CHAPTER 3

Dungeoning

In the previous chapter, we learned how to draw things to the screen and how to create a basic game loop. This chapter will build on that foundation to assemble a basic dungeon for our roguelike, but before delving into some more coding, it is important to understand a bit more about tilemaps and how they are used to assemble a dungeon.

What are tilemaps?

Tilemaps were initially used to save space and memory in games running in older computers and video game consoles. Instead of having a gigantic image with the whole level for a game, the level graphics could be assembled by combining smaller chunks together. This way, the game could only get and draw the chunks needed to display whatever the player was seeing at the moment instead of loading a potentially much larger file into memory. This had the side effect of making much easier to create level editors as the components used by a game level were separate and easy to place in new level designs. They also proved to be a good match for procedural generation as a program could create an algorithmically generated level and then find which tiles it needed to assemble it for display.

For our purposes, we define tilemaps as a grid where we place square-shaped bitmaps in each cell to assemble a dungeon and the necessary game elements. If you ever played a pen and paper role-playing game like *Dungeons & Dragons*, and had to draw a map using graph paper, you'll notice a lot of similarities between that and what our software for this chapter will do.

The dungeon used for the book's roguelike project will eventually have multiple levels. Each level will be a tilemapped grid where walls, rooms, corridors, and other elements are placed to assemble a recognizable RPG-like dungeon. Let's learn how to draw some tiles.

Drawing a tilemap

The source code for this section is under the `chapter-3/example-1-simple-tilemap` folder; you'll need it to follow along. The HTML file is the same as the other samples; it just loads Phaser and our `game.js` file which is where all the interesting bits for this section are actually happening.

Preloading a spritesheet

A spritesheet is an image file that combines many different graphics into a single file. Web games tend to use them because they require a single network transfer to land all the necessary images into the player's computer.

The kind of spritesheet our sample uses is a simple one where all images have the same size and they are placed side by side much like a very well-organized collection of stamps on a page. For example, suppose each image is 10 pixels by 10 pixels and you have ten images in two rows in the spritesheet, that means you have a single image file that is 20 pixels tall by 50 pixels wide with all your images inside.

People often call these images contained in a spritesheet sprite, but you'll also see the same noun being applied to game elements which are moving on the screen, which might be confusing if you're new to game development and are searching online for learning material. I'm going to call them tiles unless they refer to game elements that represent stuff that moves such as the player or monsters. They are all coming from the same file though.

Our spritesheet is from a freely available game art pack by Kenney,[1] and it looks gorgeous (Figure 3-1).

