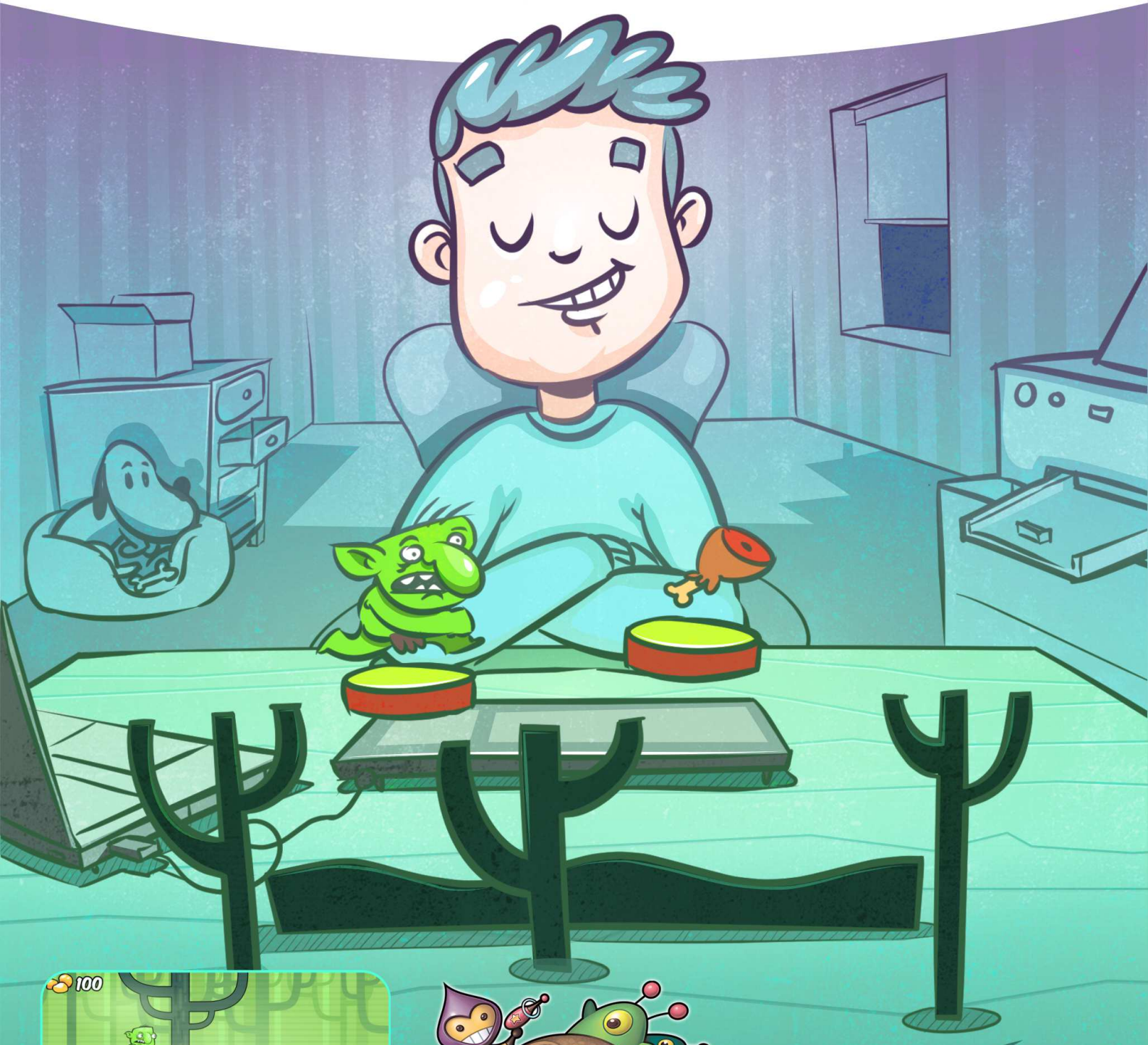
Create procedural ENDLESS RUNNER in PHASER with TypeScript



Tomáš Rychnovský

Create procedural
ENDLESS RUNNER
in **PHASER** with **Type**Script

By Tomáš Rychnovský

Create procedural endless runner in Phaser with TypeScript

by Tomáš Rychnovský

Table of Contents

1. Introduction 9

1.1 Goblin Run 9

1.2 Target audience 10

1.3 TypeScript 10

1.4 Accompanying files 12

1.5 Credits 12

Summary 12

2. Setting up the project 13

2.1. Project structure 13

2.2 Adding files 15

2.3 Combining output 15

2.4 Compiling from command line 18

Summary 18

3. Adding states 19

3.1 Creating Game 19

3.2 States 20

Summary 23

4. Calculating jump tables 24

4.1 Jump analysis 24

4.2 Generator parameters 25

4.3 Jump velocities calculation 26

4.4 Visual debug output 28

4.5 Jump table calculation 30

4.6 Calculation results 37

4.7 Jump table analysis 40

Summary 42

5. Generating platforms 43

5.1 Adding test block 43

5.2 Adding platform bounds 43

5.3 Level piece class 44

5.4 Generic Pool class 45
5.5 Generator class 46
5.6 Main Layer 50
5.7 Observing results 55
Summary 56
6. Adding player 57
6.1 Loading player assets 57
6.2 Adding Player class 57
6.3 Jumping 58
6.4 Running game 62
Summary 63
7. Enhancing generator 64
7.1 Difficulty parameters 64
7.2 Difficulty class 65
7.3 Enhancing Generator 66
7.4 Pieces queue 68
7.5 Changes in MainLayer 71
7.6 Running game 73
Summary 74
8. Adding tile graphics 75
8.1 Atlas 75
8.2 Tiles 75
8.3 Block definitions 77
8.4 Generator signals 79
8.5 Block or platform? 80
8.6 Placing tiles 82
8.7 Running game 84
Summary 85
9. Animating player 86
9.1 Skeletal animation 86
9.2 Spriter Player for Phaser 87
9.3 Loading animation 88
9.4 Adjusting player 88

9.5 Additional changes 92
9.6 Running game 93
Summary 94
10. Background layers 95
10.1 Background layout 95
10.2 Background assets 96
10.3 Background class 96
10.4 Using background 100
Summary 101
11. Spikes 102
11.1 Spikes idea 102
11.2 New Piece property 102
11.3 New parameters 103
11.4 Increasing difficulty 104
11.5 Generating spikes 104
11.6 Spikes animation 105
11.7 Defining spikes block 106
11.8 Adding spikes to main layer 106
11.9 Checking for overlaps 110
Summary 112
12. Bonus jump 113
12.1 Yet another Piece property 113
12.2 New parameters and difficulty 113
12.3 Changes in Generator 115
12.4 Animation and block definition 117
12.5 Changes into MainLayer 117
12.6 Bonus jump overlaps 118
Summary 120
13. Gold 121
13.1 Idea 121
13.2 Block definition 121
13.3 Placing gold on platforms 121
13.4 Collisions with gold 123

13.5 Font 124
13.6 Score UI element 126
13.7 Adding score to game 127
Summary 128
14. Particles 129
14.1 Mud splash 129
14.2 Dust 130
14.3 Gold and bonus jump particles 133
Summary 134
15. Shadow 136
15.1 Shadow class 136
15.2 Adding shadow 138
Summary 139
16. Record 140
16.1 Preferences 140
16.2 Record class 141
16.3 Adding record to main layer 142
16.4 Checking for new record 142
Summary 144
17. Sounds and music 145
17.1 Audiosprite 145
17.2 Music 146
17.3 Sounds class 146
17.4 Loading 148
17.5 Playing 149
Summary 151
18. Scaling and orientation 152
18.1 Adding meta tags 152
18.2 Boot state 154
18.3 Handling changes 158
18.4 Running game 158
Summary 159

19. Menu screen 160
19.1 Small adjustments 160
19.2 Menu state 162
19.3 Menu objects 165
19.4 Updating menu screen 170
19.5 Scaling 172
Summary 172
20. Loading screen 173
20.1 Loading font 173
20.2 Constructing bar 174
20.3 Resizing 175
20.4 Updating bar 175
Summary 176
21. Conclusion 177

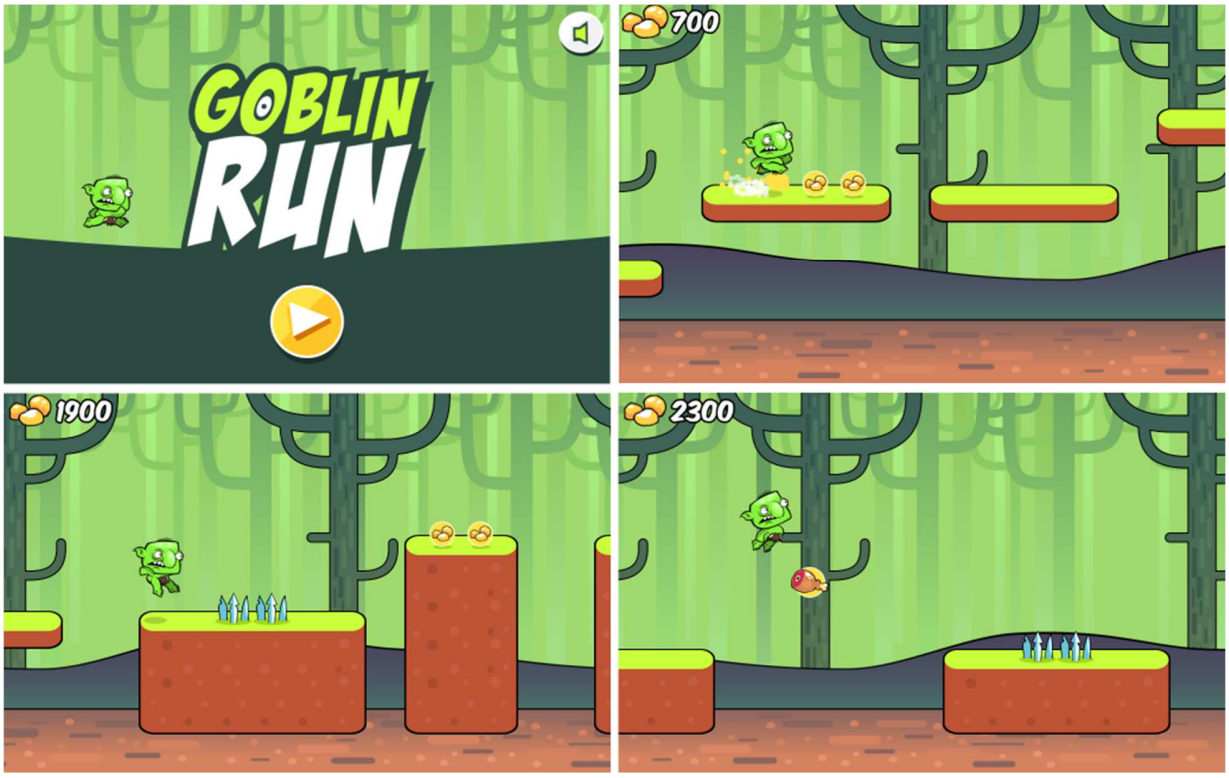
1. Introduction

Phaser is free HTML5 game engine with fast learning curve. You can learn almost every aspect of it from small examples at www.phaser.io/examples, but creating real games needs more. Beside working with engine API, you also need some general game programming knowledges and experience.

There are many books for various engines that learn you how to create game in it, and usually it is engine what is in center of interest. In this book we will focus on complete game – procedural endless runner and find ways how to build it with Phaser. I will try to transfer experience I got while developing it to you. You can take it as one big, complex tutorial.

1.1 Goblin Run

Game we will build in this book is named “Goblin Run”. It is procedurally generated endless runner with platforms build from tiles. It has some additional features like deadly spikes and bonuses for in-air jumps. Game is controlled with one button and jump height is given by its press duration. You can play final game at http://sbc.littlecolor.com/goblinrun.



Along the way you will get some insight into procedural generation and you will build very flexible generator separated from game itself, so it can be easily used in other similar games. Procedural generation has two key points: generated content must be playable and it should look nice. We will cover both.

Goblin character is animated in Spriter, tool for creating skeletal animations. You will learn how to use free Spriter Player for Phaser to play nice and smooth skeletal animations.

You will learn how to spice visual part of game with particle effects and how to play sounds and music.

We will also cover scaling and orientation handling. In the end, our game will be very flexible and will run in window of any size, even in really weird one like this:

1.2 Target audience

This book is not for you if you have no programming basics. It does not explain what is variable or how to make for loop. It also does not substitute Phaser documentation, which can be found at Phaser's site (http://phaser.io/docs/2.6.2/index), so there will not be exhaustive list or description of all possible methods for Phaser.Sprite or any other class we will use.

This book is for you if you have some programming basics. You have some basic knowledge what Phaser is and you probably already looked at some Phaser examples at www. http://phaser.io/examples. Now, you want to make something "bigger", which is beyond those simple how-to examples. Beside Phaser, you are also curious about techniques used to make some games - procedural endless runners in this case. Finally, you are looking for tutorial that does not end with working prototype, but ends with complete game.

1.3 TypeScript

All source code will be written in TypeScript. TypeScript is rapidly gaining popularity and there is no reason to be afraid of it. Unlike CoffeeScript, which is separate language with its own syntax rules, TypeScript is superset of JavaScript and all JavaScript code is automatically valid TypeScript code. If you are using JavaScript ES6, then there is even less differences. All TypeScript code is compiled into JavaScript code during compilation.

Just for example, consider this simple TypeScript class:

```
class MyClass {

    private _position: Phaser.Point = new Phaser.Point();

    public constructor(position: Phaser.Point) {
        this._position.copyFrom(position);
    }
```

```
        public moveBy(x: number, y: number): void {
            this._position.add(x, y);
        }
    }
```

It is compiled into this JavaScript code:

```
    var MyClass = (function () {

        function MyClass(position) {
            this._position = new Phaser.Point();
            this._position.copyFrom(position);
        }

        MyClass.prototype.moveBy = function (x, y) {
            this._position.add(x, y);
        };

        return MyClass;
    } ());
```

Or into this ES6 code:

```
    class MyClass {

        constructor(position) {
            this._position = new Phaser.Point();
            this._position.copyFrom(position);
        }

        moveBy(x, y) {
            this._position.add(x, y);
        }
    }
```

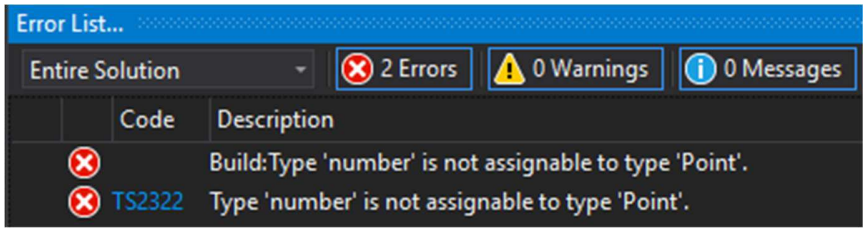
Especially, differences between TypeScript and ES6 are minimal. What TypeScript adds is type checking. All that types after colon like “x: number” tell compiler that only specific type can be used here. If you try to use different one, you get compiler error. Thanks to it, you can catch many errors before you even run your game. Imagine this mistake: you wanted to set x property of Phaser.Point representing positon and accidently, instead of:

```
    this.position.x = 100;
```

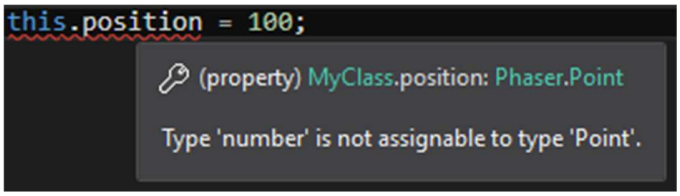
you wrote this:

```
    this.position = 100;
```

This is valid in JavaScript and you will probably get some unwanted behavior, if not crash, during runtime. In TypeScript, if position is Phaser.Point, you will get error during compilation:



Even better, if you use Visual Studio your code will be underlined in red as you write it:



With TypeScript you also get intellisense, which is problematic for JavaScript. Needless to say how big impact all this have on your productivity.

Shortly, code in this book is written in TypeScript, but if you are JavaScript developer, you can easily follow it.

1.4 Accompanying files

This book is accompanied with project folder for every chapter. In it you will find working results for relevant chapter in form of complete Visual studio solution (.sln file). In next chapter we will set up project and it also covers situation if you do not use Visual Studio.

Beside folders for chapters, there is also folder named Resources. It contains all graphics, sounds, music and fonts used in game.

1.5 Credits

Authors of assets used in this book are listed in Resources/License.txt along with other details. Briefly:

- graphics for this book was made by Tomáš Kopecký – see his portfolio at http://www.littlecolor.com/portfolio/,
- music was composed by sawsquarenoise (details in License.txt and also in chapter 17. Sounds and music). Two tracks from album RottenMage SpaceJacked OST were used, both downloaded from http://freemusicarchive.org/music/sawsquarenoise/,
- font KOMIKAX_.ttf is free Komika Poster font by Apostrophic Labs, downloaded from http://www.1001freefonts.com/komika_poster.font.

Thanks must go to Rich Davey, who is author of Phaser and puts incredible effort into its development and also to people in html5gamedevs.com community.

Summary

In this chapter we introduced game we will work on. We also briefly described similarities and differences between TypeScript and JavaScript.

In next chapter we will set up project and add first files into it.

2. Setting up the project

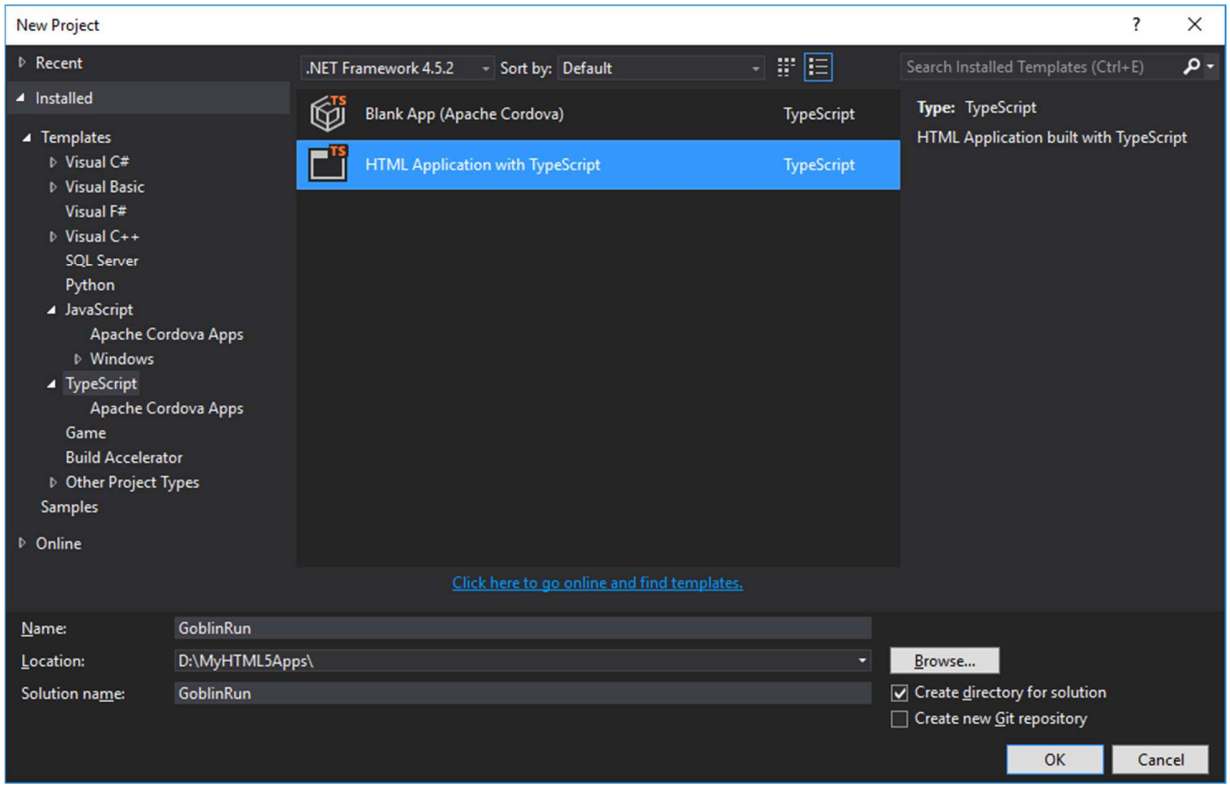
In this chapter we will setup our project. I will use Visual Studio 2015 Community edition (VS) as it is fully featured IDE free for individuals. But I believe you can easily follow with IDE of your choice. You can download VS from https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx.

In the end I will describe how to compile TypeScript project from command line if you do not use any IDE. Do not worry, it is really easy.

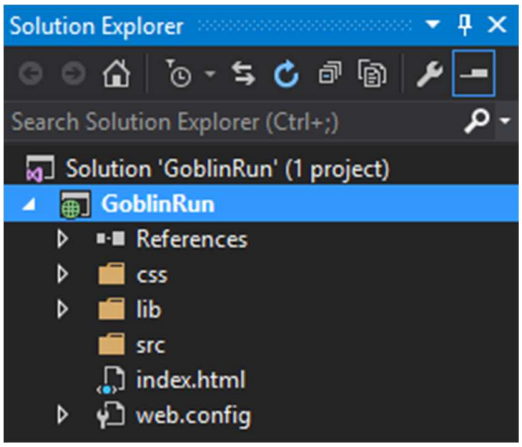
2.1. Project structure

Organizing project from beginning helps to keep overview in later phases of development. While the game we will create is simple, it is still made of many files.

First, create new TypeScript project with File -> New -> Project and choose HTML Application with TypeScript:



Add or delete default files and folders in IDE to match this structure:



On your disk add two more folders – "assets" and "js", so structure of files on disk looks like this:

Name	Date modified	Type	Size
assets	17.06.2016 17:13	File folder	
css	17.06.2016 17:13	File folder	
js	17.06.2016 17:23	File folder	
lib	17.06.2016 17:20	File folder	
src	17.06.2016 18:08	File folder	
GoblinRun.csproj	17.06.2016 17:33	Visual C# Project file	6 KB
GoblinRun.csproj.user	17.06.2016 17:33	Visual Studio Project User Options file	2 KB
index.html	16.06.2016 22:08	Firefox HTML Document	1 KB
web.config	09.02.2016 9:31	XML Configuration File	1 KB
web.Debug.config	09.02.2016 9:31	XML Configuration File	2 KB
web.Release.config	09.02.2016 9:31	XML Configuration File	2 KB

During time, VS will add other folders to the structure. Namely "bin" and "obj", but these are not important for us. Every folder has its role:

- assets – here we will store assets like images, texture atlases, sounds, etc. This folder and its content is part of game if preparing for distribution,
- css – files with css styles are here. Also this folder is part of final distribution,
- js – stores scripts. In our case we will have two of them. First one will be engine (phaser.min.js) and second will be our game (goblinrun.js). Also in this case it is part of game distribution,
- lib – this folder is not part of final game. It is important only for development and we will put TypeScript definition files here. This is not part of distribution,
- src – all our TypeScript scripts will reside here. This folder is not part of final distribution as files here get compiled into single output file (goblinrun.js).

Last important file for us is index.html. Opening this file in browser will load our scripts and launch game.

Just to summarize, if we want to make final distribution, we will take these folders: assets, css, js and index.html file. At that moment there is no mark of TypeScript and you have pure JavaScript app.

2.2 Adding files

Now, we are going to fill some of those folders with files.

First copy these three files with TypeScript definitions from typescript folder in Phaser (either on GitHub: https://github.com/photonstorm/phaser/tree/master/typescript or on disk, if you saved it locally) into lib folder in project:

- phaser.d.ts,
- pixi.d.ts,
- p2.d.ts

Copy file phaser.min.js from Phaser build folder into js folder in project.

Create app.css file in css folder with this content:

```
body {
    background: #000;
    margin: 0px 0px 0px 0px;
}
```

It is very simple style for body of our page with black background and no margin.

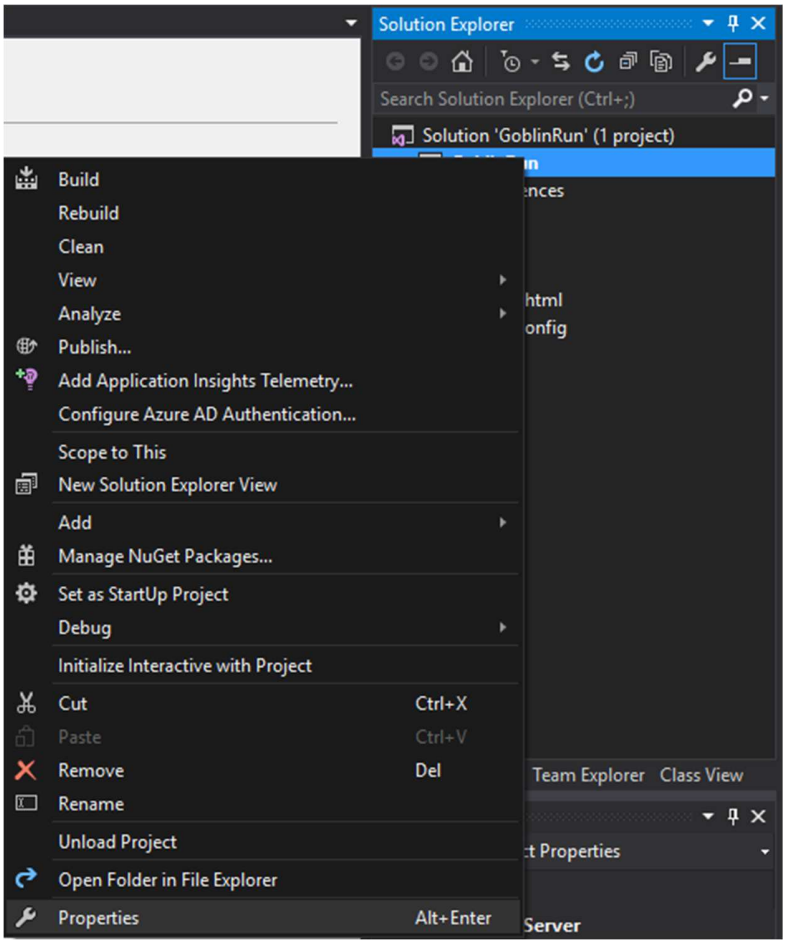
Next, edit index.html, so it looks like this:

```
<!DOCTYPE html>

<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>Goblin Run</title>
    <link rel="stylesheet" href="css/app.css" type="text/css" />

    <script src="js/phaser.min.js"></script>
    <script src="js/goblinrun.js"></script>
</head>
<body>
    <div id="content"></div>
</body>
</html>
```

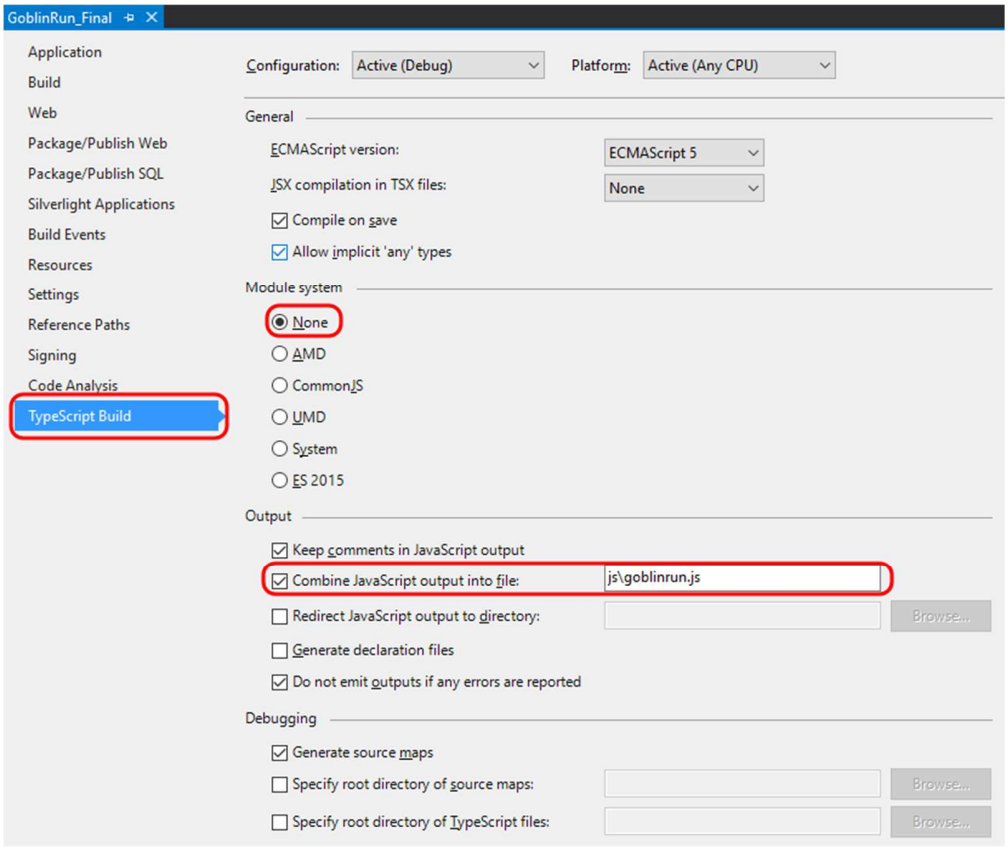
Here we are referencing our style and loading two scripts – engine script (phaser.min.js) and our game (goblinrun.js). Phaser engine will create canvas element with our game in div element with "content" id.

2.3 Combining output

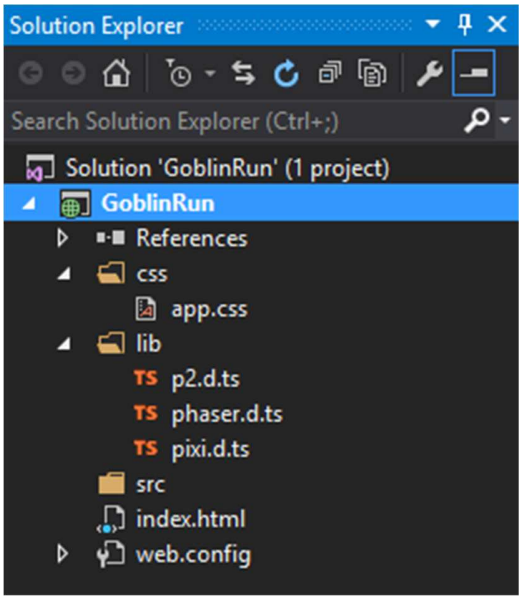
One nice feature VS has, is that it can combine all JavaScript output into one file. We will use this and output whole game into single goblinrun.js file – the file we are loading in our index.html. To set this up, right click on project in Solution Explorer on right and choose Properties in bottom:



In properties choose TypeScript Build. Set module system to None. Check "Combine JavaScript output into file:" and into field next to it write "js\goblinrun.js".



Now, our project is set up. Your project structure in VS should look like this:



2.4 Compiling from command line

If you cannot use any IDE you can compile your TypeScript project from command line. It is easier than it sounds. Just create structure of files and folders the same as described above. Then create file tsconfig.json in the same folder where index.html resides and put this content into it:

```
{
    "compilerOptions": {
        "module": "none",
        "target": "es5",
        "noImplicitAny": false,
        "sourceMap": true,
        "outFile": "js/goblinrun.js"
    },

    "exclude": [
        "node_modules"
    ]
}
```

Locate TypeScript compiler tsc.exe on your disk. On my computer it was installed into c:\Program Files (x86)\Microsoft SDKs\TypeScript\1.8\. If you want to run it just with writing "tsc" in any folder, you will have to add it to your PATH environment variable. Running tsc in folder where is tsconfig.json file located should be enough to compile you TypeScript project. All .ts files in current and all subsequent folders get compiled, and as we set output file to js/goblinrun.js, only one final file will be output into js folder.

Unfortunately, there is currently bug in TypeScript 1.8.* and doing above is how it should work but not how it works now (for reference see for example: https://github.com/Microsoft/TypeScript/issues/8344). Running tsc ends without any error but no output is created. Workaround is simple. Instead of writing just tsc, write:

```
tsc -p <full_path_to_your_tsconfig.json>\tsconfig.json
```

You will find tsconfig.json file in folder for every chapter.

Summary

In this chapter we set up project. We created all necessary folders and filled them with files, that came with Phaser – we put definitions into lib folder and engine itself into js folder. We created page file and its style for our game that will live inside of it. Finally, we told VS we want to combine all Javascript output into single file. From now on, we do not need to touch index.html file and we can focus on game itself.

3. Adding states

During this chapter we will add first source files into project. First, we will create game object itself and then add several states to it.

3.1 Creating Game

Add new file to src folder in project and name it App.ts. This file will be our entry point into game as it defines window.onload function. This function will be called when whole web page and scripts on it are loaded. Second purpose of this file is to hold class with name Global, where we will put some variables and constants we want to access from anywhere in game:

```
namespace GoblinRun {

    export class Global {
        // game
        static game: Phaser.Game;

        // game size
        static GAME_WIDTH: number = 1024;
        static GAME_HEIGHT: number = 640;
    }
}

// ---------------------------------------------------------------------------
window.onload = function () {
    GoblinRun.Global.game = new GoblinRun.Game();
};
```

Notice, that class Global is inside namespace GoblinRun. For accessing variables inside it from within the same namespace, we will just write Global.variable_name. If accessing it from other namespace we would have to write GoblinRun.Global.variable_name. Two variables in class Global define default size of our game. These are in fact constants, but it is not possible to make member variables constant. For constants I will use capital letters.

In window.onload function, we create instance of GoblinRun.Game class, which is derived from Phaser.Game class as we will see soon.

Now, let's add another file into src folder. Its name is Game.ts and contains this code:

```
namespace GoblinRun {

    export class Game extends Phaser.Game {

        // ---------------------------------------------------------------------------
        public constructor() {
            // init game
            super(Global.GAME_WIDTH, Global.GAME_HEIGHT, Phaser.AUTO, "content");

            // states
```

```
            this.state.add("Boot", Boot);
            this.state.add("Preload", Preload);
            this.state.add("Play", Play);

            // start
            this.state.start("Boot");
        }
    }
}
```

Here we are extending Phaser.Game class. In constructor we are calling constructor of super class and then we are doing some small setup, which is adding game states to game and defining, which state to start as first.

When calling superclass constructor, we are passing it our game dimensions, type of renderer and parent element on page – remember, we created in index.html. Phaser will create canvas element of requested dimensions with game and make it sibling of this element. You can check this later in your browser if inspecting page code:

```
<div id="content">
    <canvas style="display: block; width: 1024px; height: 640px; cursor: inherit;" height="640"
width="1024"></canvas>
</div>
```

Type of renderer is Phaser.CANVAS or Phaser.WEBGL or Phaser.AUTO. We are using AUTO, so either CANVAS or WEBGL renderer will be automatically chosen based on device the game runs on.

3.2 States

States can be understood as separate smaller units inside of game, each responsible for some part of it – menu state, game state, help screen state and so on. It is implementation of idea "divide and conquer" – split your game into smaller, better manageable parts. Some states are more abstract and game just goes through it in its lifetime without any special visual feedback for player. For example, it is common to put initialization into Boot state, which does some setting but has no graphical output on screen. Some states are combination of both. State in which assets are loaded is commonly called Preload and its primary purpose is to load game assets, while on the other hand it may give some visual feedback to user about loading progress.

In previous code of Game constructor, we have seen we will have three states: Boot, Preload, Play. Each of them will be separate class derived from Phaser.State. They are added to game with add method of StateManager, which has this signature:

```
add(key: string, state: any, autoStart?: boolean): void;
```

Key is name we want to call it and it is internally added into table with state keys. Second parameter has type any and you can pass instance of Phaser.State (or derived class) to it or you can pass instance of any object to it (Phaser adds game property automatically to it for you) or you can pass class name and Phaser will create new instance of this class. We chose third way and we are passing names of our state classes.

Let's now add states one by one. In src folder create new subfolder States and add file Boot.ts into it with this code:

```
namespace GoblinRun {
```

```
    export class Boot extends Phaser.State {

        // ---------------------------------------------------------------------
        public create() {
            this.game.state.start("Preload");
        }
    }
}
```

It does very little – it only starts next state. But we are just building skeleton of our game.

Add next file Preload.ts and fill it:

```
namespace GoblinRun {

    export class Preload extends Phaser.State {

        // music decoded, ready for game
        private _ready: boolean = false;

        // ---------------------------------------------------------------------
        public preload() {

        }

        // ---------------------------------------------------------------------
        public create() {

        }

        // ---------------------------------------------------------------------
        public update() {
            // run only once
            if (this._ready === false) {
                this._ready = true;

                this.game.state.start("Play");
            }
        }
    }
}
```

We are again doing almost nothing, just starting Play state. But this time we are doing it inside update method. This method is called on every frame and we want to run this piece of code only once. For this reason, there is private variable _ready, that works like barrier and state is changed only once. With this we are preparing for future when we will add sounds and music. These assets may be already loaded but still in decoding process, so we want to check regularly, if decoding is complete. Later, we will make condition check in update more complex to question this.

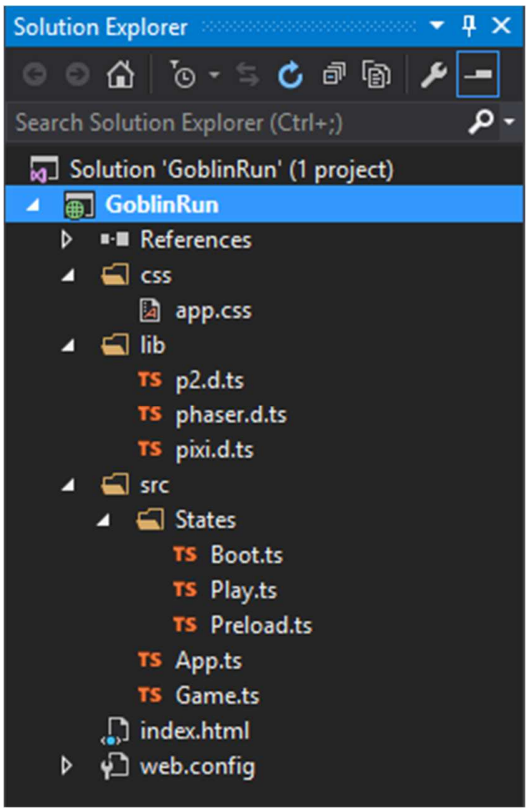
Now, add last state file Play.ts again into src/States:

```
namespace GoblinRun {

    export class Play extends Phaser.State {

        // ---------------------------------------------------------------------
        public create() {
            this.stage.backgroundColor = 0x80FF80;
        }

        // ---------------------------------------------------------------------
        public update() {
```

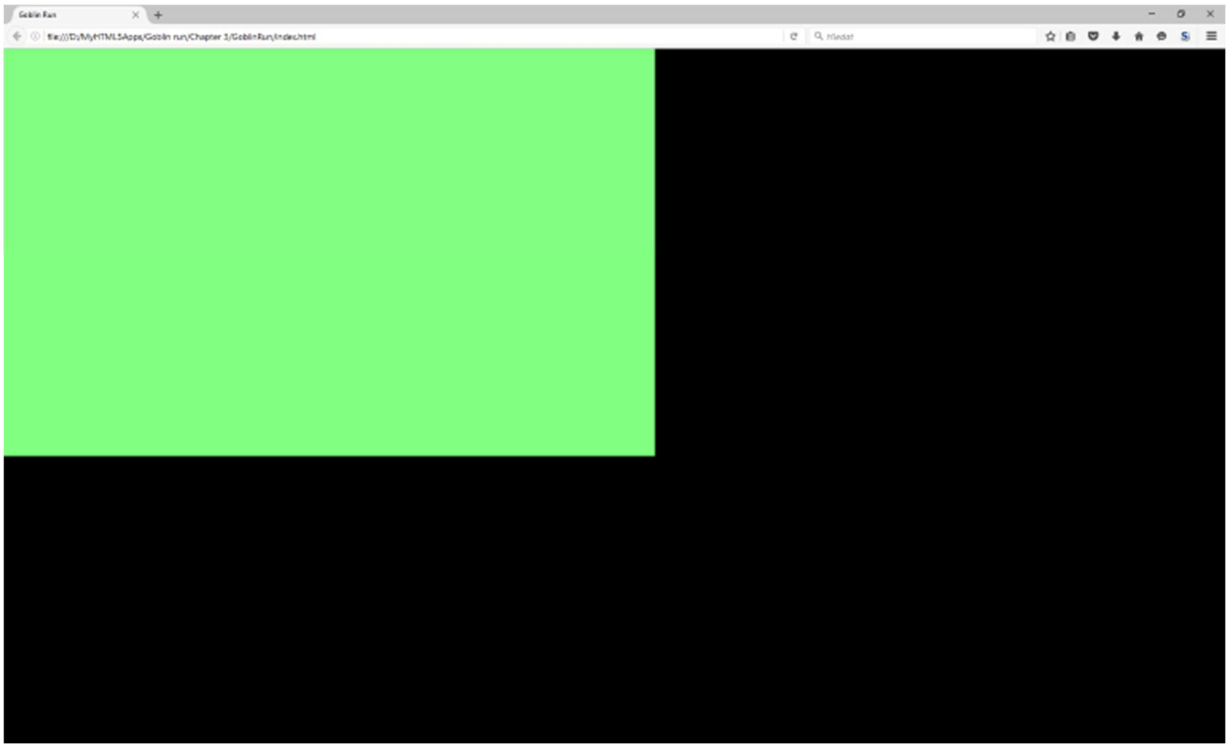
```
        }
    }
}
```

In create method we are changing background color just to give some visual feedback that we passed through all states from Boot to Play.

