
- **mMarker** - Sprite used to indicate which targets are selected.
- **mEnemy** - List of the enemy actors.
- **mParty** - List of the party actors.

These fields are set up in the constructor and the Enter function.

In the constructor, immediately after the `this` table, the `mDefaultSelector` is set. If the `mDefaultSelector` is null, it uses a default value. The default depends on the selection type: if only one target can be selected, it defaults to `WeakestEnemy`; if a side can be selected, it defaults to `SideEnemy`; and if all can be selected, it can only be `SelectAll`.

At the end of the constructor, we set the `mMarker` to use the cursor texture. The marker highlights the characters being targeted. We change the U,Vs to mirror the cursor horizontally.

Let's add the Enter and Exit functions. Copy the code from Listing 3.200

```
function CombatTargetState:Enter(actor)

    self.mEnemy = self.mCombatState.mActors["enemy"]
    self.mParty = self.mCombatState.mActors["party"]

    self.mTargets = self.mDefaultSelector(self.mCombatState)

end

function CombatTargetState:Exit()

    self.mEnemy = {}
    self.mParty = {}
    self.mTargets = {}

    self.mOnExit()

end
```

Listing 3.200: Enter and Exit for the CombatTargetState. In CombatTargetState.lua.

The Enter function sets up the `mEnemy` and `mParty` lists. The `mTargets` is filled using `mDefaultSelector` on the combat state.

When the `CombatTargetState` is exited, all the fields set up in Enter are cleared. Exit also calls the `mOnExit` function setup in the constructor.

Copy the `Render` function from Listing 3.201.

```
function CombatTargetState:Render(renderer)
    for k, v in ipairs(self.mTargets) do
        local char = self.mCombatState.mActorCharMap[v]

        local pos = char.mEntity:GetTargetPosition()
        pos:SetX(pos:X() + self.mMarkerWidth/2)
        self.mMarker:SetPosition(pos)
        renderer:DrawSprite(self.mMarker)
    end
end

function CombatTargetState:Update(dt)
end
```

Listing 3.201: The render functions renders the targeting markers. In CombatTargetState.lua.

The `Render` function draw markers on each of the targets. It loops through the actors in the `mTargets` list. We use `mActorCharMap` to get the character associated with an actor. We get the target position from the character's entity and use it to position the selection marker. Then we draw the marker.

To allow the player to change the targets, we need to implement the `HandleInput` function. Copy the code from Listing 3.202.

```
function CombatTargetState:HandleInput()

    if Keyboard.JustPressed(KEY_BACKSPACE) then
        self.mStack:Pop()
    elseif Keyboard.JustPressed(KEY_UP) then
        self:Up()
    elseif Keyboard.JustPressed(KEY_DOWN) then
        self:Down()
    elseif Keyboard.JustPressed(KEY_LEFT) then
        self:Left()
    elseif Keyboard.JustPressed(KEY_RIGHT) then
```

```

        self:Right()
    elseif Keyboard.JustPressed(KEY_SPACE) then
        self.mOnSelect(self.mTargets)
    end
end

```

Listing 3.202: The HandleInput function for the selection state. In CombatTargetState.lua.

In Listing 3.202 if the Backspace key is pressed, we pop the CombatTargetState from the stack and return to the CombatChoiceState. When the Space key is pressed, we call the `mOnSelect` callback. Each arrow key calls a direction function that we'll implement shortly, but first we need to add some utility functions. Copy the code from Listing 3.203.

```

function CombatTargetState:GetActorList(actor)
    local isParty = self.mCombatState:IsPartyMember(actor)
    if isParty then
        return self.mParty
    else
        return self.mEnemy
    end
end

function CombatTargetState:GetIndex(list, item)
    for k, v in ipairs(list) do
        if v == item then
            return k
        end
    end
    return -1
end

```

Listing 3.203: Utility functions to make targeting actors easier. In CombatTargetState.lua.

`GetActorList` takes in an actor and returns the list of actors it belongs to, either the party members or the enemies. `GetIndex` gets the index of an item in a list. We use both of these function to help us navigate around the targets.

Add the direction commands next by copying the code from Listing 3.204.

```

function CombatTargetState:Left()
    if not self.mCanSwitchSide then

```

```

        return
    end

    if not self.mCombatState:IsPartyMember(self.mTargets[1]) then
        return
    end

    if self.mTargetType == CombatTargetType.One then
        self.mTargets = { self.mEnemy[1] }
    end

    if self.mTargetType == CombatTargetType.Side then
        self.mTargets = self.mEnemy
    end
end

function CombatTargetState:Right()
    if not self.mCanSwitchSide then
        return
    end

    if self.mCombatState:IsPartyMember(self.mTargets[1]) then
        return
    end

    if self.mTargetType == CombatTargetType.One then
        self.mTargets = { self.mParty[1] }
    end

    if self.mTargetType == CombatTargetType.Side then
        self.mTargets = self.mParty
    end
end

```

Listing 3.204: Left and Right selection. In CombatTargetState.lua.

The Left and Right functions switch focus between the two sides of the battlefield. If mCanSwitchSide is false, then Left and Right do nothing. Pressing Left only moves from the player to the enemy side; if the enemy side is already selected, Left does nothing. Pressing Right moves from the enemy to the player side, so if the player side is already selected it too does nothing.

If the selection can move and its mTargetType is set to "Side", it returns the opposite side. If its mTargetType is "One", then it selects the first index of the opposite side.

Copy the code for the Up and Down functions from Listing 3.205.

```

function CombatTargetState:Up()
    if self.mTargetType ~= CombatTargetType.One then
        return
    end

    local selected = self.mTargets[1]
    local side = self:GetActorList(selected)
    local index = self:GetIndex(side, selected)

    index = index - 1
    if index == 0 then
        index = #side
    end
    self.mTargets = { side[index] }

end

function CombatTargetState:Down()
    if self.mTargetType ~= CombatTargetType.One then
        return
    end

    local selected = self.mTargets[1]
    local side = self:GetActorList(selected)
    local index = self:GetIndex(side, selected)

    index = index + 1

    if index > #side then
        index = 1
    end

    self.mTargets = { side[index] }

end

```

Listing 3.205: Navigation functions for the combat targeting. In `CombatTargetState.lua`.

The Up and Down functions change the currently selected character. These functions are only used for `CombatTargetType.One` as it's the only mode that can select individual targets. You can see how the `CombatTargetState` appears in game by looking at Figure 3.45.



Figure 3.45: Using the `CombatTargetState` to select an enemy.

Run the code, and you'll be able to select Attack and then target the various characters on both sides of the battlefield. To try out the different selection modes, alter the code in the `CombatChoiceState:OnSelect` like in Listing 3.206.

```
-- Try select sides
local state = CombatTargetState:Create(
    combatState,
    self,
    {
        selectType = CombatTargetType.Side,
        onSelect = function(targets)
            self:TakeAction(data, targets)
        end
    })
-- Or try select all
local state = CombatTargetState:Create(
    combatState,
    self,
    {
        selectType = CombatTargetType.All,
        onSelect = function(targets)
```

```

        self:TakeAction(data, targets)
    end

})

```

Listing 3.206: Changes to demonstrate the different selection modes. In CombatTargetState.lua.

We're getting closer and closer to a full combat loop. The player is prompted to take actions, and when attack is chosen, the player can choose a target to attack. The next step is to create an attack event, push it onto the stack, and have it execute.

The complete code so far is available in action-2-solution.

Attack Event

Example action-3 is a fresh project that contains the code so far. Run the action-3 project and you can attack, but confirming the attack target causes a crash. This is because the TakeAction function in CombatChoiceState hasn't been written. Let's write that function.

The TakeAction function adds the attack event to the event queue. Copy the TakeAction code from Listing 3.207.

```

function CombatChoiceState:TakeAction(id, targets)
    self.mStack:Pop() -- target state
    self.mStack:Pop() -- choice state

    local queue = self.mCombatState.mEventQueue

    if id == "attack" then
        local def = {}
        local event = CEAttack>Create(self.mCombatState,
                                       self.mActor,
                                       def,
                                       targets)
        local tp = event:TimePoints(queue)
        queue:Add(event, tp)
    end
end

```

Listing 3.207: Letting the player take action. In CombatChoiceState.lua.

The TakeAction function pops the CombatTargetState and CombatChoiceState off the stack. This leaves the CombatState internal stack empty and causes the mEventQueue to start updating again.

At the moment, the only action we support is “attack”. When “attack” is chosen we create a CEAttack object using the CombatState, the Actor to perform the attack, an attack def, and the list of targets. The attack def is empty for now, but usually it contains extra data about the attack - if it’s poisonous or has an elemental effect for instance.

Before we push our attack event onto the stack, we get the time point cost by calling its TimePoints function. Then we add the event into the queue. The position in the queue depends on the actor’s speed and the time values of other events already in the queue.

CSAttack

For our character to perform an attack, we need to create some new states. When the character is given an attack event, they enter the prone state. When the event executes, they leave the prone state, walk forward, run an attack animation, run the attack calculation, apply the results, and walk back. This is a complicated sequence but we’ll break it down into simple steps.

We’re going to use CSMove to move the character toward (and later away from) the target. The attack animation is done with CSRunAnim. CSRunAnim plays the attack animation once without looping, which is exactly what we want.

Let’s start with CSMove. It’s similar to the MoveState class which is used to move the character around a tilemap. However, it doesn’t change the character’s tile position. Also, when moving backward, the character faces forward, which is important because you never want to turn your back on the enemy!

Copy the code from Listing 3.208.

```
CSMove = { mName = "cs_move" }
CSMove.__index = CSMove
function CSMove:Create(character, context)
    local this =
    {
        mCharacter = character,
        mEntity = character.mEntity,
        mTween = nil,
        mMoveTime = 0.3,
        mMoveDistance = 32,
    }

    setmetatable(this, self)
    return this
end
```

Listing 3.208: Creating a class to move actors in combat. In CSMove.lua.

In the CSMove constructor, we store the character and entity objects in the this table. We then add a field for a tween set to nil. This tween handles the character movement and its setup in the Enter function. The mMoveTime field is the number of seconds that the movement takes. The mMoveDistance is the number of pixels to move the character. The default numbers move the character 32 pixels in 0.3 seconds.

Next let's implement the CSMove:Enter function. Copy the code from Listing 3.209.

```
function CSMove:Enter(params)

    self.mMoveTime = params.time or self.mMoveTime
    self.mMoveDistance = params.distance or self.mMoveDistance

    local backforth = params.dir

    local anims = self.mCharacter.mAnims

    if anims == nil then
        anims =
        {
            advance = { self.mEntity.mStartFrame },
            retreat = { self.mEntity.mStartFrame },
        }
    end

    local frames = anims.advance
    local dir = -1
    if self.mCharacter.mFacing == "right" then
        frames = anims.retreat
        dir = 1
    end
    dir = dir * backforth

    self.mAnim = Animation>Create(frames)

    -- Store current position
    local pixelPos = self.mEntity.mSprite:GetPosition()
    self.mPixelX = pixelPos:X()
    self.mPixelY = pixelPos:Y()

    self.mTween = Tween>Create(0, dir, self.mMoveTime)

end
```

Listing 3.209: Setting up an actor's move during combat. In CSMove.lua.

The Enter function uses the params table to optionally override the mMoveTime and mMoveDistance values. Next it locally sets the backforth variable. This determines if the character is to move forward or backward. This is not an optional variable. Valid values are 1 for forward into the battlefield and -1 to move back to the starting position.

The Enter function also sets up the animations for the actor. Animations are taken from the mCharacter object if they exist. Monsters do not have animations. If no animations are present, we create a table with the required retreat and advance animation entries, each containing only the start frame.

To decide which animation to play, retreat or advance, we check which way the character is facing. The dir value determines the direction of the movement. We store the entity position in mPixelX and mPixelY; then we create the movement tween, mTween. The tween runs from 0 to dir (which is 1 or -1) in mMoveTime seconds.

Let's finish this state. Copy the code from Listing 3.210.

```
function CSMove:Exit()
end

function CSMove:Update(dt)
    self.mAnim:Update(dt)
    self.mEntity:SetFrame(self.mAnim:Frame())

    self.mTween:Update(dt)
    local value = self.mTween:Value()
    local x = self.mPixelX + (value * self.mMoveDistance)
    local y = self.mPixelY
    self.mEntity.mX = math.floor(x)
    self.mEntity.mY = math.floor(y)
    self.mEntity.mSprite:SetPosition(self.mEntity.mX, self.mEntity.mY)
end

function CSMove:Render(renderer)
end

function CSMove:IsFinished()
    return self.mTween:IsFinished()
end
```

Listing 3.210: Remaining combat movement functions. In CSMove.lua.

The Update function in Listing 3.210 updates the animation and applies it to the entity, the tween is updated, and the sprite is moved by the tween value multiplied by the distance. The X, Y values are floored to make the sprite move whole pixels.

The IsFinished function reports when the movement is done. This state is now ready to use as part of the attack sequence!

Storyboards for Combat Events

An attack is a sequence of actions:

1. Character walks forward
2. Character plays attack animation
3. Character walks back

We've already written code to handle just this type of situation – the Storyboard! All we need to do is create some new storyboard events to control the character in combat.

Let's begin by adding a powerful, generic operation – Function. When executed, the Function operation runs a function, then steps to the next storyboard operation. Copy the code from Listing 3.211.

```
function SOP.Function(func)
    return function(storyboard)
        func()
        return EmptyEvent
    end
end
```

Listing 3.211: Adding the Function operation. In StoryboardEvents.lua.

The Function operation takes in a function. When the operation is executed, the function is called and the EmptyEvent is returned.

The next operation to add is RunState. RunState changes a statemachine to a new state, then blocks the storyboard until that state's IsFinished function returns true.

RunState helps the characters perform actions during combat. States used by RunState such as CSMove and CSRanAnim must implement IsFinished.

Copy the code from Listing 3.212.

```
function SOP.RunState(statemachine, id, params)
    return function(storyboard)
        statemachine:Change(id, params)
        return BlockUntilEvent:Create(
            function()
                return statemachine.mCurrent:IsFinished()
            end
        )
    end
end
```

Listing 3.212: Adding the RunState operation. In StoryboardEvents.lua.

With these two new events, we're ready to use storyboards to control the attack action.

The Attack Event

Time to start pulling everything together. We can move characters, make them play animations, and control everything with a storyboard. Let's put these actions into the attack event!

In /code/combat_events create a file called CEAttack.lua. Add it to the manifest and dependencies. Copy the code from Listing 3.213.

```
CEAttack = {}
CEAttack.__index = CEAttack
function CEAttack:Create(state, owner, def, targets)

    local this =
    {
        mState = state,
        mOwner = owner,
        mDef = def,
        mIsFinished = false,
        mCharacter = state.mActorCharMap[owner],
        mTargets = targets,
    }

    this.mController = this.mCharacter.mController
    this.mController:Change(CSRunAnim.mName, {'prone'})
    this.mName = string.format("Attack for %s", this.mOwner.mName)

    setmetatable(this, self)

    local storyboard =
    {
        SOP.RunState(this.mController,
            CSMove.mName, {dir = 1}),
        SOP.RunState(this.mController,
            CSRunAnim.mName, {'attack', false}),
        SOP.Function(function() this:DoAttack() end),
        SOP.RunState(this.mController,
            CSMove.mName, {dir = -1}),
        SOP.Function(function() this:OnFinish() end)
    }

    this.mStoryboard = Storyboard>Create(this.mState.mStack,
        storyboard)

    return this
end
```

Listing 3.213: Introducing the combat attack event. In CEAttack.lua.

CEAttack takes in four parameters:

- **state** - the CombatState
- **owner** - the actor performing the attack
- **def** - information about the attack
- **targets** - a list of actors to hit

The this table stores the state, owner, and target parameters in `mState`, `mOwner`, and `mTargets`. There's an `mIsFinished` flag to indicate when the event has finished and can be removed from the EventQueue.

We use the actor to get the character object from `mActorCharMap` and store it in `mCharacter`.

After setting up the this table, we store the character controller in `mController`. Then we use the controller to set the character's state to "prone". The "prone" state plays an animation to indicate the character will perform an action as soon as it's their turn.

We give the event a name which tells us who is performing this attack. The name is used for debugging the event queue.

The metatable is linked and then we're ready to create the storyboard.

The storyboard is made up of five operations. The first operation runs the CSMove state, making the character walk toward the enemy. The second operation runs CSRunAnim to play the attack animation. Next we call the SOP.Function operation; this calls the `DoAttack` function, which performs all the attack calculations and handles any special effects. The character is moved back with another CSMove, and an SOP.Function operation calls `OnFinish`.

We store the storyboard in `mStoryboard`. Next let's write the CEAttack code to determine the attack's time points. Copy the code from Listing 3.214.

```
function CEAttack:TimePoints(queue)
    local speed = self.mOwner.mStats:Get("speed")
    return queue:SpeedToTimePoints(speed)
end

function CEAttack:OnFinish()
    self.mIsFinished = true
end

function CEAttack:IsFinished()
    return self.mIsFinished
end

function CEAttack:Update() end
```

Listing 3.214: Adding code to determine how long an attack action takes. In CEAttack.lua.

The TimePoints function uses the actor speed to determine how long the attack takes. If we were making a more complex RPG, this would be the place to take into account weapon heaviness or things of that nature.

The OnFinish function sets the mIsFinished flag to true. When this flag is true, the IsFinished function returns true and the CEAttack event is removed from the event queue on the next update.

The Update function is empty because the update is handled through the storyboard.

Let's add the Execute function next. Copy the code from Listing 3.215.

```
function CEAttack:Execute(queue)
    self.mState.mStack:Push(self.mStoryboard)
end
```

Listing 3.215: Execute function for the combat attack event. In CEAttack.lua.

Execute pushes the storyboard onto the top of the stack, which starts the attack event running. The mState here is the CombatState and the stack is its internal stack for combat events.

The Storyboard calls DoAttack so we need to implement this function. Copy the code from Listing 3.216. DoAttack is responsible for the attack calculation.

```
function CEAttack:DoAttack()
    for _, target in ipairs(self.mTargets) do
        self:AttackTarget(target)
    end
end

function CEAttack:AttackTarget(target)

    local stats = self.mOwner.mStats
    local enemyStats = target.mStats

    -- Simple attack get
    local attack = stats:Get("attack")
    attack = attack + stats:Get("strength")
    local defense = enemyStats:Get("defense")

    local damage = math.max(0, attack - defense)
    print("Attacked for ", damage, attack, defense)
```

```

local hp = enemyStats:Get("hp_now")
local hp = hp - damage

enemyStats:Set("hp_now", math.max(0, hp))
print("hp is", enemyStats:Get("hp_now"))

end

```

Listing 3.216: Adding calculation to the combat attack event. In CEAttack.lua.

The DoAttack function calls AttackTarget on each target in the mTargets list. The AttackTarget function is quite simple; we sum the *attack* and *strength* stats for the attacker and subtract the *defense* of the target. The total attack is subtracted from the target's HP. We'll be revisiting this function later to upgrade the attack simulation and to add special effects.

Run the code and you'll be able to attack. The attack animation plays and the enemy HP is reduced. But if the HP falls to zero, the enemy doesn't die. We'll deal with that next. The full code for this section is available in action-3-solution.

Death

When enemies die they're removed from the battlefield with an exit transition, while party members remain on the battlefield but in a KO'ed state. Death needs to be instant. Characters must not be able to attack targets that have died. All death exit-transitions should occur at the same time, i.e. if you kill three enemies in one attack they should *all* fade out at once.

We've done quite a lot in action-3, so let's start a fresh example project: action-4.

In our CEAttack class, the actor walks toward the enemy, inflicts damage, and moves back. If the actor kills an enemy, the death effect should play as the actor walks back. Death is not handled as an event in the EventQueue; it's handled separately so that it occurs in parallel with the attack event.

For the player deaths we'll use CSRunAnim. For enemy deaths we'll create a new state, CSEnemyDie.

CSEnemyDie

Create CSEnemyDie.lua and add it to the code/combat_state directory. Then add it to the manifest and dependencies files.

Modify the EntityDefs.lua files as shown in Listing 3.217 to add CSEnemyDie to the list of gCharacterStates.

```

gCharacterStates =
{
    -- code omitted
    cs_die_enemy = CSEnemyDie,
}

```

Listing 3.217: Adding combat death to the character states. In EntityDefs.lua.

In Listing 3.217 we've added CSEnemyDie under the id "cs_die_enemy". We'll add this id to the goblin definition. Update the goblin entity as shown in Listing 3.218.

```

goblin =
{
    entity = "goblin",
    controller =
    {
        "cs_run_anim",
        "cs_hurt",
        "cs_standby",
        "cs_die_enemy",
    }
}

```

Listing 3.218: Adding combat death to the goblin. In EntityDefs.lua.

Next let's implement the CSEnemyDie state to handle enemy death. Copy the code from Listing 3.219.

```

CSEnemyDie = { mName = "cs_die" }
CSEnemyDie.__index = CSEnemyDie
function CSEnemyDie:Create(character, context)
    local this =
    {
        mCharacter = character,
        mCombatScene = context,
        mSprite = character.mEntity.mSprite,
        mTween = nil,
        mFadeColour = Vector.Create(1,1,1,1)
    }

    setmetatable(this, self)
    return this
end

function CSEnemyDie:Enter()
    self.mTween = Tween:Create(1, 0, 1)

```

```

end

function CSEnemyDie:Exit() end
function CSEnemyDie:Render(renderer) end

function CSEnemyDie:Update(dt)
    self.mTween:Update(dt)
    local alpha = self.mTween:Value()
    self.mFadeColour:SetW(alpha)
    self.mSprite:SetColor(self.mFadeColour)
end

function CSEnemyDie:IsFinished()
    return self.mTween:IsFinished()
end

```

Listing 3.219: The start of the enemy death state. In CSEnemyDie.lua.

The enemy death state uses a tween to fade out the enemy sprite. In the OnEnter function, we create a tween from 1 to 0 over a second. The Update function updates the tween, sets the mFadeColor alpha, and applies it to the sprite. When the tween finishes, the state's IsFinished function returns true.

HandleDeath function

As we said earlier, death is handled instantly, separate to the EventQueue. To deal with death events, we'll add new code to the CombatState. The first function to add is HandleDeath which we'll call from the CEAttack state when a target is killed. Copy the code from Listing 3.220.

```

function CEAttack:AttackTarget(target)
    -- code omitted
    self.mState:HandleDeath()
end

```

Listing 3.220: Hooking up death to the attack event. In CEAttack.lua.

Copy the code from Listing 3.221 to implement the HandleDeath function.

```

function CombatState:HandleDeath()

    self:HandlePartyDeath()

```

```
    self:HandleEnemyDeath()  
  
end
```

Listing 3.221: Filling out the enemy side of the combat def. In CombatState.lua.

Death is different for enemies and players, so we use separate functions to deal with each case. We'll begin with HandlePartyDeath as that's a little simpler. Copy the code from Listing 3.222.

```
function CombatState:HandlePartyDeath()  
    -- Deal with the actors  
    for _, actor in ipairs(self.mActors['party']) do  
        local character = self.mActorCharMap[actor]  
        local controller = character.mController  
        local state = controller.mCurrent  
        local stats = actor.mStats  
  
        -- is the character already dead?  
        if state.mAnimId ~= "death" then  
            -- Alive  
  
            -- Is the HP above 0?  
            local hp = stats:Get("hp_now")  
            if hp <= 0 then  
                -- Dead party actor we need to deal with  
                controller:Change(CSRunAnim.mName, {'death', false})  
                self.mEventQueue:RemoveEventsOwnedBy(actor)  
            end  
  
        end  
  
    end  
end
```

Listing 3.222: Handling death in the combat state. In CombatState.lua.

HandlePartyDeath loops over the party actors. For each actor it gets the associated character, controller, state, and stats. If a character is playing the death animation, they're already dead and we can ignore them. If they're not already dead, we check if they've *just* died. If their HP is below or at zero, we run the death animation and remove their events from the EventQueue.

When a party member dies, they stay on the battlefield and can be selected under certain conditions. For instance, you might be able to cast a resurrection spell to bring a

character back to life. The same is not true for the enemy; when they die, they immediately become unselectable. When an enemy dies, we move them from the enemy list to a special death list. The death list is stored in `mCombatState`. Copy the code from Listing 3.223.

```
CombatState.__index = CombatState
function CombatState:Create(stack, def)

    local this =
    {
        -- code omitted
        mDeathList = {}
    }
```

Listing 3.223: Recording which enemies have died. In `CombatState.lua`.

The death list stores the dead enemies while they run their death transition. Once the transition finishes, they're removed from game entirely. To make the death transition work, modify the Render and Update functions as shown in Listing 3.224 and Listing 3.225.

```
function CombatState:Render(renderer)

    -- code omitted
    for k, v in ipairs(self.mCharacters['enemy']) do
        v.mEntity:Render(renderer)
    end

    for k, v in ipairs(self.mDeathList) do
        v.mEntity:Render(renderer)
    end
```

Listing 3.224: Drawing dead characters. In `CombatState.lua`.

To render the death list, we call the Render function on each element in the list. Simple. The Update code is more involved. Copy the code from Listing 3.225.

```
function CombatState:Update(dt)

    -- code omitted

    for k, v in ipairs(self.mCharacters['enemy']) do
        v.mController:Update(dt)
    end
```

```

for i = #self.mDeathList, 1, -1 do
    local character = self.mDeathList[i]
    character.mController:Update(dt)
    local state = character.mController.mCurrent

    if state:IsFinished() then
        table.remove(self.mDeathList, i)
    end

end

```

Listing 3.225: Updating dying characters. In CombatState.lua.

In the Update function, we iterate through the death list in reverse. Reversing the iteration makes it easy to remove items from the list. We update the controller for each character in the death list. If the controller's current state is finished, then we remove the enemy from the list and the game.

We have the code to fade out the enemy; now we need the code to trigger it from an attack. Let's implement the HandleEnemyDeath function. This is called from AttackTarget. Copy the code from Listing 3.226.

```

function CombatState:HandleEnemyDeath()

    -- Reverse through list as we're going to remove
    -- them
    local enemyList = self.mActors['enemy']
    for i = #enemyList, 1, -1 do
        local actor = enemyList[i]
        local character = self.mActorCharMap[actor]
        local controller = character.mController
        local stats = actor.mStats

        local hp = stats:Get("hp_now")
        if hp <= 0 then
            -- Remove all references
            table.remove(enemyList, i)
            table.remove(self.mCharacters['enemy'], i)
            self.mActorCharMap[actor] = nil

            controller:Change("cs_die")
            self.mEventQueue:RemoveEventsOwnedBy(actor)

            -- Add to effects
            table.insert(self.mDeathList, character)
        end
    end
end

```

```
        end
    end
end
```

Listing 3.226: Handling dead in the combat state. In CombatState.lua.

The HandleEnemyDeath function moves dead characters from the enemy list to the death list.

We loop through the enemyList in reverse order. We store the character, controller, and stats objects locally for each actor. If an enemy's HP is at or below 0, we take the following steps:

1. Remove it from the enemy actor list.
2. Remove its character object from the character list and actor-character map.
3. Change its state to CSEnemyDie.
4. Remove its events from the event queue.
5. Add it to the death list.

By taking these actions, we're getting rid of any reference to the enemy and making sure it only exists in the death list. The death list will then update the enemy character until it's removed.

Run the code now and you'll be able to attack and kill enemies and players alike. You may need to reduce the HP value for the enemies and players to make it easier to test. The code so far is available as action-4-solution.

Death and Targeting

The early Final Fantasy games had a frustrating combat system. You could tell everyone in your party to attack one enemy. If the first hero killed the enemy it would disappear. On the next hero's turn they'd pointlessly attack the empty space where the enemy once was. This is currently how our game works! But we can improve it.

Example action-5 contains the code so far, or you can continue on with your own code base.

Let's edit the CEAttack class so that when it's executed, it checks whether the target still exists. Copy the code from Listing 3.227.

```
function CEAttack:Execute(queue)
    self.mState.mStack:Push(self.mStoryboard)

    for i = #self.mTargets, 1, -1 do
        local v = self.mTargets[i]
```

```

local hp = v.mStats:Get("hp_now")
if hp <= 0 then
    table.remove(self.mTargets, i)
end

if not next(self.mTargets) then
    -- Find another enemy
    self.mTargets = CombatSelector.WeakestEnemy(self.mState)
end
end

```

Listing 3.227: Making the attack event more intelligent. In CEAttack.lua.

The CEAttack:Execute function now loops through all the targets and removes any dead targets. In our case there's only ever a single target. If the target's health is at or below zero, it's dead, and there's no use attacking a dead creature. After removing invalid targets, we may be left with an empty target list and must take an educated guess as to who's best to attack next. We use the WeakestEnemy selector to select the next weakest enemy.

Run the code and tell everyone to attack the same creature. When it's defeated, they'll nicely retarget the next weakest enemy. The full code is available in action-5-solution.

Reacting to Damage

Our combat system is working well, but let's make it even better! Currently the game is poor at communicating which character has taken damage. We'll add a number of effects to indicate this more clearly. Example action-6 contains a project with all the code we've used thus far, or you can continue on with your own codebase.

When a character takes damage, we change their state to a hurt state. The hurt state performs different actions for the enemy and party characters. Party characters play a flinch animation while enemy characters flash a different color and are knocked back a little. We also create a number that bounces out of the character to show the amount of HP lost.

When a character receives damage, they need to enter the hurt state immediately so they appear to react instantly. The character might be in a state like CSRunAnim or CSStandby. CSHurt replaces the current state, executes, and restores the character's previous state. There are few ways to do this. We could use a stack but that's a little overkill. Instead, the CSHurt state stores the current state locally before replacing it, and then restores it after executing.

Copy the CSHurt code from Listing 3.228.

```

CSHurt = { mName = "cs_hurt" }
CSHurt.__index = CSHurt
function CSHurt:Create(character, context)
    local this =
    {
        mCharacter = character,
        mCombatScene = context,
        mEntity = character.mEntity,
        mPrevState = nil,
    }

    setmetatable(this, self)
    return this
end

function CSHurt:Enter(state)
    self.mPrevState = state
    local frames = self.mCharacter:GetCombatAnim('hurt')
    self.mAnim = Animation>Create(frames, false, 0.2)
    self.mEntity:SetFrame(self.mAnim:Frame())
end

function CSHurt:Exit()
end

function CSHurt:Update(dt)
    if self.mAnim:IsFinished() then
        self.mCharacter.mController.mCurrent = self.mPrevState
        return
    end
    self.mAnim:Update(dt)
    self.mEntity:SetFrame(self.mAnim:Frame())
end

function CSHurt:Render(renderer)
end

```

Listing 3.228: A state for getting hurt in combat. In CSHurt.lua.

In the CSHurt.Enter function, we store the current character state in mPrevState. We create an animation for the flinch in mAnim. The hurt animation is rather short, so the speed per frame is set to 0.2 seconds.

The CSHurt object updates until the animation is finished. Once it's finished, it replaces the mCurrent state on the character controller with its locally stored mPrevState. This

means the Exit function for CSHurt is not called, and the previous state is restored exactly as it was.

Next let's modify CEAttack:DoAttack to trigger the hurt reaction. Copy the code from Listing 3.229.

```
function CEAttack:AttackTarget(target)

    -- code omitted

    -- Change actor's character to hurt state
    local character = self.mState.mActorCharMap[target]
    local controller = character.mController
    if damage > 0 then
        local state = controller.mCurrent
        if state.mName ~= "cs_hurt" then
            controller:Change("cs_hurt", state)
        end
    end

    self.mState:HandleDeath()
end
```

Listing 3.229: Hooking up the hurt reaction to the attack event. In CEAttack.lua.

In AttackTarget shown in Listing 3.229 we check to see if any damage is inflicted. If the target is damaged, we change its state using the id "cs_hurt" and pass in the current state. If the character is already in the hurt state, we do nothing. It's best to avoid the possibility of hurt reactions stacking up.

Run the code and you'll be able to see the player characters flinch when hurt.

We need to add a hurt state for the enemies too. Create a new state called CSEnemyHurt and add it to the dependencies and manifest files. Add a reference to the EntityDefs.gCharacterStates table as shown in Listing 3.230.

```
gCharacterStates =
{
    -- code omitted
    cs_hurt_enemy = CSEnemyHurt,
}
```

Listing 3.230: Adding the combat hurt state to the list of character state. In EntityDefs.lua.

Replace the "cs_hurt" id in the Goblin def with "cs_hurt_enemy". Copy the code from Listing 3.231.

```

goblin =
{
    entity = "goblin",
    controller =
    {
        "cs_run_anim",
        "cs_standby",
        "cs_die_enemy",
        "cs_hurt_enemy",

```

Listing 3.231: Adding hurt state to the goblin. In EntityDefs.lua.

Now we're ready to implement the CSEnemyHurt state and get our enemies flinching! Copy the code from Listing 3.232.

```

CSEnemyHurt = { mName = "cs_hurt" }
CSEnemyHurt.__index = CSEnemyHurt
function CSEnemyHurt:Create(character, context)
    local this =
    {
        mCharacter = character,
        mEntity = character.mEntity,
        mCombatScene = context,
        mSprite = character.mEntity.mSprite,
        mKnockback = 3, -- pixels
        mFlashColor = Vector.Create(1,1,0,1),
        mTween = nil,
        mTime = 0.2,
    }

    setmetatable(this, self)
    return this
end

function CSEnemyHurt:Enter(state)

    self.mPrevState = state

    local pixelPos = self.mEntity.mSprite:GetPosition()
    self.mOriginalX = pixelPos:X()
    self.mOriginalY = pixelPos:Y()

    -- push the entity back a few pixels
    self.mEntity.mSprite:SetPosition(
        self.mOriginalX - self.mKnockback,

```

```

        self.mOriginalY)

        -- Set the sprite a little yellow
        self.mFlashColor = Vector.Create(1,1,0,1)
        self.mEntity.mSprite:SetColor(self.mFlashColor)

        -- Create a tween to run the effects
        self.mTween = Tween>Create(0, 1, self.mTime)

    end

    function CSEnemyHurt:Exit() end
    function CSEnemyHurt:Render(renderer) end

    function CSEnemyHurt:Update(dt)

        if self.mTween:IsFinished() then
            self.mCharacter.mController.mCurrent = self.mPrevState
            return
        end

        self.mTween:Update(dt)

        local value = self.mTween:Value()

        self.mEntity.mSprite:SetPosition(
            self.mOriginalX + self.mKnockback * value,
            self.mOriginalY)

        self.mFlashColor:SetZ(value)
        self.mEntity.mSprite:SetColor(self.mFlashColor)

    end

```

Listing 3.232: The combat state to show the enemy being hurt. In CSEnemyHurt.lua.

Note the `mName` of `CSEnemyHurt` in Listing 3.232; it's the *same* as `CSHurt`. This is important because it means when we change to state "cs_hurt" for enemies we go to `CSEnemyHurt` and for party members we go to `CSHurt`.

Like `CSHurt`, `CSEnemyHurt` stores a reference to the current state which it restores after executing. In the constructor, the most interesting fields are `mKnockback` and `mFlashColor`; these control the enemy reaction to being hurt. The `mTime` field determines how long the hurt reaction lasts. It's set to 0.2 seconds so it's pretty fast. The `mKnockback` field is the number of pixels we push the enemy back. The `mFlashColor` field is the color to flash the enemy sprite.

The Enter function takes in the current character state which we store in `mPrevState`. We store the current position of the character sprite in `mOriginalX` and `mOriginalY`. We store the original position to ensure our sprite ends up back where it started.

Next we move the sprite back `mKnockback` pixels on the X axis. This gives the impression that the sprite has been pushed back suddenly. We set the flash color to yellow and apply it to the sprite. Finally we set up the tween to go from 0 to 1 over `mTime` seconds.

The Update function restores the sprite's original color and moves it back to its original position using the `mTween`. When the tween finishes, we restore the previous state of the character controller.

Run the code and you'll be able to see the hurt effect for both enemies and party members. Example action-6-solution contains the completed code.

Combat Effects

To represent damage inflicted, numbers should appear above the character sprite. The damage numbers are part of a more general combat special effects system. Other special effects might include making a slash mark appear over the target for physical attacks, causing fire effects over the target for magical attacks, and so on.

Any effect must implement the following functions:

IsFinished Reports when the effect is finished so it can be removed from the list.

Update Updates the effect according to the elapsed frame time.

Render Renders the effect to the screen.

mPriority Controls the render order. For instance, the jumping numbers should appear on top of everything else. Lower priority numbers are rendered later. 1 is considered the highest priority.

Each effect has a priority number to control the order it's rendered in. There are very efficient ways to create a priority list, but we'll use a simple list that we sort by priority. In the `CombatState` we'll create a list call `mEffects` where we'll store and sort the effect order.

Example action-7 has all the code we've written so far, or you can continue to use your own codebase.

Copy the code from Listing 3.233.

```
CombatState.__index = CombatState
function CombatState:Create(stack, def)

    local this =
```

```
{  
    -- code omitted  
    mDeathList = {},  
    mEffectList = {},  
}
```

Listing 3.233: Adding a list of effects to the combat state. In CombatState.lua.

The `mEffectList` in Listing 3.233 is just an empty table added. Copy the code from Listing 3.234 to add support for updating the effects.

```
function CombatState:Update(dt)  
  
    -- code omitted  
  
    for i = #self.mDeathList, 1, -1 do  
  
        -- code omitted  
  
    end  
  
    for i = #self.mEffectList, 1, -1 do  
        local v = self.mEffectList[i]  
        if v:IsFinished() then  
            table.remove(self.mEffectList, i)  
        end  
        v:Update(dt)  
    end
```

Listing 3.234: Adding the effects to Update. In CombatState.lua.

The `Update` function iterates through the effect list. If any effect is finished it's removed; otherwise it's updated.

Copy the code from Listing 3.235 to add the effect render code.

```
function CombatState:Render(renderer)  
  
    -- code omitted  
  
    for k, v in ipairs(self.mDeathList) do  
        v.mEntity:Render(renderer)  
    end
```

```
for k, v in ipairs(self.mEffectList) do
    v:Render(renderer)
end
```

Listing 3.235: The effects are rendered after the death list but before the panels in Render. In CombatState.lua.

In the Render function, each effect is told to render itself.

To insert an effect into the effect list in a controlled way, we use an AddEffect function. Copy the code from Listing 3.236.

```
function CombatState:AddEffect(effect)

    for i = 1, #self.mEffectList do

        local priority = self.mEffectList[i].mPriority

        if effect.mPriority > priority then
            table.insert(self.mEffectList, i, effect)
            return
        end
    end

    table.insert(self.mEffectList, effect)
end
```

Listing 3.236: Adding effects with priorities. In CombatState.lua.

Effects are added by priority number, with big numbers at the front of the list and small ones at the bottom. This means lower priority numbers are rendered last, which draws them on top of everything else.

With the effect system in place, let's write some cool effects to spice up our combat!

Jumping Numbers

Now we're ready to add the damage numbers. In the code directory create a subfolder called "fx", and in that folder create a new file: JumpingNumbers.lua. Add it to the manifest and dependencies (remember it's using the fx subfolder).

The JumpingNumbers constructor takes in a position, a number, and an optional color value. It displays the number as a piece of text. As the effect runs, the number accelerates upwards, slows, and falls back down while fading out. To do this, we'll use a very basic physics simulation. Copy the code from Listing 3.237.

```

JumpingNumbers = {}
JumpingNumbers.__index = JumpingNumbers
function JumpingNumbers:Create(x, y, number, color)
    local this =
    {
        mX = x or 0,
        mY = y or 0,
        mGravity = 700, -- pixels per second
        mFadeDistance = 33, -- pixels
        mScale = 1.3,
        mNumber = number or 0, -- to display
        mColor = color or Vector.Create(1,1,1,1),
        mPriority = 1,
    }
    this.mCurrentY = this.mY
    this.mVelocityY = 224,
    setmetatable(this, self)
    return this
end

function JumpingNumbers:Update(dt)

    self.mCurrentY = self.mCurrentY + (self.mVelocityY * dt)
    self.mVelocityY = self.mVelocityY - (self.mGravity * dt)

    if self.mCurrentY <= self.mY then
        local fade01 = (self.mY - self.mCurrentY) / self.mFadeDistance
        self.mColor:SetW(1 - fade01)
    end
end

function JumpingNumbers:Render(renderer)

    renderer:ScaleText(self.mScale, self.mScale)
    renderer:AlignText("center", "center")

    renderer:DrawText2d(self.mX,
        math.floor(self.mCurrentY),
        tostring(self.mNumber),
        self.mColor)
end

function JumpingNumbers:IsFinished()
    -- Has it passed the fade out point?

```

```
    return self.mCurrentY <= (self.mY - self.mFadeDistance)
end
```

Listing 3.237: Creating a class for creating jumping number effects. In JumpingNumbers.lua.

Let's start with the constructor. We store the text spawn position in `mX` and `mY` and set a gravity value of 700 pixels per second. The `mGravity` value controls how fast the text falls.

The `mFadeDistance` controls when the text fades out as it drops. It's set to 33 pixels. As the text drops below its starting point, it fades from 1 to 0. When it drops 33 pixels below the starting point, it has an alpha of 0 and the effect is over. When it's at position `mY - 16.5`, it's halfway to the full fade distance so it has an alpha value of `0.5f`.

The `mScale` is the scale of the text. The `mNumber` is the number to display. The `mColor` is the starting color for the text. Finally we set `mPriority` to 1, which means it's drawn last as an effect. The priority value ensures it won't be obscured by other effects.

Below the this table we create a new field, `mCurrentY`, to track the current position of the text on the y axis. The `mY` always stores the starting position of the text.

The `mVelocity`⁶ is set to 224 pixels per second. Changing the `mVelocity` changes how rapidly and how high the text rises. Then that's it for the constructor. Next let's check out the Update.

Update

The Update function increases the `mCurrentY` value by the `mVelocity` multiplied by the delta time. This is the physics step. The position of the text is changed by the velocity. After moving the text, we reduce the velocity by the gravity. Eventually the velocity is reduced so much that it becomes negative and the text starts to drop.

At the end of the Update function, we check to see if the text has passed below the starting y position. If it has, then its alpha is set according to how far it has fallen. The variable `fade01` stores how far the text has fallen below the origin. If it's 0 it's at the origin, and if it's 1 it's at the `mFadeDistance`. We set the alpha to the inverse of `fade01`. The inverse is made by subtracting `fade01` from 1.

Render and Finish

The Render function handles text scale and alignment. It draws the text twice to produce a drop shadow effect. First a black version of the text is drawn slightly offset, then the

⁶Velocity is a measure of speed in a direction and therefore is usually a vector. In this case the direction is implied as up on the y-axis.

colored version is drawn. A shadow makes the text more readable, no matter the background.

The IsFinished function checks if the text has fallen so far below its starting position that it's totally invisible. If so, it returns true.

Let's try it out in game!

Integrating the Jumping Numbers Effect

Open up the CEAttack.lua file and find the DoAttack function. Copy the code in Listing 3.238.

```
function CEAttack:DoAttack(target)

    -- code omitted

    local entity = character.mEntity
    local dmgEffect = JumpingNumbers>Create(entity.mX, entity.mY, damage)
    self.mState:AddEffect(dmgEffect)

    self.mState:HandleDeath()

end
```

Listing 3.238: Adding the jump effect when the attack occurs. In CEAttack.lua.

When a character is attacked, we add a JumpingNumber effect to the effect manager to display the damage. The position is set to where the attacked character is standing. You can see how it looks in Figure 3.46.



Figure 3.46: Damage numbers are now displayed on a character being hurt.

Run the code and check it out. Notice how it's starting to feel like a real RPG combat system! Enemies are knocked back and display the damage they receive.

Slash Effect

The slash effect is a simple animation we'll play when a target takes damage. It gives the impression of a weapon slash.

In the art folder you'll see there's `combat_slash.png`, which if you're using action-7 has already been added to the manifest (if not copy it over and add it!). This slash has three frames that play as an animation. You can see the slash frames in Figure 3.47.

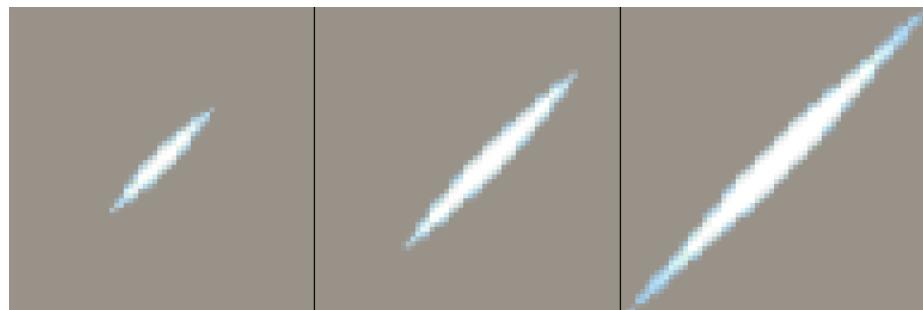


Figure 3.47: The three frames of the combat slash.

First make an entity definition for the slash. Copy the code from Listing 3.239.

```
gEntities =
{
    -- code omitted
    slash =
    {
        texture = "combat_slash.png",
        width = 64,
        height = 64,
        startFrame = 3,
        frames = {3, 2, 1}
    }
}
```

Listing 3.239: Adding an entity to represent the slash effect. In EntityDefs.lua.

The combat slash has three frames, each 64 by 64 pixels. We store the frame data in the frames field. The slash should go from big to small, so the frames here are in reverse order.

In the code/fx folder, create AnimEntityFx.lua (not a great name but descriptive). We'll use this class for all animation-based effects. Add it to the manifest and dependencies files.

Copy the code from Listing 3.240.

```
AnimEntityFx = {}
AnimEntityFx.__index = AnimEntityFx
function AnimEntityFx:Create(x, y, def, frames, spf)

    spf = spf or 0.015

    local this =
    {
        mEntity = Entity:Create(def),
        mAnim = Animation:Create(frames, false, spf),
        mPriority = 2,
    }
    this.mEntity.mX = x
    this.mEntity.mY = y
    this.mEntity.mSprite:SetPosition(x, y)
    setmetatable(this, self)
    return this
}
```

```

end

function AnimEntityFx:Update(dt)
    self.mAnim:Update(dt)
    self.mEntity:SetFrame(self.mAnim:Frame())
end

function AnimEntityFx:Render(renderer)
    self.mEntity:Render(renderer)
end

function AnimEntityFx:IsFinished()
    return self.mAnim:IsFinished()
end

```

Listing 3.240: A simple animation effects class. In AnimEntityFx.lua.

AnimEntityFx takes in an x, y position, an entity definition, a list of frames, and a speed per frame. We'll want to run some effects at different speeds so the spf is there if needed. The spf value is optional and defaults to 0.015 seconds.

Then the entity is created from the def, and the animation is created from the frames. The animation is non-looping as we want our combat effects to end. After the this table, we position the entity.

The Update function updates the animation, and the Render function renders the current frame. The IsFinished returns true when the mAnim finishes.

Integrating the Slash Effect

To get this effect in, let's return to the CEAttack.lua file and edit the DoAttack function. Copy the code from Listing 3.241.

```

function CEAttack:AttackTarget(target)

    -- code omitted

    local entity = character.mEntity
    local x = entity.mX
    local y = entity.mY
    local dmgEffect = JumpingNumbers>Create(x, y, damage)
    local slashEffect = AnimEntityFx>Create(x, y,
                                             gEntities.slash,
                                             gEntities.slash.frames)

```

```
    self.mState:AddEffect(dmgEffect)
    self.mState:AddEffect(slashEffect)
```

Listing 3.241: Adding the damage and slash effects to the attack. In CEAttack.lua.

In Listing 3.241 we've cleaned up the DoAttack code as JumpingNumbers and slash effect share some variables. The slash effect is created using the slash entity def. Once it's created, we add the effect to our effect list using the CombatState:AddEffect function.

Try it out! The code so far is available in action-7-solution. The combat feels much better with these added effects.

Enemies Fighting Back

At the moment, enemies stand there and get hit. It would be nice if they hit back. We're not going to implement a full AI solution, but they should at least do a dumb attack. Example action-8 contains all the code so far.

Currently CETurn:Execute does nothing if it's an enemy character event. Let's change that and push an attack onto the event stack. To create an attack event, we need a target. We'll choose a target at random. To do that, we need a new selector. The selector we'll use is RandomAlivePlayer. Copy the code from Listing 3.242.

```
CombatSelector =
{
    RandomAlivePlayer = function(state)

        local aliveList = {}
        for k, v in ipairs(state.mActors["party"]) do
            if v.mStats:Get("hp_now") > 0 then
                table.insert(aliveList, v)
            end
        end

        local target = aliveList[math.random(#aliveList)]
        return { target }
    end,
}
```

Listing 3.242: CombatSelector for choosing a random living player. In CombatTargetState.

The RandomAlivePlayer does as its name suggests and picks a random member of the player party that's still alive. We'll modify CEAttack to work for player and enemy attacks, but before that we're going to tweak the combat formula.

Combat Formula

To make things more explicit, let's move the attack calculation into its own file. This file will be the one place for combat formulas, to makes things easier when balancing.

Create CombatFormula.lua in the code folder. Add it to the dependencies and manifest files. We're going to add the simple melee combat formula here. Copy the code from Listing 3.243

```
--  
-- The calculations for combat  
--  
Formula = {}  
function Formula.MeleeAttack(state, attacker, target)  
  
    local stats = attacker.mStats  
    local enemyStats = target.mStats  
  
    -- Simple attack get  
    local attack = stats:Get("attack")  
    attack = attack + stats:Get("strength")  
    local defense = enemyStats:Get("defense")  
  
    local damage = math.max(0, attack - defense)  
  
    return damage  
end
```

Listing 3.243: The basic melee attack formula. In CombatFormula.lua.

The MeleeAttack is used for basic hand-to-hand combat, and we've just copied in the simple formula from CEAAttack. It works out how much damage is inflicted but it doesn't apply that damage. To apply the damage, we'll add an ApplyDamage function into the CombatState class.

ApplyDamage is a function that can be used anywhere; for normal attacks, magic attacks, item attacks, it doesn't matter – it just applies the damage. Anything fancy like dodging, defending, exploding on an attack, or doing a counter attack is handled separately in the CEAAttack class. Copy the code from Listing 3.244.

```
function CombatState:ApplyDamage(target, damage)  
    local stats = target.mStats  
    local hp = stats:Get("hp_now") - damage  
    stats:Set("hp_now", math.max(0, hp))  
    print("hp is", stats:Get("hp_now"))
```

```

-- Change actor's character to hurt state
local character = self.mActorCharMap[target]
local controller = character.mController

if damage > 0 then
    local state = controller.mCurrent
    if state.mName ~= "cs_hurt" then
        controller:Change("cs_hurt", state)
    end
end

local entity = character.mEntity
local x = entity.mX
local y = entity.mY
local dmgEffect = JumpingNumbers:Create(x, y, damage)
self:AddEffect(dmgEffect)
self:HandleDeath()
end

```

Listing 3.244: ApplyDamage function to be used after melee, magic or other damaging attacks. In main.lua.

This is all code we've written previously for CEAttack. We're just making it more easily available. Let's modify the CEAttack:AttackTarget function to reflect the changes we've made. Copy the code from Listing 3.245.

```

function CEAttack:AttackTarget(target)

    local damage = Formula.MeleeAttack(self.mState, self.mOwner, target)
    local entity = self.mState.mActorCharMap[target].mEntity

    self.mState:ApplyDamage(target, damage)

    local x = entity.mX
    local y = entity.mY
    local slashEffect = AnimEntityFx:Create(x, y,
                                             gEntities.slash,
                                             gEntities.slash.frames)

    self.mState:AddEffect(slashEffect)

end

```

Listing 3.245: CEAttack updates. In CEAttack.lua.

This is now a much simpler function. The slash is effect is applied here, but everything else is handled by the combat formula or combat state.

The EnemyAttack

The enemy also attacks using CEAttack but with different special effects. We use the def table to store whether an attack comes from an enemy or player. Let's try it now in CETurn:Execute. Copy the code from Listing 3.246.

```
function CETurn:Execute(queue)

    if self.mState:IsPartyMember(self.mOwner) then
        -- code omitted
    else
        -- do a dumb attack
        local targets = CombatSelector.RandomAlivePlayer(self.mState)
        local def = { player = false }
        local queue = self.mState.mEventQueue
        local event = CEAttack>Create(self.mState, self.mOwner, def, targets)
        local tp = self.mOwner:TimePoints(queue)
        queue:Add(event, tp)
        self.mIsFinished = true
        return
    end

end
```

Listing 3.246: Adding different attacks effects. In CETurn.lua.

In Listing 3.246 the def table contains a player flag set to false. This means the attack does not come from the player. Next copy the code for the player attack in CombatChoiceState from Listing 3.247.

```
function CombatChoiceState:TakeAction(id, targets)
    self.mStack:Pop() -- select state
    self.mStack:Pop() -- action state

    local queue = self.mCombatState.mEventQueue

    if id == "attack" then
        print("Entered attack state")
        local attack = CEAttack>Create(self.mCombatState,
                                         self.mActor,
                                         { player = true }
                                         targets)
```

Listing 3.247: Adding a tag to show it's a player attack. In CombatChoiceState.lua.

In Listing 3.247 we've marked the player's attack as coming from the player.

For the enemy attack, we'll have the enemy jump forward, do the attack, and then move back. To do the forward movement, we'll use the CSMove state. Add "cs_move" to the enemy def controller as shown in Listing 3.248.

```
goblin =
{
    entity = "goblin",
    controller =
    {
        -- code omitted
        "cs_move",
    },
}
```

Listing 3.248: Adding the combat movement state to the goblin. In EntityDefs.lua.

The slash effect for the enemies is different from the player one. In the art folder, there's a file called "combat_claw.png". Add it to the manifest. Then copy the entity entry in Listing 3.249 to the EntityDefs.lua file.

```
gEntities =
{
    -- code omitted
    claw =
    {
        texture = "combat_claw.png",
        width = 64,
        height = 64,
        startFrame = 1,
        frames = {1, 2, 3}
    }
}
```

Listing 3.249: Adding the claw def. In EntityDefs.lua..

The claw effect helps differentiate between enemy and player attacks.

Now we're finally ready to update the CEAttack, starting with the constructor. Copy the code from Listing 3.250.

```

function CEAttack>Create(state, owner, def, targets)

    -- code omitted

    setmetatable(this, self)

    local storyboard = nil

    if this.mDef.player then
        this.mAttackAnim = gEntities.slash
        this.mDefaultTargeter = CombatSelector.WeakestEnemy

        storyboard =
        {
            SOP.RunState(this.mController, CSMove.mName, {dir = 1}),
            SOP.RunState(this.mController, CSRanAnim.mName, {'attack', false}),
            SOP.Function(function() this:DoAttack() end),
            SOP.RunState(this.mController, CSMove.mName, {dir = -1}),
            SOP.Function(function() this:OnFinish() end)
        }
    else
        this.mAttackAnim = gEntities.claw
        this.mDefaultTargeter = CombatSelector.RandomAlivePlayer

        storyboard =
        {
            SOP.RunState(this.mController,
                        CSMove.mName,
                        {dir = 1, distance = 8, time = 0.1}),
            SOP.Function(function() this:DoAttack() end),
            SOP.RunState(this.mController,
                        CSMove.mName,
                        {dir = -1, distance = 8, time = 0.4}),
            SOP.Function(function() this:OnFinish() end)
        }
    end

    this.mStoryboard = Storyboard>Create(this.mState.mStack,
                                         storyboard)

    return this
end

```

Listing 3.250: Adding effects to the attack constructor. In CEAttack.lua.

The def table now tells us if this is a player or enemy attack.

The mStoryboard, describing the steps of the attack, has moved from the this table to later on in the constructor. The mStoryboard differs for player and enemy attacks. The player steps are as before but the enemy steps are new.

The enemy storyboard contains no animation operations. Instead, the enemy moves 8 pixels toward the player and then back again. The second difference between the attacks is the effects. The constructor uses the mAttackAnim animation to play the attack effect over the target. For the player it's set to slash, and for the enemy the it uses the claw animation. We also set mDefaultTargeter which is the function used to choose a new target when the previous is lost. For the player mDefaultTargeter is WeakestEnemy, and for the enemy it's RandomAlivePlayer.

Let's update the Execute function to use the mDefaultTargeter function. Copy the code from Listing 3.251

```
function CEAttack:Execute(queue)

    -- code omitted

    if not next(self.mTargets) then
        self.mTargets = self.mDefaultTargeter(self.mState)
    end

end
```

Listing 3.251: Getting a new target if the previous one has been killed for both enemies and players. In CEAttack.lua.

Now in the Execute function, when the target has been lost it uses mDefaultTargeter to pick a new target.

The final function to update is DoAttack so it uses the new self.mAttackAnim animation as an attack effect. Copy the code from Listing 3.252.

```
function CEAttack:DoAttack(target)

    -- code omitted

    local effect = AnimEntityFx>Create(x, y,
                                         self.mAttackAnim,
                                         self.mAttackAnim.frames)

    self.mState:AddEffect(effect)
```

```
end
```

Listing 3.252: Animating the attack effects. In CEAttack.lua.

Instead of always playing the slash effect on an attack, we now use the effect stored in `mAttackAnim`. Run the code and you'll see the enemies attack the party members. With this small step, we have a fully functioning battle system! Example action-8-solution contains the full code so far. Figure 3.48 shows the enemy attacking the party and the party correctly entering the death state.



Figure 3.48: Here the enemy can be seen to attack and kill one of the party members.

Over the next few sections we'll handle the win and lose conditions before moving on to add more tactical depth to the combat simulation. Good work so far!

Winners, Losers and Loot

We can simulate a battle with the `CombatState` but, as of yet, we don't have any way to handle winning or losing. When the player party wins, the characters should run a victory animation and the state should change to display a combat summary screen. The summary screen awards XP and loot, and notifies the player if any hero has leveled up or unlocked some new ability. If the player party loses combat, then we enter a

game over state. This game over state shows a game over message and prompts the player to continue (from the last save) or start a new game.

Game Over

We'll start with the Game Over state as it's quite simple. The state offers the player the option to continue from the last save but, as we don't have a save load mechanism yet, it does nothing.

Example loot-1 contains all the code so far. This project has a few changes, so I'd recommend switching to it. The party now has 3HP each, so we can easily enter the game over state. Run the game and you should die pretty much immediately.

GameOverState Reborn

When we created the Dungeon game in the first part of the book, we made a GameOver-State.lua file. There's no need to reinvent the wheel. We'll reuse that. CaptionStyles.lua has been modified in loot-1 to remove the title font, as it's not included in this project.

To make the GameOverState easier to edit, let's push it to the top of the stack in the main.lua file. Copy the code from Listing 3.253.

```
gStack:Push(ExploreState>Create(gStack,
    CreateArenaMap(),
    Vector.Create(15, 8, 1)))

gStack:Push(CombatState>Create(gStack, gCombatDef))
gStack:Push(GameOverState>Create(gStack, gWorld))
```

Listing 3.253: Pushing the GameOverState onto the stack. In main.lua.

Run the game and you'll see the GameOverState showing the "Game Over" text on top of the combat, as shown in Figure 3.49. It doesn't look good.



Figure 3.49: A bad looking game over screen.

Let's make some changes to make the screen suitable for a general game over. Copy the code from Listing 3.254.

```
GameOverState = {}
GameOverState.__index = GameOverState
function GameOverState:Create(stack, world)
    local this =
    {
        mWorld = world,
        mStack = stack,
    }

    setmetatable(this, self)

    this.mMenu = Selection:Create
    {
        data = { "New Game", "Continue" },
        spacingY = 36,
        OnSelection = function(...) this:OnSelect(...) end,
    }
    -- Select continue by default
    this.mMenu.mFocusY = 2
```

```

        this.mMenu:SetPosition(-this.mMenu:GetWidth(), 0)

        return this
    end

    function GameOverState:Enter()
        CaptionStyles["title"].color:SetW(1)
    end
    function GameOverState:Exit() end
    function GameOverState:Update(dt) end

    function GameOverState:HandleInput()
        self.mMenu:HandleInput()
    end

    function GameOverState:Render(renderer)

        renderer:DrawRect2d(System.ScreenTopLeft(),
                            System.ScreenBottomRight(),
                            Vector.Create(0,0,0,1))

        CaptionStyles["title"]:Render(renderer,
                                      "Game Over")

        renderer:AlignText("left", "center")
        self.mMenu:Render(renderer)
    end

```

Listing 3.254: Bringing the Game Over screen up to date. In GameOverState.lua.

The GameOverState has certainly become a bit fatter. We've removed the subtitle and added a simple selection menu. The menu prompts the user to "Continue" (from the last save) or start a "New Game".

Note that there's a new world parameter in the constructor. Later, world is used for loading and saving, but for now we'll ignore it. After the this table, we create a selection menu, passing in the bare minimum of settings. There are two choices, "New Game" and "Continue", and the spacing between the choices is set to 36 pixels. When either choice is selected, a new function, OnSelect, is called.

Once the mMenu is created, we set the focus to the "Continue" option by setting mFocusY to 2. We set the choice to "Continue" by default because it's the option the player will select most often. Finally we position the menu so it's more centered.

The Enter function is unchanged, but the reference to the subtitle has been removed. HandleInput now contains code telling the selection menu to handle input. The Render

function draws a full-screen black rectangle, then renders the “Game Over” title, followed by the selection menu. Run the code and you’ll see something similar to Figure 3.50.



Figure 3.50: A nicer game over screen.

To make the game over screen functional, we need to implement the `OnSelect` function. Let’s do that now. Copy the code from Listing 3.255.

```
function GameOverState:OnSelect(index, data)
    local NEWGAME = 1
    local CONTINUE = 2

    if index == NEWGAME then

        gStack = StateStack>Create()
        gWorld = World>Create()
        gStack:Push(ExploreState>Create(gStack, CreateArenaMap(),
            Vector.Create(30, 18, 1)))

    elseif index == CONTINUE then
        print("No save system. No continue.")
    end
end
```

Listing 3.255: The Select callback in GameOverState.lua.

On selecting “New Game” we recreate the global stack, world object and explore state. Then we push the explore state onto the stack in order to reset the game.

We have no save system, so for “Continue” we just print a message to the console.

Triggering Game Over

When the party is defeated, we want the camera to linger for a while, then we’ll fade to black and enter the GameOverState. We’ll use a ‘Storyboard’ to achieve this effect. First, in the main.lua file, remove the line of code shown in Listing 3.256.

```
gStack:Push(GameOverState>Create(gStack, gWorld))
```

Listing 3.256: Remove this. In main.lua.

The GameOverState is going to be triggered when the party is defeated, so we don’t need to force it on top of the stack anymore.

Open up the CombatState.lua file. Add a flag in the constructor, `mIsFinishing`, as shown in Listing 3.257. This flag tells us if the combat is finishing and running the fade out storyboard.

```
function CombatState>Create(stack, def)

    local this =
    {
        -- code omitted
        mIsFinishing = false,
    }
```

Listing 3.257: Add a flag to signal that the combat is ending. In CombatState.lua.