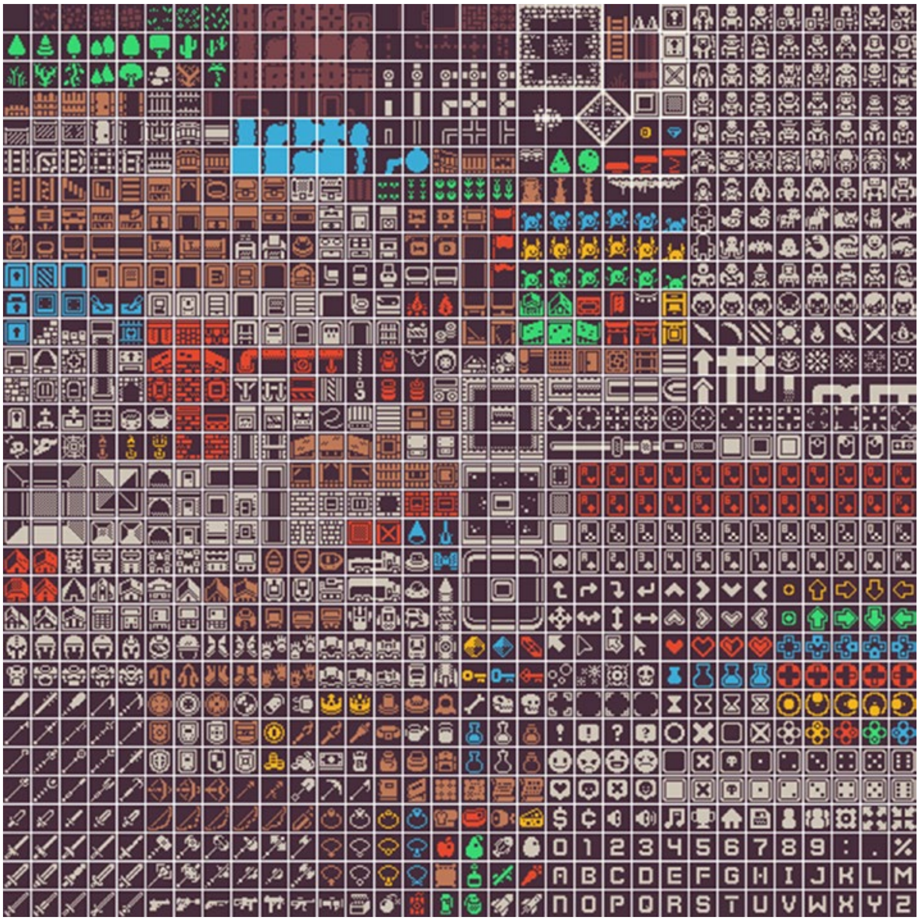


Figure 3-1. *Sample spritesheet*

[1]Kenney 1-bit art pack: www.kenney.nl/assets/bit-pack

As can be seen, there are many different tiles in it, and we'll be able to combine them for a rich roguelike experience. Each image in this spritesheet is a square with 16 pixels on each side. They are separated by gaps of 1 pixel. The source code for preloading the spritesheet needs all this information to be passed. From the `game.js` file, the `preload()` function is

```
preload: function () {
    this.load.spritesheet(
        'tiles',
        'assets/colored.png',
        {
            frameWidth: 16,
            frameHeight: 16,
            spacing: 1
        });
}
```

Much like other `preload()` functions we've seen, we use a function in the `this.load.*` namespace to load the spritesheet. The arguments for that function are the key we'll use to refer to that spritesheet later, the path to the spritesheet image, and a configuration object. There are many optional parameters that can be set in this configuration object; we're just setting the dimensions of each image in the sheet and the gap between them.

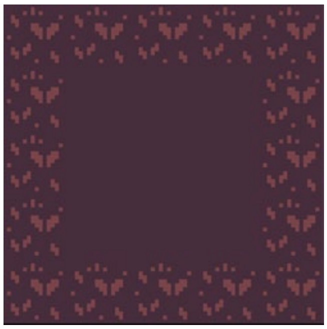
With that in place, we're ready to start drawing some tilemaps, which are grids like graph paper you might have used in school, in which we place tiles in each cell to form our dungeon image. The tiles will come from the spritesheet we saw earlier.

A basic tilemap

To represent the tilemap grid, we'll use a bidimensional array where each element is a number that matches a tile in our spritesheet. A 5x5 dungeon with textured floor on each side and empty floor everywhere else would be represented as

```
let dungeon = [
    [1,1,1,1,1],
    [1,0,0,0,1],
    [1,0,0,0,1],
    [1,0,0,0,1],
    [1,1,1,1,1]
]
```

And this would lead to a dungeon that looks like:



If you check the spritesheet, you'll see that the floor areas are the first image in the sheet and that textured floor on each side is the second image. Since arrays in JavaScript are zero indexed, those become *image 0* and *image 1* from the spritesheet.

There is an important caveat in building the map array. If you use numbers matching the spritesheet indexes and later you change the spritesheet, you'll end up needing to change all the maps or creating some routine to remap those numbers at runtime. It is better to craft a map with numbers that make sense for you and your design and remap those numbers to values that correspond to the desired tiles in the spritesheet just before drawing the map to the screen. This way, if you ever change the spritesheet you're using, you'll only need to change that mapping data.