Your project structure should be now following:



Compile game and run it in browser. You should see green rectangle of size 1024 x 640:



Summary

In this chapter we built basic skeleton for our game. We added game itself and three states. Later we will add more code into each of them.

In next chapter we will start work on generator. First, we will analyze jumps and build jump tables.

4. Calculating jump tables

Platforms in our game are procedurally generated. When generating something procedurally there are two things you have to focus on:

- it is playable,
- it looks nice

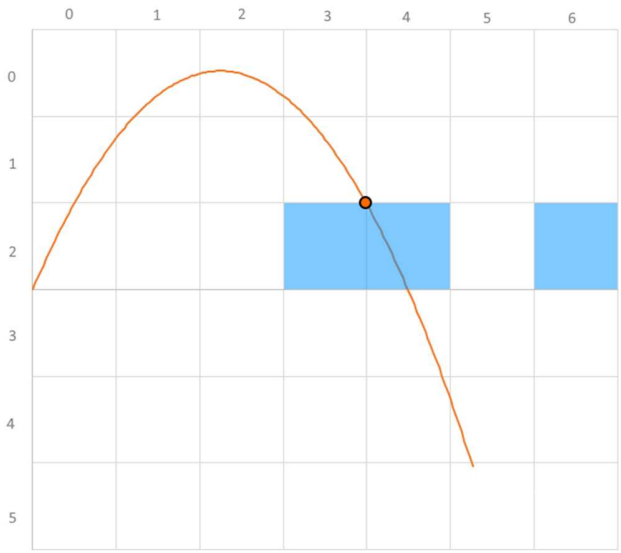
In this chapter we will prepare things to meet first criteria. We will analyze jumps and build jump tables, which are tables with some pre calculated values telling us, where we can safely place next platform so it is reachable.

4.1 Jump analysis

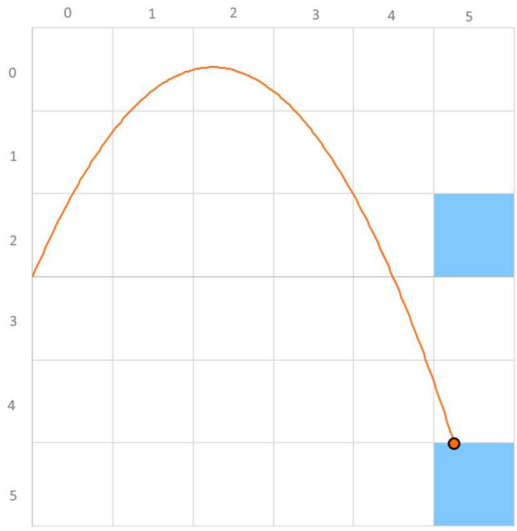
Just to remind what was written in first chapter:

- world is made of tiles,
- player's jump is controlled with one button,
- jump height depends on button press duration.

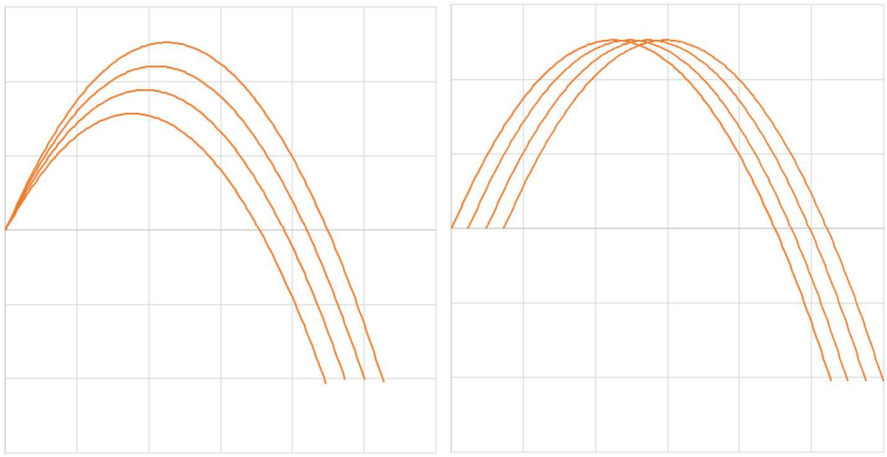
World tiles form square grid. While the tiles are aligned to grid, player is moving smoothly between them. When generating platforms, we cannot put them just randomly as some of them would be unreachable. Look at picture below. Orange curve is jump trajectory of player, blue rectangles are platforms and that small orange dot is landing point. Platform on left is reachable, while more distant platform on right not:



As platforms are not all at one height level, we have to consider another situation. On next image you can see two platforms, both in the same x distance. But, upper platform is unreachable while the bottom one can be reached:



And there is another problem. We said that jump height depends on button press duration, so jump height can be different. And player can also jump from any position on tile, not only from very beginning. Then different distances are reached:



We have to take all this into consideration when calculating jump tables.

4.2 Generator parameters

Before we delve deeper into jump table calculations, we will add class that keeps all generator parameters at one place. Create new folder Generator under src folder and create file Parameters.ts in it. Put following code into it.

```
namespace Generator {
```

```
    export class Parameters {

        // grid
        public static GRID_HEIGHT = 10;
        public static CELL_SIZE = 64;
        public static CELL_STEPS = 4;

        // gravity
        public static GRAVITY = 2400;

        // player body dimensions
        public static PLAYER_BODY_WIDTH = 30;
        public static PLAYER_BODY_HEIGHT = 90;

        // jump height params
        public static HEIGHT_MIN = Parameters.CELL_SIZE * 0.75;
        public static HEIGHT_MAX = Parameters.CELL_SIZE * 2.90;
        public static HEIGHT_STEPS = 4;

        // horizontal speed
        public static VELOCITY_X = 300;
    }
}
```

We are creating separate namespace for our generator to make it as independent on rest of the game as possible. Here is meaning of individual parameters:

- GRID_HEIGHT – height of grid in tiles – we will call them cells in generator to distinguish from visual tiles later in game. In fact, this tells us what is maximal depth of jump we will examine,
- CELL_SIZE - is our cell size in pixels. Our cells will be squares 64x64,
- CELL_STEPS – player can start jump from any point on cell. We will sample it down into four positons (beginning, first quarter, half, three quarters),
- GRAVITY – gravity force. This value will be later passed to physics engine,
- PLAYER_BODY_WIDTH and PLAYER_BODY_HEIGHT – dimensions of player body. We will use this when setting up physics for player,
- HEIGHT_MIN and HEIGHT_MAX – minimum and maximum height of jump. Jump height depends on button press duration but these values are bounds,
- HEIGHT_STEPS – similar to sampling position on cell, we will also sample only few jump heights in our calculations,
- VELOCITY_X – speed of movement of our player. Higher speed means player can jump further.

Parameters we just added will became more clear in next parts when we use them.

4.3 Jump velocities calculation

Now, add another file into Generator folder with name JumpTables.ts. In this file all jump table calculations will be done. From parameters we know minimum and maximum height of jump (HEIGHT_MIN and HEIGHT_MAX). But in game, when player jumps, we have to give him some one-time velocity impulse. So, we have to calculate how big this impulse is to reach minimum and maximum allowed height.

Into newly created file, add this code:

```
namespace Generator {

    export class JumpTables {

        private static _instance = null;

        // velocities
        private _jumpVelocities: number[] = [];

        // ------------------------------------------------------------------------
        public static get instance(): JumpTables {
            if (JumpTables._instance === null) {
                JumpTables._instance = new JumpTables();
            }

            return JumpTables._instance;
        }
```

JumpTables class is singleton. Unfortunately, private constructor is currently not allowed in TypeScript, so user can still create another instance of JumpTables accidentally (good news is that private constructors will be available since Typescript 2.0). But we will access it only through public static property instance.

```
        public constructor() {
            this.calculateJumpVelocities();
        }

        // ------------------------------------------------------------------------
        private calculateJumpVelocities(): void {
            // all height samples
            for (let i = 0; i <= Parameters.HEIGHT_STEPS; i++) {
                // maximum height of jump for this step
                let height = Parameters.HEIGHT_MIN + (Parameters.HEIGHT_MAX - Parameters.HEIGHT_MIN)
/ Parameters.HEIGHT_STEPS * i;
                // v = sqrt(-(2 * s * g))
                this._jumpVelocities[i] = -Math.sqrt(2 * height * Parameters.GRAVITY);
            }
        }
```

First thing we do in constructor is we are calling calculateJumpVelocities() method. This method will turn jump heights into velocity impulses for physics engine. Calculation is done in several steps between minimum and maximum height. If we have height and we know gravity, we can calculate velocity impulse player's body must get in direction opposite to gravity to reach this height. Distance travelled in physics is:

$$s = v * t + \frac{1}{2} * g * t^2$$

where s is our jump height, v is initial velocity we are looking for, g is gravity and t is time. We are not interested in time. Maximum height will be reached when velocity equals to g * t. After this moment gravity gets stronger than initial velocity and player starts falling down:

$$v = -g * t$$

$$t = -\frac{v}{g}$$

then:

$$s = v * \left(-\frac{v}{g}\right) + \frac{1}{2} * g * \left(-\frac{v}{g}\right)^2$$

$$s = -\frac{v^2}{g} + \frac{1}{2} * \frac{v^2}{g}$$

$$s = -\frac{1}{2}\frac{v^2}{g}$$

$$v^2 = -2 * s * g$$

$$v = \sqrt{-2 * s * g}$$

As Phaser has y axis pointing down we are changing signs in code.

```
        public get minJumpVelocity(): number {
            return this._jumpVelocities[0];
        }

        // ------------------------------------------------------------------------
        public get maxJumpVelocity(): number {
            return this._jumpVelocities[this._jumpVelocities.length - 1];
        }
    }
}
```

Last piece of code are two properties for accessing minimum and maximum velocity – velocity for smallest jump and for highest one. We are not interested in all samples between them during game, we are only interested in limit values to limit player's jump. Samples between are important for further calculation of jump tables.

4.4 Visual debug output

Next, we will calculate jump table itself. But as code for it is long and it may be hard to imagine how it works, it will be helpful to have some temporary visual output. For it, we will use Phaser.BitmapData. BitmapData is Phaser object that contains canvas to which we can draw. And then we can use it as texture for sprite or image.

Open JumpTables.ts and add all this code to the end:

```
        // ------------------------------------------------------------------------
        // ------------------------------ DEBUG -----------------------------------
        // ------------------------------------------------------------------------
        private static _DEBUG = false;
        private static _globals: any;
        private static _debugBmd: Phaser.BitmapData;

        // ------------------------------------------------------------------------
        public static setDebug(debug: boolean, gameGlobals?: any): void {
            JumpTables._DEBUG = debug;
            JumpTables._globals = gameGlobals;

            if (debug) {
                if (typeof gameGlobals === "undefined" || gameGlobals === null) {
                    console.warn("No game globals provided - switching debug off");
                    JumpTables._DEBUG = false;
                } else {
                    JumpTables.createDebugBitmap();
                }
            }
        }
```

```
        // ------------------------------------------------------------------------
        public static get debugBitmapData(): Phaser.BitmapData {
            return JumpTables._debugBmd;
        }

        // ------------------------------------------------------------------------
        private static createDebugBitmap(): void {
            let global = JumpTables._globals;

            let bmd = new Phaser.BitmapData(global.game, "Grid", global.GAME_WIDTH,
global.GAME_HEIGHT);
            bmd.fill(192, 192, 192);

            // horizontal lines
            for (let i = 0; i < global.GAME_HEIGHT; i += Parameters.CELL_SIZE) {
                bmd.line(0, i + 0.5, global.GAME_WIDTH - 1, i + 0.5);
            }

            // vertical lines
            for (let i = 0; i < global.GAME_WIDTH; i += Parameters.CELL_SIZE) {
                bmd.line(i + 0.5, 0, i + 0.5, global.GAME_HEIGHT - 1);
                // add columns header numbers
                bmd.text("" + (i / Parameters.CELL_SIZE), i + 20, 20, "24px Courier", "#FFFF00");
            }

            JumpTables._debugBmd = bmd;
        }
```

Here we create few internal variables to track whether debug is on or off. We keep reference to global game variables – we will need game width and height, and also reference to BitmapData with debug output.

setDebug() is static method that switches debug on or off. If we set it on, we check if reference to Globals is passed. Without it we print warning and turn debug off. We will call this method before class instance is created.

debugBitmapData() is static get method that returns Phaser.BitmapData object.

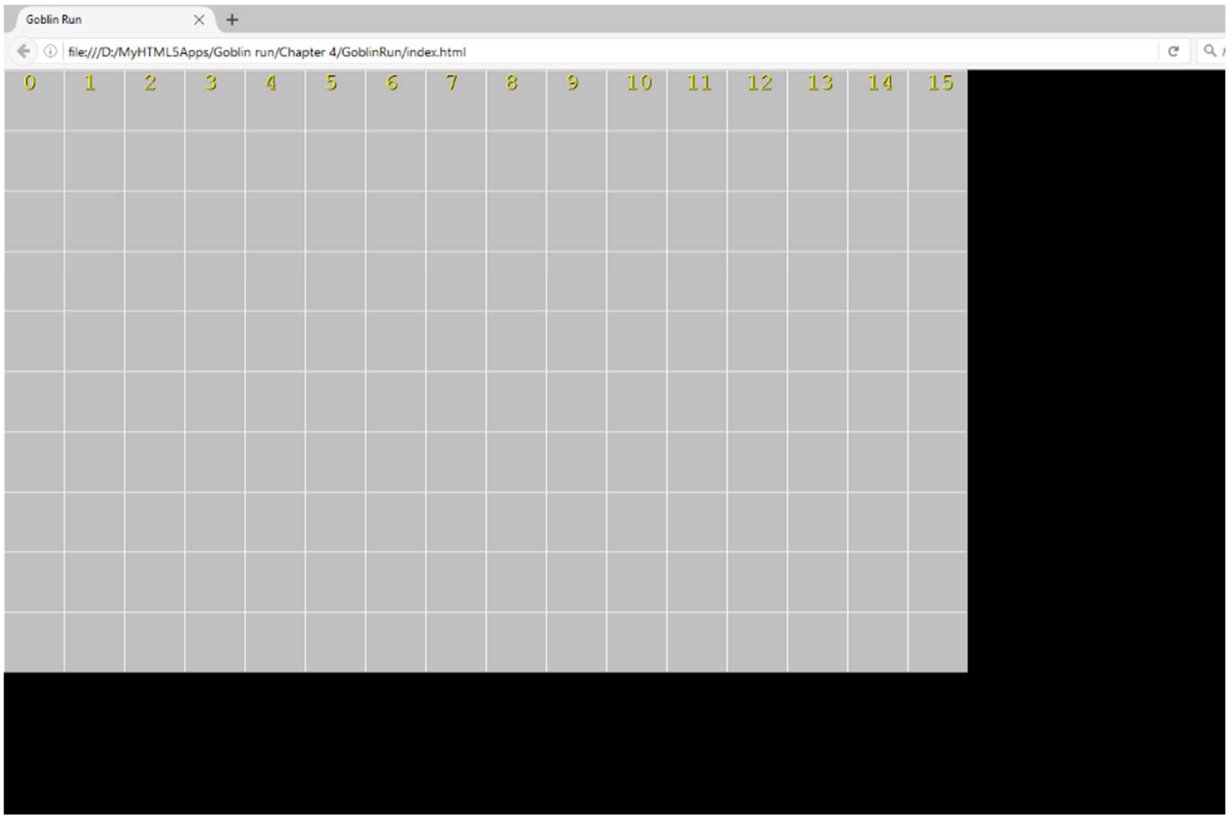
Finally, in createDebugBitmap() method we create Phaser.BitmapData object itself and then draw bunch of vertical and horizontal lines into it to form grid. When drawing vertical lines, we put number of column between them.

As last thing, we need to adjust create() method of Play state in Play.ts file (add changes in bold):

```
        create() {
            this.stage.backgroundColor = 0x80FF80;

            Generator.JumpTables.setDebug(true, GoblinRun.Global);

            Generator.JumpTables.instance;

            this.game.add.sprite(0, 0, Generator.JumpTables.debugBitmapData);
        }
```

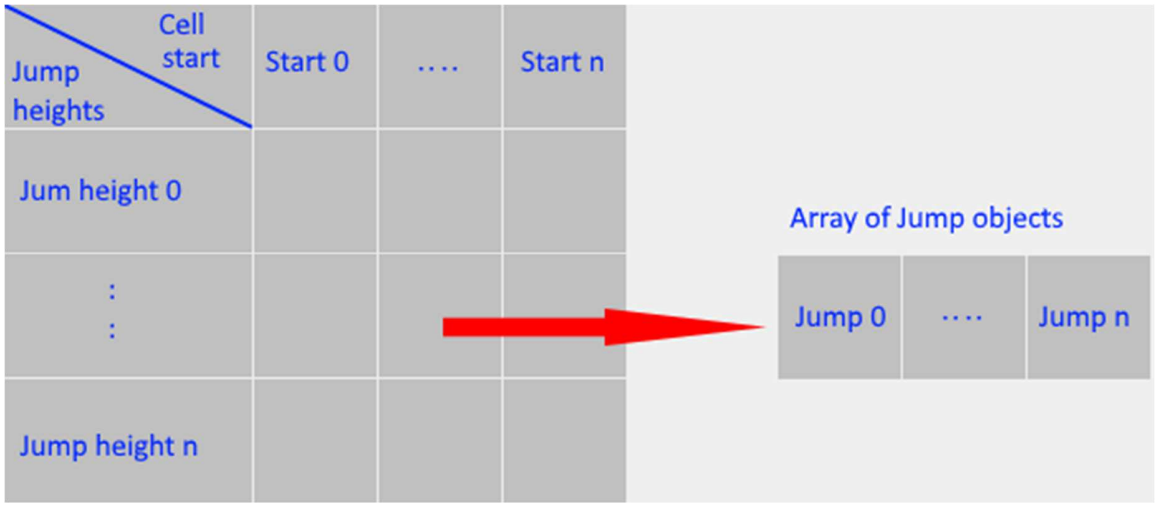
We set debug output on, which inside creates bitmap data with debug output. Call to instance() forces singleton creation. On last line we create sprite and as its texture we use bitmap with debug.

After these changes you can build game and run it in browser. You should see something like this on screen:



4.5 Jump table calculation

Now we can finally calculate jump table. It will be two-dimensional array where rows are jump heights and columns are jump starts within cell (where on cell jump starts). Each cell in this array will be one-dimensional array of all possible jump destinations for given height and start. In fact, it is three-dimensional array in the end, but looking at it as on two-dimensional array containing one-dimensional array as cell value is easier for explanation.



To store jumps in convenient way let's create another object – Jump. Create file Jump.ts in Generator folder and put this code into it:

```
namespace Generator {

    export class Jump {

        public offsetY: number = 0;
        public offsetX: number = 0;

        // ------------------------------------------------------------------------
        public toString(): string {
            return "offsetX: " + this.offsetX + ", offsetY: " + this.offsetY;
        }
    }
}
```

There is nothing special in this class. It just groups Jump properties and adds one method for debug output. offsetY and offsetX are offsets of destination cell from jump origin. Offsets are measured in cells, not pixels.

Now we have everything ready to write method that will create our table. Open JumpTables.ts file and add new property (in bold) on top:

```
namespace Generator {

    export class JumpTables {

        private static _instance = null;

        // velocities
        private _jumpVelocities: number[] = [];

        // list of possible jumps for each jump velocity and position within cell
        private _jumpDefs: Jump[][][] = [];
```

Update constructor:

```
        public constructor() {
            this.calculateJumpVelocities();
            this.calculateJumpTables();
        }
```

and create calculateJumpTables() method:

```
        private calculateJumpTables(): void {
            // all jump velocities
            for (let height = 0; height <= Parameters.HEIGHT_STEPS; height++) {

                this._jumpDefs[height] = [];

                // step from left to right on cell
                for (let step = 0; step < Parameters.CELL_STEPS; step ++) {
                    this.calculateJumpCurve(step, height);
                }
            }
        }
```

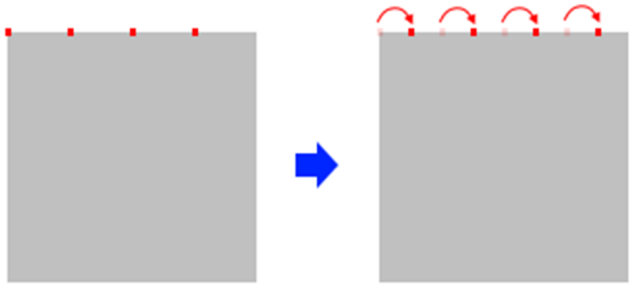
Here we create all cells of our jump table. We iterate through jump heights and for each height we iterate through all jump start offsets. This is our 2d array. In calculateJumpCurve() we will fill cell of this array with 1D array of Jump objects. Following method is rather long, so it will be split into parts.

```
    private calculateJumpCurve(step: number, jumpIndex: number): void {
        // simulation timestep
        let timeStep = 1 / 60;
```

To calculate jump curve, we will have to simulate it. Here is time step we will use. More precise the time step is, more accurate curve calculation. I chose 1/60 of seconds as this is also time step used by Phaser.

```
        // take jump velocity we calculated previously
        let velocity = this._jumpVelocities[jumpIndex];

        // start at middle of first step to spread samples better over cell
        // x and y positions are in pixels
        let x = step * Parameters.CELL_SIZE / Parameters.CELL_STEPS
            + Parameters.CELL_SIZE / Parameters.CELL_STEPS / 2;
        let y = 0;
        // y position in cells coordinates (row within grid)
        let cellY = 0;
```

x and y are start position in pixels. We have to calculate x position as offset from cell beginning based on step along cell top. To distribute starting points better across cell we shift it by half step as this image shows:

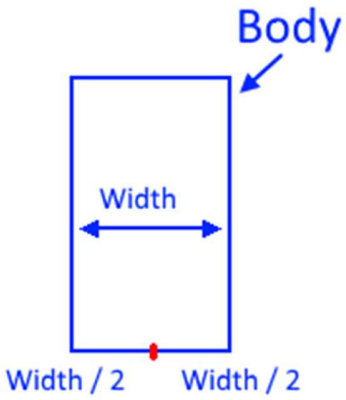


```
        // help variables to track previous position
        let prevX, prevY;

        // array of jumps from starting position to possible destinations
        let jumpDefs: Jump[] = [];
        // helper object that will help us keep track of visited cells
        let visitedList = {};
```

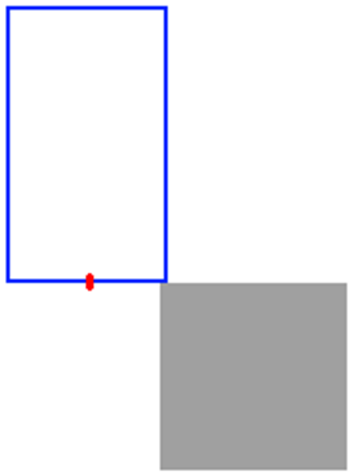
jumpDefs is our 1D array of jumps we will put into jump table as call value. visitedList is temporary helper object that will help us to check if cell was already visited.

```
        // half of player body width
        let playerWidthHalf = Parameters.PLAYER_BODY_WIDTH / 2 * 0.5;
```

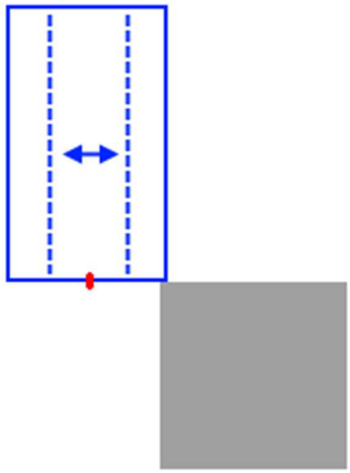
playerWidthHalf is just pre calculation of half of body width. Point we are simulating in jump is bottom center of body:



For calculations we have to check all cells from bottom left corner to bottom right corner. Notice, that we are multiplying it by 0.5, so in fact it is not half, but only quarter of width. It is to make the body narrower. Imagine situation during simulation like this:



Gray square is cell and we just found player can reach it with jump. But it can be extremely tricky and unfair to player to generate platforms that can be reached with only single pixel or two. So, we narrow body for calculations by one half. Now, body is represented with dashed lines and grey square is not reachable:



```
        // debug
        let debugBitmap = (JumpTables._DEBUG) ? JumpTables.debugBitmapData : null;
        // offset drawing of curve little bit down (by 4 cells),
        // otherwise it will be cut at top as we start jump at point [x, 0]
        let yOffset = Parameters.CELL_SIZE * 4;
```

Initial point for our simulation is on y = 0. To see something in debug drawing we offset it by 4 cells down, otherwise most important part of it would be out of the screen.

Now, we have all variables ready and we can start simulation loop. It will be terminated when our jump exceeds GRID_HEIGHT from Parameters.

```
        // simulate physics
        while (cellY < Parameters.GRID_HEIGHT) {
            // save previous position
            prevX = x;
            prevY = y;

            // adjust velocity
            velocity += Parameters.GRAVITY * timeStep;

            // new posiiton
            y += velocity * timeStep;
            x += Parameters.VELOCITY_X * timeStep;

            // draw path - small white dot
            if (JumpTables._DEBUG) {
                debugBitmap.rect(x, y+ yOffset, 2, 2, "#FFFFFF");
            }

            // left and right bottom point based on body width.
            let leftCell, rightCell;
            cellY = Math.floor(y / Parameters.CELL_SIZE);
```

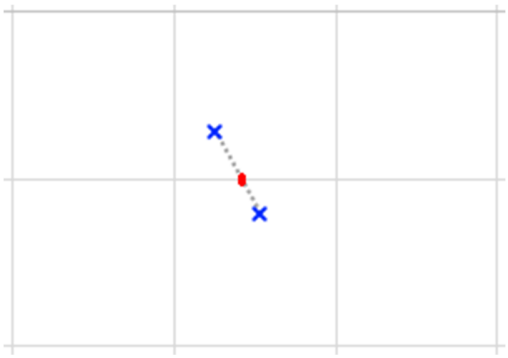
So far everything should be clear. We saved current position into prevX and prevY and calculated new one. Then we calculated y position in whole cells.

```
                // falling down
                if (velocity > 0) {
                    // crossed cell border to next vertical cell?
                    if (cellY > Math.floor(prevY / Parameters.CELL_SIZE)) {
```

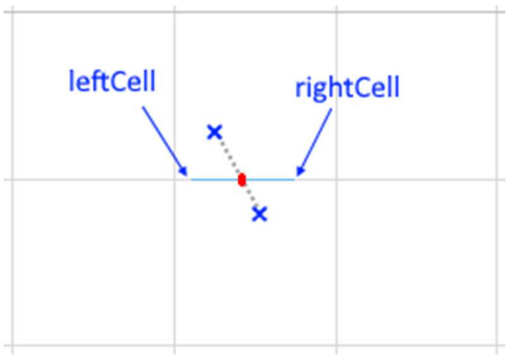
On our list of cells, we can reach with jump, will be only cells we can land on from top. Here we check if there is new cell y to examine.

```
                        // calc as intersection of line from prev. position and current position
with grid horizontal line
                        let pixelBorderY = Math.floor(y / Parameters.CELL_SIZE) *
Parameters.CELL_SIZE;
                        let pixelBorderX = prevX + (x - prevX) * (pixelBorderY - prevY) / (y -
prevY);

                        leftCell = Math.floor((pixelBorderX - playerWidthHalf) /
Parameters.CELL_SIZE);
                        rightCell = Math.floor((pixelBorderX + playerWidthHalf) /
Parameters.CELL_SIZE);
```

As our simulation is in discrete time steps, it may happen that previous position is inside one cell and second is inside another one:



For this, we have to calculate point of intersection and put it into pixelBorderY and pixelBorderX variables. For found point of intersection we calculate left and right corner of body in whole cells.




```
                        // all cells in x direction occupied with body
                        for (let i = leftCell; i <= rightCell; i++) {
                            let visitedId = i + (cellY << 8);

                            // if not already in list, then add new jump to reach this cell
```

```
                        if (typeof visitedList[visitedId] === "undefined") {
                            let jump = new Jump();

                            jump.offsetX = i;
                            jump.offsetY = cellY;

                            jumpDefs.push(jump);

                            console.log(jump.toString());
                        }
                    }
```

In next step we go from left bottom corner cell to right bottom corner cell and check if these cells were already examined previously. To make checking easy, we create unique id of cell based on its coordinates and look into visited object properties. If object does not contain this id, then this cell was not visited before and it is new cell we can jump on. In that case we create new Jump object and fill its properties.

```
                    // debug
                    if (JumpTables._DEBUG) {
                        // debug draw
                        let py = pixelBorderY + yOffset;

                        // line with original body width
                        let color = "#4040FF";
                        let pxLeft = pixelBorderX - Parameters.PLAYER_BODY_WIDTH / 2;
                        let pxRight = pixelBorderX + Parameters.PLAYER_BODY_WIDTH / 2;

                        debugBitmap.line(pxLeft, py, pxRight, py, color);

                        color = "#0000FF";
                        pxLeft = pixelBorderX - playerWidthHalf;
                        pxRight = pixelBorderX + playerWidthHalf;

                        // line with shortened body width
                        debugBitmap.line(pxLeft, py, pxRight, py, color);
                        debugBitmap.line(pxLeft, py - 3, pxLeft, py + 3, color);
                        debugBitmap.line(pxRight, py - 3, pxRight, py + 3, color);
                    }
                }
            }
```

In debug section we draw original width of body on top of the cell as well as reduced width. We can clearly see cells player can jump on.

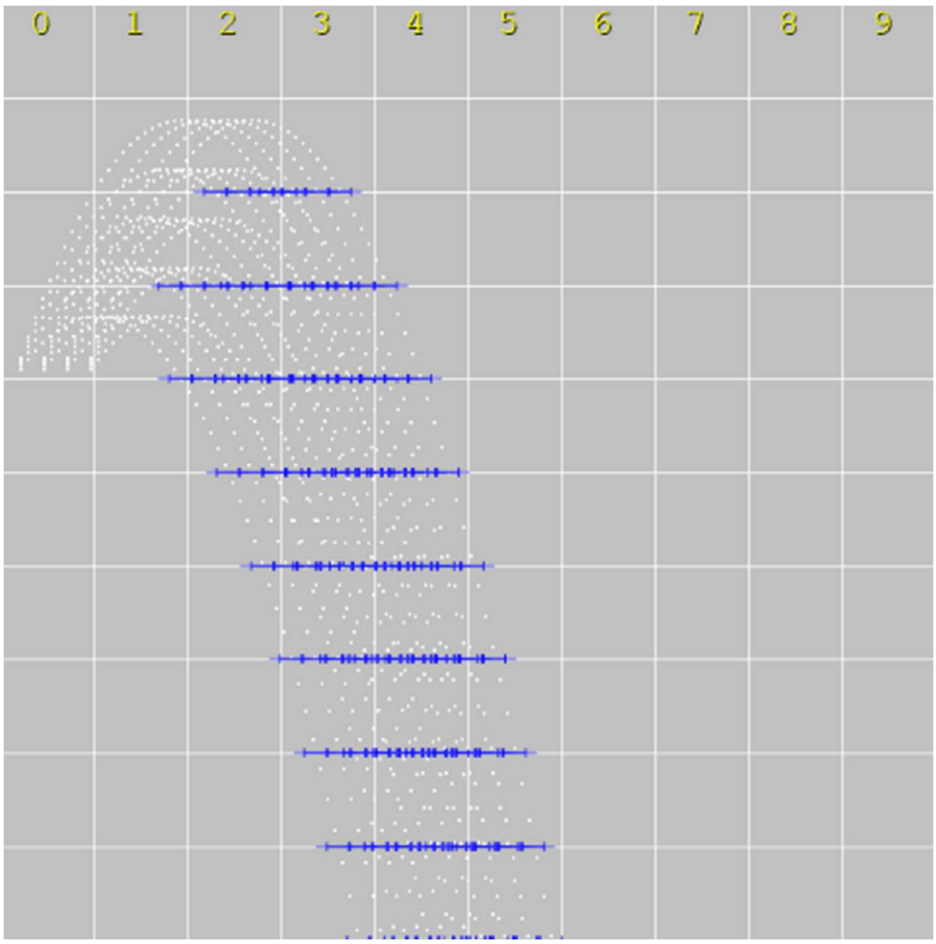
```
            leftCell = Math.floor((x - playerWidthHalf) / Parameters.CELL_SIZE);
            rightCell = Math.floor((x + playerWidthHalf) / Parameters.CELL_SIZE);

            // add grid cells to visited
            for (let i = leftCell; i <= rightCell; i++) {
                // make "id"
                let visitedId = i + (cellY << 8);
                if (typeof visitedList[visitedId] === "undefined") {
                    visitedList[visitedId] = visitedId;
                }
            }
        }
```

In this part of code, we first calculate left and right cell again, but this time for current x position, not point of intersection with grid. We add all cells, that bottom of body travelled through into list of visited cells.

```
            this._jumpDefs[jumpIndex][step] = jumpDefs;
        }
```

In the end of method, we store jump array into jump table.

Build and run the game. You should see terrible mess:



It is because on single debug image we have all jumps for all heights and all start offsets.

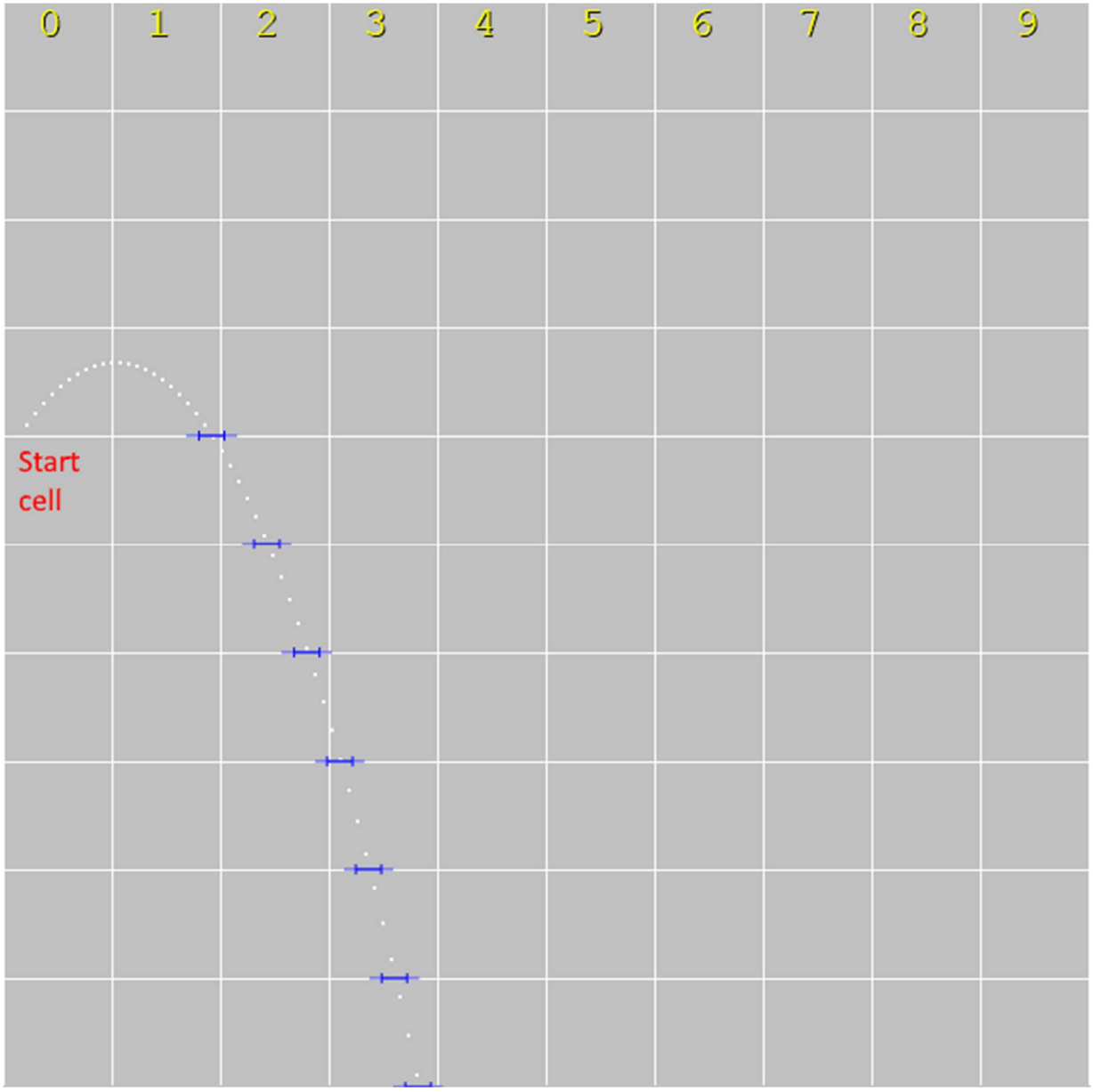
4.6 Calculation results

To understand what we calculated in previous part, let's limit calculation to only one jump height and one start offset. Adjust calculateJumpTables like this (changes are in bold) and run game again:

```
        private calculateJumpTables(): void {
            // all jump velocities
            for (let height = 0; height <= 0 /*Parameters.HEIGHT_STEPS*/; height++) {

                this._jumpDefs[height] = [];
```

```
                // step from left to right on cell
                for (let step = 0; step < 1 /*Parameters.CELL_STEPS*/; step++) {
                    this.calculateJumpCurve(step, height);
                }
            }
        }
```

Now, the result is more transparent:



Along with it, you have this output in console (you can comment it out later):

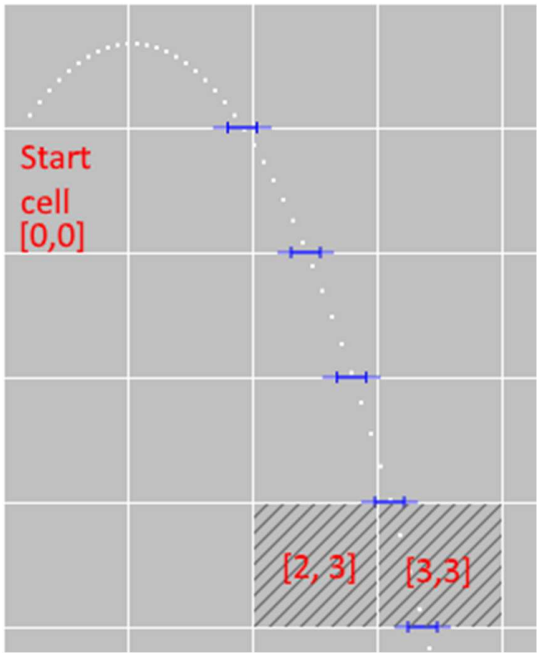
```
offsetX: 1, offsetY: 0                    goblinrun…:192:33
offsetX: 2, offsetY: 0                    goblinrun…:192:33
offsetX: 2, offsetY: 1                    goblinrun…:192:33
offsetX: 2, offsetY: 2                    goblinrun…:192:33
offsetX: 2, offsetY: 3                    goblinrun…:192:33
offsetX: 3, offsetY: 3                    goblinrun…:192:33
offsetX: 3, offsetY: 4                    goblinrun…:192:33
offsetX: 3, offsetY: 5                    goblinrun…:192:33
offsetX: 3, offsetY: 6                    goblinrun…:192:33
offsetX: 3, offsetY: 7                    goblinrun…:192:33
          :
          :
          :
```

What it all does say? White dots on image are jump trajectory. For debug drawing we shifted it 4 cells down. Cell, from top of which we start, I labeled "Start cell" on the picture and its coordinate is 0, 0. Jump goes up and then down. At first intersection with cell border, when going down, is first blue line. Longer line is original width of body. Shorter line with small borders on left and right is narrowed body. You can see it starts at cell in column 1 and ends in cell in column 2. It means we can reach two cells with this jump.

Now, look into console output. offsetX and offsetY are destination cell coordinates relative to start cell. It says, that if we make smallest jump, we can land on two cells – on one cell ahead (offsetX = 1) and on the same y level (offsetY = 0) or on second cell with offsetX = 2 and offsetY = 0;

If we look at next blue line, we see, its offset is x = 2, y = 1 and in console output there is indeed this information.

For our future platform generator, we now see, that if we want player to make small jump and next platform will be 3 cells under current, then x distance of next platform must be 2 or 3 cells:



Return back bounds in calculateJumpTables() to calculate all jumps for all heights and starts. In next part we will analyze results in jump table to prepare it for easy querying.