Next let’s add two new function calls to `OnWin` and `OnLose` in the Update code. Copy the code from Listing 3.258.

```
function CombatState:Update(dt)

    -- code omitted
    if self.mStack:Top() ~= nil then

        self.mStack:Update(dt)
```

```

elseif not self.mIsFinishing then

    self.mEventQueue:Update()

    self:AddTurns(self.mActors.enemy)
    self:AddTurns(self.mActors.party)

    if self:PartyWins() then

        self.mEventQueue:Clear()
        self:OnWin()
    elseif self:EnemyWins() then

        self.mEventQueue:Clear()
        self:OnLose()
    end
end

function CombatState:OnWin()
    self.mIsFinishing = true
end

function CombatState:OnLose()
    self.mIsFinishing = true
end

```

Listing 3.258: Adding functions to deal with winning and losing. In CombatState.lua.

When the player wins, `OnWin` is called. `OnLose` is called when they lose. In both cases the combat state is finished, so the `mIsFinishing` flag is set to true. When `mIsFinishing` is true, we're ready to fade out. Note the `mIsFinishing` flag is only checked when the event queue is empty. This ensures the game doesn't start to end while something is still happening during combat.

When the game finishes, we're going to fade out on the combat state. The player will see a fully black screen, then behind the screen we'll kill off the combat state and replace it with a game over screen. Once that's done, we'll fade the black screen out again and the player will see the game over screen. To the player this will appear seamless and smooth.

In order to implement `OnWin` and `OnLose` we need to add two extra Storyboard operations, `ReplaceState` and `UpdateState`. `ReplaceState` searches through the global stack and replaces one state with another. We'll use this to replace the `CombatState` with the `GameOverState` as we fade to black. The `UpdateState` manually updates a state, and

we use this to keep the CombatState updating as we fade out. Copy the new operations from Listing 3.259.

```
function SOP.ReplaceState(current, new)
    return function(storyboard)
        print("being asked to replace a state")
        local stack = storyboard.mStack

        for k, v in ipairs(stack.mStates) do
            if v == current then
                stack.mStates[k]:Exit()
                stack.mStates[k] = new
                stack.mStates[k]:Enter()
                return EmptyEvent
            end
        end

        print("Failed to replace state in storyboard.")
        assert(false) -- failed to replace
    end
end

function SOP.UpdateState(state, time)
    return function(storyboard)
        return TweenEvent:Create(
            Tween:Create(0, 1, time),
            state,
            function(target, value)
                target:Update(GetDeltaTime())
            end)
    end
end
```

Listing 3.259: Two new storyboard operations. In StoryboardEvents.lua.

The ReplaceState operation, when executed, gets the stack from the storyboard, iterates through the mStates table, and replaces the state current with the state new. When a state is replaced, the Exit function is called on the old state and the Enter function is called on the new one. The operation, when executed, returns an EmptyEvent. Replacing a state happens in a single frame. If no state is found, we print a message and assert as this is a programmer error!

The UpdateState updates a given state for a number of seconds by reusing the TweenEvent operation. This lets us update states that aren't on the top of the stack.

Next let's implement the OnLose function using the ReplaceState operation. Copy the code from Listing 3.260.

```
function CombatState:OnLose()

    local storyboard =
    {
        SOP.UpdateState(self, 1.5),
        SOP.BlackScreen("black", 0),
        SOP.FadeInScreen("black"),
        SOP.ReplaceState(self,
            GameOverState>Create(self.mGameStack, gWorld)),
        SOP.Wait(2),
        SOP.FadeOutScreen("black"),
    }
    local storyboard = Storyboard>Create(self.mGameStack, storyboard)
    self.mGameStack:Push(storyboard)
    self.mIsFinishing = true
end
```

Listing 3.260: A story for losing combat. In CombatState.lua.

OnLose defines a storyboard using a list of operations. The first operation updates the combat state for another 1.5 seconds, so we get to see the end of any effects. Then we display a black screen with the alpha channel set to zero. The next operation fades in the black screen. When the fade is finished, the CombatState is removed from the stack and replaced with the GameOverState. There's a 2-second wait, to allow the player to consider where they might have gone wrong, then the black screen fades out.

The storyboard object is created and pushed on the stack and starts executing on the next frame.

Run the code and you'll see the party die and the storyboard play out. You can even select "New Game" but you'll be stuck in the arena with no way to reenter combat at this point. Example `loot-1-solution` has code we've written so far.

Our game over screen is bare bones but functional. In a full game you might consider adding an image, music, or a little particle effect to help take the sting out of the player's death.

Winning and the Combat Summary

After winning a battle, the player is shown two screens. The first screen displays the experience points the party has gained. The second screen displays the items recovered from battle. Each screen is represented with a state, `XPSummaryState.lua` and `LootSummaryState.lua`.

XPSummaryState informs the player if any actors have levelled up or unlocked any abilities. JRPGs have many different ways to advance players' abilities, and this screen is a good place to update the player on any leveling progress. The levelling up scheme for the book is simple and we mainly give out experience points on this screen. You can see the screen in Figure 3.51.



Figure 3.51: What our XP Summary Screen will look like.

Adding the combat summary screens means we need to flesh the game a little more with code and data. Here is some of the data we need:

- XP value of enemies
- Total enemies killed in combat
- Loot tables for enemies
- Abilities and when they're unlocked

We'll add this data as we go.

Once we've created both combat summary screens, they need to be added into the game. Here's how they'll be used after successful combat:

1. Do Victory Dance
2. Quick Fade Out
3. Screen 1: Show XP and any level up information

4. Screen 2: Show gold and loot
5. Fade back to the game

We'll start with the XPSummaryState and add more data to the game world as we go.

XPSummaryState

Let's write the layout code for XPSummaryState first, then worry about hooking in the combat data. You can see the desired layout in Figure 3.51.

The XP summary screen is broken horizontally into five panels. The top panel is the title panel announcing that experience points are being rewarded. The second panel tells the player how many experience points they've received. The final three panels each represent a party member.

Each party member panel displays the experience and level information. Ideally we'd reuse the ActorSummary class, but our needs are a little too different. The information on this panel is *only* about experience and levels. When the player enters the XPSummaryState the XP points are given to the party members and the XP bars fill up. If any actor levels up, then a little "Level Up" notification appears on top of the party member panel.

A new ActorXPSummary class is used to display the actor level information and a new XPPopUp class displays messages on top of the ActorXPSummary. ActorXPSummary is responsible for managing its own pop-up notifications. For now the pop-ups only say "Level Up" but later we'll add pop-ups for unlocking abilities and skills.

Create XPSummaryState.lua, ActorXPSummary.lua and XPPopUp.lua, and add them to the code directory and manifest and dependencies files.

XPPopUp

The XPPopUp class creates a little box that appears on top of the actor summary. It uses a Panel as a backing image and fades in and out. The pop-up is centered around its position. Copy the code for the constructor from Listing 3.261.

```
XPPopUp = {}
XPPopUp.__index = XPPopUp
function XPPopUp:Create(text, x, y, color)
    local this =
    {
        mText = text,
        mX = x or 0,
        mY = y or 0,
        mTextColor = color or Vector.Create(1, 1, 0, 1),
```

```

mTween = nil,
mFadeTime = 0.25,
mPane = Panel:Create
{
    texture = Texture.Find("gradient_panel.png"),
    size = 3,
},
mDisplayTime = 0,
}

setmetatable(this, self)
return this
end

```

Listing 3.261: The XPPopUp constructor. In XPPopUp.lua.

The constructor takes in text for the pop-up to display, an x and y for its start position, and an optional color value to color the text. The parameters are stored in mText, mX, mY and mTextColor respectively. The position defaults to 0, 0. The color defaults to yellow.

An mTween controls for the fade transition. The tween isn't created in the constructor, but there's a reference to show it's going to be used. The mFadeTime defaults to 0.25 seconds. This is how long the pop-up takes to fade in and out. A standard panel is created and stored in mPane. Finally there's an mDisplayTime field that records how long the pop-up has been displayed. This is all code of a type we've seen before.

Let's move on to some helper functions to control the pop-up. Copy the code from Listing 3.262.

```

function XPPopUp:SetPosition(x, y)
    self.mX = x
    self.mY = y
end

function XPPopUp:TurnOn()
    self.mTween = Tween:Create(0, 1, self.mFadeTime)
end

function XPPopUp:TurnOff()
    local current = self.mTween:Value()
    self.mTween = Tween:Create(current, 0, current * self.mFadeTime)
end

function XPPopUp:IsTurningOff()
    return self.mTween:FinishValue() == 0

```

```

end

function XPPopUp:IsFinished()
    return self.mTween:Value() == 0 and self.mTween:FinishValue() == 0
end

```

Listing 3.262: XPPopUp Utility Functions. In XPPopUp.lua.

`SetPosition` is self-descriptive; it sets the pop-up's location. `TurnOn` creates the tween used to fade in the pop-up in `mFadeTime` seconds. `TurnOff` does the same but fades the pop-up out. The `TurnOff` tween doesn't assume the pop-up is fully visible. It tweens from the current alpha value and reduces the `mFadeTime` as needed. This allows us to cancel a pop-up during its fade-in transition. Players often want to skip through the summary screens, so there's no reason to hold them up longer than needed.

`IsTurningOff` reports if the pop-up is currently doing its fade-out transition. It returns true if the tween is heading towards zero. 'IsFinished' reports if the pop-up is fully off by checking if it's heading towards zero and is currently zero.

The Panel class doesn't support changing color, so let's add support now. Copy the `SetColor` function from Listing 3.263.

```

function Panel:SetColor(color)
    for k, v in ipairs(self.mTiles) do
        v:SetColor(color)
    end
end

```

Listing 3.263: Setting Color. In Panel.lua.

Panels are made of 9 tiles, so the color needs to be set for each. With that change we can carry on!

Copy the code for the pop-up `Update` and `Render` functions from Listing 3.264.

```

function XPPopUp:Update(dt)
    self.mTween:Update(dt)

    if self.mTween:IsFinished() then
        self.mDisplayTime = math.min(5, self.mDisplayTime + dt)
    end
end

function XPPopUp:Render(renderer)
    local alpha = self.mTween:Value()

```

```

self.mTextColor:SetW(alpha)
self.mPane:SetColor(Vector.Create(1,1,1,alpha))

renderer:AlignText("center", "center")
renderer:ScaleText(1.4, 1.4)

local x = self.mX
local y = self.mY
local textSize = renderer:MeasureText(self.mText)

self.mPane:CenterPosition(x, y, textSize:X() + 24, textSize:Y() + 12)
self.mPane:Render(renderer)

renderer:DrawText2d(x, y, self.mText, self.mTextColor)
end

```

Listing 3.264: XPPopUp Update and Render. In XPPopUp.lua.

The Update function updates the transition tween. If the tween's finished, the `mDisplayTime` is incremented by the time elapsed for the current frame. The Render function stores the tween value in a variable called `alpha` which it uses to fade `mTextColor` and `mPane`. Text is center aligned and scaled up a little. The backing panel is fitted around the text with a little padding. The `mPane'` and `mText'` elements are rendered and that's the pop-up fully working.

Let's test it out in `main.lua`. Copy the code from Listing 3.265.

```

local popup = nil

function update()

if Keyboard.JustPressed(KEY_SPACE) then
    popup = XPPopUp>Create("Hello World", 0, 0)
    popup:TurnOn()
end

if popup then
    popup:Update(GetDeltaTime())
    popup:Render(gRenderer)

    if popup.mDisplayTime > 2 then
        popup:TurnOff()
        if popup:IsFinished() then
            popup = nil
        end
    end
end

```

```
    end
  end
end
```

Listing 3.265: Testing the XPPopUp code. In main.lua.

Run this code and you'll be able to see the pop-up in action. You should see something like Figure 3.52.



Figure 3.52: Testing out the pop-up code.

ActorXPSummary

Now that we have the pop-up working, let's implement the ActorXPSummary class. Most of the code deals with the layout.

ActorXPSummary fills in the actor summary panels on the XP summary screen. The ActorXPSummary constructor takes in a Layout object and a panel 'id'. It uses the layout to position its internal elements.



Figure 3.53: The layout for the XP summary element.

The ActorXPSummary is shown in Figure 3.53. I contains a small portrait of the actor, the name, current XP, XP required for the next level, and the current level as text. A progress bar shows the progress towards the next level. Pop-ups appear near the center of the summary. The summary manages any pop-ups, and queues them up if there's more than one. There's also code to skip through the pop-ups if the player wants to cut short the combat summary.

Let's jump straight in with the constructor. Copy the code from Listing 3.266.

```

ActorXPSummary = {}
ActorXPSummary.__index = ActorXPSummary
function ActorXPSummary:Create(actor, layout, layoutId)
    local this =
    {
        mActor = actor,
        mLayout = layout,
        mId = layoutId,

        mXPBar = ProgressBar:Create
        {
            value = actor.mXP,
            maximum = actor.mNextLevelXP,
            background = Texture.Find("xpbackground.png"),
            foreground = Texture.Find("xpforeground.png"),
        },
        mPopUpList = {},
        mPopUpDisplayTime = 1, -- seconds
    }

    this.mAvatar = Sprite.Create()
    this.mAvatar:SetTexture(this.mActor.mPortraitTexture)
    local avatarScale = 0.8
    this.mAvatar:SetScale(avatarScale, avatarScale)
}

```

```

    this.mActorWidth = actor.mPortraitTexture:GetWidth() * avatarScale

    setmetatable(this, self)

    return this
end

```

Listing 3.266: Actor XP Summary Constructor. In ActorXPSummary.lua.

The constructor for the ActorXPSummary takes in an actor, layout and layoutId. The actor parameter is the actor being summarised, and the layout information tells us where to render the summary. All three parameters are stored in the this table.

In the this table we set up the mXPBar, a progress bar that displays the progress to the next level. A list called mPopUpList stores the pop-ups. The mPopUpDisplayTime field controls how many seconds a pop-up is displayed before being removed.

After the this table is finished, we set up the mAvatar sprite to show the actor portrait. This sprite is scaled down to 80% size because the actor summary backing panels aren't going to be very tall. The pixel width of the portrait is saved in mActorWidth which we use to position some of the text elements.

That's the constructor done! Let's move on to the Render function. Copy the code from Listing 3.267.

```

function ActorXPSummary:Render(renderer)

    renderer:ScaleText(1.25, 1.25)
    -- portrait
    local left = self.mLayout:Left(self.mId)
    local midY = self.mLayout:MidY(self.mId)
    local avatarLeft = left + self.mActorWidth/2 + 6
    self.mAvatar:SetPosition(avatarLeft, midY)
    renderer:DrawSprite(self.mAvatar)

    -- Name
    local nameX = left + self.mActorWidth + 84
    local nameY = self.mLayout:Top(self.mId) - 12
    renderer:AlignText("right", "top")
    renderer:DrawText2d(nameX, nameY, self.mActor.mName)

    -- Level
    local strLevelLabel = "Level:"
    local strLevelValue = string.format("%d", self.mActor.mLevel)
    local levelY = nameY - 42
    renderer:DrawText2d(nameX, levelY, strLevelLabel)

```

```

renderer:AlignText("left", "top")
renderer:DrawText2d(nameX + 12, levelY, strLevelValue)

-- XP
renderer:AlignText("right", "top")
local strXPLabel = "EXP:"
local strXPValue = string.format("%d", self.mActor.mXP)
local right = self.mLayout:Right(self.mId) - 18
local rightLabel = right - 96
renderer:DrawText2d(rightLabel, nameY, strXPLabel)
renderer:DrawText2d(right, nameY, strXPValue)

local barX = right - self.mXPBar.mHalfWidth

self.mXPBar:SetPosition(barX, nameY - 24)
self.mXPBar:SetValue(self.mActor.mXP, self.mActor.mNextLevelXP)
self.mXPBar:Render(renderer)

local strNextLevelLabel = "Next Level:"
local strNextLevelValue = string.format("%d", self.mActor.mNextLevelXP)

renderer:DrawText2d(rightLabel, levelY, strNextLevelLabel)
renderer:DrawText2d(right, levelY, strNextLevelValue)

local popup = self.mPopUpList[1]
if popup == nil then
    return
end
popup:Render(renderer)
end

```

Listing 3.267: The Render function for the actor's XP summary. In ActorXPSummary.lua.

The Render function first draws the actor portrait on the left side of the backing panel. Sprites are drawn from the center. To position the portrait, we take the leftmost side of the layout and add half the sprite width plus a little padding. The sprite Y position uses the midpoint of the layout. The actor's name is rendered to the right of the portrait. To get the position, we take the portrait's x position and add half the portrait width, giving us the right edge of the portrait. Then we add a little padding and draw the name. The Y position for the name uses the top of the layout with a little padding.

Under the name, near the bottom of the panel, we draw the current level. If the player gains a level after combat, the level number increases. Increasing the level number may

change the text width, i.e. going from “Level: 9” to “Level: 10”. Increasing the width can cause an unpleasant shaking effect. To prevent any shaking, the text is broken into two parts, the text label and the numeric value. The label is aligned to the right and the value to the left. The “Level” label is drawn at the same X position as the actor name and the Y is set to near the bottom of the panel. The value for the level is taken from the `mActor` and is drawn with a little padding to the right of the label. That’s all there is to do the for left side of the panel. Let’s move on to the right.

On the right we draw a label, value pair for the XP. The current XP value comes from `mActor.mXP`. Beneath the XP value, we draw the XP progress bar. The progress bar is set up using the current XP and the XP required for the next level. Under the progress bar we draw the label and value for the amount of XP required for the next level. That’s it for the right hand side of the summary panel.

In the middle of the panel, we display any pop-up messages. We check `mPopUpList`. If there’s no entry then we return; otherwise we render the entry. Pop-ups are center aligned when they’re created, so we don’t need to do much positioning.

Let’s take add the pop-up utility functions next. Copy the code from Listing 3.268.

```
function ActorXPSummary:SetPosition(x, y)
    self.mX = x
    self.mY = y
end

function ActorXPSummary:AddPopUp(text, color)
    local x = self.mLayout:MidX(self.mId)
    local y = self.mLayout:MidY(self.mId)
    local popup = XPPopUp>Create(text, x, y, color)

    table.insert(self.mPopUpList, popup)
    popup:TurnOn()
end

function ActorXPSummary:PopUpCount()
    return #self.mPopUpList
end

function ActorXPSummary:CancelPopUp()
    local popup = self.mPopUpList[1]
    if popup == nil or popup:IsTurningOff() then
        return
    end
    popup:TurnOff()
end
```

Listing 3.268: Utility functions for the Actor XP Summary. In ActorXPSummary.lua.

The SetPosition function sets the position of the summary.

AddPopUp creates a new pop-up and puts it in the queue. There are two parameters, the text to display and an optional color value for the text. The text color for pop-ups defaults to yellow. Pop-ups appear in the center of the panel, and we get this value using the layout object. The pop-up is added to the queue, turned on, and waits for its Update to be called.

The PopUpCount function tells us how many pop-ups are in the queue. This information is useful for when the player wants to skip through the combat summary but we want to ensure they've seen all the relevant messages.

The CancelPopup function gets the front pop-up in the queue and if it's not already turning off, turns it off. These functions are used by the XPSummaryState.

Let's finish the class by adding the Update function. Copy the code from Listing 3.269.

```
function ActorXPSummary:Update(dt)

    local popup = self.mPopUpList[1]
    if popup == nil then
        return
    end

    if popup:IsFinished() then
        table.remove(self.mPopUpList, 1)
        return
    end

    popup:Update(dt)

    if popup.mDisplayTime > self.mPopUpDisplayTime
    and #self.mPopUpList > 1 then
        popup:TurnOff()
    end

end
```

Listing 3.269: The Actor XP Summary Update function. In ActorXPSummary.lua.

The ActorXPSummary:Update function is only concerned with the pop-ups. Everything else on the summary is static. It tries to get the front pop-up in the queue. If no pop-up exists, it returns. Otherwise, it checks if the pop-up has finished and if so removes it from the queue, then returns. If the pop-up hasn't finished, it's updated, then there's a check to see how long the pop-up has been displaying. If the pop-up has been displayed for greater than mPopUpDisplayTime seconds and there are more pop-ups in the queue, it's turned off. This means the very last pop-up will hang around and not turn off.

Let's see this summary in action by adding a few temporary modifications to the main.lua file. Make the edits shown in Listing 3.270.

```
local layout = Layout>Create()
layout.mPanels['test'] = {x=0,y=41, width=522, height=77}
local actorHero = gWorld.mParty.mMembers['hero']
local summary = ActorXPSummary>Create(actorHero, layout, 'test')
summary:AddPopUp("Level Up!")
function update()
    summary:Update(GetDeltaTime())
    summary:Render(gRenderer)
end
```

Listing 3.270: Testing the ActorXPSummary. In main.lua.

Run this code and you should see something similar to Figure 3.54, the actor summary drawn out with no backing panel. But it's enough to see everything is working correctly. Let's move on to implement the XPSummaryState.

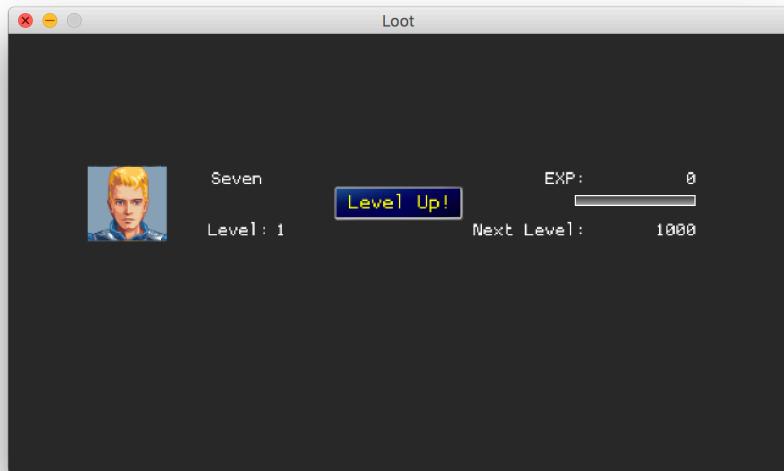


Figure 3.54: Testing the ActorXPSummary class.

XPSummaryState Implementation

Time to bring everything together! First let's set up the main.lua by pushing the XPSummaryState directly onto the stack. This lets us quickly check the changes to the

screen as we edit the code. Copy the code from Listing 3.271.

```
-- code omitted

gStack:Push(CombatState>Create(gStack, gCombatDef))
gStack:Push(XPSummaryState>Create(gStack, gWorld.mParty, {xp = 30}))
```

Listing 3.271: Pusing XPSummaryState on the stack to see the layout. In main.lua.

When laying out the combat summary screen we need to know who's in the party, so we pass in the gWorld.mParty object. The third parameter describes the combat result; here we're saying the party receives 30 XP. This result table will have more information later when it's provided by the combat state.

Open up XPSummaryState.lua and copy in the code from Listing 3.272.

```
XPSummaryState = {}
XPSummaryState.__index = XPSummaryState
function XPSummaryState>Create(stack, party, combatData)
    local this =
    {
        mStack = stack,
        mCombatData = combatData,
        mLayout = Layout>Create(),
        mXP = combatData.xp,
        mXPPerSec = 5.0,
        mXPCounter = 0,
        mIsCountingXP = true,
        mParty = party:ToArray()
    }

    local digitNumber = math.log10(this.mXP + 1)
    this.mXPPerSec = this.mXPPerSec * (digitNumber ^ digitNumber)

    this.mLayout:Contract('screen', 118, 40)
    this.mLayout:SplitHorz('screen', 'top', 'bottom', 0.5, 2)
    this.mLayout:SplitHorz('top', 'top', 'one', 0.5, 2)
    this.mLayout:SplitHorz('bottom', 'two', 'three', 0.5, 2)

    this.mLayout:SplitHorz('top', 'title', 'detail', 0.5, 2)

    this.mTitlePanels =
    {
        this.mLayout>CreatePanel("title"),
        this.mLayout>CreatePanel("detail"),
```

```

    }

    this.mActorPanels =
    {
        this.mLayout>CreatePanel("one"),
        this.mLayout>CreatePanel("two"),
        this.mLayout>CreatePanel("three"),
    }

    this.mPartySummary = {}

    local summaryLeft = this.mLayout:Left('detail') + 16
    local index = 1
    local panelIds = {"one", "two", "three"}
    for _, v in ipairs(this.mParty) do
        local panelId = panelIds[index]
        print(panelId)
        local summary = ActorXPSummary>Create(v, this.mLayout, panelId)
        -- local summaryTop = this.mLayout:Top(panelId)
        -- summary:SetPosition(summaryLeft, summaryTop)
        table.insert(this.mPartySummary, summary)
        index = index + 1
    end

    Apply(this.mLayout.mPanels["one"], print, pairs)

    setmetatable(this, self)
    return this
end

-- Other State functions are empty for now
function XPSummaryState:Enter() end
function XPSummaryState:Exit() end
function XPSummaryState:Update(dt) end
function XPSummaryState:Render(renderer) end
function XPSummaryState:HandleInput() end

```

Listing 3.272: XPSummaryState Constructor, using panels to get the layout. In XPSummaryState.lua.

There's a lot going on in this constructor, but the main body of the code just lays out backing panels. The constructor sets up values to count out XP to the party members. Let's go through and see what each part is doing.

In the this table, we add a reference to the stack and combatData. Then we create

`mLayout`, which is a layout object for our panels, and `mParty`, which is an array of the actors in the party.

The rest of the `thisTable` is concerned with counting out the experience points. The total amount of XP is awarded to each actor⁷. XP is awarded to the actors over a few seconds, but the player can skip this award process by pressing Spacebar.

Here are the fields used to give out the XP:

mXP The experience points to give out.

mXPPerSec The experience points given out to the actors per second.

mXPCounter Used to count up XP to award. This stores fractional experience points to allow small XP amounts to be awarded over long amounts of time.

mIsCountingXP A flag specifying if the state is currently counting out the experience points to the player.

The `mXPCounter` might be a little confusing, so let's do a quick example.

XP is only ever a whole number. You never have 1.5xp. But while we're counting out XP, it's useful for us to say 0.1xp was counted out this frame. The `mXPCounter` holds these fractional amounts of XP until they grow large enough to be awarded. With our current numbers, 5 XP are counted out per second, and let's say the game is running at 60 frames per second. Every frame, $5/60 = 0.08333$ XP is counted out. This isn't a whole number, so it's just put into `mXPCounter`.

Here's how the `mXPCounter` value changes during the first few frames.

```
Frame 01: mXPCounter = 0.08333
Frame 02: mXPCounter = 0.16666
Frame 03: mXPCounter = 0.25
Frame 04: mXPCounter = 0.33333
...
Frame 12: mXPCounter = 1
```

At frame 12, the XP point is awarded to the actors and `mXPCounter` is reset. We'll get to the actual code as we write the update loop, but there's an overview for now.

After setting up the `thisTable`, we do a little math! Don't worry, it's more straightforward than it appears. An RPG gives out small amounts of XP in the early game and larger amounts later on. This means we use the `XPPerSec` value for massively different amounts of XP. If we count out 30xp in 5 seconds, that means that when we reward

⁷XP awards differ from game to game. I've chosen a simple approach of giving the full total to each character. Other games may divide the XP among the surviving characters or even award the XP based on in-game actions such as who killed the most monsters etc.

1000xp it takes nearly 3 minutes to count out! The count speed needs to scale according to the amount of XP. Log10 gets the number of digits - 1 in a number, so 1000 gives 3, 100 gives 2, 10 gives 1, and 1 gives 0. Actually the number it returns is a float. The remainder tells us how close we are to getting to another digit on the number. We use the length of the number to vary the count out time. The larger the number, the faster we award XP.

The base count speed is 5xp per second. We multiple this base by the length of the total XP as a power to itself. This means the speed gets exponentially faster as the length of the number increases. It seems to work quite well. When we've finished implementing this class, you can try it out with a few different values.

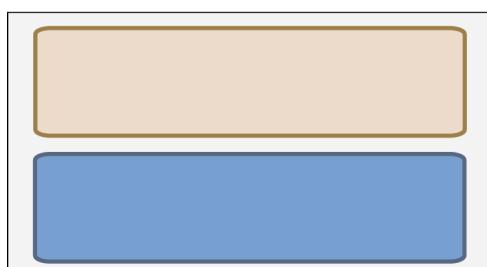
After `mXPPerSec` is assigned, we lay out the panels for the screen. To create the panels, we first take a screen-sized panel and contract it a little to give us some padding. Then we split the panel equally across the middle, horizontally. Then we split these equally-sized panels horizontally once more to give us four equally-sized panels taking up most of the screen. The bottom three of these panels display the xp summaries. The top panel we split again across the middle. We use the upper panel for the title and the lower to report how much XP was earned. You can see the steps of the layout operations in Figure 3.55.



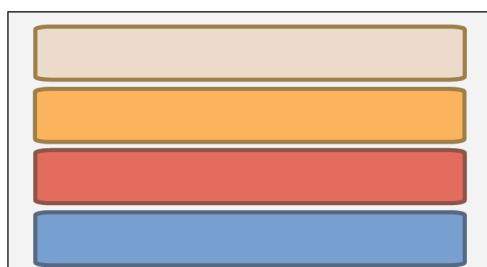
Begin with a
full screen
rectangle



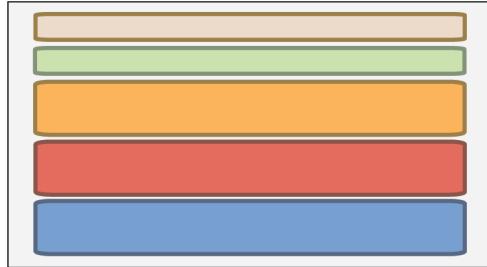
Contract 10%



Split horizontal
at 50%



Split both panels
horizontal 50%



Split top panel
horizontal 50%

Figure 3.55: Creating the layout for the XPSummaryState.

In the end we have five panels named "title", "detail", "one", "two" and "three". The "title" and "detail" panels are created and stored in the `mTitlePanels` table using the `mLayout`. The same is done for the remaining panels, but we store them in the `mActorPanels` table. These two tables make it easier to render the panels in the `Render` function.

The end of the constructor loops through the actors, creates an `ActorXPSummary` object for each one, and stores them all in the `mPartySummary` table. Each `ActorXPSummary` object is constructed using the layout and an id. We create a table 'panelIds' to make sure each actor gets the correct layout id.

Let's implement the rest of the class. Copy the code from Listing 3.273.

```
function XPSummaryState:Enter()
    self.mIsCountingXP = true
    self.mXPCounter = 0
end

function XPSummaryState:Exit()
end
```

Listing 3.273: The Enter and Exit function. In XPSummaryState.lua.

The Enter and Exit functions for `XPSummaryState` don't do much. The Enter function sets the `mIsCountingFlag` to true and resets the XP counter to zero. This makes the state ready to count out the XP. The Exit function has no code.

Let's implement the Update function next. Copy the code from Listing 3.274.

```
function XPSummaryState:ApplyXPToParty(xp)
    for k, actor in pairs(self.mParty) do

        if actor.mStats:Get("hp_now") > 0 then

            local summary = self.mPartySummary[k]

            actor:AddXP(xp)
            while(actor:ReadyToLevelUp()) do
                local levelup = actor>CreateLevelUp()
                local levelNumber = actor.mLevel + levelup.level
                summary:AddPopUp("Level Up!")
            end
        end
    end
end
```

```

-- check levelup for any special messages
-- to display as popups here

    actor:ApplyLevel(levelup)
end

end
end

function XPSummaryState:Update(dt)

    for k, v in ipairs(self.mPartySummary) do
        v:Update(dt)
    end

    if self.mIsCountingXP then

        self.mXPCounter = self.mXPCounter + self.mXPPerSec * dt
        local xpToApply = math.floor(self.mXPCounter)
        self.mXPCounter = self.mXPCounter - xpToApply
        self.mXP = self.mXP - xpToApply

        self:ApplyXPToParty(xpToApply)

        if self.mXP == 0 then
            self.mIsCountingXP = false
        end

        return
    end
end

```

Listing 3.274: The ApplyXPToParty and Update functions. In XPSummaryState.lua.

Let's begin by looking at the Update function. It first calls update on all the actor summaries. These updates handle the pop-up displays.

If the mIsCountingXP flag is true then the Update function counts out the XP to all the actors. First the mXPCounter is increased by the number of XP to be counted out per second for this frame. This number is then floored, leaving the whole number and storing it in xpToApply. The mXPCounter is reduced by the xpToApply, as it only stores fractions of experience points. The xpToApply is often zero for small XP gains. The mXP is the pool of XP we're giving out to the actors. This is reduced by xpToApply, then the

xpToApply points are given to the party. Finally there's a check to see if any experience points are left to give out. If not, the mIsCountingXP flag is set to false.

The ApplyXPToParty function gives XP to the actors. If an actor is dead they get no XP. If a lot of XP is added at once and the actor is very low level they might gain multiple levels at once! To handle this case, we use a while loop that continues to loop as long as the actor has new levels to gain. Each level is applied to the actor one by one. If an actor can level up, a level up object is created, the actor's level number is incremented, and a "Level Up" pop-up is sent to the actor summary. Then the level object is applied to the actor.

Counting out the XP points looks cool, but after the player has seen it a few times, they'll probably want a shortcut to skip over it. We'll handle this in the HandleUpdate function. Copy the code from Listing 3.275.

```
function XPSummaryState:ArePopUpsRemaining()
    for k, v in ipairs(self.mPartySummary) do
        if v:PopUpCount() > 0 then
            return true
        end
    end
    return false
end

function XPSummaryState:CloseNextPopUp()
    for k, v in ipairs(self.mPartySummary) do
        if v:PopUpCount() > 0 then
            v:CancelPopUp()
        end
    end
end

function XPSummaryState:SkipCountingXP()
    self.mIsCountingXP = false
    self.mXPCounter = 0
    local xpToApply = self.mXP
    self.mXP = 0
    self:ApplyXPToParty(xpToApply)
end

function XPSummaryState:HandleInput()
    if Keyboard.JustPressed(KEY_SPACE) then

        if self.mXP > 0 then
            self:SkipCountingXP()
            return
        end
    end
```

```

    if self:ArePopUpsRemaining() then
        self:CloseNextPopUp()
        return
    end

    self.mStack:Pop()
end
end

```

Listing 3.275: Handle Update and XP Counting Skipping. In XPSummaryState.lua.

HandleInput checks if Spacebar is pressed, which means to skip the XP handout. If it should be skipped or if it is finished, the program advances to the next combat summary screen. If Spacebar is pressed, the first check sees if there's still XP to hand out, in which case SkipCountingXP is called. SkipCountingXP hands out all the remaining XP in one frame. Any active pop-ups are also closed, but new ones are still allowed to appear. This makes sure the player gets to see any important messages about party members levelling up. If there are no more pop-ups, then we pop the XPSummaryScreen off the stack.

The SkipCountingXP function sets mIsCounting to false and the mXPCounter to zero. The remaining XP is added to the party and mXP, the XP to handout, is set to zero.

'CloseNextPopUp' iterates through the actor summaries. If there are any pop-ups, then they're closed. 'ArePopUpsRemaining' returns true if any of the actor summaries have a pop-up and false otherwise.

There's only one function left to add, the Render function. Let's add it now. Copy the code from Listing 3.276.

```

function XPSummaryState:Render(renderer)

    renderer:DrawRect2d(System.ScreenTopLeft(),
                        System.ScreenBottomRight(),
                        Vector.Create(0,0,0,1))

    for k,v in ipairs(self.mTitlePanels) do
        v:Render(renderer)
    end

    local titleX = self.mLayout:MidX('title')
    local titleY = self.mLayout:MidY('title')
    renderer:ScaleText(1.5, 1.5)
    renderer:AlignText("center", "center")
    renderer:DrawText2d(titleX, titleY, "Experience Increased!")

```

```

local xp = self.mXP
local detailX = self.mLayout:Left('detail') + 16
local detailY = self.mLayout:MidY('detail')
renderer:ScaleText(1.25, 1.25)
renderer:AlignText("left", "center")
local detailStr = string.format('XP increased by %d.', xp)
renderer:DrawText2d(detailX, detailY, detailStr)

for i = 1, #self.mPartySummary do
    self.mActorPanels[i]:Render(renderer)
    self.mPartySummary[i]:Render(renderer)
end

end

```

Listing 3.276: Rendering out the XP Summary. In XPSummaryState.lua.

First we draw a full screen black rectangle as a background, then the title and detail panels. In the center of the title panel we draw the title “Experience Increased!”, centered, at a scale of 1.6. The detail panel reports how much XP has been awarded. This is left aligned in the panel at a lower scale of 1.25. Finally the party summaries are rendered.

That’s it. We’re ready to run. Try it out. You’ll see something like Figure 3.56.



Figure 3.56: The XP summary screen.

The full code so far is available in `loot-2-solution`. Run it and try some different XP values to see how it performs. Next is the second combat summary screen; the loot!

LootSummaryState

Enemies don't just give you experience. They also drop loot and money. The `LootSummaryState` shows exactly what loot was dropped during the battle.

We'll start by changing the main file once more so the `LootSummaryTable` is displayed immediately. Copy the code from Listing 3.277.

```
gStack:Push(ExploreState>Create(gStack, CreateArenaMap(),
    Vector.Create(30, 18, 1)))
gStack:Push(CombatState>Create(gStack, gCombatDef))

gStack:Push(LootSummaryState>Create(gStack, gWorld,
{
    xp = 30,
    gold = 100,
    loot =
    {
        { id = 1, count = 2 },
        { id = 2, count = 1 },
        { id = 3, count = 1 },
    }
}))
```

`function update()`

Listing 3.277: Pushing the LootSummaryState on the stack. In `main.lua`.

The `LootSummaryState` takes in an expanded version of the `combatData` table; we now include the amount of gold found and a loot table. The loot table is a list of item ids and counts, so if you find 3 potions, they get stacked together. With this information, we're ready to write `LootSummaryState`. It's similar to but less complex than the `XPSummaryState`. Instead of counting XP, we count gold. The items we find are added to the party inventory on first entering the state.

The `LootSummary` state is broken into three horizontal panels: a title panel announcing the party has found loot, a small panel showing the gold found along with the party's current gold, and a much bigger panel showing items found. When the state starts, the found gold counts down and the party gold counts up. The player can skip this count

out by pressing Spacebar. Once it's counted out, if the player presses Spacebar again, the state closes and is popped off the stack.

Let's start with the constructor. Copy the code from Listing 3.278.

```
LootSummaryState = {}
LootSummaryState.__index = LootSummaryState
function LootSummaryState:create(stack, world, combatData)
    local this =
    {
        mStack = stack,
        mWorld = world,
        mLoot = combatData.loot or {},
        mGold = combatData.gold or 0,

        mLAYOUT = Layout:create(),
        mPanels = {},

        mGoldPerSec = 5.0,
        mGoldCounter = 0,
        mIsCountingGold = true,
    }

    local digitNumber = math.log10(this.mGold + 1)
    this.mGoldPerSec = this.mGoldPerSec * (digitNumber ^ digitNumber)

    this.mLayout:Contract('screen', 118, 40)
    this.mLayout:SplitHorz('screen', 'top', 'bottom', 0.25, 2)
    this.mLayout:SplitHorz('top', 'title', 'detail', 0.55, 2)
    this.mLayout:SplitVert('detail', 'left', 'right', 0.5, 1)
    this.mPanels =
    {
        this.mLayout>CreatePanel('title'),
        this.mLayout>CreatePanel('left'),
        this.mLayout>CreatePanel('right'),
        this.mLayout>CreatePanel('bottom'),
    }

    setmetatable(this, self)

    this.mLootView = Selection:create
    {
        data = combatData.loot,
        spacingX = 175,
        columns = 3,
        rows = 9,
```

```

    RenderItem = function(self, ...) this:RenderItem(...) end
}

local lootX = this.mLayout:Left('bottom') - 16
local lootY = this.mLayout:Top('bottom') - 16
this.mLootView:SetPosition(lootX, lootY)
this.mLootView:HideCursor()

return this
end

```

Listing 3.278: The constructor for the LootSummaryState. In LootSummaryState.lua.

The constructor stores its stack and world parameters in the this table. The combatData table parameter isn't stored directly, but its gold value is stored in mGold and its loot table is stored in mLoot. If the combatData table doesn't have a gold value, it defaults to 0. If there's no loot table, it defaults to an empty table.

Next a layout object is created in mLLayout and an empty table mPanels is created. These store the layout of the backing panels. Then finally mGoldPerSec, mGoldCounter and mIsCountingGold fields. These are used to count out the gold to the party.

With the this table set up, let's move on to the rest of the constructor. First we take the number of digits in the amount of gold to distribute and use that to determine how fast the gold should be counted out. We store this information in mGoldPerSec the same way we calculated mXPPerSec in the previous state.

Next we split the screen into panels, starting with one big screen-sized panel. We contract this initial panel to get some padding, then split it in two; top and bottom. Top takes up 25% of the space, bottom 75%. Top is then split again into two layers, "title" and "detail". The detail pane is split horizontally down the center, giving us four panels: "title", "left", "right" and "bottom". We store these panels in the mPanels table. You can see the order of operations in Figure 3.57.



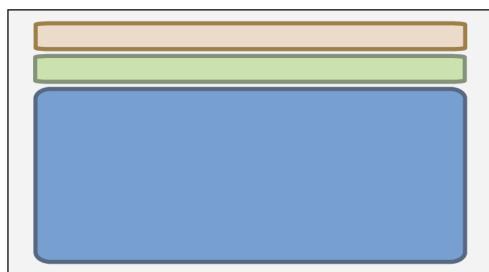
Begin with a
full screen
rectangle



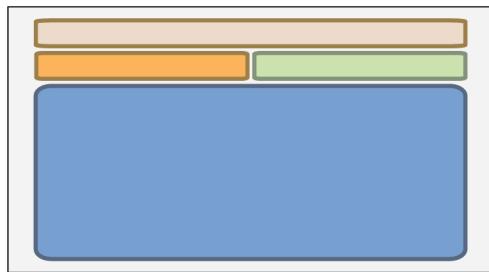
Contract 10%



Split horizontal
at 25%



Split top panel
horizontal 50%



Split detail panel
vertical 50%

Figure 3.57: Creating the panels for the loot summary screen.

Next we set the metatable, which hooks all the functions to the this table. Then we're ready to create the `mLootView`, a selection menu that displays all the loot items in the bottom panel. The `mLootView` selection takes the `combatData.loot` table as its datasource. We set it to have 3 columns and 9 rows, which fits the bottom panel nicely. Each column has 175 pixels of spacing. Each item in the `mLootView` is rendered by a `RenderItem` function which we'll implement shortly.

Finally we set the position of the `mLootView` selection to top left of the bottom panel with a little adjustment, and we hide the cursor. We never select anything in the `mLootView`, so we don't need the cursor. The menu only displays the loot. Let's add the `RenderItem`, `Enter`, and `Exit` functions next. Copy the code from Listing 3.279.

```
function LootSummaryState:RenderItem(renderer, x, y, item)
    if not item then
        return
    end

    local def = ItemDB[item.id]
    local text = def.name

    if item.count > 1 then
        text = string.format("%s x%d", text, item.count)
    end

    renderer:DrawText2d(x, y, text)
end

function LootSummaryState:Enter()
    self.mIsCountingGold = true
    self.mGoldCounter = 0

    -- Add the items to the inventory.
    for k, v in ipairs(self.mLoot) do
        self.mWorld:AddItem(v.id, v.count)
    end
end

function LootSummaryState:Exit()
end
```

Listing 3.279: RenderItem, Enter and Exit functions. In LootSummaryState.lua.

The RenderItem function takes an item parameter, which is a table with an id and count field. We use the id to look up the item definition in the ItemDB table. From the definition, we get an item name. If the item has a count of 1, then the name is rendered out. If the count is greater than 1, then we render it like "ItemName x2" replacing 2 with the count number. This keeps the loot screen concise when the party receives multiples of the same item.

The Enter function resets some of the gold counters and also adds all the items from the loot table into the inventory. The Exit function does nothing.

Next let's implement the Render and Update functions. Copy the code from Listing 3.280.

```
function LootSummaryState:Update(dt)

    if self.mIsCountingGold then

        self.mGoldCounter = self.mGoldCounter + self.mGoldPerSec * dt
        local goldToGive = math.floor(self.mGoldCounter)
        self.mGoldCounter = self.mGoldCounter - goldToGive
        self.mGold = self.mGold - goldToGive

        self.mWorld.mGold = self.mWorld.mGold + goldToGive

        if self.mGold == 0 then
            self.mIsCountingGold = false
        end

        return
    end
end

function LootSummaryState:Render(renderer)

    renderer:DrawRect2d(System.ScreenTopLeft(),
                        System.ScreenBottomRight(),
                        Vector.Create(0,0,0,1))

    for _, v in pairs(self.mPanels) do
        v:Render(renderer)
    end

    local titleX = self.mLayout:MidX('title')
    local titleY = self.mLayout:MidY('title')
    renderer:ScaleText(1.5, 1.5)
    renderer:AlignText("center", "center")
```

```

    renderer:DrawText2d(titleX, titleY, "Found Loot!")

    renderer:AlignText("left", "center")
    renderer:ScaleText(1.25, 1.25)
    local leftX = self.mLayout:Left('left') + 12
    local leftValueX = self.mLayout:Right('left') - 12
    local leftY = self.mLayout:MidY('left')
    local goldLabelStr = "Gold Found:"
    local goldValueStr = string.format("%d gp", self.mGold)
    renderer:DrawText2d(leftX, leftY, goldLabelStr)
    renderer:AlignText("right", "center")
    renderer:DrawText2d(leftValueX, leftY, goldValueStr)

    renderer:AlignText("left", "center")
    local rightX = self.mLayout:Left('right') + 12
    local rightValueX = self.mLayout:Right('right') - 12
    local rightY = leftY
    local partyGPStr = string.format("%d gp", self.mWorld.mGold)
    renderer:DrawText2d(rightX, rightY, "Party Gold:")
    renderer:AlignText("right", "center")
    renderer:DrawText2d(rightValueX, rightY, partyGPStr)

    renderer:AlignText("left", "top")
    self.mLootView:Render(renderer)
end

```

Listing 3.280: Update and Render functions for LootSummaryState. In LootSummaryState.lua.

The Update function is similar to the one in XPSummaryState, but instead of increasing XP and handling levels it just increases the party's gold value in the mWorld table. We could have written a CountOut.lua class to handle this kind of counting, as the code is almost identical, but it only occurs in these two places and there's enough difference that it's simpler in this case just to have some similar code.

The Render function draws out all the backing panels using the mPanels table. It then gets the center of the title panel and renders the text "Found Loot!" center aligned at 1.5x default scale. The text for *gold found* and *party gold* are each broken into label and value pairs. The labels are drawn at the left side of the panels, left aligned, the values are drawn on the right side, right aligned. The gold value has "gp" appended to the end of the string, which stands for *gold pieces*. Finally the mLootView is told to render out all the items. The loot is rendered over the bottom panel.

That leaves the HandleInput function to implement. Copy the code from Listing 3.281.

```

function LootSummaryState:SkipCountingGold()
    self.mIsCountingGold = false
    self.mGoldCounter = 0
    local goldToGive = self.mGold
    self.mGold = 0
    self.mWorld.mGold = self.mWorld.mGold + goldToGive
end

function LootSummaryState:HandleInput()
    if Keyboard.JustPressed(KEY_SPACE) then

        if self.mGold > 0 then
            self:SkipCountingGold()
            return
        end

        self.mStack:Pop()
    end
end

```

Listing 3.281: Adding the HandleInput function. In LootSummaryState.lua.

HandleInput detects if the user presses Spacebar. If it's pressed while gold is being counted out, the counting is immediately finished and the function returns. If gold has finished being counted out, then pressing Spacebar pops the LootSummaryState off the stack. The SkipCountingGold function is responsible for stopping the gold count and giving all the gold to the player. It works in the same manner as SkipCountingXP so there's no need go over it again.

Run your code. You should see the loot screen appear as shown in Figure 3.58, display the loot we've found, and start counting out the gold. Cool! The code so far is available in `loot-3-solution`.



Figure 3.58: The loot summary screen.

At this point we've got both parts of the combat summary screens, and it's just a matter of hooking them into the game flow.

Putting it all Together

Let's integrate the summary screens into the game flow. First we'll make it so the win condition correctly triggers the combat summary screens and transitions between them. Example loot-4 contains all the code we've written so far.

Let's clean up the main.lua file so only the CombatState.lua is loaded, and also reduce the enemies to one. Copy the code from Listing 3.282.

```
gCombatDef =  
{  
    background = "arena_background.png",  
    actors =  
    {  
        party = gWorld.mParty:ToArray(),  
        enemy = { Actor>Create(gEnemyDefs.grunt) }  
    }  
}
```

```

gStack:Push(ExploreState>Create(gStack, CreateArenaMap(),
    Vector.Create(30, 18, 1)))
gStack:Push(CombatState>Create(gStack, gCombatDef))

function update()
    local dt = GetDeltaTime()
    gStack:Update(dt)
    gStack:Render(gRenderer)
    gWorld:Update(dt)
end

```

Listing 3.282: An updated main file. In main.lua.

The loot-4 project has some additional changes. In the PartyMemberDefs.lua all the party members have had their HP restored to 200. In the EnemyDefs.lua the goblin speed has been reduced to a more manageable 5. These changes make it easier to win quickly and view the combat summary screens.

The CombatState's OnWin function causes the screen to fade out and then fade in to the XPSummaryState. The XPSummaryState will then fade out and fade in to the LootSummaryState which in turn fades back to the game. The fades are handled using a storyboard. Here's the OnWin function for CombatState. Copy the code from Listing 3.283.

```

function CombatState:OnWin()

    -- Tell all living party members to dance.
    for k, v in ipairs(self.mActors['party']) do

        local char = self.mActorCharMap[v]
        alive = v.mStats:Get("hp_now") > 0
        if alive then
            char.mController:Change(CSRunAnim.mName, {'victory'})
        end
    end

    -- Create the storyboard and add the stats.
    local combatData = self:CalcCombatData()
    local xpSummaryState = XPSummaryState>Create(self.mGameStack,
                                                gWorld.mParty,
                                                combatData)
    local storyboard =
    {

```

```

        SOP.UpdateState(self, 1.0),
        SOP.BlackScreen("black", 0),
        SOP.FadeInScreen("black", 0.6),
        SOP.ReplaceState(self, xpSummaryState),
        SOP.Wait(0.3),
        SOP.FadeOutScreen("black", 0.3),
    }

    local storyboard = Storyboard>Create(self.mGameStack, storyboard)
    self.mGameStack:Push(storyboard)
    self.mIsFinishing = true
end

function CombatState:CalcCombatData()
    -- Todo: Work out loot, xp and gold drops
    return {
        xp = 30,
        gold = 10,
        loot =
        {
            { id = 1, count = 1 }
        }
    }
end

```

*Listing 3.283: Code to move to the XPSummaryState when the player wins a battle.
In CombatState.lua.*

When the player wins, the living party members do a victory dance. This is done by iterating through the actors and telling them to play the victory animation. Dead characters don't dance!

The combatData is created by calling CalcCombatData. For now this function returns a hard coded table. We'll implement this function properly when we have a system to handle dropping loot. The combatData table is used to create the XPSummaryState. We use a storyboard to give the player time to see their party members dance before fading out.

The storyboard's first operation is to update the CombatState for 1.0 second, giving the player enough time to see the characters dance. A black screen is added to the storyboard with an alpha of 0. This screen is faded in over 0.6 seconds. When the screen is fully black, the CombatState is swapped with the XPSummaryState we created. There's then a wait of 0.3 seconds on the black screen before fading into the XPSummaryState. Then the storyboard ends and pops itself off the stack.

We create a storyboard object from the storyboard table and push it onto the stack. In the next frame, the storyboard starts to play.

Currently the player never sees the LootSummaryState. We'll fix this in the XPSummaryState. Copy the code from Listing 3.284.

```
function XPSummaryState:HandleInput()
    if Keyboard.JustPressed(KEY_SPACE) then

        -- code omitted

        -- self.mStack:Pop()
        self:GotoLootSummary()
    end
end
```

Listing 3.284: Changing the stack pop with a GotoLootSummary function call. In XPSummaryState.lua.

In HandleInput we've removed the immediate stack pop and called a new function GotoLootSummary. This sets up a storyboard. Copy the code for GotoLootSummary from Listing 3.285.

```
function XPSummaryState:GotoLootSummary()
    local lootSummaryState = LootSummaryState>Create(self.mStack,
                                                    gWorld,
                                                    self.mCombatData)

    local storyboard =
    {
        SOP.BlackScreen("black", 0),
        SOP.FadeInScreen("black", 0.2),
        SOP.ReplaceState(self, lootSummaryState),
        SOP.Wait(0.1),
        SOP.FadeOutScreen("black", 0.2),
    }

    local storyboard = Storyboard>Create(self.mStack, storyboard)
    self.mStack:Push(storyboard)
end
```

Listing 3.285: The GotoLootSummary function. In XPSummaryState.lua.

We create the LootSummary by passing in the stack, world and combatData. This story is similar to our earlier transitions; we fade to black, swap the state, wait, and then fade back in. This time the state we're swapping to is the LootSummaryState.

In the LootSummaryState we'll use one more transition to take us back to the game. Copy the code from Listing 3.286.

```
function LootSummaryState:HandleInput()
    if Keyboard.JustPressed(KEY_SPACE) then

        if self.mGold > 0 then
            self:SkipCountingGold()
            return
        end

        self.mStack:Pop()
        local storyboard =
        {
            SOP.BlackScreen("black", 1),
            SOP.Wait(0.1),
            SOP.FadeOutScreen("black", 0.2),
        }
        local storyboard = Storyboard.Create(self.mStack, storyboard)
        storyboard:Update(0) -- creates the black screen this frame
        self.mStack:Push(storyboard)
    end
end
```

Listing 3.286: Another transition to return to the ExploreState. In LootSummaryState.lua.

When the player presses Spacebar to exit the LootSummaryState, the state is popped, and a new storyboard is created and pushed onto the stack. The storyboard immediately puts up a black screen with full opacity. This stays for 0.1 seconds, then fades over 0.2 seconds. Before pushing the storyboard onto the stack Update is called once. This makes the storyboard operations get processed in the same frame and ensures that the player sees the black screen immediately. If this wasn't done, for a single frame the player would see the ExploreState.

That's the combat states hooked up! The full code is in `loot-4-solution`. Give it a go. The next step is getting in some sweet loot drops.

Loot Tables

Let's take a step back for a moment and talk about *loot tables*. In tabletop RPGs, tables are used to semi-randomly determine all sorts of things, such as combat. For instance, say you're deep in a dungeon, you turn a corner, and there's a monster! But what kind of monster? Roll a die and look up the number on the table and it will tell you; an orc, a dragon, etc.

Here's an encounter table for a sci-fi setting. Roll 2d6, then look up the encounter on the table.

Roll	Encounter
2-3	1d6 Security Robots
4-6	1 Synthetic
7-11	1 Synthwarrior
12	1 Synthetic + 1d3 Security Robots

For tabletop games, if you find a chest, a loot table tells you how much gold is in it and if there are any gems or magic artifacts. It would be great to add these tables to our game, so let's consider a naive implementation.

Chance	Loot
10%	Rare Gem
10%	Vial of Goop
12%	Ancient Book
68%	20 Gold Pieces

Here's our table of things we might find in a chest. It's pretty easy to run in code. We get a random number from 1-100, then look up the piece of loot to return from our table. Say we get 21; then we'd skip the rare gem and the vial of goop with the first 20 points and the final point would make it so we return the "Ancient Book". This works great!

But wait... before we implement the code, let's try updating the table. How about adding a magic sword with a 1% chance of appearing? Oh wait, that's a problem! If we want to add the sword, we must alter *all* the chances of every other item in the table – that is going to be a pain – surely there's an easier way!

Oddment Table to the Rescue

Enter the oddment table.

Oddment	Loot
1	Sword
10	Rare Gem
10	Vial of Goop

Oddment	Loot
12	Ancient Book
68	20 Gold Pieces

Instead of a percentage chance, we use a number called an *oddment*. The lower the number, the lower the chance that item is selected. This way, items can be added to and removed from the table without having to adjust the other numbers. They're adjusted automatically.

You might be able to guess how it works. We take a random number from zero to all the oddments summed, then go through the items adding each oddment until we reach or pass the random number.

See Listing 3.287 to demonstrate how the oddment table is used in code.

```
local ot = OddmentTable:Create
{
    { oddment = 1, item = "sword" },
    { oddment = 1, item = "book" }
}
```

Listing 3.287: Example use of the Oddment Table.

This example shows a table with two entries. Both have an oddment of 1, so there's 50% chance either gets picked. When an item is picked, the string "sword" or "book" is returned.

There's a simple project called table-1 in the example folder. Open that and add Listing 3.288 to OddmentTable.lua.

```
OddmentTable = {}
OddmentTable.__index = OddmentTable
function OddmentTable:Create(items)
    local this =
    {
        mItems = items or {}
    }

    setmetatable(this, self)

    this.mOddment = this:CalcOddment()

    return this

```

```

end

function OddmentTable:CalcOddment()
    local total = 0
    for _, v in ipairs(self.mItems) do
        total = total + v.oddment
    end
    return total
end

function OddmentTable:Pick()

    local n = math.random(self.mOddment) -- 1 - oddment
    local total = 0
    for _, v in ipairs(self.mItems) do
        total = total + v.oddment

        if total >= n then
            return v.item
        end
    end

    -- Otherwise return the last item
    local last = self.mItems[#self.mItems]
    last = last or {}
    return last.item
end

```

Listing 3.288: Oddment Table Implementation. In OddmentTable.lua.

The OddmentTable takes in one parameter, items. Each entry in the items table has two keys: oddment, which is the chance value, and item, which is the actual data.

The items table is assigned to mItems and the metatable is set. The mOddment is assigned using the CalcOddment function. CalcOddment iterates through the entries and sums up all the oddment values. The remaining function, Pick, returns one of the items according to its weighted chance. The Pick function works by choosing a number n in the range 1 to mOddment. Once that's done, we iterate through the table and add the item oddments up; once we have a number greater or equal to n, we return the associated item. If we manage to loop through the entire table, the last item is returned.

Ok, let's try a few examples out in main.lua. Copy the code from Listing 3.289.

```

LoadLibrary('Asset')
LoadLibrary('Renderer')
Asset.Run('OddmentTable.lua')

```

```

gRenderer = Renderer.Create()

local ot = OddmentTable:Create
{
    { oddment = 1, item = "sword" },
    { oddment = 1, item = "book" }
}

for i = 0, 50 do
    print(ot:Pick())
end

function update()
    gRenderer:AlignText("center", "center")
    gRenderer:DrawText2d(0, 0, "Testing the Oddment Table")
end

```

Listing 3.289: Testing out the OddmentTable. In main.lua.

In this example we've create an oddment table with two items, a sword and a book. The items have an equal oddment rating so there's a 50% chance that either is returned when Pick is called. The test code just prints out a pick 50 times to the console. Run the code and you'll see that the distribution is around 50/50. Try changing the numbers. Make the sword oddment 2 or 100 and see how the print changes. Also try adding new items to the oddment table.

Oddment tables can be used for enemy loot, encounters, special attacks, or whenever we want a little bit of random chance.

Loot Drops

Every time we defeat a creature, there's a chance they'll drop some loot. Most creatures drop a bit of gold and have a smaller chance of dropping an item. Some creatures like bosses have guaranteed drops. We'll consider the XP to be a drop, as it's something the party members only get when they defeat a creature. Therefore our drop table looks like Listing 3.290.

```

drop =
{
    xp = 10,
    gold = {0, 10},
    always = { 300 },
    chance =
    {

```

```

        { oddment = 95, item = { id = -1 } },
        { oddment = 5, item = { id = 30, count = 2 } }
    }
}

```

Listing 3.290: A code sketch of a loot drop table.

XP is an absolute number that's dropped every time. Gold is a range from 0 to 10. The always field is a table of item ids that are guaranteed to drop. The chance field is an oddment table definition with a 95% chance to drop nothing and a 5% chance to drop the item with an id of 30. We're using the item id of -1 to represent nothing.

Does 5% seem like a small chance? Remember that JRPGs involve a lot of battles with multiple enemies in each battle. If you're fighting three enemies of the same type, each has a 5% chance of dropping loot, so it's more likely than you might think.

Suitable Loot

The item database only has weapons and armor in it. These are great for high-end loot, but we also want a few potions and trinkets. Let's add some more interesting entries. The code so far is available in loot-5. It has the oddment table already included in the manifest and dependencies.

Edit the ItemDB.lua file and add the items in Listing 3.291.

```

{
    name = "Mana Potion",
    type = "useable",
    description = "Heals a small amount of MP.",
    use =
    {
        action = "small_mp_restore",
        target = "any",
        target_default = "friendly_lowest_mana",
        hint = "Choose target to restore mana."
    }
},
{
    name = "Mysterious Torque",
    type = "accessory",
    description = "A golden torque that glitters.",
    stats =
    {
        add =
        {

```

```
        strength = 10,
        speed = 10
    }
},
},
```

Listing 3.291: Adding some new items to act as loot. In ItemDB.lua.

Extending Enemy Defs

We only have a single enemy, the goblin. Let's add some drop information to its def. Here's the updated def. Copy the code from Listing 3.292.

```
goblin =
{
    id = "goblin",
    -- code omitted

    drop =
    {
        xp = 5,
        gold = {0, 5},
        always = {},
        chance =
        {
            { oddment = 95, item = { id = -1 } },
            { oddment = 3, item = { id = 11 } }
        }
    }
}
```

Listing 3.292: Extending the EnemyDef with XP and LootDrops. In EnemyDefs.lua.

The goblin gives 5xp points when killed. The drop range for gold is between 0 and 5 gold. The always table is empty and the chance table means there is a low chance of a potion being dropped. Item Id 11 refers to the index of the potion in the ItemDB table.

The Actor code uses the EnemyDef when creating an enemy. We're going extend this to optionally add a drop table. This code is quite generous with default values to make it less taxing to write out enemy definitions.

Copy the code from Listing 3.293.

```

if def.drop then
    local drop = def.drop
    local goldRange = drop.gold or {}
    local gold  = math.random(goldRange[0] or 0, goldRange[1] or 0)

    this.mDrop =
    {
        mXP = drop.xp or 0,
        mGold = gold,
        mAlways = drop.always or {},
        mChance = OddmentTable:Create(drop.chance)
    }

end

this.mNextLevelXP = NextLevel(this.mLevel)

setmetatable(this, self)
return this
end

```

Listing 3.293: Creating and setting up the drop def for the actor. In Actor.lua.

The code at the end of the Actor constructor adds the loot drop information. First we decide the gold drop amount. If there's no gold entry in the def, the value defaults to zero. Otherwise the gold is set to the range of the two passed-in numbers. The mDrop table stores the drop information. The XP is set from the def, or it defaults to zero. The mAlways table is taken directly from the drop.always field. If that doesn't exist, it defaults to an empty table. An OddmentTable is created and assigned to mChance using the drop.chance table to set it up.