The `create()` function is where we'll assemble our tilemap. The map used in the sample code for this section uses a 10x10 map.

```
let level = [
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
]
```

We're mapping 0 to mean floor and 1 to mean wall in our map. After it, we need to remap them to the correct values for the tilemap we're using. The floor in the spritesheet is indeed the same value as the value we're using, but for the wall, we're going to use image 554 which is a brick wall.

```
const wall = 554
const floor = 0
level = level.map(r => r.map(t => t == 1 ? wall : floor))
```

To draw that tilemap to the screen, we need to create a configuration object holding the information about it to hold the level data and the dimensions for each tile. Since our tiles are 16 pixels square, we store that value in a constant because we are going to use it multiple times during this sample.

```
const tileSize = 16
const config = {
    data: level,
    tileWidth:  tileSize,
    tileHeight: tileSize,
}
```

Let's use that configuration object to create a tilemap and attach a tileset to it. The tileset is what will match our spritesheet to the tilemap.

```
const map = this.make.tilemap(config);
const tileset = map.addTilesetImage('tiles', 'tiles', ↵
tileSize, tileSize, 0, 1);
```

A tilemap is created by passing the configuration object to `this.make.tilemap()`, and then an inherited function attached to the new `map` is used to add the tileset image to it. You can create all sorts of game objects using functions from `this.make.*`; they are part of the GameObjectCreator class.[2]

That `addTilesetImage`[3] function is receiving a lot of arguments; most of them are optional, but I've noticed that if I don't pass them in this sample, the map doesn't work.

[2]GameObjectCreator class documentation: https://photonstorm.github.io/phaser3-docs/Phaser.GameObjects.GameObjectCreator.html

[3]Documentation for `addTilesetImage`: https://photonstorm.github.io/phaser3-docs/Phaser.Tilemaps.Tilemap.html#addTilesetImage__anchor

Phaser supports many different map data formats beyond the arrays we're using. Many developers use map editors such as Tiled[4] to build their maps. These editors can export the map in rich formats which Phaser can import. Since we're not using such tools, we end up having to specify a lot of data that would be present in the exported map data by hand.

The first argument to `addTilesetImage` is the tileset name as exported in the map data. We don't have a map data as we're not using a map editor. We're passing `tiles` which is the same key we used in the spritesheet loading. The second argument is the key of the cached image from `preload()`, which is `tiles`. If we don't pass this second parameter, it uses the first one as the key to look for the image; it is a bit confusing. We just pass them both to be clear about what we're doing. The rest of the arguments are all data that would be present in the export from a map editor, all of which we need to explicitly pass since we're assembling everything by hand. The third and fourth arguments are the tile dimensions, their width and height. The fifth and sixth arguments are related to the margin around the spritesheet and the gap between images. All values are in pixels.

Phaser's tilemaps can have multiple layers in them, much like placing acetate sheets on top of each other in classic old-school animation or working with Adobe Photoshop layers. The layers can be used to separate game elements into background and foreground layers so that they can appear on top of each other.

There are two types of layers, dynamic and static; the former trades some speed and performance to be able to apply powerful per tile effects. For the tilemap we'll be drawing in this sample, we're going to use a static tilemap since we're not doing any kind of such effects at the moment.

```
const ground = map.createStaticLayer(0, tileset, 0, 0);
```

[4]Tiled map editor: www.mapeditor.org/

Even though we're assigning our static layer to the ground variable, we're not doing anything with it later. It is just to document that that is the ground layer, where the floor and walls are. The first argument to `createStaticLayer()` is the layer ID; this can be either a number or a string and is used by other functions to refer to the layer. We're using 0 because naming it with a string is used only when you're loading maps exported from the Tiled map editor. The second argument is the tileset you created previously.

If you load that sample in your browser, you'll see a tilemap that looks like Figure 3-2.