4.7 Jump table analysis

Now, when our jump table is ready, we will go through it and prepare it for easy querying from game. For example, game may ask what is minimal x offset of next platform if we want to generate it one cell above current platform.

On top of JumpTalbe.ts add new properties:

```
        // results of jump table analysis
        private _jumpOffsetsY: number[] = [];
        private _jumpOffsetYMax: number = 0;
        private _jumpOffsetXMins: any = {};
        private _jumpOffsetXMaxs: any = {};
```

Add call to analyzeJumpTables() in the end of calculateJumpTable():

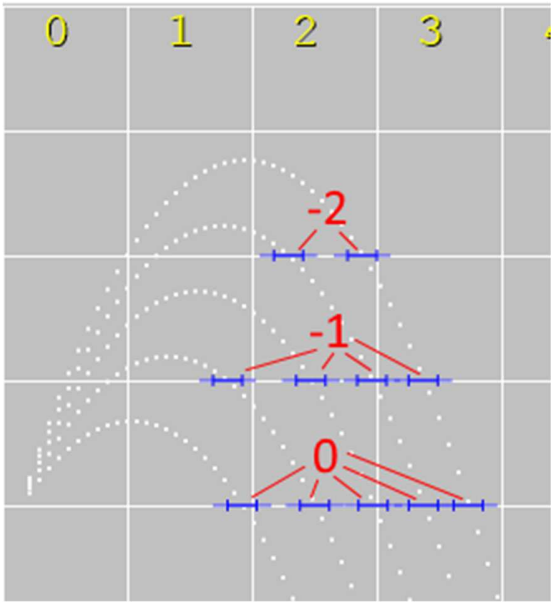
```
        private calculateJumpTables(): void {

                        :
                        :

            // analyze created jump tables
            this.analyzeJumpTables();
        }
```

Add analyzeJumpTable method():

```
        private analyzeJumpTables(): void {
            // min y
            this._jumpOffsetYMax = 0;

            // through all jump velocities
```

```
            for (let velocity = 0; velocity < this._jumpDefs.length; velocity++) {
                // get only first x position within cell and first jump for given velocity,
                // because all have the same height
                this._jumpOffsetsY[velocity] = this._jumpDefs[velocity][0][0].offsetY;
                // check for maximum offset in y direction.
                // As it is negative number, we are looking for min in fact
                this._jumpOffsetYMax = Math.min(this._jumpOffsetYMax, this._jumpOffsetsY[velocity]);
            }
```

Here we are searching for highest jump offset – how much cells up we can jump for each jump height and also what is highest possible jump regardless of jump height (= results in offset for highest jump). As offset from current position up is negative, we are looking for minimum. With our current settings in Parameters we can jump 2 cells up maximally.



Now we will calculate minimum and maximum x offset for all height levels, where we can jump regardless of button press duration. We are skipping smallest height jump as it requires shortest button press. So short, it is almost impossible to make.

```
            // find minimum and maximum offset in cells to jump to at given height level
            for (let velocity = 1; velocity < this._jumpDefs.length; velocity++) {

                // get only first startX, because it has smallest x offset
                let jumps = this._jumpDefs[velocity][0];

                for (let j = 0; j < jumps.length; j++) {
                    let jump = jumps[j];
                    let currentMin = this._jumpOffsetXMins[jump.offsetY];

                    this._jumpOffsetXMins[jump.offsetY] = (typeof currentMin !== "undefined") ?
                        Math.min(currentMin, jump.offsetX) : jump.offsetX;

                    // console.log("LEVEL: " + jump.offsetY + " - jump from " +
this.minOffsetX(jump.offsetY));
                }

                // get only last startX, because it has biggest x offset
                jumps = this._jumpDefs[velocity][this._jumpDefs[velocity].length - 1];
```

```
                for (let j = 0; j < jumps.length; j++) {
                    let jump = jumps[j];
                    let currentMax = this._jumpOffsetXMaxs[jump.offsetY];

                    this._jumpOffsetXMaxs[jump.offsetY] = (typeof currentMax !== "undefined") ?
                        Math.max(currentMax, jump.offsetX) : jump.offsetX;

                    // console.log("LEVEL: " + jump.offsetY + " - jump to " +
this.maxOffsetX(jump.offsetY));
                }
            }
        }
```

Finally, we add some query methods:

```
        public maxOffsetY(jumpIndex: number = -1): number {
            if (jumpIndex === -1) {
                return this._jumpOffsetYMax;
            } else {
                return this._jumpOffsetsY[jumpIndex];
            }
        }

        // ------------------------------------------------------------------------
        public maxOffsetX(offsetY: number): number {
            let maxX = this._jumpOffsetXMaxs[offsetY];

            if (typeof maxX === "undefined") {
                console.error("max X for offset y = " + offsetY + " does not exist");
                maxX = 0;
            }

            return maxX;
        }

        // ------------------------------------------------------------------------
        public minOffsetX(offsetY: number): number {
            let minX = this._jumpOffsetXMins[offsetY];

            if (typeof minX === "undefined") {
                console.error("min X for offset y = " + offsetY + " does not exist");
                minX = 0;
            }

            return minX;
        }
```

For example, we want to generate platform two tiles (y offset = -2) above current one. But what is x offset we can use so player can safely reach it? Call minOffsetX(-2) to get minimum x offset for given y offset and maxOffsetX(- 2) to get maximum x offset. Then we can generate new platform with x offset somewhere between these two values and y offset equal -2.

Summary

In this chapter we wrote big part of generator. We created jump tables based on game parameters. This means, what we did is very flexible. You can change minimum and maximum height of jump as well as number of height samples or other parameters and everything gets recalculated when game is run for first time.

5. Generating platforms

With jump table ready we can move to generating platforms. In this chapter we will create first version of level. Remember two basic criteria for procedurally generated content: it has to be playable, it should be nice. While our level will be playable, it will not be nice yet. We will beautify it in next chapters. Just one remark, nice does not mean nice graphics or cool effects in this context. It means that generated content is pleasant to human eye. Too random platforms look distracting.

5.1 Adding test block

First we will make some preparation work. As we are moving from our debug grid to in-game graphics, we will need some sprites. In current state we do not need final graphics, we just need something, that will be displayed to give us visual feedback. For this we will create this ugly 64x64 block with name "Block.png", which we will put into assets folder:

If you do not want to draw it by yourself, you can copy one already prepared from Resources/_RES/Sprites/.

To use it in game, we have to load it. Open Preload.ts and put new line into preload() method:

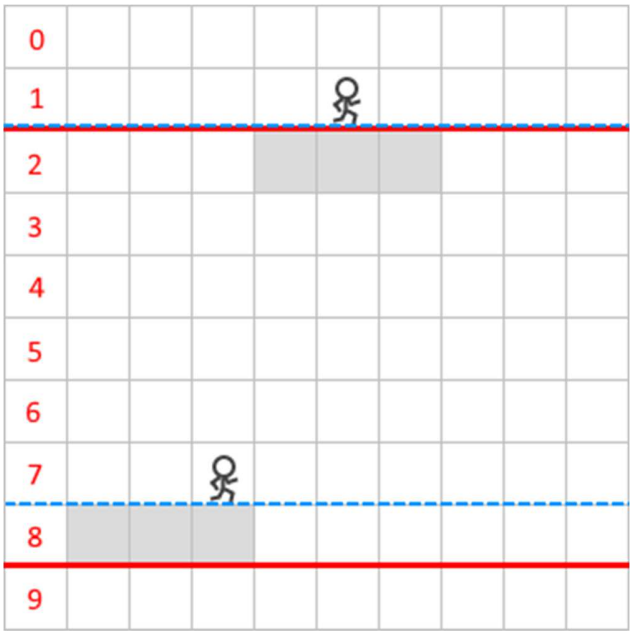
```
        public preload() {
            this.load.image("Block", "assets/Block.png");
        }
```

5.2 Adding platform bounds

Open Parameters.ts and add two new lines to our generator parameters:

```
        // bounds for generating platforms
        public static UBOUND = 2;
        public static LBOUND = 8;
```

Currently, our game is 640 pixels high. It is 10 cells. What we want is to keep generated platforms on screen as well as we want to see player all the time. As our player is always running on top of the platform, then generating it at row 0 would mean, that player is off screen. To avoid this, we will set upper bound to 2 and lower bound to 8. Red lines on following image show bounds where graphic tiles will be displayed – lower red line is on bottom edge of cell as tile graphic is 64 pixels high. Blue dashed lines show bounds our player will move on. On top there will be always at least two free rows above upmost platform, so we can see our player running. On bottom there will be always at least one empty row under bottom most platform.



5.3 Level piece class

Now, let's organize somehow pieces our level is made from. For long time from now on, when we speak about piece, it will be equal to platform. But in later stages of development, we will add bonus jump item that allows player to make another jump while still in the air – it will be another kind of piece.

Add new file into Generator folder and name it Piece.ts:

```
namespace Generator {

    export class Piece {

        // absolute position of left cell / tile
        public position = new Phaser.Point(0, 0);
        // offset from end of previous piece
        public offset = new Phaser.Point(0, 0);
        // length in cells / tiles
        public length: number;
    }
}
```

It just groups properties for piece:

- position is absolute position in world (in cells),
- offset is offset from last cell of previous piece,
- length is length of piece in cells.

In next step we will create Generator class, that will initialize and manage pieces – position them right with respect to our jump table, limit it to bounds, etc. But before that, we have to create one helper class that will help us to maintain pool of pieces.

5.4 Generic Pool class

Add new folder into src with name Helper and create file Pool.ts with following code in it:

```
namespace Helper {

    export class Pool<T> {

        private _classType: any;
        private _newFunction: Function = null;

        private _count: number = 0;
        private _pool: T[] = [];

        private _canGrow: boolean = true;

        private _poolSize: number = 0;

        // ------------------------------------------------------------------------
        constructor(classType: any, count: number, newFunction = null) {
            this._classType = classType;
            this._newFunction = newFunction;

            for (var i = 0; i < count; i++) {
                // create new item
                var item = this.newItem();
                // store into stack of free items
                this._pool[this._count++] = item;
            }
        }

        // ------------------------------------------------------------------------
        public createItem(): T {
            if (this._count === 0) {
                return this._canGrow ? this.newItem() : null;
            } else {
                return this._pool[--this._count];
            }
        }

        // ------------------------------------------------------------------------
        public destroyItem(item: T): void {
            this._pool[this._count++] = item;
        }

        // ------------------------------------------------------------------------
        protected newItem(): T {
            ++this._poolSize;

            if (this._newFunction !== null) {
                return this._newFunction();
            } else {
                return new this._classType;
            }
        }

        // ------------------------------------------------------------------------
        public set newFunction(newFunction: Function) {
            this._newFunction = newFunction;
        }

        // ------------------------------------------------------------------------
        public set canGrow(canGrow: boolean) {
            this._canGrow = canGrow;
```

```
        }
    }
}
```

This class is pretty easy to understand. It is generic pool class, so you can pool any type of objects in it. Its purpose is to maintain array of "free" instances of object of given type. We can reuse these objects instead of creating and destroying them every time. It should have positive impact on garbage collector operations. Constructor has three parameters:

- classType – object type that will be stored in pool,
- count – initial number of objects in pool,
- newFunction – optional parameter. We can pass here function, that will be used when creating new instance of classType. In it we can do additional setup of newly created object.

createItem() and destroyItem() are key methods we will work with. With the first one we are requesting new object instead of call to new(). If pool has free object on stock, we are given it. If not, two situations may follow. Either we are returned null if canGrow property is set to the false or by default new object is created and pool size is thus increased by one. By setting canGrow you can specify whether pool will be limited or unlimited. In case of unlimited pool, if there are no free items available, new ones are created.

Second method destroyItem() takes item we want to return into pool as parameter. This item is put into array of unused objects. It can be reused when you call createItem() next time.

When you are creating instance with call to new() every time you need some object, you can work in "fire and forget" mode. If you do not need that instance anymore, you just leave it to garbage collector to clean it. When working with pool, you are responsible for returning objects back into pool.

5.5 Generator class

With Pool class ready we can return to generator. Create file Generator.ts in Generator folder. In the beginning of it, we will create some internal variables and constructor:

```
namespace Generator {

    export class Generator {

        private _rnd: Phaser.RandomDataGenerator;
        private _jumpTables: JumpTables;

        private _piecesPool: Helper.Pool<Piece>;

        private _lastGeneratedPiece: Piece = null;

        // ---------------------------------------------------------------------
        public constructor(rnd: Phaser.RandomDataGenerator) {
            // random numbers generator
            this._rnd = rnd;

            // reference to jump tables
            this._jumpTables = JumpTables.instance;

            // pool of pieces
            this._piecesPool = new Helper.Pool<Piece>(Piece, 16);
        }
```

_rnd is reference to generator of random numbers that is part of Phaser.Game object and we pass it to Generator's constructor. _jumpTables is referece to jump tables we calculated in previous chapter. Generator will use them to position platforms so it is reachable by player.

We create pool of pieces. Default size of pool is 16, which should be enough. If we run out of them, pool will automatically add new items as its canGrow property is set to the true by default.

In _lastGeneratedPiece we hold reference to last piece we created, which will come handy later.

```
        private createPiece(): Piece {
            let piece = this._piecesPool.createItem();

            if (piece === null) {
                console.error("No free pieces in pool");
            }

            return piece;
        }

        // ---------------------------------------------------------------------
        public destroyPiece(piece: Piece): void {
            this._piecesPool.destroyItem(piece);
        }
```

createPiece() method is private and will be used by generator when we either request it to generate new piece randomly or force it to create new piece with fixed parameters.

destroyPiece() is public and we will call it from game when we do not need piece anymore. It will just return this unused piece into internal pool of pieces.

Two methods how to get new piece from generator are:

```
        public setPiece(x: number, y: number, length: number, offsetX: number = 0, offsetY: number =
0): Piece {
            let piece = this.createPiece();

            piece.position.set(x, y);
            piece.offset.set(offsetX, offsetY);
            piece.length = length;

            return piece;
        }
```

This method just forces generator to create piece, fill it with requested values and return it. There is no randomness. We can use this method, when we need to place some piece of known length on some known position. We will use this for first platform, which we can create little longer than other platforms and in the middle of the screen, so player has time to notice he is already in game and where on screen he is.

Key method of class is generate() method:

```
        public generate(lastPosition: Phaser.Point): Piece {

            let piece = this.createPiece();

            let ubound = Parameters.UBOUND;
            let lbound = Parameters.LBOUND;
```

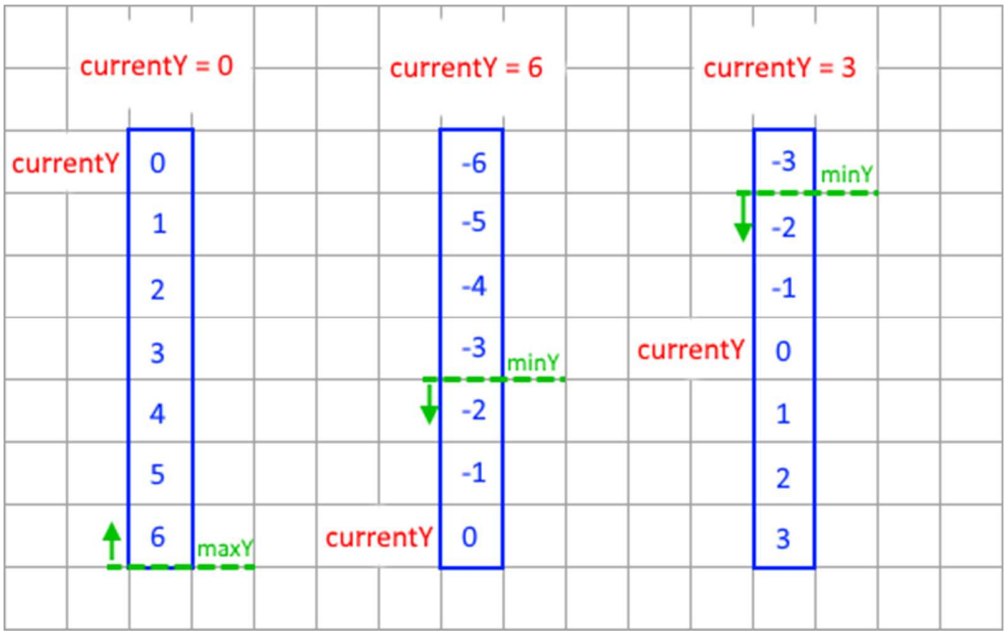
Method is passed lastPosition from game, which is point with end of previous platform or it can be any point we want it generate new piece from. Following calculation is split into finding y, x and length for new piece.

```
            // Y POSITION
            // how high can jump max
            let minY = this._jumpTables.maxOffsetY();
            // how deep can fall max
            let maxY = lbound - ubound;

            // clear last y from upper bound, so it starts from 0
            let currentY = lastPosition.y - ubound;

            // new random y position - each y level on screen has the same probability
            let shiftY = this._rnd.integerInRange(0, lbound - ubound);
            // substract currentY from shiftY - it will split possible y levels to negative
            // (how much step up (-)) and positive (how much to step down (+))
            shiftY -= currentY;
            // clamp step to keep it inside interval given with maximum
            // jump offset up (minY) and maximum fall down (maxX)
            shiftY = Phaser.Math.clamp(shiftY, minY, maxY);

            // new level for platform
            // limit once more against game limits (2 free tiles on top, 1 water tile at bottom)
            let newY = Phaser.Math.clamp(currentY + shiftY, 0, lbound - ubound);

            // shift by upper bound to get right y level on screen
            piece.position.y = newY + ubound;
            // offset of new piece relative to last position (end position of last piece)
            piece.offset.y = piece.position.y - lastPosition.y;
```

First, into minY and maxY we load how high from current positon player can get – it is limited with value stored in jump table for highest jump – and how deep he can fall. Maximum height of fall is number of rows between bounds between which we want to generate platforms. These are limit values for y positon of new piece. In our case (regarding jump table and parameters in Parameters.ts), maximum jump is -2 and maximum fall is 6.

In next step we calculate currentY as shift of last y position up by upper bound. As our y positon was previously between ubound and lbound (from 2 to 8), after this it is in range 0 to 6.

Let's choose some random y position in whole range of bounds and put it into shiftY variable. All values in this range have the same probability to be chosen. Then we subtract currentY from it. What happens is, that it moves selected value either above currentY or below it (or it equals it). We then clamp it to range from minY to maxY. This image should explain it:



There are three examples:

1. currentY = 0. It means our last platform was as high as allowed with bounds (remember, we shifted last y position up by upper bound to make it start from zero). We chose some random number in range 0 to 6 and then subtracted currentY (=0) from it. So, our random number is still some number in range from 0 to 6 (first column of blue numbers). Finally, we clamped it into range -2 to 6 (green numbers and arrows), so we still remain in 0 to 6. It means, that whatever number was randomly chosen, it will be fall down and that number says how big the fall is. In other words, if our last platform was as high as possible in game, then this code ensures, that only fall will follow (or next platform will be on the same level if number 0 was chosen). But we can be sure, that no platform will be generated higher,
2. currentY = 6. Our last platform is as deep as possible in game. If we subtract this value from selected number, we get something in range from -6 to 0. So, next platform will be on the same level (if random number was zero) or it will be up to 6 levels higher. But, our maximum jump height is -2. This is where clamping saved us and our random value is either 0, -1 or -2. Important is, that our code ensured there will be no further fall,
3. currentY = 3. Last platform was somewhere in the middle. Range of numbers changed to -3 to 3. This means, there is equal probability, that our random number is above current platform or below it (and smaller one, it is on the same level). Clamping cut off impossible jumps (-3), but did not change probabilities (as number was chosen already before clamping).

In short: deeper we get with platforms, higher the probability that next platform will be above. Higher the current platform, higher the possibility, that next one will be below it.

After all this, we just store new absolute y position of new piece and new relative (offset.y) position of it.

```
            // X POSITION
            let minX = this._jumpTables.minOffsetX(newY - currentY);
            let maxX = this._jumpTables.maxOffsetX(newY - currentY);
```

```
            // position of next tile in x direction
            let shiftX = this._rnd.integerInRange(minX, maxX);

            // new absolute x position
            piece.position.x = lastPosition.x + shiftX;
            // offset of new piece relative to last position (end position of last piece)
            piece.offset.x = shiftX;
```

Calculating new x is not so difficult to imagine. Based on y offset we calculated before, we get minimum and maximum x offset for it from jump table. Then we choose randomly between these bounds.

```
            // LENGTH
            piece.length = this._rnd.integerInRange(3, 5);


            // RESULT
            this._lastGeneratedPiece = piece;
            return piece;
        }
    }
}
```

Length is randomly selected in interval from 3 to 5.

Call to generate() returns new piece, that is reachable from last cell of previous piece.

5.6 Main Layer

We are slowly moving to real game. Therefore, we need some game objects on screen that will present it. First we create main layer, which is derived from Phaser.Group. This object will group all platforms on screen – more precisely, all tiles platforms are made from. When we started coding generator, we used more general term cell for grid element to distinguish between it and tile which is visual object. As we are now in game code, we will speak about tiles.

Main idea for this layer is that camera focuses on player, who runs and jumps from platform to platform in this group. As we are moving camera to right, our platform tiles move in opposite direction on screen. Camera reveals new tiles on right and some tiles are moving completely out of the screen on left. Because our pool of pieces in generator is limited, we cannot have big portion of level generated to the right in advance. Also for tiles we will have some limited amount (again pooled). We have to generate platforms on the fly and turn them into tiles. On the other side we have to collect these tiles if they get completely out of the screen on left and recycle them.

Let's write some code. Crete new folder Game in src folder and create file MainLayer.ts in it. Start with following code:

```
namespace GoblinRun {

    const enum eGenerateState { PROCESS_PIECE, GENERATE_PIECE }
```

This enum is used internally to keep track of state of making new pieces. Const enums in TypeScript are completely removed during compilation into Javascript and replaced with number constants on places

where used. Non const enums store mapping from names to values and in reversed direction from values to names, so are less efficient.

```
export class MainLayer extends Phaser.Group {

    private _generator: Generator.Generator;

    private _wallsPool: Helper.Pool<Phaser.Sprite>;
    private _walls: Phaser.Group;

    private _lastTile: Phaser.Point = new Phaser.Point(0, 0);
    private _state: eGenerateState;

    // piece generated with generator
    private _piece: Generator.Piece = null;
```

First, few private variables are defined:

- _generator – generator we worked on while ago. We will ask it for new pices,
- _wallsPool – pool of wall tiles. Tiles are Phaser.Sprite objects,
- _walls – another Phaser.Group. This group is child of MainLayer group to group wall tiles. Despite fact Phaser groups allow pooling out of the box, we will use pool in _wallPool, because we want some extra setup for every newly created tile. If using our Pool class, we can do it without any additional work,
- _lastTile – is position of last tile in tile coordinates. We can pass this to generator's generate() method as lastPosition parameter,
- _state – internal state when preparing new pieces – either requesting them from generator or turning them into tiles,
- _piece – last piece returned from generator.

```
    public render(): void {
        this._walls.forEachExists(function (sprite: Phaser.Sprite) {
            this.game.debug.body(sprite);
        }, this);
    }
```

This is just small helper. render() method in Phaser.State object is called automatically. Here we are in Phaser.Group object, so this method does not get called by engine. We will have to call it manually from Play state later. Purpose of this method is to debug draw physics bodies of existing tiles. Phaser draws light green semitransparent rectangle for it.

```
    public constructor(game: Phaser.Game, parent: PIXI.DisplayObjectContainer) {
        super(game, parent);

        // platforms generator
        this._generator = new Generator.Generator(game.rnd);

        // pool of walls
        this._wallsPool = new Helper.Pool<Phaser.Sprite>(Phaser.Sprite, 32, function () {
            // add empty sprite with body
            let sprite = new Phaser.Sprite(game, 0, 0, "Block");
            game.physics.enable(sprite, Phaser.Physics.ARCADE);

            let body = <Phaser.Physics.Arcade.Body>sprite.body;
            body.allowGravity = false;
```

```
                body.immovable = true;
                body.moves = false;
                body.setSize(64, 64, 0, 0);

                return sprite;
            });

            // walls group
            this._walls = new Phaser.Group(game, this);

            // set initial tile for generating
            this._piece = this._generator.setPiece(0, 5, 10);
            this._state = eGenerateState.PROCESS_PIECE;
        }
```

As first thing in constructor, we call parent's constructor. We have parent parameter here as mandatory, so new group will be immediately placed under its parent in scene graph tree.

Then we create instance of generator and pass it instance of Phaser.RandomDataGenerator, that is part of Phaser.Game object (game.rnd).

After that we create pool of wall sprites. Each wall is Phaser.Sprite object. We create 32 of them and this time we need some additional setup, therefore we pass also function as newFunction parameter. In that function we first create new instance of Phaser.Sprite, assign it our test graphics ("Block") and then set its body.

In game we are using most simple Phaser physics system – Arcade physics. When you want to use any Phaser physics system, you should start it first with call to:

```
game.physics.startSystem(name_of_system);
```

but we do not need it, as Phaser.Physics.Arcade system is running by default.

Every sprite, that shall have physics body must first enable it – this is our call to game.physics.enable(). Once the body is enabled we set it:

- allowGravity = false – we do not want our tiles to fall down from screen,
- immovable = true – we do not want tiles to react when something hits them. It should stay firmly fixed to its position,
- moves = false – we do not want physics system to move this body. We will be placing it on screen and letting it disappear (when out of the screen) as we need,
- setSize(64, 64, 0, 0) – size is 64x64 pixels, and offset from sprite's top-left corner is 0,0.

As this is our newFunction function, we have to return instance of object in the end of it.

We continue with creating _walls group and making it child of our current object. Notice, that tile sprites we created in previous step have no parent so far. So, they are not anywhere in scene tree and thus are not visible yet. We will add them to _walls group when needed.

In the end of constructor, we force generator to return piece with length 10 cells, starting in left middle of screen. We store it into _piece variable and set _state to process this piece.

In next step we will write generate() method. This method is called on every update and is passed current world camera x position converted into tiles. In Phaser camera's position returns its top left corner, so its x coordinate is left border of the screen.

```
        public generate(leftTile: number): void {
            // remove tiles too far to left
            this.cleanTiles(leftTile);
```

As first step we check for all tiles that got out of the screen on left and clear them.

```
            // width of screen rounded to whole tiles up
            let width = Math.ceil(this.game.width / Generator.Parameters.CELL_SIZE);
```

Width of screen in tiles is calculated and rounded up.

Now, most important phase starts. In constructor, we set initial piece, which is now stored in _piece variable and set _state to PROCESS_PIECE. These are setting with which we will enter this method first time. In _lastTile variable we store position of last placed tile.

```
            // generate platforms until we generate platform that ends out of the screen on right
            while (this._lastTile.x < leftTile + width) {
```

On first run following piece of code will run until we make sure that whole game screen is covered with generated platforms from left to right. On other runs it will be executed only if condition in while statement is true – which will if we scrolled enough to left and we need to generate next piece of level.

```
                switch (this._state) {

                    case eGenerateState.PROCESS_PIECE:
                        {
                            this._lastTile.copyFrom(this._piece.position);
                            let length = this._piece.length;

                            // process piece
                            while (length > 0) {
                                this.addBlock(this._lastTile.x, this._lastTile.y);

                                if ((--length) > 0) {
                                    ++this._lastTile.x;
                                }
                            }

                            // return processed piece into pool
                            this._generator.destroyPiece(this._piece);

                            // generate next platform
                            this._state = eGenerateState.GENERATE_PIECE;

                            break;
                        }
```

In _piece variable, we have piece returned by generator (either with setPiece() or generate() methods). We go from left to right along this piece and add tiles with addBlock() method. During it we update _lastTile variable to later know where last tile was put.

After turning piece into tiles we can tell to generator, we are finished with piece and it can be returned to pool of pieces. As we processed generated piece, we switch state to GENERATE_PIECE. So next time the while loop is entered, it will skip PROCESS_PIECE case and go into GENERATE_PIECE branch.

```
                case eGenerateState.GENERATE_PIECE:
                    {
                        this._piece = this._generator.generate(this._lastTile);
                        this._state = eGenerateState.PROCESS_PIECE;
                        break;
                    }
            }
        }
    }
```

If previous piece was already processed, _state is set to GENERATE_PIECE and we fall into this case. We just ask generator for next piece and change _state to process it. As this part of code does not change _lastTile position, it never exits while loop. But as we changed _state, it will jump into PROCESS_PIECE case again in next iteration. Switching between processing and generating will go as long as we need new pieces.

In this moment you may ask, why we created pool of pieces in generator, when we actually use only one piece at time – we alternately generate – process, generate – process, ... It will become important when we start to make our generation more beautiful. Then, in some situations, we will generate more than one piece at time.

Following is definition of methods cleanTiles() and addBlock():

```
    private cleanTiles(leftTile: number) : void {
        leftTile *= Generator.Parameters.CELL_SIZE;

        for (let i = this._walls.length - 1; i >= 0; i--) {
            let wall = <Phaser.Sprite>this._walls.getChildAt(i);

            if (wall.x - leftTile <= -64) {
                this._walls.remove(wall);
                wall.parent = null;
                this._wallsPool.destroyItem(wall);
            }
        }
    }
```

In cleanTiles() method we just check all wall tiles sprites for those, that are too far on left. If it is more than 64 pixels (width of tile) off screen, then we remove it from parent group and return into pool.

```
    private addBlock(x: number, y: number): void {
        // sprite  get from pool
        let sprite = this._wallsPool.createItem();
        sprite.position.set(x * 64, y * 64);

        sprite.exists = true;
        sprite.visible = true;

        // add into walls group
        if (sprite.parent === null) {
            this._walls.add(sprite);
        }
    }
```

```
    }
}
```

In addBlock() method we get x and y coordinate of tile. We pick free wall sprite from pool and place it on that position converted into pixels. We make sure sprite exists and is visible and if its parent is null, we make it child of walls group.

5.7 Observing results

We are almost ready to see something on screen. To test all that work we did, let's make some small adjustments to Play state in Play.ts file. Changes are in bold:

```
namespace GoblinRun {

    export class Play extends Phaser.State {

        private _mainLayer: MainLayer;

        // --------------------------------------------------------------------------
        public render() {
            this._mainLayer.render();
        }

        // --------------------------------------------------------------------------
        public create() {
            this.stage.backgroundColor = 0xC0C0C0;

            this.camera.bounds = null;

            //Generator.JumpTables.setDebug(true, GoblinRun.Global);
            Generator.JumpTables.instance;

            // this.game.add.sprite(0, 0, Generator.JumpTables.debugBitmapData);

            this._mainLayer = new MainLayer(this.game, this.world);
        }

        // --------------------------------------------------------------------------
        public update() {
            this.camera.x += this.time.physicsElapsed * Generator.Parameters.VELOCITY_X / 2;
            this._mainLayer.generate(this.camera.x / Generator.Parameters.CELL_SIZE);
        }
    }
}
```

On top we added variable _mainLayer, that holds reference to class we wrote in previous part. render() method is called by Phaser engine on every frame. It just calls render method of main layer, which will debug draw all physics bodies attached to wall sprites.

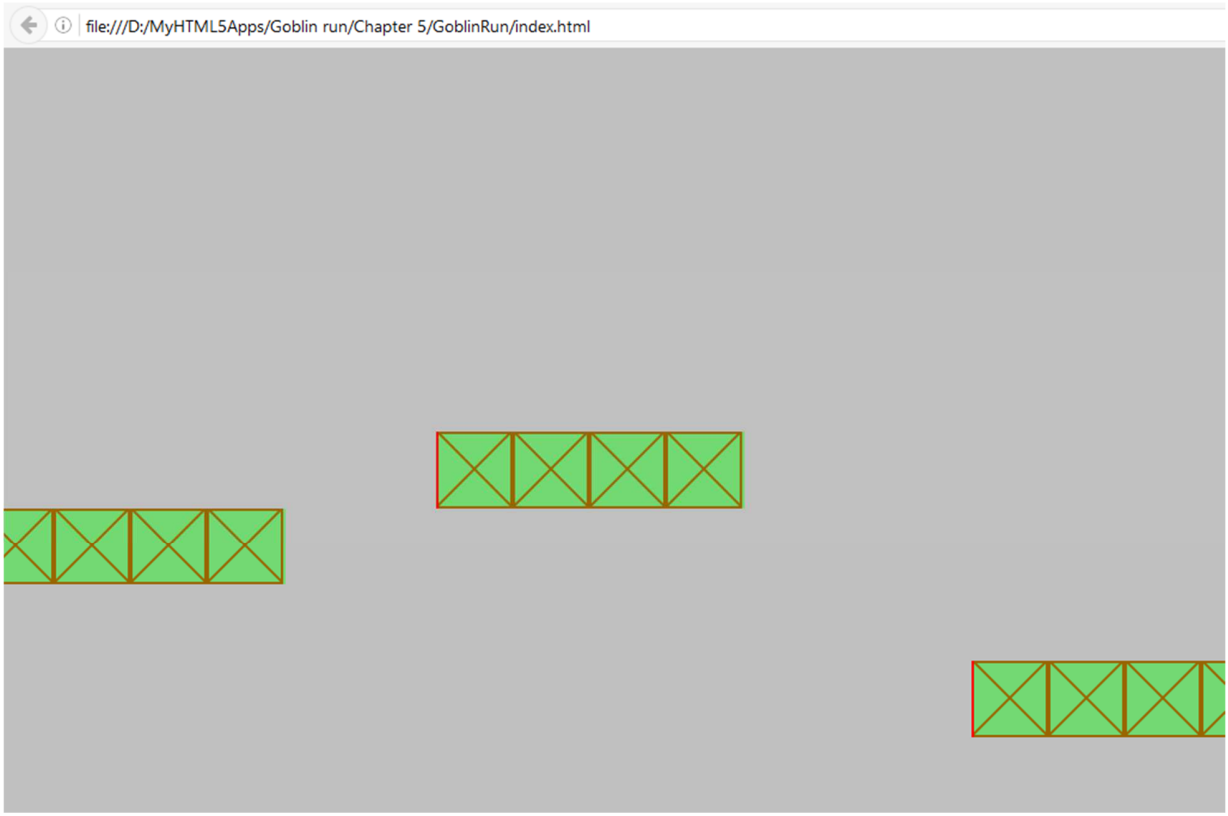
In create we change background color to neutral grey. We cancel camera bounds by setting it to null. By default, camera bounds are set to world bounds and these are set to game width and height by default. As camera cannot move outside its bounds, we could not scroll across our infinite level. Setting camera bounds to null allow you to place camera wherever you want.

We do not need debug output from jump table, so it is commented out.

Instance of MainLayer is created and added as child to world object.

In update() method camera's x is regularly updated with velocity defined in Parameters. It is divided by two to slow scrolling down a little, so we can enjoy fruits of our work. MainLayer's generate method is called on every frame with current camera position converted into tiles.

Compile and run game and you should see result like this:



Hmmm, it works, but ... it is not nice. Platforms are too random. Imagine one word for what you are missing in it. Is it "pattern"? Humans have sense for symmetry and patterns. This is exactly what we will do when we start beautifying of levels – we will bring some patterns into it.

Summary

In this chapter we made big step forwards. We finally have something that looks like game. We created generator and employed it to use jump table calculated in previous chapter. Although, it is still just kind of movie now as there is no interaction. I will notice here again – jump table is calculated on first call to instance and is based on parameters in Parameters.ts file. These parameters can be changed and it will have direct impact on generated platforms (you can for example allow higher jumps).

6. Adding player

In this chapter we will add interactivity to our game. We will add player, who will be controlled either with keyboard or mouse.

6.1 Loading player assets

First, we will load temporary placeholder sprite, we will use for player. Add sprite, like the one below, to assets folder and name it Player.png. You can copy it from Resources/_RES/Sprites. Dimensions of it are given with parameters PLAYER_BODY_WIDTH (=30) and PLAYER_BODY_HEIGHT (= 90) in Parameters.ts.

Open Preload.ts and add bold line to it:

```
        public preload() {
            this.load.image("Block", "assets/Block.png");
            this.load.image("Player", "assets/Player.png");
        }
```

6.2 Adding Player class

For player we will create new class. For now, it will be simple one, but later we can expand it with further features. Create file Player.ts in Game folder and put following lines into it:

```
namespace GoblinRun {

    export class Player extends Phaser.Sprite {

        // ---------------------------------------------------------------------
        public constructor(game: Phaser.Game) {
            super(game, 0, 0, "Player");

            // center player sprite horizontally
            this.anchor.x = 0.5;

            // enable physics for player
            game.physics.arcade.enable(this, false);

            // allow gravity
            let body = <Phaser.Physics.Arcade.Body>this.body;
            body.allowGravity = true;
        }
    }
}
```

We are extending Phaser.Sprite and in constructor we do small setup. First we center sprite horizontally with setting its x anchor to 0.5. Then we enable physical body for it and allow gravity, so it can jump and fall off platforms.

6.3 Jumping

As Player class is ready to be used, we will create instance of it and place it into game world. We let player run automatically forward and make him jump when spacebar or left mouse button is pressed. For this we have to read input from keyboard and mouse. Following code is quite along again, so we will go through it part by part. Open Play.ts and put new private variables on top of it (bold ones):

```
export class Play extends Phaser.State {

    private _mainLayer: MainLayer;

    // player
    private _player: Player;
    private _jumpTimer: number = 0;

    // status
    private _gameOver: boolean = false;

    // input
    private _jumpKey: Phaser.Key;
    private _justDown: boolean = false;
    private _justUp: boolean = false;
```

- _player is instance of Player class,
- _jumpTimer is help variable. It will prevent another jump for very short time period after player starts jumping,
- _gameOver is simple boolean that is false until player dies (either hits platform or falls down),
- _jumpKey is Phaser.Key object for key we chose as jump key (spacebar),
- _justDown and _justUp are variables that groups input from both keyboard and mouse. _justDown is true if spacebar or mouse button gets pressed. _justUp is the same for keyboard or button release. Both of it remains true until it is processed with our code.

Add new lines (the bold ones) to create() method:

```
    public create() {
        this.stage.backgroundColor = 0xC0C0C0;

        // camera
        this.camera.bounds = null;

        // physics
        this.physics.arcade.gravity.y = Generator.Parameters.GRAVITY;


        //Generator.JumpTables.setDebug(true, GoblinRun.Global);
        Generator.JumpTables.instance;

        // this.game.add.sprite(0, 0, Generator.JumpTables.debugBitmapData);

        this._mainLayer = new MainLayer(this.game, this.world);
```

```
            // set player
            this._player = new Player(this.game);
            this._player.position.set(96, 64 * 1);
            this.world.add(this._player);


            // input
            // key
            this._jumpKey = this.game.input.keyboard.addKey(Phaser.KeyCode.SPACEBAR);
            // mouse
            this.game.input.onDown.add(function () {
                this._justDown = true;
            }, this);
            this.game.input.onUp.add(function () {
                this._justUp = true;
            }, this);
        }
```

Here we first set gravity for Arcade physics engine according to value in Parameters. Then we create instance of Player and set its position. For x it is 96, which means our player will be 1.5 (96 / 64 = 1.5) tiles from left screen border. This will give player enough space and good view what platforms come from right. y position is set to 64 (1 tile) but our initial platform, as we set it in main layer in previous chapter, is on tile y = 5. So, player will start with small fall, which looks dynamically.

Next we create jump key. It is Phaser.Key object and later we can question it, whether it is down or just down, etc. For mouse input, we hook two small callbacks to signals (Phaser.Signal) that are sent whenever mouse is pressed (onDown) or released (onUp). On mobile devices this reads touch down and touch up events. Generally, onDown and onUp are dispatched whenever pointer is down or released. All we do in callbacks is we set _justDown or _justUp to the true. We are not interested in any additional information, that are passed along with this event. But if you needed for example to know, which pointer (as on mobile you can have more than one) sent event or where on screen pointer is located or other information, you could define your callback handler like this:

```
this.game.input.onDown.add(
  function(pointer: Phaser.Pointer, DOMevent: any) {
     ... your code ...
  },
this);
```

Finding what parameters are sent by Phaser when signals are dispatched is sometimes tricky. Best method is to find it in source code. But, back to our game.

Modify update() method to following:

```
        public update() {
            if (!this._gameOver) {
                this.updatePhysics();

                // move camera
                this.camera.x = this._player.x - 96;

                // generate level
                this._mainLayer.generate(this.camera.x / Generator.Parameters.CELL_SIZE);

                // check if player is still on screen
                if (this._player.y > this.game.height) {
                    console.log("GAME OVER");
                    this._gameOver = true;
```

```
            }
        }
    }
```

It will execute its content as long as there is not game over. Physics adjustments, like player velocity and jump impulses are separated in updatePhysics() method. Here we left camera position adjustment – it will be 96 pixels behind player. As player is moving through the world, it would get soon out of the screen. Camera position adjustment ensures it will follow him.

On next line we call generate() method in MainLayer. It may or may not do something. It all depends on whether we run out of generated part of level and there is need to create new pieces. It also checks whether to clean some old tiles.