Every time an enemy is created, their drop information is assigned.

When to Drop Loot

Enemies drop loot if you kill them. Currently this is the only possible outcome in our game, but in the future enemies might be able to run away, be "disappeared" by magic spells, or the party may run away. We need to consider how to handle loot drops in these cases.

In the GameState we'll record all loot dropped by enemies the party kills. To do this let's add a little code to the HandleEnemyDeath function, as shown in Listing 3.294.

```

CombatState.__index = CombatState
function CombatState:Create(stack, def)

    local this =
    {
        -- code omitted

        mEffectList = {},
        mLoot = {}
    }

```

Listing 3.294: Adding an mLoot table to the CombatState constructor. In CombatState.lua.

This mLoot list in the constructor keeps a record of all the loot dropped by each defeated creature. Let's use it in the HandleEnemyDeath function. Copy the code from Listing 3.295.

```

function CombatState:HandleEnemyDeath()

    -- code omitted

    -- Add the loot to the loot list
    table.insert(self.mLoot, actor.mDrop)

    -- Add to effects
    table.insert(self.mDeathList, character)
end
end
end

```

Listing 3.295: Adding loot to the mLootTable. In CombatState.lua.

In HandleEnemyDeath we take the actor drop table and add it in the loot table.

We already have a placeholder function called CalcCombatData. Let's implement it properly. Copy the code from Listing 3.296.

```

function CombatState:CalcCombatData()

    local drop =
    {
        xp = 0,
        gold = 0,

```

```

        loot = {}
    }

    local lootDict = {}

    for k, v in ipairs(self.mLoot) do
        drop.xp = drop.xp + v.mXP
        drop.gold = drop.gold + v.mGold

        -- Items that are always dropped
        for _, itemId in ipairs(v.mAlways) do
            if lootDict[itemId] then
                lootDict[itemId] = lootDict[itemId] + 1
            else
                lootDict[itemId] = 1
            end
        end

        local item = v.mChance:Pick()
        if item and item.id ~= -1 then

            item.count = item.count or 1

            if lootDict[item.id] then
                lootDict[item.id] = lootDict[item.id] + item.count
            else
                lootDict[item.id] = item.count
            end
        end

        for k, v in pairs(lootDict) do
            table.insert(drop.loot, {id = k, count = v})
        end

        return drop
    end

```

Listing 3.296: Implementing the CalcCombatData function. In CombatState.lua.

A little more complicated than you might have expected, right? This is because there's loot from multiple enemies and we want to make sure the same items are stacked together. For example, if you kill 3 goblins and they each drop a potion, then the screen should display "Potion x3" not "Potion, Potion, Potion". This function combines the drop information from every enemy.

The CalcCombatData function starts with the default loot drop, drop. It contains 0 xp 0 gold and no loot. Then we create a lootDict. This records the amounts of items we find. The keys are the item ids, and the values are the item counts. The this.mLoot table is a list of all drop tables from each defeated enemy.

We loop through the tables and add up the gold and the xp amounts, and store the results in the local drop table. We also loop through the mAlways table, which is a list of item ids, and insert them into the lootDict. If an entry already exists in the lootDict we increment its count by 1. If no entry exists, we add an entry using the item id and set the count to 1.

The mChance table is an oddment table, so we can call the Pick method to get a random piece of loot. If the Pick method returns a valid item id (not null or -1 the EmptyItem) then we add it into the lootDict. The mChance entries contain an item id and a count, so we add the item to the itemDict, incrementing the existing entry by the count, or we create a new entry using the id and count values.

After iterating through all the enemy loot, we're ready to parse the lootDict table. We iterate through it and add each item with its count to the drop.loot list. The drop data is then complete and can be returned.

Run loot-5-solution. Kill the goblin, and you'll see that you receive the correct loot. It's worth experimenting with this code. Add a few more goblins and see how the loot changes. Change the goblin def by adding an item to its always table. Once you're satisfied that everything is working correctly, then we're in a good place. We have a combat system that gives our party members levels and loot. This is one of the core RPG gameplay loops.

Advanced Combat

Our combat calculation at the moment is about as basic as it could possibly be. To make combat more interesting we're going to beef it up, beginning with the melee combat event.

In this section, we'll change up the combat formulas to support missing, dodging, counter attacks and critical strikes. Then we'll add more combat actions: flee, item, magic and special. To test these new actions out, we'll add a few new items we can use during combat. We'll also define some spells and two new special actions, slash and steal. This covers a good range of combat actions.

Melee Combat

Let's break melee combat down into a flow chart to define exactly what happens during an attack. Then we'll build up a basic combat system from small code snippets and pull it all together. There are many different ways to implement a combat system, and the

details depend on the type of game you're making. I'll implement a serviceable combat system that you can build on and extend as you wish!

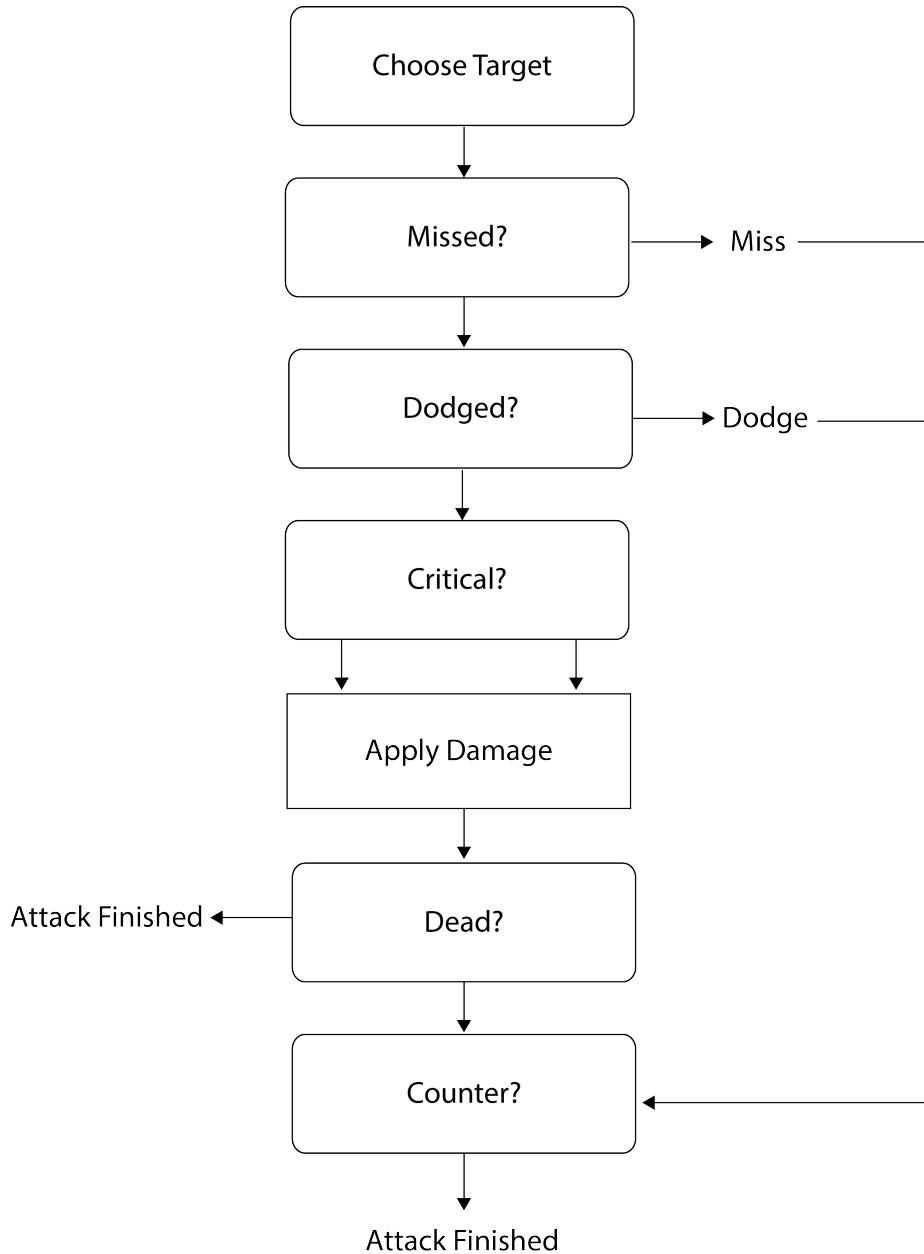


Figure 3.59: The flow of a melee attack event.

Is Figure 3.59 more complicated than you expected? There's quite a bit that goes on during an attack. This diagram doesn't even capture everything, but it's useful as a base for the code. We already take care of the choose target step using the combat queue, so we can skip that, but the rest of it we need to implement. We'll rough these steps out in small code snippets first, then pull them into our codebase. Let's start by calculating the chance to hit.

Chance to Hit

The *chance to hit* determines if an attack connects with the target. If the attack misses, the opponent may counter attack if they are able.

Listing 3.297 shows some simple code that calculates a chance to hit. We're not copying this code anywhere. We're just using it to think through the problem.

```
-- cth = Chance to Hit
local cth = 0.7
local bonus = ((speed + intelligence) / 2) / 255
local cth = cth + (bonus / 2)

local rand = math.random()
local isHit = rand <= cth -- true if the attack lands the hit, otherwise false
```

Listing 3.297: Calculating Chance to Hit.

In a full game, the chance to hit calculation can be more complicated, taking into account status effects like blinding, or weapon size, but for the book we'll just use the player's stats.

Chance to hit is represented as a number between 0 and 1. Rather than create a new stat to determine hit chance, we use the average of the attacker's intelligence and speed stats. The base chance to hit is 70% and this is increased a little by the speed and intelligence of the character. Speed and intelligence have a max value of 255, so we divide by 255 to transform it to a range from 0 - 1. Half of this number is added as a bonus to hit. To determine if there was a hit, we get a random number from 0 to 1, and if it's less than or equal to the *chance to hit* then it's a hit.

It's easy to extend this code and add a chance for a critical hit. A critical hit describes an attack that for some reason inflicts a lot more damage than usual. Perhaps the enemy was hit in the eye, or in a gap in their armor. See the critical hit code in Listing 3.298.

```
local crit = 0.15

local rand = math.random()
local isCrit = rand <= crit
local isHit = rand <= cth
```

Listing 3.298: Calculating chance to hit critically.

The critical chance is set to 15%. Figure 3.60 helps visualise the chance to hit using a bar.

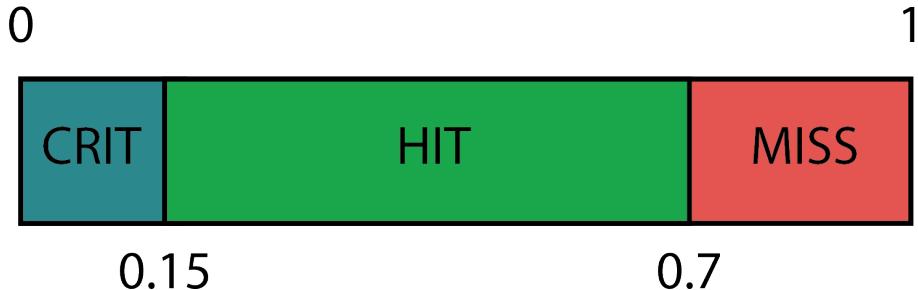


Figure 3.60: Chance to hit bar.

We get a number from 0 to 1. If it's below 0.15, 15%, then it's a critical hit. If it's below 0.7, 70%, then it's a hit. If it's above 0.7, 70%, then it's a miss.

Chance to Dodge

If the actor succeeds in the chance to hit test, then we check if the opponent manages to dodge the attack. Dodging is handled in the same way as the chance to hit test; we calculate a dodge chance 0 - 1, get a random number in that range, and if it's less than the dodge chance the opponent dodges.

Check out the code in Listing 3.299.

```
-- ctd = Chance To Dodge
local ctd = 0.03 -- 3%
local speedDiff = speed - enemySpeed
-- clamp speed diff to plus or minus 10%
speedDiff = Clamp(speedDiff, -10, 10) * 0.01

ctd = math.max(0, ctd + speedDiff)

local rand = math.random()
local isDodged = rand <= ctd
```

Listing 3.299: Calculating the dodge chance.

A dodge is when the attack should hit but the target moves out of the way. In our game, dodges do not happen often. The minimum chance is 0% and the maximum chance is 13%. Dodging is cool when it happens for the party members but less so if it commonly occurs for enemies.

The basic chance to dodge is 3% plus the difference in speed between the attacker and the enemy. The speed difference is clamped to the range of -10 to 10 and added to the base value with a `max` call to ensure it doesn't drop below 0.

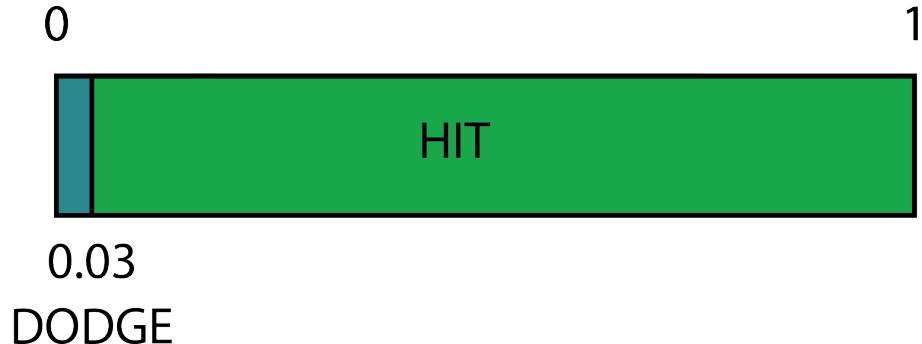


Figure 3.61: Chance to dodge bar.

Figure 3.61 shows the base dodge chance. It's quite small, but in most RPGs you perform attacks many thousands of times.

Calculating Total Damage

Now we know that if a hit connects we can determine the damage amount. The damage amount comes from the attack stat, which usually comes from the weapon, plus half the character's strength. To get the base damage, we take a random number in the range of attack to attack * 2. Then we subtract the enemy defense stat. If the hit is critical, we add an additional random number between attack and attack * 2.

Check out the code in Listing 3.300.

```
local attack = (strength / 2) + attackStat

function attackFunc()
    return math.random(attack, attack * 2)
end

local damage = attackFunc() - defenseStat
```

```
if isCrit then
    damage = damage + attackFunc()
end
```

Listing 3.300: Calculating the damage amount.

Calculating Counter Attack

Counter attacks are a type of attack that triggers immediately in response to an attempted attack. For party members, this is usually a special ability. For monsters, a counter attack is occasionally implemented with a certain percentage chance of occurring. In our game, if a player character can counter, then they always counter.

If the actor never counters, the counter stat is 0. If the actor always counters, the counter stat is 1. If set to 0.5, then there's a 50% chance of a counter occurring. Stats default to 0, so by default no characters or enemies have a counter ability because we haven't previously defined this stat.

You can see the counter code below in Listing 3.301.

```
local rand = math.rand() * 0.99999 -- in the range 0 .. 0.99999
local isCounter = rand < counterStat
```

Listing 3.301: Calculating if a counter attack takes place.

The `math.rand` is set from 0 to 0.9999 instead of 0 to 1. This is to ensure that if the character has a counter stat of 1 it always occurs but if it's 0 it never occurs.

Implementing the Combat Formula

The combat calculations need to be accessible, modular and simple. There's a project called advanced-combat-1 that contains the code so far, and we'll extend with it the new combat calculations.

The combat formulas introduced a Clamp function. Clamp isn't defined in Lua, so let's add it now in `Util.lua`. Copy the code from Listing 3.302.

```
function Clamp(value, min, max)
    return math.max(min, math.min(value, max))
end
```

Listing 3.302: Adding a Clamp function. In Util.lua.

Clamp restricts a number to a certain range. If a value is too high, it's reduced to the maximum. If it's too low, it's increased to the minimum.

We already have the code file CombatFormula.lua. That's where we'll add our new, more detailed formula code. Let's update the MeleeAttack which pulls together many of the smaller calculations.

Copy the code from Listing 3.303. This code uses a number of functions we've yet to define, but we'll be getting to them shortly.

```
HitResult =
{
    Miss      = 0,
    Dodge     = 1,
    Hit       = 2,
    Critical   = 3
}

function Formula.MeleeAttack(state, attacker, target)

    local stats = attacker.mStats
    local enemyStats = target.mStats

    local damage = 0
    local hitResult = Formula.IsHit(state, attacker, target)

    if hitResult == HitResult.Miss then
        return math.floor(damage), HitResult.Miss
    end

    if Formula.IsDodged(state, attacker, target) then
        return math.floor(damage), HitResult.Dodge
    end

    local damage = Formula.CalcDamage(state, attacker, target)

    if hitResult == HitResult.Hit then
        return math.floor(damage), HitResult.Hit
    end

    -- Critical
    assert(hitResult == HitResult.Critical)

    damage = damage + Formula.BaseAttack(state, attacker, target)
    return math.floor(damage), HitResult.Critical
end
```

Listing 3.303: Implementing the full melee attack code. In CombatFormula.lua.

The updated melee attack is more refined and now returns two values, the damage inflicted and a HitResult. The HitResult table is defined above the function and is global. A HitResult can be one of four values:

- **Miss** - The attack misses the target entirely.
- **Dodge** - The attack was true but the target dodged it.
- **Hit** - The attack hits.
- **Critical** - The attack hits critically and does more than the usual amount of damage.

To handle the new HitResult we'll update the CombatState and add code to communicate the result to the player. If there's a miss, we'll display "Miss" text over the target; we'll do the same for dodge and critical.

The MeleeAttack function begins by getting the stats of the attacker and the target. The damage variable is initialized to 0, which is the default return value. We call the IsHit formula to determine if the attack is a hit, miss, or critical strike. If it's a miss, we return the hit result as a miss and the damage as zero. If it's a dodge, we return the hit result as dodge and damage as zero. Otherwise we know a hit has occurred and it's just a question of the damage amount. To get the damage amount we call CalcDamage. If the HitResult equals "Hit" then we return the damage amount. If the HitResult equals "Critical" then we add on another BaseAttack as a bonus and return the total.

This code is broken up into functional pieces because this is an area of code that gets tweaked a lot when making an RPG.

Let's implement the rest of the Formula functions for the melee attack. Copy the code from Listing 3.304.

```
function Formula.IsHit(state, attacker, target)

    local stats = attacker.mStats
    local speed = stats:Get("speed")
    local intelligence = stats:Get("intelligence")

    local cth = 0.7 --Chance to Hit
    local ctc = 0.1 --Chance to Crit

    local bonus = ((speed + intelligence) / 2) / 255
    local cth = cth + (bonus / 2)

    local rand = math.random()
    local isHit = rand <= cth
    local isCrit = rand <= ctc
```

```

if isCrit then
    return HitResult.Critical
elseif isHit then
    return HitResult.Hit
else
    return HitResult.Miss
end
end

```

Listing 3.304: Determining if a hit lands, is critical, or misses. In CombatFormula.lua.

The IsHit formula is much the same as the formula we discussed earlier, but now it gets its data from the actors. We calculate the chance of a critical hit, a hit, or a miss and return the result.

Next is the dodge. Copy the code from Listing 3.305.

```

function Formula.IsDodged(state, attacker, target)

    local stats = attacker.mStats
    local enemyStats = target.mStats

    local speed = stats:Get("speed")
    local enemySpeed = enemyStats:Get("speed")

    local ctd = 0.03 --Chance to Dodge
    local speedDiff = speed - enemySpeed
    -- clamp speed diff to plus or minus 10%
    speedDiff = Clamp(speedDiff, -10, 10) * 0.01

    ctd = math.max(0, ctd + speedDiff)

    return math.random() <= ctd
end

```

Listing 3.305: The formula to check if an attack is dodged. In CombatFormula.lua.

For the IsDodge function, we get the speed from the attacker and the target, compare the difference, and use this to help determine the chance to dodge. We return true if the target dodges; otherwise we return false.

Next up is the BaseAttack formula. Copy the code from Listing 3.306.

```

function Formula.BaseAttack(state, attacker, target)
    local stats = attacker.mStats
    local strength = stats:Get("strength")
    local attackStat = stats:Get("attack")

    local attack = (strength / 2) + attackStat

    return math.random(attack, attack * 2)
end

function Formula.CalcDamage(state, attacker, target)
    local targetStats = target.mStats
    local defense = targetStats:Get("defense")

    local attack = Formula.BaseAttack(state, attacker, target)

    return math.floor(math.max(0, attack - defense))
end

```

Listing 3.306: The basic attack formula and damage formula. In CombatFormula.lua.

The BaseAttack is a direct implementation of the code we discussed earlier. BaseAttack is used by the CalcDamage function. CalcDamage takes the extra step of subtracting the defense value from the BaseAttack value and also ensures that the damage value doesn't go negative! (A negative damage attack would heal the target!). If the CalcDamage value is zero, we could say the target blocked the attack, but we're not going to handle this case.

Finally, we have the IsCountered function which we'll make use of shortly. Again, it's a direct implementation of the formula discussed earlier. Copy the code from Listing 3.307.

```

function Formula.IsCountered(state, attacker, target)
    local counter = target.mStats:Get("counter")
    return math.random()*0.99999 < counter
end

```

Listing 3.307: Formula to check if a counter attack occurs. In CombatFormula.lua.

A Text Animation Effect

Criticals, misses, and dodges can now happen, but there's no way to tell the player about them! To tell the player what's up, let's add a new combat effect that displays some text. Create a new CombatTextFx.lua file in the code/fx/ folder.

This effect makes some text do a small bounce, hang around just long enough to be read, and then fade out. It's very similar to the JumpingNumber effect but less dynamic so it's easier to read. Figure 3.62 shows a mockup of what the effect looks like.



Figure 3.62: Text effects to communicate what's going on in battle.

Copy the code from Listing 3.308.

```
-- Used to display "miss", "critical", "poisoned", etc
CombatTextFx = {}
CombatTextFx.__index = CombatTextFx
function CombatTextFx>Create(x, y, text, color)
    local this =
    {
        mX = x or 0,
        mY = y or 0,
        mText = text or "", -- to display
        mColor = color or Vector.Create(1,1,1,1),
        mGravity = 700, -- pixels per second
        mScale = 1.1,
        mAlpha = 1,
        mHoldTime = 0.175,
        mHoldCounter = 0,
        mFadeSpeed = 6,

        mPriority = 2,
    }
    this.mCurrentY = this.mY
    this.mVelocityY = 125,

    setmetatable(this, self)
    return this
```

```

end

function CombatTextFx:IsFinished()
    -- Has it passed the fade out point?
    return self.mAlpha == 0
end

function CombatTextFx:Update(dt)

    self.mCurrentY = self.mCurrentY + (self.mVelocityY * dt)
    self.mVelocityY = self.mVelocityY - (self.mGravity * dt)

    if self.mCurrentY <= self.mY then
        self.mCurrentY = self.mY

        self.mHoldCounter = self.mHoldCounter + dt

        if self.mHoldCounter > self.mHoldTime then
            self.mAlpha = math.max(0, self.mAlpha - (dt * self.mFadeSpeed))
            self.mColor:SetW(self.mAlpha)
        end
    end
end

function CombatTextFx:Render(renderer)

    renderer:ScaleText(self.mScale, self.mScale)
    renderer:AlignText("center", "center")

    local x = self.mX
    local y = math.floor(self.mCurrentY)
    local shadowColor = Vector.Create(0,0,0, self.mColor:W())
    renderer:DrawText2d(x + 2, y - 2, self.mText, shadowColor)
    renderer:DrawText2d(x, y, self.mText, self.mColor)

end

```

*Listing 3.308: A class to display text to the player during combat. In *CombatTextFx.lua*.*

CombatTextFx takes in four parameters: the position x, the position y, the text to display, and the color of the text. The x, y, text and color are all stored in the this table. If no color is supplied, it's set to white. The gravity determines how quickly the text falls. The gravity value is set to the same value as in JumpingNumbers, 700 pixels per second, which makes the effects appear consistent. The scale is set 1.1 which is

smaller than the JumpingNumbers as they display the more important information. For the same reason the `mPriority` value is 2, higher than `JumpingNumbers`, which means the `JumpingNumbers` are rendered on top. The opacity is tracked in `mAlpha` to make fading out easier.

Once the text drops back to its base level, we hold it in place for a short time. The duration is defined by `mHoldTime`. The `mHoldCounter` tracks how long the text has been held. The `mFadeSpeed` is the amount to fade the alpha value per second. After the this table, the `mCurrentY` and `mVelocity` values are set. The velocity is set to be lower than that of jumping numbers so the text doesn't jump as high.

The `IsFinished` function returns true if the `mAlpha` is 0, which means the text has entirely faded away.

The `Update` function is similar to the `JumpingNumbers:Update`. It does a basic physics simulation by updating the `mCurrentY` position with the velocity. The velocity in turn is updated by the `mGravity` value. We don't let the text drop below the start position; instead, we hold it there by setting its `mCurrentY` to equal the starting y position. This means the text never drops below the start point. While the text is at the starting position, the `mHoldCounter` is increased. Once the `mHoldCounter` is greater than the `mHoldTime`, we start to decrease the alpha value. The decreasing alpha value is copied into the color that's used to display the text on screen.

The `Render` function draws the text scaled and centered with a small drop shadow. We'll test the effect out by integrating it into the `CombatState`.

Updating CombatState and CEAttack

To test the melee attack out we need to update our current codebase, specifically the `CEAttack` and `CombatState`. These classes are closely coupled, so we'll update them together.

Let's start with the `AttackTarget` function in `CEAttack`. Copy the code from Listing 3.309.

```
function CEAttack:AttackTarget(target)

    local damage, hitResult = Formula.MeleeAttack(self.mState,
                                                    self.mOwner,
                                                    target)

    -- hit result lets us know the status of this attack

    local entity = self.mState.mActorCharMap[target].mEntity

    if hitResult == HitResult.Miss then
```

```

        self.mState:ApplyMiss(target)
        return
    elseif hitResult == HitResult.Dodge then
        self.mState:ApplyDodge(target)
    else
        local isCrit = hitResult == HitResult.Critical
        self.mState:ApplyDamage(target, damage, isCrit)
    end

    local x = entity.mX
    local y = entity.mY
    local effect = AnimEntityFx>Create(x, y,
                                         self.mAttackAnim,
                                         self.mAttackAnim.frames)

    self.mState:AddEffect(effect)

end

```

Listing 3.309: Updating the AttackTarget function. In CEAttack.lua.

The AttackTarget function relies heavily on the CombatState. We call the updated MeleeAttack function, which returns the damage and HitResult values. We check the HitResult, and if it's a miss we call ApplyMiss. If it's a dodge, we call ApplyDodge. Otherwise we call ApplyDamage. The ApplyDamage function takes a new third parameter that asks if the damage was done by a critical hit or not.

Before moving on to CombatState let's add the counter attack code to CEAttack.

Provided the target is still alive, it may attempt to counter after a CEAttack event. We perform the counter check just before the CEAttack finishes in the DoAttack function. Copy the code from Listing 3.310.

```

function CEAttack:DoAttack()
    for _, target in ipairs(self.mTargets) do
        self:AttackTarget(target)

        if not self.mDef.counter then
            self:CounterTarget(target)
        end
    end
end

function CEAttack:CounterTarget(target)
    local countered = Formula.IsCountered(self.mState, self.mOwner, target)
    if countered then

```

```

        self.mState:ApplyCounter(target, self.mOwner)
    end
end

```

Listing 3.310: Update and Counter Attacks. In CEAttack.

When an attack finishes, we check if it's appropriate to trigger a counter attack. A counter cannot trigger another counter attack; otherwise we could easily get stuck in a loop! If an attack is a counter attack, we give it a counter tag in the mDef table. If the counter tag doesn't exist, then for each target we call a new function, CounterTarget. CounterTarget runs the IsCountered calculation, and if this returns true it calls the CombatState:ApplyCounter function.

Let's move on to the CombatState and update the code. Copy the code from Listing 3.311.

```

function CombatState:AddTextEffect(actor, text, color)
    local character = self.mActorCharMap[actor]
    local entity = character.mEntity
    local x = entity.mX
    local y = entity.mY
    local effect = CombatTextFx>Create(x, y, text, color)
    self:AddEffect(effect)
end

function CombatState:ApplyMiss(target)
    self:AddTextEffect(target, "MISS")
end

function CombatState:ApplyDodge(target)
    local character = self.mActorCharMap[target]
    local controller = character.mController

    local state = controller.mCurrent
    if state.mName ~= "cs_hurt" then
        controller:Change("cs_hurt", state)
    end

    self:AddTextEffect(target, "DODGE")
end

```

Listing 3.311: Adding Miss and Dodge handling. In CombatState.lua.

First up, we add a utility function, AddTextEffect. This function takes an actor, finds the associated entity, and creates a CombatTextFX at the entity position.

AddTextEffect is the function we'll use for displaying any simple entity-related message to the player. The ApplyMiss function calls AddTextEffect with the text "MISS" and the target actor. Dodge is similar to miss, but we also play the hurt animation if one exists, as it shows the character moving back, away from the attack, which helps telegraph the dodge event.

Continue copying the updated code from Listing 3.312.

```
function CombatState:ApplyDamage(target, damage, isCrit)
    local stats = target.mStats
    local hp = stats:Get("hp_now") - damage
    stats:Set("hp_now", math.max(0, hp))

    -- Change actor's character to hurt state
    local character = self.mActorCharMap[target]
    local controller = character.mController

    if damage > 0 then
        local state = controller.mCurrent
        if state.mName ~= "cs_hurt" then
            controller:Change("cs_hurt", state)
        end
    end

    local entity = character.mEntity
    local x = entity.mX
    local y = entity.mY

    local dmgColor = Vector.Create(1,1,1,1)

    if isCrit then
        dmgColor = Vector.Create(1,1,0,1)
    end

    local dmgEffect = JumpingNumbers>Create(x, y, damage, dmgColor)
    self:AddEffect(dmgEffect)
    self:HandleDeath()
end
```

Listing 3.312: Applying Damage. In CombatState.lua.

The first part of ApplyDamage is much the same. We get the HP of the target, subtract the damage (making sure not to go below zero), and set that as the new HP value. Then we play the hurt animation. Toward the end we've added a new `dmgColor` variable. It controls the color of the `JumpingNumbers` that show the damage. If the `isCrit` parameter is true, then the `dmgColor` is set to yellow; otherwise it defaults to white. This that

means when a critical attack occurs the damage numbers are yellow, telling the player that something special has happened.

Let's add the ApplyCounter function. Copy the code from Listing 3.313.

```
function CombatState:ApplyCounter(target, owner)
    local alive = target.mStats:Get("hp_now") > 0

    if not alive then
        return
    end

    local def =
    {
        player = self:IsPartyMember(target),
        counter = true
    }

    -- Add an attack state at -1
    local attack = CEAttack>Create(self,
                                    target,
                                    def,
                                    {owner})
    self.mEventQueue:Add(attack, -1)

    self:AddTextEffect(target, "COUNTER")
end
```

Listing 3.313: The code to trigger and a counter. In CombatState.lua.

ApplyCounter checks the target's HP. If it's zero, the target isn't alive and therefore can't counter! If the target is alive, then we create the counter attack def, setting the player flag to true if the target is a party member and adding the counter flag. We create the attack event with time points of -1, meaning it occurs immediately, and push it onto the event queue. Finally, to let the player know that a counter has occurred, we add a text effect over the target showing the words "COUNTER".



Figure 3.63: A screenshot showing the more advanced combat and effects in action.

All the code thus far is available in advanced-combat-1-solution. Run it and try out the more involved combat; it's a bit more exciting. You can see what it should look in Figure 3.63. To test the counter, make `Formula.IsCountered` always return true, or add a counter stat to the goblin.

A Note About Polish

We've added some simple text effects and colored the damage numbers; this is the minimum required to inform the player what's going on in the battle. There's a lot more scope for polishing. Sounds, for instance. Even the effects could be more interesting. Dodge could be an animation, or it could have some wind-type effect. This type of polish is often added at the end of a game and it's something that we won't spend too much time on in the book. This is just a note to acknowledge that there's a lot more that can be done to improve game feel.

Fleeing

Our heroes only have a single option, "Attack". This doesn't make for tactical combat. We're going to start filling the action box out. Let's begin with the tactical retreat – fleeing. There's a base project in advanced-combat-2. Fleeing is only for party members.

Enemies always fight to the death! If the flee succeeds, all members of the party will flee.

Let's begin by adding the formula we use to determine if a flee is successful. Copy the code from Listing 3.314.

```
function Formula.CanFlee(state, fleer)
    local fc = 0.35 -- flee chance
    local stats = fleer.mStats
    local speed = stats:Get("speed")

    -- Get the average speed of the enemies
    local enemyCount = 0
    local totalSpeed = 0
    for k, v in ipairs(state.mActors['enemy']) do
        local speed = v.mStats:Get("speed")
        totalSpeed = totalSpeed + speed
        enemyCount = enemyCount + 1
    end

    local avgSpeed = totalSpeed / enemyCount

    if speed > avgSpeed then
        fc = fc + 0.15
    else
        fc = fc - 0.15
    end

    return math.random() <= fc
end
```

Listing 3.314: The Formula for determining if the party can flee. In CombatFormula.lua.

The basic chance of fleeing is 35%. If you tell all your characters to flee, you've got a reasonable chance of escaping. If the actor's speed is below the enemies' average speed, the chance is 20%; if it's above, it's 50%.

The "Flee" command actually has a quite a bit going on. It's going to be a CombatEvent like CEAttack. The flee sequence of events:

1. Display "Flee" at the top of the screen.
2. Make the actor go prone and face right.
3. Wait a few seconds.
4. Decide if flee succeeds (let's assume yes).
5. Display "Success!"

6. Have the actor run off the screen.
7. Hide the notice at the top of the screen.
8. Have the rest of actors run off screen.
9. Fade out into the combat summary screens.

That's quite a long sequence! You can see the flow in Figure 3.64. Once again, we'll use the Storyboard class to manage this sequence of events.



Figure 3.64: The flow of the flee behavior.

Let's begin by getting the "Flee" action into the game so the player can choose it from the action list. To add a new action, we need to edit the PartyMemberDefs.lua file. While we're here, let's also add the magic and special actions which we'll implement soon. Copy the code from Listing 3.315.

```

gPartyMemberDefs =
{
  hero =
  {
    -- code omitted
    actions = { "attack", "special", "item", "flee" }
  },
  thief =
  {
    -- code omitted
    actions = { "attack", "special", "item", "flee" }
  },
  mage =
  {
    -- code omitted
    actions = { "attack", "magic", item", "flee" }
  }
}
  
```

```
    },
}
```

Listing 3.315: Adding a new Flee action. In PartyMemberDefs.lua.

The action table is processed in CombatChoiceState which looks up a nicely formatted string from Actor.ActionLabels. Let's add the new action labels. Copy the code from Listing 3.316.

```
ActionLabels =
{
    ["attack"] = "Attack",
    ["item"] = "Item",
    ["flee"] = "Flee",
    ["magic"] = "Magic",
    ["special"] = "Special"
},
```

Listing 3.316: Adding action labels for flee, special and magic. In Actor.lua.

Now we can see the new actions when playing, as shown in Figure 3.65, but selecting them does nothing!



Figure 3.65: New actions on the combat action menu.

We need to update the OnSelect code to tell it to add the flee event into the event queue. Update CombatChoiceState code as shown in Listing 3.317.

```
function CombatChoiceState:OnSelect(index, data)

    if data == "attack" then
        -- code omitted
    elseif data == "flee" then
        self.mStack:Pop() -- choice state
        local queue = self.mCombatState.mEventQueue
        local event = CEFlee:Create(self.mCombatState, self.mActor)
        local tp = event:TimePoints(queue)
        queue:Add(event, tp)
    end
end
```

Listing 3.317: Responding to Flee action... In CombatState.lua.

When the Flee event occurs, we need to tell the player what's happening. To signal the party that are escaping, we should display a message. The combat state's mNoticePanel is perfect for this, but we need to add some code to render text and toggle drawing.

Let's update the CombatState constructor. We'll add code to control the notice and tip panels as well as a field to track if the party has fled.

Copy the code from Listing 3.318.

```
function CombatState:Create(stack, def)

    local this =
    {
        -- code omitted
        mFled = false
    }

    -- code omitted
    layout = Layout>Create()

    -- code omitted

    this.mLayout = layout

    this.mTipText = ""
    this.mShowTip = false
    this.mTipPanel = layout>CreatePanel('tip')
```

```
this.mNoticeText = ""
this.mShowNotice = false
this.mNoticePanel = layout>CreatePanel('notice')
```

Listing 3.318: Adding support for messages and fleeing to the CombatState. In CombatState.lua.

The `mFled` field reports if the party has fled. If the party has fled, we end combat.

We store the layout in `mLayout`. The layout is used to align the text in the panels. The `mTipText` and `mNoticeText` fields contain the text we show in the panel when displayed. Two flags, `mShowTip` and `mShowNotice`, toggle the text and panel rendering.

Let's add helper functions to turn these panels on and off. Copy the code from Listing 3.319.

```
function CombatState>ShowTip(text)
    self.mShowTip = true
    self.mTipText = text
end

function CombatState>ShowNotice(text)
    self.mShowNotice = true
    self.mNoticeText = text
end

function CombatState:HideTip()
    self.mShowTip = false
end

function CombatState:HideNotice()
    self.mShowNotice = false
end
```

Listing 3.319: Helper functions to turn on and off tip and notice messages. In CombatState.lua.

Let's update the Renderer so it optionally renders out these two panels and their text. Copy the code from Listing 3.320.

```
if self.mShowTip then
    local x = self.mLayout:Left('tip') + 4
    local y = self.mLayout:MidY('tip')
    renderer:AlignText("left", "center")
    renderer:ScaleText(1.1, 1.1)
```

```

        self.mTipPanel:Render(renderer)
        renderer:DrawText2d(x, y, self.mTipText)
    end

    if self.mShowNotice then
        local x = self.mLayout:MidX('notice')
        local y = self.mLayout:MidY('notice')
        renderer:AlignText("center", "center")
        renderer:ScaleText(1.33, 1.33)
        self.mNoticePanel:Render(renderer)
        renderer:DrawText2d(x, y, self.mNoticeText)
    end

```

Listing 3.320: Altering the rendering so it can show tips and notices during combat. In `CombatState.lua`.

The notice panel is shown in the center of the top of the screen, with large, centered text. The tip panel is used to describe spells and inventory items, so it's near the bottom, the text is small and it's left aligned.

Let's also update the Update function to check the `mFled` flag and end the combat. Copy the code from Listing 3.321.

```

function CombatState:Update(dt)

    -- code omitted

    if self:PartyWins() or self:PartyFled() then
        self.mEventQueue:Clear()
        self:OnWin()
    elseif self:EnemyWins() then
        self.mEventQueue:Clear()
        self:OnLose()
    end
end

function CombatState:OnFlee()
    self.mFled = true
end

function CombatState:PartyFled()
    return self.mFled
end

```

Listing 3.321: Check if the party has fled in the Update. In CombatState.lua.

The OnFlee function is called when the flee event succeeds. It just sets the mFled flag to true. The PartyFled function matches PartyWins and reports if the party has fled or not.

Let's write the flee event. Create a CEFlee.lua file in the combat_events directory and add it to the manifest and dependencies files. Then add the constructor from Listing 3.322.

```
CEFlee = {}
CEFlee.__index = CEFlee
function CEFlee:Create(state, actor)
    local this =
    {
        mState = state,
        mOwner = actor,
        mCharacter = state.mActorCharMap[actor],
        mFleeParams = { dir = 1, distance = 180, time = 0.6 },
        mIsFinished = false,
    }

    -- Decide if flee succeeds
    this.mCanFlee = Formula.CanFlee(state, actor)

    if this.mCanFlee then
        local storyboard =
        {
            SOP.Function(function()
                this:Notice("Attempting to Flee...") end),
            SOP.Wait(0.75),
            SOP.Function(function() this:DoFleeSuccessPart1() end),
            SOP.Wait(0.3),
            SOP.Function(function() this:DoFleeSuccessPart2() end),
            SOP.Wait(0.6)
        }
        this.mStoryboard = Storyboard>Create(
            this.mState.mGameStack,
            storyboard)
    else
        local storyboard =
        {
            SOP.Function(function() this:Notice("Attempting to Flee...") end),
            SOP.Wait(0.75),
            SOP.Function(function() this:Notice("Failed!") end),
            SOP.Wait(0.3),
        }
    end
end
```

```

        SOP.Function(function() this:OnFleeFail() end),
    }
    this.mStoryboard = Storyboard>Create(
        this.mState.mStack,
        storyboard)
end

-- Move the character into the prone state, as soon as the event is
-- created
this.mCharacter.mFacing = "right"
this.mController = this.mCharacter.mController
this.mController:Change(CSRunAnim.mName, {'prone'})

this.mName = string.format("Flee for %s",
    this.mOwner.mName)

setmetatable(this, self)
return this
end

function CEFlee:TimePoints(queue)
    -- Faster than an attack
    local speed = self.mOwner.mStats:Get("speed")
    return queue:SpeedToTimePoints(speed) * 0.5
end

```

Listing 3.322: The CEFlee event constructor. In CEFlee.lua.

CEFlee takes in two parameters: state, which is the combat state, and actor, which is the actor that's attempting to flee. Both parameters are stored in the this table immediately. We also store the character object associated with the actor. We'll use the mFleeParams table to initialize a CSMove state when we want to move the character. The mFleeParams table values instruct the characters to move off screen by travelling 180 pixels in 0.6 seconds. The final field in the this table is mIsFinished, a boolean that reports if the flee event has finished running.

After the this table is set up, we determine if the flee event is successful. We call CombatFormula.CanFlee and store the result in mCanFlee. Therefore we know the outcome of the event before the character actually makes the flee attempt! There are two storyboards, one for successful fleeing and one for failing. Depending on the value of mCanFlee we store the correct storyboard in mStoryboard.

Let's take a look at the successful storyboard. You'll notice it's making full use of our new SOP.Function operation. The first command puts up the notice "Attempting to Flee..." by calling CEFlee:Notice which we'll write soon. Then there's a 0.75 second pause, a call to DoFleeSuccess1, another wait, and finally a call to DoFleeSuccess2.

DoFleeSuccess1 tells the first actor to run away. DoFleeSuccess2 tells the rest of the actors to run away and hides the notice message. There's a final wait to give the party enough time to get off screen before the combat ends.

The failure storyboard starts in similar way, but a "Failure!" notice is displayed, then there's a wait operation followed by a call to OnFleeFail.

The final part of constructor faces the character right and sets them to the prone state. The time points for the CEFlee event are 50% lower than an attack. Finally we set the event name to make debugging easier.

Let's add the rest of the functions, which start to make the storyboards more readable. Copy the code from Listing 3.323.

```
function CEFlee:Notice(text)
    self.mState>ShowNotice(text)
end

function CEFlee:DoFleeSuccessPart1()
    self:Notice("Success!")
    self.mController:Change(CSMove.mName, self.mFleeParams)
end

function CEFlee:DoFleeSuccessPart2()
    for k, v in ipairs(self.mState.mActors['party']) do
        local alive = v.mStats:Get("hp_now") > 0
        local isFleer = v == self.mOwner

        if alive and not isFleer then
            local char = self.mState.mActorCharMap[v]
            char.mFacing = "right"
            char.mController:Change(CSMove.mName, self.mFleeParams)
        end
    end
    self.mState:OnFlee()
    self.mState:HideNotice()
end

function CEFlee:OnFleeFail()
    self.mCharacter.mFacing = "left"
    self.mController = self.mCharacter.mController
    self.mController:Change(CSStandby.mName)
    self.mIsFinished = true
    self.mState:HideNotice()
end
```

Listing 3.323: Storyboard function implementations. In CEFlee.lua.

The Notice function calls CombatState:ShowNotice, which we wrote earlier.

DoFleeSuccessPart1 displays the “Success” message and changes the character controller to the CSMove state using the mFleeParams table. This sets the character running off the side of the screen. DoFleeSuccessPart2 iterates through the party members and tells the rest of party members to flee. Then CombatState.OnFlee is called, which marks the OnFled flag to true and means the CombatState will start to end on the next update. It also hides the notice box.

OnFleeFail makes the character face the enemies, sets the character controller to standby, tells the event it’s finished, and hides the notice panel.

That’s the meat of the CEFlee event. Let’s finish it off by writing the few remaining functions. Copy the code from Listing 3.324.

```
function CEFlee:IsFinished()
    return self.mIsFinished
end

function CEFlee:Execute()
    self.mState.mStack:Push(self.mStoryboard)
end

function CEFlee:Update(dt) end
```

Listing 3.324: The final few functions for the flee event. In CSFlee.lua.

The IsFinished function returns the mIsFinished flag. The Execute function runs when the event is first executed and pushes the storyboard onto the stack. This fires off the flee sequence. The Update does nothing, as the storyboard on the stack does the work of updating the combat state.



Figure 3.66: The party as they flee a battle.

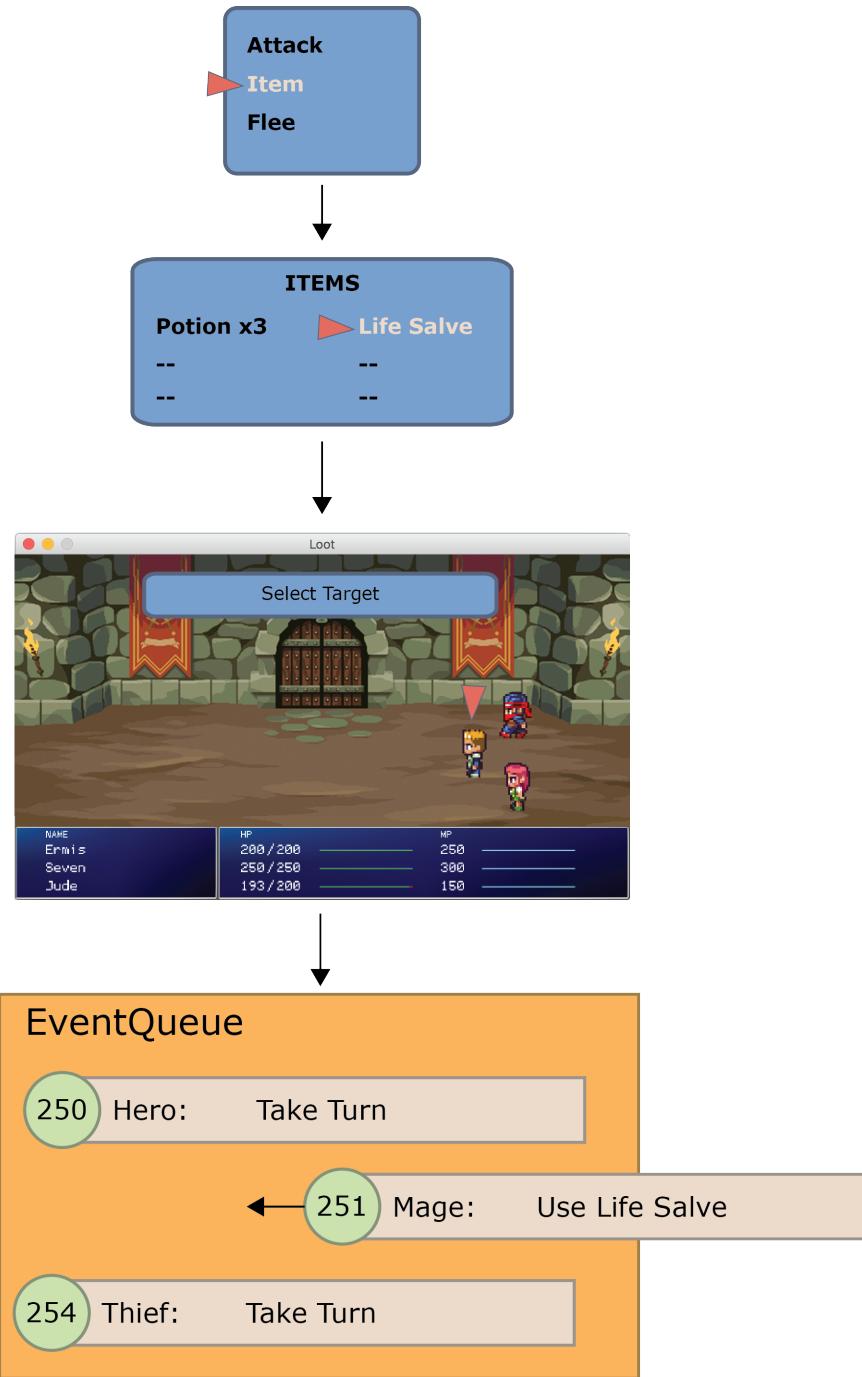
That's the CEFlee event implemented. The code so far is available in advanced-combat-solution-2. Check it out. You can see the party in mid-flee in Figure 3.66.

Items

The next combat action we'll write is the "Item" action. Adding the "Item" action is quite a bit of work, but it lays the foundation for the "Magic" and "Special" abilities.

When the player selects "Item", we'll show the player a filtered version of the inventory and allow any of items to be used. This means we need a UI element that displays a box over the combat UI and lets the player browse a list.

Once an item is chosen, the player must choose a target to use the item on. After the targets are selected, we create a CEUseItem event and add it to the event queue. You can see the flow for item use in Figure 3.67.



New use item event added to the queue.

Figure 3.67: The flow of using an item during combat.

Items have many different effects: healing, resurrection, damaging the enemy, escaping, etc. The item effect code needs to be quite general. In this section we'll write the code for three items: a heal potion, a mana potion, and a revive. Example advanced-combat-3 contains the code so far.

Let's begin by making it easy to test the items by adding them to the inventory. Then we'll go on to define the new items. We'll also add a few weapons to the inventory to make sure our filtering works correctly. Copy the code from Listing 3.325.

```
gWorld:AddItem(1) -- bone blade
gWorld:AddItem(2) -- bone armor
gWorld:AddItem(3) -- Ring of Titan
gWorld:AddItem(10, 2) -- potion x2
gWorld:AddItem(11) -- Life Salve
gWorld:AddItem(12) -- mana potion

function update()
    local dt = GetDeltaTime()
    gStack:Update(dt)
    gStack:Render(gRenderer)
    gWorld:Update(dt)
end
```

Listing 3.325: Adding useable items to the party inventory. In main.lua.

Each of these items has a different effect.

- **Heal Potion** - Restores a small amount of HP (250HP) for one actor.
- **Mana Potion** - Restores a small amount of MP (50MP) for one actor.
- **Revive Salve** - Restores a KO'ed actor to life and gives 100HP.

There are some similarities in these items. They all affect stats and they all require a single target. But their functionality is quite different. Let's add them to ItemDB and you'll see how we define each one's properties.

Copy the code from Listing 3.326.

```
{  
    name = "Heal Potion",  
    type = "useable",  
    description = "Heal a small amount of HP.",
```

```

use =
{
    action = "hp_restore",
    restore = 250,
    target =
    {
        selector = "MostHurtParty",
        switch_sides = true,
        type = "One"
    },
    hint = "Choose target to heal."
}
},
{
    name = "Life Salve",
    type = "useable",
    description = "Restore a character from the brink of death.",
    use =
    {
        action = "revive",
        restore = 100,
        target =
        {
            selector = "DeadParty",
            switch_sides = true,
            type = "One"
        },
        hint = "Choose target to revive."
    }
},
{
    name = "Mana Potion",
    type = "useable",
    description = "Heals a small amount of MP.",
    use =
    {
        action = "mp_restore",
        restore = 50,
        target =
        {
            selector = "MostDrainedParty",
            switch_sides = true,
            type = "One"
        },
        hint = "Choose target to restore mana."
}
}

```

```
    },
},
```

Listing 3.326: Beefier item definitions. In ItemDB.lua.

All three of these items have a name that is used when displaying the item to the player. Each item has its type field set to useable. They all have a description. We show the description on the combat state's tip panel above the combat UI. All the items also contain a table called use. The use table describes how the item works when the player selects it in combat. The use table contains the follow fields:

- **action** - The name of a function we'll define later. It applies the item's effect to the combat scene including any special effects.
- **restore** - Optional. Piece of data used by the action. In this case it's the amount of HP or MP to restore. We could easily make a more powerful heal potion by just copying the existing one, changing the name, and upping the restore amount.
- **target** - Describes which entities are targeted by the item. It's used to set up the CombatTargetState. We've seen this before for the melee attack.
- **hint** - A tip displayed during the targeting phase.

Each of the three items introduces a new selector. Now is as good of a time as any to add the selectors to the CombatTargetState code file. First let's consider what they actually select:

- **MostHurtParty** - Find the party member with the lowest HP that is below their max HP value.
- **MostDrainedParty** - Find the party member with the lowest MP that is below their max MP value.
- **DeadParty** - Find the first dead party member or return the first actor in the list.

The MostHurtParty and MostDrainedParty select the actors who need healing and magic restoration most urgently. If the mage has 100/100 HP and the warrior has 200/250, the mage has the lowest HP but it's the warrior who should be selected by default. That's why why we check that the HP is below the max amount.

As we add the new selectors, we're also going to group some common code into functions. Copy the updates from Listing 3.327.

```
local function WeakestActor(list, onlyCheckHurt)
    local target = nil
    local health = 99999

    for k, v in ipairs(list) do
```

```

local hp = v.mStats:Get("hp_now")
local isHurt = hp < v.mStats:Get("hp_max")
local skip = false
if onlyCheckHurt and not isHurt then
    skip = true
end

if hp < health and not skip then
    health = hp
    target = v
end

return { target or list[1] }
end

local function MostDrainedActor(list, onlyCheckDrained)
    local target = nil
    local magic = 99999

    for k, v in ipairs(list) do
        local mp = v.mStats:Get("mp_now")
        local isHurt = mp < v.mStats:Get("mp_max")
        local skip = false
        if onlyCheckDrained and not isHurt then
            skip = true
        end

        if mp < magic and not skip then
            magic = mp
            target = v
        end
    end

    return { target or list[1] }
end

CombatSelector =
{
    WeakestEnemy = function(state)
        return WeakestActor(state.mActors["enemy"], false)
    end,

    WeakestParty = function(state)

```

```

        return WeakestActor(state.mActors["party"], false)
    end,

MostHurtEnemy = function(state)
    return WeakestActor(state.mActors["party"], true)
end,

MostHurtParty = function(state)
    return WeakestActor(state.mActors["party"], true)
end,

MostDrainedParty = function(state)
    return MostDrainedActor(state.mActors["party"], true)
end,

DeadParty = function(state)
    local list = state.mActors["party"]

    for k, v in ipairs(list) do
        local hp = v.mStats:Get("hp_now")
        if hp == 0 then
            return { v }
        end
    end
    -- Just return the first
    return { list[1] }
end,
-- code omitted

```

Listing 3.327: Updating the CombatSelector table. In CombatTargetState.lua.

WeakestActor is a function that returns the weakest actor from a list. It takes in the parameter onlyCheckHurt, which changes its behavior to only include actors with HP below their maximum. Sometimes we want to target an enemy with the lowest absolute HP, i.e. to attack, while in other cases the most damaged is more important, i.e. for healing.

MostDrainedActor does the same as WeakestActor but uses the MP instead of the HP stat.

A host of new selectors are defined, wrapping up the helper functions. The DeadParty selector is different. It looks through the list of actors in the party and returns the first with 0 HP. If no such actor exists, the first actor in the list is returned.

That's all the selectors we need to add for now. Next we'll implement a UI element to display a browseable list to the player during combat.

Upgrading the Combat Menus

In combat, when we choose the “Item” action, it needs to open a selection menu showing a list of all the usable items. This means we need a way to filter the inventory items so we only see the usable ones.

Let’s add a `FilterItems` function. It applies a filter function to each item in the inventory. If the function returns true, it’s added to a list and the list is returned. The `FilterItems` function is pretty short. Copy the implementation from Listing 3.328.

```
function World:FilterItems(Predicate)
    local list = {}
    for k, v in ipairs(self.mItems) do
        local def = ItemDB[v.id]

        if Predicate(def) then
            table.insert(list, v)
        end
    end
    return list
end
```

Listing 3.328: Adding a function to filter the inventory items. In `World.lua`.

To display the items we’d ideally reuse the `CombatChoiceState` but we want to show more than one column, have a title, show tooltips, and be a lot bigger! Therefore we’ll create a new state called `BrowseListState`. `BrowseListState` displays a selection menu on top of a panel, with a title, and it has a few callbacks to control how it functions. By changing these callbacks and altering the data source, we can reuse `BrowseListState` for the special attacks and magic.

When the user selects the “Items” action, we’ll create a `BrowseListState` object, fill it with the appropriate items, and push it to the top of the stack. When the user chooses an item, it will fire off a callback for us to handle. Figure 3.68 shows how the combat state internal stack changes as an item is used.

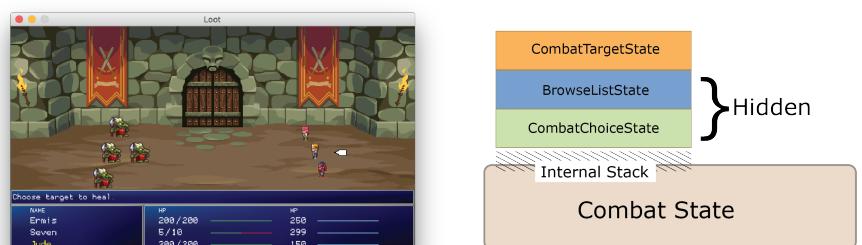
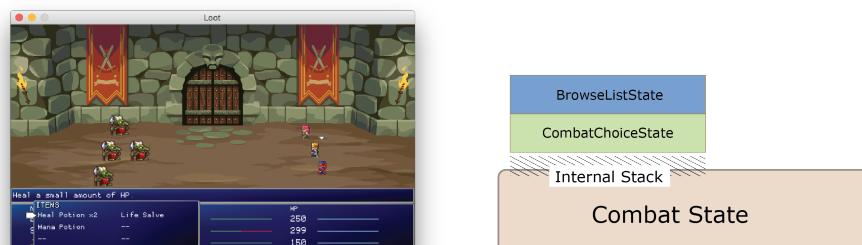
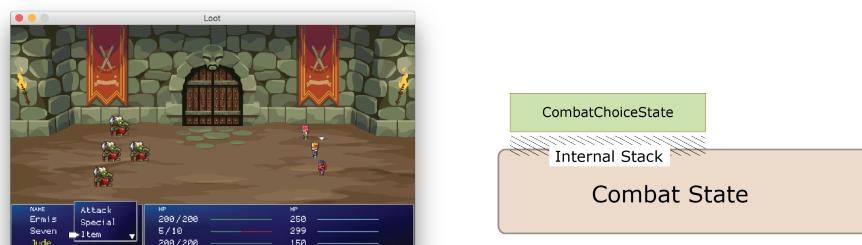


Figure 3.68: How the internal combat state changes as an item is used.

Create a new file, BrowseListState.lua, and add it the dependencies and manifest files. Then copy the code for the constructor from Listing 3.329.

```
BrowseListState = {}
BrowseListState.__index = BrowseListState
function BrowseListState:Create(params)

    params = params or {}

    local this =
    {
        mStack = params.stack,
        mX = params.x or 0,
        mY = params.y or 0,
        mWidth = params.width or 264,
        mHeight = params.height or 75,
        mTitle = params.title or "LIST",

        mOnExit = params.OnExit or function() end,
        mOnFocus = params.OnFocus or function() end,
        mUpArrow = gWorld.mIcons:Get('uparrow'),
        mDownArrow = gWorld.mIcons:Get('downarrow'),
        mHide = false
    }

    local data = params.data or {}
    local columns = params.columns or 2
    local displayRows = params.rows or 3
    local itemCount = math.max(columns, #data)
    local maxRows = math.max(displayRows, itemCount / columns)
    local selectCallback = params.OnSelection or function() end
    this.mSelection = Selection:Create
    {
        data = data,
        columns = columns,
        displayRows = displayRows,
        rows = maxRows,
        spacingX = 132,
        spacingY = 19,
        OnSelection = function(...) selectCallback(this, ...) end,
        RenderItem = params.OnRenderItem
    }
}
```

```

    }

    setmetatable(this, self)

    this.mBox = this>CreatePanel(this.mX, this.mY, this.mWidth, this.mHeight)
    this:SetArrowPosition()
    this.mSelection:SetPosition(this.mX - 24, this.mY - 24)

    return this
end

```

Listing 3.329: Creating BrowseListState constructor. In BrowseListState.lua.

The constructor takes a single table of parameters, `params`. This table contains data for the selection menu, the callbacks, and the stack.

Here are the possible parameters in full:

- **stack** - The stack the BrowseListState is pushed onto. Used to pop itself off when Backspace is pressed.
- **x, y** - The x and y position of the box, positioned from the top left.
- **width, height** - Dimensions of the backing panel.
- **title** - Title displayed at the top of the panel.
- **data** - Data source for the list.
- **columns** - How many columns the data should be formatted in.
- **rows** - Number of rows to display.
- **OnSelection** - Called when the user presses Enter on a selection.
- **OnRenderItem** - Render callback for each item of the selection.
- **OnExit** - Callback triggered when the state is exited.
- **OnFocus** - Callback for when the player changes the current selection. The new selection is passed to this callback.

All of these parameters have sensible defaults but the stack is required.

The stack, position, title, width and height are added to the `this` table. The `OnExit` and `OnFocus` callbacks are also added to `this` table, as are the Up and Down arrow sprites to indicate if there is more data in the list. The `this` table has an `mHide` field set to false that can hide the `BrowseListState` without popping it off the stack. We hide the box when the player is prompted to select a target for an item.

Near the end of the constructor we work out the values for the selection box, including calculating the number of rows required to store all the data. The selection menu is then created and assigned to `mSelection`. Finally the backing panel is created and the arrow positions are set.

Let's continue by adding the function to create the backing panel and the function to set the position of the arrows. Copy the code from Listing 3.330.

```

function BrowseListState:SetArrowPosition()
    local arrowPad = 9
    local arrowX = self.mX + self.mWidth - arrowPad
    self.mUpArrow:SetPosition(arrowX, self.mY - arrowPad)
    self.mDownArrow:SetPosition(arrowX, self.mY - self.mHeight + arrowPad)
end

function BrowseListState>CreatePanel(x, y, width, height)

    local panel = Panel:Create
    {
        texture = Texture.Find("gradient_panel.png"),
        size = 3,
    }
    panel:Position(x, y, x + width, y - height)

    return panel
end

```

Listing 3.330: BrowseListState panel creation function and arrow positioner. In BrowseListState.lua.

The SetArrowPosition function positions the arrow sprites on the right of the box at the top and bottom. The CreatePanel function creates a new panel using the x, y, width and height parameters.