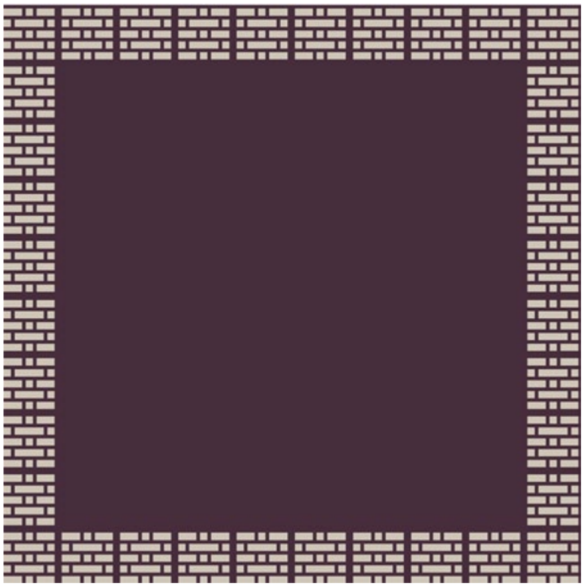


Figure 3-2. *Basic tilemap*

And that is how you draw a tilemap. There was a lot to digest in this section, and it is very beneficial to check the linked documentation for the Phaser functions. Another important exercise to do now is experiment with that map array and different values. Can you place four pillars in that room? A skeleton on the ground?

What about drawing a dungeon? Well, that is our next sample.

A basic dungeon

It is important at this point to understand why I left the procedural generation part of the book to future chapters. Many people think that the foremost feature of a roguelike is procedural generation; to be honest, I'm on that camp as well.

Still, if we leave it to later chapters in the book, we can nail down lots of the basic mechanics of our game and have a better understanding of Phaser and game development by the time we reach those chapters. This way, we can play with procedural generation and appreciate how it alters and enriches the whole game development experience instead of learning both game development basics and procedural generation at the same time.

The next sample is in the `chapter-3/example-2-basic-dungeon` folder, and it is exactly the same code as the previous sample. The only change is that we alter the level array to be a real dungeon-like map instead of a 10 by 10 simple grid. Another small change was to alter the `game.js` included in the HTML to mark it as a *JavaScript module* so that we can use `import` inside it to load the map data from a different file. The level data has been placed outside the main source code because it is massive, which is also the reason why I'm not pasting it in here.

Load it and you'll see a dungeon like Figure 3-3.

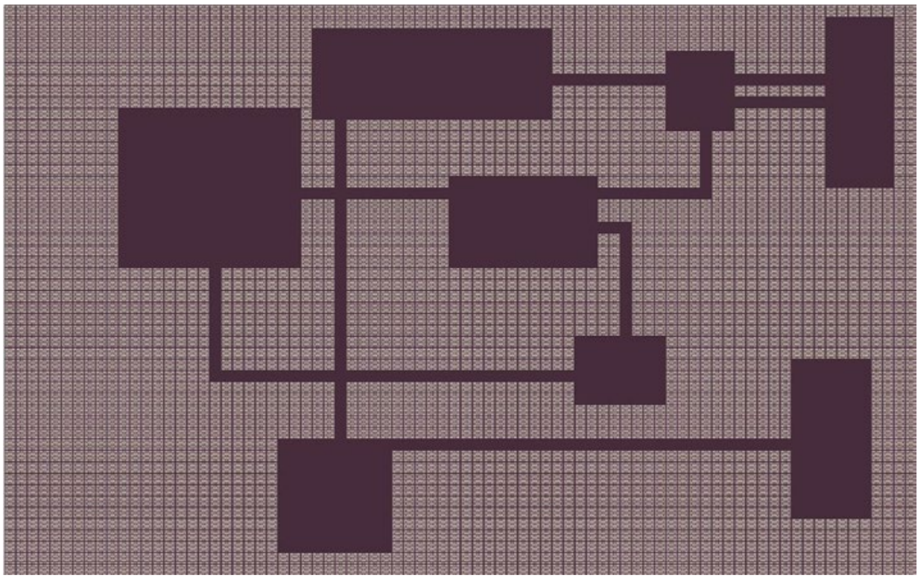


Figure 3-3. *Basic dungeon*

Adding a player character

This sample will be quite familiar as it combines techniques we learned in this and the previous chapters. The folder for it is `chapter-3/example-3-playable-dungeon`. As our sample code grows and reuses parts of the previous samples, I'll only show what changed or what is new. It is best to read these chapters with the source code open in your computer or at least refer to that code later before moving on to the next chapters.

Our player character image is coming from the same spritesheet as the dungeon elements, so we don't need to change the sample `preload()` function to load any extra image file.