Last part is check for player's y position. If it is higher than height of game, then player fell off screen and we flag game over.

```
    private updatePhysics(): void {
        let body = <Phaser.Physics.Arcade.Body>this._player.body;
```

Put body reference into local body variable. It has two reasons. We do not have to write this._player.body every time, we want do something with it and as we cast it to Phaser.Physics.Arcade.Body, we get (at least in Visual Studio) intellisense code completion, because IDE knows type of it (by default, body type is any).

```
        // collision with walls
        let wallCollision = this.physics.arcade.collide(this._player, this._mainLayer.walls);
```

This single line checks collisions between player and walls. Now, you will get error, as walls are private to MainLayer. One possibility is to make it public or write get accessor in MainLayer class. We will do second after we finish updatePhysics() method.

When calling arcade.collide() you can use its results in several levels of interest:

- interested only whether collision happen or not? Returned value is true if any,
- interested in on which side of your object (left, right, top, bottom) collision happen? Then examine body.touching object after the call. This is what we will do, because it is OK, if we are colliding with walls with player's bottom – we are running on it. But it is not OK if our player hits wall with his nose (from right),
- interested in information which particular object collided with player. In this case you can add callbacks to parameters of the call and these callbacks will be called for every collision that occurs. We will use it later in game when collecting gold and special bonus jumps.

```
        // move
        if (wallCollision && body.touching.right) {
            body.velocity.set(0, 0);
            this._gameOver = true;
            console.log("GAME OVER");
            return;
        }
```

As said, some collisions are OK (from bottom), some are not (from right). So, we check results stored in body.touching object. If we hit wall, we clear player's velocity to stop it and flag game over. Stopping does not mean here that player stays frozen on spot. Remember, he is affected with gravity. What stops is horizontal move and also vertical velocity is cleared for this frame. But as physics simulation is still running, he will immediately start falling in following frames. If you wanted him to freeze, then add body.allowGravity = false;

```
        // set body velocity
        body.velocity.x = Generator.Parameters.VELOCITY_X;
```

If not game over, we keep player running with velocity set in Parameters.ts file.

```
        // read keyboard
        if (this._jumpKey.justDown) {
            this._justDown = true;
        }
        if (this._jumpKey.justUp) {
            this._justUp = true;
        }
```

Here we read key we defined as jump key. We do the same as in onDown and onUp callbacks for pointer. So, _justDown and _justUp variables are set to true regardless of input source.

```
        let jumpTable = Generator.JumpTables.instance;

        // start jump
        if (this._justDown && body.touching.down && this.game.time.now > this._jumpTimer) {
            body.velocity.y = jumpTable.maxJumpVelocity;
            this._jumpTimer = this.game.time.now + 150;
            this._justDown = false;
        }

        // stop jump
        if (this._justUp && body.velocity.y < jumpTable.minJumpVelocity) {
            body.velocity.y = jumpTable.minJumpVelocity;
        }
```

Jump is started if set of conditions is met. Player pressed either jump key or mouse button, body is standing on platform and short time since last successful jump passed. If all of this is true, then body is given velocity impulse for highest jump as read from jumpTable. Small delay before next jump is added. This small delay is not here to prevent player from jumping. It is more or less safety belt for weird thing that may happen. Imagine your player is jumping very slowly (in water or on moon). Physics engine can still report collision with ground several frames for some rounding or so, and player may press jump button several times during it. Nothing horrible would happen in this small game, but it is better to avoid possible problems in advance.

Jump stops if key or mouse button is released and if player is still going up. In fact, jump is shortened instead of instant stopping. In such case, velocity is set to velocity of smallest jump.

From above it is clear, it is almost impossible to make the smallest jump. Player would have to press and release key really very quickly.

We did not end with jumps yet. We want to make jumping as much convenient to player as possible. We want player to press key or button for every jump he wants to make – holding it will not fire another one. First attempt would be to set _justDown only if on ground and ignore presses if still in the air. But this does not behave well. If player needs to make several jumps in fast pace one after another, then he does not recognize, whether still 3 pixels above ground or already on. It then results in feeling that game ignores some jumps. So, we will clear _justDown in case player is going up and record request for next jump during fall. If key is still pressed when player lands, it will start new jump immediately. If player releases key before landing, then _justDown is cleaned and no jump is made.

```
            // if down pressed, but player is going up, then clear it
            if (body.velocity.y <= 0) {
                this._justDown = false;
            }

            // if key is released then clear down press
            if (this._justUp) {
                this._justDown = false;
            }

            // just up was processed - clear it
            this._justUp = false;
        }
```

If going up, we ignore down presses. If key is released, then clear jump request if any. Always clear _jumpUp after it was processed by method.

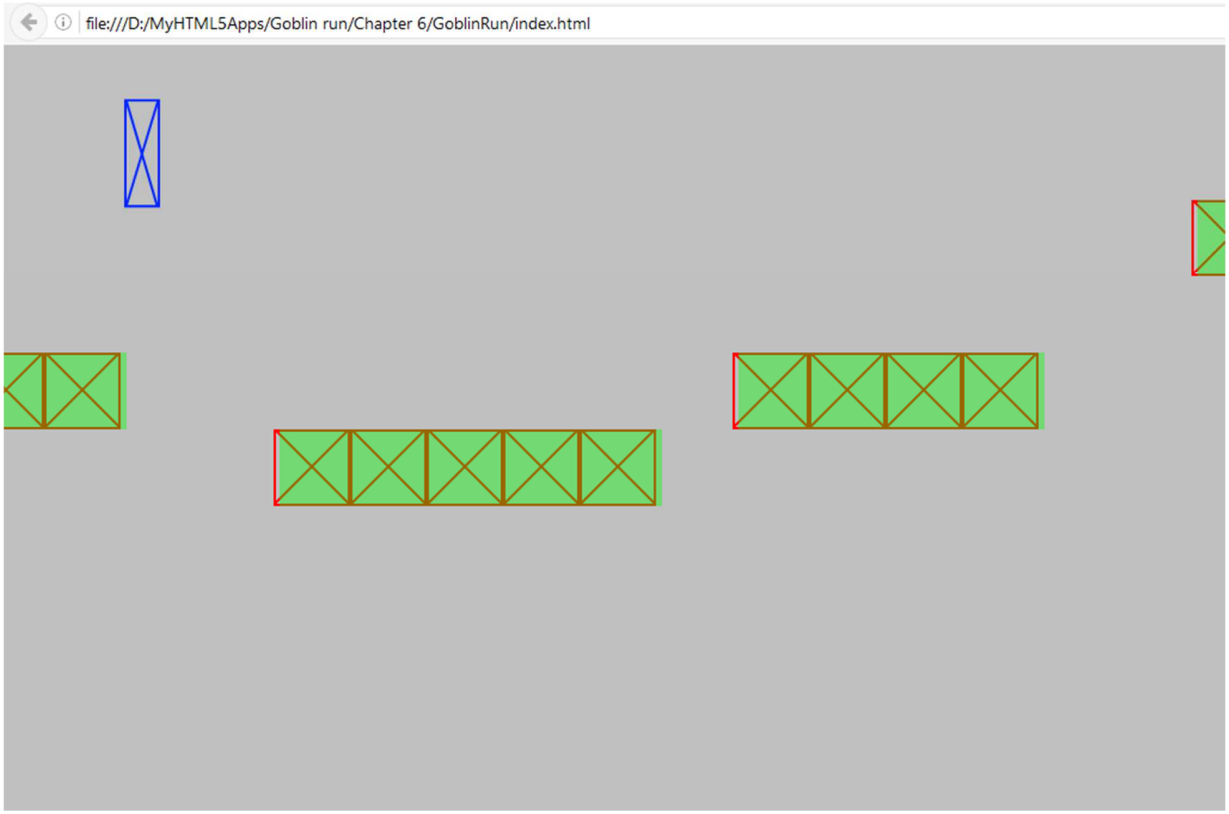
Last thing we have to do before we run the game, is to add public read only property to MainLayer.ts. Open it and in the end add these lines:

```
        public get walls(): Phaser.Group {
            return this._walls;
        }
```

This will get rid of the error in collide call.

6.4 Running game

Compile and run the game. This is what you should see:



You can play it – jump from platform to platform. Make small jumps with short presses or high ones with long presses. Let player fall down off screen or hit platform. In both cases you should get "GAME OVER" in console.

Summary

In this chapter we added interactivity to game. It is now playable and you can try how long you can survive. Level is still ugly – from generation point of view. Platforms are too random. In next chapter we will make it look better.

7. Enhancing generator

In this chapter we will do two things. We will add some way how to control difficulty of game and we will enhance generator to produce more visually pleasing results. More pleasing means not so random.

7.1 Difficulty parameters

To make game more challenging, we should think of what difficulty parameters we could have. Here we will identify two and in next chapters we will add next ones. Both of them will be bind to distance in tiles player has reached:

- platforms will be generated shorter and shorter,
- horizontal gap between platforms in the beginning of game will be smaller to make jumps easier for player.

We will create some constant values for these difficulty parameters in Parameters.ts file. So, open it and add these lines:

```
        // --- DIFFICULTY ---
        // platform length
        public static PLATFORM_LENGTH_MIN = 2;
        public static PLATFORM_LENGTH_MAX = 5;
        public static PLATFORM_LENGTH_DECREASER_MIN = 0;
        public static PLATFORM_LENGTH_DECREASER_MAX = -2;
        public static PLATFORM_LENGTH_DECREASER_START_TILE = 100;
        public static PLATFORM_LENGTH_DECREASER_END_TILE = 200;

        // jump length
        public static JUMP_LENGTH_DECREASER_MIN = -1;
        public static JUMP_LENGTH_DECREASER_MAX = 0;
        public static JUMP_LENGTH_DECREASER_START_TILE = 0;
        public static JUMP_LENGTH_DECREASER_END_TILE = 50;
```

Parameters for platform length say what is minimum and maximum length of it. In our case 2-5 tiles. With growing distance player passed, we will decrease value for maximum. In the beginning decreaser value will be zero and during game it can grow up to -2. Game will be most difficult when platforms are only 2-3 tiles long. Start tile and end tile says when decreasing of platform length starts and when it reaches its maximum. We will start calculations on distance 100 tiles and when 200 tiles distance is reached, our upper bound for maximum platform length will be decreased by -2.

For jump length we define similar constants. We do not have to define minimum and maximum jump distance as it is in our jump table for every height level. We just need to say how much shorten it. We will start with -1 at beginning and end with 0 at tile 50.

7.2 Difficulty class

With our parameters ready we can create new source file Difficulty.ts in Generator folder. Put following code part by part into it:

```
namespace Generator {

    export class Difficulty {

        private _rnd: Phaser.RandomDataGenerator;

        // platform length
        private _platformLengthDecrease: number;
        // jump length
        private _jumpLengthDecrease: number;

        // ---------------------------------------------------------------------
        public constructor(rnd: Phaser.RandomDataGenerator) {
            this._rnd = rnd;

            // maximum length of platform
            this._platformLengthDecrease = Parameters.PLATFORM_LENGTH_DECREASER_MIN;
            // jump width decreaser to make jumps easier in game beginnig
            this._jumpLengthDecrease = Parameters.JUMP_LENGTH_DECREASER_MIN;
        }
```

In future we will need random numbers generator. We will use the one game already has. So, it is passed into constructor. We also define two private variables to hold actual values for our difficulty parameters. In constructor it is initialized with starting values.

```
        public get platformLengthDecrease(): number {
            return this._platformLengthDecrease;
        }

        // ---------------------------------------------------------------------
        public get jumpLengthDecrease(): number {
            return this._jumpLengthDecrease;
        }
```

Just two accessors to get values of private variables.

```
        public mapLinear(x: number, a1: number, a2: number, b1: number, b2: number): number {
            x = Phaser.Math.clamp(x, a1, a2);
            return Phaser.Math.mapLinear(x, a1, a2, b1, b2);
        }
```

Phaser.Math has method mapLinear() that maps value x from interval a1 to a2 into interval b1 to b2. This is what we need. For platform length we pass current tile and we want to map it from interval given with start and end tile into interval with our difficulty adjustment (0 ... -2 in case of platform length decrease). Unfortunately, this method does not check whether x is inside a1 to a2 interval. For this reason, we create our own method mapLinear(), that wraps Phaser.Math.mapLinear() method, but passes x value for current tile after it is clamped into a1 to a2 interval.

```
        public update(tileX: number): void {
            // platform length
            this._platformLengthDecrease = Math.round(this.mapLinear(tileX,
                Parameters.PLATFORM_LENGTH_DECREASER_START_TILE,
Parameters.PLATFORM_LENGTH_DECREASER_END_TILE,
                Parameters.PLATFORM_LENGTH_DECREASER_MIN,
Parameters.PLATFORM_LENGTH_DECREASER_MAX));
```

```
            // jump length
            this._jumpLengthDecrease = Math.round(this.mapLinear(tileX,
                Parameters.JUMP_LENGTH_DECREASER_START_TILE,
Parameters.JUMP_LENGTH_DECREASER_END_TILE,
                Parameters.JUMP_LENGTH_DECREASER_MIN, Parameters.JUMP_LENGTH_DECREASER_MAX));
        }
```

In update we recalculate both parameters. We call our version of mapLinear() method and round result to whole number.

```
        public toString(): string {
            return "platformLengthDecrease: " + this._platformLengthDecrease +
                ", jumpLengthDecrease: " + this.jumpLengthDecrease;
        }
    }
}
```

With simple helper method we can print actual difficulty values to console.

7.3 Enhancing Generator

In next step we will make enhancements to Generator class. We will adjust it several times more during this book when we add new features. But for now, we will make it more flexible primarily. Idea is to make it react to difficulty parameters. Second change is to control generating more tightly with "forced" parameters. These parameters, if passed into method, will be used instead of calculations made inside.

First add bold line on top of Generator.ts file:

```
namespace Generator {

    const UNDEFINED = -10000;

    export class Generator {
```

This line defines constant UNDEFINED, which we will use later. Now, change method signature for generate():

```
        private generate(lastPosition: Phaser.Point, difficulty: Difficulty,
            length: number, offsetX: number, offsetY: number, bonusJump: boolean): Piece {
```

Newly, we are passing difficulty object into method as well as some parameters we want new piece to have. It means, that piece generation may not be so random as before. We can, for example, force offset X to be 2 in new piece. Here is place, where UNDEFINED constant is used. If value we pass is equal to this, then generator knows, it is up to it to calculate value. If not equal UNDEFINED, then passed value will be used.

We also changed method access from public to private. This will make sense later after we make additional changes in generator. We will not need to call this method directly from game anymore.

As lot of small changes was done in method's code. I will put whole new code here and bold changed or added parts:

```
            let piece = this.createPiece();
```

```
            let ubound = Parameters.UBOUND;
            let lbound = Parameters.LBOUND;


            // Y POSITION
            // how high can jump max
            let minY = this._jumpTables.maxOffsetY();
            // how deep can fall max
            // let maxY = lbound - ubound;
            let maxY = -minY;
```

With this change we limit maximum vertical step down that can be done by Generator. Previously, next platform could be generated very deep compared to previous one. Now, we limit it to the same value that is possible for step up. Flow of platforms will not be so jumpy now.

```
            // clear last y from upper bound, so it starts from 0
            let currentY = lastPosition.y - ubound;

            let shiftY = offsetY;
            if (shiftY === UNDEFINED) {
                // new random y position - each y level on screen has the same probability
                shiftY = this._rnd.integerInRange(0, lbound - ubound);
                // substract currentY from shiftY - it will split possible y levels to negative
                // (how much step up (-)) and positive (how much to step down (+))
                shiftY -= currentY;
                // clamp step to keep it inside interval given with maximum
                // jump offset up (minY) and maximum fall down (maxX)
                shiftY = Phaser.Math.clamp(shiftY, minY, maxY);
            }

            // new level for platform
            // limit once more against game limits (2 free tiles on top, 1 water tile at bottom)
            let newY = Phaser.Math.clamp(currentY + shiftY, 0, lbound - ubound);

            // shift by upper bound to get right y level on screen
            piece.position.y = newY + ubound;
            // offset of new piece relative to last position (end position of last piece)
            piece.offset.y = piece.position.y - lastPosition.y;
```

We can now force offset y to specific value with method's argument. Here we check if this value was passed into method or whether we want generator to calculate it for us.

```
            // X POSITION
            let shiftX = offsetX;
            // calculate if offsetX is not forced or offsetY was forced, but final value is
different
            if (shiftX === UNDEFINED || (offsetY !== UNDEFINED && offsetY !== piece.offset.y)) {
                let minX = this._jumpTables.minOffsetX(piece.offset.y);
                let maxX = this._jumpTables.maxOffsetX(piece.offset.y);

                // decrease maximum jump distance with jump decreaser in difficulty to
                // make jumps easier in the beginning of game
                // But be sure it does not fall under minX
                maxX = Math.max(minX, maxX + difficulty.jumpLengthDecrease);

                // position of next tile in x direction
                shiftX = this._rnd.integerInRange(minX, maxX);
            }
```

```
            // new absolute x position
            piece.position.x = lastPosition.x + shiftX;
            // offset of new piece relative to last position (end position of last piece)
            piece.offset.x = shiftX;
```

We do the same for offset x. Beside this, we read amount by which maximum distance should be decreased from difficulty object.

```
            // LENGTH
            if (length !== UNDEFINED) {
                piece.length = length;
            } else {
                // decrease maximum length of platform with difficulty progress
                piece.length = this._rnd.integerInRange(Parameters.PLATFORM_LENGTH_MIN,
                    Parameters.PLATFORM_LENGTH_MAX + difficulty.platformLengthDecrease);
            }
```

For platform length we again check if generator is forced to use some value passed as method's argument or whether it has to calculate it. Calculation takes into account current difficulty and decreases maximum platform length.

```
            console.log(difficulty.toString());

            // RESULT
            this._lastGeneratedPiece = piece;
            return piece;
        }
```

You can log difficulty used to generate piece or comment it out later.

7.4 Pieces queue

We will continue to make changes into Generator.ts. So far, we generated only one piece and then turned it into tiles in game. We also said, that we want to make our level visually more pleasing. For this we already limited maximum downfall. But it is not enough. What is visually pleasing for people are patterns and symmetry. For this, we will generate more pieces at time and put them into queue, that will be processed piece by piece, until it is empty. If empty, game will ask for new pieces. These pieces will form simple random patterns repeated multiple times. Our pieces were so far only platforms and this will be true also for next lines. Later in book we will generate other types of pieces, but for now it is all about platforms.

Before we do changes into Generator class, add these lines into Parameters.ts:

```
        // --- GENERATOR ---
        // probability to generate random piece in percent
        public static GENERATE_RANDOM = 50;
        // keep length of all platforms in pattern the same? (in percent)
        public static KEEP_LENGTH_IN_PATTERN = 75;
```

To keep level interesting, we still keep some randomness in it. Probability to generate completely random piece is 50%.

Second parameter will make sense, when we generate patterns. It sets probability, that all platforms in pattern have the same length.

Now, back to Generator.ts. Add these new private variables on top of it:

```
        // pieces queue
        private _piecesQueue: Generator.Piece[] = [];
        private _piecesQueueTop: number = 0;
        private _hlpPoint: Phaser.Point = new Phaser.Point();
```

Pieces queue will be simple array of Pieces. In _piecesQueueTop we track current length of queue. _hlpPoint is help variable and as it is object, we allocate memory for it only once and keep reference to it here to prevent unnecessary memory allocation every time we need it.

Next, add these methods:

```
        public get hasPieces(): boolean {
            return this._piecesQueueTop > 0;
        }

        // ------------------------------------------------------------------------
        private addPieceIntoQueue(piece: Generator.Piece): void {
            // put new piece into queue and increase its length
            this._piecesQueue[this._piecesQueueTop++] = piece;
        }

        // ------------------------------------------------------------------------
        public getPieceFromQueue(): Generator.Piece {
            // if no pieces in queue then return null
            if (this._piecesQueueTop === 0) {
                return null;
            }

            // get first piece in queue
            let piece = this._piecesQueue[0];

            // shift remaining pieces left by 1
            for (let i = 0; i < this._piecesQueueTop - 1; i++) {
                this._piecesQueue[i] = this._piecesQueue[i + 1];
            }

            // clear last slot in queue and decrease queue top
            this._piecesQueue[--this._piecesQueueTop] = null;

            return piece;
        }
```

hasPieces() is getter that returns whether queue is not empty. addPieceIntoQueue() simply adds piece into queue and increases its length. In getPieceFromQueue() we first check if queue is empty or not. Then we pick first item and shift rest to the left by one position. Last position is cleared (set to null) and picked item is returned.

In setPiece() do only small change in bold:

```
        public setPiece(x: number, y: number, length: number, offsetX: number = 0, offsetY: number =
0): Piece {
            let piece = this.createPiece();

            piece.position.set(x, y);
            piece.offset.set(offsetX, offsetY);
            piece.length = length;
```

```
            this.addPieceIntoQueue(piece);

            return piece;
        }
```

So far, our game called generate() method, but we made it private. From now on, game will call new method generatePieces(). It is up to this method, whether it will generate one random piece or several pieces forming pattern. Either is chosen, all generated pieces end in queue.

```
        public generatePieces(lastTile: Phaser.Point, difficulty: Difficulty): void {
            let probability = this._rnd.integerInRange(0, 99);

            if (probability < Parameters.GENERATE_RANDOM) {
                this.generateRandomly(lastTile, difficulty);
            } else {
                this.generatePattern(lastTile, difficulty);
            }
        }
```

Passed parameters are position of last tile and current difficulty. Probability of setting completely random piece is given with GENERATE_RANDOM parameter. Here is one point for your future improvements. Instead of using fixed parameter, you can make decision based on some difficulty parameter and, for example, make level messier in later stages.

```
        private generateRandomly(lastTile: Phaser.Point, difficulty: Difficulty): void {
            let piece = this.generate(lastTile, difficulty, UNDEFINED, UNDEFINED, UNDEFINED, false);

            // add to queue
            this.addPieceIntoQueue(piece);
        }
```

This is what we did so far – it generates random platform. We do not force new piece to use any values, we let it completely on generator. And after piece is generated, it is put into queue.

Next method generates pattern and is more interesting. For our small game patters will be really simple and their length as well as number of repeating are given with hardcoded values. Here is another point for improvement. You can bind both to some difficulty parameters to make your levels more interesting.

```
        private generatePattern(lastTile: Phaser.Point, difficulty: Difficulty): void {
            // save index of first new piece
            let oldQueueTop = this._piecesQueueTop;
            // where to start generating
            let hlpPos = this._hlpPoint;
            hlpPos.copyFrom(lastTile);
```

As we will repeat pattern multiple times, we need to save position of first new piece in it. hlpPos will help us to track last tile on each generated platform.

```
            // same length for all pices?
            let length = UNDEFINED;
            if (this._rnd.integerInRange(0, 99) < Parameters.KEEP_LENGTH_IN_PATTERN) {
                length = this._rnd.integerInRange(Parameters.PLATFORM_LENGTH_MIN,
                    Parameters.PLATFORM_LENGTH_MAX + difficulty.platformLengthDecrease);
            }
```

Randomly choose, whether all platforms in pattern will have the same length or whether length will be UNDEFINED and generator will randomize it. If length is chosen to be the same, it is calculated with regard to current difficulty.

```
            // how many pieces to repeat in pattern
            let basePices = 2;

            for (let i = 0; i < basePices; i++) {
                let piece = this.generate(hlpPos, difficulty, length, UNDEFINED, UNDEFINED, false);

                hlpPos.copyFrom(piece.position);
                // get last tile of piece
                hlpPos.x += piece.length - 1;

                // add to queue
                this.addPieceIntoQueue(piece);
            }
```

As said previously, number of platforms in pattern for our small game is hardcoded and it will be two. These two platforms are generated randomly, but generator is forced to use length we calculated before (which still may be UNDEFINED).

After each platform is generated, hlpPos is adjusted to point on last tile of it. In the end platform is added into queue.

```
            // repeat pattern X times
            let repeat = 1;

            for (let i = 0; i < repeat; i++) {

                // repeat all pieces in pattern
                for (let p = 0; p < basePices; p++) {
                    // get first piece in pattern to repeat as template
                    let templetePiece = this._piecesQueue[oldQueueTop + p];

                    // replicate it
                    let piece = this.generate(hlpPos, difficulty, length,
                        templetePiece.offset.x, templetePiece.offset.y, false);

                    hlpPos.copyFrom(piece.position);
                    hlpPos.x += piece.length - 1;

                    // add to stack
                    this.addPieceIntoQueue(piece);
                }
            }
        }
```

When we have basic pattern we can repeat it. In our game we repeat it only once. We force generator to generate new pieces with the same length and offset as previous pieces have. We must use generate() method and not simply copy it, because it ensures, that new pieces are kept in vertical bounds for platforms and are reachable.

7.5 Changes in MainLayer

We are almost ready to try our new generator. But before that, we have to do small adjustment in MainLayer.ts

First, comment out reference to Piece, we will not need it anymore. And add private variable holding reference to Difficulty:

```
        // piece generated with generator
        // private _piece: Generator.Piece = null;

        private _difficulty: Generator.Difficulty;
```

In constructor create new difficulty (line in bold):

```
            // platforms generator
            this._generator = new Generator.Generator(game.rnd);

            // object that holds level difficulty progress
            this._difficulty = new Generator.Difficulty(game.rnd);

            // pool of walls
                   :
```

And in the end of it do not store generated piece in _piece variable anymore as we commented it out a few lines above:

```
            // set initial tile for generating
            // this._piece = this._generator.setPiece(0, 5, 10);
            this._generator.setPiece(0, 5, 10);
```

Last method, that needs changes is generate(). Changes are in particular branches of case statement.

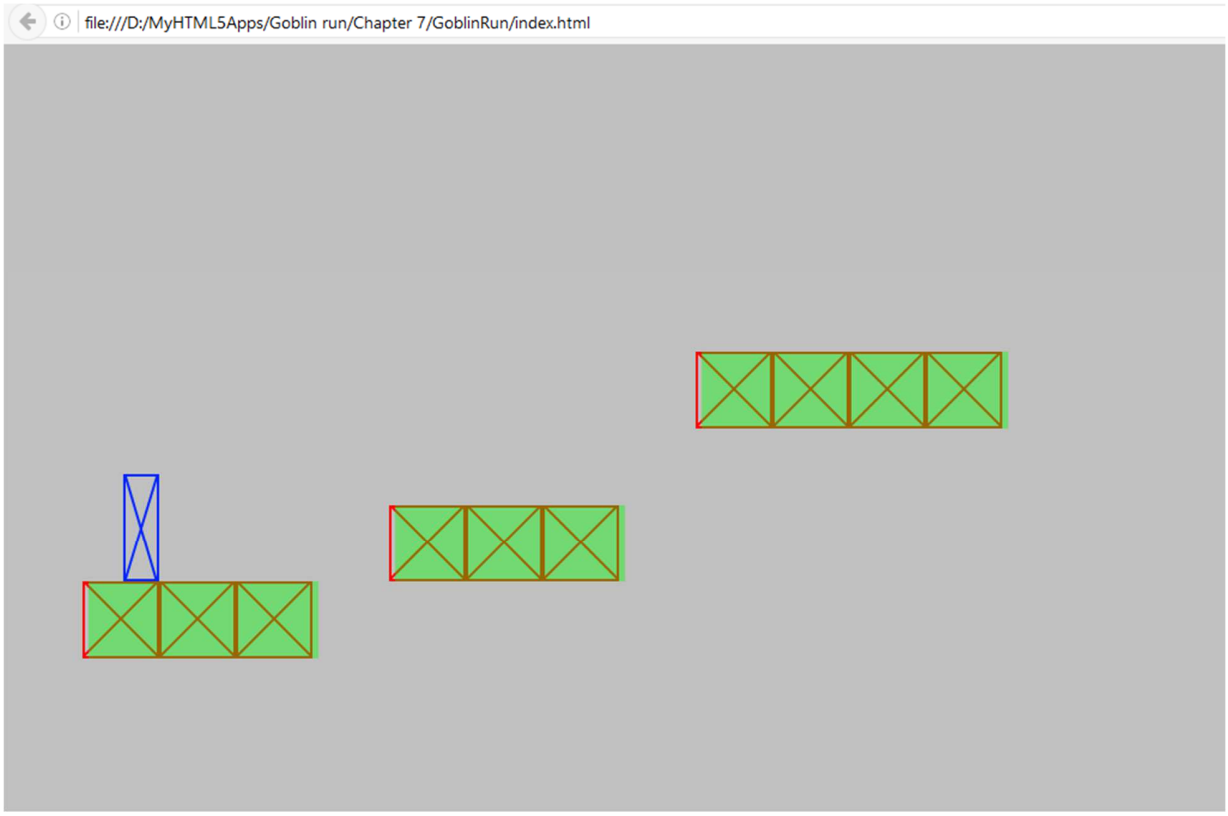
```
                    case eGenerateState.PROCESS_PIECE:
                        {
                            // check if queue not empty - should never happen
                            if (!this._generator.hasPieces) {
                                console.error("Pieces queue is empty!");
                            }

                            let piece = this._generator.getPieceFromQueue();

                            this._lastTile.copyFrom(piece.position);
                            let length = piece.length;

                            // process piece
                            while (length > 0) {
                                this.addBlock(this._lastTile.x, this._lastTile.y);

                                if ((--length) > 0) {
                                    ++this._lastTile.x;
                                }
                            }

                            // return processed piece into pool
                            this._generator.destroyPiece(piece);

                            // generate next platform
                            if (!this._generator.hasPieces) {
                                this._state = eGenerateState.GENERATE_PIECE;
                            }

                            break;
                        }
```

```
                case eGenerateState.GENERATE_PIECE:
                    {
                        this._difficulty.update(leftTile);

                        this._generator.generatePieces(this._lastTile, this._difficulty);

                        this._state = eGenerateState.PROCESS_PIECE;

                        break;
                    }
```

We changed processing of pieces to work with queue of pieces in generator. We do not change state while there are still pieces in queue. Once it is empty, we switch state to GENERATE_PICES. Before we ask generator for new pieces, we update difficulty. It is not necessary to update it more frequently.

7.6 Running game

It is time to compile and run game.



Give it some time and you will observe, that platforms are more organized. It is not so random now. Play long enough to see, that difficulty parameters work. After some time, platforms are getting shorter and shorter.

Summary

In this chapter we enhanced generator, so it produces much better results now. We are still looking at temporary placeholder graphics, which starts to be little boring. In next chapter we will change it for final platforms graphics.

8. Adding tile graphics

In this chapter we will add final graphics for platform tiles. First, we will examine tiles layout, define rules for placing it in level and then implement it in code.

8.1 Atlas

In files accompanying this book you will find folder Resources. In this folder on path _RES/Sprites/ you will find individual game sprites. Part of them form atlas we will use in this chapter. Atlas with its JSON metadata is already ready in folder _RES/export/. It is also in Chapter 8 folder in GoblinRun/assets/, so everything we need to do is load it during assets loading in Preload state.

Atlas was created with Spritor (former name PicOpt), which is my tool for creating atlases. This tool is in Resources folder and you can run it with Editor.bat. Tool is very simple and short but little outdated tutorial on using it is here: http://sbcgamesdev.blogspot.cz/2012/10/sprite-atlas-tool-part-i-creating-atlas.html. As you can see, name of it is pretty messy: Spritor – PicOpt – Editor. But there are two important things: tool works and produces TexturePacker compatible export and you do not have to use it. If you wish to recreate atlas by yourself, you can use any tool you are used to work with.

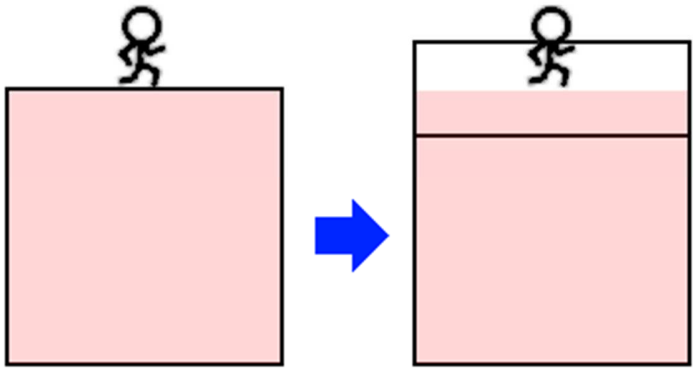
Why I am not using TexturePacker (https://www.codeandweb.com/texturepacker) or ShoeBox (http://renderhjs.net/shoebox/) as other people do? I created Spritor in deep past and it still works great. It also has some features that other tools do not have. Like it can load fonts from Littera (http://http://kvazars.com/littera/) and mix it into atlas (tutorial on this: http://sbcgamesdev.blogspot.cz/2016/03/phaser-tutorial-merging-fonts-into.html) or you can set and export sprite anchors directly in JSON metadata (http://sbcgamesdev.blogspot.cz/2015/04/phaser-tutorial-sprites-and-custom.html).

To summarize: atlas is ready to be loaded in assets folder of Chapter 8. Open Preload.ts and in preload() add line in bold:

```
        public preload() {
            this.load.image("Block", "assets/Block.png");
            this.load.image("Player", "assets/Player.png");

            this.load.atlas("Sprites", "assets/Sprites.png", "assets/Sprites.json");
        }
```

8.2 Tiles

Now, let's think of tiles layout in our game. Grid for our tiles is 64x64 pixels, but we want pseudo perspective projection. Instead of our player running on top of tile, we want him to run "inside" the tile:

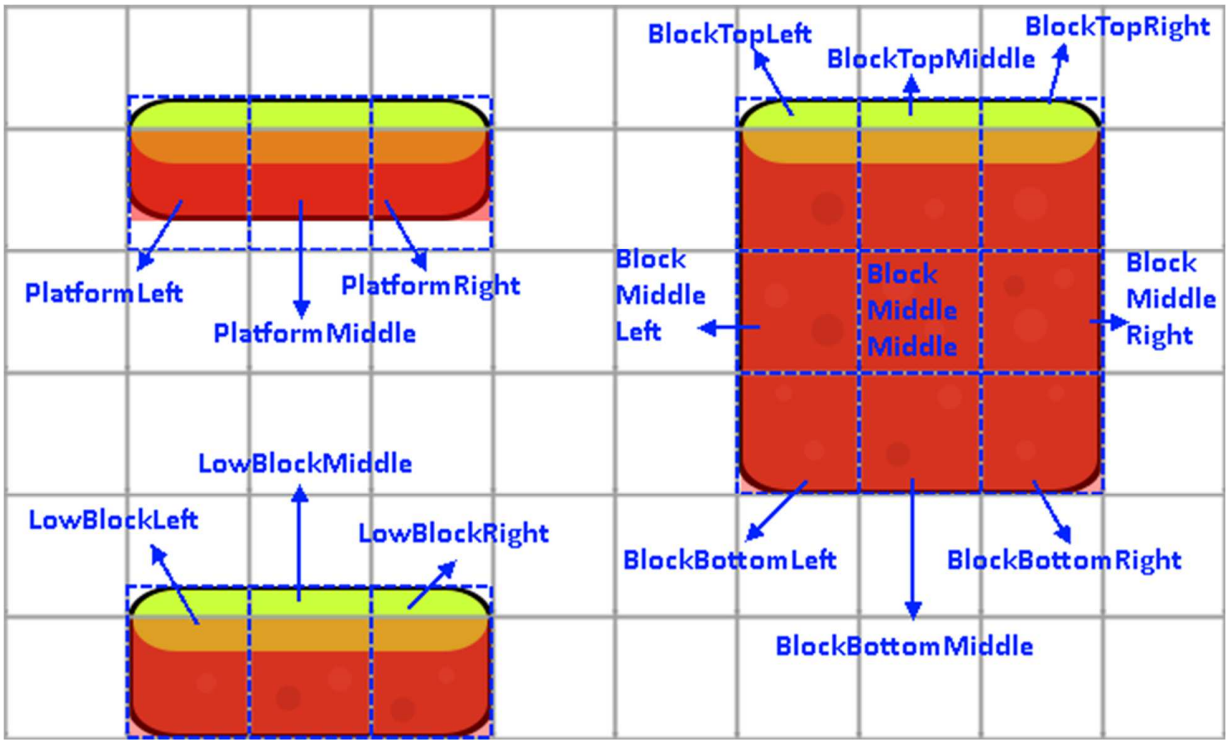


Collision for tile remains 64x64 as transparent red box shows.

For grater visual variability we will present platforms in two ways. Either as platform floating in the air or as big solid block standing in swamp. Platform will be formed with single line of tiles, while block will be formed with variable number of lines – from top down to swamp.

On image below, there is platform in top left. Blue labels are tile sprite names in atlas. Blue dashed box is size of tile sprite. Although our tiles are 64x64, we make it here larger to include top area (which is 16 pixels). Red transparent box over shows, what part we want to make work as physics body.

Solid block is on right. It is made of three rows of tiles. Middle row is repeated as many times as needed. Or zero times if solid block height is 2 (it is skipped). Top row tiles are 80 (64 + 16) pixels high again. There is one special case, if we need to create solid block high only 1 tile. For it we need special tiles as depicted in bottom left corner. We will call this block “low block”.



If we summarize it, we need some system to:

- set anchor of sprite, to shift it up,
- set position of body,
- set size of body

8.3 Block definitions

Add new file into Game folder and name it BlockDefs.ts. Start with code for new ITileDef interface:

```
namespace GoblinRun {

    export interface ITileDef {
        name: string;
        anchorX: number;
        anchorY: number;
        bodyOffsetX: number;
        bodyOffsetY: number;
        bodyWidth: number;
        bodyHeight: number;
    }
```

Definition of tile will need to have all these properties. And any object with these properties is considered to be ITileDef.

Add new class BlockDefs and put static 2D array of ITileDefs into it:

```
    export class BlockDefs {

        public static PLATFORM: ITileDef[][] = [
            [
```

```
                { name: "PlatformLeft", anchorX: 0, anchorY: 0.2, bodyOffsetX: 0, bodyOffsetY: 16,
bodyWidth: 64, bodyHeight: 48 },
                { name: "PlatformMiddle", anchorX: 0, anchorY: 0.2, bodyOffsetX: 0, bodyOffsetY: 16,
bodyWidth: 64, bodyHeight: 48 },
                { name: "PlatformRight", anchorX: 0, anchorY: 0.2, bodyOffsetX: 0, bodyOffsetY: 16,
bodyWidth: 64, bodyHeight: 48 }
            ]
        ];
```

PLATFORM 2D array has only one row and three columns, which is all we need to define three tiles for platform (left, middle, right tile). We define name for each tile, which is atlas name of sprite. We set anchor x and y, though we use only y in our game. Anchor y is set to 0.2 as height of tile is 80 pixels and we want to shift it up by 16 pixels (16 / 80 = 0.2). Last four properties define physics body of tile, its position from top left corner of tile sprite and its size.

Add two more 2D arrays for block and low block:

```
        public static BLOCK: ITileDef[][] = [
            [
                { name: "BlockTopLeft", anchorX: 0, anchorY: 0.2, bodyOffsetX: 0, bodyOffsetY: 16,
bodyWidth: 64, bodyHeight: 64 },
                { name: "BlockTopMiddle", anchorX: 0, anchorY: 0.2, bodyOffsetX: 0, bodyOffsetY: 16,
bodyWidth: 64, bodyHeight: 64 },
                { name: "BlockTopRight", anchorX: 0, anchorY: 0.2, bodyOffsetX: 0, bodyOffsetY: 16,
bodyWidth: 64, bodyHeight: 64 }
            ],

            [
                { name: "BlockMiddleLeft", anchorX: 0, anchorY: 0, bodyOffsetX: 0, bodyOffsetY: 0,
bodyWidth: 64, bodyHeight: 64 },
                { name: "BlockMiddleMiddle", anchorX: 0, anchorY: 0, bodyOffsetX: 0, bodyOffsetY: 0,
bodyWidth: 64, bodyHeight: 64 },
                { name: "BlockMiddleRight", anchorX: 0, anchorY: 0, bodyOffsetX: 0, bodyOffsetY: 0,
bodyWidth: 64, bodyHeight: 64 }
            ],

            [
                { name: "BlockBottomLeft", anchorX: 0, anchorY: 0, bodyOffsetX: 0, bodyOffsetY: 0,
bodyWidth: 64, bodyHeight: 64 },
                { name: "BlockBottomMiddle", anchorX: 0, anchorY: 0, bodyOffsetX: 0, bodyOffsetY: 0,
bodyWidth: 64, bodyHeight: 64 },
                { name: "BlockBottomRight", anchorX: 0, anchorY: 0, bodyOffsetX: 0, bodyOffsetY: 0,
bodyWidth: 64, bodyHeight: 64 }
            ]
        ];

        public static LOW_BLOCK: ITileDef[][] = [
            [
                { name: "LowBlockLeft", anchorX: 0, anchorY: 0.2, bodyOffsetX: 0, bodyOffsetY: 16,
bodyWidth: 64, bodyHeight: 64 },
                { name: "LowBlockMiddle", anchorX: 0, anchorY: 0.2, bodyOffsetX: 0, bodyOffsetY: 16,
bodyWidth: 64, bodyHeight: 64 },
                { name: "LowBlockRight", anchorX: 0, anchorY: 0.2, bodyOffsetX: 0, bodyOffsetY: 16,
bodyWidth: 64, bodyHeight: 64 }
            ]
        ];
    }
}
```

8.4 Generator signals

Following step may be surprising. We will again open generator and make small change into it. We will not change way it generates pieces, but we will let it dispatch signals when piece is generated. It will become clear soon why. Open Generator.ts file and add this on top of it:

```
    export class Generator {

        // signals
        // dispatch new piece, previous piece
        public onRandomPlatform: Phaser.Signal = new Phaser.Signal();
        // dispatch new piece, previous piece, position in pattern, repeat order, pattern base piece
        public onPatternPlatform: Phaser.Signal = new Phaser.Signal();
```

We will use Phaser.Signal to inform anyone who is interested when random platform or pattern platform is generated. Phaser.Signal keeps list of listeners and if we want to inform these listeners, we have to dispatch signal. Change method generateRandomly() to this:

```
        private generateRandomly(lastTile: Phaser.Point, difficulty: Difficulty): void {
            let prevPiece = this._lastGeneratedPiece;
            let piece = this.generate(lastTile, difficulty, UNDEFINED, UNDEFINED, UNDEFINED, false);

            // add to queue
            this.addPieceIntoQueue(piece);

            // dispatch signal - let listeners know, random platform has been generated
            // pass: new piece, previous piece
            this.onRandomPlatform.dispatch(piece, prevPiece);
        }
```

It now informs all listeners that new piece is generated and what was previous piece.