Let's finish off the BrowseListState class. Copy the code from Listing 3.331.

```

function BrowseListState:Enter()
    self.mOnFocus(self.mSelection:SelectedItem())
end

function BrowseListState:Exit()
    self.mOnExit()
end

function BrowseListState:Update(dt) end

function BrowseListState:Hide()
    self.mHide = true
end

function BrowseListState>Show()
    self.mHide = false
    self.mOnFocus(self.mSelection:SelectedItem())
end

```

```

end

function BrowseListState:Render(renderer)

    if self.mHide then
        return
    end

    self.mBox:Render(renderer)

    if self.mSelection:CanScrollUp() then
        renderer:DrawSprite(self.mUpArrow)
    end
    if self.mSelection:CanScrollDown() then
        renderer:DrawSprite(self.mDownArrow)
    end

    renderer:ScaleText(1.1, 1.1)
    renderer:AlignText("left", "top")
    local shadow = Vector.Create(0,0,0,1)
    renderer:DrawText2d(self.mX + 6, self.mY - 3, self.mTitle, shadow)
    renderer:DrawText2d(self.mX + 5, self.mY - 2, self.mTitle)

    renderer:AlignText("left", "center")
    renderer:ScaleText(1.0, 1.0)
    self.mSelection:Render(renderer)

end

function BrowseListState:HandleInput()
    if Keyboard.JustPressed(KEY_BACKSPACE) then
        self.mStack:Pop()
        return
    end
    local prevIndex = self.mSelection:GetIndex()

    self.mSelection:HandleInput()

    if prevIndex ~= self.mSelection:GetIndex() then
        self.mOnFocus(self.mSelection:SelectedItem())
    end
end

```

Listing 3.331: BrowseListState functions. In BrowseListState.lua.

The Enter function calls the OnFocus callback and passes in the currently selected item. The Exit function calls the mOnExit callback. The Update function does nothing. The Hide and Show functions toggle the mHide flag. The Show flags calls the OnFocus callback.

The Render function returns immediately if the mHide flag is true, rendering nothing. Otherwise the backing panel, mBox, is rendered. The up and down arrows are rendered if the selection box can be scrolled up or down. The title text is drawn near the top of the panel with a drop shadow and then the selection menu is rendered.

If Backspace is pressed, the HandleInput function pops the BrowseListState off the stack. Otherwise the selection menu is told to handle any input. If the menu index changes then the OnFocus callback is called, passing in the newly focused item.

That's it for the BrowseListState class. We're ready to integrate it into the CombatChoiceState.

Item Action in CombatChoiceState

When the "Items" action is selected, we need to create the BrowseListState. Copy the code from Listing 3.332.

```
function CombatChoiceState:OnSelect(index, data)

    -- code omitted

    elseif data == "flee" then

        -- code omitted

    elseif data == "items" then

        self:OnItemAction()

    end
end
```

Listing 3.332: Adding code for when an item is selected. In CombatChoiceState.lua.

The OnSelect callback now recognizes when the "Items" action has been selected and calls a new OnItemAction function.

Let's implement the OnItemAction function, which sets up the BrowseListState. The BrowseListState uses a lot of callbacks, and we implement these inside the function. Copy the code from Listing 3.333.

```

function CombatChoiceState:OnItemAction()

    -- 1. Get the filtered item list
    local filter = function(def)
        return def.type == "useable"
    end
    local filteredItems = gWorld:FilterItems(filter)

    -- 2. Create the selection box
    local x = self.mTextbox.mSize.left - 64
    local y = self.mTextbox.mSize.top
    self.mSelection:HideCursor()

    local OnFocus = function(item)
        local text = ""
        if item then
            local def = ItemDB[item.id]
            text = def.description
        end
        self.mCombatState:ShowTip(text)
    end

    local OnExit = function()
        self.mCombatState:HideTip("")
        self.mSelection>ShowCursor()
    end

    local OnRenderItem = function(self, renderer, x, y, item)
        local text = "--"
        if item then
            local def = ItemDB[item.id]
            text = def.name
            if item.count > 1 then
                text = string.format("%s x%02d", def.name, item.count)
            end
        end
        renderer:DrawText2d(x, y, text)
    end

    local OnSelection = function(selection, index, item)
        if not item then
            return
        end

```

```

local def = ItemDB[item.id]
local targeter = self:CreateItemTargeter(def, selection)
self.mStack:Push(targeter)
end

local state = BrowseListState:Create
{
    stack = self.mStack,
    title = "ITEMS",
    x = x,
    y = y,
    data = filteredItems,
    OnExit = OnExit,
    OnRenderItem = OnRenderItem,
    OnFocus = OnFocus,
    OnSelection = OnSelection,
}
self.mStack:Push(state)

end

```

Listing 3.333: Implementing OnItemAction. In CombatChoiceState.lua.

In the OnItemAction we use a filter to get a list of all the “useable” items in the inventory. This is the list we show the player. We take the CombatChoiceState position and offset it 64 pixels to the left, where we’ll render our list of items. We also hide the selection cursor so there’s no confusion about which UI element the player is interacting with.

The first callback we implement is OnFocus. Every time the player selects an item, we change the tooltip message to describe what it does using the OnFocus callback. The OnExit callback is called when the player backs out of the BrowseListState. This hides the tooltip completely and shows the CombatChoiceState cursor so a new action can be chosen.

The OnRenderItem function is called for each item that’s displayed. If there’s no item the text “–” is drawn. Otherwise the item name is rendered with the item count. The OnSelection callback is called when the player selects an item. If the player selects an empty entry, nothing happens. If there *is* an item, then we push a CombatTargetState onto the stack, so the player can choose the target to use the item on.

With the callbacks defined, actually creating the BrowseListState is only a few lines of code. Once we push it onto the stack, it’s displayed to the player. Refer to Figure 3.68 to get a visual feel for how this works in regards to the CombatState.

At this point, the player can choose the “Items” action and see a list of useable items. If the player tries to use an item, it all falls apart because we haven’t defined the CreateItemTargeter function. Let’s define it now. This function creates a

CombatTargetState using the information in the item definition. Copy the code from Listing 3.334.

```
function CombatChoiceState:CreateItemTargeter(def, browseState)

    local targetDef = def.use.target

    self.mCombatState:ShowTip(def.use.hint)
    browseState:Hide()
    self:Hide()

    local OnSelect = function(targets)
        self.mStack:Pop() -- target state
        self.mStack:Pop() -- item box state
        self.mStack:Pop() -- action state

        local queue = self.mCombatState.mEventQueue
        local event = CEUseItem:Create(self.mCombatState,
                                       self.mActor,
                                       def,
                                       targets)
        local tp = event:TimePoints(queue)
        queue:Add(event, tp)

    end

    local OnExit = function()
        browseState:Show()
        self:Show()
    end

    return CombatTargetState>Create(self.mCombatState,
    {
        targetType = targetDef.type,
        defaultSelector = CombatSelector[targetDef.selector],
        switchSides = targetDef.switch_sides,
        OnSelect = OnSelect,
        OnExit = OnExit
    })
end
```

Listing 3.334: CreateItemTargeter creates a suitable selector for a given item. In CombatChoiceState.lua.

In CreateItemTargeter we store the targeting information from the item definition in a

local variable, targetDef. Then we change the tooltip from the item description to the hint about targeting.

When we enter the CombatTargetState we hide the browseable list and the list of combat actions. When we exit, the OnExit callback gets called and shows the browseable list and combat actions again. This means that if we back out of the targeting state by hitting Backspace, we'll be back in the item browser.

Next we define the OnSelect callback. It's called when we select a target to use the item on. When a target is selected, we pop the stack three times; once to remove the targeter, once to remove the item browser, and one last time to remove the combat actions box. At this point the combat stack is empty and the event queue will start running again. With the stack cleared, we create a CEUseItem event and add it to the event queue.

Everything we do in combat takes the form of an event. That includes using items. To use an item we use the CEItemUse event.

The CEUseItem

The CEUseItem event is where the magic happens. When executed it performs the following steps:

1. Tells the actor to walk forward.
2. Plays a use item animation (with an effect).
3. Displays a notice dialog with the item name at the top of the screen.
4. Applies the item effect.
5. Waits for the effect to finish and dismisses the notice.
6. Walks back.

That's a lot of steps! As with CEFlee we manage the steps with a storyboard. It's clear we need to do some prep work. First, we need to define a special effect, used when a character uses an item.

Special Effects Definitions

Different items trigger different special effects when used. Let's define some of these special effects. We'll begin by adding some new art files to the manifest. Copy the code from Listing 3.335.

```
textures =
{
    -- code omitted
    ['fx_use_item.png'] =
    {
```

```

        path = "art/fx_use_item.png",
},
['fx_restore_hp.png'] =
{
    path = "art/fx_restore_hp.png",
},
['fx_restore_mp.png'] =
{
    path = "art/fx_restore_mp.png",
},
['fx_revive.png'] =
{
    path = "art/fx_revive.png",
},
}
,
```

Listing 3.335: Adding the textures for the combat effects. In manifest.lua.

These files are all in the art directory of advanced-combat-3. The fx_use_item.png is displayed above the actor when an item is used. The other fx may or may not be played, depending on what an item does. To use these special effects, we need to add them to the EntityDefs.lua file. Copy the code from Listing 3.336.

```

gEntities =
{
    -- code omitted
    fx_restore_hp =
    {
        texture = "fx_restore_hp.png",
        width = 16,
        height = 16,
        startFrame = 1,
        frames = {1, 2, 3, 4, 5}
    },
    fx_restore_mp =
    {
        texture = "fx_restore_mp.png",
        width = 16,
        height = 16,
        startFrame = 1,
        frames = {1, 2, 3, 4, 5, 6}
    },
    fx_revive =
    {
        texture = "fx_revive.png",

```

```

        width = 16,
        height = 16,
        startFrame = 1,
        frames = {1, 2, 3, 4, 5, 6, 7, 8}
    },
    fx_use_item =
    {
        texture = "fx_use_item.png",
        width = 16,
        height = 16,
        startFrame = 1,
        frames = {1, 2, 3, 4, 4, 3, 2, 1}
    }
}

```

Listing 3.336: Adding the combat effects to the entity defs. In EntityDefs.lua.

In Figure 3.69 you can see the textures used in Listing 3.336.

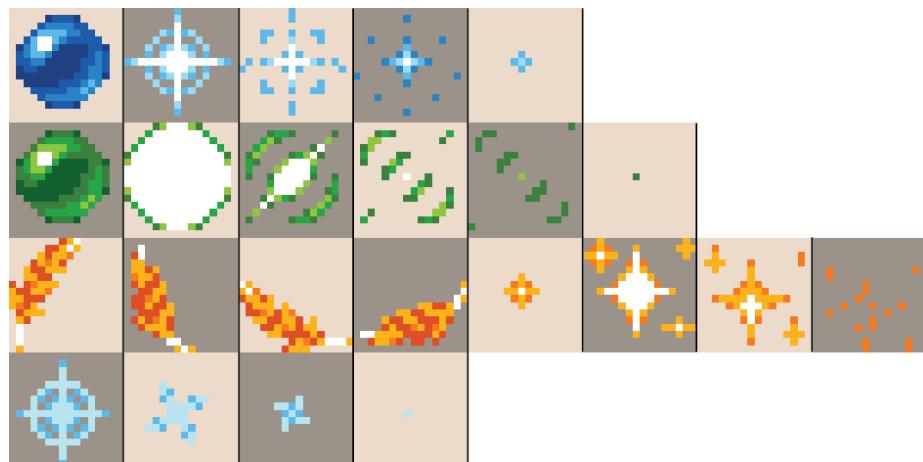


Figure 3.69: The frames of the special effects for the items.

Each of the effects is a strip of 16 x 16 pixel frames. Some of the effects are longer than others. For the item-use effect, we run the frames forward and then in reverse. This makes it appear as if the little blue twinkle is appearing and then disappearing.

Item Actions

Items can perform many actions during combat. We need a way to define item actions in code and then to link items and actions together. To do this, we'll create a file called

CombatActions.lua and store all the item functions in one place. Create CombatActions.lua and add it to the manifest and dependencies files.

Let's begin by adding some utility functions that are common to a lot of the item action functions. Copy the code from Listing 3.337.

```
local function AddAnimEffect(state, entity, def, spf)
    local x = entity.mX
    local y = entity.mY + (entity.mHeight * 0.75)

    local effect = AnimEntityFx:Create(x, y, def, def.frames, spf)
    state:AddEffect(effect)
end

local function AddTextNumberEffect(state, entity, num, color)
    local x = entity.mX
    local y = entity.mY

    local text = string.format("+%d", num)
    local textEffect = CombatTextFx:Create(x, y, text, color)
    state:AddEffect(textEffect)
end

local function StatsCharEntity(state, actor)
    local stats = actor.mStats
    local character = state.mActorCharMap[actor]
    local entity = character.mEntity
    return stats, character, entity
end
```

Listing 3.337: Combat Action Utility functions. In CombatActions.lua.

AddAnimEffect adds an animated special effect to the CombatState just above an entity's sprite. The AddAnimEffect is called to display special effects above a target. The AddTextNumberEffect is similar but displays text. We'll use AddTextNumberEffect when we heal a character or restore their HP. The text is displayed over the middle of the sprite so it doesn't overlap any animation effects.

The StatsCharEntity helper function returns the stats, character, and entity for a given actor. We use these variables to play the special effects and apply the item's function.

Let's move on to defining the combat actions. We'll store all the item functions in one big table called CombatActions. Each function is given a name, the key, and stores a function, the value. All functions take in the same parameters: state, owner, targets, and def. These are the combat state, the actor using the item, the actors that are

targets, and the item definition. Copy the code from Listing 3.338 to add the first set of item functions.

```
CombatActions =
{
    ["hp_restore"] =
        function(state, owner, targets, def)

            local restoreAmount = def.use.restore or 250
            local animEffect = gEntities.fx_restore_hp
            local restoreColor = Vector.Create(0, 1, 0, 1)

            for k, v in ipairs(targets) do

                local stats, character, entity = StatsCharEntity(state, v)

                local maxHP = stats:Get("hp_max")
                local nowHP = stats:Get("hp_now")

                if nowHP > 0 then
                    AddTextNumberEffect(state, entity, restoreAmount,
                        restoreColor)
                    nowHP = math.min(maxHP, nowHP + restoreAmount)
                    stats:Set("hp_now", nowHP)
                end

                AddAnimEffect(state, entity, animEffect, 0.1)
            end

        end,

    ['mp_restore'] =
        function(state, owner, targets, def)

            local restoreAmount = def.use.restore or 50
            local animEffect = gEntities.fx_restore_mp
            local restoreColor = Vector.Create(130/255, 200/255, 237/255, 1)

            for k, v in ipairs(targets) do

                local stats, character, entity = StatsCharEntity(state, v)

                local maxMP = stats:Get("mp_max")
                local nowMP = stats:Get("mp_now")
                local nowHP = stats:Get("hp_now")
```

```

        if nowHP > 0 then
            AddTextNumberEffect(state, entity, restoreAmount,
                restoreColor)
            nowMP = math.min(maxMP, nowMP + restoreAmount)
            stats:Set("mp_now", nowMP)
        end

        AddAnimEffect(state, entity, animEffect, 0.1)
    end
end,

['revive'] =
function(state, owner, targets, def)

    local restoreAmount = def.use.restore or 100
    local animEffect = gEntities.fx_revive
    local restoreColor = Vector.Create(0, 1, 0, 1)

    for k, v in ipairs(targets) do
        local stats, character, entity = StatsCharEntity(state, v)

        local maxHP = stats:Get("hp_max")
        local nowHP = stats:Get("hp_now")

        if nowHP == 0 then

            nowHP = math.min(maxHP, nowHP + restoreAmount)

            character.mController:Change(CSStandby.mName)

            stats:Set("hp_now", nowHP)

            AddTextNumberEffect(state, entity, restoreAmount, restoreColor)
        end

        AddAnimEffect(state, entity, animEffect, 0.1)
    end
end
}

```

Listing 3.338: A list of the item actions. In CombatActions.lua.

In Listing 3.338 we define three actions that our items can use.

Our first item action is hp_restore. HP restore plays an animation over all the targets and displays how much HP they've recovered. The hp_restore function gets the restore

amount from the item def or defaults to 250. It uses the "fx_restore_hp" as the animation effect to be played and sets the text color to green. We check the target's HP and only restore the HP of live characters, and we never increase it above the maximum HP amount.

The `mp_restore` is the same as `hp_restore` but applies to the MP stat. Once again, it only works if the target is alive.

The 'revive' action is applied to characters that are dead. It restores some of their HP and changes their state from CSDie to CStandby bringing them back into the game. On the next update they'll have a CETurn event pushed into the event queue automatically. Item actions only display text effects for recovered HP and MP if they succeed.

We're now ready to implement the `CEUseItem` event itself.

The Item Combat Event

Create and add `CEUseItem.lua` in the `combat_events` folder. Let's begin with the constructor. Copy the code from Listing 3.339.

```
CEUseItem = {}
CEUseItem.__index = CEUseItem
function CEUseItem:Create(state, owner, item, targets)

    local this =
    {
        mState = state,
        mOwner = owner,
        mItemDef = item,
        mTargets = targets,
        mIsFinished = false,
        mCharacter = state.mActorCharMap[owner],
    }

    gWorld:RemoveItem(item.id)

    this.mController = this.mCharacter.mController
    this.mController:Change(CSRunAnim.mName, {'prone'})

    local storyboard =
    {
        SOP.Function(function() this:ShowItemNotice() end),
        SOP.RunState(this.mController, CSMove.mName, {dir = 1}),
        SOP.RunState(this.mController, CSRunAnim.mName, {'use', false}),
        SOP.Function(function() this:DoUseItem() end),
        SOP.Wait(1.3), -- time to read the notice
    }
}
```

```

        SOP.RunState(this.mController, CSMove.mName, {dir = -1}),
        SOP.Function(function() this:DoFinish() end)
    }
    this.mStoryboard = Storyboard>Create(this.mState.mStack, storyboard)
    this.mName = string.format("%s using %s", owner.mName, item.name)

    setmetatable(this, self)
    return this
end

function CEUseItem:TimePoints(queue)
    local speed = self.mOwner.mStats:Get("speed")
    return queue:SpeedToTimePoints(speed)
end

```

Listing 3.339: CEUseItem constructor. In CEUseItem.lua.

The CEUseItem constructor takes in the combat state, the actor using the item, the item definition, and the targets as parameters. All the parameters are stored in the this table. We also add an mIsFinished flag set to false and a reference to the character.

The constructor immediately removes the item from the inventory. This means that multiple characters can't try to use the same item at once. It also means that if an actor dies before using an item, that item is gone forever! The constructor sets the actor to the prone state, to indicate that they're going to perform an action soon, then we create the storyboard.

The storyboard uses the Function operation we defined for CEFlee. First it displays a notice on screen with the name of the item being used. Then the actor walks toward the enemy and plays the use item animation. The DoUseItem function is called, there's a wait, then the actor steps back to the party. Finally DoFinish is called which marks the event as finished. The storyboard is stored in mStoryboard.

A debug name is set by combining the item and the actor name.

The TimePoints function uses the player's speed to calculate the number of time points the use item event takes. A more advanced implementation might give each item a different time point amount, but we'll keep things relatively simple!

Let's add the rest of the implementation. Copy the code from Listing 3.340.

```

function CEUseItem>ShowItemNotice()
    local str = string.format("Item: %s", self.mItemDef.name)
    self.mState>ShowNotice(str)
end

function CEUseItem:DoUseItem()

```

```

self.mState:HideNotice()
local pos = self.mCharacter.mEntity:GetSelectPosition()
local effect = AnimEntityFx>Create(pos:X(), pos:Y(),
                                    gEntities.fx_use_item,
                                    gEntities.fx_use_item.frames, 0.1)
self.mState:AddEffect(effect)

local action = self.mItemDef.use.action
CombatActions[action](self.mState,
                      self.mOwner,
                      self.mTargets,
                      self.mItemDef)
end

function CEUseItem:DoFinish()
    self.mIsFinished = true
end

function CEUseItem:Execute(queue)
    self.mState.mStack:Push(self.mStoryboard)
end

function CEUseItem:IsFinished()
    return self.mIsFinished
end

function CEUseItem:Update() end

```

Listing 3.340: CEUseItem storyboard and event functions. In CEUseItem.lua.

The ShowItemNotice function reveals the notice panel with the name of the item being used.

DoUseItem hides the notice panel, creates a use-effect above the entity, and plays it by calling the CombatState:AddEffect function. This draws blue sparkles over the entity, indicating that they're using an item. Then we get the action id from the item def and use it to call the action associated with the item.

DoFinish sets the mIsFinished flag to true. When the event queue next updates, it will remove the CEUseItem event from the queue and process whichever event is next. The Execute event pushes the story stack on top of the stack.



Figure 3.70: Using a potion during combat.

All the code so far is available in advanced-combat-3-solution. Try it out and you'll be able to use items during combat as shown in Figure 3.70!

Magic

That was a lot of code to get the "Item" action working. Luckily we can reuse some of it for the next action, "Magic". The "Magic" action lets our mage character cast spells. There's a new project available in advanced-combat-4 with all the code we've written so far.

Look at the party member definitions, in PartyMemberDefs.lua, and you'll see that only the mage has the "Magic" action. No other party members can use magic.

When "Magic" is selected, we open a browsable window showing all the spells and their MP costs. If the actor doesn't have enough MP to cast a spell, its entry is grayed out. Selecting a valid spell pushes a targeting state onto the stack. Once targets are chosen, a CECastSpell event is created and added to the event queue. The time point value differs from spell to spell depending on its power. When the CECastSpell event executes, the character walks forward, the program plays an animation, special effects fire off, the spell effect is applied, and the character walks back.

To fully implement the magic system we need the following:

- A combat formula or two for magical attacks.
- A list of spell definitions.
- A way to store available spells in the actor.
- Spell browsing code.
- Spell effect code.
- CECastSpell event.

For our game we'll give the mage three base spells: a fire spell, an ice spell, and an electric spell. To demonstrate spells that affect multiple targets, we'll also add one variant of the fire spell called "Burn".

Let's start by adding the combat formulas for magic.

Magic Attack Formula

Unlike the melee formula, we don't handle counteracting (or reflection) when dealing with magical attacks. Open up the CombatFormula.lua file and add the code in Listing 3.341.

```
function Formula.IsHitMagic(state, attacker, target, spell)
    -- Spell hit information determined by the spell
    local hitchance = spell.base_hit_chance
    if math.random() <= hitchance then
        return HitResult.Hit
    else
        return HitResult.Miss
    end
end

function Formula.CalcSpellDamage(state, attacker, target, spell)

    -- Find the basic damage
    local base = spell.base_damage
    if type(base) == 'table' then
        base = math.random(base[1], base[2])
    end

    local damage = base * 4

    -- Increase power of spell by caster
    local level = attacker.mLevel
    local stats = attacker.mStats

    local bonus = level * stats:Get("intelligence") * (base / 32)

    damage = damage + bonus

```

```

-- Apply elemental weakness / strength modifications
if spell.element then
    local modifier = target.mStats:Get(spell.element)
    damage = damage + (damage * modifier)
end

-- Handle resistance [0..255]
local resist = math.min(255, target.mStats:Get("resist"))
local resist01 = 1 - (resist / 255)
damage = damage * resist01

return math.floor(damage)
end

function Formula.MagicAttack(state, attacker, target, spell)

    local damage = 0
    local hitResult = Formula.IsHitMagic(state, attacker, target, spell)

    if hitResult == HitResult.Miss then
        return math.floor(damage), HitResult.Miss
    end

    -- Dodging spells not allowed.
    local damage = Formula.CalcSpellDamage(state, attacker, target, spell)

    return math.floor(damage), HitResult.Hit
end

```

Listing 3.341: Formulas for spell damage. In CombatFormula.lua.

The formulas in Listing 3.341 are used for offensive spells like fire, ice, etc. Other more obscure spells may use entirely different formulas.

The chance of a magic spell hitting the target is usually 100%. The hit chance is defined in the spell definition, which we've yet to write. The base_hit_chance is a number from 0 to 1. As always, 1 is 100% and 0 is 0%. To determine if the spell hits, we get a random number between 0 and 1, and if it's below the base_hit_chance it hits.

The CalcSpellDamage formula works out the damage for offensive magical spells. This code is based on a formula from the Final Fantasy games. It can be tweaked as desired for your own game. Here's the basic formula:

$$\text{Damage} = \text{Spell Power} * 4 + (\text{Level} * \text{Magic Power} * \text{Spell Power} / 32)$$

We begin by deciding on a basic damage amount for the spell. The base damage is defined in the spell def as either a single number or a range. If a range is defined, each time the spell is cast, a single number is randomly chosen from the range. Once we have the base damage, we multiply it by four. Then a bonus is applied according to the power of the caster. The bonus is calculated using the caster's intelligence, the caster's level, and the spell's base damage. After the bonus is applied, we have the raw damage that the spell inflicts. The target can resist some of this damage, depending on their stats.

After working out the base damage in `CalcSpellDamage` we check to see if the spell is elemental. The element information changes the effectiveness of the spell depending on the target. For instance, a demon might absorb fire damage, a machine might totally resist element attacks, etc. Elemental resistances are just stats that share the name of the element: "fire", "electric", "air", and "ice" in our case. A demon might have a "fire" stat of -1.5, which means the damage would be nullified and the fire element would absorb half the attack. A machine might have a "fire" stat of -1.0 meaning it would ignore all damage. A value of -0.5 would mean only half damage would be applied, and likewise if something was vulnerable to fire a value of 1.0 would mean the damage would be applied twice. The default value of 0 means no additional changes are made to the base damage value.

Damage	Damage Type	Target	Fire Resist	Electric Resist	Final Damage
10	Fire	Demon	-1.5	0	+5HP
10	None	Demon	-1.5	0	-10HP
10	Fire	Goblin	0	0	-10HP
10	Electric	Wild Mage	-0.5	-0.5	-5HP
10	Fire	Robot	-1	-1	0HP
10	Electric	Robot	-1	-1	0HP
10	Fire	Waterkin	1	1	-20HP

After elemental checks, there's a magical resistance check. Magical resistance is another stat, just called "resist", which can have a value of 0 - 255, usually 0. This number is converted to 0 to 1, which can be thought of as 0 to 100% reduction in damage. The final damage value is floored to remove any fractions and returned.

The `MagicAttack` formula does the full damage calculation for offensive spells. It checks if the spell hits, and returns the hit result and damage amount.

With these functions implemented, we can calculate how much damage a simple spell inflicts.

Magic Spell Definitions

The code in Listing 3.341 takes its number from a spell definition, but we haven't defined any spells yet. Let's add some definitions. Create a new file called SpellDB.lua. Add it to the manifest and dependencies. Then copy in the code from Listing 3.342.

```
SpellDB =
{
    ["fire"] =
    {
        name = "Fire",
        action = "element_spell",
        element = "fire",
        mp_cost = 8,
        cast_time = 1.25,
        base_damage = {5, 15}, -- multiplied by level
        base_hit_chance = 1,
        time_points = 10,
        target =
        {
            selector = "WeakestEnemy",
            switch_sides = true,
            type = "One"
        }
    },
    ["burn"] =
    {
        name = "Burn",
        action = "element_spell",
        element = "fire",
        mp_cost = 16,
        cast_time = 2,
        base_damage = {5, 15},
        base_hit_chance = 1,
        time_points = 20,
        target =
        {
            selector = "SideEnemy",
            switch_sides = true,
            type = "Side"
        }
    },
    ["ice"] =
    {
        name = "Ice",
        action = "element_spell",
        element = "ice",
        mp_cost = 12,
        cast_time = 1.5,
        base_damage = {5, 15},
        base_hit_chance = 1,
        time_points = 15,
        target =
        {
            selector = "StrongestEnemy",
            switch_sides = true,
            type = "Both"
        }
    }
}
```

```

element = "ice",
mp_cost = 8,
cast_time = 1.8,
base_damage = {7, 17},
base_hit_chance = 1,
time_points = 10,
target =
{
    selector = "WeakestEnemy",
    switch_sides = true,
    type = "One"
},
["bolt"] =
{
    name = "Bolt",
    action = "element_spell",
    element = "electric",
    mp_cost = 8,
    cast_time = 1.1,
    base_damage = {4, 14},
    base_hit_chance = 1,
    time_points = 10,
    target =
    {
        selector = "WeakestEnemy",
        switch_sides = true,
        type = "One"
    }
},
}

```

Listing 3.342: Adding some spell definitions. In SpellDB.lua.

The spell definitions are similar to the item definitions but with a few additions. Let's have a look at the fields.

- **name** - The name displayed in the browse spell box.
- **action** - Name of a combat action that applies the spell effect to the combat state.
- **element** - Extra data for the element_spell action. Describes the element of the spell. Optional.
- **mp_cost** - How much mana is required to cast the spell.
- **cast_time** - The time points required to cast the spell.

- **base_damage** - The basic range of damage to feed into the spell calculation. This can also be a single number.
- **base_hit_chance** - Spell's basic chance to hit; 1 here mean 100% chance.
- **target** - The targeting information, much the same as in the item definitions.

The fire, ice and bolt spells are basically the same with minor variations in speed and damage. The "Burn" spell is the same as the "Fire" spell but it works on all opponents, it's a tiny bit weaker, and it takes a little longer to cast. This is a good selection of spells for us to start with. To make the spells look cool, we need some animations. Let's add those next.

Magic Effect Animations

Each of our spells uses custom animations, so we need to include the sprites and entity definitions. In the art folder you'll find fx_electric.png, fx_ice.png and fx_fire.png. Add them to the manifest. Then we'll update the entity defs as in Listing 3.343.

```
gEntities =
{
    -- code omitted
    fx_fire =
    {
        texture = "fx_fire.png",
        width = 32,
        height = 48,
        startFrame = 1,
        frames = {1, 2, 3}
    },
    fx_electric =
    {
        texture = "fx_electric.png",
        width = 32,
        height = 16,
        startFrame = 1,
        frames = {1, 2, 3}
    },
    fx_ice_1 =
    {
        texture = "fx_ice.png",
        width = 16,
        height = 16,
        startFrame = 1,
        frames = {1, 2, 3, 4}
    },
    fx_ice_2 =
```

```

{
    texture = "fx_ice.png",
    width = 16,
    height = 16,
    startFrame = 5,
    frames = {5, 6, 7, 8}
},
fx_ice_3 =
{
    texture = "fx_ice.png",
    width = 16,
    height = 16,
    startFrame = 9,
    frames = {9, 10, 11, 12}
},
fx_ice_spark =
{
    texture = "fx_ice.png",
    width = 16,
    height = 16,
    startFrame = 13,
    frames = {13, 14, 15, 16}
}
}

```

Listing 3.343: Defining the spell effect entities. In EntityDefs.lua.

The fire and electric animations are fairly straightforward sprite animations. The ice effect is more complicated and uses four different animations. All the animations play on top of the spell target. The code that plays the spell animations and applies the damage is stored in the CombatAction database, much like the item effect code.

Magic Spell Actions

We'll add the action "element_spell" to the CombatActions next. This action is associated with each of our spells, and the differences are taken from the spell def. Copy the code from Listing 3.344.

```

['element_spell'] =
function(state, owner, targets, def)

    for k, v in ipairs(targets) do
        local _, _, entity = StatsCharEntity(state, v)

```

```

local damage, hitResult = Formula.MagicAttack(state,
    owner, v, def)

if hitResult == HitResult.Hit then
    state:ApplyDamage(v, damage)
end

if def.element == "fire" then
    AddAnimEffect(state, entity, gEntities.fx_fire, 0.06)
elseif def.element == "electric" then
    AddAnimEffect(state, entity, gEntities.fx_electric, 0.12)
elseif def.element == "ice" then
    AddAnimEffect(state, entity, gEntities.fx_ice_1, 0.1)
    local x = entity.mX
    local y = entity.mY

    local spk = gEntities.fx_ice_spark
    local effect = AnimEntityFx:Create(x, y, spk,
        spk.frames, 0.12)
    state:AddEffect(effect)

    local x2 = x + entity.mWidth * 0.8
    local ice2 = gEntities.fx_ice_2
    effect = AnimEntityFx:Create(x2, y, ice2, ice2.frames, 0.1)
    state:AddEffect(effect)

    local x3 = x - entity.mWidth * 0.8
    local y3 = y - entity.mHeight * 0.6
    local ice3 = gEntities.fx_ice_3
    effect = AnimEntityFx:Create(x3, y3, ice3,
        ice3.frames, 0.1)
    state:AddEffect(effect)
end
end
end

```

Listing 3.344: The combat action for basic spell attacks. In CombatActions.lua.

The element_spell action takes in four arguments: combat state, caster, targets, and spell def. The basic action of this function is simple. Each target has a damage value calculated using the MagicAttack formula. This damage is then applied by calling CombatState:ApplyDamage. The remaining code handles the special effects.

We play the appropriate animation for the spell according to its element. The ice spell plays a sparkle animation on the center of the target and three ice shards around the target.

Magic Spell Unlocking

We need a way for entities to know what spells they have, so we'll define this in the actor definition file and have a table in the actor. All of our spells have unique names, as used in SpellDB.lua. We use these names to indicate if a spell is available or not. Let's start by adding a spell table to the actor def. Copy the code from Listing 3.345.

```
mage =
{
    -- code omitted
    magic = {"fire", "burn"}
},
```

Listing 3.345: Giving the mage some starting spells. In PartyMemberDefs.lua.

In the PartyMemberDefs table we add a magic table with an entry for "fire" and "burn" spells. This means the mage can use these spells right from the start of the game. You can add the others later for testing, but I want to keep some locked off for now.

The magic table needs to be read by the Actor class so it knows which spells are unlocked. Copy the code from Listing 3.346.

```
function Actor:Create(def, ...)
    local this =
    {
        -- code omitted
        mMagic = ShallowClone(def.magic or {})
    }
}
```

Listing 3.346: Adding magic spells to the actor. In Actor.lua.

Unlocked magic spells are stored in the mMagic table. We clone the magic table to prevent the definition from being altered when the an actor unlocks a new spell.

Magic Spell Combat UI

Now that we're tracking which spells a mage knows, we're ready to populate the spell browsing dialog box.

The spell browser is launched from the CombatChoiceState. Only actors with the "Magic" action are able to launch this dialog. The code is similar to how we created the "Item" action, and uses the same states as in Figure 3.68. We'll begin by adding code to check if the magic action is selected in the OnSelect callback. Copy the code from Listing 3.347.

```

function CombatChoiceState:OnSelect(index, data)

    if data == "attack" then
        -- code omitted
    elseif data == "magic" then

        self:OnMagicAction()

    elseif data == "special" then -- we'll be handling this action soon!
        -- Todo
    end

```

Listing 3.347: Handling the magic action selection during combat. In `CombatChoiceState.lua`.

If the player selects “Magic” then the function `OnMagicAction` gets called, so let’s implement that next. Copy the code from Listing 3.348.

```

function CombatChoiceState:OnMagicAction()

    local actor = self.mActor

    -- 2. Create the selection box
    local x = self.mTextbox.mSize.left - 64
    local y = self.mTextbox.mSize.top
    self.mSelection:HideCursor()

    local OnRenderItem = function(self, renderer, x, y, item)
        local text = "--"
        local cost = "0"
        local canCast = false
        local mp = actor.mStats:Get("mp_now")

        local color = Vector.Create(1,1,1,1)
        if item then
            local def = SpellDB[item]
            text = def.name
            cost = string.format("%d", def.mp_cost)

            canCast = mp >= def.mp_cost

            if not canCast then
                color = Vector.Create(0.7, 0.7, 0.7, 1)
            end
        end
    end

```

```

        renderer:AlignText("right", "center")
        renderer:DrawText2d(x + 96, y, cost, color)
    end
    renderer:AlignText("left", "center")
    renderer:DrawText2d(x, y, text, color)
end

local OnExit = function()
    self.mSelection:ShowCursor()
end

local OnSelection = function(selection, index, item)
    if not item then
        return
    end
    local def = SpellDB[item]
    local mp = actor.mStats:Get("mp_now")

    if mp < def.mp_cost then
        return
    end

    local targeter = self>CreateActionTargeter(def, selection,
                                                CECastSpell)
    self.mStack:Push(targeter)
end

local state = BrowseListState:Create
{
    stack = self.mStack,
    title = "MAGIC",
    x = x,
    y = y,
    data = actor.mMagic,
    OnExit = OnExit,
    OnRenderItem = OnRenderItem,
    OnSelection = OnSelection
}
self.mStack:Push(state)
end

```

Listing 3.348: OnMagicAction implementation. In CombatChoiceState.lua.

The OnMagicAction function sets up the spell browser and pushes it onto the stack. Like the "Item" action, we use the BrowseListState class.

In the OnMagicAction function we start by storing the actor casting the spell in a local called actor. Then we calculate the position for the spell browser. We hide the cursor on the action menu, so the player isn't confused about which UI element they're interacting with. Before creating the spell browser we define the callbacks to plug into it.

The OnRenderItem callback is called each time a spell entry is drawn in the browser. Unlike items, spells cost MP to use, so we write this cost next to the spell name. The default text color is set to white. The selection box may contain empty cells, in which case the name is set to “–” with an MP cost of “0”. If there *is* a spell, we look up the definition and extract the name and the cost. We store the name in a variable called text. Just because the character knows a spell doesn't mean they can cast it. They may have insufficient mana. We check the spell cost against the actor's MP, and if the actor cannot afford to cast the spell, the color variable is changed to gray. The spell name is drawn left aligned at the position passed in. The cost is drawn right aligned 96 pixels to the right of the name.

The OnExit callback is very simple. It just reenables the cursor for the action selection box.

The OnSelection callback is called when the player selects a spell to cast. If they select an empty cell, the callback does nothing. Otherwise we look up the spell definition in the spell database. Then we check if the player has enough mana. If not, we return immediately. Otherwise a targeter state is created and pushed onto the stack to choose the target(s) of the spell.

Copy the code from Listing 3.349 to add the spell targeter code.

```
function CombatChoiceState:CreateActionTargeter(def, browseState,
    combatEvent)

    local targetDef = def.target

    browseState:Hide()
    self:Hide()

    local OnSelect = function(targets)
        self.mStack:Pop() -- target state
        self.mStack:Pop() -- spell browse state
        self.mStack:Pop() -- action state

        local queue = self.mCombatState.mEventQueue
        local event = combatEvent>Create(self.mCombatState,
            self.mActor,
            def,
            targets)
        local tp = event:TimePoints(queue)
        queue:Add(event, tp)
    end
end
```

```

local OnExit = function()
    browseState:Show()
    self:Show()
end

return CombatTargetState>Create(self.mCombatState,
{
    targetType = targetDef.type,
    defaultSelector = CombatSelector[targetDef.selector],
    switchSides = targetDef.switch_sides,
    OnSelect = OnSelect,
    OnExit = OnExit
})
end

```

Listing 3.349: The Spell targeter. In CombatChoiceState.lua.

The CreateActionTargeter function is similar to the one we wrote for the item, but no hint information is displayed. We'll use this function for both spell and special moves targets. In this case the combatEvent passed into the CreateActionTargeter is set to CECastSpell, a cast spell combat event. The targeter is created using the spell def's target field. Once the targets are chosen, a cast spell event is created and added to the event queue.

Run the game and you'll be able to check out the magic menu. To make it easier to test, increase the mage's speed stat in PartyMemberDefs so they get the first turn. We can browse the spell list, but it will crash when selecting a target. We need to define the CECastSpell event.

CECastSpell

Casting a spell, like using an item, involves a number of steps which we handle with a storyboard. CECastSpell works well for basic spells but to implement a more complicated spell, like a summon spell, it would better to create an entirely new event.

The mage has a special animation for casting a spell. We'll add the frames to end of the anims table in EntityDefs.lua.

```

gCharacters =
{
    -- code omitted
    mage =
    {
        -- code omitted

```

```

anim = 
{
    -- code omitted
    cast = {11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 25},
}

```

Listing 3.350: Adding the cast spell animation. In EntityDefs.lua

Create and add CECastSpell.lua in the combat_events directory. Then add the file to the dependencies and manifest files.

Copy the CECastSpell code from Listing 3.351.

```

CECastSpell = {}
CECastSpell.__index = CECastSpell
function CECastSpell:Create(state, owner, spell, targets)
    local this =
    {
        mState = state,
        mOwner = owner,
        mSpell = spell,
        mTargets = targets,
        mIsFinished = false,
        mCharacter = state.mActorCharMap[owner],
    }

    this.mController = this.mCharacter.mController
    this.mController:Change(CSRunAnim.mName, {'prone', true})

    local storyboard =
    {
        SOP.Function(function() this:ShowSpellNotice() end),
        SOP.RunState(this.mController, CSMove.mName, {dir = 1}),
        SOP.Wait(0.5),
        SOP.RunState(this.mController, CSRunAnim.mName, {'cast', false}),
        SOP.Wait(0.12),
        SOP.NoBlock(
            SOP.RunState(this.mController, CSRunAnim.mName, {'prone'})
        ),
        SOP.Function(function() this:DoCast() end),
        SOP.Wait(1.0),
        SOP.Function(function() this:HideSpellNotice() end),
        SOP.RunState(this.mController, CSMove.mName, {dir = -1}),
        SOP.Function(function() this:DoFinish() end)
    }
    this.mStoryboard = Storyboard.Create(this.mState.mStack, storyboard)
}

```

```

this.mName = string.format("%s is casting %s",
    spell.name, this.mOwner.mName)

setmetatable(this, self)
return this
end

function CECastSpell:TimePoints(queue)
    local speed = self.mOwner.mStats:Get("speed")
    local tp = queue:SpeedToTimePoints(speed)
    return tp + self.mSpell.time_points
end

function CECastSpell>ShowSpellNotice()
    self.mState:ShowNotice(self.mSpell.name)
end

function CECastSpell:HideSpellNotice()
    self.mState:HideNotice()
end

function CECastSpell:DoCast()
    local pos = self.mCharacter.mEntity:GetSelectPosition()
    local effect = AnimEntityFx>Create(pos:X(), pos:Y(),
        gEntities.fx_use_item,
        gEntities.fx_use_item.frames, 0.1)
    self.mState:AddEffect(effect)

    local mp = self.mOwner.mStats:Get("mp_now")
    local cost = self.mSpell.mp_cost
    local mp = math.max(mp - cost, 0) -- don't handle fizzle

    self.mOwner.mStats:Set("mp_now", mp)

    local action = self.mSpell.action
    CombatActions[action](self.mState,
        self.mOwner,
        self.mTargets,
        self.mSpell)
end

function CECastSpell:DoFinish()
    self.mIsFinished = true
end

```

```

function CECastSpell:IsFinished()
    return self.mIsFinished
end

function CECastSpell:Update(dt) end

function CECastSpell:Execute(queue)
    self.mState.mStack:Push(self.mStoryboard)

    for i = #self.mTargets, 1, -1 do
        local v = self.mTargets[i]
        local hp = v.mStats:Get("hp_now")
        local isEnemy = not self.mState:IsPartyMember(v)
        if isEnemy and hp <= 0 then
            table.remove(self.mTargets, i)
        end
    end

    if not next(self.mTargets) then
        local selector = CombatSelector[self.mSpell.target.selector]
        self.mTargets = selector(self.mState)
    end
end

```

Listing 3.351: Implementing the cast spell combat event. In CECastSpell.lua.

The code is similar to the use item event, but there are some notable differences. When the actor steps out, the notice displays the spell name. The notice is hidden after the cast animation is played.

The DoCast function plays the *use item* effect above the entity to signify they're doing something. The actor's MP is reduced, but there's no check to see if they have enough MP at this point. It's assumed there is enough. The combat action, associated with the spell, is retrieved and executed.

The time points for CECastSpell takes the actor's speed value added to the time point value for the spell.

The Execute function pushes the storyboard onto the stack. It also removes all enemies that have died from the target list. If no targets remain, then the spell's selector is used to try to find a new target.



Figure 3.71: Casting the burn spell.

Run the code and you'll be able to see the spells in action, as shown in Figure 3.71. There's a finished example in advanced-combat-4-solution.

Special Moves

A special move is an attack or ability that may require MP to perform. We're going to add two special abilities:

- **Slash** - An attack that hits all enemies.
- **Steal** - A chance to take an item from an enemy.

Implementing these two special moves should give you a good idea of how to implement additional abilities in your own game.

Let's start with slash. There's a project called advanced-combat-5 containing our code so far. We'll begin by enabling the abilities in the actors. For the hero we'll add slash, and for the thief we'll add steal. Modify the PartyMemberDefs.lua file as shown in Listing 3.352.

```

gPartyMemberDefs =
{
    hero =
    {
        -- code omitted
        special = { "slash" }
    },
    thief =
    {
        -- code omitted
        special = { "steal" }
    }
}

```

Listing 3.352: Adding special abilities for the hero and thief. In PartyMemberDefs.lua.

The new data in the definition needs storing in the Actor class. Special abilities are just a list of unique names that refer to an ability. We haven't defined how "slash" or "steal" work yet, but we'll add that as we go.

Copy the code from Listing 3.353.

```

function Actor:Create(def)

    -- code omitted

    local this =
    {
        -- code omitted
        mSpecial = ShallowClone(def.special or {})
    }

```

Listing 3.353: Storing the special attack information. In Actor.lua.

The special table is added the same way we added the magic table. The special abilities are cloned from the definition, so during the game the definition file won't be changed. If the actor definition doesn't define any special abilities, an empty table is used.

We need to let the player select the "Special" action during combat, and that means editing the CombatChoiceState.

Copy the code from Listing 3.354.

```

function CombatChoiceState:OnSelect(index, data)

    if data == "attack" then

```

```

-- code omitted

elseif data == "special" then

    self:OnSpecialAction()

end
end

```

*Listing 3.354: Creating a special function to handle special abilities in the combat UI.
In CombatChoiceState.lua.*

Selecting “Special” brings up a list of special moves for the player to choose from. We set up the special move browser in the function OnSpecialAttack. When the player chooses “Special” in the combat action menu, this function is called.

The browser dialog displays the special abilities along with their MP cost. Currently we don’t know the MP cost, so it needs defining somewhere! For the spells we had SpellDB.lua. For the specials we’ll create SpecialDB.lua. Make this lua file and add it to the manifest and dependencies files.

Copy the definitions from Listing 3.355.

```

SpecialDB =
{
    ['slash'] =
    {
        name = "Slash",
        mp_cost = 15,
        action = "slash",
        time_points = 20,
        target =
        {
            selector = "SideEnemy",
            switch_sides = false,
            type = "Side"
        }
    },
    ['steal'] =
    {
        name = "Steal",
        mp_cost = 0,
        action = "steal",
        time_points = 20,
        target =
    }
}

```

```

    {
        selector = "WeakestEnemy",
        switch_sides = false,
        type = "One"
    }
}
}
}

```

Listing 3.355: Defining special ability data. In SpecialDB.lua.

The slash ability costs 15 mp and targets the entire enemy side. It's associated with an action called "slash" that we've yet to define. The steal ability costs no mana, only targets a single enemy, and is associated with a "steal" action that we've also yet to define. Both abilities can only target the enemy side. With the "mp_cost" defined, we can create our browser dialog. Copy the code from Listing 3.356.

```

function CombatChoiceState:OnSpecialAction()

    local actor = self.mActor

    -- Create the selection box
    local x = self.mTextbox.mSize.left - 64
    local y = self.mTextbox.mSize.top
    self.mSelection:HideCursor()

    local OnRenderItem = function(self, renderer, x, y, item)
        local text = "--"
        local cost = "0"
        local canPerform = false
        local mp = actor.mStats:Get("mp_now")

        local color = Vector.Create(1,1,1,1)
        if item then
            local def = SpecialDB[item]
            text = def.name
            cost = string.format("%d", def.mp_cost)

            if def.mp_cost > 0 then

                canPerform = mp >= def.mp_cost

                if not canPerform then
                    color = Vector.Create(0.7, 0.7, 0.7, 1)
                end
            end
        end
        renderer:DrawText(x, y, text, color)
    end
end

```

```

        renderer:AlignText("right", "center")
        renderer:DrawText2d(x + 96, y, cost, color)
    end
end
renderer:AlignText("left", "center")
renderer:DrawText2d(x, y, text, color)
end

local OnExit = function()
    self.mCombatState:HideTip("")
    self.mSelection:ShowCursor()
end

local OnSelection = function(selection, index, item)
    if not item then
        return
    end
    local def = SpecialDB[item]
    local mp = actor.mStats:Get("mp_now")

    if mp < def.mp_cost then
        return
    end

    -- Find associated event
    local event = nil
    if def.action == "slash" then
        event = CESlash
    elseif def.action == "steal" then
        event = CESteal
    end

    local targeter = self>CreateActionTargeter(def, selection, event)
    self.mStack:Push(targeter)
end

local state = BrowseListState>Create
{
    stack = self.mStack,
    title = "Special",
    x = x,
    y = y,
    data = actor.mSpecial,
    OnExit = OnExit,
    OnRenderItem = OnRenderItem,
}

```

```

        OnSelection = OnSelection
    }
    self.mStack:Push(state)
end

```

Listing 3.356: The Special ability browser. In CombatChoiceState.lua.

The OnSpecialAction function sets up the special action browser. It assigns the actor to local variable actor and then creates the position for the browser. The cursor for the action menu is hidden. All the callbacks for the browsing state are then defined.

The OnRenderItem callback is similar to OnRenderItem for the magic spells, but some abilities have a mana cost of 0 and in this case the mana number isn't drawn.

The OnExit callback restores the cursor to the action menu.

The OnSelection callback looks up the special move in the special move database. It checks if the actor has enough MP to perform the move. If not, it returns immediately. If the actor does have the MP then the CombatEvent, associated with the special move, is assigned. A targeter is created using combat event and it's pushed onto the stack.

The end of the OnSpecialAction function creates the special action browser and pushes it onto the stack.

Run the code and you can browse the special moves, but if you try and select one it crashes because CESlash and CESteal haven't been defined. Let's define them next.

Create CESlash.lua and CESteal.lua files in the combat_event directory and add them to the manifest and dependencies.

CESlash

Slash is a special move and so has its own unique animation. Add the animation data for the hero in EntityDefs.lua as shown in Listing 3.357.

```

gCharacters =
{
    hero =
    {
        -- code omitted
        anims =
        {
            -- code omitted
            slash = {11, 12, 13, 14, 15, 16, 17, 18, 11},

```

Listing 3.357: Adding the slash animation for the hero. In EntityDefs.lua.

Then let's implement CESlash. Copy the code from Listing 3.358.

```
CESlash = {}
CESlash.__index = CESlash
function CESlash:Create(state, owner, def, targets)

    local this =
    {
        mState = state,
        mOwner = owner,
        mDef = def,
        mFinished = false,
        mCharacter = state.mActorCharMap[owner],
        mTargets = targets,
        mIsFinished = false
    }

    this.mController = this.mCharacter.mController
    this.mController:Change(CSRunAnim.mName, {'prone'})
    this.mName = string.format("Slash for %s", this.mOwner.mName)

    setmetatable(this, self)

    local storyboard = nil

    this.mAttackAnim = gEntities.slash
    this.mDefaultTargeter = CombatSelector.SideEnemy

    storyboard =
    {
        SOP.Function(function() this:ShowNotice() end),
        SOP.Wait(0.2),
        SOP.RunState(this.mController, CSMove.mName, {dir = 1}),
        SOP.RunState(this.mController, CSRunAnim.mName,
            {'slash', false, 0.05}),
        SOP.NoBlock(
            SOP.RunState(this.mController, CSRunAnim.mName, {'prone'})
        ),
        SOP.Function(function() this:DoAttack() end),
        SOP.RunState(this.mController, CSMove.mName, {dir = -1}),
        SOP.Wait(0.2),
        SOP.Function(function() this:OnFinish() end)
    }

    this.mStoryboard = Storyboard>Create(this.mState.mStack,
```

```

        storyboard)

    return this
end

function CESlash:TimePoints(queue)
    local speed = self.mOwner.mStats:Get("speed")
    local tp = queue:SpeedToTimePoints(speed)
    return tp + self.mDef.time_points
end

function CESlash:ShowNotice()
    self.mState:ShowNotice(self.mDef.name)
end

function CESlash:Execute(queue)
    self.mState.mStack:Push(self.mStoryboard)

    for i = #self.mTargets, 1, -1 do
        local v = self.mTargets[i]
        local hp = v.mStats:Get("hp_now")
        if hp <= 0 then
            table.remove(self.mTargets, i)
        end
    end

    if not next(self.mTargets) then
        -- Find another enemy
        self.mTargets = self.mDefaultTargeter(self.mState)
    end
end

function CESlash:OnFinish()
    self.mIsFinished = true
end

function CESlash:IsFinished()
    return self.mIsFinished
end

function CESlash:Update()
end

function CESlash:DoAttack()
    self.mState:HideNotice()

```

```

local mp = self.mOwner.mStats:Get("mp_now")
local cost = self.mDef.mp_cost
local mp = math.max(mp - cost, 0)

self.mOwner.mStats:Set("mp_now", mp)

for _, target in ipairs(self.mTargets) do
    self:AttackTarget(target)

    if not self.mDef.counter then
        self:CounterTarget(target)
    end
end
end

-- Decide if the attack is countered.
function CESlash:CounterTarget(target)
    local countered = Formula.IsCountered(self.mState, self.mOwner, target)
    if countered then
        self.mState:ApplyCounter(target, self.mOwner)
    end
end

function CESlash:AttackTarget(target)

    local damage, hitResult = Formula.MeleeAttack(self.mState,
                                                    self.mOwner,
                                                    target)

    local entity = self.mState.mActorCharMap[target].mEntity

    if hitResult == HitResult.Miss then
        self.mState:ApplyMiss(target)
        return
    elseif hitResult == HitResult.Dodge then
        self.mState:ApplyDodge(target)
    else
        local isCrit = hitResult == HitResult.Critical
        self.mState:ApplyDamage(target, damage, isCrit)
    end

    local x = entity.mX
    local y = entity.mY
    local effect = AnimEntityFx>Create(x, y,
                                        self.mAttackAnim,

```

```

        self.mAttackAnim.frames)

    self.mState:AddEffect(effect)
end

```

Listing 3.358: Implementing the combat slash event. In CESlash.lua.

For the slash, the actor walks forward, the slash animation plays, then the actor applies the attack to the targets and walks back. We could use the CEAttack event to do this, but making a separate CESlash class makes it easier to add special effects. The CESlash class is very similar to CEAttack but there's an added notice to announce the special attack, and there are also a few more Wait operations in the storyboard to give the player time to recognize that a special attack is happening.

In the DoAttack we reduce the actor's mana by the special move cost. There's no support for failing due to lack of mana.



Figure 3.72: The hero just after slashing multiple enemies.

Try it out! The attack is pretty devastating at the moment, as you can see from Figure 3.72. It still triggers counter attacks, so it can cause the user to take quite a bit of damage.

CESteal

The enemies currently have loot that they sometimes drop when they die. We could use steal to grab some of this loot, but I think it's more interesting if there are special items that can only be accessed using the steal action. Therefore we're going to extend the enemy def with a new parameter, `steal_item`. Each creature only ever has one stealable item, and this field may be null for enemies that have nothing to steal.

A thief can only steal an item from an enemy once. If the actor has nothing to steal, then the notice "There's nothing to steal." is displayed on screen.

Let's begin by adding a stealable potion to the goblin definition.

```
gEnemyDefs =
{
    goblin =
    {
        -- code omitted
        steal_item = 10,
```

Listing 3.359: Add the potion as a stealable item. In EnemyDefs.lua.

This `steal_item` is copied into the Actor class and stored locally when the enemy is created. Let's add the modification now. Copy the code from Listing 3.360.

```
Actor.__index = Actor
function Actor:Create(def)

    local growth = ShallowClone(def.statGrowth or {})

    local this =
    {
        -- code omitted
        mStealItem = def.steal_item,
    }
```

Listing 3.360: Adding a stealable item field to the actor. In Actor.lua.

The `mStealItem` field can be null. With these two changes made, each goblin now holds a potion that can be stolen during combat.

Let's add a new formula to CombatFormula.lua to calculate the steal chance. Copy the code from Listing 3.361.

```

function Formula.Steal(state, attacker, target)

    local cts = .05 -- 5%

    if attacker.mLevel > target.mLevel then
        cts = (50 + attacker.mLevel - target.mLevel)/128
        cts = Clamp(cts, .05, .95)
    end

    local rand = math.random()
    return rand <= cts
end

```

Listing 3.361: Calculating the chance to steal. In CombatFormula.lua.

This steal formula is roughly based on the formula in Final Fantasy 6. If the target is a higher level than the thief, the chance is 5%. If the thief is higher level than the target, then depending on the difference the thief has a chance from 5% to 95%. A random number is chosen. If it's below the steal number then it's a success, otherwise it's a failure.

The steal combat event is most sophisticated event we've implemented so far. The thief teleports behind the enemy, tries to steal, teleports back, and then we show if the theft was successful.

Let's add the animations for the theft. Copy the code from Listing 3.362.

```

gCharacters =
{
    -- code omitted
    thief =
    {
        -- code omitted
        anims =
        {
            -- code omitted
            steal_1 = {41, 42, 43, 44, 45},
            steal_2 = {46, 47, 48, 49, 50, 51, 52, 53},
            steal_3 = {49, 48, 43, 44, 45},
            steal_4 = {45, 44, 43, 42, 41},
            steal_success = {54},
            steal_failure = {55}
        }
    }
}

```

Listing 3.362: Adding the steal animations. In EntityDefs.lua.

There are a few animations here: `steal_1` animates the thief disappearing, `steal_2` animates the thief reappearing and doing the steal, `steal_3` animates the thief disappearing again.

Let's implement the event. Copy the code from Listing 3.363.

```
CESteal = {}
CESteal.__index = CESteal
function CESteal:Create(state, owner, def, targets)

    local this =
    {
        mState = state,
        mOwner = owner,
        mDef = def,
        mFinished = false,
        mCharacter = state.mActorCharMap[owner],
        mTargets = targets,
        mIsFinished = false,
        mSuccess = false,
    }

    this.mController = this.mCharacter.mController
    this.mController:Change(CSRunAnim.mName, {'prone'})
    this.mName = string.format("Steal for %s", this.mOwner.mName)

    this.mOriginalPos = this.mCharacter.mEntity.mSprite:GetPosition()

    setmetatable(this, self)

    local storyboard = nil

    this.mAnim = gEntities.slash
    this.mDefaultTargeter = CombatSelector.WeakestEnemy

    storyboard =
    {
        SOP.Function(function() this:ShowNotice() end),
        SOP.Wait(0.2),
        SOP.RunState(this.mController, CSMove.mName, {dir = 1}),
        SOP.RunState(this.mController, CSRunAnim.mName,
            {'steal_1', false}),
        SOP.Function(function() this:TeleportOut() end),
        SOP.RunState(this.mController, CSRunAnim.mName,
            {'steal_2', false}),
        SOP.Wait(1.1),
    }
}
```

```

SOP.Function(function() this:DoSteal() end),
SOP.RunState(this.mController, CSRunAnim.mName,
{'steal_3', false}),
SOP.Function(function() this:TeleportIn() end),
SOP.RunState(this.mController, CSRunAnim.mName,
{'steal_4', false}),
SOP.Function(function() this>ShowResult() end),
SOP.Wait(1.0),
SOP.Function(function() this.mState:HideNotice() end),
SOP.RunState(this.mController, CSMove.mName, {dir = -1}),
SOP.Wait(0.2),
SOP.Function(function() this:OnFinish() end)
}

this.mStoryboard = Storyboard>Create(this.mState.mStack,
storyboard)

return this
end

function CESteal:TeleportOut()
local target = self.mTargets[1]
local entity = self.mState.mActorCharMap[target].mEntity
local width = entity.mTexture:GetWidth() + 32

local pos = entity.mSprite:GetPosition()
pos:SetX(math.floor(pos:X() - width / 2))

self.mCharacter.mEntity.mSprite:SetPosition(pos)
end

function CESteal:ShowResult()
if self.mSuccess then
    self.mController:Change(CSRunAnim.mName, {'steal_success'})
else
    self.mController:Change(CSRunAnim.mName, {'steal_failure'})
end
end

function CESteal:TeleportIn()
self.mCharacter.mEntity.mSprite:SetPosition(self.mOriginalPos)
end

function CESteal:TimePoints(queue)
local speed = self.mOwner.mStats:Get("speed")

```

```

        return queue:SpeedToTimePoints(speed)
end

function CESteal:ShowNotice()
    self.mState:ShowNotice(self.mDef.name)
end

function CESteal:Execute(queue)
    self.mState.mStack:Push(self.mStoryboard)

    for i = #self.mTargets, 1, -1 do
        local v = self.mTargets[i]
        local hp = v.mStats:Get("hp_now")
        if hp <= 0 then
            table.remove(self.mTargets, i)
        end
    end

    if not next(self.mTargets) then
        -- Find another enemy
        self.mTargets = self.mDefaultTargeter(self.mState)
    end
end

function CESteal:OnFinish()
    self.mIsFinished = true
end

function CESteal:IsFinished()
    return self.mIsFinished
end

function CESteal:Update()
end

function CESteal:DoSteal()

    local target = self.mTargets[1]

    self.mState:HideNotice()

    if not target.mStealItem then
        self.mState:ShowNotice("Nothing to steal.")
        return
    end

```

```

end

self.mSuccess = self:StealFrom(target)

if self.mSuccess then
    local id = target.mStealItem
    local def = ItemDB[id]
    local name = def.name
    gWorld:AddItem(id)
    target.mStealItem = nil
    local notice = string.format("Stolen: %s.", name)
    self.mState:ShowNotice(notice)
else
    self.mState:ShowNotice("Steal failed.")
end

end

function CESteal:StealFrom(target)

local success = Formula.Steal(self.mState,
                               self.mOwner,
                               target)

local entity = self.mState.mActorCharMap[target].mEntity

local x = entity.mX
local y = entity.mY
local effect = AnimEntityFx>Create(x, y,
                                    self.mAnim,
                                    self.mAnim.frames)

self.mState:AddEffect(effect)

return success
end

```

Listing 3.363: Implementing the steal event. In CESteal.lua.

The steal event uses the same signature as other combat events, but this code only targets a single enemy. It uses a storyboard to handle the progress of the event. A notice is displayed on screen announcing the steal attempt, then the character walks forward, the character teleports to the enemy, the DoSteal function runs, the character teleports back, there's a wait, any notices are hidden, and the character walks back.

The DoSteal function determines if the steal is successful or not, and then displays another notice if the steal fails or succeeds, or if it can never succeed.

If the target we're stealing from doesn't have an mStealItem field, then there's nothing to steal. In this case we display the notice "Nothing to steal". If there's something to steal we call the StealFromTarget function, which runs the steal formula. If the steal succeeds or fails we play a steal animation over the target to show that the steal action has occurred. Back in DoSteal if the steal was unsuccessful we display a "Steal failed." notice.

When the steal action is successful, we need to update the inventory. First we take the item id from the mStealItem and use it to get the item definition. Then we set the target's mStealItem to nil because we've now stolen that item. We use the item def to post a notice telling the player which item they've stolen. We end by adding the stolen item to the party inventory.



Figure 3.73: The steal special ability in action.