In Chapter 2, we built a simple game loop that allowed us to change the position of the displayed text by reading the state of the arrow keys in the `update()` function. A similar approach would suit an action RPG more than the roguelike we're building because those games tend to rely more on quick real-time action than the tactical pondering that turn-based games are usually known for. Phaser is genre agnostic, but it is a bit biased toward real-time action and has many built-in features that support such use case. Being turn based is one of the requirements we placed on our roguelike which means that we need to build our own turn-based mechanics on top of what Phaser offers.

This is the point in our source code where things start becoming more complex in terms of organization and planning. Adding a player character may sound like a simple task, but to accomplish that, we are going to have to implement lots of features that are a part of the core game mechanics. It is a lot of work, but by breaking it down into smaller pieces, we'll be able to handle it. A key step in making all this manageable is to stop throwing everything inside `preload()`, `create()`, and `update()` and start building little modules and classes to help. In this sample, we're going to build some new modules including a turn manager and a dungeon manager and a player character class.

Much of the abstractions and workflows present in this book are coming from pen and paper RPGs and wargames. If you've never played one of those, I think it is beneficial to learn more about them as you read this book. There are many YouTube channels and podcasts that record play sessions, including play sessions with professional actors. Spending some time checking those games out might flesh out the mechanics we're building here in this chapter.

It begins with a dungeon manager

As mentioned earlier, Phaser has a ton of features, but it is not biased toward roguelikes. To create a more ergonomic project, we're going to build auxiliary modules that abstract some of Phaser away so that we can think more in terms of our roguelike than in terms of Phaser.

The main responsibility of our dungeon manager is to load the level and connect the Phaser plumbing necessary to show it on the screen. Some of the code that was in the `create()` function in the previous sample will now be a part of the dungeon module. As our roguelike becomes more complex, this module will accrue more and more functionality. For this sample, we'll use it to load our premade level and create the necessary tilemap, tileset, and dynamic layer for our game. We need to switch to a dynamic layer because the player will be moving on that layer, and in a static layer, it is impossible to change tiles.

In the future, when we start doing procedural generation, this module internals can be changed while the rest of the game remains the same. Part of the refactoring of these routines into it is preparing the groundwork for those future chapters.

These are the responsibilities of the dungeon manager:

- Loading the premade level
- Remapping the numbers used in that level to tiles from our spritesheet
- Creating the tileset, tilemap, and dynamic layer used by our map

The code for the dungeon manager is inside the *dungeon.js*; let's go over it. We're using ES6 modules; if their usage and structure are not clear to you, check out the documentation about them at MDN Web Docs.[5]

We begin by importing the level data:

```
import level from "./level.js"
```

[5]JavaScript modules documentation: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules`

All of the code for the dungeon manager is contained in an object literal called dungeon which we export as the *default export* at the end of the file. Inside the dungeon object, we create a sprites literal object to map human-readable keys to the values used by our spritesheet.

```
sprites: {
    floor: 0,
    wall: 554,
}
```

We'll use those values later in a mapping function much like the chapter-3/example-1-simple-tilemap/ sample did.

An initialize() function is used to handle all the code that was previously handled by the create() inside our previous samples. This function receives the current scene that is calling it as an argument.

```
initialize: function (scene) {
    this.scene = scene

    scene.level = level.map(r => r.map(t => t == 1 ? ↩
    this.sprites.wall : this.sprites.floor))

    const tileSize = 16
    const config = {
        data: scene.level,
        tileWidth: tileSize,
        tileHeight: tileSize,
    }

    const map = scene.make.tilemap(config)
    const tileset = map.addTilesetImage('tiles', 'tiles', 16, ↩
    16, 0, 1)
    this.map = map.createDynamicLayer(0, tileset, 0, 0)
}
```

It first saves a reference to the scene because in the future our game entities will import the dungeon manager and might need to do something to the scene.

As can be seen, the code is a combination of the first and second samples for this chapter as it uses a mapping call to replace the ones in the map with the corresponding wall value from the spritesheet like the first sample, but it is using an externally loaded level data like the second sample.

The rest of the code is almost a copy and paste from the previous one but with some important changes. The dynamic layer created is saved to `dungeon.map`; this will be used by the player character class to inspect the map and decide upon its movement. Before we implement the player class, we must talk about turns and turn management.