Do similar change into generatePattern on two places (changes in bold):

```
        private generatePattern(lastTile: Phaser.Point, difficulty: Difficulty): void {

                        :
                        :

            for (let i = 0; i < basePices; i++) {
                let prevPiece = this._lastGeneratedPiece;
                let piece = this.generate(hlpPos, difficulty, length, UNDEFINED, UNDEFINED, false);

                        :
                        :

                // add to queue
                this.addPieceIntoQueue(piece);

                // dispatch signal - let listeners know, pattern platform has been generated
                // pass: new piece, previous piece, position in pattern, repeat order, pattern base
piece
                this.onPatternPlatform.dispatch(piece, prevPiece, i, 0, null);
            }

                        :
                        :

            for (let i = 0; i < repeat; i++) {

                // repeat all pieces in pattern
                for (let p = 0; p < basePices; p++) {
```

```
                    let prevPiece = this._lastGeneratedPiece;

                        :
                        :

                    // add to stack
                    this.addPieceIntoQueue(piece);

                    // dispatch signal - let listeners know, pattern platform has been generated
                    // pass: new piece, previous piece, position in pattern, repeat order, pattern
base piece
                    this.onPatternPlatform.dispatch(piece, prevPiece, p, i + 1, templetePiece);
                }
            }
        }
```

For platforms in pattern we dispatch more information along with signal – what is new piece, what was previous one, order of piece in pattern, order of repeat of pattern and template piece if repeating pattern.

8.5 Block or platform?

As said before, platforms in our game can be visually represented in two ways. Either as platform floating in the air or as solid block planted in swamp. What appearance is chosen will be selected in MainLayer class.

We are not setting it in Generator. So, here comes point why we are sending signals when platform is generated. Why not add some “platformOrBlock” property to Piece class and set it in Generator? So far our generator is very generic. It can be set up with Parameters and it produces simple pieces of level. These pieces have only position, offset from previous piece and length (we will add other properties later). But whether platform is represented with platform or block is specific to game and its visual representation. What if our game platforms were represented only with blocks or only with platforms? What if we needed to set something game specific to every generated piece?

Putting game specific decisions into game code will keep generator clean and reusable for other games.

In MainLayer.ts we will do two sets of changes. First set will focus on processing signals from generator and second one on composing platforms and blocks from tiles. Open MianLayer.ts file and at top of it add code in bold:

```
namespace GoblinRun {

    const enum eGenerateState { PROCESS_PIECE, GENERATE_PIECE }

    interface IGamePiece extends Generator.Piece {
        isPlatform: boolean;
    }
```

With IGamePiece we define new interface that extends Piece class in generator. It adds game specific property isPlatform, that says whether piece is platform if true or block if false. When instance of Piece is created in generator, this property is not present in object. If you attempt to read it, you get “undefined”. We have to set it first. Good thing is, that if we treat Pieces as IGamePiece, we have type checked access to isPlatform property.

Go into constructor and under line creating generator instance subscribe to listen to generator signals:

```
            // platforms generator
            this._generator = new Generator.Generator(game.rnd);
            this._generator.onRandomPlatform.add(this.onRandomPlatform, this);
            this._generator.onPatternPlatform.add(this.onPatternPlatform, this);
```

Now, we will write both callbacks. First one is called when random platform is generated:

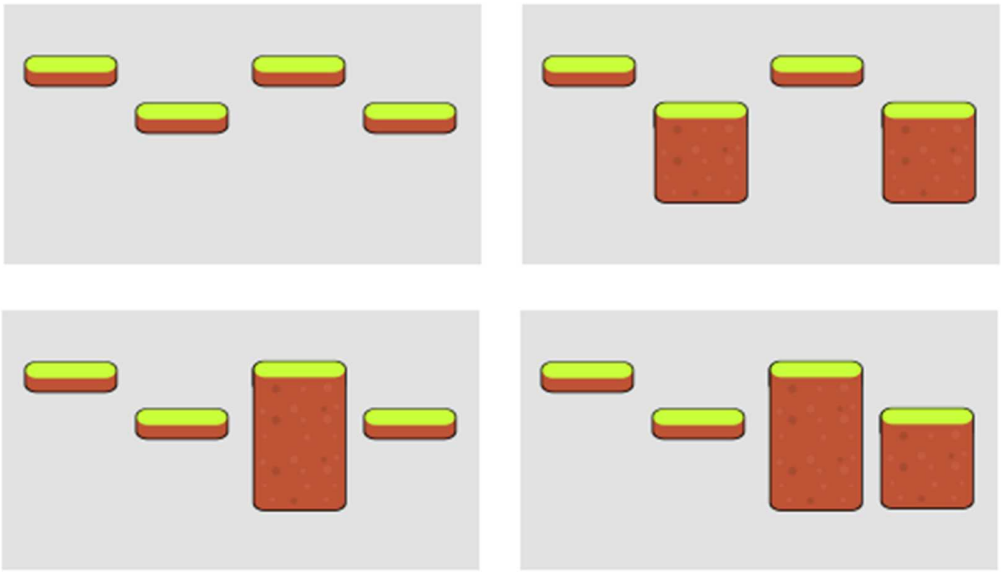
```
        public onRandomPlatform(piece: IGamePiece, previous: IGamePiece): void {
            this.setPlatform(piece);
        }
```

In method arguments we treat passed Pieces as IGamePieces. In Typecript / Javascript we can do this. In languages like Java or C++ we would get error. Method setPlatform() will be defined shortly and it is place where we will decide on if piece is platform or block.

Second callback is called when piece within pattern is generated:

```
        public onPatternPlatform(piece: IGamePiece, previous: IGamePiece,
            position: number, repeat: number, template: IGamePiece): void {

            // first platform in pattern?
            if (position === 0 && repeat === 0) {
                this.setPlatform(piece);
            } else if (repeat === 0) {  // still in base pieces?
                // randomly decide on whether to follow previous piece setting
                if (this.game.rnd.integerInRange(0, 99) < 50) {
                    piece.isPlatform = previous.isPlatform;
                } else {
                    this.setPlatform(piece);
                }
            } else {
                // high probability to follow base pices settings
                if (this.game.rnd.integerInRange(0, 99) < 90) {
                    piece.isPlatform = template.isPlatform;
                } else {
                    this.setPlatform(piece);
                }
            }
        }
```

Here we do few things to increase visual variability of generated patterns. If callback is called with very first piece in pattern, we just let setPlatform() method set platform or block. If we got called on next pieces, that are still part of base pattern, we randomly select whether to follow previous block setting or maybe change it. Probability is 50% to use first block setting for this piece. If we are already repeating base pattern we give only small chance (10%) to change setting and big chance (90%) to follow it as set in base pattern represented here with template piece. On picture below you can see few possible results we can get in case when first piece was set to platform. First picture shows most probable arrangement, while others have less chance to appear. Analogously, if first piece is set to block, final arrangement will tend to be made more from blocks than from platforms.



Now we will add missing setPlatform() method:

```
        private setPlatform(piece: IGamePiece): void {
            // draw as block or platform?
            let platformProb = 100 - (piece.position.y - Generator.Parameters.UBOUND) * 20;
            piece.isPlatform = this.game.rnd.integerInRange(0, 99) < platformProb;
        }
```

It is surprisingly short. Our strategy is, that it would be nice if platforms on top of the screen were represented with platforms and platforms in bottom with blocks. So, we calculate probability of being floating platform based on piece y position. While probability of platform for bottom most row is less than zero, you can still encounter it during gameplay sometimes. It is because it can follow setting within pattern. But it is ok as it spices game presentation.

8.6 Placing tiles

Now, we will move to second set of changes into MainLayer. So far we were placing tiles, represented with placeholder graphics “Block”, in addBlock() method. We will completely rewrite this method, changing its name and arguments, but first add new enum eTileType on top of MainLayer.ts file (code in bold):

```
    const enum eGenerateState { PROCESS_PIECE, GENERATE_PIECE }

    const enum eTileType { LEFT, MIDDLE, RIGHT }

    interface IGamePiece extends Generator.Piece {
        isPlatform: boolean;
    }
```

We will shortly see how to use it.

Change following line in constructor – texture key is changed from “Block” to “Sprites” atlas:

```
                // add empty sprite with body
                let sprite = new Phaser.Sprite(game, 0, 0, "Sprites");
```

In generate() method, in part that is processing generated platforms into tiles do changes in bold:

```
                            this._lastTile.copyFrom(piece.position);
                            let length = piece.length;
                            let tileType = eTileType.LEFT;

                            // process piece
                            while (length > 0) {
                                this.addTiles(this._lastTile.x, this._lastTile.y, tileType,
(<IGamePiece>piece).isPlatform);

                                if ((--length) > 0) {
                                    ++this._lastTile.x;
                                }

                                tileType = (length === 1) ? eTileType.RIGHT : eTileType.MIDDLE;
                            }
```

Here we are using new eTleType enum. We need to track, which part of block or platform we are currently processing. We begin with LEFT tile and then we continue with MIDDLE tiles as long as we are not on last tile (RIGHT).

Previously we called addBlock() method that took only x and y position of tile (you can delete it). We replace it here with call to addTiles(), that takes not only position, but also tile type and whether piece is platform or block.

New addTiles() method is long, but not difficult to catch. It simply puts tile or column of tiles, which depends on platform value, into graphics layer. It sets correct sprite from atlas and adjusts its anchor and physical body as defined in BlockDefs.

```
        private addTiles(x: number, y: number, type: eTileType, platform: boolean): void {

            let defs: ITileDef[][];

            // find right defs
            if (platform) {
                defs = BlockDefs.PLATFORM;
            } else if (y === Generator.Parameters.LBOUND) {
                defs = BlockDefs.LOW_BLOCK;
            } else {
                defs = BlockDefs.BLOCK;
            }
```

Pick correct 2D array of ITileDefs. If we need to draw block and y position of top tile is equal to bottom most row, then we can draw only low block.

```
            // number of vertical tiles
            let rowsCount = platform ? 1 : Generator.Parameters.LBOUND - y + 1;
```

Get number of tiles to put vertically. Platform is only one tile high. Block is higher, except for low block, which is also one tile high. Start processing of all tiles in loop:

```
            for (let r = y; r < y + rowsCount; r++) {

                // find correct block definition
                let blockDef: ITileDef;
                if (defs !== BlockDefs.BLOCK) {
                    blockDef = defs[0][type];
                } else {
                    if (r === y) {
```

```
                    blockDef = defs[0][type];
                } else if (r < y + rowsCount - 1) {
                    blockDef = defs[1][type];
                } else {
                    blockDef = defs[2][type];
                }
            }
```

Inside chosen defs, which is 2D array of ITileDefs, we have to find correct tile definition with help of tile type and row being currently processed.

```
            // sprite  get from pool
            let sprite = this._wallsPool.createItem();
            sprite.position.set(x * 64, r * 64);

            sprite.exists = true;
            sprite.visible = true;
```

Pick free tile sprite from pool and set its position and make sure it exists and is visible.

```
            // adjust sprite to match block definition
            sprite.frameName = blockDef.name;
            sprite.anchor.set(blockDef.anchorX, blockDef.anchorY);
            let body = <Phaser.Physics.Arcade.Body>sprite.body;
            body.setSize(blockDef.bodyWidth, blockDef.bodyHeight, blockDef.bodyOffsetX,
blockDef.bodyOffsetY);
```

Because we spent some time making definitions for tiles, we can now benefit from simply copying their properties into tile sprite. In few lines it does all we need. Anchor is set to correct values as well as body is resized and offset as we need.

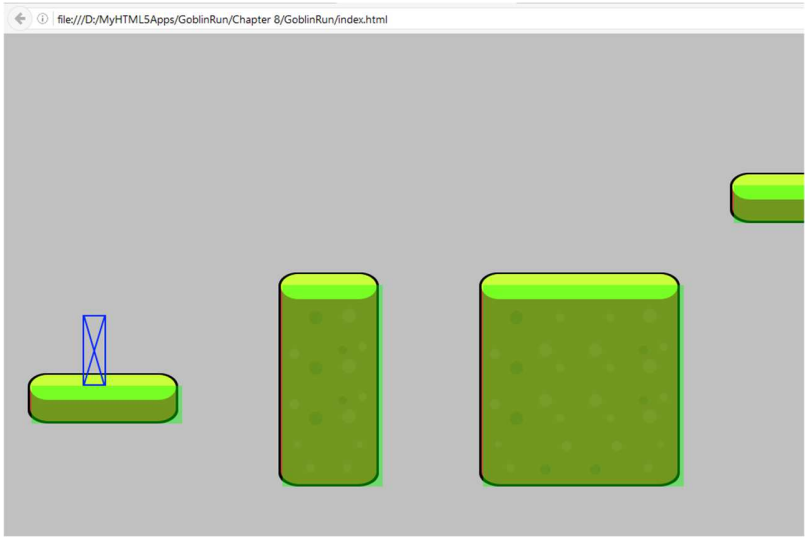
```
            // add into walls group
            if (sprite.parent === null) {
                this._walls.add(sprite);
            }
        }
    }
```

Make sure sprite is in walls group and if not, add it.

You can delete old addBlock() method.

8.7 Running game

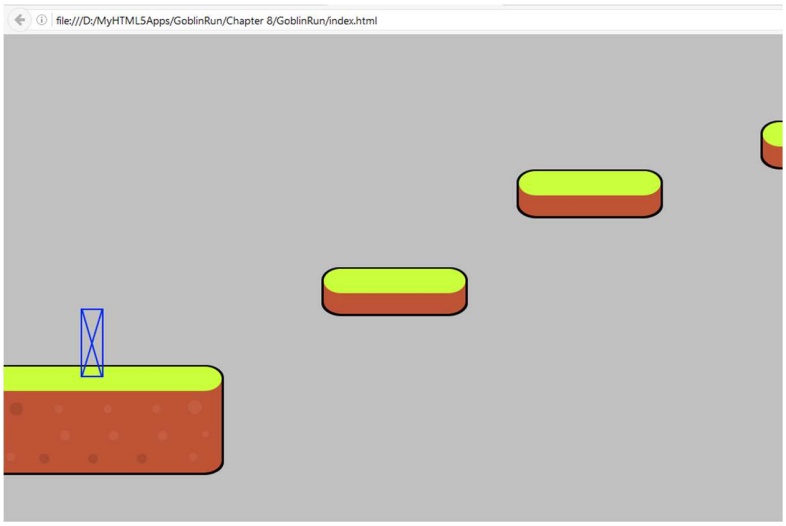
As in previous chapters, we can now compile and run game. This is what you should see on screen:



We still have debug draw for physical bodies of tiles on. We can switch it off. Go into Play.ts file and comment out line in render() method:

```
        public render() {
            //this._mainLayer.render();
        }
```

Run the game again.



Summary

In this chapter we finally replaced some ugly rectangles with final graphics. Our level is now nicely composed from platforms floating in the air and solid blocks growing from bottom. As my feeling is that blue rectangle for player does not fit here anymore, we will replace it in next chapter with animated goblin. And because we are making full game, not only some tutorial example, its animation will be very smooth and skeletal!

9. Animating player

In previous chapter we implemented final graphics for tiles into game. In this chapter we will do the same with player. Phaser has animation system in it, but it is based on frames. We will use it later in book for spikes. For player we will use skeletal animation as it gives really nice results our game deserves.

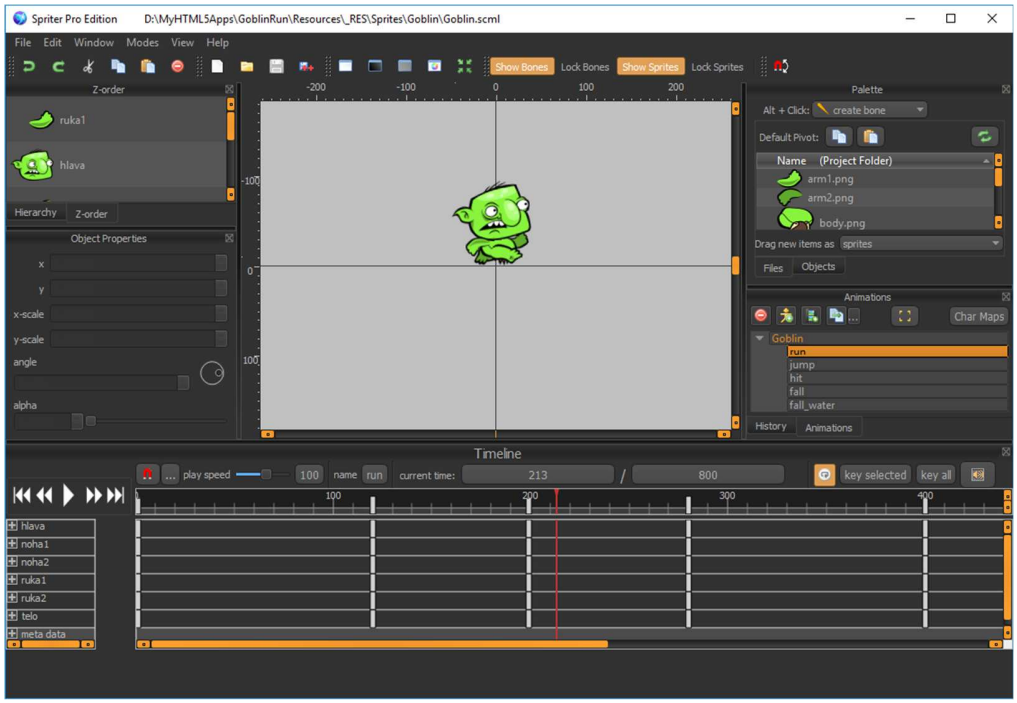
9.1 Skeletal animation

Skeletal animations have several benefits. Individual parts, character is made of, are usually small and it has positive impact on texture size. Animation is smooth, because character's parts are interpolated between key positions.

On the other side, it takes more processing time when running, and composing animations from parts requires new capability artist must have.

These are parts our hero is made of and complete hero on right:

For animating him was used Spriter, tool by BrashMonkey (https://brashmonkey.com/). Spriter comes in two versions – free (https://brashmonkey.com/download_spriter/) and Pro (https://brashmonkey.com/buy-spriter-pro/). Differences between versions are listed here: https://brashmonkey.com/spriter-features/.



Free version is enough for our simple animations and if you do not want to mess with Spriter, it is again already prepared in Resources folder (Resources/_RES/Sprites/Goblin/Goblin.scml) in code files accompanying this book.

Parts of goblin body are part of sprites atlas we are using. Spriter animations are saved in either .scml (which is .xml) or .scon (which is .json) file format. It saves only sprite names and some properties for them, but not any graphics data.

For our game we will need these animations:

- run – goblin runs,
- jump – starts when player initiates jump. Ends with some pose like flying in the air,
- fall – starts when either player falls off platform or player jumped and now starts to fall down again. This animation starts with the same pose jump animation ended,
- hit – played when goblin hits platform or block with his pretty green face,
- fall_water – played when goblin reached maximum y position on screen. Which means in our game he is in murky water.

9.2 Spriter Player for Phaser

Phaser does not play Spriter animations out of the box. We need some piece of code that integrates Spriter with Phaser. I wrote it in past for my games and it is freely available on GitHub as Spriter Player for Phaser (https://github.com/SBCGames/Spriter-Player-for-Phaser). It supports Spriter Pro features like character maps, tags, collision boxes, etc. and can load both .scml and .scon file formats.

To add it into our game, download GitHub project on disk. In folder Build, you will find three files:

- spriter.d.ts – Typescript definitions for Spriter Player,
- spriter.js – compiled and ready to use library,
- spriter.min.js – minified library.

Take spriter.d.ts and put it into lib folder of project. Then take spriter.min.js and copy it into js folder. Then open index.html and add new script (bold line):

```
    <script src="js/phaser.min.js"></script>
    <script src="js/spriter.min.js"></script>
    <script src="js/goblinrun.js"></script>
```

This is all we have to do to have access to Spriter features.

9.3 Loading animation

Goblin animation is ready to be used. Just copy file Gobllin.scml from Resources/_RES/Sprites/Goblin/ into assets folder of game and rename it to Goblin.xml.

Loading is in fact made in two steps. First we have to use standard Phaser assets loading process to load .xml file with Spriter animation and then we will use Spriter Player to turn it into character. For first step, open Preload.ts and add new line in bold (you can also comment out our previous placeholders for player and tiles as well as delete them from assets):

```
        public preload() {
            //this.load.image("Block", "assets/Block.png");
            //this.load.image("Player", "assets/Player.png");

            // atlas
            this.load.atlas("Sprites", "assets/Sprites.png", "assets/Sprites.json");

            // spriter anim
            this.load.xml("GoblinAnim", "assets/Goblin.xml");
        }
```

9.4 Adjusting player

We can proceed to second step, but first little theory. Character created with Spriter Player is Spriter.SpriterGroup object, which *is* Phaser.Group. It extends it and adds some new methods we need to control its animation. It means, you can treat it as any other Phaser.Group object – make it child of some hierarchy, scale it, rotate it, blend it, tint it, …

While all this is great, we have small problem. With Arcade physics in Phaser you cannot add physical body to group. To solve it, we will still need our Player as extension of Phaser.Sprite and we will add Spriter.SpriteGroup as its child. All physics setting will be done at Player (Phaser.Sprite) level and Spriter animation will only work as visual decoration of it. So, we need to make Player sprite invisible. For this, there is sprite with name “Empty” in sprite atlas. This sprite is fully transparent 16x16 block. We will use it as frame for Player and adjust body properties to create body with settings in Parameters.ts (PLAYER_BODY_WIDTH, PLAYER_BODY_HEIGHT).

Open Player.ts file and on top add private variable referencing Spriter character:

```
namespace GoblinRun {
```

```
    export class Player extends Phaser.Sprite {

        private _spriterGroup: Spriter.SpriterGroup;
```

Then change line calling super constructor to use Sprites atlas and Empty frame:

```
        // ---------------------------------------------------------------------------
        public constructor(game: Phaser.Game) {
            super(game, 0, 0, "Sprites", "Empty");
```

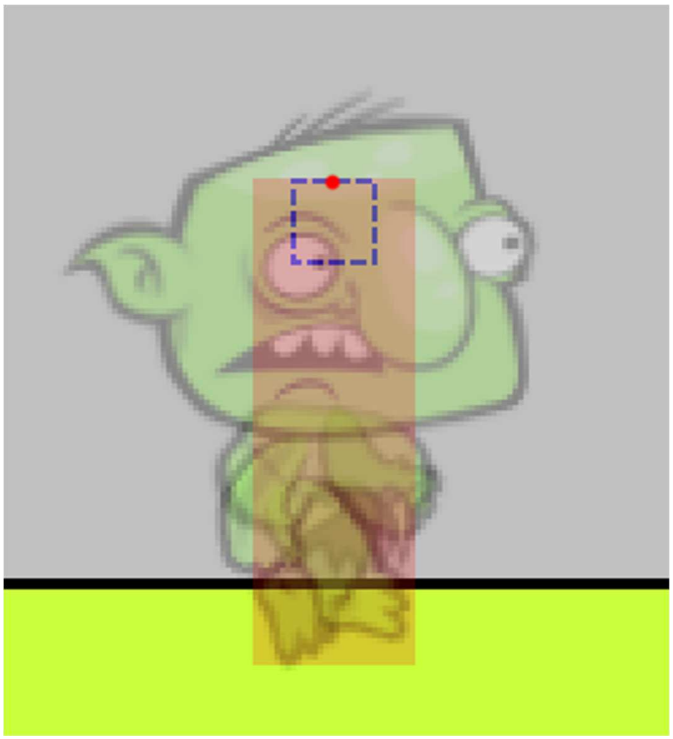
Following few lines remain unchanged from previous implementation, so just skip it:

```
            // center player sprite horizontally
            this.anchor.x = 0.5;

            // enable physics for player
            game.physics.arcade.enable(this, false);

            // allow gravity
            let body = <Phaser.Physics.Arcade.Body>this.body;
            body.allowGravity = true;
```

Rest of the listing is all new code, so I will not bold it. First we set size and position of physical body. By default, size of Arcade physics body is the same as size of sprite, but we want it to match settings in Parameters.ts. We also want it center horizontally:

```
            // set body size according to values in Generator.Parameters
            let bodyW = Generator.Parameters.PLAYER_BODY_WIDTH;
            let bodyH = Generator.Parameters.PLAYER_BODY_HEIGHT;
            let bodyOffsetX = -bodyW / 2 + this.width * this.anchor.x;
            let bodyOffsetY = 0;
            // set body size and offset
            body.setSize(bodyW, bodyH, bodyOffsetX, bodyOffsetY);
```

What we did is illustrated on this picture:



Red dot is sprite position. Dashed line surrounds Empty frame assigned to Player sprite. Red transparent rectangle is physics body for him. Offsets of physics bodies are measured from top left of sprite. For x offset we had to do small calculation to offset it from red dot instead.

We can proceed to second step of loading Spriter character. We already have .xml data loaded. Now, we will turn these data into game object – Spriter.SpriterGroup.

```
            // create Spriter loader - class that can change Spriter file into internal structure
            let spriterLoader = new Spriter.Loader();

            // create Spriter file object - it wraps XML/JSON loaded with Phaser Loader
            let spriterFile = new Spriter.SpriterXml(game.cache.getXML("GoblinAnim"));

            // proces Spriter file (XML/JSON) with Spriter loader - outputs Spriter animation which
you can instantiate multiple times with SpriterGroup
            let spriterData = spriterLoader.load(spriterFile);

            // create actual renderable object - it is extension of Phaser.Group
            this._spriterGroup = new Spriter.SpriterGroup(this.game, spriterData, "Sprites",
"Goblin", "fall", 120);
```

We first create Spriter.Loader object. This object can be used many times to process .xml/.json Spriter data. For some larger game with lot of animation, you can put it into some global variables accessible across game. Next we create Spriter.SpriterXml which extends Spriter.SpriterFile and which wraps .xml data. Similarly you can wrap .json data with Spriter.SpriterJSON object that also extends Spriter.SpriterFile. Spriter.SpriterFile is generic enough to be processed with Spriter.Loader regardless it contains .xml or .json data.

Spriter.Loader then reads file data and returns Spriter.Spriter object into spriterData variable. This object already contains whole structure of Spriter animation. It is again reusable and you can use it multiple times

to create multiple independent instances of the same character on screen. We have only one and on next line we are finally creating group with Goblin. Parameters passed are Phaser.Game reference, character data, atlas in which parts of character are stored, entity, animation and animation speed.

One Spriter file can have several entities. We have only one and we named it Goblin. Under this entity we have several animations as described earlier. We set Goblin to start with fall animation. Playback speed is set to 120% of original animation speed. It is to prevent "sliding" on platform. If your character is moving in some certain speed and animation speed does not match it, it looks like his feet are sliding on surface. Simple solution is to adjust animation playback speed.

```
        // set position size
        this._spriterGroup.position.set(-5, 60);

        // adds SpriterGroup to Phaser.World to appear on screen
        this.addChild(this._spriterGroup);
    }
```

Next we adjust Goblin position to match collision box nicely as on image above and add it as child of Player.

Following methods will be called when we want to react on some event in game and set appropriate animation.

```
    public animateJump(): void {
        this._spriterGroup.playAnimationByName("jump");
    }

    // ------------------------------------------------------------------------
    public animateDeath(): void {
        let body = <Phaser.Physics.Arcade.Body>this.body;
        body.enable = false;

        this._spriterGroup.playAnimationByName("fall_water");
    }

    // ------------------------------------------------------------------------
    public animateHit(): void {
        this._spriterGroup.playAnimationByName("hit");
    }
```

Only explanation is needed for animateDeath() method. During game loop we will check player's y position and if it is over some limit, we will set fall_water animation. To immediately stop further physics processing of player we disable his body.

```
    public updateAnim(standing: boolean, velY: number, gameOver: boolean): void {

        if (!gameOver) {
            if (standing) {
                if (this._spriterGroup.currentAnimationName !== "run") {
                    this._spriterGroup.playAnimationByName("run");
                }
            } else if (velY > 0) {
                if (this._spriterGroup.currentAnimationName !== "fall") {
                    this._spriterGroup.playAnimationByName("fall");
                }
            }
        }
```

```
            this._spriterGroup.updateAnimation();
        }
    }
}
```

To update Goblin's animation we need to call updateAnimation() method on Spriter.SpriterGroup every frame. We also do several checks and set animation accordingly. If player is standing on platform, then animation must be run. If player is in the air and is moving down (either falling from platform or jumping), then his animation should be fall (unless gameOver is set true – in such case animation already may be set to hit as player ran into wall).

9.5 Additional changes

Player class has been rewritten. We have to make a few small changes also into Play state. Open Play.ts and locate update() method. Whole new update() method is here as there are changes in curly braces placement:

```
        public update() {
            if (!this._gameOver) {
                this.updatePhysics();

                // move camera
                this.camera.x = this._player.x - 192;
```

In first part we only moved camera little bit more behind player because Goblin is wider than previous rectangle placeholder. Next, make following changes in bold.

```
                // generate level
                this._mainLayer.generate(this.camera.x / Generator.Parameters.CELL_SIZE);
            }


            // check if player is still on screen
            if (this._player.y > this.game.height - 104) {
                this._player.y = this.game.height - 104;
                this._gameOver = true;

                this._player.animateDeath();
                console.log("GAME OVER - fall");
            }
```

Close first if block under call to _mainLayer.generate(). Next, new conditional check for player position is not part of previous if block anymore.

If player is falling, either because he missed platform or because he hit some block with nose, we check his vertical position and if it is higher than some distance (104 pixels) from bottom of the screen, we trigger death animation.

```
            // update player animations
            let body = <Phaser.Physics.Arcade.Body>this._player.body;
            this._player.updateAnim(body.velocity.y >= 0 && body.touching.down, body.velocity.y,
this._gameOver);
        }
```

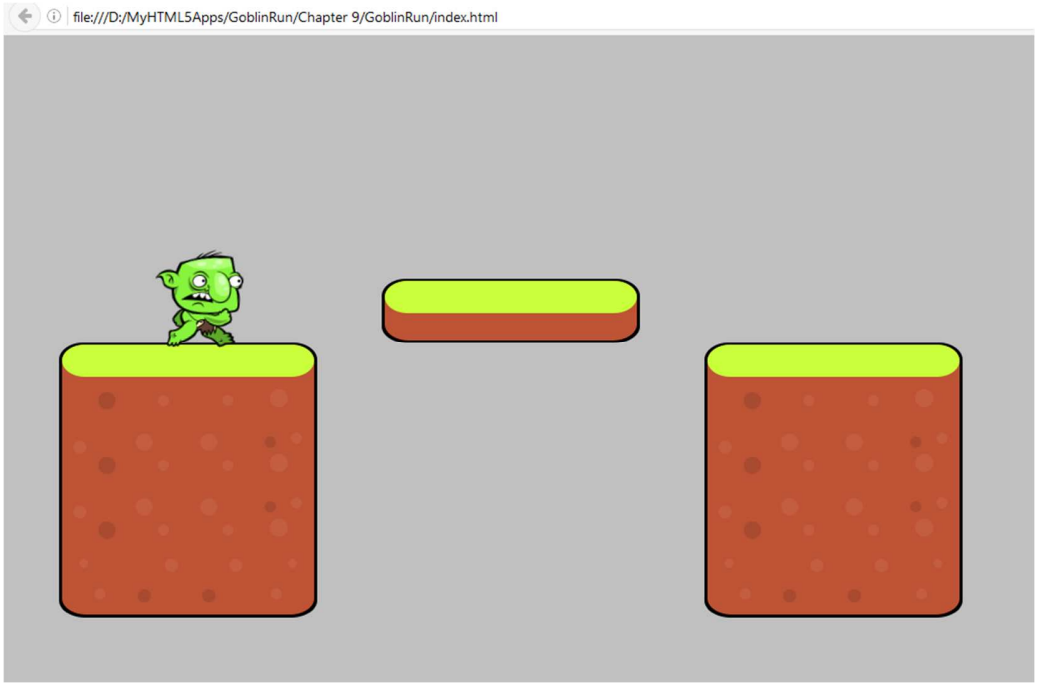
Regardless gameOver variable is set or not, we keep updating of player animation.

Now, locate updatePhysics() method and add calls to change animation of player when he hits wall and jumps. Changes are in bold:

```
        private updatePhysics(): void {

                  :
                  :

            // move
            if (wallCollision && body.touching.right) {
                body.velocity.set(0, 0);
                this._gameOver = true;

                this._player.animateHit();
                console.log("GAME OVER - hit");
                return;
            }

                  :
                  :

            // start jump
            if (this._justDown && body.touching.down && this.game.time.now > this._jumpTimer) {
                body.velocity.y = jumpTable.maxJumpVelocity;
                this._jumpTimer = this.game.time.now + 150;
                this._justDown = false;

                this._player.animateJump();
            }

                  :
                  :

        }
```

9.6 Running game

Compile and run game and enjoy smooth animation of main character:



Summary

In this chapter we used Spriter Player for Phaser to implement skeletal animation for our player. It creates smooth animation with much smaller texture space occupied compared to use of frame animation.

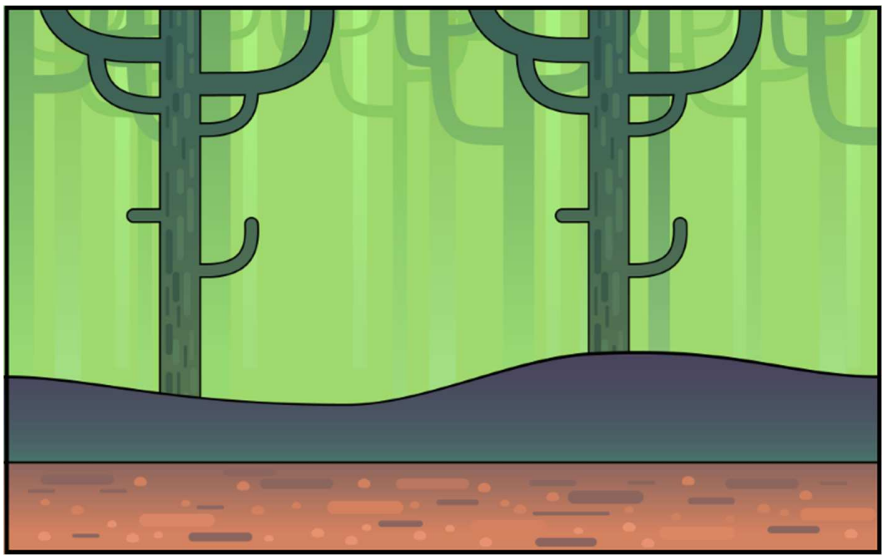
We have nice platforms and nicely animated character, but our background is still little dull. We will change it in next chapter.

10. Background layers

This chapter will bring background into our game. In the end of it you will see how big change it is, despite of the fact that code is simple and straightforward.

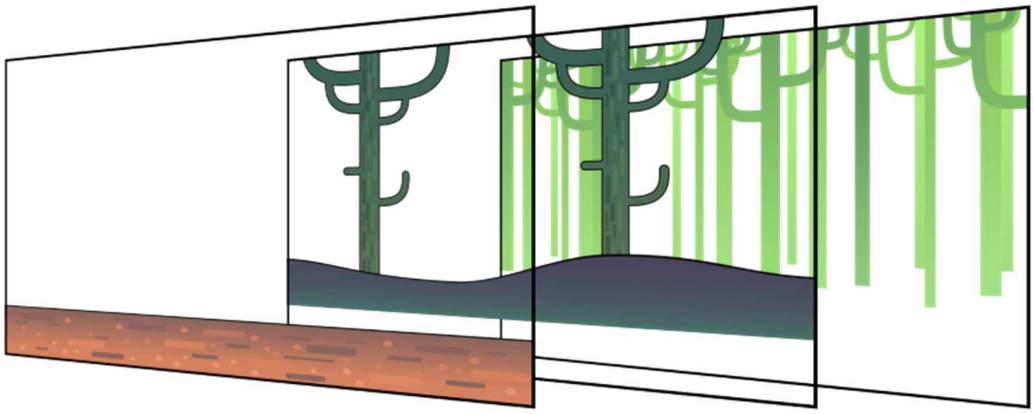
10.1 Background layout

First, we have to think about background layout. This is how we want it to look:



When game is scrolling we want individual layers to scroll in different speeds and thus create parallax scrolling effect.

Bottom most layer is solid green color. Over it are trees in various shades of green. This layer will scroll in slowest speed – it will be 25% of main layer speed. Next layer is hill with trees. This layer will scroll faster – 50% of main layer. This layer also will be most complicated from code point of view, because it will not be single image as other layers. It will be combination of single image for hills and group for trees. These trees will be generated in random distances from each other to break repetitiveness. Top most background layer is mud in bottom. It will scroll in the same speed as platforms, but it will also move from left to right a little to create illusion of waving.



While it looks like lot of work. It will be surprisingly easy to implement it all.

10.2 Background assets

First go into Resources folder in files accompanying this book and copy from Resources/_RES/Sprites/ these files into assets folder of game:

- Hill.png
- Mud.png
- TreesBg.png

Image of tree growing from hill is already part of Sprites atlas.

Open Preload.ts and add new lines into it:

```
        public preload() {

                    :
                    :

            // background layer sprites
            this.load.image("Mud", "assets/Mud.png");
            this.load.image("Hill", "assets/Hill.png");
            this.load.image("TreesBg", "assets/TreesBg.png");
        }
```

10.3 Background class

Now, we will create Background class. It will be responsible for moving layers as well as for generating trees on hill.

In Game folder create new Background.ts file and start putting code into it step by step. First, let's define some constants:

```
namespace GoblinRun {

    export class Background extends Phaser.Group {

        private static TREE_DIST_MIN = 300;
        private static TREE_DIST_MAX = 800;
```

This will be used when generating trees on hill. It gives minimum and maximum distance of new tree from previous one.

Next, we will add some variables:

```
        private _treesBg: Phaser.TileSprite;
        private _trees: Phaser.Group;
        private _hill: Phaser.TileSprite;
        private _mud: Phaser.TileSprite;

        private _nextTreeX: number = 0;
        private _treeWidth: number;
```

Most of them simply keep reference to game objects we will create. _nextTree is x position of next tree on the hill that we will generate. _treeWidth just caches width of tree sprite.

In constructor we will create all objects representing layers:

```
        public constructor(game: Phaser.Game, parent: PIXI.DisplayObjectContainer) {
            super(game, parent);

            // heights
            let treesHeight = game.cache.getImage("TreesBg").height;
            let hillHeight = game.cache.getImage("Hill").height;
            let mudHeight = game.cache.getImage("Mud").height;
```

Here we got heights of individual images. We will use it in y position calculations later.

```
            // trees bg
            this._treesBg = new Phaser.TileSprite(game, 0, 0, game.width, treesHeight, "TreesBg");
            this._treesBg.fixedToCamera = true;
            this.add(this._treesBg);
```

We are creating our first layer – green trees in background. For it we use Phaser.TileSprite. Tilesprites are sprites with repeating texture. As we can scroll texture inside Phaser.TileSprite it is ideal for us. We want it as wide as is our game width, so image may repeat multiple times horizontally, but with only one repeating vertically.

Be careful when you use tilesprites. It breaks WebGL render batch, so extra draw call is issued. For this reason, it has no sense to put images we use solely for tilesprites into atlas. It does not save any draw calls.