Run the code and you can try it out. You should be able to fire off the steal action like in Figure 3.73. The code so far is available in advanced-combat-5-solution.

That's it for advanced combat. There's always more that can be done such as enemy abilities, status effects, etc. but with what we've covered in this section these should now be quite approachable. We're going end this section of the book with a small game called Arena.

The Arena

In the previous sections we've built a robust combat engine. A combat engine to be proud of! It's time to put that engine into a game. We'll make a simple arena type game. The player fights in a combat arena. If he wins, the next stage of the arena opens and gives the player access to an even more challenging battle. If the player loses, they're ejected from the arena. They can heal up outside, then try again. We'll let the player redo previous battles to allow a little bit of grinding.

A Slimmer Main File

Open example arena-1. This is the base project we'll develop from. This project contains the map with the arena. The map itself contains two party members that can be hired, and three chests of equipment. You can see the map in game in Figure 3.74.



Figure 3.74: The arena map with party members and chests of loot.

Our main.lua file only needs code related to the new arena game. Previous test code can now be removed. Make your main.lua appear as in Listing 3.364.

```
LoadLibrary('Asset')
Asset.Run('Dependencies.lua')
```

```

gRenderer = Renderer.Create()
gStack = StateStack.Create()
gWorld = World.Create()

gWorld.mParty:Add(Actor>Create(gPartyMemberDefs.hero))
gStack:Push(ExploreState>Create(gStack, CreateArenaMap(),
    Vector.Create(30, 18, 1)))

function update()
    local dt = GetDeltaTime()
    gStack:Update(dt)
    gStack:Render(gRenderer)
    gWorld:Update(dt)
end

```

Listing 3.364: Cleaning up our main.lua file for the Arena game. In main.lua.

The main.lua loads the arena. There's only one party member, the hero.

Sensible Stats

The party member stats are currently all over the place to help test the combat. It's time to assign more sensible defaults!

Copy the updated PartyMemberDefs code from Listing 3.365.

```

gPartyMemberDefs =
{
    hero =
    {
        id = "hero",
        stats =
        {
            ["hp_now"] = 36,
            ["hp_max"] = 36,
            ["mp_now"] = 5,
            ["mp_max"] = 5,
            ["strength"] = 10,
            ["speed"] = 16,
            ["intelligence"] = 10,
        },
        actionGrowth =
        {
            [5] =

```

```

{
    ['special'] = {'slash'},
}
},
statGrowth =
{
    ["hp_max"] = Dice:Create("2d25+25"),
    ["mp_max"] = Dice:Create("1d5+2"),
    ["strength"] = gStatGrowth.fast,
    ["speed"] = gStatGrowth.fast,
    ["intelligence"] = gStatGrowth.med,
},
portrait = "hero_portrait.png",
name = "Seven",
actions = { "attack", "item", "flee" },
level = 0,
},
thief =
{
    id = "thief",
    stats =
    {
        ["hp_now"] = 34,
        ["hp_max"] = 34,
        ["mp_now"] = 5,
        ["mp_max"] = 5,
        ["strength"] = 10,
        ["speed"] = 15,
        ["intelligence"] = 10,
    },
    statGrowth =
    {
        ["hp_max"] = Dice:Create("2d25+15"),
        ["mp_max"] = Dice:Create("2d10+5"),
        ["strength"] = gStatGrowth.med,
        ["speed"] = gStatGrowth.fast,
        ["intelligence"] = gStatGrowth.med,
    },
    actionGrowth =
    {
        [2] =
        {
            ['special'] = { 'steal' }
        }
    },
}
,
```

```

        portrait = "thief_portrait.png",
        name = "Jude",
        actions = { "attack", "item", "flee" },
        level = 0,
    },
    mage =
    {
        id = "mage",
        stats =
        {
            ["hp_now"] = 32,
            ["hp_max"] = 32,
            ["mp_now"] = 10,
            ["mp_max"] = 10,
            ["strength"] = 8,
            ["speed"] = 10,
            ["intelligence"] = 20,
        },
        statGrowth =
        {
            ["hp_max"] = Dice:Create("2d25+18"),
            ["mp_max"] = Dice:Create("1d5+2"),
            ["strength"] = gStatGrowth.med,
            ["speed"] = gStatGrowth.med,
            ["intelligence"] = gStatGrowth.fast,
        },
        actionGrowth =
        {
            [1] =
            {
                ['magic'] = { 'bolt' },
            },
            [2] =
            {
                ['magic'] = { 'fire', 'ice' }
            },
            [4] =
            {
                ['magic'] = { 'burn' }
            }
        },
        portrait = "mage_portrait.png",
        name = "Ermis",
        actions = { "attack", "item", "flee" },
        magic = { },
    }
}

```

```
    level = 0,  
},
```

Listing 3.365: Updating the party stats. In PartyMemberDefs.lua.

In Listing 3.365 the party member special abilities have been removed. Opening an empty menu is no fun, so we've also removed the "Special" action. We'll unlock these abilities as the player levels up. To determine which abilities are unlocked, we use a new actionGrowth table. This table lists the actions and abilities unlocked at each level.

The other change to the definition is a general reduction in stats. Our characters start on level 0, which means they'll gain a level immediately after winning their first battle.

The stats for the weapons and spells don't need changing. They're fine for the arena. The party inventory now starts empty. The player must search the map to find weapons.

We'll increase the goblin stats to make them a little stronger, and change the item they drop. Copy the code from Listing 3.366.

```
gEnemyDefs =  
{  
    goblin =  
    {  
        id = "goblin",  
        stats =  
        {  
            ["hp_now"] = 100,  
            ["hp_max"] = 100,  
            ["mp_now"] = 0,  
            ["mp_max"] = 0,  
            ["strength"] = 15,  
            ["speed"] = 6,  
            ["intelligence"] = 2,  
            ["counter"] = 0,  
        },  
        name = "Arena Goblin",  
        actions = { "attack" },  
        steal_item = 10,  
        drop =  
        {  
            xp = 150,  
            gold = {5, 15},  
            always = nil,  
            chance =  
            {  
                { oddment = 1, item = { id = -1 } },
```

```
        { oddment = 3, item = { id = 10 } }
    }
}
}
```

Listing 3.366: Beefing up the goblin stats. In EnemyDefs.lua.

The goblin is now stronger, faster, and has more health. We've upped the amount of gold dropped, but there are no guaranteed item drops. An arena needs more than one monster, so once we have the arena working we'll introduce another enemy type or two.

Locking Abilities

RPGs don't give out all the character abilities at the start of the game. Instead, they're found, bought, or unlocked as the party levels up. In our game, we unlock new abilities at certain levels. We've already locked off some abilities. Now we'll add code to unlock them on levelling up.

The best place to add our new code is in the Actor class's CreateLevelUp and ApplyLevel functions. Copy the code from Listing 3.367.

```
function Actor>CreateLevelUp()

    -- code omitted

    local level = self.mLevel + levelup.level
    local def = gPartyMemberDefs[self.mId]
    levelup.actions = def.actionGrowth[level] or {}

    return levelup
end

function Actor:ApplyLevel(levelup)

    -- code omitted

    for action, v in pairs(levelup.actions) do
        self:UnlockMenuAction(action)
        self:AddAction(action, v)
    end

    -- Restore HP and MP on level up
end
```

```

local maxHP = self.mStats:Get("hp_max")
local maxMP = self.mStats:Get("mp_max")
self.mStats:Set("mp_now", maxMP)
self.mStats:Set("hp_now", maxHP)

end

```

Listing 3.367: Code to unlock abilities for the player characters. In Actor.lua.

The CreateLevelUp function checks if any special abilities or spells are unlocked at this level. We add any unlocked abilities to the action table and unlock them using the ApplyLevel function. Unlock abilities are displayed on the XPSummaryState screen.

The ApplyLevel function loops over the levelup.action table and calls UnlockMenuAction and AddAction on each one. We'll implement these functions in the Actor class next. The ApplyLevelUp function also restores the actor's HP and MP.

Let's implement the new unlock functions. Copy the code from Listing 3.368.

```

function Actor:UnlockMenuAction(id)
    for _, v in ipairs(self.mActions) do
        if v == id then
            return
        end
    end
    table.insert(self.mActions, id)
end

function Actor:AddAction(action, entry)

    local t = self.mSpecial
    if action == 'magic' then
        t = self.mMagic
    end

    for _, v in ipairs(entry) do
        table.insert(t, v)
    end
end

```

Listing 3.368: New functions to unlock special abilities. In Actor.lua.

The UnlockMenuAction function unlocks any new action menus such as Magic or Special. It takes in one parameter, id. If the id doesn't exist in the mActions table we add it, which has the effect of adding a new action menu.

The AddAction function unlocks a new action inside a menu. It gets the relevant menu table in a hard coded way by testing the value of the action name. A table t is set to mSpecial by default or set to mMagic if the id equals "magic". The new action is then inserted into table t.

Together these functions let us unlock special abilities and spells as the actor levels up. Next we'll write code to notify the player each time a new ability or spell is gained.

Let's change the XPSummaryState so it reads the new actions table and displays a pop-up for each new ability or spell. Copy the code from Listing 3.369.

```
function XPSummaryState:UnlockPopUps(summary, actions)

    for k, v in pairs(actions) do

        local color = Vector.Create(1, 0.843, 0, 1)
        local db = SpecialDB
        if k == 'magic' then
            db = SpellDB
            color = Vector.Create(0.57, 0.43, 0.85, 1)
        end

        for _, id in ipairs(v) do
            local name = db[id].name
            local msg = string.format("+ %s", name)
            summary:AddPopUp(msg, color)
        end
    end
end

function XPSummaryState:ApplyXPToParty(xp)

    -- code omitted

    while(actor:ReadyToLevelUp()) do
        local levelup = actor>CreateLevelUp()
        local levelNumber = actor.mLevel + levelup.level
        summary:AddPopUp("Level Up!")

        self:UnlockPopUps(summary, levelup)

        actor:ApplyLevel(levelup.actions)
    end

    -- code omitted
```

Listing 3.369: Adding a notification for when a party member gains a new skill or spell.

In XPSummaryState.lua.

The ApplyXPToParty function calls UnlockPopUps to show a message for each new ability. UnlockPopUps takes in the actor summary and the list of unlocked actions. For each unlocked action we get the relevant action database, either SpellDB or SpecialDB, and look up the action def.

The pop-up text uses a different color for each database. A golden color is used for special abilities and a purple color for magic. We add a + sign to the start of the ability name. For instance, if we unlock the fire spell, then the pop-up shows the text "+ Fire".

The Arena Game Loop

The arena has five rounds, and a round cannot be played until the previous one is unlocked. Round 1 is always unlocked. We need new code to handle round progression.

Near the spawn position are the two big doors to the arena. We'll begin by adding triggers to the bottom of the doors. Activating a trigger creates an ArenaState and pushes it onto the stack.

The ArenaState is a full screen menu that shows the player the five rounds they must win. Initially, only the first round is selectable. When the user selects the first round, a CombatState is created and pushed onto the stack. After combat, the player is returned to the ArenaState menu.

If the player wins a round, the next round is unlocked. If the player loses, they can have another go. At any time the player can quit the arena state and wander the arena grounds. Once the player wins the final round, we'll show them a new message saying "Well done! Champion of the Arena."

Adding the Trigger

The arena has a big door, three tiles across. Let's add triggers along the bottom of the door. Look in the tile editor and you'll see that the bottom of the door takes up the positions 14, 4, 15, 4, and 16, 4. The trigger itself creates a new ArenaState object and pushes it onto the global stack. Copy the code from Listing 3.370.

```
function CreateArenaMap()

    local loot =
    {
        -- code omitted
    }

    local RecruitNPC =
        function(map, trigger, entity, x, y, layer)
```

```

    -- code omitted

end

local EnterArena =
function(map, trigger, entity, x, y, layer)
    gStack:Push(ArenaState>Create(gWorld, gStack))
end

```

Listing 3.370: Adding the EnterArena function. In map_arena.lua.

In Listing 3.370 we've add a new EnterArena function that pushes an ArenaState object onto the global stack.

Next let's add an action definition, create the triggers, and place them on the door. Copy the code from Listing 3.371.

```

return
{
    version = "1.1",

    -- code omitted

    actions =
    {
        -- code omitted
        enter_arena =
        {
            id = "RunScript",
            params = { EnterArena }
        }
    },
    trigger_types =
    {
        -- code omitted
        arena = { OnUse = "enter_arena" },
    },
    triggers =
    {
        -- code omitted
        { trigger = "arena", x = 14, y = 4 },
        { trigger = "arena", x = 15, y = 4 },
        { trigger = "arena", x = 16, y = 4 },
    },
}

```

Listing 3.371: Editing the CreateArenaMap function. In map_arena.lua.

In order to set up the trigger we define an action, enter_arena, that uses RunScript to call the EnterArena function.

We use the OnUse trigger type for the door and hook it up to the enter_arena action. This means that when the player faces the door and presses Spacebar, the trigger activates. In the triggers table we add triggers along the bottom of the door.

Try to run the code. You can approach the door, but using it causes a crash. To fix this, we need to define the ArenaState class.

An ArenaState

The arena state controls what the player sees when entering the arena. We draw the title “Welcome to the Arena” above the selection menu that lists the rounds. The player can use the selection menu to start a round or press Backspace to return to the explore state.

Create the ArenaState.lua, add it to the manifest and dependencies files, and copy its constructor from Listing 3.372.

```
ArenaState = {}
ArenaState.__index = ArenaState
function ArenaState:Create(world, stack)
    local this =
    {
        mWorld = world,
        mStack = stack,
        mRounds =
        {
            {
                mName = "Round 1",
                mLocked = false,
            },
            {
                mName = "Round 2",
                mLocked = true,
            },
            {
                mName = "Round 3",
                mLocked = true,
            },
            {
                mName = "Round 4",
                mLocked = true
            }
        }
    }
    return setmetatable(this, {__index = ArenaState})
end
```

```

        },
        {
            mName = "Round 5",
            mLocked = true
        },
    },
}

local layout = Layout>Create()
layout:Contract('screen', 118, 40)
layout:SplitHorz('screen', 'top', 'bottom', 0.15, 0)
layout:SplitVert('bottom', '', 'bottom', 2/3, 0)
layout:SplitVert('bottom', 'bottom', '', 0.5, 0)
layout:Contract('bottom', -20, 40)
layout:SplitHorz('bottom', 'header', 'bottom', 0.18, 2)
this.mPanels =
{
    layout>CreatePanel('top'),
    layout>CreatePanel('bottom'),
    layout>CreatePanel('header'),
}
this.mLayout = layout

this.mSelection = Selection>Create
{
    data = this.mRounds,
    spacingY = 25,
    rows = #this.mRounds,
    RenderItem =
        function(self, renderer, x, y, item)
            this:RenderRoundItem(renderer, x, y, item)
        end,
    OnSelection =
        function(index, item)
            this:OnRoundSelected(index, item)
        end,
}

this.mSelection.mY = 18
-- Center align horizontally w/o cursor width
gRenderer:ScaleText(1.25, 1.25)
local txtSize = gRenderer:MeasureText(": Locked")
local xPos = -this.mSelection:GetWidth() / 2

```

```

xPos = xPos + this.mSelection.mCursorWidth / 2
xPos = xPos - txtSize:X() / 2
this.mSelection.mX = xPos

setmetatable(this, self)
return this
end

```

Listing 3.372: Creating an basic arena state constructor. In ArenaState.lua.

The ArenaState takes in two parameters, the world and the global stack, which we add to its this table. Information about each round is stored in the this table under mRounds. Each round has a name and a flag to indicate if it is locked or not. All but the first round begins locked. Later we'll add enemy information for each round.

After the this table is constructed, we create the background panels. There's a single panel across the top of the screen for the title. In the center of the screen we add a tall panel, with a header, that displays the rounds. The panels are stored in mPanels and the layout is stored in mLayout. We display the rounds using a selection menu.

The selection menu is created using the mRounds list as the datasource. The menu has one column by default, with 25 pixels of space between each element. The number of rows is set to the length of the mRounds table. The RenderItem and OnSelection callbacks are handled by the functions RenderRoundItem and OnRoundSelected functions, which we'll add shortly.

We center align the selection menu in the bottom box. Each entry in the menu displays the mName of the round plus a string, which is either : Locked or : Open. Using this information, we work out the width to center the box.

Let's implement the rest of the ArenaState. Copy the code from Listing 3.373.

```

function ArenaState:RenderRoundItem(renderer, x, y, item)

    local lockLabel = "Open"
    if item.mLocked then
        lockLabel = "Locked"
    end
    local label = string.format('%s: %s', item.mName, lockLabel)
    renderer:DrawText2d(x, y, label)
end

function ArenaState:Render(renderer)

    renderer:AlignText("center", "center")
    renderer:ScaleText(1.5, 1.5)

```

```

renderer:DrawRect2d(-System.ScreenWidth() / 2,
                     -System.ScreenHeight() / 2,
                     System.ScreenWidth() / 2,
                     System.ScreenHeight() / 2,
                     Vector.Create(0, 0, 0, 1))
for k, v in ipairs(self.mPanels) do
    v:Render(renderer)
end

local titleX = self.mLayout:MidX("top")
local titleY = self.mLayout:MidY("top")
renderer:DrawText2d(titleX, titleY, "Welcome to the Arena")

renderer:ScaleText(1.25, 1.25)

local headerX = self.mLayout:MidX("header")
local headerY = self.mLayout:MidY("header")
renderer:DrawText2d(headerX, headerY, "Choose Round")

renderer:AlignText("left", "center")
self.mSelection:Render(renderer)
end

function ArenaState:HandleInput()
    if Keyboard.JustReleased(KEY_BACKSPACE) or
        Keyboard.JustReleased(KEY_ESCAPE) then
        self.mStack:Pop()
    end

    self.mSelection:HandleInput()
end

function ArenaState:OnRoundSelected(index, item) end
function ArenaState:Enter() end
function ArenaState:Exit() end
function ArenaState:Update(dt) end

```

Listing 3.373: The functions for the ArenaState. In ArenaState.lua.

RenderRoundItem is called by the selection menu to render each entry. The round's mName is rendered, followed by a colon and "Locked" or "Open", depending on its current state.

The HandleInput function checks if Backspace or Escape is pressed, and if so it pops the ArenaState off the stack. This returns the player to the ExploreState. We call the

HandleInput function for the mSelection menu to let the player browse the menu and choose a round.

In the Render function, we draw a black rectangle covering the entire screen. Then we render all the panels. We use the mLayout field to get the center point of the title panel and draw the title text "Welcome to the Arena". We scale the text down a little and draw "Choose Round" in the header panel. Finally we draw the mSelection menu to display all the rounds.

We'll implement the menu's OnRoundSelected callback function shortly. The Update, Enter and Exit are not used at this time and so are empty.

Run the code. Use the doors and you'll see something like Figure 3.75.



Figure 3.75: The ArenaState screen.

The full code so far is available in arena-solution-1.

Arena Game Loop

The player can see the arena rounds. The next task is to let them enter combat. Example arena-2 contains the code so far, or you can continue with your own codebase.

When the player selects a round, we check if it's unlocked, and if it is we create a new combat state and push it onto the stack. When combat finishes, we check the result

and decide if the next round unlocks. If the player dies, we skip the game over and return to the arena screen instead.

Let's start by filling out the `OnRoundSelected` function. Copy the code from Listing 3.374.

```
function ArenaState:OnRoundSelected(index, item)

    if item.mLocked then
        return -- can't play locked rounds
    end

    local enemyDefs = item.mEnemy or { gEnemyDefs.goblin }
    local enemyList = {}
    for k, v in ipairs(enemyDefs) do
        enemyList[k] = Actor:Create(v)
    end

    local combatDef =
    {
        background = "arena_background.png",
        actors =
        {
            party = self.mWorld.mParty:ToArray(),
            enemy = enemyList,
        },
        canFlee = false,
        OnWin =
        function()
            self:WinRound(index, item)
        end,
        OnDie =
        function()
            self:LoseRound(index, item)
        end,
    }
    local state = CombatState>Create(self.mStack, combatDef)
    self.mStack:Push(state)
end
```

Listing 3.374: Triggering combat in the `OnRoundSelected` function. In `ArenaState.lua`.

The `OnRoundSelected` function is a callback for the selection menu. It's called when the player chooses a round. The first if-statement checks if the round is locked, and if it is we return immediately. The player cannot choose locked rounds. If the round is unlocked, then we set up a combat state and push it onto the stack.

To create the combat state, we need to know which enemies to add. Each round has different enemies. Enemies are defined for each round as a table of ids called `mEnemy`. None of our rounds have enemy information at the moment, so to keep things ticking along we've made the `mEnemy` table default to one goblin. When setting up the round, we create actors from the ids in the `mEnemy` list. We store the actors in the `enemyList` table. Then we pass the `enemyList` table into the combat state by adding it to the combat definition.

The combat definition is stored locally as `combatDef`. We've seen this table before, but this one has a few new fields. The new fields are `canFlee`, `OnWin` and `OnDie`. The `canFlee` flag prevents flee from working if set to false. This is useful for boss fights, arenas and scripted battles. The `OnWin` function is called when the player wins, and the `OnDie` function is called when the party is defeated. The `OnDie` function is called instead of going to the `GameOverState`. We need to modify the `CombatState` to add code to support these new callbacks. The `OnWin` and `OnDie` callbacks call new functions, `WinRound` and `LoseRound`, which we'll implement shortly.

In the rest of the `combatDef` we continue to use the arena background. The actors involved in combat are taken from the player party, and for the enemies we use the list we just created.

The final part of the function uses the `combatDef` to create a combat state and pushes it onto the stack. Run the code and you'll be able to enter the first round. If you win at this point the next round remains locked, and if you lose you'll be taken to the game over state. Let's fix this by making some changes to the `CombatState` constructor. Copy the code from Listing 3.375

```
function CombatState:Create(stack, def)

    local this =
    {
        -- code omitted
        mCanFlee = true,
        OnDieCallback = def.OnDie, -- can be nil
        OnWinCallback = def.OnWin, -- can be nil
    }

    if def.canFlee ~= nil then
        this.mCanFlee = def.canFlee
    end

```

Listing 3.375: Extending the `CombatState` with some new fields. In `CombatState.lua`.

In the constructor, the `mCanFlee` flag is set to true by default. If there's a value for it in the `def`, then we use that value. The callback functions are added directly to the `this` table and it's ok if they're nil.

Let's write the code to make fleeing fail if `mCanFlee` is false. We need to make these changes in `CEFlee.lua`. Copy the code from Listing 3.376.

```
function CEFlee:Create(state, actor)
    local this =
    {
        -- code omitted
    }

    -- Decide if flee succeeds
    this.mCanFlee = Formula.CanFlee(state, actor)

    if not this.mState.mCanFlee then
        this.mCanFlee = false
    end
```

Listing 3.376: Stop fleeing succeeding if it's turned off in the combatstate. In `CEFlee.lua`.

In `CEFlee` we force flee to fail if the combat state's `mCanFlee` is false.

Next let's hook in the callback for dying during combat. When all the party members get knocked out, we call the `OnLose` function. Copy the code from Listing 3.377.

```
function CombatState:OnLose()

    -- code omitted

    local storyboard

    if self.OnDieCallback then
        storyboard =
        {
            SOP.UpdateState(self, 1.5),
            SOP.BlackScreen("black", 0),
            SOP.FadeInScreen("black"),
            SOP.RemoveState(self),
            SOP.Function(self.OnDieCallback),
            SOP.Wait(2),
            SOP.FadeOutScreen("black"),
        }
    else
        storyboard =
        {
            SOP.UpdateState(self, 1.5),
```

```

        SOP.BlackScreen("black", 0),
        SOP.FadeInScreen("black"),
        SOP.ReplaceState(self, GameOverState:Create(self.mGameStack,
            gWorld)),
        SOP.Wait(2),
        SOP.FadeOutScreen("black"),
    }
end

local storyboard = Storyboard:Create(self.mGameStack, storyboard)
self.mGameStack:Push(storyboard)
end

```

Listing 3.377: Adding support for a death callback in the combat state. In CombatState.lua.

The normal behavior for OnLose is to load a storyboard that fades to the game over state. In Listing 3.377 we check for the presence of an OnDieCallback function. If the callback exists, then we fade to black, remove the CombatState, and call the OnDieCallback. We use a new storyboard event, RemoveState, that goes through the stack and deletes the passed-in state. Copy the code from Listing 3.378.

```

function SOP.RemoveState(state)
    return function(storyboard)

        local stack = storyboard.mStack

        for i = #stack.mStates, 1, -1 do
            local v = stack.mStates[i]
            if v == state then
                v:Exit()
                table.remove(stack.mStates, i)
                break
            end
        end
        return EmptyEvent
    end
end

```

Listing 3.378: RemoveState Operation. In StoryboardEvents.lua.

Figure 3.76 shows the how the stack changes as the player chooses a round and loses combat.

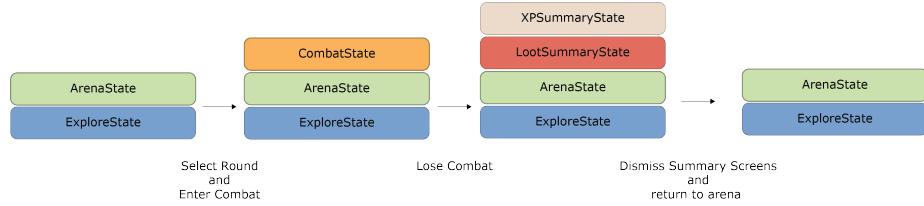


Figure 3.76: The state flow for failing a round of the arena.

Let's finish the CombatState modifications by adding support for OnWinCallback. When the party wins the battle we check for the OnWinCallback, and if it exists create and run an alternate storyboard that calls the OnWinCallback.

```

function CombatState:OnWin()

    -- code omitted

    local storyboard =
    {
        -- code omitted
    }

    if self.OnWinCallback then

        storyboard =
        {
            SOP.UpdateState(self, 1.0),
            SOP.BlackScreen("black", 0),
            SOP.FadeInScreen("black", 0.6),
            SOP.ReplaceState(self, xpSummaryState),
            SOP.Function(function()
                self.OnWinCallback()
            end)
        }

    end

    local storyboard = Storyboard>Create(self.mGameStack, storyboard)
    self.mGameStack:Push(storyboard)
end

```

Listing 3.379: Adding the OnWinCallback into the OnWin function. In CombatState.lua.

If you run the code as it stands, it crashes once combat ends. This is because we haven't implemented WinRound and LoseRound in the ArenaState. The LoseRound function increases all the party member's HP from 0 to 1 so they're not walking around unconscious. It doesn't do anything else, so after losing combat the player continues to the xp and loot screens before returning to the arena state. Copy the code from Listing 3.380.

```
function ArenaState:LoseRound(index, item)

    local party = self.mWorld.mParty.mMembers
    for k, v in pairs(party) do
        local hp = v.mStats:Get("hp_now")
        hp = math.max(hp, 1)
        v.mStats:Set("hp_now", hp)
    end

end
```

Listing 3.380: Recovering the defeated party members in LoseRound. In ArenaState.lua.

LoseRound loops through the party members, and if their HP is less than 1, it sets it to 1.

The WinRound code unlocks the next round, if there is a next round. Copy the code from Listing 3.381.

```
function ArenaState:WinRound(index, item)

    -- Check for win
    if index == #self.mRounds then
        self.mStack:Pop()
        local state = ArenaCompleteState>Create()
        self.mStack:Push(state)
    end

    -- Move the cursor to the next round if there is one
    self.mSelection:MoveDown()

    -- Unlock the newly selected round
    local nextRound = self.mSelection:SelectedItem()
    nextRound.mLocked = false
end
```

Listing 3.381: Implementing the WinRound function to unlock new arena rounds. In ArenaState.lua.

In WinRound we first check if the player has won the final round. If so, we push on a new state, ArenaCompleteState, that displays a game over message.

On winning a round, we move the menu cursor down to the next round and unlock it. This way the player can advance through all the rounds as they play. Check out the state flow for winning in Figure 3.77.

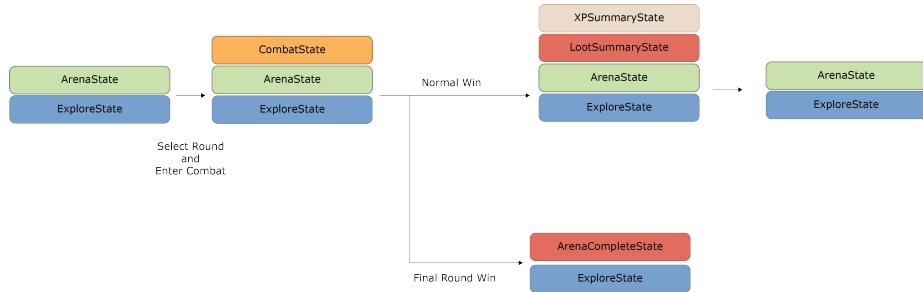


Figure 3.77: The state flow for winning a round in the arena.

That's the main structure of the arena finished. Before moving on to add more interesting enemies, let's implement the ArenaCompleteState.

Create ArenaCompleteState.lua in the code directory and add it to the manifest and dependencies file. Copy the code from Listing 3.382.

```

ArenaCompleteState = {}
ArenaCompleteState.__index = ArenaCompleteState
function ArenaCompleteState:Create()
    local this = {}

    setmetatable(this, self)
    return this
end

function ArenaCompleteState:Enter() end
function ArenaCompleteState:Exit() end
function ArenaCompleteState:Update(dt) end
function ArenaCompleteState:HandleInput() end

function ArenaCompleteState:Render(renderer)
    renderer:DrawRect2d(System.ScreenTopLeft(),
                        System.ScreenBottomRight(),
                        Vector.Create(0,0,0,1))
end
  
```

```
CaptionStyles["title"]:Render(renderer,  
    "You win!")  
CaptionStyles["subtitle"]:Render(renderer,  
    "Champion of the Arena.")  
end
```

Listing 3.382: A simple congratulations message on finishing the arena. In ArenaCompleteState.lua.

The ArenaCompleteState is bare-bones and uses CaptionStyles to display a big message saying “You win!” and “Champion of the Arena.” You can see the screen displayed in Figure 3.78.

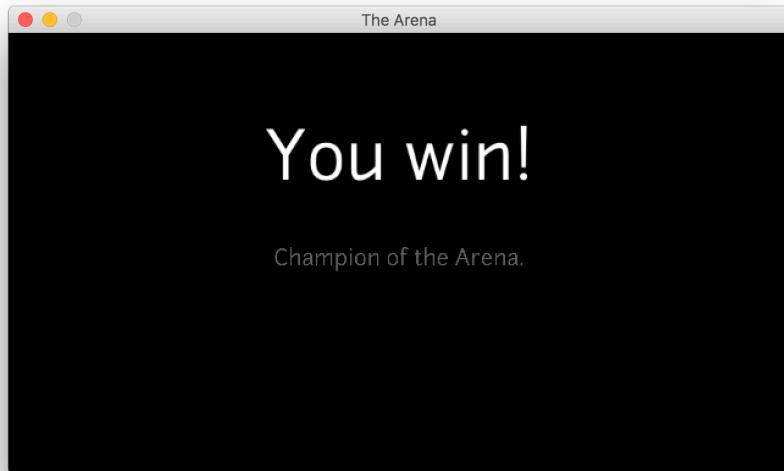


Figure 3.78: The “You Win” message for the arena.

Making the Arena Dangerous

The arena is made up of five rounds. Currently all rounds default to the same enemy roster, one goblin. Let’s introduce a few new enemies and mix up the enemies in each round.

The arena-2 project contains “ogre.png” and “green_dragon.png” in its art folder. Add these to the dependencies file. These are the new enemies for the arena.

To use new sprites, we need new entity definitions. Copy them from Listing 3.383.

```

gEntities =
{
    goblin =
    {
        -- code omitted
    },
    dragon =
    {
        texture = "green_dragon.png",
        startFrame = 1,
        width = 128,
        height = 64,
    },
    ogre =
    {
        texture = "ogre.png",
        startFrame = 1,
        width = 64,
        height = 64,
    },
}

```

Listing 3.383: New entity definitions. In EntityDefs.lua.

A little further down in the EntityDef.lua file, we'll add the character definitions. Copy the code from Listing 3.384.

```

gCharacters =
{
    -- code omitted
    goblin =
    {
        -- code omitted
    },
    dragon =
    {
        entity = "dragon",
        controller =
        {
            "cs_move",
            "cs_run_anim",
            "cs_standby",
            "cs_die_enemy",
            "cs_hurt_enemy"
        },
        state = "cs_standby",
    }
}

```

```

},
ogre =
{
    entity = "ogre",
    controller =
    {
        "cs_move",
        "cs_run_anim",
        "cs_standby",
        "cs_die_enemy",
        "cs_hurt_enemy"
    },
    state = "cs_standby",
}
}

```

Listing 3.384: Adding the character definitions for the enemies. In EntityDefs.lua.

Next let's add the enemy definitions. Copy the code from Listing 3.385.

```

gEnemyDefs =
{
    goblin =
    {
        -- code omitted
    },
    ogre =
    {
        id = "ogre",
        stats =
        {
            ["hp_now"] = 150,
            ["hp_max"] = 150,
            ["mp_now"] = 0,
            ["mp_max"] = 0,
            ["strength"] = 25,
            ["speed"] = 2,
            ["intelligence"] = 1,
            ["counter"] = 0,
        },
        name = "Ogre",
        actions = { "attack" },
        steal_item = 12,
        drop =
        {

```

```

xp = 200,
gold = {40, 50},
always = nil,
chance =
{
    { oddment = 1, item = { id = -1 } },
    { oddment = 3, item = { id = 10 } }
}
},
dragon =
{
    id = "dragon",
    stats =
    {
        ["hp_now"] = 200,
        ["hp_max"] = 200,
        ["mp_now"] = 0,
        ["mp_max"] = 0,
        ["strength"] = 35,
        ["speed"] = 6,
        ["intelligence"] = 20,
        ["counter"] = 0.1,
    },
    name = "Green Dragon",
    actions = { "attack" },
    steal_item = 11,
    drop =
    {
        {
            xp = 350,
            gold = {250, 300},
            always = nil,
            chance =
            {
                { oddment = 1, item = { id = -1 } },
                { oddment = 3, item = { id = 10 } }
            }
        }
    }
}

```

Listing 3.385: New arena definitions. In `EnemyDefs.lua`.

Now that the new enemies have been added, we're ready to use them in the arena!

Let's update the round data in the ArenaState constructor. We want the arena to get progressively harder, so we increase the enemy difficulty each round. Copy the code from Listing 3.386.

```
function ArenaState:Create(world, stack)
    local this =
    {
        mWorld = world,
        mStack = stack,
        mRounds =
        {
            {
                mName = "Round 1",
                mLocked = false,
                mEnemy =
                {
                    gEnemyDefs.goblin,
                }
            },
            {
                mName = "Round 2",
                mLocked = true,
                mEnemy =
                {
                    gEnemyDefs.goblin,
                    gEnemyDefs.goblin,
                    gEnemyDefs.goblin,
                }
            },
            {
                mName = "Round 3",
                mLocked = true,
                mEnemy =
                {
                    gEnemyDefs.goblin,
                    gEnemyDefs.ogre,
                }
            },
            {
                mName = "Round 4",
                mLocked = true,
                mEnemy =
                {
                    gEnemyDefs.ogre,
                    gEnemyDefs.ogre,
                }
            }
        }
    }
```

```

},
{
    mName = "Round 5",
    mLocked = true,
    mEnemy =
    {
        gEnemyDefs.dragon,
    }
},
}
}

```

Listing 3.386: Filling in the final round data. In ArenaState.lua.

Run the game and you'll be able to recruit a party, equip weapons, and defeat monsters. Then you'll be able to level up and unlock new skills. This is a full working combat system with a mini game. Well done! You can see the final round of the arena in Figure 3.79.



Figure 3.79: A single hero facing a dragon in the arena.

The full code is available in arena-2-solution. In the next section we're going to make a full small game with a quest and a story!

Chapter 4

Quests

An RPG needs to give the player a quest: save the world, discover your hidden past, stop the bad guy, etc. In this section we'll create a small game with a simple central quest. To create a fully-featured small game we'll use everything we've learned so far and more! This means new maps, enemies, items and cutscenes. We'll also create a quest tracking and save / load system.

Let's get to it!

Planning

We're going to create a minimal RPG demonstrating everything we've learned so far. The game will be short and stick to the normal RPG conventions. Once it's created, you'll be able to expand the game into a fully-featured RPG or confidently create your own.

Plot and Flow

Let's start by considering the game at a high level.

The game is set in a standard medieval-type fantasy world.

We begin in a small town with the party standing inside the major's house. An intro cutscene plays where the major tells you to retrieve a gemstone from the caves to the north. Once the intro finishes, you're free to leave and explore the town. There are a few shops where you can stock up on supplies and a few NPCs to talk to. At any time you can leave by taking the path out of town. Leaving the town takes you to the world map.

The world map is dangerous, and while walking across it you may encounter enemies. The town can be reentered from world map by walking onto its tile. Apart from the town, the only other notable landmark is a dark cave at the top of the map.

The cave contains bats and other nasty creatures but also treasure to discover. Halfway through the cave there's a simple puzzle. At end of the cave you'll find the gemstone and encounter a boss enemy. Once the boss is defeated you can take the gem back to town. After giving the gem to the major, there's a plot twist; the major will turn into a monster! Once you defeat the major, you win and it's game over.

By analysing this basic script we can create a simple game flow as shown in Figure 4.1.

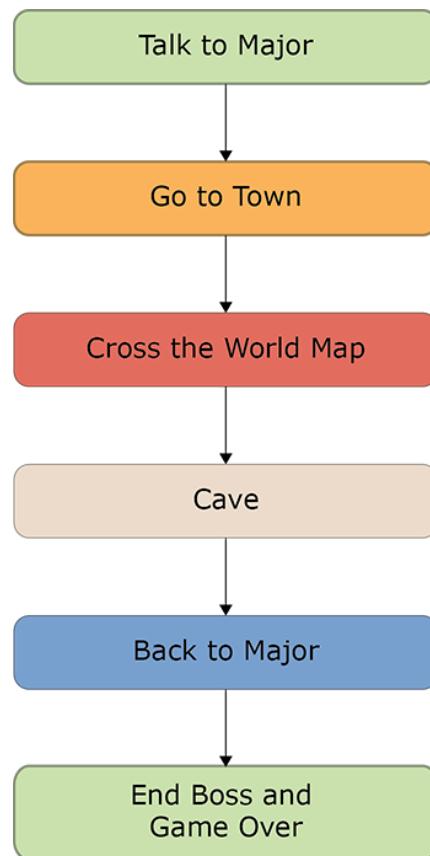


Figure 4.1: The high-level flow of the game.

This high-level plan lets us see what content is needed to make the game. Game content includes things like art assets, cutscenes, maps, monsters, items, etc.

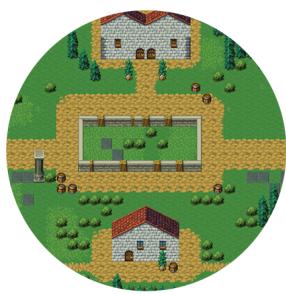
Asset List

To make a game, we examine the requirements, create a task list, do the tasks, and update the list as new information is discovered. Games are exploratory programming; it's hard to know all the requirements in advance, therefore it's best to dive in and start swimming, revising your heading as you go.

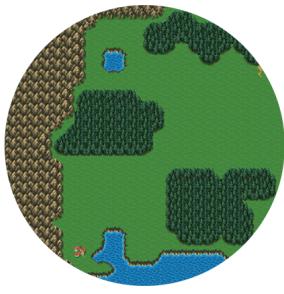
By looking at the high-level overview we can see the game falls nicely into three pieces:

- Town
- Overmap
- Cave

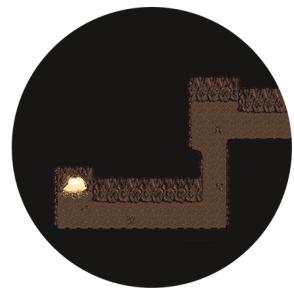
Each of the pieces is represented by one map. A snapshot of each map can be seen in Figure 4.2.



Town



World Map



Cave

Figure 4.2: The three game maps.

To make things easier to follow, the tilesets and map layouts will be provided. We'll concentrate on adding the interactive elements and creating the game. Let's take each map in isolation and make a list of what needs doing for that map. We'll begin with a first pass of all the maps, then we'll come back and add the quest information on top.

High-Level Task List

1. Create the basic town, overworld and cave maps.
2. Add the quest information into each map.
3. Add save / load.
4. Add and balance random spawns.
5. Add front menus and credits.
6. Polish.

The maps and tilesets are provided in the example projects but for now let's just get a high-level feel of what needs doing, starting with the town map.

The Town

Let's make an initial task list for the town.

1. Create the base town map.
2. Add a shop.
3. Add an inn.
4. Add NPCs.
 1. The major
 2. Guards
 3. 3 x Wanderers
5. Add a trigger to the world map.

Our town is the classic RPG town; green grass with pathways, one or two important buildings, and some NPCs milling around. The central building is the town hall. The map shouldn't take too long to navigate so we'll make the town roughly 60 by 60 tiles. The movement code takes 0.3 second to walk a character across a tile, so even if we trek across the full town it only takes 18 seconds, which isn't too bad.

We'll make the map bigger than the town. It will be 60 x 120 tiles; much taller than it is wide. At the top of the map we have a secret area the player cannot see from the main map. This secret area contains all the building interiors. If you enter the shop door, you'll teleport to its interior at the top of the map. When you exit you'll teleport back to the doorway. You can see the room the major's building is linked to in Figure 4.3.



Figure 4.3: The full town map and the link between the major's house interior and exterior.

Example 1-quests contains the town map and tilesets. You are free to create your own map but this is the one we'll refer to in the book. At this point there are no other project files.

We'll deal with the content of the town in the next chapter.

World Map

RPGs often use two different scales for maps. Maps like the town are human-scale; bushes and treasure chests are of a reasonable size when compared to the characters. The world map uses a zoomed out, representation scale. We use the world map for long-distance travel. Instead of a tile representing a single bush it represents a small forest or mountain range.

Here are our overworld map tasks.

1. Create the map.
2. Add the town trigger.
3. Add the cave trigger.
4. Add a random encounter system.
5. Set up the encounter zones.

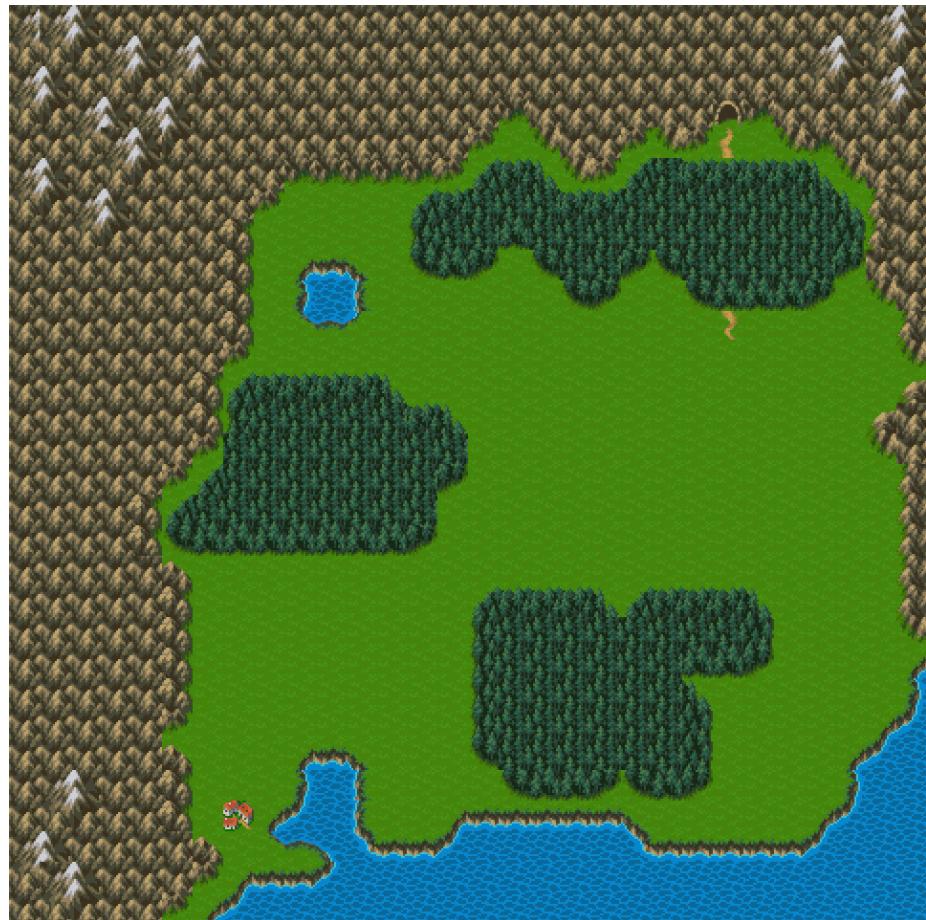


Figure 4.4: The full world map.

You can see the world map in Figure 4.4. Our world map shows the town we started in as a single tile, near the coast at the bottom of the map. This map uses a random encounter system to make travel a little dangerous. We'll want different monsters in different zones, i.e. the forest and plains areas, and some way to define the zones.

On the way to the cave we want the player to have a few random encounters. The number of encounters is determined by the distance to travel on the map and the encounter rate. We'll make the map 30 x 30 tiles and later we'll set an encounter rate so the player has around 3 battles before getting to the cave.

A world map is a great place to add secrets and interesting areas but we're keeping this game very linear; the only two places are the starting town and the cave. These will both have transition triggers.

Cave

1. Create the map.
2. Add the enter / exit triggers.
3. Add a cave puzzle.
4. Add a boss trigger and cutscene.
5. Add a boss defeated cutscene.

The cave is a series of rooms and tunnels with some random encounters, a few scattered chests, and one puzzle room. The puzzle is very simple. The player must pass through a room with two statues flanking an open door. Then a little later they'll pass through a room with one statue and a space for a second. The exit for this room is closed. The player needs to go down a side route to find the missing statue. Bringing the statue back and placing it on the space opens the exit. The room has some triggers to give hints to the player about what they need to do. This is a first dungeon so the puzzle is simple.

The cave ends with a boss fight and a cutscene. Then the player is given the gemstone.

Return to the Town

After the caves are finished, the party returns to the town to visit the major. The major needs some logic to recognize the party has succeeded in their quest and react appropriately. We'll need a few cutscenes and then end on one final fight.

Wiring Everything Up

Once we've created the maps with the content we want, we'll wire up the rest of game. This includes things like a start screen and menu, saving and loading, and code to let you continue or restart the game if you die.

On to the first map; the town!

The Town

In this chapter we fill out the town by adding an introduction cutscene, NPCs, shops, an inn, and the triggers to enter and leave.

The Setup

To create our game we'll use example quests-4. This project is based on the arena, but the arena assets have been replaced by the town map and its tileset.

The updated manifest for quests-4 is shown in Listing 4.1.

```
manifest =
{
    scripts =
    {
        -- code omitted

        [ 'map_town.lua' ] =
        {
            path = "map/map_town.lua"
        },
    }
    textures =
    {
        -- code omitted
    }
}
```

Listing 4.1: Adding the map to the manifest. In manifest.lua.

The map_town.lua has been added to a map directory in the root of the project. The maps are set up in the usual way; a Tiled Lua export, wrapped up in a function call.

The map files are also in the dependencies.lua file as shown in Listing 4.2.

```

Apply({
    -- code omitted
    "map_town.lua"
},
function(v) Asset.Run(v) end)

```

Listing 4.2: Adding the map to the dependencies. In dependencies.lua.

The map database has had any references to the arena removed and now references the town map, as shown in Listing 4.3.

```

MapDB =
{
    ["town"] = CreateTownMap,
}

```

Listing 4.3: Hooking up the map creation functions. In MapDB.lua.

The town map is now loaded when the game runs, as you can see in Listing 4.4.

```

LoadLibrary('Asset')
Asset.Run('Dependencies.lua')

gRenderer = Renderer.Create()
gStack = StateStack>Create()
gWorld = World>Create()

gWorld.mParty:Add(Actor>Create(gPartyMemberDefs.hero))
gStack:Push(ExploreState>Create(gStack, CreateTownMap(),
    Vector.Create(5, 9, 1)))

function update()
    local dt = GetDeltaTime()
    gStack:Update(dt)
    gStack:Render(gRenderer)
    gWorld:Update(dt)
end

```

Listing 4.4: Loading the town map. In main.lua.

The start point is set to X:5, Y:9 on layer 1. This means the player starts inside the major's house in front of his desk as shown in Figure 4.5.

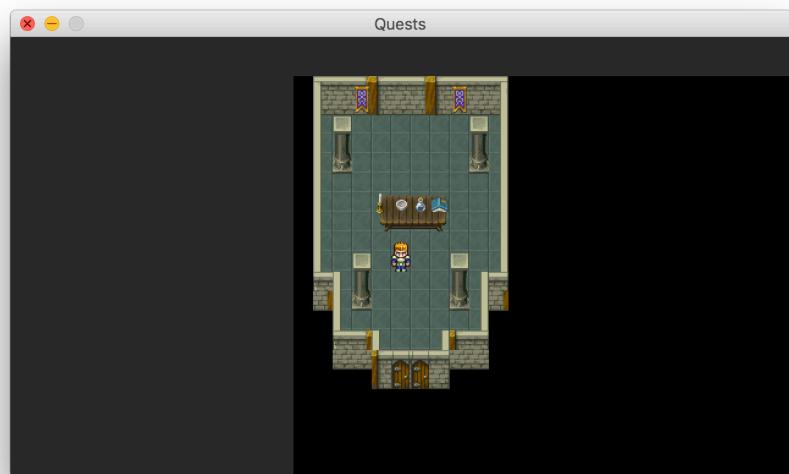


Figure 4.5: The starting point for the quest game.

Teleport Triggers

The player starts in a room with no way out! We need to add a trigger to the doorway that teleports him to the main town area. The town has plenty of other buildings, and they need triggers too.

Let's add building triggers next. Each building has its interior stored at the top of the map. When the player moves through a doorway, we teleport the player to the interior of the building, in a single frame. This works quite nicely. You can see the doorway links in Figure 4.6.

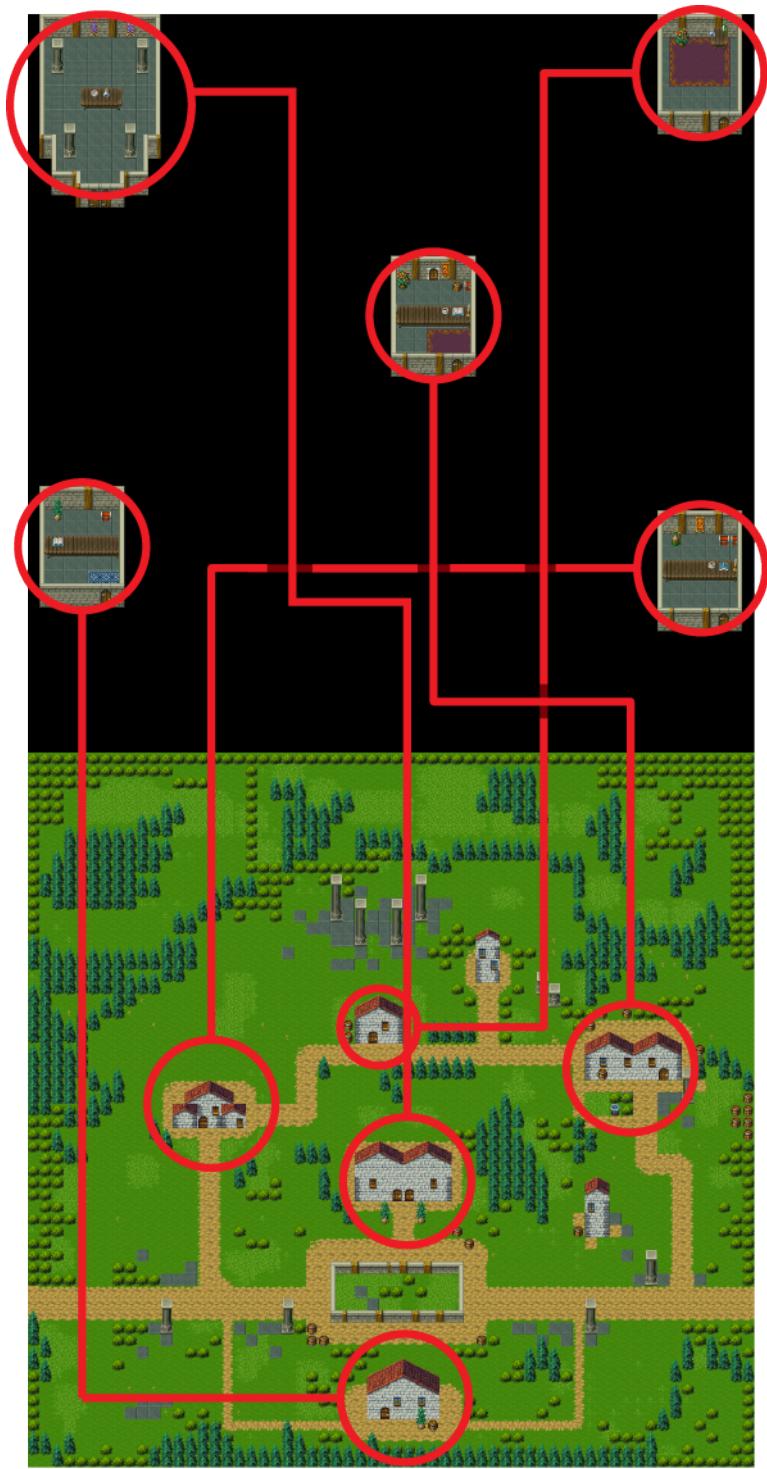


Figure 4.6: The starting point for the quest game.

To add the teleporters we update the map's triggers, trigger_types and actions tables. There are five buildings with doors and for each door we add a building_to_map trigger and a corresponding map_to_building trigger. These triggers use our existing teleport action. Copy the code from Listing 4.5.

```
function CreateTownMap()
{
    -- code omitted
    on_wake = {},
    actions =
    {
        major_to_map = { id = "Teleport", params = {30, 98} },
        potions_to_map = { id = "Teleport", params = {29, 116} },
        inn_to_map = { id = "Teleport", params = {52, 88} },
        arms_to_map = { id = "Teleport", params = {14, 92} },
        empty_to_map = { id = "Teleport", params = {28, 85} },

        map_to_major = { id = "Teleport", params = {5, 14} },
        map_to_potions = { id = "Teleport", params = {6, 47} },
        map_to_inn = { id = "Teleport", params = {35, 28} },
        map_to_arms = { id = "Teleport", params = {57, 49} },
        map_to_empty = { id = "Teleport", params = {57, 8} },
    },
    trigger_types =
    {
        in_major = { OnEnter = "map_to_major" },
        out_major = { OnEnter = "major_to_map" },
        in_potions = { OnEnter = "map_to_potions" },
        out_potions = { OnEnter = "potions_to_map" },
        in_arms = { OnEnter = "map_to_arms" },
        out_arms = { OnEnter = "arms_to_map" },
        in_inn = { OnEnter = "map_to_inn" },
        out_inn = { OnEnter = "inn_to_map" },
        in_empty = { OnEnter = "map_to_empty" },
        out_empty = { OnEnter = "empty_to_map" },
    },
    triggers =
    {
        { trigger = "in_major", x = 30, y = 97 },
        { trigger = "in_major", x = 31, y = 97 },
        { trigger = "out_major", x = 5, y = 15 },
    }
}
```

```

{ trigger = "out_major", x = 6, y = 15 },
{ trigger = "in_potions", x = 29, y = 115 },
{ trigger = "in_arms", x = 14, y = 91 },
{ trigger = "in_empty", x = 28, y = 84 },
{ trigger = "in_inn", x = 52, y = 87 },
{ trigger = "out_empty", x = 57, y = 9 },
{ trigger = "out_inn", x = 35, y = 29 },
{ trigger = "out_potions", x = 6, y = 48 },
{ trigger = "out_arms", x = 57, y = 50 },
},

```

Listing 4.5: Adding actions and triggers to the town map. In map_town.lua.

In Listing 4.5 we create a list of teleport actions. Each action teleports the player to a specific location on the map. We don't teleport directly on top of a door tile; instead we teleport just in front of it. This gives the impression that the character moves *through* the door. Each action is hooked to a trigger type with the OnEnter callback. These triggers are placed on the map on the interior and exterior doorways of the various buildings.

Basic NPCs

With the triggers in place we can fully explore the town. But it feels empty. Let's give it a little life by adding some NPCs. Before we add characters we must define them. These are the NPCs we'll have on the map:

1. The Major
2. The Inn Keeper
3. The Potion Master
4. The Blacksmith
5. Villager 1
6. Villager 2

Six NPCs doesn't make for a super populous town but it works well for our RPG needs. The artwork for the NPCs is already included in the project in the walk_cycle.png texture. Open up EntityDefs.lua and let's define the NPC entities. Copy the code from Listing 4.6.

```

gEntities =
{
    -- code omitted
    npc_major =
    {

```

```

        texture = "walk_cycle.png",
        width = 16,
        height = 24,
        startFrame = 105,
        tileX = 11,
        tileY = 3,
        layer = 1
    },
    npc_1 =
    {
        texture = "walk_cycle.png",
        width = 16,
        height = 24,
        startFrame = 121,
        tileX = 11,
        tileY = 3,
        layer = 1
    },
    npc_2 =
    {
        texture = "walk_cycle.png",
        width = 16,
        height = 24,
        startFrame = 137,
        tileX = 11,
        tileY = 3,
        layer = 1
    },
}

```

Listing 4.6: Adding entity defs for the major and NPCs. In EntityDefs.lua.

We use the entity defs in Listing 4.6 to create the characters for the town. Let's update the gCharacters table next. (We're also taking this opportunity to add some more states to the thief and mage so we can control them in cutscenes.) Copy the code from Listing 4.7.

```

gCharacters =
{
    thief =
    {
        -- code omitted
        controller =
        {
            -- code omitted

```

```

        "follow_path",
        "move"
    },
    state = "npc_stand",
},
mage =
{
    -- code omitted
controller =
{
    -- code omitted
    "follow_path",
    "move"
},
state = "npc_stand",
},
-- code omitted

npc_major =
{
    entity = "npc_major",
    anims =
    {
        up      = {97,  98,  99,  100},
        right   = {101, 102, 103, 104},
        down    = {105, 106, 107, 108},
        left    = {109, 110, 111, 112},
    },
    facing = "down",
    controller = {"follow_path", "move", "npc_stand"},
    state = "npc_stand"
},
npc_inn_keeper =
{
    entity = "npc_1",
    anims =
    {
        up      = {113, 114, 115, 116},
        right   = {117, 118, 119, 120},
        down    = {121, 122, 123, 124},
        left    = {125, 126, 127, 128},
    },
    facing = "down",
    controller = {"npc_stand"},
```

```

        state = "npc_stand"
    },
    npc_blacksmith =
{
    entity = "npc_1",
    anims =
    {
        up      = {113, 114, 115, 116},
        right   = {117, 118, 119, 120},
        down    = {121, 122, 123, 124},
        left    = {125, 126, 127, 128},
    },
    facing = "down",
    controller = {"npc_stand"},
    state = "npc_stand"
},
    npc_potion_master =
{
    entity = "npc_1",
    anims =
    {
        up      = {113, 114, 115, 116},
        right   = {117, 118, 119, 120},
        down    = {121, 122, 123, 124},
        left    = {125, 126, 127, 128},
    },
    facing = "down",
    controller = {"npc_stand"},
    state = "npc_stand"
},
    npc_villager_1 =
{
    entity = "npc_1",
    anims =
    {
        up      = {113, 114, 115, 116},
        right   = {117, 118, 119, 120},
        down    = {121, 122, 123, 124},
        left    = {125, 126, 127, 128},
    },
    facing = "down",
    controller = {"npc_stand"},
    state = "npc_stand"
},
    npc_villager_2 =

```

```

{
    entity = "npc_2",
    anims =
    {
        up      = {129, 130, 131, 132},
        right   = {133, 134, 135, 136},
        down    = {137, 138, 139, 140},
        left    = {141, 142, 143, 144},
    },
    facing = "down",
    controller = {"npc_stand"},
    state = "npc_stand"
},
}

```

Listing 4.7: Updating the character definitions. In EntityDefs.lua.

Most NPCs on the map don't move, which means we don't need to define walk cycles for them. They only need a single frame. The major NPC, however, does more during the intro cutscene so we've given him a move controller.

To add NPCs to the map, let's modify the map def's OnWake table. Copy the code from Listing 4.8.

```

on_wake =
{
{
    id = "AddNPC",
    params = {{ def = "npc_major", id = "major", x = 5, y = 4 }}
},
{
    id = "AddNPC",
    params = {{ def = "npc_inn_keeper", id = "inn_keeper",
        x = 35, y = 23 }}
},
{
    id = "AddNPC",
    params = {{ def = "npc_blacksmith", id = "blacksmith",
        x = 57, y = 44 }}
},
{
    id = "AddNPC",
    params = {{ def = "npc_potion_master", id = "potion_master",
        x = 6, y = 42 }}
},
}

```

```

{
    id = "AddNPC",
    params = {{ def = "npc_villager_1", id = "villager_1",
        x = 47, y = 101 }}
},
{
    id = "AddNPC",
    params = {{ def = "npc_villager_2", id = "villager_2",
        x = 35, y = 78 }}
},
}
,
```

Listing 4.8: Adding the NPCs to the town using the on_wake function. In map_town.lua.

In Listing 4.8 we add the six new NPCs using the on_wake table. The characters include shopkeepers, the major, and a few villagers. Load up the map and wander around. You'll see the town is a bit more lively now as in Figure 4.7.



Figure 4.7: NPCs have been added to the town map.

The code so far is available in `quests-4-solution`.

At the moment you cannot interact with any of the NPCs. All the NPCs perform different actions. The major is quite complicated because he's important to the quest, the people

on the main map just say one line, and the shopkeepers sell you items. Let's start by writing the code for the shopkeepers.

Shops

Most RPG games have some sort of shop. There are many different ways to design and implement a shop. Our shop will be fully functional but basic. The town has two shops; an arms shop for weapons and armor, and a potion shop. The shop code needs to be generic enough to support multiple types of shop.

Example quests-5 contains the codebase so far, or you can continue on with your own.

When the player talks to a shopkeeper, we push a ShopState onto the global stack. This state takes up the entire screen and displays the items the shop is selling. The player is also able to sell items to the shop. The shop screen displays the party's gold, and any item the player cannot afford is grayed out. At the bottom of the screen we draw the party members. As the player browses through the items, we change the character sprites to indicate if the current selection is better than what they currently have equipped.

Our shop's inventory is unlimited. If you see a sword in the shop, you can buy one or you can buy one thousand; there's no stock limit.