Creating a turn manager

There are many ways to code a turn manager. Game developers can overengineer this as much as they want, and part of the charm of a roguelike can actually be the nifty complex ways the turn mechanics play out.

My favorite turn-based computer games all had the same mechanics regarding turn management: each character would have an amount of points to use in their turn, doing actions would cost points, and the turn was over when you were out of points. Phaser will call `update` many times per second so we can't simply block the actions there and handle player input in an imperative way. We'll have to code our own turn manager on top of the frequent calls to `update` to implement the mechanics I outlined at the start of the paragraph. The code will resemble a state machine; each game entity will change their state between having points to spend, being out of points, and refreshing their points. A simple way of implementing a turn manager is simply handling the player movement as in an action game and, after each move, iterating over the other game entities and their actions. Our sample will do something a bit more involved than that without actually going toward a super complex solution; our objective will be to implement mechanics that are similar to the ones outlined earlier.

All our game entities, may they be the player, monsters, or something else we invent in the future, will be new JS classes. These classes will necessarily implement the following methods:

Method	**Explanation**
turn()	Called when it is their turn. Should perform all actions needed for that turn.
over()	Returns a Boolean flagging if the turn for that entity is over or not.
refresh()	Called before a new turn takes place.

At the beginning of a turn, our manager will call `refresh()` on each entity. Then each entity will perform their `turn()`. If `over()` returns `true` for all entities, a new turn begins. The reason behind having an `over()` method is so that if you don't return `true` in it, that entity will get another call to `turn()`. This enables an entity to have many actions per turn in the future, such as creating a monster that moves many tiles in a turn while the player moves just one. This can instill fear in the player quite easily.

The turn manager is in its own module inside `turnManager.js`; it is a singleton and is used by the `update()` code in `game.js`. The code is inside a literal object called `tm` (short for turn manager, makes it easier to paste code in the book because it is shorter). We'll use a JavaScript `Set`[6] to hold

[6]MDN Web Docs documentation for Sets: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

the entities present in the dungeon and provide functions to add and remove entities from this set besides that it mimics the preceding workflow by having `turn()`, `over()`, and `refresh()` functions that call the similarly named functions for each entity present in the set.

Let's go over the code used for managing entities.

```
entities: new Set(),
addEntity: (entity) => tm.entities.add(entity),
removeEntity: (entity) => tm.entities.remove(entity),
```

Using a `Set()` to hold an entity prevents us from adding the same entity twice; these kinds of bugs are sometimes hard to track down so using a data structure that doesn't support adding the same entity more than once makes our code safer. There are two functions, one for adding an entity and another for removing it; we're not using the remove function in this sample, but we'll use it in the future so it was easier to implement it already.

Next, let's implement the code for `turn()` which is responsible for calling the `turn()` method of each entity. As written earlier, we could've opted for a simpler turn manager, but I don't think it would be as fun as this one. What the `turn()` function does is to loop over the entities set, checking if each entity turn is `over()`; if it is not, then it picks that entity and calls its `turn()` method and then *breaks the loop*.

This breaking is important because it enables the `turn()` for that entity to be called again before calling `turn()` on other entities as the loop will run again from the start after the break during the next scene `update()` call, thus allowing our entities to have *multiple actions per turn* which will come in handy once we start building new character classes and monsters.

Phaser runs the scene `update()` cycle very fast; that's how the game gets 60 fps. The problem with that is that if we simply call `tm.turn()` on each `update()`, our game runs too fast. What I mean by that is that if our player presses the down arrow key to move its character down a tile and we’re running at 60 fps, then the key will register as pressed down for multiple iterations of the `update()` cycle, causing the character to sprint in that direction very fast. Our turn handling mechanics are not broken, they’d just be running new turns very fast.