Tilesprite is part of the world as any other game object. As player runs to right we are moving camera over world to keep him in view. It means, that our tilesprite would move to left and get out of view soon. But we want to keep it on fixed position, because we will not move tilesprite itself, we will only change its texture offset. Phaser allows you to set fixedToCamera property to true to achieve this. Most often you will use it to fix game GUI on screen when your world is scrolling, but you are not limited only to it.

```
            // trees group / pool
            this._trees = new Phaser.Group(game, this);
            this._trees.createMultiple(4, "Sprites", "Tree", false);
            // width of tree sprite
            this._treeWidth = game.cache.getFrameByName("Sprites", "Tree").width;
```

As next, we are creating Phaser.Group for trees on hill. We call createMultiple() method to create pool of four, disabled by default. Then we take Tree frame from cache and store its width. We will use it later to check whether tree sprite is completely out of screen.

```
            // hill
            this._hill = new Phaser.TileSprite(game, 0, game.height - mudHeight - hillHeight,
game.width, hillHeight, "Hill");
            this._hill.fixedToCamera = true;
            this.add(this._hill);

            // mud
            this._mud = new Phaser.TileSprite(game, 0, game.height - mudHeight, game.width,
mudHeight, "Mud");
            this._mud.fixedToCamera = true;
            this.add(this._mud);
        }
```

Hill and Mud are again Phaser.TileSprites and are created in the same way as background with trees. We only position them into different height.

Because all objects needed for background are constructed, we can write updateLayers() method, that will update it:

```
        public updateLayers(x: number): void {
            // move all three tilesprites
            this._mud.tilePosition.x = -x + Math.sin(Phaser.Math.degToRad((this.game.time.time / 30)
% 360)) * 20;
            this._hill.tilePosition.x = -x * 0.5;
            this._treesBg.tilePosition.x = -x * 0.25;

            // move trees layer and remove/add trees
            this.manageTrees(x * 0.5);
        }
```

We pass camera x positon into it. Inside we are setting tilePosition.x for all tilesprites. It offsets texture used, which makes illusion of scrolling inside it. As we want background to scroll against player, we have to use negative x. Position is multiplied with speed we want for each layer.

For mud we are using little bit wild calculation. We want it not only scroll, but also wave a little from left to right. To achieve it we compose scrolling with waving into resulting position. For waving, we take current time in milliseconds. As it changes too quickly, we divide it by 30 to slow it down. Then we use modulo 360 to turn it into number in interval 0-359, which can be treated as angle. We convert this angle into radians with call to Phaser helper method Phaser.Math.degToRad(). We pass radians into Math.sin() method to get sin of angle. Sin is in range -1 to 1. Finally, we multiply this with 20, which is maximum distance to left and right from current positon. This will make our mud smoothly move from left to right, while it still be scrolling as we run right.

For updating trees on hill we create separate manageTrees() method. Problem is, that _trees is group in world, but we want to move it as slow as hill tilesprite. _trees group is not fixed to camera, so it moves as fast as main layer with hero and platforms. We have to compensate it somehow. Simple consideration will tell us that if main layer scrolled 1000 pixels and we want _trees group to scroll only half of it, we have to move it by 500 pixels in direction of move. It is the same when you are going by car. Houses fixed on streets

are moving quickly, but cars moving little bit slower than you, seems to pass slowly, because they are moving in the same direction as you.

Content of manageTrees() looks like this:

```
        private manageTrees(x: number): void {
            // move trees layer
            this._trees.x = x;
```

Here we are moving whole layer with trees in direction of move.

```
            // remove old
            this._trees.forEachExists(function (tree: Phaser.Sprite) {
                if (tree.x < x - this._treeWidth) {
                    tree.exists = false;
                }
            }, this);
```

We go through all trees in group that exist (are enabled) and if its positon is less than current x position of layer less tree width, we disable it. Disabled tree is ready to be reused when needed.

```
            // add new tree(s)
            while (this._nextTreeX < x + this.game.width) {
                // save new tree position
                let treeX = this._nextTreeX;

                // calcultate position for next tree
                this._nextTreeX += this.game.rnd.integerInRange(Background.TREE_DIST_MIN,
Background.TREE_DIST_MAX);

                // get unused tree sprite
                let tree = <Phaser.Sprite>this._trees.getFirstExists(false);
                // if no free sprites, exit loop
                if (tree === null) {
                    break;
                }

                // position tree and make it exist
                tree.x = treeX;
                tree.exists = true;
            }
        }
    }
}
```

In this part of code, we check whether positon of next tree is already in camera. If yes, we will place tree there. Offset of next tree from current one is chosen randomly in range we defined in the beginning of class.

We take first unused tree from _trees group and if it is null, we immediately break loop. If tree is available, we set its positon and enable it with setting exists to true.

Loop runs as long as _nextTree is within game width. In the beginning of game, it is set to 0 and it means, it will automatically fill first screen with trees from left to right.

10.4 Using background

We are almost ready. We can now use our new Background class. Open Play.ts and add reference to Background instance on top of it:

```
namespace GoblinRun {

    export class Play extends Phaser.State {

        // background
        private _bg: Background;
```

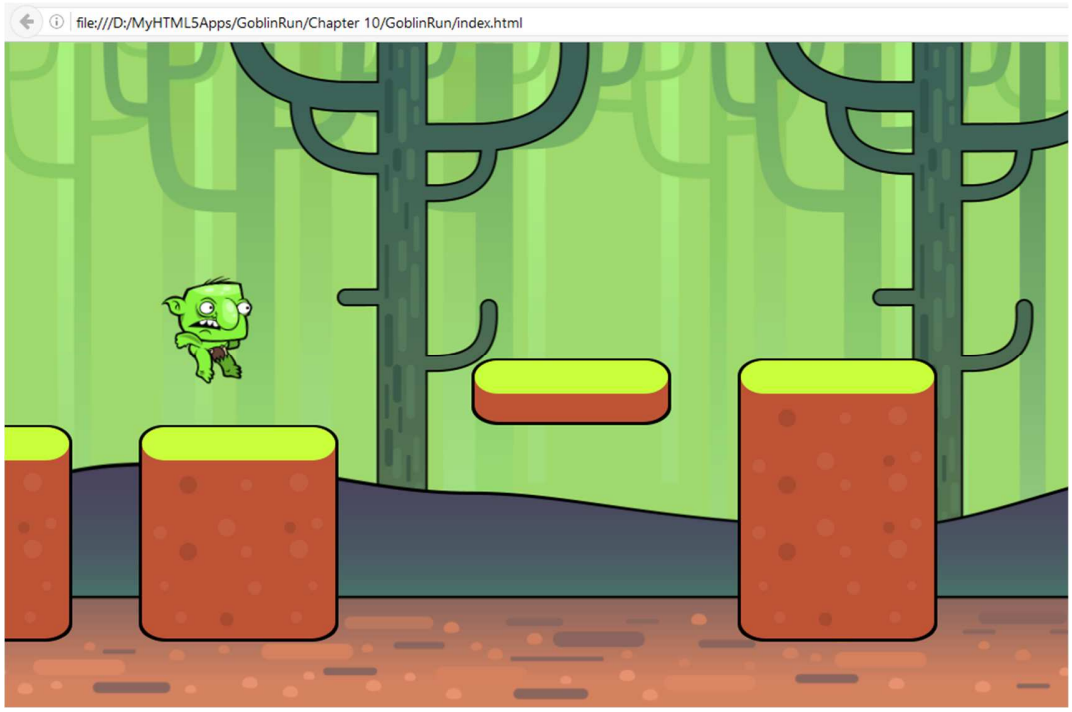
In create method change color of background from grey to pale green and create new instance of Background:

```
        public create() {
            this.stage.backgroundColor = 0xA0DA6F;

                :
                :

            // this.game.add.sprite(0, 0, Generator.JumpTables.debugBitmapData);


            // background layers
            this._bg = new Background(this.game, this.world);

                :
                :
```

Go to update method and add following lines in the very end of it:

```
        public update() {

                :
                :

            // move background
            this._bg.updateLayers(this.camera.x);
        }
```

Compile and run game. You should see this result:



Summary

With added background our game looks like real game. In next chapter we will return to generator and make it more challenging with adding deadly spikes.

11. Spikes

Our game starts to look good, but after some time it may become dull. It would be fine if we added some additional challenges into it. This is what we will do in this and next chapter. In this one we will add spikes our hero has to jump over.

Following frames will form spikes animation:

These frames are already in sprite atlas.

Adding spikes into game will need to make two relatively independent steps. First, we have to add it into generator, so it knows what spikes are and can generate it. Secondly, we have to add it into game and represent it somehow on screen, test collisions, etc.

11.1 Spikes idea

Before we start implementation, let's discuss idea behind spikes. From time to time, we want on some platforms spikes. When spikes are placed on platform, we do not want it to look too random. In fact, we do not want it to look randomly at all. For this, we will choose from sets of prebuilt patterns. Possible point of view can be that for platforms we want some randomness, because they are pieces of dirt and grass and were placed by nature. On the other hand, spikes were placed there by some thinking entity like human and in most human work you can find some symmetry or pattern.

Other point is, that platforms are getting shorter and shorter as game goes on. It would be unfair to player to place spikes on very short platforms. Moreover, there would not be enough space to form some pattern. To solve this, platforms with spikes will be at least 5 tiles long and maximum length will be 9 tiles. This brings another fresh breeze into game, because we will place long platform between short ones, which will increase visual variability.

As spikes are something that make game more difficult, we want to control how often they are spawn. We will increase spikes probability with growing time and distance passed.

Now to implementation.

11.2 New Piece property

Open Piece.ts in Generator folder and add new property - public variable into it (bold lines):

```
namespace Generator {
```

```
    export class Piece {

                :
                :

        // spikes pattern
        public spikesPattern: number;
    }
}
```

Whole spike pattern will be stored in single number!

11.3 New parameters

In file Parameters.ts we will add two sets of new parameters. First set will control difficulty and second one will define spike patterns. Open file and first add this in the end:

```
        // spikes
        public static SPIKES_PROB_MIN = 0;
        public static SPIKES_PROB_MAX = 25;
        public static SPIKES_PROB_START_TILE = 30;
        public static SPIKES_PROB_END_TILE = 80;
```

It says, what is minimum and maximum probability, that platform has spikes on itself. As for all previous difficulty parameters, we calculate current probability based on distance passed in tiles. From minimum to the maximum we will get between start tile and end tile. Current settings are to start from tile 30 and reach maximum on tile 80. Fifty tiles are short distance in our game but we primarily want to test it. We can tweak difficulty setting later.

Add second set of new parameters:

```
        // --- SPIKE PATTERNS ---
        public static SPIKE_PATTERNS: number[][] = [
            [],                                     // 0
            [],                                     // 1
            [],                                     // 2
            [],                                     // 3
            [],                                     // 4
            [0b00100],                              // 5
            [0b001100],                             // 6
            [0b0011100, 0b0010100],                 // 7
            [0b00011000, 0b00100100],               // 8
            [0b000111000, 0b001101100, 0b001000100] // 9
        ];
```

SPIKE_PATTERNS is 2d array of numbers. For each possible length of platform, we define array of possible spike patterns. Spike patterns are defined with single binary number. Ones are spikes and zeroes are holes between them. We have to be careful to not create patterns, that are impossible for player to pass. With our current jump and speed setting, we can get over three consecutive tiles. It is quite challenging but possible. For platforms with length less than 5 tiles we define no patterns. If platform is chosen to have spikes, we set its minimum length to 5. You can change this for your game, but be sure to test whether game is still playable.

11.4 Increasing difficulty

To make use of our new difficulty parameters, we will open Difficulty.ts and do lot of small changes. Start with adding new private variable to hold current spikes probability and initialize it in constructor:

```
        // spikes probability
        private _spikesProbability: number;

        // ------------------------------------------------------------------------
        public constructor(rnd: Phaser.RandomDataGenerator) {

                        :
                        :

            // initial spikes probability
            this._spikesProbability = Parameters.SPIKES_PROB_MIN;
        }
```

Then add getter method for it as we did with all previous difficulty parameters:

```
        public get spikesProbability(): number {
            return this._spikesProbability;
        }
```

Parameter has to be updated, so add necessary lines into update() method (changes in bold):

```
        public update(tileX: number): void {

                    :
                    :

            // spikes probability
            this._spikesProbability = Math.round(this.mapLinear(tileX,
                Parameters.SPIKES_PROB_START_TILE, Parameters.SPIKES_PROB_END_TILE,
                Parameters.SPIKES_PROB_MIN, Parameters.SPIKES_PROB_MAX));
        }
```

You can add new difficulty parameter also to debug output to simply print current difficulty if needed:

```
        public toString(): string {
            return "platformLengthDecrease: " + this._platformLengthDecrease +
                ", jumpLengthDecrease: " + this._jumpLengthDecrease +
                ", spikesProbabilty: " + this._spikesProbability;
        }
```

Changes into Difficulty.ts were simple. We just mimicked code for older parameters.

11.5 Generating spikes

After all that preparations, change into generator is surprisingly simple. Open Generator.ts, locate generate() method and add bold code after “LENGTH” code section:

```
        private generate(lastPosition: Phaser.Point, difficulty: Difficulty,
            length: number, offsetX: number, offsetY: number, bonusJump: boolean): Piece {

                    :
                    :

            // LENGTH

                    :
                    :
```

```
            // SPIKES
            if (this._lastGeneratedPiece !== null && this._lastGeneratedPiece.spikesPattern === 0 &&
                !bonusJump &&
                (this._rnd.integerInRange(0, 99) < difficulty.spikesProbability)) {

                // adjust length - make piece longer
                piece.length = this._rnd.integerInRange(5, 9);

                // choose spikes pattern randomly
                let patternDefs = Parameters.SPIKE_PATTERNS[piece.length];
                piece.spikesPattern = patternDefs[this._rnd.integerInRange(0, patternDefs.length -
1)];

            } else {
                piece.spikesPattern = 0;
            }


            // console.log(difficulty.toString());

                        :
                        :
        }
```

There are few conditions, that must be fulfilled if spikes shall be generated. We do not generate it on consecutive platforms. If previous platform had spikes on it then current one cannot. bonusJump parameter passed into method must be false. We will get to what is bonus jump and how to implement it in next chapter. Last condition is test if random number in range from 0 to 99 is less than current probability taken from current difficulty.

If all met, then randomly prolong platform between 5 and 9 tiles.

Finally, take array with all possible patterns for new platform length and choose randomly one of them.

If conditions are not met, then simply clean spikes for current platform setting it to zero.

11.6 Spikes animation

At present generator randomly generates spikes. But it has no visual representation in game yet. As said in the beginning, spikes will be animated. Let's create new class, that will group frame definitions for all our animations. Add file Animations.ts into Game folder. For now, it will be simple as spikes are our first animation made of atlas frames:

```
namespace GoblinRun {

    export class Animations {

        public static SPIKE_ANIM = ["Spikes1", "Spikes2", "Spikes3", "Spikes4", "Spikes3",
"Spikes2"];

    }
}
```

All we did is, that we defined SPIKE_ANIM as array of strings. These strings are names of frames in sprite atlas. We will reference it when creating spikes animation. It will loop infinitely – there is one cycle of spikes going up and down in array.

It is not necessary to do it like this. We can define strings array at place where needed. But this approach helps us to group all animations under one class and at one place. Later, if you want to make change, you exactly know where to look for it.

11.7 Defining spikes block

When adding tiles, we created file BlockDefs.ts with ITileDef interface and BlockDefs class. It helped us to define various platform blocks in simple way. We can use it also for defining spikes.

Open BlockDefs.ts file and in the very end of BlockDefs class add this lines:

```
        // SPIKES
        public static SPIKES: ITileDef = {name: "spikes", anchorX: 0.5, anchorY: 1, bodyOffsetX: 9,
bodyOffsetY: 17, bodyWidth: 45, bodyHeight: 34};
```

There is only one slight change in meaning of name property. For platform blocks it was name of frame in atlas. For spikes it will be name of animation that will be created on sprite.

11.8 Adding spikes to main layer

Now we will add spikes to main layer. It will bring many changes into MainLayer class. In next chapters we plan to add bonus for mid-air jump as well as some gold to gather. So, we will treat all these features (spikes, bonus jump, gold) as “item” and will make some simple system how to put it into game. It will slightly copy what we built for tiles but with some changes.

Open file MainLayer.ts and on top of it add lines in bold:

```
                :
                :

    export const enum eItemType {SPIKE, BONUS_JUMP, GOLD }

    export class Item extends Phaser.Sprite {
        itemType: eItemType;
    }

    export class MainLayer extends Phaser.Group {

        private _generator: Generator.Generator;

        private _wallsPool: Helper.Pool<Phaser.Sprite>;
        private _walls: Phaser.Group;

        private _itemsPool: Helper.Pool<Item>;
        private _items: Phaser.Group;

                :
                :
```

Here we create another enum to distinguish between various types of items. We also create very simple extension of Phaser.Sprite class which we named Item and which has only one additional property itemType of type eItemType.

If you remember Chapter 8, part 8.5, we added isPlatform property to Generator.Piece class only with IGamePiece interface. We considered Generator.Piece objects to be objects in compliance with

IGamePiece interface, but we never created new instances of it. In callbacks we treated passed Generator.Piece objects as IGamePiece objects. If we wanted to work with Generator.Piece objects as with IGamePiece objects, we had to cast it, which was only inside generate() method. If we called "new IGamePiece()", we would get error. Now, we are creating new Item class to extend Phaser.Sprite. As it is class, it has constructor and we can create instances of Items and later work with Item objects. We can here also choose interface approach and add itemType only through it. But then we would have to create pool of Phaser.Sprites instead of Items and every time we wanted to work with Item object, we would have to cast Phaser.Sprite object to it.

Back to code. On last two lines, we are creating pool of Items and separate group for them. This group will be drawn over walls group.

Next, we will create items group and initialize it in constructor. Add bold lines after crating walls group:

```
    public constructor(game: Phaser.Game, parent: PIXI.DisplayObjectContainer) {

                :
                :

        // walls group
        this._walls = new Phaser.Group(game, this);

        // pool of items
        this._itemsPool = new Helper.Pool<Item>(Item, 32, function () {
            // empty item
            let item = new Item(game, 0, 0, "Sprites");

            // add animations
            item.animations.add("spikes", Animations.SPIKE_ANIM, 10, true);

            // enable physics
            game.physics.enable(item, Phaser.Physics.ARCADE);

            // setup physics
            let body = <Phaser.Physics.Arcade.Body>item.body;
            body.allowGravity = false;
            body.immovable = true;
            body.moves = false;

            return item;
        });

        // items group
        this._items = new Phaser.Group(game, this);

                :
                :
    }
```

We are creating pool of 32 Items. Initialization is the same as for wall tiles with one exception. We add spikes animation to newly created sprite. Later, we can add more animations here.

We will slightly adjust generate method:

```
    public generate(leftTile: number): void {

        // remove tiles too far to left
        this.cleanTiles(leftTile);
        // do the same for items
        this.cleanItems(leftTile);
```

```
                :
                :

                            // process piece
                            while (length > 0) {
                                this.addTiles(this._lastTile.x, this._lastTile.y,
                                    tileType, (<IGamePiece>piece).isPlatform,
                                    (piece.spikesPattern & (1 << (length - 1))) > 0);

                                                :
                                                :
                            }

                :
                :
        }
```

First, we want to remove old Items, that are too far on the left. We will define cleanItems() method in a moment. In while loop we are newly passing one more parameter into addTiles() method. This parameter is boolean saying, whether there is spike on current platform column or not. We will need to add this parameter into addTiles() method header too.

To calculate, whether there is spike we mask its pattern with shifted 1 (bit zero). Remember, whole spike pattern is encoded in one binary number.

cleanItems() definition is here:

```
        private cleanItems(leftTile: number): void {
            leftTile *= Generator.Parameters.CELL_SIZE;

            for (let i = this._items.length - 1; i >= 0; i--) {
                let item = <Item>this._items.getChildAt(i);

                if (item.x - leftTile <= -64) {
                    this._items.remove(item);
                    item.parent = null;
                    this._itemsPool.destroyItem(item);
                }
            }
        }
```

We check if Item is behind left screen border and if yes, we remove it and return into pool.

Change addTiles() method header with adding new spike argument and add new code lines in the bottom of it (changes in bold):

```
        private addTiles(x: number, y: number, type: eTileType, platform: boolean, spike: boolean):
void {

                        :
                        :

            for (let r = y; r < y + rowsCount; r++) {

                        :
                        :

                // spikes
                if (spike && r === y) {
                    let spikeSprite = this._itemsPool.createItem();
                    spikeSprite.itemType = eItemType.SPIKE;
```

```
                    spikeSprite.position.set(x * 64 + 32, r * 64 + 8);

                    spikeSprite.exists = true;
                    spikeSprite.visible = true;

                    this.setupItem(spikeSprite, BlockDefs.SPIKES, true, true);

                    if (spikeSprite.parent === null) {
                        this._items.add(spikeSprite);
                    }
                }
            }
        }
```

First we check if this platform tile has spike on it and if it is top tile in case of drawing whole column of tiles for block. Then we set its itemType property to SPIKE. We will need this shortly to distinguish collisions with various objects. For copying variables from ITileDef block definition into newly created item sprite, we write separate setupItem() method. We do this, because we will use it later for other Item types too. Method looks like this:

```
        private setupItem(item: Item, def: ITileDef, animated: boolean, syncAnim: boolean): void {
            // anchor
            item.anchor.set(def.anchorX, def.anchorY);
            // body dimensions and offset
            (<Phaser.Physics.Arcade.Body>item.body).setSize(
                def.bodyWidth, def.bodyHeight, def.bodyOffsetX, def.bodyOffsetY);

            if (animated) {
                item.animations.play(def.name);

                // if request to synchronize animation with other items of the same type
                if (syncAnim) {
                    let prevItem = this.getItemOfType(item.itemType);
                    if (prevItem !== null) {
                        item.animations.currentAnim["_frameIndex"] =
prevItem.animations.currentAnim["_frameIndex"];
                        item.animations.currentAnim["_timeNextFrame"] =
prevItem.animations.currentAnim["_timeNextFrame"];
                    }
                }
            } else {
                // stop any previous animation
                item.animations.stop();
                // set frame
                item.frameName = def.name;
            }
        }
```

It takes Item we want to set up, ITileDef for this item, whether it is animated and whether we want to synchronize animation of this item with animation of other items of the same type on screen. Various objects are added into game in various times. So, their animation, while the same, can be in different part of loop. In case of spikes, it would look ugly if three consecutive spikes started played their animation with different animation offsets.

In first part of method we simply copy values from iTileDef into new item. Then we check if item is animated or not. If not animated, we stop any animation that may be running on it and set its frame. If animated, we play animation of name stored in ITileDef as name property.

If animation synchronization is requested, we look if there is another item of the same type on screen (with getItemOfType() method) and if yes, we copy some of its properties into new item. Notice, we are using square brackets and properties have underscore in name. It means, these are Phaser's private properties and are not intended for usage by game programmer. Unfortunately, we need this little hack to achieve what we want. In Javascript, there is no problem to use dot notation. But in Typescirpt you would get error as Phaser's private properties are not listed in phaser.d.ts definitions.

getItemOfType() method simply loops through all items currently added into items group and looks for first item with the same type:

```
        private getItemOfType(type: eItemType): Item {
            for (let i = 0; i < this._items.length; i++) {
                let object = <Item>this._items.getChildAt(i);
                if (object.itemType === type) {
                    return object;
                }
            }

            return null;
        }
```

As a last needed change into MainLayer, let's add simple getter for items group:

```
        public get items(): Phaser.Group {
            return this._items;
        }
```

Optionally, if you need debug draw of spikes bodies, you can change render method and add bold lines:

```
        public render(): void {
            //this._walls.forEachExists(function (sprite: Phaser.Sprite) {
            //    this.game.debug.body(sprite);
            //}, this);

            this._items.forEachExists(function (item: Phaser.Sprite) {
                this.game.debug.body(item);
            }, this);
        }
```

11.9 Checking for overlaps

To determine whether player hit spikes, we will check for overlaps of player's body with spikes bodies. We will do this on the same place, where we check for collision with walls – in updatePhysics() method of Play state. Open Play.ts file, locate updatephysics() and do changes in bold:

```
        private updatePhysics(): void {
            let body = <Phaser.Physics.Arcade.Body>this._player.body;

            // overlap with items - spikes, bonuses, ...
            this.physics.arcade.overlap(this._player, this._mainLayer.items, this.onOverlap, null,
this);
            if (this._gameOver) {
                return;
            }


            // clear touching
            body.touching.none = true;
```

```
            body.touching.up = body.touching.down = body.touching.left = body.touching.right =
false;

            // collision with walls
            let wallCollision = this.physics.arcade.collide(this._player, this._mainLayer.walls);

                                :
                                :
        }
```

We check for overlap between player and items. If there is any, callback method onOverlap() is called to resolve it. If overlap() method set _gameOver to true, we return.

Call to overlap() in Arcade physics also sets touching flags. This is something we do not want as we want these flags to be set only in call to collide() method. In next chapter we will implement new item – bonus jump. Imagine, we call overlap() and find that player picked it from right. It sets touching.right to true. It would later in method cause game over even if there is no collision with walls, because we test touching.right after call to collide(). To solve this, we have to clear all flags manually.

Now, we can write our onOverlap() callback:

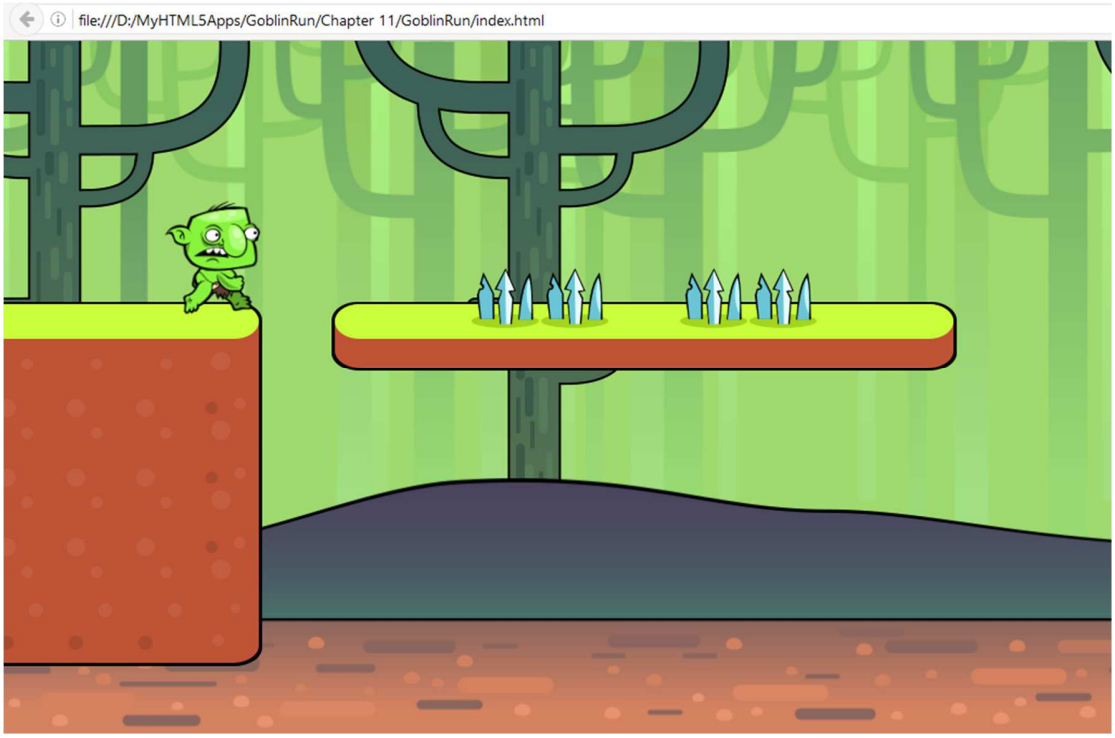
```
        private onOverlap(player: Phaser.Sprite, item: Item): void {

            if (item.itemType === eItemType.SPIKE) {
                <Phaser.Physics.Arcade.Body>this._player.body.velocity.set(0, 0);

                this._player.animateHit();
                console.log("GAME OVER - spike");

                this._gameOver = true;
            }
        }
```

We check type of item and if it is spikes we let player die. Because we immediately stop calling collide() in game loop when _gameOver is set to true, physics stop keeping player on platform and he will fall through it down to murky water.

As a last, optional step, you can uncomment call to render() in render() method if you need to see debug draw for spikes:

```
        public render() {
            this._mainLayer.render();
            //this.game.debug.body(this._player, "RGBA(255, 0, 0, 0.2)");
        }
```

Now, compile and run the game. After some time, you may encounter first spikes. Remember, we set first possible occurrence on tile 30 in Parameters.ts (SPIKES_PROB_START_TILE = 30).



Summary

With spikes added, our game is now pretty more challenging. In next chapter we will add another feature increasing challenge. We will add special bonus that gives player in-air jump.

12. Bonus jump

Let's continue in adding challenges into game. In this chapter we will add another one. We will place new type of item in level – bonus jump, which will allow player to make extra jump while still in the air. With it he can make it over some larger gaps. To give our goblin enough power to jump in air, he will have to eat tasty piece of meat:

To make thing spicier, bonus can be generated from one to three times consecutively, so player will have to overcome large distance while still in the air. We will bind this to difficulty to create harder situations later.

In resource files accompanying this book at Resources/_RES/Sprites/, you can find Bonus0.png … Bonus8.png images. These images create slightly swinging animation. Sprites are already in sprite atlas and you can reference them with Bonus0 … Bonus8 names.

12.1 Yet another Piece property

We will add new property into Piece class in generator. Open Piece.ts and add this line in the very end of class:

```
        // bonus jump
        public bonusJump: boolean;
```

Now, it is clear, why we called class Piece and not Platform. So far we were generating only platforms, regardless whether it had spikes on it or not. Now we are generating different kind of object – bonus. If you plan to extend generator in future, you can think of more features. What about jet pack? When picked, player can fly in Flappy bird manner for some time and you can generate yet another kind of piece – stalactites from bottom and top to make game more challenging!

12.2 New parameters and difficulty

As with spikes, we will need some additional parameters to control how often bonus is generated and also how many bonuses can be spawn consecutively.

Open Parameters.ts file and in part with difficulty parameters add these new ones:

```
        // bonus jump probability
        public static BONUS_JUMP_PROB_MIN = 0;
        public static BONUS_JUMP_PROB_MAX = 30;
        public static BONUS_JUMP_START_TILE = 50;
        public static BONUS_JUMP_END_TILE = 200;
```

```
        // bonus jump count
        public static BONUS_JUMP_COUNT_MIN = 1;
        public static BONUS_JUMP_COUNT_MAX = 3;
        public static BONUS_JUMP_COUNT_START_TILE = 50;
        public static BONUS_JUMP_COUNT_END_TILE = 300;
```

This is the same as with other difficulty parameters earlier in this book. We put minimum and maximum value for difficulty parameter and these bounds are reached from start tile to end tile. To calculate current value based on distance travelled, open Difficulty.ts and on top of it add two new private variables:

```
        // bonus jump probability
        private _bonusJumpProbability: number;
        // bonus jump count
        private _bonusJumpCount: number;
```

In constructor set its initial values:

```
            // initial bonus jump probability
            this._bonusJumpProbability = Parameters.BONUS_JUMP_PROB_MIN;
            // intial bonus jump count
            this._bonusJumpCount = Parameters.BONUS_JUMP_COUNT_MIN;
```

Add two getters:

```
        public get bonusJumpProbability(): number {
            return this._bonusJumpProbability;
        }

        // ------------------------------------------------------------------------
        public get bonusJumpCount(): number {
            return this._bonusJumpCount;
        }
```

In update() method calculate current difficulty values. It is again done in the same way as for all previous difficulty parameters:

```
            // bonus jump probability
            this._bonusJumpProbability = Math.round(this.mapLinear(tileX,
                Parameters.BONUS_JUMP_START_TILE, Parameters.BONUS_JUMP_END_TILE,
                Parameters.BONUS_JUMP_PROB_MIN, Parameters.BONUS_JUMP_PROB_MAX));

            // bonus jump count
            this._bonusJumpCount = Math.round(this.mapLinear(tileX,
                Parameters.BONUS_JUMP_COUNT_START_TILE, Parameters.BONUS_JUMP_COUNT_END_TILE,
                Parameters.BONUS_JUMP_COUNT_MIN, Parameters.BONUS_JUMP_COUNT_MAX));
```

Finally, for debug purposes, you can adjust toString() method to print new difficulty parameters:

```
        // ------------------------------------------------------------------------
        public toString(): string {
            return "platformLengthDecrease: " + this._platformLengthDecrease +
                ", jumpLengthDecrease: " + this._jumpLengthDecrease +
                ", spikesProbabilty: " + this._spikesProbability +
                ", bonusJumpProbability: " + this._bonusJumpProbability +
                ", bonusJumpCount: " + this._bonusJumpCount;
        }
```

Changes done in this part were nothing than routine.

12.3 Changes in Generator

Now we will make changes into Generator class. These changes will be very simple though in many places.

As first, we will add onBonusJump signal:

```
        // dispatch new piece, previous piece, jump number
        public onBonusJump: Phaser.Signal = new Phaser.Signal();
```

As with random platforms and patterns, we will dispatch signal to inform game that bonus was generated. In our game, we will not listen to this signal. But from beginning, we are building generator as independent on game as possible. Different games may have use for it.

Next we have to make two changes into generate() method. Locate section, where new offset for x is calculated and add bolded lines:

```
        private generate(lastPosition: Phaser.Point, difficulty: Difficulty,
            length: number, offsetX: number, offsetY: number, bonusJump: boolean): Piece {

                                :
                                :

            // X POSITION
            let shiftX = offsetX;
            // calculate is offsetX is not forced or offsetY was forced, but final value is
different
            if (shiftX === UNDEFINED || (offsetY !== UNDEFINED && offsetY !== piece.offset.y)) {
                let minX = this._jumpTables.minOffsetX(piece.offset.y);
                let maxX = this._jumpTables.maxOffsetX(piece.offset.y);

                // if bonus jump or previous piece was bonus jump,
                // then make gap at least one cell width (if possible) (= offset 2)
                if (bonusJump || (this._lastGeneratedPiece !== null &&
this._lastGeneratedPiece.bonusJump)) {
                    minX = Math.min(Math.max(minX, 2), maxX);
                }

                // decrease maximum jump distance with jump decreaser in difficulty to
                // make jumps easier in the beginning of game
                // But be sure it does not fall under minX
                if (!bonusJump) {
                    maxX = Math.max(minX, maxX + difficulty.jumpLengthDecrease);
                }

                // position of next tile in x direction
                shiftX = this._rnd.integerInRange(minX, maxX);
            }

                                :
                                :
```

When generating bonus jump, we do not want it stick closely to previous piece. As well we do not want new platform to be stick to previous piece if it was bonus jump. We want at least one cell gap (= offset at least 2) if possible. For this, we try to set new minX to at least 2, but we make sure it does not exceed maxX.

One of our difficulty parameters is decreasing maximum gap in the beginning of game. We do not want this to take effect if we are generating bonus jump. So, add condition that will skip it for bonus jumps.

Second change is close to method's end:

```
                    :
                    :

            // BONUS JUMP
            piece.bonusJump = bonusJump;


            // console.log(difficulty.toString());


            // RESULT
            this._lastGeneratedPiece = piece;
            return piece;
        }
```

It simply sets property of Piece.

Head to generatePieces() method. It will change a lot. Below is new version of it with changes in bold:

```
        public generatePieces(lastTile: Phaser.Point, difficulty: Difficulty): void {
            let probability = this._rnd.integerInRange(0, 99);

            if (probability < difficulty.bonusJumpProbability &&
                this._lastGeneratedPiece !== null && !this._lastGeneratedPiece.bonusJump) {
                this.generateBonusJump(lastTile, difficulty);
            } else {
                probability = this._rnd.integerInRange(0, 99);

                if (probability < Parameters.GENERATE_RANDOM) {
                    this.generateRandomly(lastTile, difficulty);
                } else {
                    this.generatePattern(lastTile, difficulty);
                }
            }
        }
```

If there is probability for generating bonus jump, we check, whether previous piece was also bonus jump. If not, we call generateBonusJump(), which will be listed soon. If yes, we generate platforms instead. Reason for this check is to prevent too many bonus jumps series one after another. generateBonusJump() creates one series of bonus jumps and how many of them will be there is controlled with difficulty parameter bonusJumpCount:

```
        private generateBonusJump(lastTile: Phaser.Point, difficulty: Difficulty): void {
            // random number of consecutive jump bonuses
            let jumps = this._rnd.integerInRange(Parameters.BONUS_JUMP_COUNT_MIN,
difficulty.bonusJumpCount);

            let piece;
            let prevPiece: Piece = this._lastGeneratedPiece;

            for (let i = 0; i < jumps; i++) {
                // first jump in row of jumps?
                if (i === 0) {
                    piece = this.generate(lastTile, difficulty, 1, UNDEFINED, UNDEFINED, true);
                } else {
                    piece = this.generate(prevPiece.position, difficulty, 1, prevPiece.offset.x,
prevPiece.offset.y, true);
                }

                // add to stack
                this.addPieceIntoQueue(piece);
```

```
                // dispatch signal
                this.onBonusJump.dispatch(piece, prevPiece, i);

                prevPiece = piece;
            }
        }
```

First we determine how many consecutive bonus jumps to generate and then we do it in loop. To prevent generation being too random, we generate randomly only first. Others use the same x and y offsets. New bonus jumps are put into queue of pieces exactly in the same way as platforms. For every bonus jump generated, we notify all interested listeners with onBonusJump signal.

This closes changes needed on generator side. Rest of changes is on game side.

12.4 Animation and block definition

Now we will make changes into Animations and BlockDefs classes, similar to what we made when adding spikes.

Open Animations.ts and add this lines:

```
        public static BONUS_JUMP_ANIM = [
            "Bonus0", "Bonus1", "Bonus2", "Bonus3", "Bonus4",
            "Bonus4", "Bonus4", "Bonus3", "Bonus2", "Bonus1",
            "Bonus0", "Bonus5", "Bonus6", "Bonus7", "Bonus8",
            "Bonus8", "Bonus8", "Bonus7", "Bonus6", "Bonus5"];
```

It defines frames for bonus jump animation.

Then open BlockDefs.ts and in the bottom add this line:

```
        // BONUS JUMP
        public static BONUS_JUMP: ITileDef = { name: "bonusJump", anchorX: 0.5, anchorY: 0.5,
bodyOffsetX: 7, bodyOffsetY: 7, bodyWidth: 50, bodyHeight: 50 };
```

As for spikes, it defines animation name, anchors and body size and offset.

12.5 Changes into MainLayer

With animation and block defined, we can start changes in MainLayer class. Open it and as first thing add new animation in constructor (line in bold):

```
                // add animations
                item.animations.add("spikes", Animations.SPIKE_ANIM, 10, true);
                item.animations.add("bonusJump", Animations.BONUS_JUMP_ANIM, 10, true);
```

Locate generate() method and in PROCESS_PIECE branch of switch statement change code inside while loop (changes again in bold):

```
                            // process piece
                            while (length > 0) {
                                if (piece.bonusJump) {
                                    this.addBonus(this._lastTile.x, this._lastTile.y);
                                } else {
                                    this.addTiles(this._lastTile.x, this._lastTile.y,
                                        tileType, (<IGamePiece>piece).isPlatform,
                                        (piece.spikesPattern & (1 << (length - 1))) > 0);
                                }
```

```
                                                :
                                                :
                        }
```

If piece is bonus jump, we will use addBonus() method to set its visual representation. This method is short and straightforward:

```
        private addBonus(x: number, y: number): void {
            let jumpBonus = this._itemsPool.createItem();
            jumpBonus.itemType = eItemType.BONUS_JUMP;

            jumpBonus.position.set(x * 64 + 32, y * 64);

            jumpBonus.exists = true;
            jumpBonus.visible = true;

            this.setupItem(jumpBonus, BlockDefs.BONUS_JUMP, true, false);

            if (jumpBonus.parent === null) {
                this._items.add(jumpBonus);
            }
        }
```

We get free item from pool and set its type to BONUS_JUMP. Then we set its position, make it visible and call setupItem to apply anchors and physics body settings defined in BONUS_JUMP block def.