Our shop will also buy the player's items. The shop is extremely fortunate, in that it has unlimited funds to buy any player item. The shop never buys key items (as that would break quests!). The shop only buys what's in the player inventory, not what the player has equipped. You want to sell it? Unequip it first!

Any item the player sells to the shop is destroyed; it's not added the shop's stock. For instance, sell a one-of-a-kind healing potion to the potion shop and you can't buy it back; it's gone.

Here's a sketch of the shop layout Figure 4.8.



Figure 4.8: Shop layout sketch.

As you can see in Figure 4.8, the shop state displays a lot of information. We'll run through it quickly and touch on some of the key decisions that led to this layout.

In the top left is the shop title. We have a number of different shops, so the title tells the player which shop they're in; an item shop, a weapon shop, a spell shop, etc. To the right of the title is the navigation pane containing three choices: "Buy", "Sell" and "Exit".

The player can use the navigation pane to choose if they want to buy, sell or exit the shop. The next pane shows a selection box of items. By default these are the items the player can buy, but if the player chooses the "Sell" option then the party inventory is shown. There's a scrollbar that appears if there's more stock than window space. The pane to the right contains the party gold and data about the selected item. We display the number of items already in the party inventory or equipped by a party member. This way you can easily tell if you're buying or selling something you already own.

The next pane contains the selected item's description.

The final pane tells us which party members can use the item. This pane contains the character sprite for each member of the party. If the party member can equip an item, they get a little white arrow beneath them. If the item is "better" than what they currently have, they get a green arrow at the top left of their sprite. If the selected item is already equipped, they get a small E at the bottom left of the sprite. The equipment hint icons can be seen in Figure 4.9.

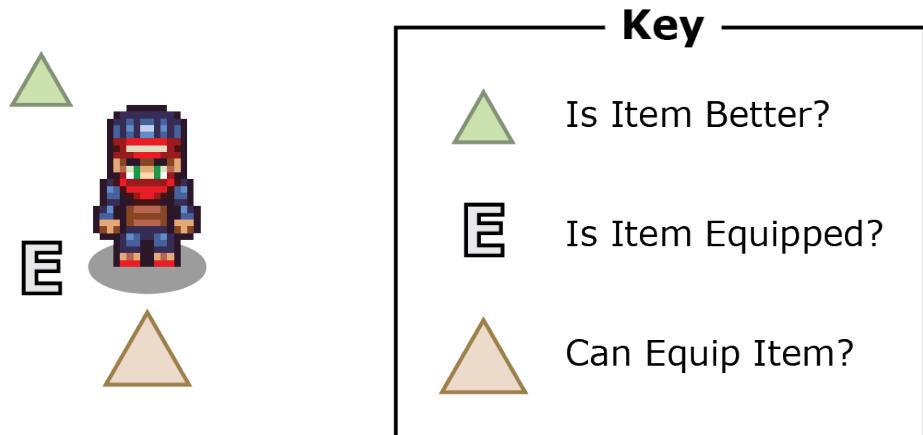


Figure 4.9: Equipment hint icons for a party member.

You can probably see that there's a surprising amount of logic going on, even for our relatively simple shop! Let's consider the interaction flow and then dive into some code.

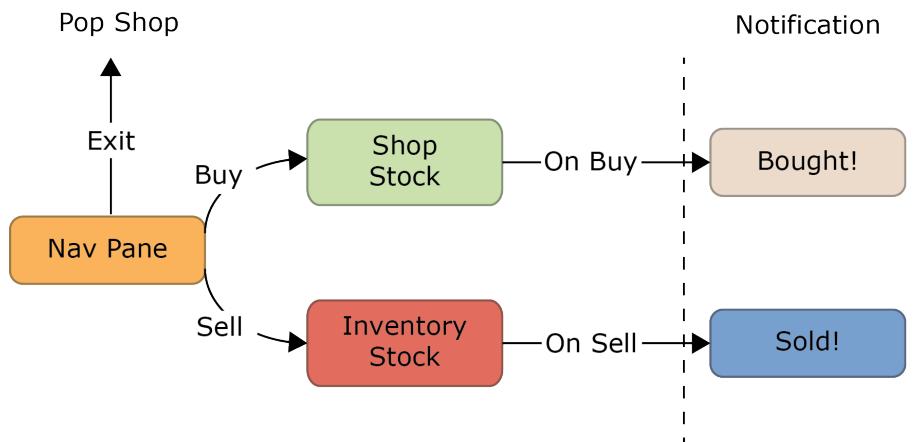


Figure 4.10: Shop flow sketch.

The flow diagram, in Figure 4.10, shows how the player moves through the shop UI. Choosing "Exit" pops the ShopState off the stack. Choosing "Buy" switches focus to the buy pane and the navigation pane is replaced with a "What do you want to buy?" message. The "Sell" choice works in a similar way.

On buying or selling an item, we display a dialog box to tell the player the transaction was successful. After dismissing, the dialog box focus returns to the shop state. Press Backspace and you'll change focus from the shop pane to the navigation pane. Press Backspace again and you'll exit the shop.

Shop Code Foundations

Before creating ShopState.lua itself, we need to update the existing code (as always!). Let's start with prices. In our world prices are fixed, so we'll add a price field to each item def. Copy the code from Listing 4.9.

```
ItemDB =  
{  
    [-1] =  
    {  
        -- code omitted  
        price = 0  
    },  
    {  
        -- code omitted  
        price = 300,  
    },  
    {  
        -- code omitted  
        price = 500,  
    },  
    {  
        -- code omitted  
        price = 1000  
    },  
    {  
        -- code omitted  
        price = 300  
    },  
    {  
        -- code omitted  
        price = 500,  
    },  
    {  
        -- code omitted  
        price = 1000,  
    },  
    {  
        -- code omitted  
        price = 100,  
    },  
    {  
        -- code omitted  
        price = 250,  
    },  
}
```

```

    -- code omitted
    price = 2000,
},
{
    -- code omitted
    price = 50,
},
{
    -- code omitted
    price = 100,
},
{
    -- code omitted
    price = 100,
},
{
    -- code omitted
    price = 1000
},
}

```

Listing 4.9: Adding a price field in ItemDB.lua.

The prices in Listing 4.9 represent reasonable values for our game items, but feel free to adjust them.

To display the items in the shop we use the `World:DrawItem` function but with a few updates. Copy the code from Listing 4.10.

```

function World:DrawItem(menu, renderer, x, y, item, color)

    color = color or Vector.Create(1, 1, 1, 1)

    if item then
        local itemDef = ItemDB[item.id]
        local iconSprite = self.mIcons:Get(itemDef.icon or itemDef.type)
        if iconSprite then
            iconSprite:SetColor(color)
            iconSprite:SetPosition(x + 6, y)
            renderer:DrawSprite(iconSprite)
        end
        renderer:AlignText("left", "center")
        renderer:DrawText2d(x + 18, y, itemDef.name, color)

        if item.count then

```

```

        local right = x + menu.mSpacingX - 64
        renderer:AlignText("right", "center")
        local countStr = string.format("%02d", item.count)
        renderer:DrawText2d(right, y, countStr, color)
    end
else
    renderer:AlignText("center", "center")
    renderer:DrawText2d(x + menu.mSpacingX/2, y, " - ", color)
end
end

```

Listing 4.10: Extending DrawItem with optional count and color parameters. In World.lua.

The changes to the DrawItem function are minor; we've added an extra color field. The color is used for the text and icon color and defaults to white. There's an extra if statement, if item.count, that only draws out the item count if the field exists. When the player is selling items we show an item count, but for the shop inventory we don't draw a count. Remember, our shop has unlimited stocks.

When an item get focus, such as the "Bone Sword", we want to ask how many of the party members have it equipped. To find this out we'll add a new EquipCount function to the Party and Actor class. Copy the code from Listing 4.11.

```

function Party:EquipCount(itemId)

    local count = 0

    for _,v in pairs(self.mMembers) do
        count = count + v:EquipCount(itemId)
    end

    return count
end

```

Listing 4.11: Adding EquipCount to the Party. In Party.lua.

The EquipCount for the Party just calls EquipCount for each actor in the party and returns the summed result. Copy the EquipCount function for the Actor from Listing 4.12.

```

function Actor:EquipCount(itemId)

    local count = 0

```

```

for _, v in pairs(self.mEquipment) do
    if v == itemId then
        count = count + 1
    end
end

return count
end

```

Listing 4.12: Adding EquipCount to the Actor. In Actor.lua.

The Actor.EquipCount function looks through all the equipment slots for any item with a matching id, and if found it increases a count. The count is then returned.

Before moving on to implementing the ShopState, there's one last thing to note. In the art folder there's a new file up_caret.png. This is used in the shop to indicate which characters can use a selected item. If you're using your own codebase, copy it over (from example Quests-5) and add it to the manifest.

Implementing the Shop State

We're ready to implement the shop state. There's a quite a lot going on so we'll add the code section by section, starting with the constructor.

The Constructor

Let's begin by creating a new state called ShopState.lua. Add the file to the manifest and dependencies files. Below is a bare bones state that creates some backing panels arranged as in Figure 4.8.

```

ShopState = {}
ShopState.__index = ShopState
function ShopState:Create(stack, world, def)

    def = def or {}

    local this
    this =
    {
        mStack = stack,
        mWorld = world,
        mDef = def,
        mPanels = {},
    }

```

```

mName = def.name or "Shop",
mActors = {},
mActorCharMap = {},
}

for _, v in pairs(this.mWorld.mParty.mMembers) do
    table.insert(this.mActors, v)
    local def = gCharacters[v.mId]
    local char = Character>Create(def, {})
    this.mActorCharMap[v] = char
end

local layout = Layout>Create()
layout:Contract('screen', 118, 40)
layout:SplitHorz('screen', 'top', 'bottom', 0.12, 1)
layout:SplitVert('top', 'title', 'choose', 0.75, 1)
layout:SplitHorz('bottom', 'top', 'char', 0.55, 1)
layout:SplitHorz('char', 'desc', 'char', 0.25, 1)
layout:SplitVert('top', 'inv', 'status', 0.35, 1)

this.mPanels =
{
    layout>CreatePanel("title"),
    layout>CreatePanel("choose"),
    layout>CreatePanel("desc"),
    layout>CreatePanel("char"),
    layout>CreatePanel("inv"),
    layout>CreatePanel("status")
}

this.mLayout = layout

setmetatable(this, self)
return this
end

function ShopState:Enter() end
function ShopState:Exit() end
function ShopState:HandleInput() end
function ShopState:Update(dt) end

function ShopState:Render(renderer)
    for k, v in ipairs(self.mPanels)do
        v:Render(renderer)
    end

```

```
end
```

Listing 4.13: Laying out the panels of the shop. In ShopState.lua.

The ShopState, in Figure 4.8, renders out the shop's backing panels. You can see the panes in Figure 4.11. The only two functions that matter at the moment are the constructor and the Render function.

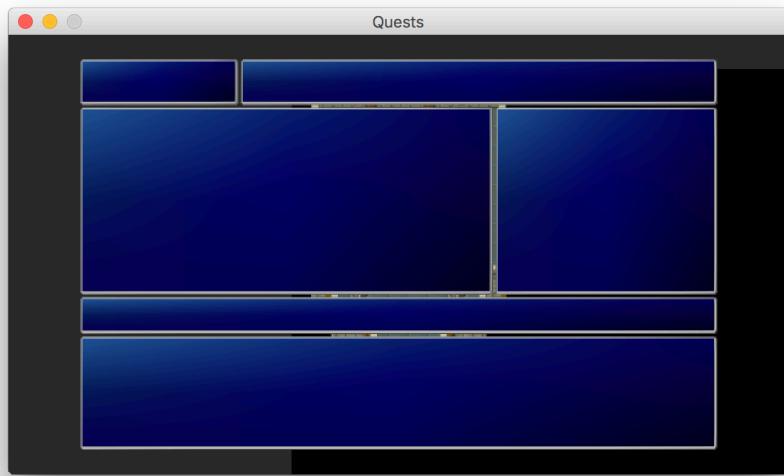


Figure 4.11: Shop panel layout.

The constructor takes in 3 parameters: the stack, the world, and the def. The stack is the stack the shop state is pushed on and off. The world is used to get the party and inventory information. The def is a shop definition table that describes the shop's stock and name. We can create many different types of shop from the ShopState class by changing the values in the def.

We store the stack, world and def in the this table. We also add a table for the backing panels and the name of the shop. The shop name defaults to "Shop". The this table also contains mActors, a list of all the actors in the party, and mActorCharMap, which links the actors to their characters, displayed at the bottom of the shop.

After the this table is set up, the mActorCharMap is filled with characters representing each actor. We make a layout object to create the backing panels. The panels are laid out according to our mockup. We fill in the mPanel table with all the panels we want to render, and store the layout in mLayout.

The Render function draws out all the panels. There are a few empty functions that the stack requires but which we don't use.

To test out this code, let's change main.lua to automatically push a ShopState onto the stack.

Copy the code from Listing 4.14.

```
local startPos = Vector.Create(5, 9, 1)

local hero = Actor:Create(gPartyMemberDefs.hero)
local thief = Actor:Create(gPartyMemberDefs.thief)
local mage = Actor:Create(gPartyMemberDefs.mage)
gWorld.mParty:Add(hero)
gWorld.mParty:Add(thief)
gWorld.mParty:Add(mage)
gStack:Push(ExploreState:Create(gStack, CreateTownMap(), startPos))

gWorld.mGold = 1000
gWorld:AddItem(1, 3)
gWorld:AddItem(2, 3)
gWorld:AddItem(10, 5)

-- Equip the sword (from inventory)
hero:Equip(Actor.EquipSlotId[1], {id = 1, count = 3})

local shopDef =
{
    name = "Arms Shop",
    stock =
    {
        { id = 1, price = 300 },
        { id = 2, price = 350 },
        { id = 4, price = 300 },
        { id = 5, price = 350 },
        { id = 7, price = 300 },
        { id = 8, price = 300 },
    },
    sell_filter = "arms",
}
gStack:Push(ShopState:Create(gStack, gWorld, shopDef))
```

Listing 4.14: Automatically pushing a shop onto the state stack. In main.lua.

In the main.lua we create a full party and add them to the world. We also add gold and items to the party inventory and equip the hero with a bone sword.

The shopDef is new but reasonably self explanatory. The name field is the name of the shop. The stock table is a list of items the shop sells with prices. The sell filter is used to filter the items the shop is willing to purchase from the player. In this case, "arms" means the shop buys weapons or armor but not potions. The exact implementation of the filter will become clearer as we implement the ShopState.

Finally the ShopState is pushed onto the stack. Run the code and you'll be presented with a set of panels that look like the skeleton of the shop as in Figure 4.11. Let's get to work filling that skeleton in!

Let's beef up the constructor with some selection menus. Copy the code from Listing 4.15.

```
function ShopState:Create(stack, world, def)

    -- code omitted

    this =
    {
        -- code omitted
        mState = "choose", -- "choose", "buy", "sell"
        mTabs =
        {
            ["buy"] = 1,
            ["sell"] = 2,
            ["exit"] = 3,
        },
        mChooseSelection = Selection>Create
        {
            data =
            {
                "Buy",
                "Sell",
                "Exit",
            },
            rows = 1,
            columns = 3,
            OnSelection = function(...) this:ChooseClick(...) end,
        },
        mStock = Selection>Create
        {
            data = def.stock,
            displayRows = 5,
            RenderItem = function(...) this:RenderStock(...) end,
            OnSelection = function(...) this:OnBuy(...) end,
        },
    }
```

```

mInventory = nil,
mScrollbar = Scrollbar>Create(Texture.Find("scrollbar.png"), 145),
mFilterList =
{
    ['all'] = function(def) return true end,
    ['arms'] = function(def)
        return def.type == 'armor' or def.type == 'weapon'
    end,
    ['support'] = function(def)
        return def.type == 'useable'
    end
},
mCanUseSprite = Sprite.Create(),
-- Used every frame
mItemInvCount = 0,
mItemEquipCount,
mItemDescription = 0,
}

```

Listing 4.15: Filling out the this table for the shop. In ShopState.lua.

We've expanded the `this` table significantly, so let's go through it field by field. First is `mState` which stores a string that tells us what state the shop is in. The shop has three possible states.

- Buy
- Sell
- Choose

The `mState` field defaults to "choose". This means the selection box with the "Buy", "Sell" and "Exit" options has focus. If we choose "Buy" we enter the buy state, and if we choose "Sell" we enter the sell state. When we enter these states, we initialize list boxes and change the cursor focus.

Next we define `mTabs`. This contains the text for each option in the "choose" state and also maps the text to a number. Associating each state with a number makes the programming simple later. We define `mChooseSelection`, which draws the options in the navigation pane and lets the player browse them. The data is a list of the three options. We lay these out in a straight line by defining one column with three rows, as you can see in Figure 4.12.



Buy

► Sell

Exit

Figure 4.12: The shop navigation pane.

When the player chooses one of the options, we call `OnChooseClick`. This function handles changing the state, or exiting if the player chooses "Exit". We'll define it shortly.

Near the bottom of the shop we display the stock. When "Buy" is selected, the stock uses a selection menu to show the inventory stock. The stock data is taken from the def. It displays the stock in one column with five rows. Each item in the shop displays a price and an icon to indicate the item type. To render the shop items, we use the `RenderStock` callback. When the player buys an item, we use the `OnBuy` callback.

??? PICTURE HERE? -> Flow diagram for stock and inventory display?

The `mInventory` field stores a selection menu that displays a filtered version of the player inventory. The `mInventory` menu shows what the shop owner is willing to purchase from the player. When the player buys or sells an item, their inventory changes. This means we need to recreate the `mInventory` menu whenever a change occurs. We handle this in a special function, `CreateInvSelection`, that we'll get to later.

There are times when there's so much stock in the shop that it can't all be shown on screen at once. This is going to happen often when the player is selling items. To make the stock easier to browse, we use a scrollbar. The scrollbar hooks up to the `mInventory` or `mStock` menus when there are more items than can fit on screen at once. The height is set to 145 pixels, which makes it fit nicely inside the stock panel.

After setting up the scrollbar, we set up the `mFilterList`. The `mFilterList` is a list of functions that filter the player's inventory. Remember in the shop def the `sell_filter` field was set to `arms?` This field is a key for the `mFilterList` table.

There are three filter keys:

- **all** - The shop will buy anything the player has.
- **arms** - The shop will buy armor or weapons.
- **support** - The shop buys support items like potions.

Each filter takes in an item def and returns true if the shop is willing to buy that item.

After the filter list, there are a few more fields. The `mCanUseSprite` is a sprite that's drawn under a character if they can use a selected item. It's also used to draw a green arrow if the item is better than what they're currently using.

The `mItemInvCount`, `mItemEquipCount`, and `mItemDescription` fields all store data about the currently selected item.

- `mItemInvCount` - How many of this item the player has in the inventory.
- `mItemEquipCount` - How many of these items the party members have equipped.
- `mItemDescription` - The description of the selected item.

This is a big constructor! Let's finish it off. Copy the code from Listing 4.16.

```
function ShopState:Create(stack, world, def)

    def = def or {}

    local this
    this =
    {
        -- code omitted
    }
    -- code omitted

    this.mCanUseSprite:SetTexture(Texture.Find("up_caret.png"))

    local filterId = def.sell_filter or 'all'
    this.InvFilter = this.mFilterList[filterId]

    local cX = layout:Left("choose") + 24
    local cY = layout:MidY("choose")
    this.mChooseSelection:SetPosition(cX, cY)

    local sX = layout:Left("inv") - 16
    local sY = layout:Top("inv") - 18
    this.mPriceX = layout:Right("inv") - 56
    this.mStock:SetPosition(sX, sY)

    local scrollX = layout:Right("inv") - 14
    local scrollY = layout:MidY("inv")
    this.mScrollbar:SetPosition(scrollX, scrollY)

    setmetatable(this, self)
    this.mInventory = this>CreateInvSelection()
    this.mStock:HideCursor()
    this.mInventory:HideCursor()
    return this
end
```

Listing 4.16: Setting up code at the end of the ShopState constructor. In ShopState.lua.

In Listing 4.16 we set up the last details for the shop state.

We set the mCanUseSprite texture to the up_caret.png. The image is used to indicate if a player can use or equip an item.

We store the shop buy filter in `filterId`. This filter is set to the `sell_filter` from the def or it defaults to all. We use the `filterId` to get the filter function from the `mFilterList`, which we store in `this.InvFilter`.

Next we position some of the UI elements. We position the `mChooseSelection` menu in the middle of the “choose” panel. We position the `mStock` selection menu in the middle of the “inv” panel. In `mPriceX` we store the X position where we’ll render the price of each item. Finally we position the scrollbar to the right and center of the `inv` panel.

Next we hook up the metatable and call `CreateInventorySelection`. This creates the menu for a filtered version of the player’s inventory. The focus cursor is hidden for the inventory and stock menus, leaving the focus active on the `mChooseSelection` menu, where we want it.

That’s the shop constructor done! It’s pretty lengthy but the shop has a lot going on. Now let’s build up the rest of the class.

Support Functions

The shop needs numerous support functions to work its magic. We’ll implement the support function now, and they’ll get used in the `HandleInput` and `Render` functions that we’ll add at the end.

The shop needs to filter the player inventory and create a selection menu for items to sell. To do this, let’s implement `CreateInvSelection`. Copy the code from Listing 4.17.

```
function ShopState:CreateInvSelection()

    local stock = self.mWorld:FilterItems(self.InvFilter)

    local inv = Selection:Create
    {
        data = stock or {},
        displayRows = self.mStock.mDisplayRows,
        RenderItem = function(...) self:RenderInventory(...) end,
        OnSelection = function(...) self:OnSell(...) end,
        spacingX = 225,
    }
    local x = self.mStock.mX
    local y = self.mStock.mY

    inv:SetPosition(x, y)
    inv:HideCursor() -- hidden by default
    return inv
end
```

Listing 4.17: The `CreateInvSelection` function. In `ShopState.lua`.

CreateInvSelection creates a new selection menu from the player's inventory. It filters the inventory so it only contains items the shop is willing to buy.

The function starts by filtering the player inventory using the filter stored in InvFilter. This returns a copy of the inventory, minus any filtered items, that we use as a datasource for the inv selection menu. The render callback for the menu is set to RenderInventory and the selection callback is set to OnSell. We set the column spacing to 225 pixels. The menu's position and number of rows are taken from the mStock selection menu. Both the mStock and the mInventory appear in the same place, just at different times. The selection menu has its cursor hidden by default, so it doesn't appear to have focus.

Let's take a look at the OnBuy and OnSell callbacks next. The OnBuy callback is called when selecting something from the shop's stock menu. The OnSell callback is called when the player selects an item to sell. Copy the code from Listing 4.18.

```
function ShopState:OnBuy(index, item)

    local gold = self.mWorld.mGold

    if gold < item.price then
        return -- can't afford
    end

    gold = gold - item.price
    self.mWorld.mGold = gold
    self.mWorld:AddItem(item.id)

    local name = ItemDB[item.id].name
    local message = string.format("Bought %s", name)
    self.mStack:PushFit(gRenderer, 0, 0, message)
end

function ShopState:OnSell(index, item)

    local gold = self.mWorld.mGold
    local def = ItemDB[item.id]
    self.mWorld:RemoveItem(item.id)
    gold = gold + def.price
    self.mWorld.mGold = gold

    -- Refresh inventory display
    self.mInventory = self>CreateInvSelection()
    self.mInventory>ShowCursor()

    -- Attempt to restore the index
    for i = 1, index - 1 do
```

```

        self.mInventory:MoveDown()
    end

    local message = string.format("Sold %s", def.name)
    self.mStack:PushFit(gRenderer, 0, 0, message)
end

```

Listing 4.18: Buy and Sell callbacks. In ShopState.lua.

OnBuy and OnSell are both callbacks from selection menus. When called they are passed two parameters, the item index in the menu and the item itself. Each item has an id field that can be used to look up the item def.

OnBuy first gets the player's gold amount, If there's not enough gold to afford the item's price, then the function returns immediately. No sale. Otherwise the price is subtracted from the player's gold store and the item is added to the inventory. We notify the player by pushing a message box onto the stack announcing that the item has been purchased.

OnSell removes the item from the player inventory and increases the player's gold by the item price (as defined in the item def). If the player only has one item to sell, it means the item needs to be totally removed from the inventory menu when sold. Therefore, after selling an item we recreate the menu and move the cursor down to where the item used to be. This stops the menu state from being out of sync with the underlying inventory data. As before, we tell the player something's happened using a dialog box announcing that the item has been sold.

Let's implement RenderStock and RenderInventory next. Why two functions? Well, the inventory menu and stock menu are slightly different. The stock menu is a list of items and prices. The inventory menu is a list of items, an item count and price. The inventory menu has a little more data it needs to squeeze into each row. The stock menu also grays out any item the player cannot afford.

Copy the code from Listing 4.19.

```

function ShopState:RenderStock(menu, renderer, x, y, item)
    if item == nil then
        return
    end

    local color = nil
    if self.mWorld.mGold < item.price then
        color = Vector.Create(0.6, 0.6, 0.6, 1)
    end

    self.mWorld:DrawItem(menu, renderer, x, y, item, color)

    renderer:AlignTextX("right")

```

```

local priceStr = string.format(": %d", item.price)
renderer:DrawText2d(self.mPriceX, y, priceStr, color)
end

function ShopState:RenderInventory(menu, renderer, x, y, item)

    if not item then
        return
    end

    self.mWorld:DrawItem(menu, renderer, x, y, item)

    local def = ItemDB[item.id]
    renderer:AlignTextX("right")
    local priceStr = string.format(": %d", def.price)
    renderer:DrawText2d(self.mPriceX, y, priceStr)
end

```

Listing 4.19: The RenderStock and RenderInventory functions. In ShopState.lua.

The functions in Listing 4.19 are similar. They return immediately if the item is nil. They both use mWorld:DrawItem to draw the item name and icon.

The RenderStock function draws each item that's for sale in the shop. The text color defaults to white, but if the player can't afford the item we set the color variable to gray. The color is used for the mWorld:DrawItem function and the price text. Each item of stock contains a price field. We draw the price to the right of the stock panel using the mPriceX field to position it.

The RenderInventory function draws out the player's filtered inventory. The price comes from the item def rather than the item entry itself, and no text is grayed out.

Next let's implement two functions to determine if we show the shop stock or the player inventory. Copy the code from Listing 4.20.

```

function ShopState:BackToChooseState()
    self.mState = "choose"
    self.mChooseSelection:ShowCursor()
    self.mInventory:HideCursor()
    self.mStock:HideCursor()
end

function ShopState:ChooseClick(index, item)

    if index == self.mTabs.buy then
        self.mChooseSelection:HideCursor()

```

```

        self.mStock:ShowCursor()
        self.mState = "buy"
    elseif index == self.mTabs.sell then
        self.mChooseSelection:HideCursor()
        self.mInventory:ShowCursor()
        self.mState = "sell"
    else
        self.mStack:Pop() -- Remove self off the stack
    end
end

```

Listing 4.20: Functions to control the shop state. In ShopShate.lua.

In Listing 4.20 the BackToChoose function moves focus from the buy / sell pane to the navigation pane. We set the shop state to “choose”, hide the cursor for the stock and inventory menus, and show the cursor for the choose menu.

The ChooseClick function is a callback that’s called when the player selects “Buy”, “Sell” or “Exit” from the choose menu.

The “Buy”, “Sell” or “Exit” options have their index stored in the mTabs table. If the index is 0, the player has chosen to enter the buy state. We hide the choose menu cursor and reveal the cursor for the mStock menu. We set mState to “buy” which makes the renderer and input work for the shop stock.

If the index is 1, we enter the sell state. We hide the choose menu cursor and reveal the mInventory cursor. We set the state to “Sell” which makes the renderer and input work for the player inventory.

If the index is 2, then we exit by popping the shop state off the stack, returning the player to the game.

We’re not quite done with the mChooseSelection yet, as we need to implement RenderChooseFocus. This function displays the correct data in the buy / sell pane as the player flicks between the “Buy”, “Sell” and “Exit” options. Copy the code from Listing 4.21.

```

function ShopState:RenderChooseFocus(renderer)
    renderer:AlignText("left", "center")
    self.mChooseSelection:Render(renderer)

    local focus = self.mChooseSelection:GetIndex()

    if focus == self.mTabs.buy then
        self.mStock:Render(renderer)
        self:UpdateScrollbar(renderer, self.mStock)
    elseif focus == self.mTabs.sell then

```

```

        self.mInventory:Render(renderer)
        self:UpdateScrollbar(renderer, self.mInventory)
    else

        end
end

```

Listing 4.21: Function to update shop presentation. In ShopState.lua.

When the choose menu has focus, the RenderChooseFocus function is called each frame. We check the selected choice; if it's "Buy" we render the mStock, if it's "Sell" we render the mInventory, and if it's "Exit" we rendering nothing. This function is called from the main Render function that we'll implement soon.

Ok, let's finish off the support code with a grab bag of three functions. Copy the code from Listing 4.22.

```

function ShopState:IsEquipmentBetter(actor, itemDef)

    if itemDef.type == "weapon" then
        local diff = actor:PredictStats("weapon", itemDef)
        return diff.attack > 0
    elseif itemDef.type == "armor" then
        local diff = actor:PredictStats("armor", itemDef)
        return diff.defense > 0
    end

    return false
end

function ShopState:SetItemData(item)
    if item then
        local party = self.mWorld.mParty
        self.mItemCount = self.mWorld:ItemCount(item.id)
        self.mItemEquipCount = party:EquipCount(item.id)
        local def = ItemDB[item.id]
        self.mItemDef = def
        self.mItemDescription = def.description
    else
        self.mItemCount = 0
        self.mItemEquipCount = 0
        self.mItemDescription = ""
        self.mItemDef = nil
    end
end

```

```

function ShopState:UpdateScrollbar(renderer, selection)

    if selection:PercentageShown() <= 1 then
        local scrolled = selection:PercentageScrolled()
        local caretScale = selection:PercentageShown()
        self.mScrollbar:SetScrollCaretScale(caretScale)
        self.mScrollbar:SetNormalValue(scrolled)
        self.mScrollbar:Render(renderer)
    end
end

```

Listing 4.22: Final support functions required for the shop state. In `ShopState.lua`.

First up is `IsEquipmentBetter`. This takes in an actor and a piece of equipment and answers the question, “If the player equips this, will they be stronger?” `IsEquipmentBetter` only works for weapons and armor; it’s harder to accurately answer this question for accessories. We use the actor’s `PredictStats` function to compare the attack and defense rating between the equipped and the selected item. This is a good enough comparison for the shop hint information.

The `SetItemData` function takes in an item and calculates some information about it. This function records the item def, number of items in the inventory, number of items equipped, and the description. If a null is passed in, all the item details are cleared.

The final function is `UpdateScrollbar` and, unsurprisingly, it updates the scrollbar and renders it out. `UpdateScrollbar` takes two parameters, the renderer and the selection menu it’s hooked up to. The function first checks if the selection menu needs a scrollbar at all. If the percentage of the selection menu shown is 100%, the scrollbar isn’t needed and therefore isn’t drawn.

If the selection menu can’t display all its data, we set up the scrollbar and draw it.

That’s the support functions finished. Next let’s implement `HandleInput` and start using some of these functions.

Shop Input

Copy the code from Listing 4.23.

```

function ShopState:HandleInput()
    if self.mState == "choose" then
        self.mChooseSelection:HandleInput()
    elseif self.mState == "buy" then
        self.mStock:HandleInput()
    end
end

```

```

    if Keyboard.JustPressed(KEY_BACKSPACE) then
        self:BackToChooseState()
    end

elseif self.mState == "sell" then
    self.mInventory:HandleInput()

    if Keyboard.JustPressed(KEY_BACKSPACE) then
        self:BackToChooseState()
    end
end

end

```

Listing 4.23: The HandleInput function. In ShopState.lua.

The selections menus do the heavy lifting when it comes to input. We check the shop state and tell the correct selection menu to handle the player's input.

In the “choose” state the mChooseSelection menu is told to handle the input. In the “buy” state, the mStock menu handles the input. In the “sell” state, we tell the mInventory menu to handle input. If Backspace is pressed in the “sell” or “buy” state, the we change the state back to “choose”.

Render

Ok, let's start drawing the shop out. To prevent the render function from getting too big, we've split the character rendering into a separate function.

The DrawCharacters function is long because it has a lot of positioning information. Copy the code from Listing 4.24.

```

function ShopState:DrawCharacters(renderer, itemDef)

    local x = self.mLayout:Left("char") + 64
    local y = self.mLayout:MidY("char")

    local selectColor = Vector.Create(1, 1, 1, 1)
    local powerColor = Vector.Create(0, 1, 1, 1)
    local shadowColor = Vector.Create(0, 0, 0, 1)

    for _, actor in ipairs(self.mActors) do

        local char = self.mActorCharMap[actor]

```

```

local entity = char.mEntity
entity.mSprite:SetPosition(x, y)
entity:Render(renderer)

local canUse = false
local greaterPower = false
local equipped = false

if itemDef then
    canUse = actor:CanUse(itemDef)
    equipped = actor:EquipCount(itemDef.id) > 0
    greaterPower = self:IsEquipmentBetter(actor, itemDef)
end

if canUse then
    local selectY = y - (entity.mHeight / 2)
    selectY = selectY - 6 -- add space
    self.mCanUseSprite:SetPosition(x, selectY)
    self.mCanUseSprite:SetColor(selectColor)
    renderer:DrawSprite(self.mCanUseSprite)
end

if greaterPower then
    local selectY = y + (entity.mHeight / 2)
    local selectX = x - (entity.mWidth / 2)
    selectX = selectX - 6 -- spacing
    self.mCanUseSprite:SetPosition(selectX, selectY)
    self.mCanUseSprite:SetColor(powerColor)
    renderer:DrawSprite(self.mCanUseSprite)
end

if equipped then
    local equipX = x - (entity.mWidth / 2)
    local equipY = y - (entity.mWidth / 2)
    equipX = equipX - 3

    renderer:AlignText("right", "center")
    renderer:DrawText2d(equipX + 2, equipY - 2, "E", shadowColor)
    renderer:DrawText2d(equipX, equipY, "E")
end

x = x + 64
end
end

```

Listing 4.24: The function to draw the characters at the bottom of the screen. In ShopState.lua.

The DrawCharacter function draws each member of the party at the bottom of the screen. Each member has 3 elements that may be drawn depending on the currently selected item. If the character can use an item, the mCanUseSprite is rendered at the bottom of the character's feet. If the selected item is better than the item equipped, a small green arrow is rendered at the top left of the player. If the player has the currently selected item equipped already, a small letter E is drawn at the bottom left of the character. You can see the equipment hints in Figure 4.13.



Figure 4.13: Screenshot of the equipment hints in action.

Let's run through exactly how this all goes down in the function. We use x and y variables to mark the center left of the panel we'll draw the characters in. Then we define three colors:

- select color - colors the mCanUseSprite when the character can use the item selected.
- power color - colors the mCanUseSprite when it's better than the character's current item.
- shadow color - used for the letter E drop shadow. Shadowing the letter makes it more readable as it's quite small.

Every character in the party is drawn from the left to the right. We iterate through the self.mActor table and use the mActorCharMap to find the associated character object. We use three local variables: canUse, greaterPower, and equipped, all initialized to false to determine what hints we draw around the character.

If an itemDef exists, we call CanUse to test if the character can equip it. If EquipCount tells us the character has one or more of these items equipped, we set equipped to true. To determine if greaterPower should be set to true, we call IsEquipmentBetter.

With these values set, we render all the item hints around the character. Before the loop ends we increase x by 64 pixels, which is where we draw the next character.

Ok. That's the character drawing handled. All that remains is the Render function itself. Copy the code from Listing 4.25.

```
function ShopState:Render(renderer)
    for k, v in ipairs(self.mPanels)do
        v:Render(renderer)
    end

    renderer:AlignText("center", "center")
    renderer:ScaleText(1.25, 1.25)
    local tX = self.mLayout:MidX("title")
    local tY = self.mLayout:MidY("title")
    renderer:DrawText2d(tX, tY, self.mName)

    local x = self.mLayout:MidX("choose")
    local y = self.mLayout:MidY("choose")

    if self.mState == "choose" then

        self:RenderChooseFocus(renderer)
        self:SetItemData(nil)

    elseif self.mState == "buy" then

        local buyMessage = "What would you like?"
        renderer:AlignText("center", "center")
        renderer:DrawText2d(x, y, buyMessage)
        renderer:AlignText("left", "center")
        self.mStock:Render(renderer)
        local item = self.mStock:SelectedItem()
        self:SetItemData(item)
        self:UpdateScrollbar(renderer, self.mStock)

    elseif self.mState == "sell" then
```

```

local sellMessage = "Show me what you have."
renderer:AlignText("center", "center")
renderer:DrawText2d(x, y, sellMessage)
renderer:AlignText("left", "center")
self.mInventory:Render(renderer)
local item = self.mInventory:SelectedItem()
self:SetItemData(item)
self:UpdateScrollbar(renderer, self.mInventory)
end

-- Let's write the status bar
renderer:AlignText("right", "center")
renderer:ScaleText(1.1, 1.1)
local statusX = self.mLayout:MidX("status") - 4
local statusY = self.mLayout:Top("status") - 18
local valueX = statusX + 8

local gpText = "GP:"
local invText = "Inventory:"
local equipText = "Equipped"

local gpValue = string.format("%d", self.mWorld.mGold)
local invValue = string.format("%d", self.mItemCount)
local equipValue = string.format("%d", self.mEquipCount)

renderer:DrawText2d(statusX, statusY, gpText)
renderer:DrawText2d(statusX, statusY - 32, invText)
renderer:DrawText2d(statusX, statusY - 50, equipText)

renderer:AlignText("left", "center")
renderer:DrawText2d(valueX, statusY, gpValue)
renderer:DrawText2d(valueX, statusY - 32, invValue)
renderer:DrawText2d(valueX, statusY - 50, equipValue)

renderer:AlignText("left", "center")
renderer:ScaleText(1,1)
local descX = self.mLayout:Left("desc") + 8
local descY = self.mLayout:MidY("desc")
renderer:DrawText2d(descX, descY, self.mItemDescription)

self:DrawCharacters(renderer, self.mItemDef)
end

```

Listing 4.25: Implementing the Shop render function. In ShopState.lua.

Like the DrawCharacters function, the Render function is pretty lengthy but it's mostly layout code.

We start by rendering all the panels. Then we render the shop title in the top left panel. We use the x, y from the middle of the "title" layout to center align the text.

We store the center position of the "choose" panel in variables x and y. We use this position to replace the "Buy", "Sell" and "Exit" options with a message when in the buy or sell state.

If mState is "choose", we call RenderChooseFocus to draw the stock or inventory menu, depending on the current selection. We also set the item data to nil because you cannot select an item while in the "choose" state.

If mState is "buy", we draw the text "What would you like to buy?" in the navigation pane. We render the mStock menu, set the item data, and update the scrollbar.

If mState is "sell", we draw the text "Show me what you have." in the navigation pane, we render the mInventory menu, set the item data, and update the scrollbar.

The status pane contains three label, value pairs of text. We store the top right position of the pane in statusX and statusY. We render the labels right aligned and the values left aligned. The labels are positioned using statusX and the values are positioned using valueX. This centers the labels and values around the middle of the status pane. The data for values comes from the SetItemData function that we called earlier.

The description for the currently selected item is drawn in the description panel. The text is aligned to the center left of the panel.



Figure 4.14: A fully functioning shop.

The final line of the Render function calls DrawCharacters to draw the party in the bottom pane. Try it out. You'll see something like Figure 4.14. It's a fully functioning shop, where you can buy and sell items! Next we'll add triggers to the map to open the shop when the player talks to the shopkeeper.

Triggering the Shop

To add the shops to the map, we'll use the trigger system. The shop trigger creates a shop state and pushes it onto the stack. Let's start by adding a new action called "OpenShop" in Actions.lua. Copy the code from Listing 4.26.

```
Actions =
{
    -- code omitted

    OpenShop = function(map, def)
        return function(trigger, entity, tX, tY, tLayer)
            gStack:Push(ShopState>Create(gStack, gWorld, def))
        end
    end,
}
```

Listing 4.26: Adding a new action: OpenShop. In Actions.lua.

The OpenShop action takes in a shop def, creates a shop state, and pushes it onto the stack.

It seems sensible to associate the shop definition information with the map where it appears, so let's add the shop defs into the CreateTownMap function. Copy the code from Listing 4.27.

```
function CreateTownMap()

    local shopDefPotions =
    {
        name = "Potion Shop",
        stock =
        {
            { id = 10, price = 50 },
            { id = 11, price = 500 },
            { id = 12, price = 100 },
        }
    }
}
```

```

    },
    sell_filter = "support",
}

local shopDefArms =
{
    name = "Arms Shop",
    stock =
    {
        { id = 1, price = 300 },
        { id = 2, price = 350 },
        { id = 4, price = 300 },
        { id = 5, price = 350 },
        { id = 7, price = 300 },
        { id = 8, price = 300 },
    },
    sell_filter = "arms",
}
-- code omitted

```

Listing 4.27: Adding shop definitions to the map. In map_town.lua.

Our starting town has two shops, so we have two shop definitions. The first is the potion shop, selling healing, mana and resurrect potions and buying only support / “useable” items. The second shop is the weapons and armor shop, selling and buying weapons and armor.

The shop definitions are local to the town function, so we can reference them directly when we create the triggers. Copy the code from Listing 4.28 to set up the triggers.

```

actions =
{
    -- code omitted

    open_potion_shop = { id = "OpenShop", params = { shopDefPotions } },
    open_arms_shop = { id = "OpenShop", params = { shopDefArms } }
},
trigger_types =
{
    -- code omitted

    potion_shop = { OnUse = "open_potion_shop" },
    arms_shop = { OnUse = "open_arms_shop" },
}

```

```
triggers =
{
    -- code omitted

    { trigger = "potion_shop", x = 6, y = 44 },
    { trigger = "arms_shop", x = 57, y = 46 },
},
```

Listing 4.28: Adding the shop triggers to the town map. In map_town.lua.

In Listing 4.28 we define two actions that open shops using the shop defs. The triggers are positioned in front of each of the shopkeepers. The shops opens when you face a shopkeeper and press Space, therefore we use the “OnUse” trigger type. If you fire up the game, you can now check out the shops!

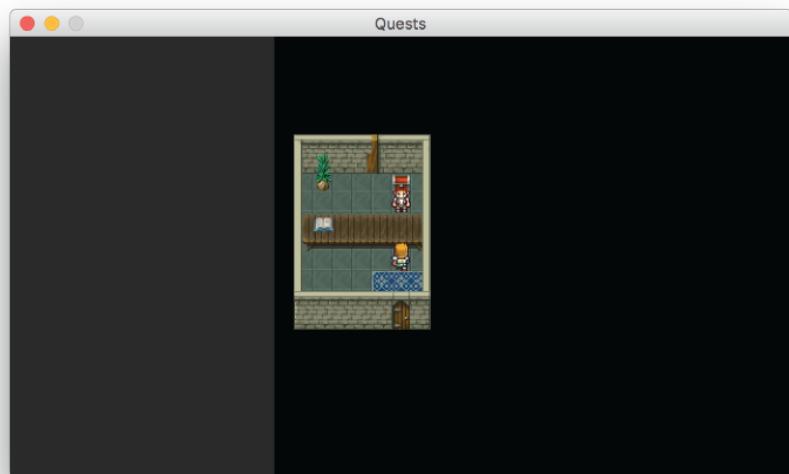


Figure 4.15: Trigger the item shop.

Example quests-5-solution contains the code so far.

The Inn and NPCs

The inn restores the party health for a small amount of gold. We use a trigger for the inn, but unlike the shops we won't create a new InnState. Instead, we use a dialog box.

There's a project quests-6 with the code so far, or you can continue using your own codebase.

Open Actions.lua and copy the code from Listing 4.29.

```
Actions =
{
    -- code omitted

    OpenInn = function(map, def)

        def = def or {}
        local cost = def.cost or 5
        local lackGPMsg = "You need %d gp to stay at the Inn."
        local askMsg = "Stay at the inn for %d gp?"
        local resultMsg = "HP/MP Restored!"

        askMsg = string.format(askMsg, cost)
        lackGPMsg = string.format(lackGPMsg, cost)

        local OnSelection = function(index, item)
            if index == 2 then
                return
            end

            gWorld.mGold = gWorld.mGold - cost
            gWorld.mParty:Rest()

            gStack:PushFit(gRenderer, 0, 0, resultMsg)
        end

        return function(trigger, entity, tX, tY, tLayer)

            local gp = gWorld.mGold

            if gp >= cost then
                gStack:PushFit(gRenderer, 0, 0, askMsg, false,
                {
                    choices =
                    {
                        options = {"Yes", "No"},

```

```

        OnSelection = OnSelection
    },
})
else
    gStack:PushFit(gRenderer, 0, 0, lackGPMsg)
end

end,
}

```

Listing 4.29: Adding an OpenInn action. In Actions.lua.

OpenInn is an action. It takes an optional inn definition table. The only thing the definition table contains is how much it costs to stay at the inn. The cost defaults to 5 gold pieces.

At the start of the OpenInn action, we store the cost in a variable called cost. This defaults to 5 gold pieces if there's no def, or no entry for cost in the def. Then we set up three messages: one if you can't afford the inn, one to ask if you'd like to stay, and one for after you've stayed. We'll use these messages to create different dialog boxes to show the player.

The first dialog box we show asks the player if they want to stay. We need to set up callbacks for the response. We do that in a local OnSelection function. The two choices are:

1. "Yes"
2. "No".

If the index is 2, we return immediately; the player doesn't want to stay at the Inn.

If the index is 1, the player wants to stay. The cost is subtracted from the party gold and a new Rest function is called. A new dialog box is created announcing that the party health and mana have been restored.

To bring up the inn in-game, we run the OpenInn action when the inn trigger is activated. The OpenInn action checks the party funds. If there isn't enough gold, the lackGPMsg is displayed using a dialog box.

Let's add the Rest function to the Party to easily restore the party health. Copy the code from Listing 4.30.

```

function Party:Rest()

    for _, v in pairs(self.mMembers) do

```

```

local stats = v.mStats
if stats:Get("hp_now") > 0 then
    stats:Set("hp_now", stats:Get("hp_max"))
    stats:Set("mp_now", stats:Get("mp_max"))
end
end

end

```

Listing 4.30: Rest function to restore the party's health and mana. In Party.lua.

The Rest function sets the HP and MP to the max values for each live party member. Dead party members do not come back to life by resting. If we had status effects and that kind of thing, these could also be cured here.

Now that we have the action, let's add a trigger to run it from the map. First we'll add a def to the map for the inn. Copy the code from Listing 4.31.

```

function CreateTownMap()

    -- code omitted

    local innDef =
    {
        cost = 5
    }

```

Listing 4.31: Definition table for the inn on the town map. In map_town.lua.

The inn def stores the cost for a night at the inn. Now let's add a trigger to map. Copy the code from Listing 4.32.

```

return {

    -- code omitted

    actions =
    {
        -- code omitted
        open_inn = { id = "OpenInn", params = { innDef } }
    },
    trigger_types =
    {
        -- code omitted

```

```

inn = { OnUse = "open_inn" },
},
triggers =
{
    -- code omitted
    { trigger = "inn", x = 35, y = 25 },
}

```

Listing 4.32: Adding a trigger for the inn to the map. In map_town.lua.

We define the OpenInn action with the innDef, then create a trigger and place it on the map. The placement position can be seen in Figure 4.16, it's on the tile containing the book.

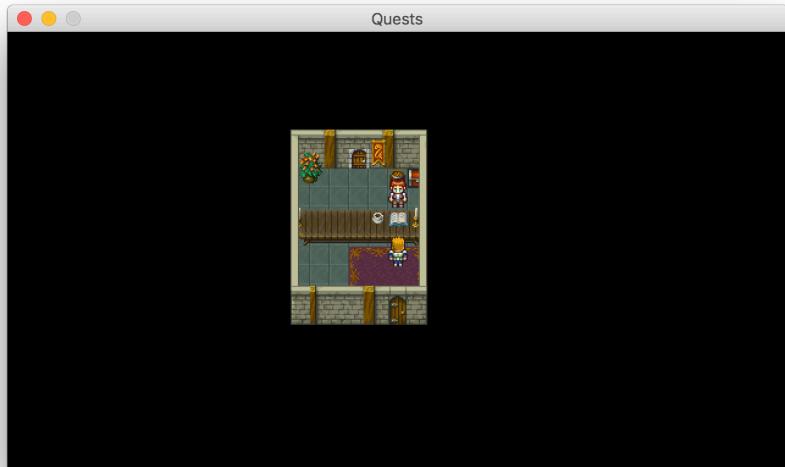


Figure 4.16: The inn and its trigger.

Run the game and you'll be able to visit the inn. (To test it properly alter the party gold and health.) Our town is getting there. It has shops and an inn, but it could still do with a little more life.

Let's add NPCs. Each NPC has its own dialog box which is managed by a new action called ShortText. This action displays a dialog box with text, above the trigger location. Open up the Actions.lua file and copy the code from Listing 4.33.

```

Actions =
{
    -- code omitted

    ShortText = function(map, text)
        return function(trigger, entity, tX, tY, tLayer)
            tY = tY - 4
            local x, y = map:TileToScreen(tX, tY)
            gStack:PushFix(gRenderer, x, y, 9*32, 2.5*32, text)
        end
    end
}

```

Listing 4.33: Adding a ShortText action. In Actions.lua.

The ShortText action creates a fixed size dialog box a few tiles above the trigger location. To help position the dialog box we use a new map function, TileToScreen. This function takes in an x, y tile coordinate and returns a screen space position. Copy the code from Listing 4.34.

```

function Map:TileToScreen(tX, tY)
    local x = -self.mCamX + self.mX + tX * self.mTileWidth
    local y = -self.mCamY + self.mY - tY * self.mTileHeight
    return x, y
end

```

Listing 4.34: Going from tile to screen. In Map.lua.

We'll use the ShortText action for NPC dialog, but it can be used for anything! We can add to items like a well ("This is the town well. Dry.") or a statue ("A statue of Harold the Great"), etc.

In the town map we add two ShortText actions as triggers on the NPCs. Copy the code from Listing 4.35.

```

actions =
{
    -- code omitted
    npc_1_talk = { id = "ShortText", params =
    {
        "The major was ill for a long time."
    }},
    npc_2_talk = { id = "ShortText", params =
    {
        "These stone columns are ancient. How old? No one knows."
    }}
}

```

```

        },
    },
    trigger_types =
    {
        -- code omitted
        npc_1 = { OnUse = "npc_1_talk" },
        npc_2 = { OnUse = "npc_2_talk" },
    },
    triggers =
    {
        -- code omitted
        { trigger = "npc_1", x = 47, y = 101 },
        { trigger = "npc_2", x = 35, y = 78 },
    }
}

```

Listing 4.35: Adding NPCs dialog to the town map. In map_town.lua.

Ok, we've added a little flavor text to the town map and perhaps a little foreshadowing. That's all we'll do for descriptive triggers. You can see them in action in Figure 4.17. You're welcome to take it further if you like!



Figure 4.17: Speaking to NPCs with the ShortText action.

To test the code, find the NPCs on the map and press Space while facing them. Example quest-6-solution contains the code so far.

The Major's Cutscene

The game starts with our party in the town hall. The major says a few words introducing the quest, and then control is given to the player. The party members all stand around the main hero during the cutscene. At the end of the cutscene they walk into the hero, joining the party.

This cutscene needs a new action, RemoveNPC. This action removes the party members from the map at the end of the cutscene. Copy the code from Listing 4.36.

```
Actions =
{
    -- code omitted

    RemoveNPC = function(map, npcId)
        return function(trigger, entity, tX, tY, tLayer)
            local char = map.mNPCbyId[npcId]

            local x = char.mEntity.mTileX
            local y = char.mEntity.mTileY
            local layer = char.mEntity.mLayer

            map:RemoveNPC(x, y, layer)
        end
    end,
}
```

Listing 4.36: Adding a new RemoveNPC action. In Actions.lua.

This action looks up the character and uses its position to remove it from the map. The action only needs an NPC's id to remove it.

The game starts in the major's office, so let's make sure that's the player's start position. Copy the code from Listing 4.37.

```
local startPos = Vector.Create(5, 9, 1)

local hero = Actor:Create(gPartyMemberDefs.hero)
local thief = Actor:Create(gPartyMemberDefs.thief)
local mage = Actor:Create(gPartyMemberDefs.mage)
gWorld.mParty:Add(hero)
gWorld.mParty:Add(thief)
gWorld.mParty:Add(mage)
gStack:Push(ExploreState:Create(gStack, CreateTownMap(), startPos))
gWorld.mGold = 500
```

Listing 4.37: Setting the player's start position. In Map.lua.

In Listing 4.37 we put the hero in front of the desk.

Our introduction cutscene is made in the same way we've made all the earlier cutscenes; we create the storyboard and push it onto the stack. Copy the code from Listing 4.38.

```
local sayDef = { textScale = 1.3 }
local intro =
{
    SOP.BlackScreen(),
    SOP.RunAction("AddNPC",
        {"handin", { def = "thief", id="thief", x = 4, y = 10}}, {GetMapRef}),
    SOP.RunAction("AddNPC",
        {"handin", { def = "mage", id="mage", x = 6, y = 11}}, {GetMapRef}),
    SOP.FadeOutScreen(),
    SOP.MoveNPC("major", "handin",
        {
            "right",
            "right",
            "left",
            "left",
            "down"
        }),
    SOP.Say("handin", "major", "So, in conclusion...", 2.3,sayDef),
    SOP.Wait(0.75),
    SOP.Say("handin", "major", "Head north to the mine.", 2.3,sayDef),
    SOP.Wait(2),
    SOP.Say("handin", "major", "Find the skull ruby.", 2.3,sayDef),
    SOP.Wait(2),
    SOP.Say("handin", "major", "Bring it back here to me.", 2.5, sayDef),
    SOP.Wait(1.75),
    SOP.Say("handin", "major",
        "Then I'll give you the second half of your fee.", 3.5, sayDef),
    SOP.Wait(1.75),
    SOP.Say("handin", "major", "Do we have an agreement?", 3.0, sayDef),
    SOP.Wait(1),
    SOP.Say("handin", "hero", "Yes.", 1.0, sayDef),
    SOP.Wait(0.5),
    SOP.Say("handin", "major", "Good.", 1.0, sayDef),
    SOP.Wait(0.5),
    SOP.Say("handin", "major",
        "Here's the first half of the fee...", 3.0, sayDef),
    SOP.Wait(1),
```

```

SOP.Say("handin", "major", "Now get going.", 2.5, sayDef),

-- Party members can walk into the hero and
-- return control to the player.
SOP.NoBlock(
    SOP.MoveNPC("thief", "handin",
    {
        "right",
        "up",
    })), SOP.FadeOutChar("handin", "thief"),
SOP.RunAction("RemoveNPC", {"handin", "thief"}, {GetMapRef}),
SOP.NoBlock(
    SOP.MoveNPC("mage", "handin",
    {
        "left",
        "up",
        "up",
    })), SOP.FadeOutChar("handin", "mage"),
SOP.RunAction("RemoveNPC", {"handin", "mage"}, {GetMapRef}),
SOP.Wait(0.1),
SOP.HandOff("handin")
}

local storyboard = Storyboard>Create(gStack, intro, true)
gStack:Push(storyboard)

```

Listing 4.38: Creating an intro cutscene. In main.lua.

There's nothing new in cutscene, but let's quickly run through it. The town map gets the id "handin". We start with a black screen that fades into the meeting in the major's office. While the screen is black, the thief and mage characters are added to the map behind the player. Then the major walks back and forth a little and we jump into the conversation. It's mainly the major talking but the hero does say "Yes" once, breaking the JRPG silent protagonist rule!

After the conversation, the thief and mage walk towards the player, fade out, and are removed from the map. The player is then free to control the hero and start playing the game.

Before moving on, let's add a trigger to let the player talk to the major. This isn't a simple dialog box trigger. The major says different things if the player has found the

gemstone. For now we'll only fill in the reply for when the player doesn't have the gem. Copy the code from Listing 4.39.

```
function CreateTownMap()

    -- code omitted

    local TalkToMajor =
    function (map, trigger, entity, x, y, layer)
        local message =
            "Go to the mine and get the gem. Then we can talk."
        local action = Actions.ShortText(map, message)
        action(trigger, entity, x, y, layer)
    end

    return
{
    -- code omitted
    action =
    {
        -- code omitted
        major_talk =
        {
            id = "RunScript",
            params = { TalkToMajor }
        }
    },
    trigger_types =
    {
        -- code omitted
        major = { OnUse = "major_talk" },
    },
    triggers =
    {
        -- code omitted
        { trigger = "major", x = 5, y = 5 },
    }
}
```

Listing 4.39: Add a talk trigger to the major. In map_town.lua.

A RunScript trigger is added to the major's position. When activated, the trigger calls the local TalkToMajor function. For now, this function displays a simple textbox that reminds the player where they need to go, as you can see in Figure 4.18.



Figure 4.18: The Major reminding the player about his quest.

This feels like the start of a full game! The code so far can be found in `quest-7-solution`. Run it and you'll see the starting quest.

The Big Wide World

To finish off the town map, we need triggers that let the player leave the town and enter the world map. To do this, the world map needs to exist.

Example `quest-8` contains the code so far, with map `"map_world.lua"` and tileset `"world_tileset.png"` loaded into the manifest and dependencies. It's important to include `"map_world.lua"` before `"MapDB.lua"` in the dependencies because `"MapDB.lua"` references it! Either switch to `quest-8` or copy the files into your own codebase.

Update the `MapDB` to look like Listing 4.40.

```
MapDB =
{
    ["town"] = CreateTownMap,
    ["world"] = CreateWorldMap
}
```

Listing 4.40: Adding the world map to the `MapDB`. In `MapDB.lua`.

To make testing the map easier, we'll stop the intro cutscene playing by commenting out the lines shown in Listing 4.41, in main.lua.

```
--local storyboard = Storyboard>Create(gStack, intro, true)
--gStack:Push(storyboard)
```

Listing 4.41: Temporarily stopping the intro cutscene. In main.lua.

We don't have a generic "change map" action, so let's add one. It'd be nice to do a quick fade out and fade in, so we'll use a storyboard to help us out. Copy the code from Listing 4.42.

```
Actions =
{
    -- code omitted

    ChangeMap = function(map, destinationId, dX, dY)

        local storyboard =
        {
            SOP.BlackScreen("blackscreen", 0),
            SOP.FadeInScreen("blackscreen", 0.5),
            SOP.ReplaceScene(
                "handin",
                {
                    map = destinationId,
                    focusX = dX,
                    focusY = dY,
                    hideHero = false
                }),
            SOP.FadeOutScreen("blackscreen", 0.5),
            SOP.Function(function() gWorld:UnlockInput() end),
            SOP.HandOff(destinationId)
        }

        return function(trigger, entity, tX, tY, tLayer)
            gWorld:LockInput()
            local storyboard = Storyboard>Create(gStack, storyboard, true)
            gStack:Push(storyboard)
        end
    end
}
```

Listing 4.42: Adding a new ChangeMap action. In Actions.lua.

The ChangeMap actions takes in four arguments:

- **map** - the current map which is passed in for every action
- **id** - the id of the map to load
- **dX and dY** - the destination tile position for the player on the new map

Calling the action unloads the current map, loads the new one, and moves the player to the specified position. To make this look nice, we use a storyboard. The storyboard adds a blackscreen with an alpha of zero, then fades it in, swap the maps, and fades the screen back out. This uses code we've seen before.

There are two new World functions here: LockInput and UnlockInput. We use these functions to lock off player control while the map transition occurs; we don't want the player setting off another trigger or bringing up a menu while we're moving between maps. Let's implement these functions in the world. Copy the code from Listing 4.43.

```
function World:Create()
    local this =
    {
        -- code omitted
        mLockInput = false,
    }
    setmetatable(this, self)
    return this
end

function World:IsInputLocked()
    return self.mLockInput
end

function World:LockInput()
    self.mLockInput = true
end

function World:UnlockInput()
    self.mLockInput = false
end
```

Listing 4.43: Adding input locks to world. In World.lua.

We need one more piece of code to lock off player input. Copy the code from Listing 4.44.

```

function ExploreState:HandleInput()

    if gWorld:IsInputLocked() then
        return
    end
    -- code omitted

```

Listing 4.44: Adding input lock to player. In ExploreState.lua.

With these code changes, we can disable the player's control while moving from map to map. When the transition action is triggered, the storyboard is pushed onto the stack and allowed to play out.

Let's add exit triggers on the town road. Copy the code from Listing 4.45.

```

actions =
{
    -- code omitted
    to_world_map_left = { id = "ChangeMap",
        params = { "world", 6, 26 } },
    to_world_map_right = { id = "ChangeMap",
        params = { "world", 8, 26 } }
},
trigger_type
{
    -- code omitted
    world_left = { OnEnter = "to_world_map_left" },
    world_right = { OnEnter = "to_world_map_right" },
},
triggers =
{
    -- code omitted
    { trigger = "world_left", x = 0, y = 105 },
    { trigger = "world_left", x = 0, y = 106 },
    { trigger = "world_left", x = 0, y = 107 },

    { trigger = "world_right", x = 59, y = 105 },
    { trigger = "world_right", x = 59, y = 106 },
    { trigger = "world_right", x = 59, y = 107 },
}

```

Listing 4.45: Adding town to world transition triggers. In map_town.lua.

We've added two types of triggers. Your destination on the world map changes depending on if you leave the left side or the right side of the map, as shown in Figure 4.19. This isn't a requirement, but it makes the game a little nicer.

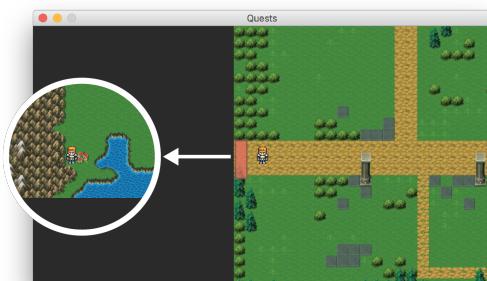
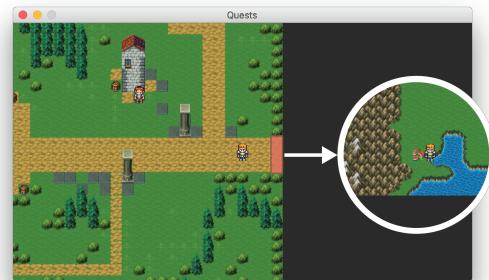


Figure 4.19: Leaving town using a different exit results in a different position on the world map.

You can leave the town - but there's no way to get back! Let's add a transition to the world map. Copy the code from Listing 4.46.

```
function CreateWorldMap()

    -- code omitted

    actions =
    {
        to_town = { id = "ChangeMap", params = { "town", 1, 106 } },
    },
    trigger_types =
    {
        enter_town = { OnEnter = "to_town" },
    },
    triggers =
    {
```

```
{ trigger = "enter_town", x = 7, y = 26 },  
},
```

Listing 4.46: Adding a world to town transition trigger. In map_world.lua.