To cope with that, there is a simple debouncing code in the turn manager. It keeps track of when `turn()` last run in milliseconds and only allows it to be called again if 150 milliseconds passed since the last call. It is like putting a break on a fast car so that you can move a bit slower and enjoy the view. We store a property in the `tm` object called `lastCall` and initialize it with the current date; there is also an `interval` property that is set to the amount of milliseconds we want to wait between turns.

```
turn: () => {
    let now = Date.now()
    let limit = tm.lastCall + tm.interval
    if (now > limit) {
        for (let e of tm.entities) {
            if (!e.over()) {
                e.turn()
                break;
            }
        }
        tm.lastCall = Date.now()
    }
}
```

The most interesting part is the breakable loop as mentioned earlier. With that module done, it becomes much easier to implement and understand the player class.

The player class

The player character is a class not because we're thinking about implementing multiple players, but because this will be the pattern used by other game entities, and once we implement other character types, they can inherit from this base class. The code for the player class is inside `player.js`.

The player class, which is the default export of `player.js`, imports the dungeon manager which is a singleton so it has access to the scene and the level data to calculate its movement.

In this game entity, we're using the concept of movement points which is common in wargames and tactical RPG games. Basically, a game entity has a quantity of movement points to use per turn. Each time they move, they spend a movement point. Once the movement points of the entity reach zero, their `turn()` is `over()`. Our player character will start with one movement point and in each `refresh()` will get that point back. In the future, once we add more complexity to the game, we'll have other points as well, but for now that is all we need since this sample is only concerned with movement.

The constructor for our player class receives as argument the coordinates where the player character is placed on the map. In that function, we store the coordinates, create and store the cursor keys used for movement, store a reference to the sprite used for that character, and draw it in the map (which the class has access because it imported the dungeon module).

```
constructor(x, y) {
    this.movementPoints = 1
    this.cursors = dungeon.scene.input.keyboard.createCursorKeys()
    this.x = x
    this.y = y
    this.sprite = 29

    dungeon.map.putTileAt(this.sprite, this.x, this.y)
}
```

Besides storing a bunch of references for future use, there is a function which we haven't seen before: `putTileAt()`.[7] This is from the dynamic layer class and allows us to place a different tile at a given coordinate. We'll use that to simulate the player movement in the map by switching the destination tile sprite with the player character sprite and the previous location back to the floor sprite.

Implementing both `refresh()` and `over()` is easy now that we understand the mechanics.

```
refresh() {
    this.movementPoints = 1
}

over() {
    return this.movementPoints == 0
}
```

Quite straightforward isn't it? The `turn()` function is a bit more involved, and it resembles the code used in Chapter 2 to move the text. At the beginning of the `turn()` function, we store the current player's position and create a Boolean to store if the player moved or not.

```
let oldX = this.x
let oldY = this.y
let moved = false
```

[7]Phaser 3 documentation for putTileAt: https://photonstorm.github.io/phaser3-docs/Phaser.Tilemaps.DynamicTilemapLayer.html#putTileAt__anchor

Then, check if the player has movement points left; check each cursor key and update coordinates as needed.

```
if (this.movementPoints > 0) {
    if (this.cursors.left.isDown) {
        this.x -= 1
        moved = true
    }

    if (this.cursors.right.isDown) {
        this.x += 1
        moved = true
    }

    if (this.cursors.up.isDown) {
        this.y -= 1
        moved = true
    }

    if (this.cursors.down.isDown) {
        this.y += 1
        moved = true
    }

    if (moved) {
        this.movementPoints -= 1
    }
}
```

If moved is true, subtract a point from the movement points. This is what will eventually cause over() to return true and end the player's turn. By the end of that part of the code, the coordinates for the player character will be at the new position, but the screen is not updated yet, so we can actually revert the movement if the player is actually moving into a wall.

```
let tileAtDestination = dungeon.map.getTileAt(this.x, this.y)
if (tileAtDestination.index == dungeon.sprites.wall) {
    this.x = oldX
    this.y = oldY
}
```

The getTileAt()[8] function is the inverse function of putTileAt() which we've seen before. Finally, it is just a matter of drawing the player character in the new position and flipping the tile in the old position to a floor tile.

```
if (this.x !== oldX || this.y !== oldY) {
    dungeon.map.putTileAt(this.sprite, this.x, this.y)
    dungeon.map.putTileAt(dungeon.sprites.floor, oldX, oldY)
}
```

The player class is now complete. It doesn't do much except handling movement, but that is our current project. It is time to integrate all of this back into the scene.