As bonus can be picked by player, we will need some way, how to let it disappear. For it we will define removeItem() method:

```
        public removeItem(item: Item): void {
            this._items.remove(item);
            item.parent = null;
            this._itemsPool.destroyItem(item);
        }
```

12.6 Bonus jump overlaps

If you run game now, you will see bonus jumps are generated and visible on screen. But player will not pick it on collision. To solve this and to handle extra in-air jump, we have to make a few changes into Play state. Open Play.ts and add new private variable on top (bold code):

```
        // player
        private _player: Player;
        private _jumpTimer: number = 0;
        private _bonusJump: boolean = false;
```

Then go to updatePhysics() method and locate part commented as start jump. Again, make changes in bold:

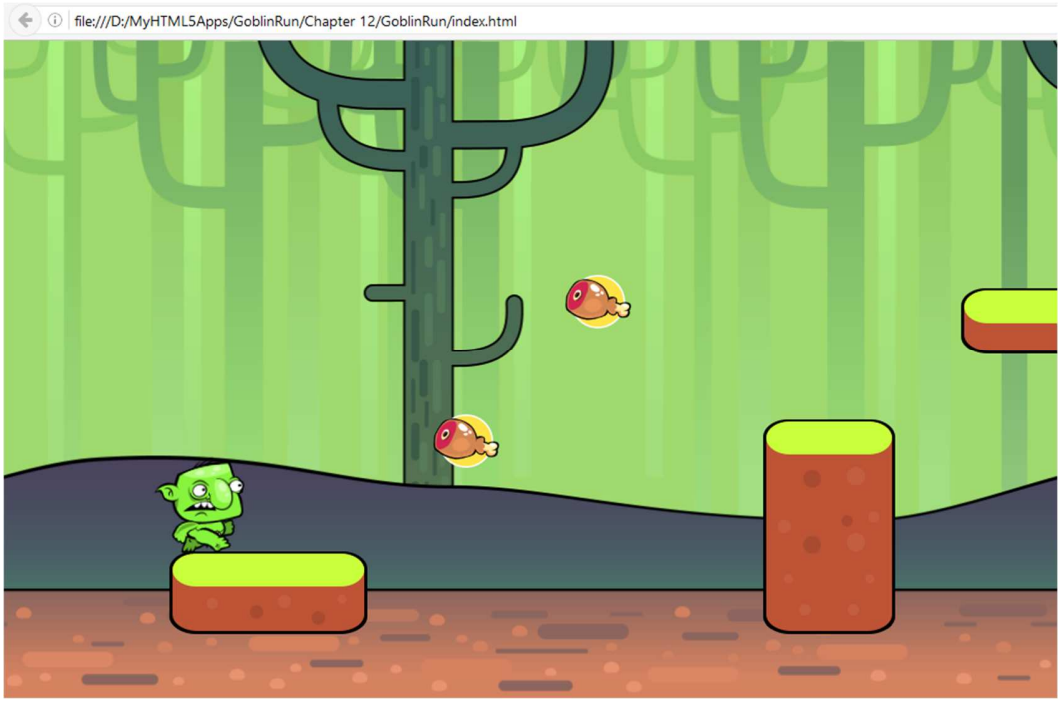
```
            // start jump
            if ((this._justDown && body.touching.down && this.game.time.now > this._jumpTimer) ||
                (this._justDown && this._bonusJump)) {
                body.velocity.y = jumpTable.maxJumpVelocity;
                this._jumpTimer = this.game.time.now + 150;
                this._justDown = false;
                this._bonusJump = false;

                this._player.animateJump();
            }
```

We added another condition to make jump. Carefully check if parentheses in your code are the same as in listing – we enclosed also original condition! Previously, jump was only possible if player is standing on floor. Now, it is also possible if _bonusJump is true. It will be set true when player picks bonus jump item, which is handled in onOverlap() callback (changes in bold):

```
        private onOverlap(player: Phaser.Sprite, item: Item): void {

            if (item.itemType === eItemType.SPIKE) {

                        :
                        :

            } else if (item.itemType === eItemType.BONUS_JUMP) {
                this._bonusJump = true;
                this._mainLayer.removeItem(item);

            }
        }
```

In onOverlap() callback we check what type of item player collided with. If it is BONUS_JUMP, we set internal _bonusJump variable true and ask main layer to remove it from screen.

Compile and run game again. Now, when you collide with bonus jump, it should disappear and you can make another jump while still in the air:



Remember, we set parameters so that bonus jumps appear later in game. Tweak parameters if you do not want to wait.

Summary

In this chapter we increased challenge a bit again. With added spikes and bonus jumps our game is more interesting. In next chapter we will add some gold our goblin can gather.

13. Gold

In previous two chapters we increased game challenge. In this chapter we will spice it with gold nuggets our goblin can gather. It does not increase challenge, but it is nice and it spices game. It also shows that we can add some new elements without making changes into generator.

13.1 Idea

Idea is to give player something what can be picked when running. In our case it will be gold nugget:

It actually has no impact on difficulty, it will only add some score. But for us it is good opportunity to show how to import bitmap fonts and how to make UI item in game. Total score points will be displayed in top left corner like this:

As already mentioned, adding gold will not require changes in generator. Instead, we can think of it as platform decoration and we will add only very simple rules into MainLayer to put gold on platform when it is processed.

13.2 Block definition

We will again use ITileDef interface to define block in BlockDefs class. Open it and add this line in the end:

```
        // GOLD
        public static GOLD: ITileDef = { name: "Gold", anchorX: 0.5, anchorY: 1, bodyOffsetX: 0,
bodyOffsetY: 0, bodyWidth: 43, bodyHeight: 43 };
```

Gold will not be animated, so in this case name is name of frame with gold image in atlas. If it seems too static to you, create additional frames and try to change it into animated sprite. Just follow how we did it for spikes and bonus jumps.

13.3 Placing gold on platforms

Our strategy for placing gold on platforms is to choose random number from 3 to 6. Then with every processed platform decrease it and if value equals zero, put gold on top of platform. With this, platforms will not be overloaded with gold which would look ugly.

Open MainLayer.ts file and add these three bold lines on top of MainLayer class:

```
    export class MainLayer extends Phaser.Group {

        private static GOLD_COUNTER_MIN = 3;
        private static GOLD_COUNTER_MAX = 6;
        private _timeForGold = MainLayer.GOLD_COUNTER_MIN;

                        :
                        :
```

We defined minimum and maximum bounds for internal counter _timeForGold, which is initially set to lowest value.

Locate generate() method and make changes in bold in PROCESS_PIECE branch of switch statement:

```
        public generate(leftTile: number): void {

                            :
                            :

                    case eGenerateState.PROCESS_PIECE:
                        {

                                        :
                                        :

                            // decrease gold counter
                            if (!piece.bonusJump && piece.spikesPattern === 0) {
                                --this._timeForGold;
                            }


                            // process piece
                            while (length > 0) {
                                if (piece.bonusJump) {
                                    this.addBonus(this._lastTile.x, this._lastTile.y);
                                } else {
                                    this.addTiles(this._lastTile.x, this._lastTile.y,
                                        tileType, (<IGamePiece>piece).isPlatform,
                                        (piece.spikesPattern & (1 << (length - 1))) > 0);

                                    // add gold
                                    if (this._timeForGold === 0 && length > 1) {
                                        this.addGold(this._lastTile.x, this._lastTile.y);
                                    }
                                }

                                            :
                                            :

                            }


                            // reset gold counter?
                            if (this._timeForGold === 0) {
                                this._timeForGold = this.game.rnd.integerInRange(
                                    MainLayer.GOLD_COUNTER_MIN,MainLayer.GOLD_COUNTER_MAX);
                            }

                                        :
                                        :

                            break;
```

```
                    }

                     :
                     :

    }
```

First, we decrease _timeForGold counter, but only in case that currently processed piece is platform, not bonus jump and has no spikes on it.

Inside loop processing piece we check whether counter is zero and if remaining length of platform is higher than one. To make it look nice, we will generate as many gold nuggets on platform as is its length minus one. Gold will be centered between tiles. So, on platform with length two, there will be only one gold. On platform with length five, there will be four gold nuggets, etc. Actual item representing gold is added in addGold() method, which will be presented soon.

After loop finished, we check if counter is zero and needs resetting with new random value.

Here is code for addGold() method:

```
    private addGold(x: number, y: number): void {
        let gold = this._itemsPool.createItem();
        gold.itemType = eItemType.GOLD;

        gold.position.set(x * 64 + 64, y * 64 + 8);

        gold.exists = true;
        gold.visible = true;

        this.setupItem(gold, BlockDefs.GOLD, false, false);

        if (gold.parent === null) {
            this._items.add(gold);
        }
    }
```

This method is very similar to addBonus() method we wrote for bonus jumps. It just takes free item from pool, sets its parameters and makes it visible and child of items group.

If you run game now, you will see gold on some platforms, but goblin will just ignore it as we did not yet defined collisions handling between it and player.

13.4 Collisions with gold

Adding collision with gold is very easy as we have everything needed already prepared. Go into Play state in Play.ts file and new else-if statement in the end of onOverlap() method:

```
    private onOverlap(player: Phaser.Sprite, item: Item): void {

        if (item.itemType === eItemType.SPIKE) {

                        :
                        :

        } else if (item.itemType === eItemType.BONUS_JUMP) {

                        :
                        :
```

```
            } else if (item.itemType === eItemType.GOLD) {
                this._mainLayer.removeItem(item);
            }
        }
```

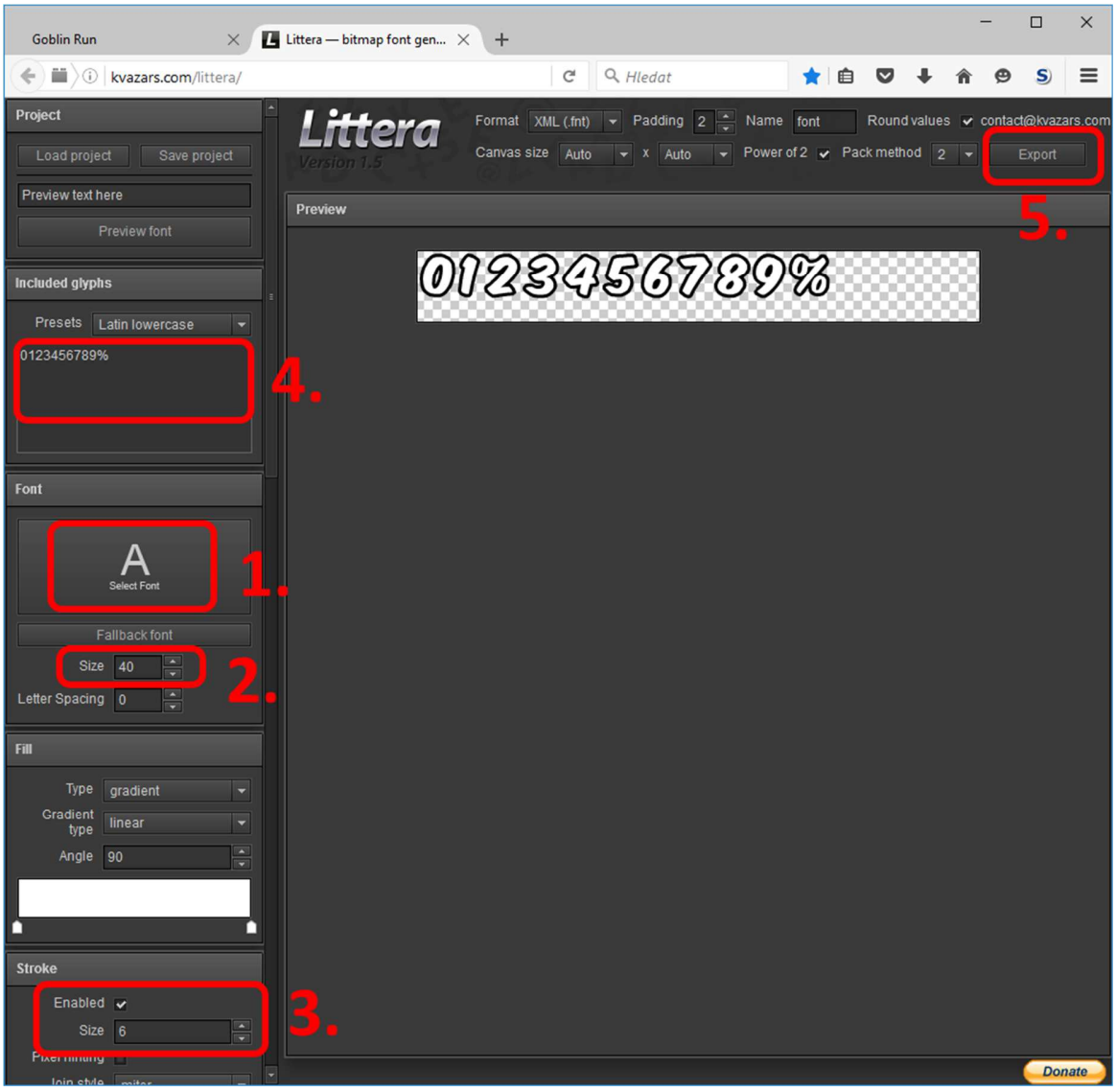
For the time being, we are just removing gold player collides with.

After this change goblin can pick gold on platforms. Next, we will add some UI element to display how many points player got for gathered gold.

13.5 Font

We will need font for displaying score. There is already one prepared in assets. You can also find it in Resources/_RES/Font/. For game, two files are important – Font.png and Font.xml. Font.png is image with all font characters. In fact, it is sprite atlas. Font.xml is file with metadata for it. It has information on where in atlas is character placed and how big it is along with some font specific properties like how much advance in x direction when character is printed, what are x and y offsets, etc. For some fonts, metadata can also contain kerning data, which are data for space adjustments between specific pairs of characters. For example, for pair “VA” space between characters should be smaller than for “MA”. If the same space is applied to all character pairs, it feels wider for “VA” than for “MA” because of “V” shape.

Font used in game is free Komika Poster font by Apostrophic Labs. If you want to create files needed for game by yourself, download it from http://www.1001freefonts.com/komika_poster.font. Downloaded .zip contains several .ttf files. We use KOMIKAX_.ttf. To turn it into game files, I used on-line tool Littera by Kvazars at http://kvazars.com/littera/. Littera is very easy to use, look on image below. First, choose font, you want to use. Set its size and define whether you want outlines (Stroke) around it and how thick. Finally, put into “Included glyphs” window all characters you need – we need only numbers and “%”. You can define more properties like glow or shadow and all changes are displayed in main window in real time.



You can load and save your project to local disk. File LitteraProject.ltr in resources is example of it. Font used is saved along with other data, so when you save Littera project, you do not need .ttf file anymore.

When you are happy with results, press Export button in top right and you will be offered to download .zip file that contains atlas and metadata.

Metadata has .fnt extension by default. Change it to .xml to prevent problems with some servers. I also made little correction inside of it. Close to top, there is tag “common” with “lineHeight” property. When you apply anchors on bitmap text, it takes into account this property. This property contains complete line height, including ascent and descent, which is not needed for our simple numbers. To position text it sometimes requires hit and miss approach. In our case, we will make things simple and set lineHeight value to 40, which is more or less height of characters we included into export. We can do it because none of our characters has descent and also we will not use it for multiline text. If we set anchor y to 0.5 we will know that characters are vertically centered.

With font files ready and copied into assets folder of game, we can load them in Preload state. Open Preload.ts and in preload() method add these lines:

```
            // font
            this.load.bitmapFont("Font", "assets/Font.png", "assets/Font.xml");
```

13.6 Score UI element

For element displaying score we will build separate class derived from Phaser.Group. Add new file ScoreUI.ts into Game folder and put this code into it:

```
namespace GoblinRun {

    export class ScoreUI extends Phaser.Group {

        private _icon: Phaser.Sprite;
        private _text: Phaser.BitmapText;

        private _tween: Phaser.Tween;

        // ---------------------------------------------------------------------
        public constructor(game: Phaser.Game, parent: PIXI.DisplayObjectContainer) {
            super(game, parent);

            // icon
            this._icon = new Phaser.Sprite(game, 0, 0, "Sprites", "GoldUI");
            this._icon.anchor.set(0.5, 0.5);
            this.add(this._icon);

            // text
            this._text = new Phaser.BitmapText(game, this._icon.width / 2 * 1.2, 0,
                "Font", "0", 40, "left");
            this._text.anchor.y = 0.5;
            this.add(this._text);

            // tween
            this._tween = game.add.tween(this._icon.scale).
                to({ x: 1.2, y: 1.2 }, 100,
                function (k: number) {
                    return Math.sin(Math.PI * k);
                }, false, 0);
        }

        // ---------------------------------------------------------------------
        public set score(score: number) {
            this._text.text = "" + score;
        }

        // ---------------------------------------------------------------------
        public bounce(): void {
            if (!this._tween.isRunning) {
                this._tween.start();
            }
        }
    }
}
```

ScoreUI element has two parts – icon and text. Beside this, it holds one reusable tween, that makes very short icon size bounce when gold is picked. In constructor we build all these parts step by step. Only interesting thing is custom easing function for tween. All easing functions take one parameter, that is in range from 0 to 1. Easing function should calculate and return value based on this parameter. Usually

values from 0 to 1 are returned to tween target property from start value to end value. In our case we need only small bounce of icon. For it we tween icon's scale property and we want it to be the same after bounce as it was before it. So, we do not tween from start to end value, but we tween from start through end to start value. To achieve this, our easing function returns values starting and ending with zero.

Two public methods will help us to control this class. With score() setter we can change displayed value and with bounce() method we can fire icon bounce.

Now we have to wire this new class into game.

13.7 Adding score to game

Open Play.ts file and add new private variables into it:

```
        // score
        private _score: number = 0;
        private _scoreUI: ScoreUI;
```

Then in create() method, just before input section, create new instance of ScoreUI class:

```
            // score UI on screen
            this._scoreUI = new ScoreUI(this.game, this.world);
            this._scoreUI.fixedToCamera = true;
            this._scoreUI.cameraOffset.set(45, 30);
```

As we do not want score UI element scroll with game we set it fixed to camera. With camera offset we can fine tune its position to leave some small space on left and top.

Finally, in onOverlap() callback method add a few bolded lines in part handling overlap with gold:

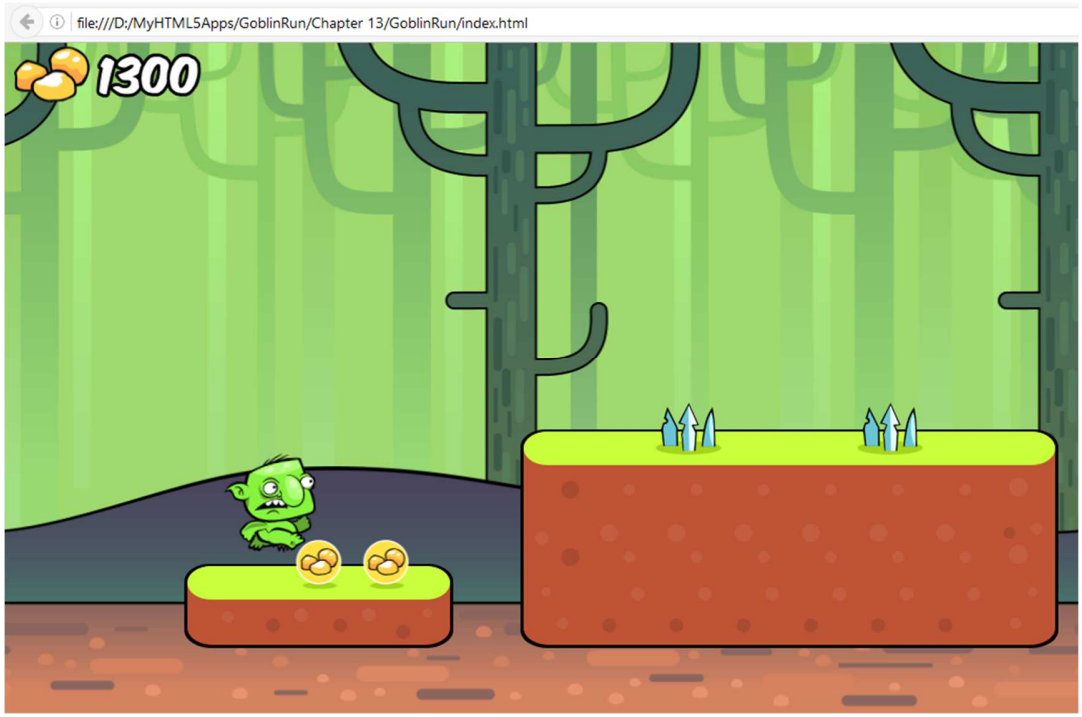
```
            } else if (item.itemType === eItemType.GOLD) {
                this._mainLayer.removeItem(item);

                // add score and make bounce effect of score icon
                this._score += 100;
                this._scoreUI.score = this._score;
                this._scoreUI.bounce();
            }
```

For every gathered gold nugget, we add 100 points. Then we update text in score UI element and let it bounce.

As score numbers can cover goblin character if platforms are upper part of screen, we will move him a little bit to right. Go into update() method and on top of it change camera distance from 192 to 256.

```
                // move camera
                this.camera.x = this._player.x - 256; //192;
```

Now compile and run game. This is result you should see on screen:



Summary

In this chapter we added gold nuggets to gather during run and score to game. We did not need to make any changes into generator as gold nuggets placed on platforms were treated only as simple platform decoration. If you have some collectable item, that is more bind to how level is generated, you may consider to shift its generating into generator.

In next chapter we will try to polish game a little with particles.

14. Particles

In this chapter we will add particles into game to make it look better. It is easy task but requires usually lot of writing because particle systems have many settings to fine tune behavior of individual particles. Every particle system has two parts – emitter and particles, and default particle system built into Phaser is no exception. We will create several emitters for different occasions:

- when player falls into mud, we will spawn mud particles to make big splash,
- when player runs or lands from air, we will add dust particles,
- when player gathers gold, we will spawn gold sparks,
- when player picks bonus jump, we will let it nicely disappear.

14.1 Mud splash

First, we will add mud particles that splash when player falls into murky water. These are sprites we will use:

Open Player.ts file in Game folder and on top of it add new private variable:

```
        private _mudEmitter: Phaser.Particles.Arcade.Emitter;
```

Head to constructor and in the end of it create _mudEmitter:

```
            let emitter = new Phaser.Particles.Arcade.Emitter(game, 10, 60, 20);
            emitter.makeParticles("Sprites", ["MudParticle0", "MudParticle1"]);
            emitter.setXSpeed(-100, 100);
            emitter.setYSpeed(-500, -200);
            emitter.gravity = -Generator.Parameters.GRAVITY + 800;

            this.addChild(emitter);
            this._mudEmitter = emitter;
```

Mud emitter is child of player sprite. In first line we set its relative position and last parameter determines number of particles this emitter has. Twenty should be enough to make big splash. With next call to makeParticles() we define texture and frames to use. We have two different mud particles and when it comes to emitting, emitter will choose randomly from them. On next lines, speed parameters of emitter are set. We set x speed from -100 to 100 to splash equally into sides and y speed from -500 to -200 to splash up vertically.

Interesting line is the one where we set gravity. With arcade physics running, particles are affected with gravity set for physics world. If we set some gravity for particle emitter, then both these values affect all particles. To set different gravity for emitter than for physics world, reset it with negative value of world gravity first and then add gravity you need. Our world gravity is defined in generator parameters.

After setting gravity we just add emitter into scene tree. As it is added as last child, it will be rendered on top of all other children, which is exactly what we need – we want to make mud splash over goblin hero.

Locate animateDeath() method and add line in bold:

```
        public animateDeath(): void {

                    :
                    :

            // emit mud particles
            this._mudEmitter.explode(0, 20);
        }
```

We will spawn all particles at once when player falls down.

Because we have Player class opened, let's make one more small change. Add following getter which we will need soon in next parts:

```
        public get animName(): string {
            return this._spriterGroup.currentAnimationName;
        }
```

You can build and run the game and observe results. Mud particles explode when goblin falls into mud:

14.2 Dust

Next particles we add will present dust when goblin is running or landing on ground from jump. Emitter for mud was part of player, so it was moving together with him. All individual particles are children of emitter. In fact, emitter is derived from Phaser.Group. It means, if you move emitter all particles will move too. This is not something we want for dust particles. We want them to spawn at some position and stay there. For this reason, we cannot add dust emitter to player. We also want dust particles to be in front of all current layers. As a solution we will add dust emitter into Play state as layer over player and under the score UI. Dust particle image looks like this:

Open Play.ts and add a few new private variables:

```
        // dust emitter
        private _dustEmitter: Phaser.Particles.Arcade.Emitter;
        private _touchingDown: boolean = false;
        private _runCounter: number = 0;
```

In _dustEmitter we will hold reference to emitter. _touchingDown will store last state of physics body.touching.down to easily detect when player landed on platform. _runCounter is help variable used to countdown time for next particle when goblin is running.

Go into create() method and between player and score UI put new lines in bold:

```
            // set player
                      :
                      :

            // dust particles
            let emitter = new Phaser.Particles.Arcade.Emitter(this.game, 0, 0, 16);
            emitter.makeParticles("Sprites", ["DustParticle"]);
            emitter.setYSpeed(-50, -20);
            emitter.setRotation(0, 0);
            emitter.setAlpha(1, 0, 500, Phaser.Easing.Linear.None);
            emitter.gravity = -Generator.Parameters.GRAVITY;

            this.world.add(emitter);
            this._dustEmitter = emitter;


            // score UI on screen
                      :
                      :
```

Here again we set emitter and its parameters. We will let particles rise slowly up, they will not rotate and they will fade out.

Next changes are in update() method:

```
        public update() {

                        :
                        :

            // update player animations
            let body = <Phaser.Physics.Arcade.Body>this._player.body;
            this._player.updateAnim(body.velocity.y >= 0 && body.touching.down, body.velocity.y,
this._gameOver);


            // dust particles
            if (!this._touchingDown && body.touching.down && !this._gameOver) {
                this.emitDustLanding();
            }
            this._touchingDown = body.touching.down;

            this.emitDustRunning();


            // move background
            this._bg.updateLayers(this.camera.x);
        }
```

If in previous frame _touchingDown was false and now it is true, then we landed on ground and it is time to spawn dust particles wit emitDustLanding() method. We spawn it only if not game over otherwise it would be spawned also when player lands on spikes.

On every frame we call emitDustRunning(). It checks _runCounter to see whether it is time to spawn particles.

Let's now define both emitDustLanding() and emitDustRunning() methods:

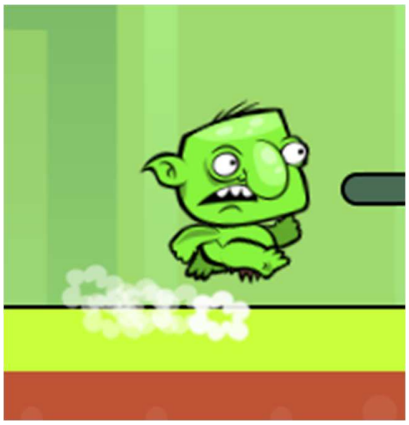
```
    public emitDustLanding(): void {
        this._dustEmitter.emitX = this._player.x + 20;
        this._dustEmitter.emitY = this._player.y + 90;
        this._dustEmitter.setXSpeed(-100, 0);
        this._dustEmitter.explode(500, 2);
        this._dustEmitter.setXSpeed(0, 100);
        this._dustEmitter.explode(500, 2);
    }
```

With emitX and emitY we can set point where to spawn particle. When player lands on platform we want to spawn four particles. Two of them moving in random speed right and two to left. Four particles are too low number to be sure that part of it will move left and part right. There is high probability that all of them will move in single direction. To prevent it we emit first two of them moving left and then others two moving right.

```
    public emitDustRunning(): void {
        if (this._player.animName !== "run") {
            return;
        }

        let counter = Math.floor(this.game.time.time / 250);
        if (counter > this._runCounter) {
            this._runCounter = counter;

            this._dustEmitter.emitX = this._player.x;
            this._dustEmitter.emitY = this._player.y + 80;
            this._dustEmitter.setXSpeed(-100, 0);
            this._dustEmitter.emitParticle();
        }
    }
```

Before we emit particles when running, we first check whether player is in run animation. Then we check whether counter increased and so it's time to emit new particle. We take current time in milliseconds, divide it by 250 and round down to nearest integer. If result is higher than last value stored in _runCounter, then we store new value and emit new particle slowly moving to the left.

As with mud particles you can immediately test result:



14.3 Gold and bonus jump particles

Last two emitters will be created together as both of them reside in MainLayer. When player picks gold nugget we will add explosion of gold sparks that scales down quickly and when player picks bonus jump we will scale down and rotate its image. For bonus jump particles we will use image Bonus0 from atlas. For gold sparks we will use image named GoldParticle:

Open MainLayer.ts and in top of it add two new private variables to hold emitters references:

```
        private _bonusEmitter: Phaser.Particles.Arcade.Emitter;
        private _goldEmitter: Phaser.Particles.Arcade.Emitter;
```

Go into constructor and near to end, after creating group for items, add call to constructEmitters() method.

```
                    :
                    :

            // items group
            this._items = new Phaser.Group(game, this);

            // emitters
            this.constructEmitters();

                    :
                    :
```

Into this new method we will put all the code needed to set up emitters:

```
        public constructEmitters(): void {
            // bonus jump
            let emitter = new Phaser.Particles.Arcade.Emitter(this.game, 0, 0, 5);
            emitter.makeParticles("Sprites", "Bonus0");
            emitter.setXSpeed(0, 0);
            emitter.setYSpeed(0, 0);
            emitter.setRotation(-360, -360);
            emitter.lifespan = 500;
            emitter.setScale(1.0, 0, 1.0, 0, 500);
            emitter.gravity = -Generator.Parameters.GRAVITY;

            this.add(emitter);
```

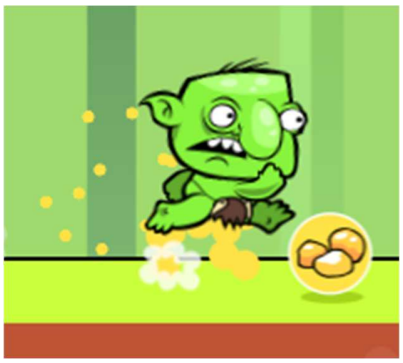
```
            this._bonusEmitter = emitter;

            // gold
            emitter = new Phaser.Particles.Arcade.Emitter(this.game, 0, 0, 48);
            emitter.makeParticles("Sprites", "GoldParticle");
            emitter.setXSpeed(-150, 150);
            emitter.setYSpeed(-200, 50);
            emitter.setRotation(0, 0);
            emitter.lifespan = 550;
            emitter.setScale(1.0, 0, 1.0, 0, 550);
            emitter.gravity = -Generator.Parameters.GRAVITY;

            this.add(emitter);
            this._goldEmitter = emitter;
        }
```

We are setting parameters for both emitters. Try to read it and guess how final effect will look like. Then head to removeItem() method. It is good place where to add code for spawning particles.

```
        public removeItem(item: Item): void {

            // make particle effect
            if (item.itemType === eItemType.BONUS_JUMP) {
                this._bonusEmitter.emitX = item.x;
                this._bonusEmitter.emitY = item.y;
                this._bonusEmitter.emitParticle();
            } else if (item.itemType === eItemType.GOLD) {
                this._goldEmitter.emitX = item.x;
                this._goldEmitter.emitY = item.y - 25;
                this._goldEmitter.explode(500, 12);
            }

                        :
                        :
        }
```

Code is very simple. We only set position where new particles shall be spawned. In case of bonus jump only one particle is emitted. In case of gold we let explode twelve of them.

if you now run the game you will have all four particle effects in place.



Summary

Finally, word of warning. With particles you can simulate lot of effects in game from smoke, fire, rain to sparks and rocket trails. Lot of particles can eat big portion of processing power. So it is good to keep its

number proportional to what you want to achieve. Sometimes frame animation with prerendered effect may be less costly than spawning tens or hundreds of particles.

In our game we kept number of particles low and with added effects it looks a bit better again.

15. Shadow

In this chapter we will add small sprite image under player that will represent shadow. We want to display it over walls but under objects. We also want it completely disappear when player jumps over mud. In Phaser we cannot set z depth like in some other engines (for example Unity). Solution would be to put shadow and all code it needs into MainLayer class. But we also need player's actual position to update it and MainLayer does know nothing about player. More, MainLayer is about level, not about something related to player. To keep game well-arranged and tidy we will create separate class for shadow, instantiate it during creating all game visual layers and nest it into MainLayer at needed position from out of it.

For shadow sprite we will use sprite named Shadow from atlas:

15.1 Shadow class

In Game folder add new file Shadow.ts and start with adding private variables:

```
namespace GoblinRun {

    export class Shadow extends Phaser.Sprite {

        private _playerPosition: Phaser.Point;
        private _walls: Phaser.Group;
```

We will keep reference to player's position to place shadow on right place when updating it. Reference to walls in MainLayer is kept because we will project shadow on top of the current platform below player.

In constructor we create shadow sprite and fill these references. By default, shadow is not visible.

```
        public constructor(game: Phaser.Game, playerPosition: Phaser.Point, walls: Phaser.Group) {
            super(game, 0, 0, "Sprites", "Shadow");

            this._playerPosition = playerPosition;
            this._walls = walls;

            this.anchor.set(0.5, 0.5);
            this.visible = false;
        }
```

Key part is postUpdate() method. This method is automatically called by Phaser during world's post-update cycle after all updates were done during regular update cycle. As the method is little bit longer we will go through it part by part.

```
        public postUpdate(): void {

            // left edge of shadow sprite
            let xLeft = this._playerPosition.x - 20;
```

```
            let minYleft = Number.MAX_VALUE;

            // right edge of shadow sprite
            let xRight = this._playerPosition.x + 20;
            let minYright = Number.MAX_VALUE;
```

Idea is to cast two rays downwards, offset from player's current position to the right and left. We can hit multiple wall tiles when there is solid block growing from mud. So, we will track which one is the highest. We start in infinite depth given with Number.MAX_VALUE and if we also end with this value, we know there was no wall tile below.

Now let's iterate through all wall tiles and check if ray hits them. If yes, check its y level and if tile is higher (smaller y) than previous one, store its y.

```
            // go through all walls and find the top one under left and right edge
            for (let i = 0; i < this._walls.length; i++) {
                let wall = <Phaser.Sprite>this._walls.getChildAt(i);

                // is left edge on this wall tile?
                if (wall.x <= xLeft && wall.x + 64 > xLeft) {
                    // is it higher than previous wall
                    minYleft = Math.min(minYleft, wall.y);
                }

                // is right edge on this wall tile?
                if (wall.x <= xRight && wall.x + 64 > xRight) {
                    // is it higher than previous wall
                    minYright = Math.min(minYright, wall.y);
                }
            }
```

After loop we can check results. We will draw sprite only if there is some tile below both rays and only if both rays found it on the same y level. During game, player is higher than platform all the time with exception when he falls into mud. In such case player can get under platform. To prevent displaying shadow on platform that is above player, we need to add one additional condition. Shadow will be displayed only if found y level is under player's y.

```
            // if:
            //  1. found some tile under left edge AND
            //  2. right egde is on the same height level AND
            //  3. both are under (with higher y) player
            // then calculate shadow scale and display it
            if (minYleft < Number.MAX_VALUE && minYleft === minYright && minYleft >
this._playerPosition.y) {
                // calculate x scale for shadow. Higher the player above platform, smaller the scale
                let scale = 1 / (1 + (minYleft - this._playerPosition.y) / 500);
                this.scale.x = scale;

                // update shadow position and make it visible
                this.position.set(this._playerPosition.x, minYleft);
                this.visible = true;
            } else {
                // if conditions for displaying shadow are not met, hide it
                this.visible = false;
            }
        }
    }
}
```

To make shadow more dynamic we scale it based on distance between it and player. Higher the distance smaller the shadow. If any of conditions for displaying shadow was not met, then make it invisible.

15.2 Adding shadow

Shadow class is ready, so we can use it in game. Open Play.ts and in part with private variables add line in bold:

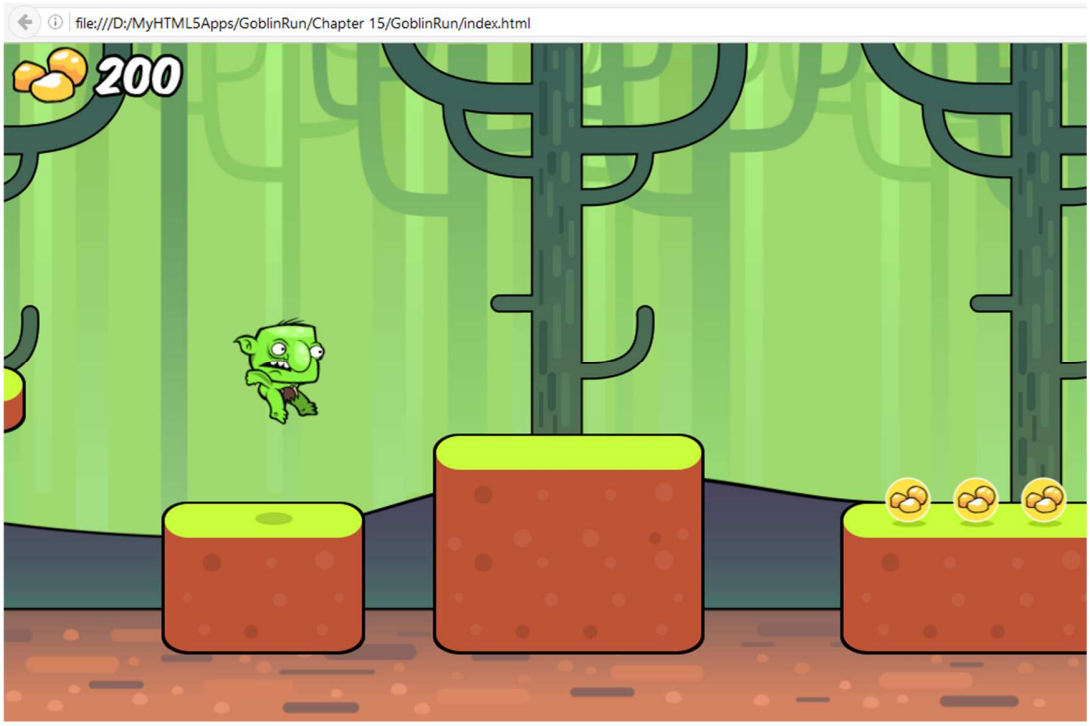
```
        // player
        private _player: Player;
        private _jumpTimer: number = 0;
        private _bonusJump: boolean = false;
        // shadow
        private _shadow: Shadow;
```

In create() method find part where player is created and just below it add new lines in bold:

```
            // set player

                   :
                   :

            // create shadow
            this._shadow = new Shadow(this.game, this._player.position, this._mainLayer.walls);
            // we want to place shadow on platforms, but under items
            let wallIndex = this._mainLayer.getChildIndex(this._mainLayer.walls);
            this._mainLayer.addChildAt(this._shadow, wallIndex + 1);
```

First we create shadow. Then we find index of walls group inside main layer. As we want shadow to be displayed over walls we add it into main layer at position higher by one.

After these changes you can build and run the game. You should now see shadow below player that disappears when he is above mud and scales down when jumping above platform.



Summary

While it may seem like small change, shadows increase atmosphere in games a lot. Our game is almost complete, we are adding smaller and smaller details. In next chapter we will visually mark top distance player reached and we will also save it in preferences.

16. Record

This will be another short chapter. We will add two things in it. First, we will save the furthest distance player reached in game preferences, so it is preserved when game is started later again. Second, we will add visual marker showing it.

16.1 Preferences

Let's start with new Preferences class. In this chapter we will use it only for storing record value, but later we can add other preferences like whether sound is on or off. Add new file directly into src folder and name it Preferences.ts. Preferences class will be singleton class which instance is available through instance() getter.

```
namespace GoblinRun {

    export class Preferences {

        private static _instance: Preferences = null;

        public record: number = 0;

        // ------------------------------------------------------------------------
        public static get instance(): Preferences {
            if (Preferences._instance === null) {
                Preferences._instance = new Preferences();
            }

            return Preferences._instance;
        }
```

Currently our only preference is record. In instance() getter we check whether there is already valid instance of Preference class and if not, it is created.

Next two methods are responsible for loading and saving. Both use localStorageSupported() method which will be defined as last.

```
        public load(): void {

            if (this.localStorageSupported()) {
                let dataString = localStorage.getItem("goblinrun_save");

                // no saved data?
                if (dataString === null || dataString === undefined) {
                    console.log("No saved settings");
                    return;

                } else {
                    console.log("loading settings: " + dataString);

                    // fill settings with data from loaded object
                    let data = JSON.parse(dataString);
```

```
                    // record
                    this.record = data.record;
                }
            }
        }
```

Data in local storage are stored as string key/value pairs. In load() method we first check whether local storage is supported. If yes, we ask browser for data saved with key “goblinrun_save”. If data are there, string will be returned, otherwise null. With help of JSON.parse() we turn this string into object and then copy properties from it into Preferences.

```
        public save(): void {

            if (this.localStorageSupported()) {

                let dataString = JSON.stringify(this);

                console.log("saving settings: " + dataString);

                localStorage.setItem("goblinrun_save", dataString);
            }
        }
```

Save() method is even simpler. Again, it first checks whether local storage is supported. If yes, Preferences object is turned into string with call to JSON.stringify() and then stored under key “goblinrun_save”.

To close this class, here is helper localStorageSupported() method:

```
        private localStorageSupported(): boolean {
            try {
                return "localStorage" in window && window["localStorage"] !== null;
            } catch (e) {
                return false;
            }
        }
    }
}
```

Now open Game.ts and immediately after calling super constructor add lines in bold:

```
            // init game
            super(Global.GAME_WIDTH, Global.GAME_HEIGHT, Phaser.AUTO, "content");

            // load saved settings
            Preferences.instance.load();
```

Calling to instance() ensures there will be Preference object with at least default values if no saved data are present.

16.2 Record class

Now to visual side of record. Record marker will be thick white dashed line across screen with number of “meters” in bottom. One meter will be one tile in our game.