That's it! We can now go and explore the world - exciting stuff! Run the game and you'll be able to leave the town, as well as revisit it. Note that no matter which direction you enter the town from, you'll always end up in the same place. We could write a more complicated trigger to fix this, but that's left as an exercise for the reader¹.

The complete code is available as quest-8-solution. Next we'll move on to the world map, random encounters, and entering the cave!

The World Map

The world map represents the world at a much larger scale than the town map. We saw the world map when we made the town, but in this section we'll make it functional. The two features we'll add to the world map are random encounters and a trigger for the cave. Let's start by implementing a random encounter system.

Random Encounters

The world of JRPGs is dangerous and teeming with monsters, who would like nothing more than to eat lost adventurers. As the party travels over plains, forests and rivers they'll be set upon by enemies. To model these encounters we use the combat state code we've already developed and used for the arena.

Combat is triggered randomly as the player travels through dangerous areas. Random encounters are easier to program than showing bands of enemies roaming around the map and, like so many aspects of RPGs, this mechanic comes from tabletop gaming. Knowing when to trigger combat is a bit of a balancing act; too many combat events and moving around becomes annoying, too few and it becomes boring.

There's no reason we couldn't have non-enemy world map events - finding treasure, travelling traders, or small set pieces, but in the book we'll only deal with traditional combat.

¹So there are a couple of ways to approach this. One is to store information about the tile the player has just left and the one he's entering and then query that. The other is to check which direction the player is facing when the trigger is activated and change the destination based on that. The ChangeMap action would need updating or a new action needs making that cares about the enter direction for a tile.

Using the Collision Tileset for Encounters

As a rule, any maps outside of a town or city in a JRPG have random encounters. We'll use the random encounter system for both the world map and the cave system.

Different areas of a map have different enemies. For instance, on the world map, in forests you might find goblins but on the plains you might meet wild animals.

We need a way to mark up our map tiles so different areas can have different encounter types. To do this we'll reuse the "collision" layer. The "collision" layer contains two colors, red for blocking and transparent for passable. We'll add more colored tiles to represent different encounter types.

In the quest-9-solution example folder, there's a new world map with a modified collision layer. We've extended the collision tileset as shown in Figure 4.20.

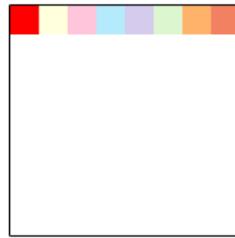


Figure 4.20: Encounter tileset.

The colored tiles in Figure 4.20 are associated with encounter rates stored in the map def. We won't worry about the exact encounters at this stage. Instead, we'll mark out the main areas where we'd like different encounters. The different encounter zones can be seen Figure 4.21.

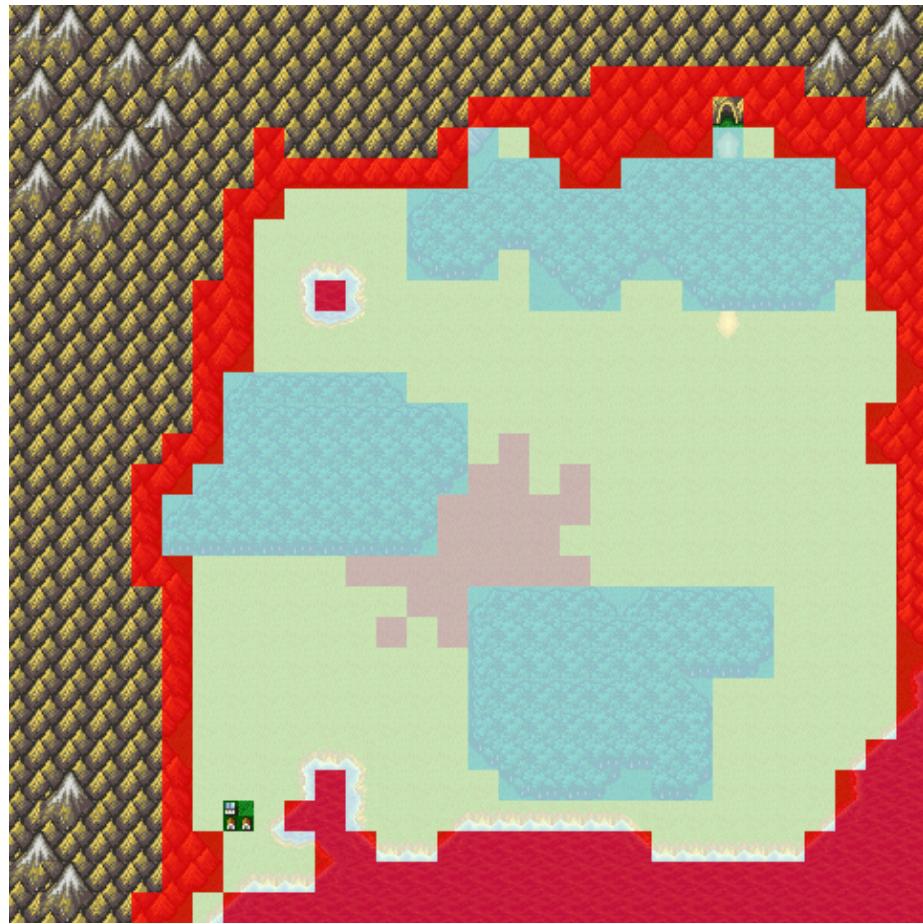


Figure 4.21: Encounter layer.

Figure 4.21 uses three colors to mark the different encounter types. The forest is blue, the plains are green, and the pass through the forest is orange. Note that we've used a checkerboard like pattern; this is a simple way to limit the encounter rate. It stops the player from hitting two rounds of combat in two subsequent moves, which is irritating when you're trying to get somewhere!

Adding Encounter Data to the Map

There's a project with the code thus far in `quests-10`. It has the new overworld map with the extended collision layer for encounters. It also has the following sprites:

- `combat_bg_forest.png`

- combat_bg_field.png
- goblin_field.png

Let's start by adding an encounter table where we'll define the encounter type for each colored tile. Copy the code from Listing 4.47.

```
function CreateWorldMap()

local encountersWorld =
{
    -- 1. yellow: plains
    {
        {
            oddment = 95,
            item = {}
        },
        {
            oddment = 5,
            item =
            {
                background = "combat_bg_field.png",
                enemy =
                {
                    "goblin_field",
                    "goblin_field",
                    "goblin_field",
                }
            }
        }
    },
    -- 2. blue: forest
    {
        {
            oddment = 90,
            item = {}
        },
        {
            oddment = 10,
            item =
            {
                background = "combat_bg_field.png",
                enemy =
                {
                    "goblin_forest",
                    "goblin_forest",
                }
            }
        }
    }
}
```

```

                "goblin_forest",
            }
        }
    },
-- 3. pink: forest pass
{
{
    oddment = 85,
    item = {}
},
{
    oddment = 6,
    item =
    {
        background = "combat_bg_field.png",
        enemy =
        {
            "goblin_field",
            "goblin_forest",
            "goblin_field",
        }
    }
},
{
    oddment = 6,
    item =
    {
        background = "combat_bg_forest.png",
        enemy =
        {
            "goblin_forest",
            "goblin_field",
            "goblin_forest",
        }
    }
}
}

```

Listing 4.47: The Encounter Tables. In map_world.lua.

In the overworld map def, we define the first three colors of the encounter tileset. Yellow is the first tile in the tileset, counting from the top left corner, so it's associated with the first entry in encountersWorld. This makes our setup code a little simpler.

We use each entry in the encounter table to set up an oddment table. The oddment values contain the list of enemies that attack the party. If the enemy table is empty, then the player doesn't have to fight anyone. Each entry contains a list of ids we use as keys to the EnemyDB table when creating the enemy actors. We haven't defined "forest_goblin" or "plains_goblin" yet, but we will shortly.

Every time a player moves to a new tile, we perform a calculation to see if combat is triggered. Before getting to the calculation, let's consider the average distance we'd like the player to walk before entering combat. Our map is quite small, and about 3 combat encounters on the way to the cave seems reasonable. The distance to the cave is about ~25 tiles, so we'd like the encounter rate to be about 3 in 25. Dividing 3 by 25 gives us an encounter rate of 0.12 or 12%. If each tile on the journey to the cave gives a 12% chance of combat, then we arrive at the correct encounter rate. We could simply make a single encounter type and set it to 12% but it's better to have a little variety.

The plains have the lowest encounter rate and contain the plains goblin, which is a weaker enemy type. The forest has a high encounter rate and contains forest goblins. The pass through the forest has the highest encounter rate and a mix of both goblin types. The chances are 5%, 10% and 15%, which averages out to a 30% encounter chance. We're only putting encounters on half the tiles, so the chance is more like 15%, which is close enough to our 12% target, even though the distributions are quite different.

Let's add the encounter data to the map def table. Copy the code from Listing 4.48.

```
return
{
    version = "1.1",
    luaversion = "5.1",
    orientation = "orthogonal",
    width = 30,
    height = 30,
    tilewidth = 16,
    tileheight = 16,
    properties = {},
    encounters = encountersWorld,
    -- code omitted
```

Listing 4.48: Adding encounter data into the map definition. In map_world.lua.

Each time the player moves onto a new tile, we check if there's a colored tile in the collision layer. If there is, we pick from the encounter oddment table. If the oddment table returns a nil, then the player's safe and may continue on. Otherwise there's a battle. Encounter data for the map is optional, so we don't need to update the town map!

Checking for Encounters

Let's add a list of oddment tables to the map class that we'll fill with encounter data. Copy the code from Listing 4.49.

```
Map = {}
Map.__index = Map
function Map:Create(mapDef)
    -- code omitted

    local encounterData = mapDef.encounters or {}
    this.mEncounters = {}
    for k, v in ipairs(encounterData) do
        local oddTable = OddmentTable:Create(v)
        this.mEncounters[k] = oddTable
    end

    setmetatable(this, self)
```

Listing 4.49: Adding an encounter oddment table in Map.lua.

The data for the `mEncounters` table comes from the def; if the data isn't present, it defaults to an empty table. For each encounter type we create an `OddmentTable` and add it to the `mEncounters` table.

Next let's add code to test for an encounter each time the player moves. Copy the code Listing 4.50.

```
function MoveState:Enter(data)

    -- code omitted

    if self.mMoveX ~= 0 or self.mMoveY ~= 0 then

        local x = self.mEntity.mTileX
        local y = self.mEntity.mTileY
        local layer = self.mEntity.mLayer

        local trigger = self.mMap:GetTrigger(x,y, layer)

        if trigger then
            trigger:OnExit(self.mEntity, x, y, layer)
        else
            self.mMap:TryEncounter(x, y, layer)
        end
    end
```

Listing 4.50: Move to Encounter. In MoveState.lua.

In Listing 4.50, when the player moves to a new tile, we try to get a trigger. If there's no trigger, then we test to see if there's an encounter. We don't want encounters to occur at the same time as a normal trigger; the player shouldn't have to fight a monster just as they move onto the safety of the town tile on the overworld map.

Let's implement the TryEncounter function for the map. Copy the code from Listing 4.51.

```
function Map:TryEncounter(x, y, layer)

    -- Get the tile id from the collision layer (layer +2)
    local tile = self:GetTile(x, y, layer + 2)
    if tile <= self.mBlockingTile then
        return
    end

    -- Get the index into the encounter table
    local index = tile - self.mBlockingTile
    local odd = self.mEncounters[index]

    if not odd then
        print("Encounter data missing!", index)
        return
    end

    local encounter = odd:Pick()

    -- Empty encounter
    if not next(encounter) then
        return
    end

    print("Encounter!", next(encounter))

end
```

Listing 4.51: Try to trigger an encounter by implementing TryEncounter. In Map.lua.

Each map tile is represented by a number. This number is a simple counter; the first tile in the first set is 1, the second tile is 2, and so on. The first collision tile id is stored in mBlockingTile. Any number greater than mBlockingTile is one of our encounter tiles.

Let's run through the TryEncounter function. First we grab the tile from the collision layer indexed by the x, y and layer values. If there's no tile set the id is 0, if it's the

collision tile it equals `mBlockTile`, and if it's an encounter tile it's a number greater than `mBlockingTile`.

If the tile is equal or less than the number stored in `mBlockingTile` we return immediately, no encounter today! Otherwise we subtract the `mBlockingTile` number from the tile id, which makes the first encounter tile equal 1 and the second equal to 2, etc.; perfect for indexing our encounter table.

With the new index we get the value held in the `mEncounters` table and store it in `odd`. If the encounter table is nil we return immediately. This should never happen, so the code prints a warning message. (It'd only occur if we had an encounter tile but no encounter data associated with the map.)

Assuming we successfully retrieved an oddment table we call the `Pick` function to find out which encounter the player will face.

We call `next(encounter)` to test for an empty table, in which case the player is safe, and there's no encounter. Otherwise we need to load the combat state. But wait! We don't have a good way to start combat yet, so for now we print an "Encounter!" message. To create a way to launch combat, we'll add a new Combat action.

A Combat Action

The combat action uses a def to create a combat state, fades the screen to black, and pushes the combat state onto the stack. Copy the code from Listing 4.52.

```
Actions =
{
    -- code omitted
    Combat = function(map, def)
        return function(trigger, entity, tX, tY, tLayer)

            def.background = def.background or "combat_bg_plains.png"
            def.enemy = def.enemy or { "grunt" }

            -- Convert id's to enemy actors
            local enemyList = {}
            for k, v in ipairs(def.enemy) do
                local enemyDef = gEnemyDefs[v]
                enemyList[k] = Actor:Create(enemyDef)
            end

            local combatState = CombatState:Create(gStack,
            {
                background = def.background,
                actors =
                {

```

```

        party = gWorld.mParty:ToArray(),
        enemy = enemyList,
    },
    canFlee = def.canFlee,
    OnWin = def.OnWin,
    OnDie = def.OnDie

})

local storyboard =
{
    SOP.BlackScreen("blackscreen", 0),
    SOP.FadeInScreen("blackscreen", 0.5),
    SOP.Function(
        function()
            gStack:Push(combatState)
        end)
}

local storyboard = Storyboard>Create(gStack, storyboard)
gStack:Push(storyboard)
end
end

```

Listing 4.52: Adding the Combat action. In Actions.lua.

The combat action def contains a background image and list of enemy ids. The ids are used with the gEnemyDefs table to create the enemy actors. If no data is provided, the background defaults to "combat_bg_plains" and the enemy list defaults to a single goblin.

When the combat action runs, it takes the list of enemy ids, looks up the defs, creates the actors, and adds them to the enemyList table. All this data is then passed into the CombatState constructor.

A storyboard is created to handle the transition into combat. The screen is set to black with a 0 alpha and then fades to a fully black screen. The combat state is then pushed on top, ready to start combat.

Finally the action pushes the storyboard on the global stack and ends.

Goblins!

Our overworld map is full of goblins. We already have one goblin type, but we're going to add two more. We'll add the goblin definitions to the EntityDB and gEnemyDefs tables. Copy the code from Listing 4.53.

```
gEntities =
{
    -- code omitted

    goblin_field =
    {
        texture = "goblin_field.png",
        startFrame = 1,
        width = 32,
        height = 32,
    },
    goblin_forest =
    {
        texture = "goblin_forest.png",
        startFrame = 1,
        width = 32,
        height = 32,
    },
    -- code omitted
}

-- code omitted

gCharacters =
{
    -- code omitted
    goblin_field =
    {
        entity = "goblin_field",
        controller =
        {
            "cs_move",
            "cs_run_anim",
            "cs_standby",
            "cs_die_enemy",
            "cs_hurt_enemy",
        },
        state = "cs_standby",
    },
    goblin_forest =
    {
        entity = "goblin_forest",
        controller =
        {
```

```

    "cs_move",
    "cs_run_anim",
    "cs_standby",
    "cs_die_enemy",
    "cs_hurt_enemy",
},
state = "cs_standby",
},
}

```

Listing 4.53: Adding goblins to . In EntityDB.lua.

The entity defs are quite simple. They point to the sprite and give the pixel dimensions. Copy the code from Listing 4.54 to define the enemy stats.

```

gEnemyDefs =
{
    -- code omitted

    goblin_field =
    {
        id = "goblin_field",
        stats =
        {
            ["hp_now"] = 75,
            ["hp_max"] = 75,
            ["mp_now"] = 0,
            ["mp_max"] = 0,
            ["strength"] = 12,
            ["speed"] = 8,
            ["intelligence"] = 8,
            ["counter"] = 0,
        },
        name = "Field Goblin",
        actions = { "attack" },
        steal_item = 10,
        drop =
        {
            xp = 50,
            gold = {5, 15},
            always = nil,
            chance =
            {
                { oddment = 1, item = { id = -1 } },
            }
        }
    }
}

```

```

        { oddment = 3, item = { id = 10 } }
    }
},
goblin_forest =
{
    id = "goblin_forest",
    stats =
    {
        ["hp_now"] = 100,
        ["hp_max"] = 100,
        ["mp_now"] = 0,
        ["mp_max"] = 0,
        ["strength"] = 11,
        ["speed"] = 9,
        ["intelligence"] = 9,
        ["counter"] = 0,
    },
    name = "Forest Goblin",
    actions = { "attack" },
    steal_item = 10,
    drop =
    {
        xp = 75,
        gold = {15, 20},
        always = nil,
        chance =
        {
            { oddment = 1, item = { id = -1 } },
            { oddment = 1, item = { id = 10 } }
        }
    }
}

```

Listing 4.54: Adding the enemy definitions. In EnemyDefs.lua.

The forest goblin is the tougher of the two types new of goblins.

Tying Encounters into the World Map

Our goblins are ready and our combat action is created. Let's add random encounters to the world!

To get random encounters working we'll update the TryEncounter function. Copy the code from Listing 4.55.

```

function Map:TryEncounter(x, y, layer)

    -- Code omitted

    if not next(encounter) then
        return
    end

    local action = Actions.Combat(self, encounter)
    action(nil, nil, x, y, layer)

end

```

Listing 4.55: Update TryEncounter to trigger combat actions. In Map.lua.

At the end the TryEncounter function, we now create a combat action and run it. That's all we need to make the encounter work. Try running around the world map and you'll quickly see it in action as shown in Figure 4.22.



Figure 4.22: A combat encounter in the forest near the cave.

Cave Trigger

Our map has two places of interest, the town and the cave. We already have a trigger for the town, so now we need to add one for the cave.

Example quest-10 contains two new files, `map_cave.lua` and `cave_tilesheet.png`, which are referenced in the dependencies and the manifest files. If you're using your own codebase, make sure to copy these files across.

Let's update the `MapDB.lua` with the cave map. Copy the code from Listing 4.56.

```

MapDB =
{
    ["town"] = CreateTownMap,
    ["world"] = CreateWorldMap,
    ["cave"] = CreateCaveMap,
}

```

Listing 4.56: Adding the cavemap. In MapDB.lua.

On the world map, the cave entrance is at X:23 Y:3. That's where we'll add the enter trigger. Copy the code from Listing 4.57.

```

return
{
    actions =
    {
        to_town = { id = "ChangeMap", params = { "town", 1, 106 } },
        to_cave = { id = "ChangeMap", params = { "cave", 12, 110 } },
    },
    trigger_types =
    {
        enter_town = { OnEnter = "to_town" },
        enter_cave = { OnEnter = "to_cave" },
    },
    triggers =
    {
        { trigger = "enter_town", x = 7, y = 26 },
        { trigger = "enter_cave", x = 23, y = 3 },
    },
}

```

Listing 4.57: Adding a cave trigger. In map_world.lua.

This trigger uses the same change map action we used for moving between the town and the world. The player can now travel over to the cave and enter it. In the next section, we'll set up the cave that the player is going to explore!

The Cave Map

The cave is our first dungeon. The major has tasked the party to venture deep inside the cave and return with a gemstone. Random encounters in the cave are a little more frequent than on the world map; it's a dangerous place! We'll be adding some treasure chests, a simple puzzle, and a boss fight triggered via a small cutscene.

The goal of the dungeon is to find and retrieve a gemstone. We'll update the basic cave map to add the gemstone on an altar at the end of the dungeon. No other changes need to be made to the map. We'll add everything using code.

Example quest-11 contains the code so far, as well as an updated cave map. The source file for the cave map is available in the original assets folder and exported version under maps/map_cave.lua.

This chapter introduces a bunch of new art to the project, listed below.

- cave_drake.png
- cave_bat.png
- cave_shade.png
- combat_bg_cave.png
- door.png
- sphere.png
- teleport.png

These are all in the manifest file and the art folder.

Adding Cave Monsters

To encounter monsters, we must first set up the entities, characters and enemy defs. Copy the code from Listing 4.58.

```
gEntities =
{
    -- code omitted

    cave_drake =
    {
        texture = "cave_drake.png",
        startFrame = 1,
        width = 128,
        height = 64,
    },
    cave_bat =
    {
        texture = "cave_bat.png",
        startFrame = 1,
        width = 32,
        height = 32
    },
    cave_shade =
    {
```

```
        texture = "cave_shade.png",
        startFrame = 1,
        width = 64,
        height = 64
    },
    -- code omitted
}

-- code omitted

gCharacters =
{
    -- code omitted
    cave_drake =
    {
        entity = "cave_drake",
        controller =
        {
            "cs_move",
            "cs_run_anim",
            "cs_standby",
            "cs_die_enemy",
            "cs_hurt_enemy",
        },
        state = "cs_standby",
    },
    cave_bat =
    {
        entity = "cave_bat",
        controller =
        {
            "cs_move",
            "cs_run_anim",
            "cs_standby",
            "cs_die_enemy",
            "cs_hurt_enemy",
        },
        state = "cs_standby",
    },
    cave_shade =
    {
        entity = "cave_shade",
        controller =
        {
```

```

    "cs_move",
    "cs_run_anim",
    "cs_standby",
    "cs_die_enemy",
    "cs_hurt_enemy",
},
state = "cs_standby",
}

```

Listing 4.58: Adding monster entity defs. In EntityDB.lua.

The creature setup is nothing we haven't seen before. Each creature has the standard combat states and is associated with an entity definition that defines an image to represent it. Next let's define the stats and drops for the enemies in gEnemyDefs. Copy the code from Listing 4.59.

```

gEnemyDefs =
{
    -- code omitted

    cave_drake =
    {
        id = "cave_drake",
        stats =
        {
            ["hp_now"] = 1000,
            ["hp_max"] = 1000,
            ["mp_now"] = 0,
            ["mp_max"] = 0,
            ["strength"] = 20,
            ["speed"] = 13,
            ["intelligence"] = 20,
            ["counter"] = 0,
        },
        name = "Cave Drake",
        actions = { "attack" },
        steal_item = 10,
        drop =
        {
            xp = 750,
            gold = 1000,
            always = nil,
            chance =
            {
                { oddment = 1, item = { id = -1 } },

```

```

        { oddment = 1, item = { id = 10 } }
    }
}
},
cave_bat =
{
    id = "cave_bat",
    stats =
    {
        ["hp_now"] = 100,
        ["hp_max"] = 100,
        ["strength"] = 9,
        ["speed"] = 16,
        ["intelligence"] = 3,
    },
    name = "Bat",
    actions = { "attack" },
    drop = { xp = 50 }
},
cave_shade =
{
    id = "cave_shade",
    stats =
    {
        ["hp_now"] = 200,
        ["hp_max"] = 200,
        ["strength"] = 20,
        ["speed"] = 5,
        ["intelligence"] = 5,
    },
    name = "Shade",
    actions = { "attack" },
    drop =
    {
        xp = 100,
        gold = {5, 50},
        chance =
        {
            { oddment = 3, item = { id = -1 } },
            { oddment = 1, item = { id = 12 } }
        }
    }
}

```

Listing 4.59: Defining the cave creatures. In EnemyDefs.lua.

The drake is the boss monster, so it has some pretty beefy stats and drops a nice amount of gold. The cave bat and shade are easier enemies that are encountered regularly as the player explores the cave. Bats don't drop anything but the shade has a chance to drop a mana potion.

Boss Fight

With the enemies defined, we're ready to add the cave-specific code. We'll start at the end of the cave and work our way back. The cave dungeon climaxes with a boss fight. This area of the map is shown in Figure 4.23.

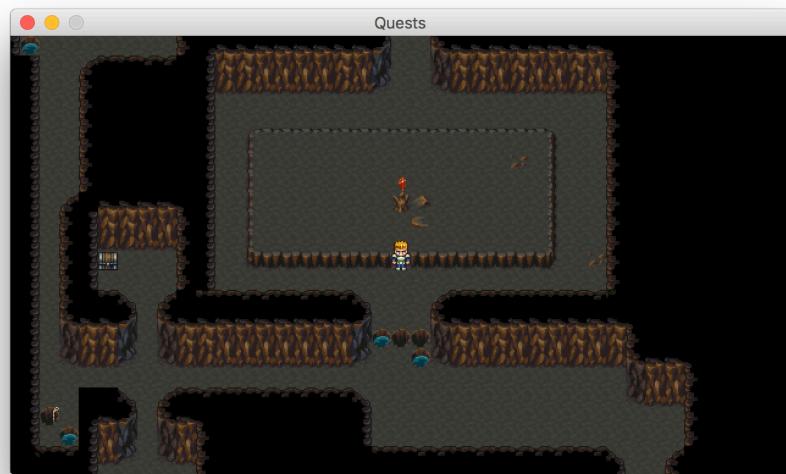


Figure 4.23: The area where the boss is encountered.

When the player enters the altar area, they see the gem they're searching for. In true Indiana Jones style, when the player reaches for the gem, things get more dangerous. We play a short cutscene, using a storyboard, and then launch into combat with the cave drake.

To make testing easier, let's update the main.lua file so the player spawns near the altar. Copy the code from Listing 4.60.

```
local startPos = Vector.Create(27, 26, 1)  
  
-- code omitted
```

```
gStack:Push(ExploreState>Create(gStack, CreateCaveMap(), startPos))
```

Listing 4.60: Setting the starting map as the cave and positioning the player in front of the altar.

To allow the player to pick up the gemstone, it needs an inventory definition. The gem item is important to the quest, so we'll make it a *key item*. Copy the definition from Listing 4.61.

```
ItemDB =
{
    -- code omitted
    {
        name = "Gemstone",
        type = "key",
        description = "Red gemstone shaped like a small skull."
    },
}
```

Listing 4.61: Adding the skull gemstone as a quest item. In ItemDB.lua.

The gem is added to the end of the item database, giving it an index of 14.

When we load the cave map, we use a little code to add the gemstone on the top of the altar. Every map has a wake_up table, which is a list of actions that's called when the map is loaded. We'll add a RunScript action to the wake_up table that calls a new function, InsertGem. The InsertGem function edits the tilemap to make the gem appear over the altar. Copy the code from Listing 4.62.

```
function CreateCaveMap()

    -- code omitted

    local InsertGem = function(map, trigger, entity, x, y, layer)

        local tileGemId = 214

        map:WriteTile
        {
            x = 27,
            y = 23,
            layer = 2,
            tile = 0,
            detail = tileGemId
    
```

```

    }

end

return
{
    version = "1.1",
    luaversion = "5.1",
    orientation = "orthogonal",
    width = 60,
    height = 120,
    tilewidth = 16,
    tileheight = 16,
    properties = {},
    on_wake =
    {
        {
            id = "RunScript",
            params = { InsertGem }
        }
    },
}

```

Listing 4.62: Inserting the gem into the cave map. in map_cave.lua.

The sprite for the gem is contained in the cavemap_tileset.png file. The gem is the same as any other tile in the tile map. The gem tile number is 214. We write the gem tile to the detail section of the second layer in the cave map, just above the altar.

Putting the gem on the second layer means the player can walk behind it. The wake_up table calls the InsertGem function when the map loads.

So far so good. When the map is loaded, the gem tile is inserted over the altar. Note that this happens **every time** the map is loaded. If we enter the map, take the gem, leave the map, and come back - there will be a new gem! That's not good! We'll fix this issue a little later, but first let's get the boss fight working.

To trigger the boss fight, the player must pick up the gem. Taking the gem is a one-off event and we use a RunScript action to run a new TakeGem function. Copy the code from Listing 4.63.

```

function CreateCaveMap()

    local gemstoneId = 14

    -- code omitted

```

```

local TakeGem = function(map, trigger, entity, x, y, layer)

    -- Clear the gem from the deco layer
    local tiles = map.mMapDef.layers[5].data
    tiles[map:CoordToIndex(x, y - 1)] = 0
    map:RemoveTrigger(x, y, layer)
    gWorld:AddKey(gemstoneId)

    local combatDef =
    {
        background = "combat_bg_cave.png",
        enemy = { "cave_drake" },
        canFlee = false
    }

    local sayDef = { textScale = 1.5 }
    local bossTalk =
    {
        SOP.Function(function() gWorld:LockInput() end),
        SOP.Say("handin", "hero", "Gemstone received.", 1, sayDef),
        SOP.Wait(0.3),
        SOP.Say("handin", "hero", "Trespass in MY house?", 3, sayDef),
        SOP.Wait(0.2),
        SOP.Say("handin", "hero", "KILL MY SERVANTS?", 3, sayDef),
        SOP.Wait(0.2),
        SOP.Say("handin", "hero", "STEAL?", 3, sayDef),
        SOP.Wait(0.2),
        SOP.Say("handin", "hero", "You must be punished!", 3, sayDef),
        SOP.Function(function() gWorld:UnlockInput() end),
        SOP.RunAction("Combat", { map, combatDef }),
        SOP.HandOff("handin"),
    }

    local storyboard = Storyboard>Create(gStack, bossTalk, true)
    gStack:Push(storyboard)

end

-- code omitted

return
{
    -- code omitted
    actions =
    {

```

```

take_gem = { id = "RunScript", params = { TakeGem } },
},
trigger_types =
{
    use_altar = { OnUse = "take_gem" }
},
triggers =
{
    { trigger = "use_altar", x = 27, y = 24 },
}

```

Listing 4.63: Boss Encounter function. In map_cave.lua.

The code in Listing 4.63 places a trigger on the altar that calls TakeGem when the player tries to use it. At the top of the file we define the gemstoneId variable to store the id of the gem in the item database.

The TakeGem function clears the gem tile from the map by setting the detail section of the second layer to zero. It then removes the trigger so the gem can't be taken twice. We use the gemstoneId value to add the gem to the player inventory.

The player now has the gem, but they're not going to be able to just walk away! After the player takes the gem, we play a cutscene and load a combat state.

We start by defining the combat event. This is a boss fight, so we don't allow fleeing. The only enemy in the battle is the cave drake we defined earlier. Before combat starts, we use some text boxes to threaten the player. Then we enter combat. The start of the cutscene can be seen in Figure 4.24. During the cutscene we lock the player input; otherwise it would be possible for the player to dismiss the textboxes and walk away!



Figure 4.24: The start of the cutscene that fires when the player takes the gem.

It might be educational to see exactly what's happening with the stack here. At the bottom of the stack is the ExploreState for the cave map. On top of that is storyboard. When the fight starts, the storyboard pushes a combat state on the top of itself. When the combat finishes, it pops itself off the stack, returning to the cutscene. The cutscene then ends and pop itself off the stack too and we're back to the ExploreState. You can see this visualized in Figure 4.25.

Explore State



Storyboard



Dialog Box



Combat State



Storyboard



Explore State



Figure 4.25: The state flow for the boss battle.

After the player defeats the boss, we don't want to force them to walk all the way back through the dungeon. Therefore we spawn in a teleporter that teleports the player back to the cave entrance. We add the teleporter using a trigger. Around the altar we add a number of triggers that fire when the player *exits* the tile. This trigger is called LeaveAltarArea. Copy the code from Listing 4.64.

```
function CreateCaveMap()

    -- code omitted

    local LeaveAltarArea = function(map, trigger, entity, x, y, layer)

        local tileTeleportId = 198
        local tileDirtId = 104

        -- Do you have gem?
        if not gWorld:HasKey(gemstoneId) then
            return
        end

        -- Is the teleporter added?
        local trigger = map:GetTrigger(27, 28, 1)
        if trigger then
            return
        end

        map:WriteTile
        {
            x = 27,
            y = 28,
            layer = 1,
            tile = tileDirtId,
            detail = tileTeleportId
        }

        local trigger = Trigger:Create
        {
            OnEnter = function()
                entity:SetTilePos(12, 111, 1, map)
            end
        }
    end
}
```

```

map:AddFullTrigger(trigger, 27, 28, 1)

end

return
{
    actions =
    {
        -- code omitted
        leave_with_gem =
        {
            id = "RunScript",
            params = { LeaveAltarArea }
        },
    },
    trigger_types =
    {
        -- code omitted
        leave_altar = { OnExit = "leave_with_gem" },
    },
    triggers =
    {
        -- code omitted
        { trigger = "leave_altar", x = 27, y = 23 },
        { trigger = "leave_altar", x = 28, y = 24 },
        { trigger = "leave_altar", x = 26, y = 24 },
        { trigger = "leave_altar", x = 27, y = 25 },
    },
}

```

Listing 4.64: LeaveAltarArea function. In cave_map.lua.

The LeaveAltarArea function only runs under certain conditions:

1. The gem is in the party inventory.
2. Tile x:27 y:28 does **not** contain a trigger.

The gem inventory check ensures that the trigger doesn't fire if the player walks up to the altar, doesn't take the gem, and then walks away. The trigger location check ensures that once the teleporter has been added to the map, we don't try to add it again.

Assuming the conditions are met, the function creates a teleporter nearby. The teleporter is made of two parts, a special tile written to the detail layer and a trigger for a teleport action. When the player steps onto the teleporter, it teleports them to the start of the dungeon as is shown in Figure 4.26.



Figure 4.26: Using the teleporter to return to the start of the dungeon.

That's the boss done. Try it out! You can steal the gem, kill the boss, and teleport back to the start of the dungeon. You may need to change some stats or boost your party because at level 0 killing the drake is *challenging*!

Leaving and Looting

After the boss battle, the party can teleport back to the start of the caves but they can't exit! Let's solve that by adding a transition back to the world map. While we're tweaking the triggers again, we're going to add a few chests with some loot to the cave. Copy the code from Listing 4.65.

```
function CreateCaveMap()

    -- code omitted

    return
{
    id = id,
    name = "Cave",

    -- code omitted

    on_wake =
    {
        -- code omitted
        {
            id = "AddChest",
            params = { "chest", { { id = 10 } }, 39, 92 },
        },
        {
            id = "AddChest",

```

```

        params = { "chest", { { id = 10 } }, 12, 27 },
    },

-- code omitted
},
actions =
{
    -- code omitted
    to_world_map = { id = "ChangeMap", params = { "world", 23, 4 } },
},
trigger_types =
{
    -- code omitted
    leave_cave = { OnEnter = "to_world_map" },
},
triggers =
{
    -- code omitted

    { trigger = "leave_cave", x = 11, y = 109 },
    { trigger = "leave_cave", x = 12, y = 109 },
    { trigger = "leave_cave", x = 13, y = 109 },
}
,
```

Listing 4.65: An exit trigger and some loot. In cave_map.lua.

In Listing 4.65 we add a couple of chests, each containing a heal potion. We also define a trigger that returns the player to the world map, and we place a number of these triggers in front of the exit. The chest and trigger locations can be seen in Figure 4.27.

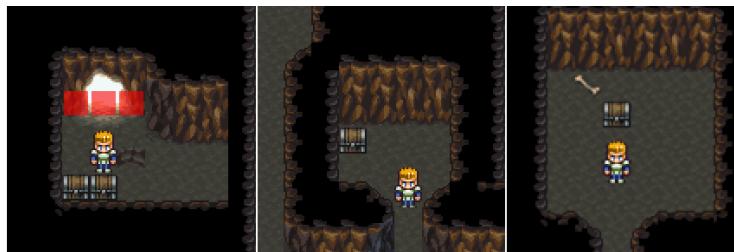


Figure 4.27: The chests and triggers to exit the cave. The triggers are marked in red.

Door Puzzle

What's a dungeon without a puzzle? In the cave we'll implement a simple puzzle. It's a lock and key puzzle. There is a locked door. The player must find a "key" and place it on the tile next to the door.

The puzzle provides few text hints. Instead, it relies on the player being able to read the environment. To get to the locked door, the player first passes through an open doorway. In front of that door are two golden spheres resting on top of two pressure plate tiles. The next room has a similar layout but there's only one sphere. One of the pressure plates is empty and the door is closed. The player must go down a side passage, find the missing sphere, and bring it to the open pressure plate. When the player places it on the plate, the door opens, allowing the player to proceed. This is a nice simple puzzle for a first dungeon. You can see the layout in Figure 4.28.



Figure 4.28: The layout of the puzzle.

A lock and key puzzle refers to any kind of puzzle that requires a specific item to remove something blocking the player's progress. The best key puzzles show the lock before the player finds the key. In our case, the side corridor is partially obscured and initially looks like it can't be entered. By the time the player approaches the entrance of the side corridor, they can see the locked door.

That's the high-level view. Example quests-12 contains a fresh project for implementing the puzzle, or you can continue with your own codebase.

In the cave map file, let's define a `SetupDoorPuzzle` function that gets called from the `on_wake` table when the map loads.

The SetupDoorPuzzle function loads in the door entities, tiles, golden spheres, and triggers required for the player to solve the puzzle. First let's add entity definitions for the door and spheres. Copy the definitions from Listing 4.66.

```
gEntities =
{
    -- code omitted
    door_left =
    {
        texture = "door.png",
        width = 16,
        height = 32,
        startFrame = 1,
        frames = {1, 3, 5, 7}
    },
    door_right =
    {
        texture = "door.png",
        width = 16,
        height = 32,
        startFrame = 2,
        frames = {2, 4, 6, 8}
    },
    sphere =
    {
        texture = "sphere.png",
        width = 16,
        height = 16,
        startFrame = 1,
    },
}
```

Listing 4.66: Adding the doors and sphere to the entity defs. In EntityDefs.lua.

In Listing 4.66 we've added all the entities we need for the puzzle.

To open the locked door, the player needs to pick up the key-sphere. Let's add it as an item to the item database. Copy the code from Listing 4.67.

```
ItemDB =
{
    -- code omitted
    {
        name = "Keystone Orb",
        type = "key",
        description = "A heavy stone orb."
```

```
    }
}
```

Listing 4.67: Adding the keystone to the item database. In ItemDB.lua.

The keystone orb is a key item, and we only ever have one. Items are given an id according to their position in the database. Our keystone has the id 15.

Now that the item exists, we can pick it up and add it to the inventory.

Let's add the SetupDoorPuzzle to the map. Copy the code from Listing 4.68.

```
function CreateCaveMap()

    local gemstoneId = 14
    local keystoneId = 15

    -- code omitted

    local SetupDoorPuzzle = function(map, trigger, entity, x, y, layer)
        local tileDoorPlateId = 182

        local tileLocations =
        {
            { 39, 56, true},
            { 42, 56, true},
            { 38, 41, true},
            { 41, 41, false}
        }
        local keyX = 56
        local keyY = 40

        for k, v in ipairs(tileLocations) do
            local x = v[1]
            local y = v[2]
            local isFilled = v[3]

            map:WriteTile
            {
                x = x,
                y = y,
                layer = 1,
                tile = tileDoorPlateId,
                collision = true
            }
        end
    end
}
```

```

        if isFilled then
            local sphere = Entity:Create(gEntities['sphere'])
            sphere:SetTilePos(x, y, 1, map)
        else
            local trigger = Trigger:Create
            {
                OnUse = function(...)
                    DoorPlateInteract(map, ...)
                end
            }
            map:AddFullTrigger(trigger, x, y, 1)
        end
    end

    if not gWorld:HasKey(keystoneId) then
        -- Add the key sphere
        local key = Entity:Create(gEntities['sphere'])
        key:SetTilePos(keyX, keyY, 1, map)

        -- We need to add a trigger for the keystone
        local keyTrigger = Trigger:Create
        {
            OnUse = function(...)
                KeyStoneInteract(map, ...)
            end
        }
        map:AddFullTrigger(keyTrigger, keyX, keyY, 1)
    end

    local doorLeft = Entity:Create(gEntities['door_left'])
    local doorRight = Entity:Create(gEntities['door_right'])
    doorLeft:SetTilePos(39, 39, 1, map)
    doorRight:SetTilePos(40, 39, 1, map)
end

return
{
    -- code omitted
    on_wake =
    {
        {
            id = "RunScript",
            params = { SetupDoorPuzzle }
        },
    }
}

```

Listing 4.68: Adding the SetupDoorPuzzle function. In map_cave.lua.

This is a surprisingly fat function. Let's go through it and see what's going on.

At the top of the CreateCaveMap function, we store the id for keystone item. The keystone is the item the player needs to open the door. We use the id to add and remove the item as the player navigates the puzzle.

The SetupDoorPuzzle function adds the doors and other puzzle entities to the map. The tileDoorPlateId is an id in the cave tileset that refers to a pressure plate. This tile helps identify the location where the orb should be placed.

We record the position for the missing orb in the variables keyX and keyY which places it at the end of a nearby corridor.

There are two doorways in the game. In front of each doorway are a pair of pressure plates. The information about the pressure plate locations is stored in the table tileLocations. Each entry in the tileLocations contains 3 pieces of data: the x and y position for the pressure plate and a boolean indicating if it has an orb on top of it.

For each entry in the tileLocations table, we write the pressure plate tile onto the map, with collision set to true. If the third element is true we also add an orb on top of the pressure plate. The orb is created using the "sphere" entity def we added earlier.

If a pressure plate is missing an orb, we add an OnUse trigger onto it. The OnUse trigger calls the DoorPlateInteract function, which gives the player the ability to place an orb at that location. If there's no orb in the party inventory, we display a hint instead.

After the pressure plates have been added into the world, we're ready to add the missing orb. We only add the orb if it's not already in the player inventory. This means the player won't find a duplicate orb if they pick it up, leave the map, and then return later! (Maybe now you can guess why many RPGs use pushing puzzles that reset every time you re-enter the map!)

Assuming the player hasn't already picked up the orb, we create a new sphere entity and insert it into the map at the orb location. To allow the player to interact with the sphere, we add a trigger that calls the KeyStoneInteract function. We haven't written KeyStoneInteract yet, but it lets the player pick up the orb. The missing orb's location is shown in Figure 4.29.

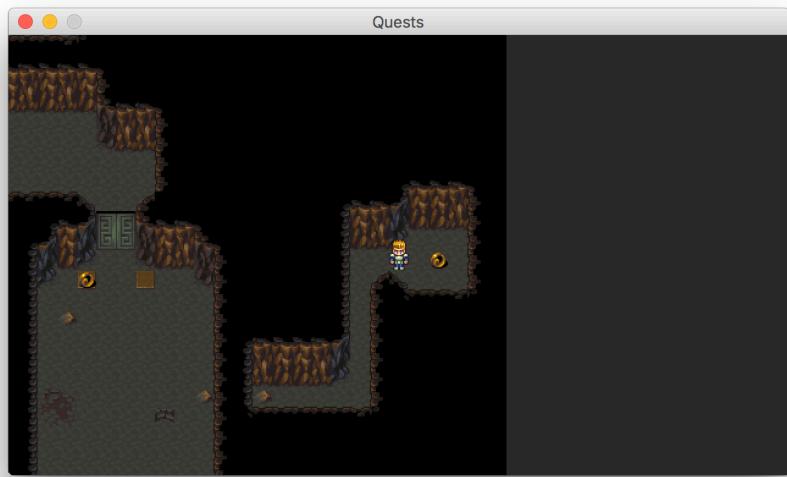


Figure 4.29: The location of the missing orb.

The final part of the function creates two door entities to block the second passageway.

Before we implement the two interaction functions, let's take a detour to the StoryboardEvents.lua file. When the player finally places the keystone on the pressure plate, we want the doors to slide open, and this requires an animation! Storyboards don't handle animations yet. Let's change that. Copy the code from Listing 4.69.

```
AnimateEvent = {}
AnimateEvent.__index = AnimateEvent
function AnimateEvent>Create(entity, anim)
    local this =
    {
        mEntity = entity,
        mAnimation = anim,
    }
    setmetatable(this, self)
    return this
end

function AnimateEvent:Update(dt)
    self.mAnimation:Update(dt)
    local frame = self.mAnimation:Frame()
    self.mEntity:SetFrame(frame)
```

```

end

function AnimateEvent:Render() end

function AnimateEvent:IsBlocking()
    return true
end

function AnimateEvent:IsFinished()
    return self.mAnimation:IsFinished()
end

-- code omitted

function SOP.Animate(entity, params)
    return function(storyboard)
        local animation = Animation>Create(unpack(params))
        return AnimateEvent>Create(entity, animation)
    end
end

```

Listing 4.69: Adding a storyboard event to play animations on an entity. In `StoryboardEvents.lua`.

The AnimateEvent is a StoryboardEvent that blocks the storyboard until an animation on an entity has finished. There's also a storyboard operation Animate to make it easy to use as a storyboard operation. The Animate operation takes in an entity for the animation to act upon and a table of parameters to create the animation. With this we can open the door! Let's jump back to the cave map and finish off the interaction functions. Copy the code from Listing 4.70.

```

function CreateCaveMap()

    -- code omitted

    local KeyStoneInteract = function(map, trigger, entity, x, y, layer)
        layer = layer or 1

        local entity = map:GetEntity(x, y, layer)
        map:RemoveEntity(entity)
        map:RemoveTrigger(x, y, layer)
        gWorld:AddKey(keystoneId)
        gStack:PushFit(gRenderer, 0, 0, "Keystone received.")
    end

```

Listing 4.70: The KeyStoneInteraction function. In map_cave.lua.

The keystone interact function is pretty simple. It gets the entity at the trigger location, removes the entity, removes the trigger, and adds the keystone item to the player inventory.

Next let's add the DoorPlateInteract function. This one is a little more complicated, as it needs to animate the doors. Copy the code Listing 4.71.

```
function CreateCaveMap()

    -- code omitted
    local DoorPlateInteract = function(map, trigger, entity, x, y, layer)
        layer = layer or 1

        if gWorld:HasKey(keystoneId) then

            map:RemoveTrigger(x, y, layer)
            gWorld:RemoveKey(keystoneId)
            local sphere = Entity>Create(gEntities['sphere'])
            sphere:SetTilePos(x, y, layer, map)

            local leftDoor = map:GetEntity(39, 39, 1)
            local rightDoor = map:GetEntity(40, 39, 1)
            local leftAnim =
            {
                gEntities['door_left'].frames,
                false
            }
            local rightAnim =
            {
                gEntities['door_right'].frames,
                false
            }

            local sayDef = { textScale = 1.5 }
            local storyboard =
            {
                SOP.Say("handin", "hero", "Keystone removed.", 1, sayDef),
                SOP.NoBlock(SOP.Animate(leftDoor, leftAnim)),
                SOP.Animate(rightDoor, rightAnim),
                SOP.Function(
                    function()
                        map:RemoveEntity(leftDoor)
                        map:RemoveEntity(rightDoor)
                        caveState.completed_puzzle = true
                )
            }
        end
    end
end
```

```

        end),
        SOP.HandOff("handin")
    }

    gStack:Push(Storyboard:Create(gStack, storyboard, true))
else
    local sayX, sayY = map:TileToScreen(x, y - 1)
    gStack:PushFit(gRenderer, sayX, sayY,
                    "Looks like something goes here...")
end
end

```

Listing 4.71: The DoorPlateInteract function. In map_cave.lua.

The DoorPlateInteract function fires when the player uses the empty door plate. The first test is if the player is carrying the keystone. If not, we display the hint “Looks like something goes here...” and then exit. Hopefully with this hint, the player will go grab the stone.

If the player has the keystone, the trigger on the door plate is cleared, the keystone is removed from the inventory, and a sphere entity is added on top of the door plate. Then comes the door animation. We get both the left and right door entities and set up an animation for each, using the frame list from their entity def. Then we create a storyboard to animate the doors opening. We want both doors to open at the same time, so the left door animation is told not to block while the second operation blocks as it animates the right. These two events animate the door opening frame by frame. Once the animations are finished, the door entities are totally removed, so the player can pass. We push the storyboard onto the stack and it plays immediately.

That’s the door puzzle done! The code so far is available in quests-12-solution. You may have caught the issue that if the player opens the doors, leaves the map, and returns, the doors will be shut again. This problem occurs because we don’t track the state of the game. We’ll start tracking state soon, but next we’re going to add random encounters to the cave.

Random Encounter Setup

We’ve already written the random encounter code for the world map. Now we’re going to reuse it to add encounters to the cave.

In the quests-13 project, we’ve updated the map_cave.lua collision layer with the new encounter data. The original asset folder also contains the updated .tmx file. The updated map is shown in Figure 4.30.

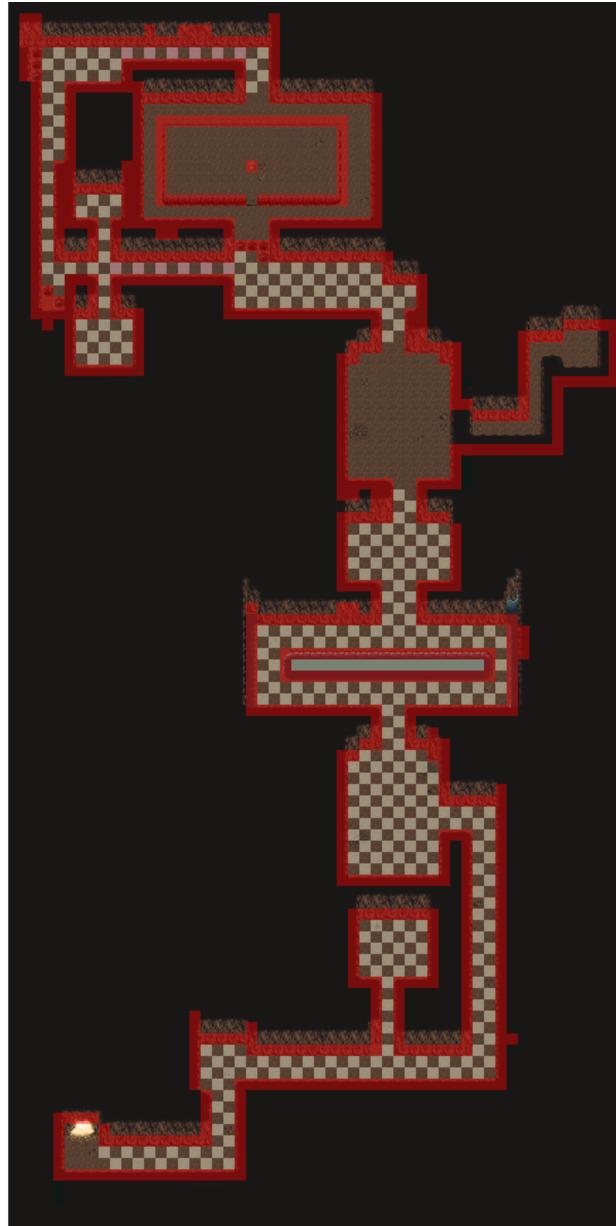


Figure 4.30: The collision and encounter data for the cavemap.

There are only two encounter colors used in the cave map, but it's mainly all pale yellow. A few corridors near the end are red. We'll be making the red areas a little more challenging, but otherwise the cave is all of a uniform difficulty.

More interesting perhaps is that there are a few areas in the cave with no encounter tiles: the areas near the door, the puzzle, and the boss. This is a personal preference, but I think if there's a puzzle it's better not to be randomly attacked while trying to figure it out! When trying to escape the cave, I also wanted to give the player a little bit of safety. Before the boss I wanted no encounters so the build-up isn't interrupted.

Now that we have the encounter tiles, we need to define the encounter table at the top of map_cave.lua. Copy the code from Listing 4.72.

```
function CreateCaveMap()

    -- code omitted

    local encountersCave =
    {
        -- 0. Yellow, nearly everywhere
        {
            { oddment = 92, item = {} },
            {
                oddment = 4,
                item =
                {
                    background = "combat_bg_cave.png",
                    enemy =
                    {
                        "cave_bat",
                        "cave_bat",
                        "cave_bat",
                    }
                }
            },
            {
                oddment = 2,
                item =
                {
                    background = "combat_bg_cave.png",
                    enemy =
                    {
                        "cave_bat",
                        "cave_shade",
                        "cave_bat",
                    }
                }
            },
            {
                oddment = 1,
            }
        }
    }
}
```

```

item =
{
    background = "combat_bg_cave.png",
    enemy =
    {
        "cave_bat",
        "cave_bat",
        "cave_bat",
        "cave_bat",
        "cave_bat",
    }
},
-- 2. Pink, corridors of death
{
    { oddment = 95, item = {} },
    {
        oddment = 5,
        item =
        {
            background = "combat_bg_field.png",
            enemy =
            {
                "cave_shade",
                "cave_shade",
                "cave_shade",
            }
        }
    }
}
}

```

Listing 4.72: Making the encounter tables for the caves. In map_cave.lua.

The first set of encounter data is represented by the yellow color, and is usually 3 bats, sometimes 4 bats, sometimes 2 bats and a shade. In the pink corridors you'll always encounter 3 shades.

We need to hook this into the map data structure, which is shown in Listing 4.73.

```

function CreateCaveMap()

    -- code omitted

    return

```

```
{
    version = "1.1",
    luaversion = "5.1",
    orientation = "orthogonal",
    width = 60,
    height = 120,
    tilewidth = 16,
    tileheight = 16,
    properties = {},
    encounters = encountersCave,
```

Listing 4.73: Hooking the encounter data into the map def. In map_cave.lua.

Now we can wander around the cave and battle with the things found there! That's all there is to do for the cave until we start to track the game state. We've done well; the player can enter, battle their way to the door puzzle, solve the puzzle, find the gem, and defeat the boss. Next we'll implement the return to the major.

Game State

The player changes the game world as they play the game; they complete puzzles, kill enemies and collect loot. We need to track these changes. Tracking game state makes it easier to build quests and write the save / load system.

In this section we'll start tracking important game data, improve the quest logic using the game state, and implement a save / load system.

Let's consider an imaginary game state as a Lua table, shown in Listing 4.74.

```
state =
{
    current_act = 1,
    acts =
    {
        {
            parents_dead = false,
            talked_to_alfred = false,
            fallen_in_well = false,
            maps =
            {
                ['manor'] =
                {
                    used_chest_1 = false,
                    used_chest_2 = false,
```

```

        },
        ['garden'] =
        {
            picked_up_rusty_key = false,
        }
    }
},
{
    times_trained = 0,
    discovered_joker_clue_1 = false,
    discovered_joker_clue_2 = false,
    discovered_joker_clue_3 = false,
    had_suit_made = false,
    maps =
    {
        ... state important to maps in act II
    }
},
{
    crimes_stopped = 0,
    visited_crime_scene_1 = false,
    visited_asylum = false,
    maps =
    {
        ... state important to maps in act III
    }
}
}
}

```

Listing 4.74: An example game state.

Listing 4.74 is an example of a game state based on a popular superhero. The player starts at Act 1, and as they play the game the state changes. The game actors and puzzles refer to the game state to determine if the player can progress.

In Act 1, when we enter the hero's manor and talk to the butler, he'll comment how it's a nice day, but if the parents_dead flag is true, he instead comforts the player. If we're in Act 2, the manor might load a slightly different map, the hero's bedroom will have matured, the butler will comment on a villain called the Joker terrorizing the city, etc.

The game state table stores important game variables and applies the player's changes to the world. When we load a map, we also load the changes the player has made to the map. In JRPGs, the player has limited powers to change the maps. They can open chests, solve puzzles and kill bosses. If the player opens a chest, they shouldn't ever

be able to open it again. This is a bit of game state that needs recording. When a map is loaded, the chest checks the game state and sets its state to *closed* or *looted*.

Later in this section, when we want to save and load games, we'll save and load the game state table too. This is how we record the player's progress.

The game we're making is relatively simple, so the game state will also be relatively simple. So far we've used key items to help track state, e.g. if the player has the keystone we know the door puzzle isn't solved and also that the cavemap shouldn't put the keystone on the ground.

Let's consider a sketch of the gamestate for our game, as shown in Listing 4.75.

```
state =
{
    defeated_cave_drake = false,
    -- nothing to track in world or town
    maps =
    {
        town = {},
        world = {},
        cave =
        {
            completed_puzzle = false,
            chests_looted =
            {
                [1] = false,
                [2] = false
            }
        }
    }
}
```

Listing 4.75: The game state for our example game.

The game state stores global important information as well as local information for each map. The cave map tracks if the puzzle has been completed and also contains a table to track which chests have been looted. The data about the drake is globally important, so it's stored at the root of the game state, rather than in the cave map table.

Tracking Game State

Now that we have an idea of what game state means and what we'd like to track, let's write some code! Example quests-14 has all the code so far, and this where we'll be starting off.

Create a new file, “DefaultGameState.lua”, and add it to the manifest and dependencies files. We’ll store the default game state in here, which we use when starting a new game. This file contains a single function, GetDefaultGameState(), as shown in Listing 4.76.

```
function GetDefaultGameState()
    return
{
    defeated_cave_drake = false,
    maps =
{
    {
        town = {},
        world = {},
        cave =
{
            completed_puzzle = false,
            chests_looted = {}
        }
    }
}
end
```

Listing 4.76: Creating a default game state with the bits of information we want to track about the world. In DefaultGameState.lua.

The code in Listing 4.76 is similar to the previous sketch, Listing 4.75, but the chests_looted table is empty. Each chest will register itself in the chest_looted table automatically. This means we don’t have to do any manual steps when adding chests to a map; their state will persist automatically.

The game state information is global for our game, so we’ll store it in the World class. Listing 4.77 shows where to add it.

```
function World:Create()
    local this =
{
    -- code omitted
    mGameState = GetDefaultGameState()
}
setmetatable(this, self)
return this
end
```

Listing 4.77: Adding game state information to the world. In World.lua.

Now that our world has state information, it's time to start making use of it. Most maps need to track some game state, therefore when we create a map we'll pass in the global game state so any stored changes can be applied.

We create maps in two places in code, the main.lua file and the ReplaceScene storyboard event. Copy the updated ReplaceScene event from Listing 4.78.

```
function SOP.ReplaceScene(name, params)
    return function(storyboard)

        -- code omitted

        local mapDef = MapDB[params.map](gWorld.mGameState)
        state.mMap = Map:Create(mapDef)
        state.mMapDef = mapDef -- update the mapdef
```

Listing 4.78: Passing game state to the map creation function. In StoryboardEvents.lua.

The ReplaceScene function now passes in the game state when a map is created. We've also added a new line to make sure the ExploreState has an up-to-date def for the current map. We'll be making use of the ExploreState.mMapDef later on.

Next let's update the main file. In this file we load the cave map directly to make testing easier. We'll keep that as is but modify the call to pass in the game state. Copy the code from Listing 4.79.

```
do
    local cavemap = CreateCaveMap(gWorld.mGameState)
    gStack:Push(ExploreState:Create(gStack, cavemap, startPos))
end
```

Listing 4.79: Creating a cave map with state. In main.lua.

In Listing 4.79, we wrap the ExploreState creation code in a do-end block. This stops the cavemap variable from hanging around in memory, as it has quite a common name.

We now pass the game state through to the maps, but there's no code in the maps to store it. Let's fix that. At the same time we'll give each map def a name and an id field. The id is the same as the one used in the MapDB.lua file. The name field we'll use later to tell the player which map they're currently on.

Copy the code from Listing 4.80 to update the town map.

```

function CreateTownMap(state)
    local id = "town"
    local townState = state.maps[id]

    -- code omitted
    return
{
    id = id,
    name = "Town",
    -- code omitted
}
end

```

Listing 4.80: Adding an id and name to the town map. In map_town.lua.

That's the town done. Copy the code from Listing 4.81 to update the world map.

```

function CreateWorldMap(state)
    local id = "world"
    local worldState = state.maps[id]

    -- code omitted
    return
{
    id = id,
    name = "World",
    -- code omitted
}
end

```

Listing 4.81: Adding an id and name to the world map. In map_world.lua.

That's the world updated. Copy code from Listing 4.82 for the final map, the cave.

```

function CreateCaveMap(state)
    local id = "cave"
    local caveState = state.maps[id]

    -- code omitted
    return
{
    id = id,

```

```

        name = "Cave",
        -- code omitted
    }
end

```

Listing 4.82: Adding an id and name to the cave map. In map_cave.lua.

That's the cave up to date too.

The id field lets us look up the game state associated with a given map.

Next let's make use of the map's new name field. When the player opens the InGameMenuState the top panel should display the current map name. Copy the code from Listing 4.83.

```

InGameMenuState.__index = InGameMenuState
function InGameMenuState:Create(stack, mapDef)
    local this =
    {
        mMapDef = mapDef,

```

Listing 4.83: Storing the map name when we create the InGameMenu state. In InGameMenuState.lua.

Now let's make sure the InGameMenuState gets passed the current map def from the ExploreState. Copy the code from Listing 4.84

```

function ExploreState:HandleInput()

    -- code omitted

    if Keyboard.JustPressed(KEY_LALT) then
        local menu = InGameMenuState:Create(self.mStack, self.mMapDef)
        return self.mStack:Push(menu)
    end

end

```

Listing 4.84: Passing the map name to the InGameMenustate. In ExploreState.lua.

Finally we can update the code in FrontMenuState to actually display the current map name.

```

function FrontMenuState>Create(parent)

    -- code omitted

    this =
    {
        -- code omitted

        mTopBarText = parent.mMapDef.name,

```

Listing 4.85: Displaying the map name in the FrontMenuState.lua.

Run the game, open the menu, and marvel at the correct map name, as shown in Figure 4.31. That final piece of this menu has been a long time coming!



Figure 4.31: The map is now correctly named on the in-game menu screen.

Persisting Puzzles

When we call the CreateCaveMap function, we'll change how the map is created, depending on the game state.

Let's update the TakeGem function first. We'll set the defeated_cave_drake flag to true when the player wins the battle with the drake. Copy the code from Listing 4.86.

```

local TakeGem = function(map, trigger, entity, x, y, layer)

    -- code omitted

    local combatDef =
    {
        background = "combat_bg_cave.png",
        enemy = { "cave_drake" },
        canFlee = false,
        OnWin = function()
            state.defeated_cave_drake = true
        end
    }
}

```

Listing 4.86: Updating state when cave drake is defeated. In map_cave.lua.

In Listing 4.86 we've added a callback to the combat def so we're notified when the player wins, and that's where we set the "defeated_cave_drake" to true.

Next let's update the game state when the door puzzle is correctly solved. Copy the code from Listing 4.87.

```

local DoorPlateInteract = function(map, trigger, entity, x, y, layer)
    layer = layer or 1

    if gWorld:HasKey(keystoneId) then

        -- code omitted

        local storyboard =
        {
            SOP.NoBlock(SOP.Animate(leftDoor, leftAnim)),
            SOP.Animate(rightDoor, rightAnim),
            SOP.Function(
                function()
                    map:RemoveEntity(leftDoor)
                    map:RemoveEntity(rightDoor)
                    caveState.completed_puzzle = true
                end)
        }
}

```

Listing 4.87: Recording the puzzle state. In map_cave.lua.