[8]Phaser 3 documentation for getTileAt(): https://photonstorm.github.io/phaser3-docs/Phaser.Tilemaps.DynamicTilemapLayer.html#getTileAt__anchor

Updating the scene

The `game.js` file for this sample is much simpler than the previous ones since we extracted most of the logic contained in them into the modules we just implemented. It is quite similar to the previous sample, but at the top, we start by importing our new modules and the player class.

```
import dungeon from "./dungeon.js"
import tm from "./turnManager.js"
import PlayerCharacter from "./player.js"
```

Compared to the previous sample, the only changes are to the `create()` and `update()` functions. The `preload()` remains the same and just loads the spritesheet.

Look at how streamlined the new `create()` function is:

```
create: function () {
    dungeon.initialize(this)
    let player = new PlayerCharacter(15, 15)
    tm.addEntity(player)
}
```

It just initializes the dungeon manager passing the scene itself, and then it creates a new player instance and adds it to the turn manager.

The `update()` function is also quite simple. It checks to see if the turns are `over()`; if they are, then all entities are `refresh()` and then `turn()` is called over and over again.

```
update: function () {
    if (tm.over()) {
        tm.refresh()
    }
    tm.turn()
}
```

When you load that sample in the browser, you'll see a dungeon with a player character in the room at the top-left corner, just like Figure 3-4. You can use the arrow keys to move the character around. Holding a key pressed will slowly move the character in that direction, thanks to our debouncing code. You'll collide with walls, and you can save some movement points by moving diagonally by pressing both arrow keys at the same time as the `turn()` code checks for all of the inputs in a single iteration.

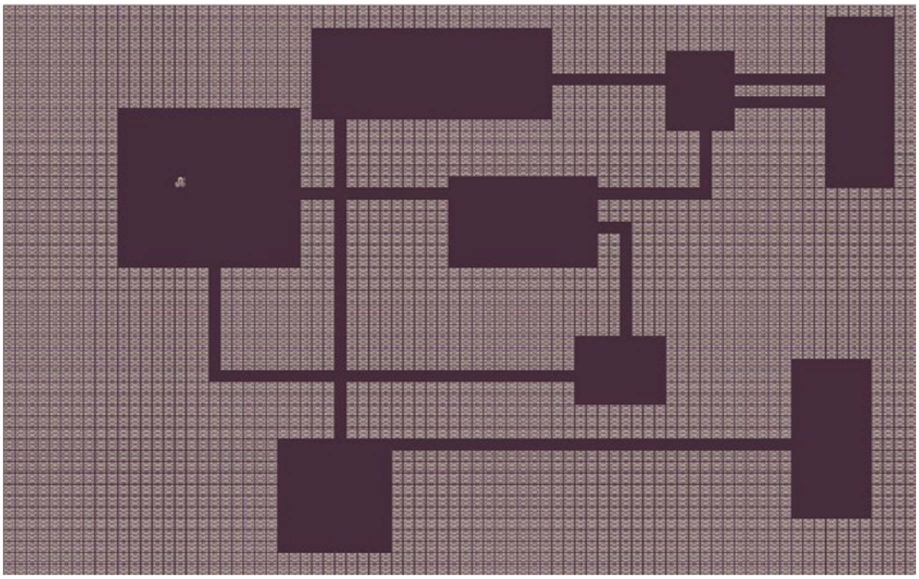


Figure 3-4. *Playable dungeon*

Exercise

Can you alter the player class so that it has more moves per turn? Can you make the player dig through walls?

Summary

This chapter finally started us in the journey of roguelike development. You worked hard and now you have both a dungeon and a moving character. Let us recap what we've learned:

- How to use Phaser scene lifecycle functions such `preload()`, `create()`, and `update()` in a roguelike development setting
- How to implement turn-based mechanics on top of a genre-agnostic game development library
- What tilemaps are and how to use them

Study and get to know the final sample well; we'll be using and improving upon the dungeon and the turn manager modules and the player class because in the next chapter, we're adding monsters.