When the doors finish opening and are removed, we set the completed_puzzle flag to true. If the puzzle has been completed, we want to make sure it doesn't reset when the player returns to the cave.

Let's modify SetupDoorPuzzle to check for the completed_puzzle flag, and if it's true we won't insert the doors. Copy the code from Listing 4.88.

```
local SetupDoorPuzzle = function(map, trigger, entity, x, y, layer)
    local tileDoorPlateId = 182

    local isFourthTileFilled = false

    if caveState.completed_puzzle then
        isFourthTileFilled = true
    end

    local tileLocations =
    {
        { 39, 56, true},
        { 42, 56, true},
        { 38, 41, true},
        { 41, 41, isFourthTileFilled}
    }
    local keyX = 56
    local keyY = 40

    for k, v in ipairs(tileLocations) do
        -- code omitted
    end

    if caveState.completed_puzzle then
        -- Don't proceed after this point if the puzzle
        -- is already solved.
        -- This stops the keystone being added
        -- The doors being added.
        return
    end

    -- code omitted
end
```

*Listing 4.88: Code to only reset the door puzzle if hasn't previously been completed.
In map_cave.lua.*

The SetupPuzzle function adds four pressure plates to the map by default. Three pressure plates have an orb on top. If the player has already finished the puzzle, the SetupPuzzle function adds an orb to the fourth pressure plate too. Orbs are added depending on the value of a boolean, so we just change the forth boolean to true if the completed_puzzle flag is set.

If the puzzle is already complete, we don't want to add the keystone or locked doors, so we leave the function before this setup code is executed.

Try it now. Though I'd suggest turning off random encounters! Solve the puzzle, exit the cave, come back, and the puzzle is still solved! Great.

Persistent Chests

If you open a treasure chest, leave the map, and return - the treasure respawns. That's an infinite item bug! We need a way to record if a chest has been looted. We'll write the code so we don't need to change any of the chests we've already placed, and any chest we add in the future will work automatically.

Each chest is given an id based on the order it's loaded in the map. The first chest added by the AddChest function is chest 1, the second is chest 2, and so on. We use this id as an index into the game state's `looted_chests` table. If there's no existing entry, then the chest hasn't been looted. If there's an entry set to true, then the chest has been looted.

Let's modify the AddChest action to implement this. Copy the code from Listing 4.89.

```
AddChest = function(map, entityId, loot, x, y, layer)

    layer = layer or 1

    map.mContainerCount = map.mContainerCount or 0
    map.mContainerCount = map.mContainerCount + 1
    local containerId = map.mContainerCount
    local mapId = map.mMapDef.id
    local state = gWorld.mGameState.maps[mapId]
    local isLooted = state.chests_looted[containerId] or false

    return function(trigger, entity, tX, tY, tLayer)

        local entityDef = gEntities[entityId]
        assert(entityDef ~= nil)
        local chest = Entity:Create(entityDef)

        chest:SetTilePos(x, y, layer, map)

        if isLooted then
            -- Open chest and return before
            -- setting trigger.
            chest:SetFrame(entityDef.openFrame)
            return
        end
    end
}
```

```

-- Define open chest function
local OnOpenChest = function()

    -- code omitted

    -- Remove the trigger
    map:RemoveTrigger(chest.mTileX, chest.mTileY, chest.mLayer)
    chest:SetFrame(entityDef.openFrame)
    state.chests_looted[containerId] = true
end

-- code omitted
end
end,

```

Listing 4.89: Modifying the AddChest action, so chest state persists. In Actions.lua.

When the AddChest action runs, it assigns the chest a containerId by adding 1 to the mContainerCount of the current map. If the mContainerCount doesn't exist it's set to 0, which means the first chest has an id of 1.

The containerId uniquely identifies the chest for the current map. The chests_looted table contains the looted data for each chest. When a chest is loaded, we find out if it's been looted by using its id to index the chests_looted table. In the AddChest action, the looted state of the current chest is stored in the boolean isLooted.

If isLooted is true, we add the chest to the map using the open sprite and exit the function before a trigger is added to the chest. When the player opens a chest, we index the chests_looted table with its id and set its looted value to true.

You can try this out! To make the test a little easier, spawn the player by the cave entrance. All the code so far is available in quest-14-solution.

Saving and Loading

The world now tracks the state of the game, and when we return to a map the chests and puzzles are as we left them. This is pretty cool, but it's just a prelude to the ultimate persistence system; saving and loading. Adding saving and loading allows our players to actually shut down the game, have their dinner, and carry on playing a different day. A pretty important feature!

Originally, due to hardware limitations, JRPGs only allowed the character to save at designated points, either a save-point on the map or commonly anywhere on the world map. Later, save points became more of a gameplay mechanism and marked a "safe zone" on the map for the player to try to reach.

We'll stick to traditional JRPG conventions and allow saving at specific points or anywhere on the world map.

Saving the game is simple in theory; we take the parts of the game we want to save, put them into a Lua table, and write it to disk. To load the game, we load the saved Lua table and use it to overwrite the parts of the game state. The load code needs to execute some code to sync everything up (for instance if the current map id saved as "the jail", we need to unload the current map and load the jail map).

What Needs Saving?

Let's decide what's important to save. Then we'll define a table to describe the data to extract from the current game state.

Here a list of the important bits of data.

- Current Map Id
- X, Y and Layer position of map
- GameState table
- Time played
- Gold
- KeyItems
- Who is in the party
- Inventory

Then for each party member

- Stats
- Equipment
- Level
- Actions

First we'll deal with saving and loading to a Lua table in memory, then we'll write code to save the table to disk.

The data we want to save is scattered around a little. Some of it's simple like the `mGold` count; that's just a field in the `gWorld` table. But what about the player position? The X and Y values are stored in the `ExploreState` on the `gStack`. When we're saving, the `ExploreState` isn't guaranteed to be on top of that stack, so we need code to search out certain bits of data.

Example `quests-15` contains the code so far and a new file called "`PrintTable.lua`". This file lets us print out any Lua table in a nicely formatted way. I'm not going to cover this code, but you're welcome to take a look. You don't need to look at this file, or

understand it. All you need to know is that if you call PrintTable(t) it will print out the contents of t in a nice way²!

Earlier in the book, we wrote the ShallowClone function. It copies all the values of a table into a new table. This works great for arrays, but to clone a table like gWorld.mGameState we need something that clones tables even when they're nested inside each other. Let's add another function called DeepClone. It clones a table and all its subtables. It's a simple implementation and doesn't deal with loops or non-Lua data structures, like vectors. Copy the code from Listing 4.90.

```
function DeepClone(t)
    local clone = {}
    for k, v in pairs(t) do
        if type(v) == "table" then
            clone[k] = DeepClone(v)
        else
            clone[k] = v
        end
    end
    return clone
end
```

Listing 4.90: A very simple Deep Clone function. In Util.lua.

Listing 4.91 shows a small test to demonstrate that DeepClone is working correctly.

```
local list = { b = { "hello", "goodbye" } }
print(list.b[1])                                -- "hello"
local cloneList = DeepClone(list)
print(cloneList.b[1])                            -- "hello"
list.b[1] = "change"
print(list.b[1])                                -- "change"
print(cloneList.b[1])                            -- "hello"
```

Listing 4.91: Test of the simple DeepClone function.

Try changing DeepClone in Listing 4.91 to ShallowClone and see how the result changes.

Let's define a SaveScheme table to describe the data we want to save. To make things simple, we'll start by describing the important gWorld data. Add a file called SaveScheme.lua to the project and include it in the manifest and dependencies files. Then copy the code from Listing 4.92.

²I think with this caveat, everyone is going to want to look inside this file. The code requires a good understanding of Lua to parse.

```

SaveScheme =
{
    ['gWorld'] =
    {
        meta = "marked",
        fields =
        {
            ['mTime'] = { meta = "copy" },
            ['mGold'] = { meta = "copy" },
            ['mItems'] = { meta = "copy" },
            ['mKeyItems'] = { meta = "copy" },
            ['mGameState'] = { meta = "copy" },
        }
    }
}

```

Listing 4.92: Start of the SaveScheme table. In SaveScheme.lua.

The save scheme has one entry, gWorld, the table we want to extract data from. We don't want to save *everything* in the gWorld table, so we have some metadata meta set to "marked". The marked tag means we only want to save the fields specified in the fields subtable. Each entry in the fields table has a meta tag, and they're all set to "copy", which means "save everything".

Next let's write code that takes the SaveScheme and returns a table of the values we want to save. We'll call this function Extract. For now, you can write this function at the bottom of the SaveScheme.lua file. Copy the code from Listing 4.93.

```

function Copy(source)
    if type(source) == "table" then
        return DeepCopy(source)
    else
        return source
    end
end

function Extract(t, scheme)
    local data = {}

    for id, v in pairs(scheme) do

        local source = t[id]
        local meta = v.meta

        if meta == "marked" then

```

```

        data[id] = Extract(source, v.fields)
    elseif meta == "copy" then
        data[id] = Copy(source)
    end

end

return data
end

```

Listing 4.93: A function to extract data using a save scheme. In SaveScheme.lua.

The Extract function creates an empty table called data that we'll fill with the data we want to extract. It takes in two arguments: t, the table we're extracting the data from; and scheme, a description of the data to extract.

To fill in the data table, we loop over the scheme. This scheme is the one defined in SaveScheme.lua. Currently the scheme table only has one entry, gWorld. For each entry in the scheme, we look up the corresponding data in the t table. For our scheme this means we store the gWorld value in the local variable called source. The meta flag for the field is stored in meta. If the meta flag is set to "marked", then we call the Extract method, recursively passing in the contents of the source and the fields section of the scheme. The second call to Extract looks through the marked fields and sees that they all have their meta flag set to "copy". This means that each value from the gWorld table is passed into the Copy function, which returns a copy of the value passed in.

In this way we extract all the data we want from the game. The recursion can be tricky to understand, but don't worry too much about it. We only deal with recursion while building the save system. Once we've built the system, you can use it and never think about recursion again!

Let's see the Extract function in action. Switch to main.lua and copy the code from Listing 4.94.

```

local tmpSave = nil
function update()
    local dt = GetDeltaTime()
    gStack:Update(dt)
    gStack:Render(gRenderer)
    gWorld:Update(dt)

    if Keyboard.JustPressed(KEY_S) then
        tmpSave = Extract(_G, SaveScheme)
        PrintTable(tmpSave)
    end
end

```

Listing 4.94: Adding a quicksave in main.lua.

Run the game, hit 'S', and our extracted values are displayed in the console window. This uses the PrintTable function from quests-15 to pretty-print our extracted data.

You might be wondering what _G is. _G is Lua's global table containing all the values in our program. Want to see something cool? Try calling PrintTable(_G) when 'S' is pressed!

Listing 4.95 contains the console output when we save.

```
{  
    ["gWorld"] =  
    {  
        ["mItems"] =  
        {  
            {  
                ["id"] = 1,  
                ["count"] = 2,  
            },  
            {  
                ["id"] = 2,  
                ["count"] = 3,  
            },  
            {  
                ["id"] = 10,  
                ["count"] = 5,  
            },  
            {  
                ["mTime"] = 1137.8860000012,  
                ["mKeyItems"] = {},  
                ["mGold"] = 5,  
                ["mGameState"] =  
                {  
                    ["maps"] =  
                    {  
                        ["world"] = {},  
                        ["town"] = {},  
                        ["cave"] =  
                        {  
                            ["completed_puzzle"] = false,  
                            ["chests_looted"] = {},  
                        },  
                        {  
                            ["defeated_cave_drake"] = false,  
                        },  
                    },  
                },  
            },  
        },  
    },  
}
```

```
    },
},
```

Listing 4.95: Extracted Save Data from the console.

So far so good. We've extracted the data we want to save from the gWorld table and cloned it.

Let's write another function, called Patch, that writes this data back into the gWorld object. Again, for now, we'll do this at the bottom of SaveScheme.lua. Copy the code from Listing 4.96.

```
function Patch(data, save, scheme)

    for id, v in pairs(scheme) do

        local source = save[id]

        local meta = v.meta

        if meta == "marked" then
            Patch(data[id], source, v.fields)
        elseif meta == "copy" then
            data[id] = Copy(source)
        end

    end

end
```

Listing 4.96: The Patch function. In SaveScheme.lua.

The function Patch uses the scheme to find fields and tables in the save table and overwrite them with values from the data table.

Let's add a Load function that makes use of the Patch function. Copy the code from Listing 4.97.

```
local tmpSave = nil
function update()

    -- code omitted

    if Keyboard.JustPressed(KEY_L) then
        if tmpSave then
```

```

        Patch(_G, tmpSave, SaveScheme)
    end
end

```

Listing 4.97: Adding a load shortcut. In main.lua.

If you try it out, it's hard to notice if anything even happens! To see it working, open the in-game menu, press L, and watch the "Time" value change. As you save and load the game state, you can see the time value update. If you want to get a little more inventive, you can edit the tmpSave table, add an item or remove one, then watch things update. When we load and save, the map isn't reloaded (yet), so if you want to see changes saved and loaded on the map, you'll need to leave the map and come back.

Let's extend the SaveScheme to define more data to save and load. We'll start with the party members. The party is tricky to save. It's stored as a Party object, and each member is an actor containing texture references and other complicated relationships. We cannot simply clone these tables.

The important parts of the party class need to be carefully extracted. Then, when we load, we clear the party list and recreate it, patching in the new values. The party members may be different throughout the game. Sometimes there is only 1 member, sometimes there are 3, etc. We need to be able to handle this too.

Let's iterate on the SaveScheme a little. Copy the code from Listing 4.98.

```

SaveScheme =
{
    meta = 'marked',
    fields =
    {
        ['gWorld'] =
        {
            meta = "marked",
            fields =
            {
                ['mTime'] = { meta = "copy" },
                ['mGold'] = { meta = "copy" },
                ['mItems'] = { meta = "copy" },
                ['mKeyItems'] = { meta = "copy" },
                ['mGameState'] = { meta = "copy" },
                ['mParty'] =
                {
                    meta = 'marked',
                    meta_before_patch = nil, -- we'll come to this later
                    fields =
                    {

```

Listing 4.98: Everything we need to save out of the world. In SaveScheme.lua.

The first thing to note is the meta field at the top level of the scheme. The ['gWorld'] key is set as a marked field to save. This makes the scheme definition more consistent and the Extract/ Patch functionality easier to extend.

The scheme in Listing 4.98 now handles the party member data. There's a new `meta_before_patch` field. This is a function that gets called before we patch in the data. These calls can be added to any metadata for any field. In this case, we have a function that clears the party and then replaces it with new actors loaded from the data table. We'll implement this function shortly.

The final thing to notice is that the `mMembers` meta field is set to a new value, "each". The "each" tag means: "For each of the values in the `mMembers` table, extract the data using the meta data under values". This way, "each" lets us deal with lists of entries that share the same layout. In this case we're grabbing some of the actor data. We record all the important variables we want to persist; the XP, stats, equipment, and so on. We don't clone the entire `mStats` table because it's a class; instead we just take the base stats and modifiers.

Let's implement the function value for the `meta_before_patch` field. Copy the code from Listing 4.99.

```
-- code omitted
['mParty'] =
{
    meta = 'marked',
    meta_before_patch = function(data, source, scheme, dataParent, id)
        dataParent[id] = Party:Create() -- clear the mParty table
        local members = source.mMembers
        for k, v in pairs(members) do
            local def = gPartyMemberDefs[v.mId]
            local actor = Actor:Create(def)
            dataParent[id]:Add(actor)
        end
    end,
}
```

Listing 4.99: The `meta_before_patch` implementation used to load the party member data. In `SaveScheme.lua`.

The `meta_before_patch` function takes in five parameters:

- `data` - A value from the game memory. In this case it's the `mParty` object stored in the `gWorld` table.
- `source` - The saved data associated with the field we're loading. In this case the source is everything we saved out from the `mMembers` table.
- `scheme` - The part of the scheme describing what we're saving out.
- `dataParent` - This is the parent of data, so in this case the data is `mMembers` and the parent is `gWorld`.
- `id` - The name of the field we're editing. In our case it's "mMembers".

The function itself recreates the `mParty` field and adds the actors from the save data. After the function finishes, we continue the patching and overwrite the default xp and level etc. with the saved values.

We're nearly ready to update the Patch and Extract functions, but first let's totally finish the save scheme. We need to save the important data from the `ExploreState`. The `ExploreState` is hard to access, so we use a special function to extract the data and a special one to load it back in. Copy the code from Listing 4.100.

```
SaveScheme =
{
    meta = 'marked',
    fields =
    {
        ['gWorld'] =
        {
            -- code omitted
        },
        ['gStack'] =
        {
            meta = 'function',
            meta_extract_function = function(data, scheme)
                -- Get ExploreState
                local exploreState = nil
                for k, v in ipairs(data.mStates) do
                    if v.__index == ExploreState then
                        exploreState = v
                        break
                    end
                end
                -- Extract the start pos and map id
                return
            {
                mapId = exploreState.mMapDef.id,
                x = exploreState.mHero.mEntity.mTileX,
                y = exploreState.mHero.mEntity.mTileY,
                layer = exploreState.mHero.mEntity.mLayer
            }
        end,
    }
},
meta_after_patch = function(data, save, scheme, dataParent, id)

    -- Clear the stack
    gStack = StateStack>Create()
```

```

local stackData = save['gStack']
local map = MapDB[stackData.mapId]
map = map(gWorld.mGameState)
local startPos = Vector.Create(stackData.x,
                               stackData.y,
                               stackData.layer)

local explore = ExploreState:Create(gStack,
                                    map,
                                    startPos)
gStack:Push(explore)

end
}

```

Listing 4.100: A function to extract the data we need for loading the map.

In Listing 4.100 we've added extra code to save out the gStack. The meta field for gStack is set to "function". This means we save this data manually using a custom function. The `meta_extract_function` loops through the `gStack.mStates` to find the `ExploreState`. (It does this by checking each of the states until it finds one with an `ExploreState` metatable.) Once it finds the `ExploreState` it gets the x, y and layer position of the hero and the id of the current map. That's all we need to save. These values are returned in a table to save out. Later when we call the `Patch` function, we use a special `meta_after_patch` function to load in the `ExploreState` data.

This `meta_after_patch` function is called after the patching has finished and when we know that `gWorld` data has been loaded in. This function clears the global stack, gets the map and hero location data we saved out, then creates the new map with the up-to-date `GameState` and pushes a new `ExploreState` onto the stack.

With the save scheme finished, let's write the final implementation of `Patch` and `Extract`. First create a new file: `Save.lua`. This will contain the `Patch` and `Extract` functions. Remember to remove any code at the bottom of `SaveScheme` prior to making this file. Add `Save.lua` to the manifest and dependencies files.

Copy the new `Patch` and `Extract` code from Listing 4.101.

```

Save = {}

function Save:Copy(source)
    if type(source) == "table" then
        return DeepClone(source)
    else
        return source
    end
}

```

```

end

function Save:ExtractEach(t, scheme)
    local data = {}
    for k, v in pairs(t) do
        data[k] = Save:Extract(v, scheme)
    end
    return data
end

function Save:ExtractMarked(t, scheme)
    local data = {}

    for id, v in pairs(scheme) do
        local source = t[id]
        data[id] = Save:Extract(source, v)
    end

    return data
end

function Save:Extract(source, scheme)
    if scheme.meta == 'marked' then
        return Save:ExtractMarked(source, scheme.fields)
    elseif scheme.meta == 'copy' then
        return Save:Copy(source)
    elseif scheme.meta == 'each' then
        return Save:ExtractEach(source, scheme.value)
    elseif scheme.meta == 'function' then
        return scheme.meta_extract_function(source, scheme)
    end
end

```

Listing 4.101: Create a Save class with the Extract methods. In Save.lua.

The Save table holds the Extract and Patch functions which we call directly.

The Extract function takes in two parameters, the source we're extracting data from and the scheme that describes what we want to extract. The code is similar to our previous Extract implementation. The Extract function calls Copy, ExtractEach, ExtractMarked, or a custom function, depending on the scheme's metadata.

We've already covered ExtractMarked and Copy, but the ExtractEach function is new. The ExtractEach function loops through a source table and extracts data from each entry using the same metadata.

Let's add the Patch functions next. Copy the code from Listing 4.102.

```

function Save:PatchEach(data, save, scheme)
    for id, v in pairs(save) do
        Save:Patch(data[id], save[id], scheme, data, id)
    end
end

function Save:PatchMarked(data, save, scheme)

    for id, v in pairs(scheme) do
        Save:Patch(data[id], save[id], v, data, id)
    end

end

function Save:Patch(data, save, scheme, dataParent, id)
    local meta = scheme.meta

    if scheme.meta_before_patch then
        scheme.meta_before_patch(dataParent, save,
                                scheme, dataParent, id)
    end

    if meta == 'marked' then
        Save:PatchMarked(data, save, scheme.fields)
        if dataParent then
            dataParent[id] = data
        end
    elseif meta == 'copy' then
        dataParent[id] = Save:Copy(save)
    elseif meta == 'each' then
        -- Create entry if it's missing
        dataParent[id] = dataParent[id] or {}
        Save:PatchEach(data, save, scheme.value)
    end

    if scheme.meta_after_patch then
        scheme.meta_after_patch(dataParent, save,
                                scheme, dataParent, id)
    end
end

```

Listing 4.102: Adding the Patch functions to the save class. In Save.lua.

The Patch function is an extended version of the previous implementation.

There's a new check for a `meta_before_patch` entry in the scheme. If this function exists, it's called before doing the rest of the patch. There's a matching `meta_after_patch` function that is called after a patch is applied.

The `Patch` function supports a new `PatchEach` function, which is called when there's an `each` meta tag. Before calling the `PatchEach` function, we check if the destination table exists, and if it doesn't we create it. The `PatchEach` function iterates through the save data entries and writes each into the target data table.

To test out the new save and load code, let's update our `main.lua`. Copy the code from Listing 4.103.

```
local tmpSave = nil
function update()
    -- code omitted

    if Keyboard.JustPressed(KEY_S) then
        tmpSave = Save:Extract(_G, SaveScheme)
    end

    if Keyboard.JustPressed(KEY_L) then
        if tmpSave then
            Save:Patch(_G, tmpSave, SaveScheme)
        end
    end
end
```

Listing 4.103: Updating Update function with quick load save keys. In main.lua.

Save the game in the cave by pressing S, then go the world map and load the save data by pressing L. When you load, you'll return to the cave you just left! Pretty cool. Next we need to write the save data to the hard disk and add some UI to make it easier to use.

Save and Loading Lua Data

Dinodeck has a function that can write a string to a file. Unfortunately, we have a Lua table not a string. To make things easier, we'll cheat a bit and use some premade code that turns a Lua table into a string.

In the `quest-15` code directory, there's a file called `Blob.lua` which has already been added to the manifest and dependency files. It contains a `Blob` class that encodes and decodes Lua tables to and from strings. The `Blob` class has two functions that we care about, `Encode` and `Decode`. You can see an example of it in use in Listing 4.104.

```

local example_data = { field = { 2, 3, 4}, t = { t = {} } }
local str = Blob:Encode(example_data)
local data = Blob:Decode(str)
-- data and example_data are equal copies of each other

```

Listing 4.104: Use of the blob class.

We use the Blob code to transform Lua save data into a string that we can save to disk.

In Dependencies.lua we need to add the SaveGame library so we can write and read from the disk. Update your code to match Listing 4.105.

```

Apply({
    "Renderer",
    -- code omitted
    "SaveGame"
},
function(v) LoadLibrary(v) end)

```

Listing 4.105: Adding the SaveGame library. In Dependencies.lua.

Let's update the Save class to write and read from disk. Copy the code from Listing 4.106.

```

Save =
{
    mSaveGame = SaveGame.Create("save.dat")
}

-- code omitted

function Save:DoesExist()
    return not (self.mSaveGame:Read() == "")
end

function Save:Save()
    local tmpSave = self:Extract(_G, SaveScheme)
    self.mSaveGame:Write(Blob:Encode(tmpSave))
end

function Save:Load()
    local tmpSave = self.mSaveGame:Read()
    if tmpSave == "" then
        return
    end

```

```

    end
    tmpSave = Blob:Decode(tmpSave)
    self:Patch(_G, tmpSave, SaveScheme)
end

```

Listing 4.106: Update the Save class to write and read from disk. In Save.lua.

We've added three functions. DoesExist returns true if a previous save exists, otherwise it returns false. Save extracts the save data and writes it to disk. Load reads the save data and patches it into the game.

Update the main.lua file to use these functions and then you can test it out. Run the game and save, then shut down the program, open it back up, and load your previous progress!

All code is available in quest-15-solution.

Save and Load From The Menu

Asking the player to press a random key on the keyboard to save and load a game isn't user friendly. Let's hook the save and load functionality into the game menus.

On the very first screen we should display a "Continue" option if there's an existing saved game. Let's add that next! Example quests-16 contains the code so far and an updated title_screen.png image.

The TitleScreenState class is the same as the one from the Dungeon mini-game at the end of chapter 1. Let's edit the main game, so this state is the first thing on the game stack. Copy the code from Listing 4.107.

```

LoadLibrary('Asset')
Asset.Run('Dependencies.lua')

gRenderer = Renderer.Create()

function SetupNewGame()
    gStack = StateStack>Create()
    gWorld = World>Create()

    local startPos = Vector.Create(5, 9, 1)

    local hero = Actor>Create(gPartyMemberDefs.hero)
    local thief = Actor>Create(gPartyMemberDefs.thief)
    local mage = Actor>Create(gPartyMemberDefs.mage)
    gWorld.mParty:Add(hero)
    gWorld.mParty:Add(thief)

```

```

gWorld.mParty:Add(mage)

gWorld.mGold = 5
gWorld:AddItem(1, 3)
gWorld:AddItem(2, 3)
gWorld:AddItem(10, 5)

local sayDef = { textScale = 1.3 }
local intro =
{
    -- code omitted
}

do
    local map = MapDB['town'](gWorld.mGameState)
    gStack:Push(ExploreState>Create(gStack, map, startPos))
end

return Storyboard>Create(gStack, intro, true)
end

local storyboard = SetupNewGame()
gStack:Push(TitleScreenState>Create(gStack, storyboard))
math.randomseed( os.time() )

function update()
    local dt = GetDeltaTime()
    gStack:Update(dt)
    gStack:Render(gRenderer)
    gWorld:Update(dt)
end

```

Listing 4.107: Getting the main game set up with a start menu. In main.lua.

The main.lua code has a new SetupNewGame function that wraps up the game setup code. We'll call the SetupNewGame function to restart the game from the game over state. Inside the setup function the town map is loaded and put on the stack, the player is positioned in the town hall, and we push the title screen state on top of the explore state. The title screen state takes in the introduction storyboard so it can trigger it when the player starts a new game.

Note that we're seeding the random number generator with the system time. If you don't seed the generator, it generates the same numbers each time the program runs, which is useful for debugging but not so great for gameplay!

When we create the actors in the SetupNewGame function, we assume they start at level

0 but this might not always be the case. Let's add code so that if the actor def specifies a certain level, the created actor will have the correct stats and abilities.

Copy the code from Listing 4.108.

```
function Actor:Create(def)

    -- code omitted

    self.mNextLevelXP = NextLevel(this.mLevel)
    setmetatable(this, self)
    this:DoInitialLeveling()
    return this
end

function Actor:DoInitialLeveling()

    -- Only for party members
    if not gPartyMemberDefs[self.mId] then
        return
    end

    -- Get total XP current level represents
    for i = 1, self.mLevel do
        self.mXP = self.mXP + NextLevel(i - 1)
    end
    -- Set level back to 0
    self.mLevel = 0
    self.mNextLevelXP = NextLevel(self.mLevel)

    -- unlock all abilities / add stats
    while(self:ReadyToLevelUp()) do
        local levelup = self>CreateLevelUp()
        self:ApplyLevel(levelup)
    end
end
```

Listing 4.108: Adding initial levelling up. In Actor.lua.

When an actor is created, we now call the DoInitialLeveling function. This function checks if the actor is a party member and if so we cash in all the levels for XP, then we use that XP to level up, unlocking all the skills and stat bonuses along the way.

Next let's update the TitleScreenState and make the "Continue" option work. Copy the code from Listing 4.109.

```

TitleScreenState = {}
TitleScreenState.__index = TitleScreenState
function TitleScreenState:Create(stack, storyboard)
    local this
    this =
    {
        mTitleBanner = Sprite.Create(),
        mStack = stack,
        mStoryboard = storyboard,
    }

    this.mTitleBanner:SetTexture(Texture.Find("title_screen.png"))
    this.mTitleBanner:SetPosition(0, 100)

    this.mShowContinue = Save:DoesExist()

    local data = {}
    if this.mShowContinue then
        table.insert(data, "Continue")
    end
    table.insert(data, "Play")
    table.insert(data, "Exit")

    this.mMenu = Selection>Create
    {
        data = data,
        spacingY = 24,
        OnSelection = function(index)
            this:OnSelection(index)
        end
    }

    -- Code to center the menu
    this.mMenu.mCursorWidth = 50
    this.mMenu.mX = -this.mMenu:GetWidth()/2 - this.mMenu.mCursorWidth
    this.mMenu.mY = -50

    setmetatable(this, self)
    return this
end

```

Listing 4.109: The constructor for the title screen state. In TitleScreenState.lua.

The TitleScreenState state takes in the stack and the introduction storyboard. The this table contains three members: the stack, the sprite for the title image, and the

storyboard. The sprite uses the "title_screen.png" texture and is positioned in the top half of the screen. The Save.DoesExist method tells us if a save file exists or not and therefore if we are able to continue. We use this function to set the "mShowContinue" flag that controls if the "Continue" option is shown.

The data source for the selection menu is created from a table of the strings "Continue", "New", and "Exit". The "Continue" item is only added if the mShowContinue flag is set to true. The final part of the constructor positions the menu. When an item on the menu is selected the OnSelection function is called.

Let's update the OnSelection callback to hook up the code for the "Continue" option. Copy the code from Listing 4.110.

```
function TitleScreenState:OnSelection(index)

    if not self.mShowContinue then
        index = index + 1
    end

    if index == 1 then
        Save:Load()
    elseif index == 2 then
        self.mStack:Pop()
        self.mStack:Push(self.mStoryboard)
        -- We're in the update function, so update the storyboard
        -- before it gets rendered.
        self.mStoryboard:Update(0)
    elseif index == 3 then
        System.Exit()
    end
end
```

Listing 4.110: Updating the title screen OnSelection callback. In TitleScreenState.lua.

The OnSelection function takes an index for the currently selected menu option. If the continue option is enabled the 1st index is "Continue", otherwise it's "New Game". We increase the index value by 1 if we cannot continue, making "New" index always index 2.

If the index is 1, the player wants to continue and therefore we load the save game file. If the index is 2, the player wants to start a new game, so we pop the title state off the stack, and push on the storyboard. We also tell the storyboard to update because otherwise the storyboard gets a Render call before it gets a chance to update. If the index is 3, the player wants to exit and we exit the program.

Example quests-16 has a save file in the root directory. If you run the code you'll be able to see all the menu options, including continue. Try starting a new game and continuing.

In Game Menu Save and Load

We want the player to save on the world map or in any safe town. We'll add a can_save field to the map def to tag which maps the player can save on. If the player can save on a map then they'll be able to access the "Save" option in the in-game menu. The player can load at any time from the in-game menu. To keep things simple, we only support a single save game file.

Let's add the new field to the map definitions. First add it to the world as in Listing 4.111.

```
function CreateWorldMap(state)
    -- code omitted
    return
{
    can_save = true,
    -- code omitted
}
end
```

Listing 4.111: Adding the can_save field to the world map. In map_world.lua.

Next add the tag to the town as in Listing 4.112.

```
function CreateTownMap(state)
    -- code omitted
    return
{
    can_save = true,
    -- code omitted
}
end
```

Listing 4.112: Adding the can_save field to the town map. In map_town.lua.

And finally add the can_save tag to the cave as in Listing 4.113.

```
function CreateCaveMap(state)
    -- code omitted
    return
{
    can_save = false,
    -- code omitted
}
end
```

Listing 4.113: Adding the can_save field to the cave map. In map_cave.lua.

We allow saving on the world and town maps but not in the cave.

Next let's prepare the FrontMenuState for the save and load options. Copy the code from Listing 4.114.

```
FrontMenuState = {}
FrontMenuState.__index = FrontMenuState
function FrontMenuState:Create(parent)

    -- code omitted

    local this
    this =
    {
        -- code omitted

        mSelections = Selection:Create
        {
            spacingY = 32,
            data =
            {
                { id = "items", text = "Items" },
                { id = "status", text = "Status" },
                { id = "equipment", text = "Equipment" },
                { id = "save", text = "Save" },
                { id = "load", text = "Load" }
            },
            RenderItem = function(...) this:RenderMenuItem(...) end,
        }
    }
end
```

Listing 4.114: Adding Save and Load options. In FrontMenuState.lua.

In Listing 4.114 we add “Save” and “Load” entries to the mSelections menu and update all the entries with a text and an id field. The id field uniquely identifies the menu item. The text field is the text displayed on screen. We use the id field to lock off the “Save” option when it’s not available. Separating the menu options into id and text means we can change the text whenever we want and not worry about breaking the menu. This is useful if you ever want to localize your game.

We've added a new RenderItem callback that can gray out the “Save” entry if we're in a position where we cannot save. Copy the code from Listing 4.115.

```
function FrontMenuState:RenderMenuItem(menu, renderer, x, y, item)
```

```

local color = Vector.Create(1, 1, 1, 1)
local canSave = self.mParent.mMapDef.can_save
local text = item.text

if item.id == "save" and not canSave then
    color = Vector.Create(0.6, 0.6, 0.6, 1)
end

if item.id == "load" and not Save:DoesExist() then
    color = Vector.Create(0.6, 0.6, 0.6, 1)
end

if item then
    renderer:DrawText2d(x, y, text, color)
end
end

```

Listing 4.115: Implementing the RenderMenuItem callback. In FrontMenuState.lua.

The RenderMenuItem function in Listing 4.115 acts like the default renderer unless the current map has its can_save flag set to false. In this case we gray out the “Save” label to indicate the game cannot be saved at this time. We also gray out the “Load” label if there’s no game to load. A screenshot of the new menu options can be seen in Figure 4.32.



Figure 4.32: The menu options for saving and loading. Save is disabled because the player is in a dungeon.

Visually the menu is looking correct, but we need to hook everything up. Let's edit the OnMenuClick function. Copy the code from Listing 4.116.

```
function FrontMenuState:OnMenuClick(index, item)

    if item.id == "items" then
        return self.mStateMachine:Change("items")
    end

    if item.id == "save" then
        if self.mParent.mMapDef.can_save then
            Save:Save()
            self.mStack:PushFit(gRenderer, 0, 0, "Saved!")
        end
        return
    end

    if item.id == "load" then
        if Save:DoesExist() then
            Save:Load()
            gStack:PushFit(gRenderer, 0, 0, "Loaded!")
        end
        return
    end

    self.mInPartyMenu = true
    self.mSelections:HideCursor()
    self.mPartyMenu>ShowCursor()
    self.mPrevTopBarText = self.mTopBarText
    self.mTopBarText = "Choose a party member"
end
```

Listing 4.116: Updating the OnMenuClick callback for load and save. In FrontMenuState.lua.

The code in Listing 4.116 actually performs the load and save operations. To let the player know something's happened, we push a message box onto the screen.

Save Points

The cave ends with a dangerous boss fight; it would be extremely frustrating to die at the hands of the boss and then have the nearest save be outside the cave entrance!

Therefore we're going to add a simple save point closer to the boss. Using a save point fully heals the player and asks if they'd like to save the game.

Let's start by creating the entity to represent the save point. There's an image file called `save_point.png` in the art folder that's already been added to the manifest. Let's create an entity for the save point so we can place it on the map. Copy the code from Listing 4.117.

```
gEntities =
{
    -- code omitted
    save_point =
    {
        texture = "save_point.png",
        width = 16,
        height = 16,
        startFrame = 1,
    }
}
```

Listing 4.117: Adding the save point entity. In EntityDefs.lua.

Next let's add an action, `AddSavePoint`, that inserts a save point and a trigger into the world. Copy the code from Listing 4.118.

```
Actions =
{
    -- code omitted

    AddSavePoint = function(map, x, y, layer)

        return function(trigger, entity, tX, tY, tLayer)

            local entityDef  = gEntities["save_point"]
            local savePoint = Entity:Create(entityDef)
            savePoint:SetTilePos(x, y, layer, map)

            local function AskCallback(index)
                if index == 2 then
                    return
                end
                Save:Save()
                gStack:PushFit(gRenderer, 0, 0, "Saved!")
            end
        end
    end
}
```

```

local trigger = Trigger>Create(
{
    OnUse = function()
        gWorld.mParty:Rest()
        local askMsg = "Save progress?"
        gStack:PushFit(gRenderer, 0, 0, askMsg, false,
        {
            choices =
            {
                options = {"Yes", "No"},
                OnSelection = AskCallback
            },
        })
    end
})
map:AddFullTrigger(trigger, x, y, layer)
end
end,

```

Listing 4.118: Creating a new AddSavePoint action. In Actions.lua.

The AddSavePoint action, in Listing 4.118, takes in the map and a location to place the save point. When the action runs, it gets the “save_point” entity def from the gEntities table, creates an entity, and places it at the specified location on the map. It then places a trigger at the same location.

Using the save point automatically restores the party’s health and mana by calling the mParty.Rest function. It then opens a dialog box to ask if the player would like to save. If the player selects “Yes”, the game is saved and a dialog box with the text “Saved!” is pushed onto the stack.

Let’s use the action to add a save game point to the cave before the boss encounter. Copy the code from Listing 4.119.

```

function CreateCaveMap(state)

    -- code omitted

    return
{
    -- code omitted
    on_wake =
    {
        -- code omitted
    }
}

```

```
    id = "AddSavePoint",
    params = { 26, 29, 1 }
},
```

Listing 4.119: Adding a save point before the boss. In map_cave.lua.

In Listing 4.119 we place the save point near the boss area. You can see it in action in Figure 4.33. For testing, spawn the hero next to the save point or place it near the cave entrance.



Figure 4.33: Saving the game at the boss save point.

Try it out. We can now save and load games! The save / load scheme is pretty easy to extend, so if there's more data we want to save in the future it's easy to add.

We're now ready to move on to the finale of the questing section!

Finale

Our small RPG is drawing to a close. The player can start the game, receive a quest, and save and load the game progress. We left off the quest with the player defeating a cave monster and finding the gemstone that the major wanted. Let's wrap up the storyline.

In the finale, we'll handle the player returning the gemstone. The player will return to town and give the stone to the major, only to find out the major is possessed by an evil demon who wants the stone for nefarious reasons! We'll launch one last boss battle and then end the game.

The Showdown

Example quests-17 contains the code so far. We've added the following files:

- combat_bg_town.png
- demon.png

Let's add the demon entity and character definitions. Open up EntityDefs.lua and copy in the code from Listing 4.120.

```
gEntities =
{
    -- code omitted
    demon_major =
    {
        texture = "demon.png",
        startFrame = 1,
        width = 176,
        height = 176,
    },
}

gCharacters =
{
    -- code omitted
    demon_major =
    {
```

```

entity = "demon_major",
controller =
{
    "cs_move",
    "cs_run_anim",
    "cs_standby",
    "cs_die_enemy",
    "cs_hurt_enemy",
},
state = "cs_standby",
}
}

```

Listing 4.120: Adding the final demon enemy to the entity defs. In EntityDefs.lua.

Next let's fill in the enemy definition. Copy the code from Listing 4.121.

```

gEnemyDefs =
{
    -- code omitted

    demon_major =
    {
        id = "demon_major",
        stats =
        {
            ["hp_now"] = 1,
            ["hp_max"] = 1,
            ["mp_now"] = 0,
            ["mp_max"] = 0,
            ["strength"] = 20,
            ["speed"] = 13,
            ["intelligence"] = 20,
            ["counter"] = 0,
        },
        name = "Demon Major",
        actions = { "attack" },
        steal_item = 10,
        drop =
        {
            xp = 750,
            gold = {15, 20},
            always = nil,
            chance =
            {

```

```
        { oddment = 1, item = { id = -1 } },
        { oddment = 1, item = { id = 10 } }
    },
},
},
```

Listing 4.121: Let's add the final boss to the enemy definitions. In EnemyDefs.lua.

When we first enter the game, the major paces around the table and ends up at a different location than where he started. We need to make sure the interact trigger matches the major entity location. We do this by adding a flag in the gamestate. Copy the code from Listing 4.122.

```
function GetDefaultGameState()
    return
{
    defeated_cave_drake = false,
    maps =
{
    town =
{
        quest_given = false,
    },
}
```

Listing 4.122: Track if we've been given the quest. In DefaultGameState.lua.

Next let's set the flag to true after the opening cutscene. Copy the code from Listing 4.123.

```
local intro =
{
    -- code omitted
    SOP.Function(
        function()
            gWorld.mGameState.maps.town.quest_given = true
            gWorld.mGold = gWorld.mGold + 500
        end),
    SOP.Wait(0.1),
    SOP.HandOff("handin")
}
```

Listing 4.123: Setting the quest given flag in the opening cutscene. In main.lua.

At the end of the cutscene we give the player 500 gold as the first half of the major's reward.

Next let's add a RunScript action to the town map on_wake table that repositions the major if the quest_given flag is true. Copy the code from Listing 4.124.

```
return
{
    -- code omitted
    on_wake =
    {
        -- code omitted, needs to go after AddNPCs
        {
            id = "RunScript",
            params = { MoveMajor }
        },
    }
}
```

Listing 4.124: Add a function to MoveMajor. In map_town.lua.

The major is added to the town by an action in the on_wake table. The major can't be repositioned until he exists on the map, therefore we add the MoveMajor action at the end of the on_wake table.

The MoveMajor script lives in the CreateTownMap function. Copy the code from Listing 4.125.

```
function CreateTownMap(state)

    -- code omitted

    local function MoveMajor(map)

        if townState.quest_given then
            local removeAction = Actions.RemoveNPC(map, "major")
            removeAction()
            local addAction = Actions.AddNPC(map,
            {
                def = "npc_major",
                id = "major",
                x = 5,
                y = 5,
                layer = 1
            })
            addAction()
        end
    end

```

Listing 4.125: Adding function to correctly position the major. In map_town.lua.

The MoveMajor function is called when the map loads. It checks the game state to see if the major has given out the quest. If the major hasn't given the quest, it means we're right at the start of the game, ready to run the intro cutscene and can leave the major alone. If the quest has been given, then the map is being loaded at a later point, and the major needs moving to the same spot as his trigger. The simplest way to move the major is to remove him from the map and then add him back using the RemoveNPC and AddNPC actions.

With the major in the correct location, let's update the TalkToMajor function to trigger the final encounter. Copy the code from Listing 4.126.

```
local function TalkToMajor(map, trigger, entity, x, y, layer)

    local gemstoneId = 14

    if gWorld:HasKey(gemstoneId) then

        local combatDef =
        {
            background = "combat_bg_town.png",
            enemy = { "demon_major" },
            canFlee = false,
        }

        local sayDef = { textScale = 1.5 }
        local bossTalk =
        [
            SOP.BlackScreen("blackscreen", 0),
            SOP.Say("handin", "hero",
                "Gemstone removed.", 1.75, sayDef),
            SOP.Wait(0.3),
            SOP.Say("handin", "major",
                "Ah, the gemstone. Thank you.", 3, sayDef),
            SOP.Wait(0.2),
            SOP.Say("handin", "major",
                "Fools!", 3, sayDef),
            SOP.Wait(0.2),
            SOP.Say("handin", "major",
                "How long I've waited.", 3, sayDef),
            SOP.Wait(0.2),
            SOP.Say("handin", "major",
                "Killing the major, taking his form...", 3, sayDef),
            SOP.Wait(0.2),
            SOP.Say("handin", "major",
                "The major has been removed.", 3, sayDef)
        ]
    end
end
```

```

        "To return to the corporeal realm...", 3, sayDef),
SOP.Wait(0.2),
SOP.Say("handin", "major",
        "and FEED!", 3, sayDef),
SOP.Wait(0.2),
SOP.RunAction("Combat", { map, combatDef }),
SOP.NoBlock(
    SOP.Say("handin", "major", "Noooooo!", 1.5, sayDef)
),
SOP.FadeOutChar("handin", "major", 1.75),
SOP.Wait(0.2),
SOP.FadeInScreen(),
SOP.Function(
    function()
        gStack = StateStack>Create()
        gStack:Push(GameOverState>Create(gStack, gWorld, true))
    end)
}

local storyboard = Storyboard>Create(gStack, bossTalk, true)
gStack:Push(storyboard)
else
    local message =
        "Go to the mine and get the gem. Then we can talk."
    local action = Actions.ShortText(map, message)
    action(trigger, entity, x, y, layer)
end
end

```

Listing 4.126: Triggering the final battle with the major. In map_town.lua.

In Listing 4.126 we check to see if the player is carrying the gemstone. If the player doesn't have the stone, then the major repeats the quest details. If the player does have the gemstone we play a cutscene and start combat.

The final battle is against the demon possessing the major. In the combat definition we add a single enemy, "demon_major", that we defined previously. It's a boss fight so canFlee is set to false. We use the "combat_bg_town.png" texture for the town combat background. You can see the combat state in action in Figure 4.34.



Figure 4.34: The final showdown with the demon.

The cutscene begins by informing the player that they've given the gemstone to the major. Then the major demon monologues a little and we trigger the combat. Combat occurs halfway through the cutscene. The combat state is pushed on top of the stack, in effect pausing the cutscene until the combat finishes. If the player loses, the combat stack is destroyed and replaced when the player starts a new game or continues. If the player wins the cutscene continues.

After the player wins the major screams "Nooo" and fades out. Then we fade to black and go to the game over screen.

Final Game Over

We're nearly at end of our first fully featured game. All that's left is to update game over state. Copy the code from Listing 4.127.

```
GameOverState = {}
GameOverState.__index = GameOverState
function GameOverState:Create(stack, world, won)
    local this =
    {
        mWorld = world,
        mStack = stack,
```

```

        mWon = false,
    }

    if won == true then
        this.mWon = true
    end

    setmetatable(this, self)

    if not this.mWon then

        this.mShowContinue = Save:DoesExist()
        local data = {}
        if this.mShowContinue then
            table.insert(data, "Continue")
        end
        table.insert(data, "New Game")

        this.mMenu = Selection>Create
        {
            data = data,
            spacingY = 36,
            OnSelection = function(...) this:OnSelect(...) end,
        }
        this.mMenu:SetPosition(-this.mMenu:GetWidth(), 0)
    end

    return this
end

```

Listing 4.127: Updated constructor for the game over state. In GameOverState.lua.

The GameOverState handles two cases: when the player wins the game and when the player dies. It takes in three parameters: the stack, the world, and a boolean won which is set to true if the player has won the game.

If the player hasn't won the game, we display a selection menu. The selection menu always contains the "New Game" option, and if there's a save file it contains the "Continue" option too.

Let's update the rest of the functions.

```

function GameOverState:Enter()
    CaptionStyles["title"].color:SetW(1)
    if self.mWon then
        CaptionStyles["subtitle"].color:SetW(1)
    end

```

```

        end
    end

    function GameOverState:HandleInput()
        if not self.mWon then
            self.mMenu:HandleInput()
        end
    end

    function GameOverState:Render(renderer)

        renderer:DrawRect2d(System.ScreenTopLeft(),
                            System.ScreenBottomRight(),
                            Vector.Create(0,0,0,1))

        if self.mWon then
            CaptionStyles["title"]:Render(renderer,
                "The End")

            CaptionStyles["subtitle"]:Render(renderer,
                "Want to find out what happens next? Write it!")
        else
            CaptionStyles["title"]:Render(renderer,
                "Game Over")

            renderer:AlignText("left", "center")
            self.mMenu:Render(renderer)
        end
    end

    function GameOverState:OnSelect(index, data)

        -- reset text
        gRenderer:SetFont(CaptionStyles["default"].font)
        gRenderer:ScaleText(CaptionStyles["default"].scale)

        if not self.mShowContinue then
            index = index + 1
        end

        if index == 1 then
            Save:Load()
        elseif index == 2 then
            local storyboard = SetupNewGame()
            gStack:Push(storyboard)
        end
    end

```

```
    storyboard:Update(0)
end
end
```

Listing 4.128: Updating the GameOver state. In GameOverState.lua.

In the Enter function, the subtitle caption is only set to visible if the player wins the game.

The Render function draws a black rectangle over the entire screen. If the player wins, then “The End” is displayed with the subtitle “Want to find out what happens next? Write it!”, as shown in Figure 4.35. If the player hasn’t won, then “Game Over” is displayed with a selection menu asking the player what they want to do next.



Figure 4.35: The final game over screen.

In the OnSelect callback the fonts are reset, then the index is altered if “Continue” is displayed. If the player selects “Continue” we load the last saved game. If the player selects “New Game” we destroy the stack and start a new game with the opening cutscene. There’s a call to Update just to make sure the storyboard’s update is called before it’s rendered for the first time.

That’s it! Run the code in `quests-17-solution` and you’ll be able to play a fully-featured JRPG you’ve written. Pretty amazing!

Closing

Together we've gone from rendering a sprite on screen, to a full, if small, JRPG game. We've got maps to explore, monsters to defeat, loot to sell, levels to gain, abilities to unlock; you should be proud! It's been tough but you've come through and hopefully enjoyed the journey! Legions start game projects but few finish anything. You've proven you have the ability to finish, and it's up to you to decide what project you want to tackle next.

You've learned how to structure a complicated piece of software and crafty ways to use state machines for animation and game flow. How to use Lua to help script cutscenes. How to break down a battle system into extendable pieces. All these skills can be used not just for a 2D JRPG but for many different domains. Want to create a 3D Final Fantasy 7 style game? With the code we've written and a 3D engine, such as Unity, you're more than halfway there! You can draw upon this pool of skills for all your future projects.

The small games we've made are fully functional but rough. A commercial game needs more content and polish. If you're going down this route, it's worth spending time making content creation easier. Content means maps, special effects, abilities, and so on. The faster you can iterate on content the better. In the book we leveraged Tiled, but the more tools you can build into your game engine, the better. That said, you can create a full RPG just repeating what we've done in the book and I'm eager to see what people create.

Keep in touch! If you have any questions, want to share a project or just say "Hi!", you can send me an email at dan@howtomakeanrpg.com. For updates on the book and related RPG material, please join the mailing list (if you haven't already) at http://howtomakeanrpg.com/mailing_list/. Additional material is available on the website <http://howtomakeanrpg.com/>, and I'll send brief updates and news via Twitter @howtomakeanrpg.

"It's a dangerous business, Frodo, going out your door. You step onto the road, and if you don't keep your feet, there's no knowing where you might be swept off to." — J.R.R. Tolkien, The Lord of the Rings

Good luck!

Daniel Schuller

Chapter 5

Appendix

The appendices dig a little deeper into the Lua language and better explain some of the included code files.

Appendix A - Classes in Lua

Different programming languages have different styles of writing and thinking about computer programs. These different styles are known as *paradigms*. A language like Haskell is a functional language, a language like C++ is an object-orientated language and Prolog is a logic programming language. Lua is a *multi-paradigm* programming language; it can use any of these paradigms but you, the programmer, must do a little of the leg work. To write our game code we use a light object-orientated approach.

Lua has no classes but that's ok; we'll make our own simple class system.

Basically a class is a template used to stamp out unique instances of that class called objects. For instance we might have a Dog class. We can use this class to create particular dog instances such as: Spot, Rover and Paws. Each of these dogs shares a Bark function but they differ in name.

Listing 6.1 shows how we'd write a Dog class in Lua.

```
Dog = {}      -- table for the class
Dog.__index = Dog -- tells object to check the Dog class table for fields
function Dog:Create()

    -- This is a unique dog instance
    -- It's just an empty table at the moment
    local this =
    {
```

```

}

-- Self is a special keyword
-- Here it refers to Dog class
-- This line of code is equivalent to
--     setmetatable(this, Dog)
--

-- Because of the __index field this line means:
--

--     if you can't find a entry in the this table,
--     look in the Dog table for it.
--

setmetatable(this, self)
return this
end

function Dog:Bark()
    print("Woof")
end

-- Let's use it

spot = Dog>Create()
rover = Dog>Create()
paws = Dog>Create()

paws:Bark() -- > Woof

```

Listing 6.1: An example Dog class in Lua.

In Listing 6.1 we create a table called Dog this is the class. We create a new instance of the Dog class by calling the Create function. Each dog object has access to all the functions the Dog class defines.

That's all these is to our Lua class system. Let's modify the Dog class, as shown in Listing 6.2, so each dog has a unique name.

```

Dog = {}
Dog.__index = Dog
function Dog>Create(dogName)
    local this =
    {
        name = dogName
    }
    setmetatable(this, self)

```

```

        return this
end

function Dog:Bark()
    print(self.name .. ": Bark!")
end

spot = Dog:Create("spot")
spot:Bark() --> "spot: Bark!"

paws = Dog:Create("paws")
paws:Bark() --> "paws: Bark!"

```

Listing 6.2: Naming Dogs.

In Listing 6.2 we give each dog a name that's stored in it's unique this table. When we call the Bark function on spot, spot first checks it's local this table for Bark and doesn't find it, so it then checks the Dog table for Bark, finds the function and calls it. The Bark function looks for the name variable in the self object which is the spot dog instance. The name here equals "spot" so "spot: Bark!" is printed out.

This is as much of a class system as we need to write games in Lua. When creating classes in Lua, as a rule of thumb, I'll create each class in it's own file that shares the class name. In the case of the Dog class, I'd put the code in a file called Dog.lua. This code is available in the examples folder as **lua_classes**.

Appendix B - State Machines

State Machines are a fundamental building block of game development and we'll be using them often.

Players interact with a game in many different ways. For instance, the control scheme for the options menu is very different than on the world map. We interact with these two *states* differently. A state machine tracks game states and makes it easy for us to swap between them.

States aren't just for menus, we use them wherever we want to make complex code easier to reason about. For instance, during combat a player may have an "idle" state and an "attack" state. In the idle state the player plays an idle animation on loop. When the player presses the attack key, we switch to the attack state. The attack state ignores any further keypresses and runs an attack animation when it's entered. When the attack animation finishes it restores the idle state.

We require each state in the state machine to implement that same set of functions. It can implement more but it *must* implement these to be considered a state. Here are the functions:

- **Render** - The state uses this function to draw anything it wants on the screen.
- **Update** - The state uses this function to update its internal logic.
- **Enter** - This function is called once when we first change to this state.
- **Exit** - This function is called once when we first exit this state.

Don't worry too much about the details at the moment as we'll be using the state machine *a lot* during the course of the book!

The diagram in Figure 6.1 shows a simple idle-attack state machine. Game code can get complicated fast. Imagine your player is attacking, then *while* he's attacking, he's hit by arrow - mid attack! He needs to cancel the attack animation and respond appropriately to taking damage. State Machines handle this kind of transition well for the type of game we're making. They tame code complexity.

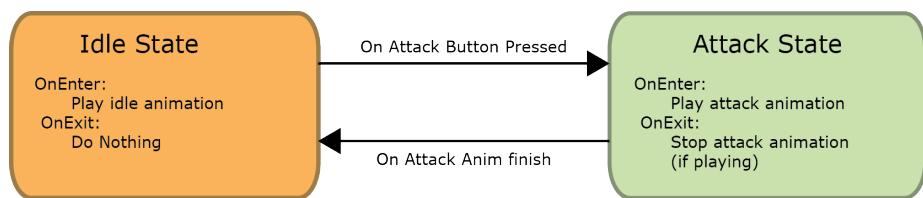


Figure 6.1: A simple state machine with two states: *idle* and *attack*.

The code for the state machine is available in examples/state_machine. But it's also included below in Listing 6.3.

```

StateMachine = {}
StateMachine.__index = StateMachine
function StateMachine:Create(states)
    local this =
    {
        empty =
        {
            Render = function() end,
            Update = function() end,
            Enter = function() end,
            Exit = function() end
        },
        states = states or {}, -- [name] -> [function that returns state]
        current = nil,
    }

    this.current = this.empty
    setmetatable(this, self)

```

```

        return this
end

function StateMachine:Change(stateName, enterParams)
    assert(self.states[stateName]) -- state must exist!
    self.current:Exit()
    self.current = self.states[stateName]()
    self.current:Enter(enterParams)
end

function StateMachine:Update(dt)
    self.current:Update(dt)
end

function StateMachine:Render(renderer)
    self.current:Render(renderer)
end

```

Listing 6.3: Lua code for a State Machine.

Listing 6.3 shows the full state machine code. In the constructor you'll notice there's a state called `empty`; it implements all the required functions of a state but does nothing. It's the default state for the state machine. You can see how the current state is set to the `empty` state at the end of the constructor.

The state machine expects to have its `Update` and `Render` functions called each frame. It passes each call on to the currently focused state. In the default case this means they're passed on to the `empty` state and nothing more is done.

Using a State Machine for Simple Menu

Let's create a very simple menu system using the state machine code. The full code for this example is in `examples/statemachine_menu`. We'll create a menu with two states; "Start Menu" and "Settings Menu". Each state draws its name on the screen and we'll let the player move from one state to the other by pressing keys.

First let's define the `StartMenuState` and `SettingsMenuState` as shown in Listing 6.4.

```

-- Start Menu State
--

StartMenuState = {}
StartMenuState.__index = StartMenuState
function StartMenuState:Create(statemachine)

```

```

local this =
{
    mStates = statemachine,
}
setmetatable(this, self)
return this
end

function StartMenuState:Enter() end
function StartMenuState:Exit() end

function StartMenuState:Update(dt)
    if Keyboard.JustPressed(KEY_S) then
        self.mStates:Change("settings")
    end
end

function StartMenuState:Render(renderer)
    renderer:AlignText("center", "center")
    renderer:DrawText2d(0, 0, "START MENU")
end

-- 
-- Settings State
-- 

SettingsMenuState = {}
SettingsMenuState.__index = SettingsMenuState
function SettingsMenuState>Create(statemachine)
    local this =
    {
        mStates = statemachine,
    }
    setmetatable(this, self)
    return this
end

function SettingsMenuState:Enter() end
function SettingsMenuState:Exit() end

function SettingsMenuState:Update(dt)
    if Keyboard.JustPressed(KEY_B) then
        self.mStates:Change("start")
    end
end

```

```

function SettingsMenuState:Render(renderer)
    renderer:AlignText("center", "center")
    renderer:DrawText2d(0, 0, "SETTINGS MENU")
end

```

Listing 6.4: The StartMenuState and SettingsMenuState implementations.

The two states in Listing 6.4 are really simple. They do nothing in the Enter and Exit functions. The constructor stores the statemachine as `mState`. In the Update loop the `SettingsMenuState` changes to the `StartMenuState` when the B key is pressed. The player can return to the `SettingsMenuState` by pressing the S key. The Render function render the name of the state in the center of the screen.

To use these states they need adding to a state machine. The code for this is shown below.

```

states = StateMachine>Create()
states.mStates =
{
    ["start"] = function() return StartMenuState>Create(states) end,
    ["settings"] = function() return SettingsMenuState>Create(states) end,
}
states:Change("start")

function update()
    states:Update(GetDeltaTime())
    states:Render(gRenderer)
end

```

Listing 6.5: The code to run a state machine.

In Listing 6.5 we create the state machine then add all the states. Each state has a name. We use the state names to switch between the states. By default the active state for the state machine is the *empty* state. We want to start on the "StartMenu" state, so we call `states:Changes("start")` before the update loop. In the update loop we call update and render on the state machine which, in turn, updates and renders the active state.

State Machine Uses

State Machines are used all over the place in games, for handling network states, enemy AI, game state, special effects, etc. Any system that has mutliple distinct stages can be controlled by a state machine. In the book we use them to control character animation and overall gamestate.

StateStack

There are times when we want to track multiple states at once and in this case we'll use a StackStack. The implementation is covered in the user interface section of the book. It's used to keep track of multiple dialog boxes.

Appendix C - Tweening

The word *tween* comes from the word **inbetweening** which is a common technique in computer animation. A tween gradually changes a number from one value to another in a fixed time. For example you could use a tween to change some menu text from white to yellow in, say, 0.03 seconds. The fixed duration of a tween is useful because it lets you predict the future exactly. It's also easy to reverse: if you're half way through moving from white to yellow and you suddenly decide you need to reverse this transition; that's easy to do!

Tweens move a value from one number to another but they don't have to do this at a constant speed. They can start slowly and then finish fast or the opposite and so on. The tween still takes the same amount of time but its speed doesn't have to be linear. This is great for smoothly moving things on screen, pop-ups even controlling cameras.

It's hard to give a sense of how a Tween works with a static image. Therefore I've included a special project called Tween. You can check it out in examples/tween.

We'll use a basic Tween class in this book with a few different tweening functions¹. It's not an exhaustive implementation but it's enough for our needs. The number of tweening functions make the tween code quite big. You can check out the code in example tween in the intro examples.

Example Use of a Tween

Let's say we're booting up our game and we'd like the logo to gradually fade in. That's a pretty good place to use a tween. The update function might look like Listing 6.6.

```
-- From 0 to 1 in 2 seconds
local fadeTween = Tween:Create(0, 1, 2)
local logo = Sprite>Create()
logo:SetTexture(Texture.Find("logo.png"))
local logoColor = Vector>Create(1,1,1,0) -- totally transparent
logo:SetColor(logoColor)

function update()
```

¹The Tween code supplied in the book is based on Robert Penner's tween implementations. You can check out his work here: <http://robertpenner.com/easing/>.

```

-- draw the logo

if fadeTween:IsFinished() then
    -- respond to input and start the game
    return
end

-- Tween is still running
fadeTween:Update(GetDeltaTime())
local v = fade:Value()

logoColor:SetW(v) -- Set the alpha channel to the tween's value
logo:SetColor(logoColor)

end

```

Listing 6.6: Fading in a logo with a tween.

In Listing 6.6 we create a tween that goes from 0 to 1 over 2 seconds. We use the 0 to 1 value to control the logo opacity. 0 means the graphic is totally transparent and 1 means the graphic is totally opaque. After creating the tween we create the logo and a color for it. Then we define the update loop.

If the tween is finished that means the logo is totally opaque and we deal with starting the game. In this case we return from the update function immediately. Otherwise we update the tween, passing in the delta time so it knows how much time has passed since it was last updated. We get the current value of the tween and store it in v. We set the alpha channel of the logo color to v altering its transparency. Then we exit the update loop and we do it again next iteration until the tween is finished.

We might want a smoother fade that starts slow and finishes fast; in that case we can change the tween function from the default, Linear, to EaseInQuad. We can do this with a simple change shown in Listing 6.7.

```

-- code omitted

local fadeTween = Tween:Create(0, 1, 2, Tween.EaseInQuad)

```

Listing 6.7: Changing the tween function.

That's enough of looking at the Tween class we'll be using throughout the book so we'll see many more examples!