Add new file Record.ts into Game folder. As the class is simple whole listing is here:

```
namespace GoblinRun {

    export class Record extends Phaser.Group {
```

```
        // ------------------------------------------------------------------------
        public constructor(game: Phaser.Game, parent: PIXI.DisplayObjectContainer, record: number) {
            super(game, parent);

            // position in pixels in world
            let x = record * 64;

            // dashed line
            let line = new Phaser.Sprite(game, x, 0, "Sprites", "Record");
            line.anchor.x = 0.5;
            this.add(line);

            // number
            let num = new Phaser.BitmapText(game, x, game.height - 15, "Font", "" + record, 40,
"center");
            num.anchor.set(0.5, 1);
            this.add(num);
        }
    }
}
```

Because record marker is composed from sprite and text, we extend Phaser.Group class. When creating object instance, we pass it current record in constructor. It is turned from meters/tiles into pixels and sprite is placed on calculated position as well as text with record value.

16.3 Adding record to main layer

To add record marker to game open MainLayer.ts and in constructor place new lines in bold just before emitters are created:

```
            // items group
                  :
                  :


            // record
            let recordDistance = Preferences.instance.record;
            if (recordDistance > 0) {
                let record = new Record(game, this, recordDistance);
                this.add(record)
            }


            // emitters
                  :
                  :
```

We first get record value from preferences and if it is greater than zero we create new instance of record and add it to main layer.

Last thing we need to do is checking for new record distance and storing it in preferences. This is done in Play state.

16.4 Checking for new record

Open Play.ts and on all places where we set this._gameOver = true, we will change it to call to new method this.gameOver(). There are three such places (changed line is in bold):

1. in update():

```
                        :
                        :

        // check if player is still on screen
        if (this._player.y > this.game.height - 104) {
            this._player.y = this.game.height - 104;
            this.gameOver();

            this._player.animateDeath();
            console.log("GAME OVER - fall");
        }

                        :
                        :
```

2. in updatePhysics():

```
                        :
                        :

        // move
        if (wallCollision && body.touching.right) {
            body.velocity.set(0, 0);
            this.gameOver();

            this._player.animateHit();
            console.log("GAME OVER - hit");
            return;
        }

                        :
                        :
```

3. and in onOverlap():

```
                        :
                        :

            <Phaser.Physics.Arcade.Body>this._player.body.velocity.set(0, 0);

            this._player.animateHit();
            console.log("GAME OVER - spike");

            this.gameOver();

                        :
                        :
```

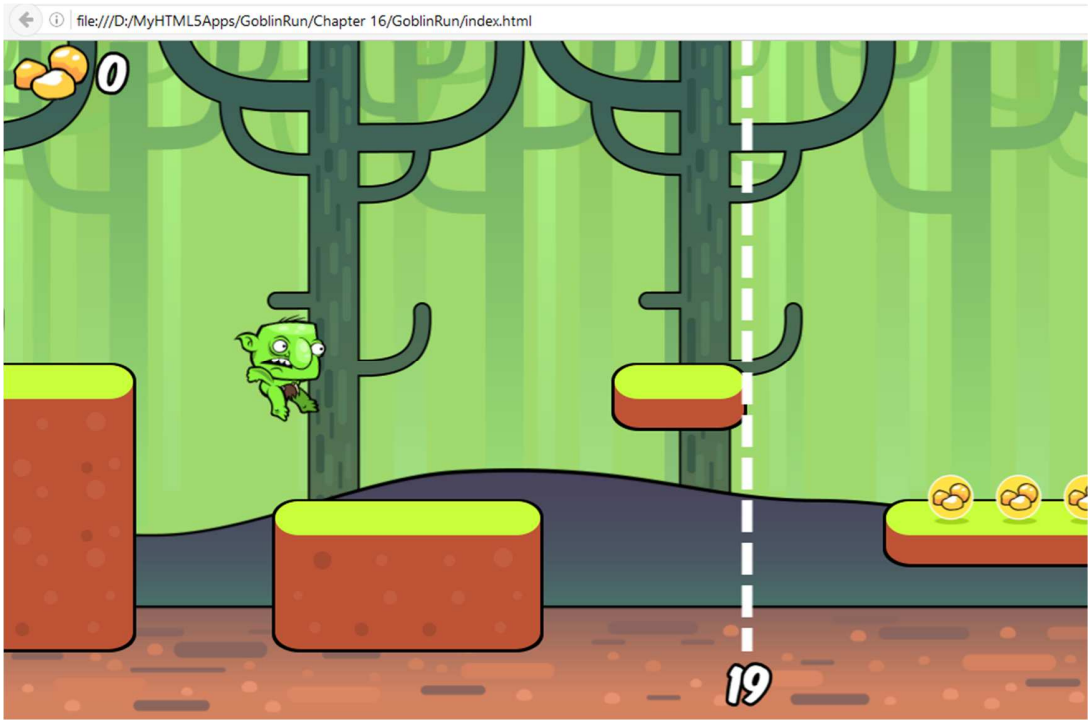
Listing for new gameOver() method is following:

```
    private gameOver(): void {
        // game over already set?
        if (this._gameOver) {
            return;
        }

        this._gameOver = true;

        // check distance for new record
        let settings = Preferences.instance;
        let newDistance = Math.floor(this._player.x / 64);
        // new record?
```

```
            if (newDistance > settings.record) {
                settings.record = newDistance;
                settings.save();
            }
        }
```

First, safety check is done whether _gameOver is already set or not. Then we calculate new distance reached with converting player's x position to meters/tiles. If new distance is higher than last one stored in preferences, we save it.

If you build and run game, then you should see maximum reached distance marked like this:



Summary

This chapter closed adding of new features to gameplay. Game runs well, but so far it is without any sound. In next chapter we will add sound and music, and be sure it will increase user experience a lot.

17. Sounds and music

In this chapter we will add sounds and music into game. Audio is the crappiest thing in whole HTML5 game development. Phaser hides a lot of details for you, but it is almost sure you will encounter some problems when working on your game. In text I will describe some of them.

17.1 Audiosprite

For sound effects we will use audiosprite. Audiosprite is single sound file combined from individual sounds with silence gaps between them. Along with it you need metadata for markers that say where sound starts, where it ends and whether it loops or not. Using audiosprite can have some benefits like overcoming limitations of some browsers, especially older ones, or faster loading. Unfortunately, if browser does not support Web Audio API (like Internet Explorer) only one instance of sound can be played at time. This limitation applies also to the whole audiosprite. Solution I used in some of my games is to have two sets of sound assets, one as single audiosprite and second as individual sound effect files. When loading assets, I check browser and if it is IE individual sound files are loaded. This solution requires additional class that has its own play() method and resolves whether to call play() on individual sound or whether to play marker within audiosprite. To keep thing simple, we will use only audiosprite here.

In files accompanying this book you can find audiosprite already prepared for use in folder Resources/_RES/Sound. There are three files, two of them, Sfx.ogg and Sfx.m4a, are audiosprites itself and Sfx.json contains metadata. Beside this in the same folder you can find all individual sound effects and also music files which will be discussed later.

Copy files Sfx.ogg and Sfx.m4a into assets folder.

If you want to create audiosprite by yourself, you will need npm (Node Package Manager) installed. Tool by Tõnis Tiigi for creating audiosprites along with installation instructions can be found here: https://github.com/tonistiigi/audiosprite. You will also need FFmpeg codecs installed (you can download it here: https://ffmpeg.zeranoe.com/builds/). If you are Windows user, you need path to bin folder of codecs added to your environment PATH variable or you can set it temporarily in batch file. Such file (_run.bat) is already created in resources folder and has this content:

```
Setlocal

set "PATH=%path%;d:\Utils\FFMPEG\bin"

audiosprite --output Sfx --path assets --export ogg,m4a --format jukebox --bitrate 64 --samplerate
44100 --channels 1 end.wav bonus_jump.wav gold.wav hit.wav jump.wav land.wav mud_fall.wav select.wav
```

In case you want to use it do not forget to change path to FFmpeg/bin folder on second line according to path on your disc.

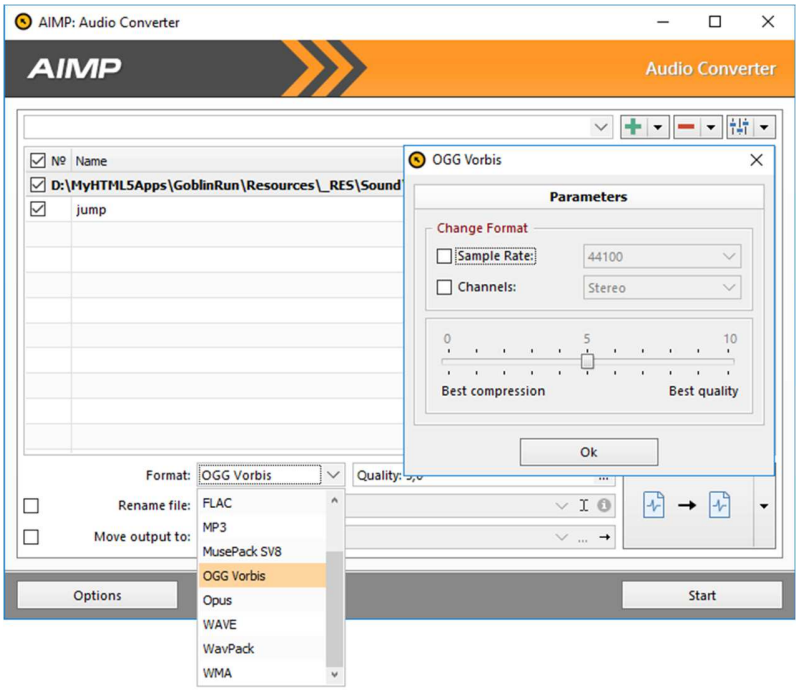
17.2 Music

For our game we will use music composed by sawsquarenoise, which is licensed under Creative Commons Attribution 4.0 International license (CC BY 4.0 - https://creativecommons.org/licenses/by/4.0/). It was downloaded from http://freemusicarchive.org/music/sawsquarenoise/ and we will use two tracks from album RottenMage SpaceJacked OST:

- sawsquarenoise_-_05_-_OST_05_Go_Go_Go.mp3 for game music,
- sawsquarenoise_-_07_-_OST_07_Lets_Rest.mp3 for menu music.

Files are stored in the same resources folder as other sounds. Both of them are stored in .ogg and .m4a formats. Copy them all (files MusicGame.ogg, MusicGame.m4a, MusicMenu.ogg and MusicMenu.m4a) into assets folder.

For converting from format to format, AIMP Audio Converter proved to be useful. You can get it here: http://www.aimp.ru/index.php?do=download. With simple UI you have control over all parameters of target format:



17.3 Sounds class

When loading audiosprite you can proceed in the similar way like when loading texture atlas – pass file with sound data and metadata file as second parameter. There is also option to not load metadata from url, but pass it directly as object. We will choose the second way. If you already looked into Sfx.json you noticed that every sound has three properties:

- start in seconds,
- end in seconds,
- loop

If we passed this file when loading as url, we could do no changes into it. If we pass it inside code as object, we can tweak it first if needed. For example, on iOS devices I had problem with short sounds. It was completely skipped. As a solution I found I can change sound start time by 0.1 sec to start earlier. We will not do this here to keep things simple.

To keep code transparent let's create new class Sounds in file Sounds.ts in src folder. Here we will copy json object created with audiosprite tool. It will also store reference to created audiosprite object as well as to two audio objects for music tracks.

First, we will start with few interfaces to describe structure of audiosprite metadata. These are optional and in fact we do not need them for our game. But I will show it here along with some code later as it can be used when tweaking audiosprite properties. It will allow IDE to give you intellisense help and it will check audiosprite metadata structure for type errors during compile time.

```
namespace GoblinRun {

    export interface IAudioSprite {
        start: number;
        end: number;
        loop: boolean;
    }
```

We will start from the most nested object. Every audiosprite has three properties: start, end and loop.

```
    export interface ISpriteMap {
        [n: string]: IAudioSprite;
    }
```

From the basic audiosprites dictionary is build, that has pairs key-IAudioSprite. To define interface for it, we use Typescirpt's indexer syntax where indices (keys) are strings.

```
    export interface IAudioJSON {
        resources: string[];
        spritemap: ISpriteMap;
    }
```

Finally, we define IAudioJSON interface, which contains that dictionary and string array with assets file names.

Now we will start Sounds class itself:

```
    export class Sounds {

        // sfx
        public static sfx: Phaser.AudioSprite;
        // musix
        public static musicGame: Phaser.Sound;
        public static musicMenu: Phaser.Sound;
```

In these variables we will keep references to created audio objects. We will create it only once in Preload state and use it across game.

Next we create AUDIO_JSON static variable of type IAudioJSON and initialize it with metadata from Sfx.json. Just copy and paste it:

```
        // definition of markers for sfx
        public static AUDIO_JSON: IAudioJSON = {
```

```
            "resources": [
                "assets\\Sfx.ogg",
                "assets\\Sfx.m4a"
            ],
            "spritemap": {
                "end": {
                    "start": 0,
                    "end": 1.2299319727891156,
                    "loop": false
                },
                "bonus_jump": {
                    "start": 3,
                    "end": 3.225736961451247,
                    "loop": false
                },
                "gold": {
                    "start": 5,
                    "end": 5.395873015873016,
                    "loop": false
                },
                "hit": {
                    "start": 7,
                    "end": 7.09859410430839,
                    "loop": false
                },
                "jump": {
                    "start": 9,
                    "end": 9.184943310657596,
                    "loop": false
                },
                "land": {
                    "start": 11,
                    "end": 11.123083900226757,
                    "loop": false
                },
                "mud_fall": {
                    "start": 13,
                    "end": 13.482630385487528,
                    "loop": false
                },
                "select": {
                    "start": 15,
                    "end": 15.052879818594104,
                    "loop": false
                }
            }
        };
    }
}
```

17.4 Loading

We will load sounds in Preload state along with all other assets. Add new lines in bold:

```
        public preload() {

                        :
                        :

            // sound fx
            // iterate through all audiosprites
            //for (let property in Sounds.AUDIO_JSON.spritemap) {
            //    let audioSprite = Sounds.AUDIO_JSON.spritemap[property];
```

```
            //    console.log("name: " + property + ", value: " + JSON.stringify(audioSprite));
            //}
            this.load.audiosprite("Sfx", Sounds.AUDIO_JSON.resources, null, Sounds.AUDIO_JSON);

            // music
            this.load.audio("MusicGame", ["assets/MusicGame.ogg", "assets/MusicGame.m4a"]);
            this.load.audio("MusicMenu", ["assets/MusicMenu.ogg", "assets/MusicMenu.m4a"]);
        }
```

Commented code is here as hint if you need to adjust audiosprite values before loading it. It iterates through all keys in AUDIO_JSON's spritemap object and prints its properties. Here, defined interfaces come handy because audioSprite's variable type will be inferred to IAudioSprite.

In uncommented code we simply ask Phaser's loader to add one audiosprite and two music tracks into load queue. As different browsers support different sound file formats, we pass all possible asset urls as array. Phaser will use the first supported in currently running browser.

In create() method we will create all audio objects from loaded data:

```
        public create() {
            // sound
            Sounds.sfx = this.add.audioSprite("Sfx");

            // music
            Sounds.musicGame = this.add.audio("MusicGame");
            Sounds.musicGame.loop = true;
            Sounds.musicMenu = this.add.audio("MusicMenu");
            Sounds.musicMenu.loop = true;
        }
```

References to created objects are stored in Sounds class. For both music tracks we set loop property to true to make it play infinitely.

Last change is in update() method. When sound assets are loaded, it may still take some time to decode it, especially if it is in compressed format. Before we leave Preload state and go into Play state we wait till all sound assets are decoded and ready to be used.

```
        public update() {
            // run only once
            if (this._ready === false &&
                this.cache.isSoundDecoded("Sfx") &&
                this.cache.isSoundDecoded("MusicGame") &&
                this.cache.isSoundDecoded("MusicMenu")) {

                this._ready = true;

                this.game.state.start("Play");
            }
        }
```

17.5 Playing

Now when everything is ready we can play sounds where needed. Open Play.ts file and in the end of create() method start music with this line:

```
            // start music
            Sounds.musicGame.play();
```

We will stop music when player lands in mud. Locate update() method and stop music before we set death animation:

```
                        :
                        :

        // check if player is still on screen
        if (this._player.y > this.game.height - 104) {
            this._player.y = this.game.height - 104;
            this.gameOver();

            // stop music
            Sounds.musicGame.stop();

            this._player.animateDeath();
            console.log("GAME OVER - fall");
        }

                        :
                        :
```

I onOverlap() callback, we will add sound effects when bonus jump and gold are picked.

```
    private onOverlap(player: Phaser.Sprite, item: Item): void {

        if (item.itemType === eItemType.SPIKE) {
                        :
                        :
        } else if (item.itemType === eItemType.BONUS_JUMP) {
                        :
                        :

            Sounds.sfx.play("bonus_jump");

        } else if (item.itemType === eItemType.GOLD) {
            this._mainLayer.removeItem(item);

            Sounds.sfx.play("gold");

                        :
                        :
        }
    }
```

Rest of the sound effects will be played from Player class. Open Player.ts and follow changes below.

```
    public animateJump(): void {
        this._spriterGroup.playAnimationByName("jump");
        Sounds.sfx.play("jump");
    }
```

```
    public animateHit(): void {
        this._spriterGroup.playAnimationByName("hit");
        Sounds.sfx.play("hit");
    }
```

```
    public updateAnim(standing: boolean, velY: number, gameOver: boolean): void {

        if (!gameOver) {
            if (standing) {
```

```
                if (this._spriterGroup.currentAnimationName !== "run") {
                    this._spriterGroup.playAnimationByName("run");
                    Sounds.sfx.play("land");
                }
            } else if (velY > 0) {
                                :
                                :
            }
        }
                    :
                    :
    }
```

Only change worth of commenting is in animateDeath():

```
    public animateDeath(): void {
        let body = <Phaser.Physics.Arcade.Body>this.body;
        body.enable = false;

        this._spriterGroup.playAnimationByName("fall_water");

        // play splash sound
        Sounds.sfx.play("mud_fall");
        // play end sound with some delay
        this.game.time.events.add(1000, function () {
            Sounds.sfx.play("end");
        }, this);

        // emit mud particles
        this._mudEmitter.explode(0, 20);
    }
```

When player falls into mud splash sound is played. We add new event delayed one second and when this time passes we play some game over sound (named “end” in our case).

Summary

With sounds added our game feels almost complete but it is still displayed only in part of available space leaving rest of it black. In next chapter we will add scaling to fill whole available space.

18. Scaling and orientation

In this chapter we will focus on scaling and orientation changes. We want to resize game, so it fills all available screen space. In case of mobile devices, we want it react to orientation change.

There are many strategies how to scale game and its choice depends on game type. In some games you can let player see more of the surroundings if display space is larger. In others you may want to keep view range as fixed as possible and scale everything instead to match it.

Our default game size is 1024 x 640 and game runs in landscape. We have nothing to display below or above, so we will scale it vertically to fill whole available height. To prevent image distortion, we will scale it horizontally with the same value. But to match available display width, player may see more or less of incoming level. If our default aspect ratio is 1.6 (1024 / 640), ratio used during gameplay may be different based on display dimensions. Main question is whether it will affect gameplay if player can see a few tiles more or less in front of him? Answer is no.

18.1 Adding meta tags

Until now, our index.html was very simple. Now, we will add lot of meta tags to it as well as do other small changes. Open it and do changes in bold:

```
<!DOCTYPE html>

<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>Goblin Run</title>
    <link rel="stylesheet" href="css/app.css" type="text/css" />

    <meta http-equiv="X-UA-Compatible" content="chrome=1,IE=Edge">

    <meta name="viewport" content="width=device-width initial-scale=1 maximum-scale=1 user-
scalable=0 minimal-ui" />
    <meta name="apple-mobile-web-app-capable" content="yes" />
    <meta name="apple-mobile-web-app-status-bar-style" content="black" />
    <meta name="apple-mobile-web-app-title" content="Goblin Run">

    <meta name="format-detection" content="telephone=no">
    <meta name="HandheldFriendly" content="true" />
    <meta name="robots" content="noindex,nofollow" />

    <meta name="full-screen" content="yes" />
    <meta name="screen-orientation" content="landscape" />
    <meta name="x5-fullscreen" content="true" />
    <meta name="360-fullscreen" content="true" />

    <link rel="shortcut icon" href="assets/favicon.ico" type="image/x-icon" />

    <script src="js/phaser.min.js"></script>
    <script src="js/spriter.min.js"></script>
```

```
    <script src="js/goblinrun.js"></script>
</head>
<body>
    <div id="content"></div>
    <div id="orientation"></div>
</body>
</html>
```

Most of the changes take place in header. First, we add lot of meta tags. These tags are used to add additional information about html document. In general, we can say that meta tags we added here are mainly useful for mobile browsers to instruct them how to work with page. What we want is to fill whole screen, hide other UI elements if possible, prevent zooming, etc.

Some of the tags work only for specific browser, some of them are used across wider range of browsers. For example, x5-fullscreen is specific to QQ Mobile Browser. In fact, I found it very difficult to get information about some of them. One of good resources is here: https://github.com/joshbuchea/HEAD, while it still does not cover all.

Beside meta tags, we added also this line in head:

```
    <link rel="shortcut icon" href="assets/favicon.ico" type="image/x-icon" />
```

With it we can set favicon for our game. This icon is displayed at several places like page tab in browser, in bookmarks, etc.

There is a lot of on-line tools for generating favicons from image, but in many of them I had problem with transparency. One, working well for me, is at this link: http://www.prodraw.net/favicon/.

In Chapter 18 folder in files accompanying this book you can find favicon.ico image in assets folder. You can also find it in Resources/_RES/Other. Copy it into assets folder.

In body part of page, we added new div tag with id orientation. This adds layer over our content. Most of the time it is not displayed, but when mobile device rotates into incorrect orientation, then it becomes visible. To define what to display and how, we adjust app.css file in css folder. Add following lines into it:

```
#orientation {
    margin: 0 auto;
    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    background-image: url(../assets/orientation.jpg);
    background-repeat: no-repeat;
    background-position: center;
    background-color: rgb(0, 0, 0);
    z-index: 999;
    display: none;
}
```

These settings say we want image orientation.jpg displayed in center of screen on black background. It shall cover whole screen and with high z-index we determine it will be on top. Initially, its display parameter is set to none making it hidden. We will control visibility of this element from within the game.

Image orientation.jpg can be also found in Resources/_RES/Other. Copy also this file into assets folder.

18.2 Boot state

Key changes in this chapter will take place in boot state. Open Boot.ts in States folder and in top of it add two private variables:

```
        private _userScale: Phaser.Point = new Phaser.Point(1, 1);
        private _gameDims: Phaser.Point = new Phaser.Point();
```

_userScale holds scale values we use for both x and y dimensions. _gameDims is size of game before it is scaled to match available space. Its y value will be always 640, default height of our game. Only x will change depending of width of view.

We will start with init() method. It is one of reserved method names Phaser recognizes in state. If Phaser finds init() method in state it is called as very first one.

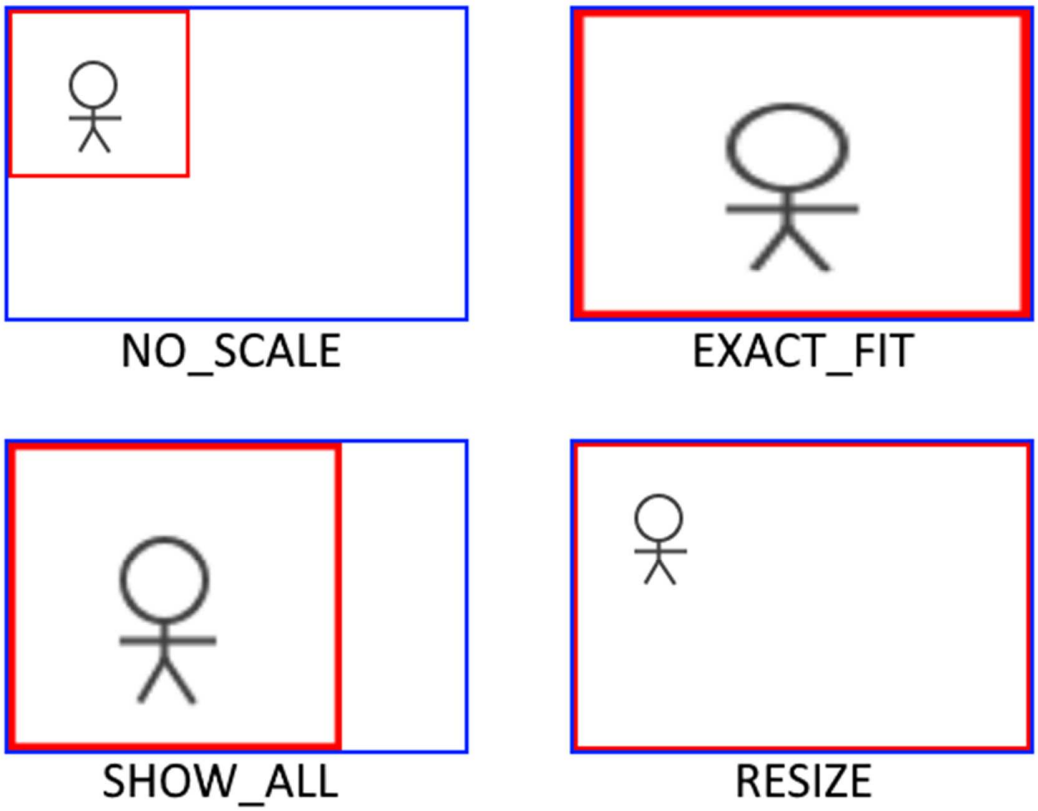
```
        public init(): void {

            this.calcGameDims();

            this.scale.scaleMode = Phaser.ScaleManager.USER_SCALE;
            this.scale.setUserScale(this._userScale.x, this._userScale.y);
            this.scale.pageAlignHorizontally = true;
            this.scale.pageAlignVertically = true;
            this.scale.setResizeCallback(this.gameResized, this);

            if (!this.game.device.desktop) {
                this.scale.forceOrientation(true, false);
                this.scale.onOrientationChange.add(this.orientationChange, this);
            }
        }
```

As a first thing, we call calcGameDims() method, which will be introduced soon. Then we set scale mode. Phaser has several scale modes. To understand what they do we have to distinguish between game size and display size. Game size are game dimensions we passed to Phaser when constructing Phaser.Game object or changed later with call to ScaleManager.setGameSize(). All rendering is done into canvas of this size. Display size is either size of screen or size of parent element in html document. After game is rendered it is displayed on screen. It may be stretched or not to match display size depending on scale mode. Modes are:

- NO_SCALE – this is default mode we used so far. Game is not scaled, pixels on screen are displayed 1:1. If game size is smaller than display size, blank space is left around,
- EXACT_FIT – this stretches game to fill whole available display space ignoring aspect ratio. If aspect of display area is different to aspect of game you may get ugly stretching. This mode is usable if available space is not given by screen but some parent element that has the same aspect as game but different size. In fact, game size is not touched. What is scaled is canvas game is rendered to before it is presented to user,
- SHOW_ALL – this mode scales game but preserves aspect ratio. If game aspect is different from display area aspect, you will get blank space either vertically or horizontally. Again game size is not changed,
- RESIZE – in this mode game size dimensions are changed to match size of available display space. Game is not stretched, but whole game is enlarged,

- USER_SCALE – this mode allows you to define how much to scale game vertically and horizontally. This does not change game size, but in combination with changing game size through ScaleManager.setGameSize() method it is most flexible solution.

Look on image bellow to see how modes work. USER_SCALE is not shown there as its result depends on user scale settings passed to ScaleManager.setUserScale(). Blue box is display area and red box is game after scaling mode is applied.



In our game we will use USER_SCALE mode. We want to stretch game vertically and horizontally. We want to use combination of stretching and changing game width to preserve aspect ratio. SHOW_ALL would also work. If we changed game width to match aspect of display area, we could then leave it on Phaser to scale game with SHOW_ALL mode and it would fill whole display area. But for some reason, I usually want to have maximum control over game scaling.

Back to code. After setting scale mode and user scale parameters we instruct scale manager to center game horizontally and vertically. It is not needed in our case because we are scaling game to fill whole available area, but if, for example, SHOW_ALL mode is used and there is blank space on right, it will be equally divided to left and right.

Then we set resize callback to gameResized() method. Whenever game is resized this method gets called. Inside of it we will recalculate user scale parameters as well as game dimensions.

In case our game runs on mobile device, we force orientation to landscape and set another callback to handle orientation changes.

Now we can define calcGameDims() method:

```
        private calcGameDims(): void {
            let winWidth = window.innerWidth;
            let winHeight = window.innerHeight;

            // calculate scale y. Size after scale is truncated (scaleY * game height).
            // Add small amount to height so scale is bigger for very small amount and we do not
loose
            // 1px line because of number precision
            let scaleY = (winHeight + 0.01) / Global.GAME_HEIGHT;

            // get game width with dividing window width with scale y (we want scale y
            // to be equal to scale x to aviod stretching). Then adjust scale x in the same way as
scale y
            let gameWidth = Math.round(winWidth / (winHeight / Global.GAME_HEIGHT));
            let scaleX = (winWidth + 0.01) / gameWidth;

            // save new values
            this._userScale.set(scaleY, scaleX);
            this._gameDims.set(gameWidth, Global.GAME_HEIGHT);
        }
```

First, we calculate scale for y dimension. Inside Phaser, when calculating final size, this value is multiplied with game height and then truncated. Small imprecisions can lead to loosing 1px line if calculated value before truncation is something like 199.99999. To prevent this, we add very small amount to available height.

Then we calculate new width of game. We want to preserve aspect ratio, so scale for x will be the same as for y. We divide available width with y scale and round to nearest integer. From this we recalculate scale for x dimension with the same trick we used when calculating y scale. In the end y scale can differ from x scale, but only for very small amount so distortion of image is negligible.

For example, on my monitor with 1920 x 1200 resolution available space in Firefox browser is 1920 x 1091. Default game size 1024 x 640 is recalculated to 1126 x 640. Scale x is 1.7051599 and scale y is 1.7047031. When Phaser calculates size of scaled game in pixels it gets 1.7051599 * 1126 = 1920.010047 for width and 1.7047031 * 640 = 1091,009984 for height. It is truncated to 1920 x 1091 which is size of available space. As game width was recalculated from 1024 to 1126, player can see about 10% more horizontally.

Now, we can define callbacks. First one is gameResized():

```
        public gameResized(scaleManger: Phaser.ScaleManager, bounds: Phaser.Rectangle): void {

            if (!scaleManger.incorrectOrientation) {
                let oldScaleX = this._userScale.x;
                let oldScaleY = this._userScale.y;

                // recalculate game dims
                this.calcGameDims();
                let dims = this._gameDims;
                let scale = this._userScale;

                // any change in game size or in scale?
                if (dims.x !== this.game.width || dims.y !== this.game.height ||
```

```
                    Math.abs(scale.x - oldScaleX) > 0.001 || Math.abs(scale.y - oldScaleY) > 0.001)
{
                    // set new game size and new scale parameters
                    this.scale.setGameSize(dims.x, dims.y);
                    this.scale.setUserScale(scale.x, scale.y);

                    // has current state onResize method? If yes call it.
                    let currentState: Phaser.State = this.game.state.getCurrentState();
                    if (typeof (<any>currentState).onResize === "function") {
                        (<any>currentState).onResize(dims.x, dims.y);
                    }
                }
            }
        }
```

If game orientation is incorrect, we do not care about resizing the game. It will be covered with image instructing player to change orientation. If orientation is correct, we first recalculate game dimensions and scale. Then we check if there is any change and if yes, we send new values to engine.

As last thing we check if currently active game state has method onResize(). If yes, we call it with new game dimensions. This allows states that implement onResize() to react on game size changes. If state does not have this callback method implemented nothing happens.

orientationChange() callback will call different method for entering and leaving incorrect orientation. One method makes orientation change image visible while other hides it. Both methods check current state for onPause() and onResume() callbacks. This is easy way how to give states opportunity to react. If your game should pause and resume in any state when orientation changes you can do it right here, just set this.game.paused = true/false.

```
        public orientationChange(scaleManger: Phaser.ScaleManager, previousOrientation: string,
previouslyIncorrect: boolean): void {

            if (scaleManger.isLandscape) {
                this.leaveIncorrectOrientation();
            } else {
                this.enterIncorrectOrientation();
            }
        }

        // ---------------------------------------------------------------------------
        public enterIncorrectOrientation(): void {
            // show change orientation image
            document.getElementById("orientation").style.display = "block";

            // if current state has onPause method then call it.
            let currentState: Phaser.State = this.game.state.getCurrentState();
            if (typeof (<any>currentState).onPause === "function") {
                (<any>currentState).onPause();
            }
        }

        // ---------------------------------------------------------------------------
        public leaveIncorrectOrientation(): void {
            // hide change orientation image
            document.getElementById("orientation").style.display = "none";

            // if current state has onResume method then call it.
            let currentState: Phaser.State = this.game.state.getCurrentState();
            if (typeof (<any>currentState).onResume === "function") {
                (<any>currentState).onResume();
```

```
            }
        }
```

18.3 Handling changes

Game now sends messages about size and orientation changes. Let's implement callbacks in Play state to process them. Open Play.ts and add these simple methods in the end of it:

```
        public onResize(width: number, height: number): void {
            this._bg.resize();
        }

        // --------------------------------------------------------------------------
        public onPause(): void {
            this.game.paused = true;
        }

        // --------------------------------------------------------------------------
        public onResume(): void {
            this.game.paused = false;
        }
```

When game is resized we need to do only one thing – resize tilesprites used for background layers. So we have to create new resize() method in Background class. Open it and in the end add this method:

```
        public resize(): void {
            let newWidth = this.game.width;

            this._treesBg.width = newWidth;
            this._hill.width = newWidth;
            this._mud.width = newWidth;
        }
```

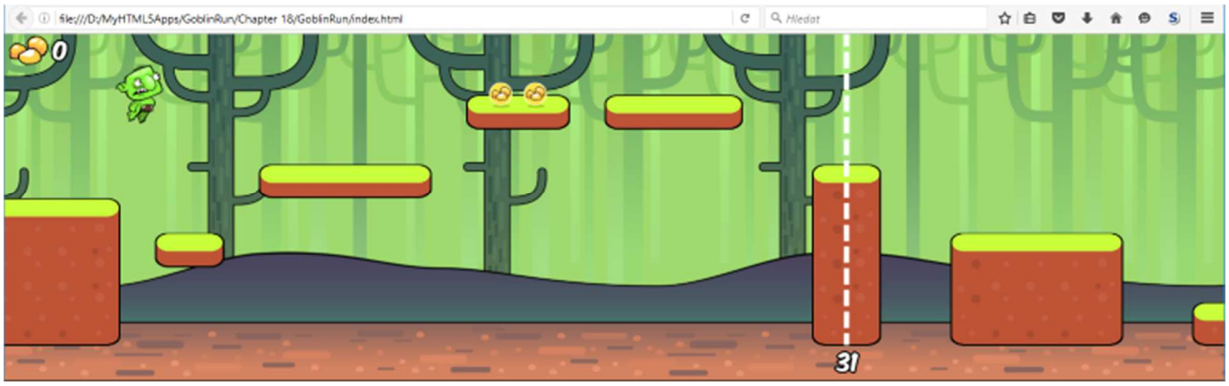
All we do is we set new width for all tilesprites.

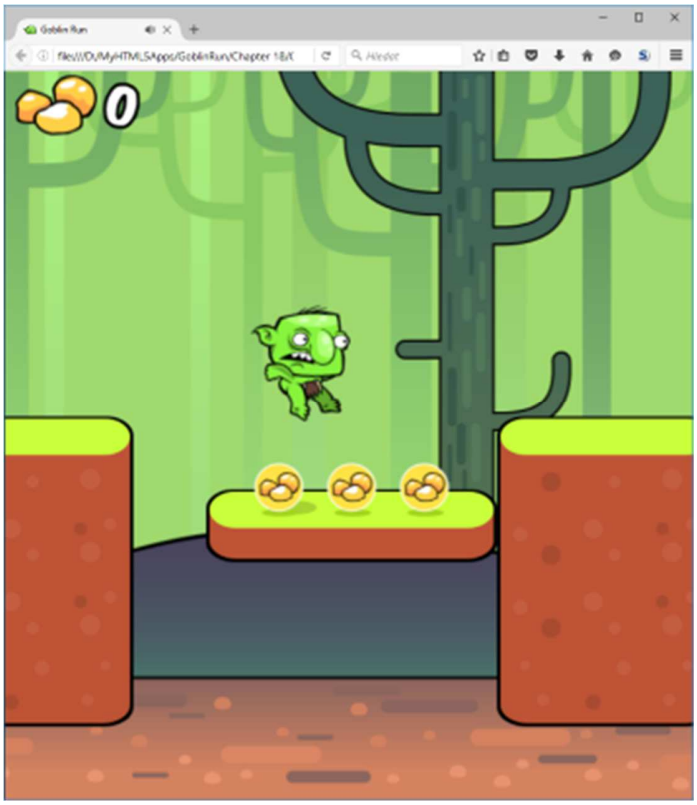
Our game is simple, but in onResize() callback you can for example reposition your UI elements, etc.

18.4 Running game

Compile and run game. You can resize your browser window to really strange dimensions and game should keep working nicely.

Also test changes of orientation on mobile devices. When you turn device into incorrect position it should pause and resume when you turn it back.





Summary

In this chapter we implemented very flexible scaling mechanism. It is best to think of scaling from early game development stages. Key for choosing scaling strategy is type of game and its impact on gameplay.

In next chapter we will build some simple menu screen.

19. Menu screen

As our game is complete, we will now work on some simple menu screen. Word *simple* can be misleading because we will introduce a few interesting concepts during it. This is how final screen will look like:

Let's split it into parts:

- trees in background - slowly moving,
- ground in foreground – little curved (+50 pixels down in the middle), will be recreated dynamically as we need to react to screen size changes,
- running goblin – will run along curved ground surface. We will use quadratic Bézier curve to calculate its y position. Goblin will run behind title,
- title – we will add some tweens to its angle and scale. For tween we will use custom tween function,
- play button – will use another custom tween function,
- sound button – will toggle sound output and current state will be saved in user's preferences.

19.1 Small adjustments

First, we will do some small changes into current code so we can later focus only on new menu state.

Open file Play.ts and in the end of create() method, just before we start music, reset variables (bold code):

```
                :
                :

        // reset variables
        this._gameOver = false;
        this._score = 0;
        this._bonusJump = false;
        this._justDown = this._justUp = false;

        // start music
        Sounds.musicGame.play();
    }
```

We will traverse from menu state to play state and back and we need these variables clean and set to initial values whenever play state starts (not only first time).

In the end of gameOver() method add these lines:

```
    private gameOver(): void {

                :
                :

        // return to menu
        this.time.events.add(3000, function () {
            this.game.state.start("Menu");
        }, this);
    }
```

Three seconds after player died, he will be returned back to main menu.

In Game.ts add new Menu state:

```
        // states
        this.state.add("Boot", Boot);
        this.state.add("Preload", Preload);
        this.state.add("Play", Play);
        this.state.add("Menu", Menu);
```

Next, open Preload.ts and in the very end of update() method change state to start from “Play” to “Menu”:

```
    public update() {
        // run only once
        if (this._ready === false &&
            this.cache.isSoundDecoded("Sfx") &&
            this.cache.isSoundDecoded("MusicGame") &&
            this.cache.isSoundDecoded("MusicMenu")) {

            this._ready = true;

            this.game.state.start("Menu");
        }
    }
```

Next time when we run game, it will start with new Menu state.

Last small change we need to do is to prepare preferences for storing and retrieving sound setting. Open Preferences.ts and add new public variable sound on top of it:

```
    public record: number = 0;
    public sound: boolean = true;
```

Then head to load() method and add bolded lines:

```
                        :
                        :

                // record
                this.record = data.record;
                // sound
                this.sound = data.sound;
            }
```

Now we are ready for working only inside new menu state.

19.2 Menu state

Code for menu is pretty long so we will split it into many parts and explain all new concepts along the way. First just create new file Menu.ts in States folder and put a few constants and private variables into it:

```
namespace GoblinRun {

    export class Menu extends Phaser.State {

        private static GROUND_HEIGHT = 250;
        private static GROUND_CP_Y = 50;

        private static GOBLIN_Y = Menu.GROUND_CP_Y - 10;
        private static GOBLIN_SPEED = 200;

        private static BG_SPEED_X = 10;

        private static SOUND_BUTTON_OFFSET = 50;


        private _ground: Phaser.Sprite;
        private _treesBg: Phaser.TileSprite;
        private _sound: Phaser.Button;

        private _goblin: Spriter.SpriterGroup;
        private _runDirection: number;
```

Names of constants and variables imply to which feature of menu screen it is related. We will explain it when particular feature is implemented.

In create() method, which is as our entry point into state, we will assemble whole menu screen from individual parts. Most of them are more complex, so we will create separate methods for them. These methods will be introduced later along state development and you can now comment calls to them out to be able to test game after each new feature is added – just do not forget to uncomment it later again:

```
        public create() {

            // light green color for background
            this.stage.backgroundColor = 0xA0DA6F;
```

Pale green is our base color for menu screen.

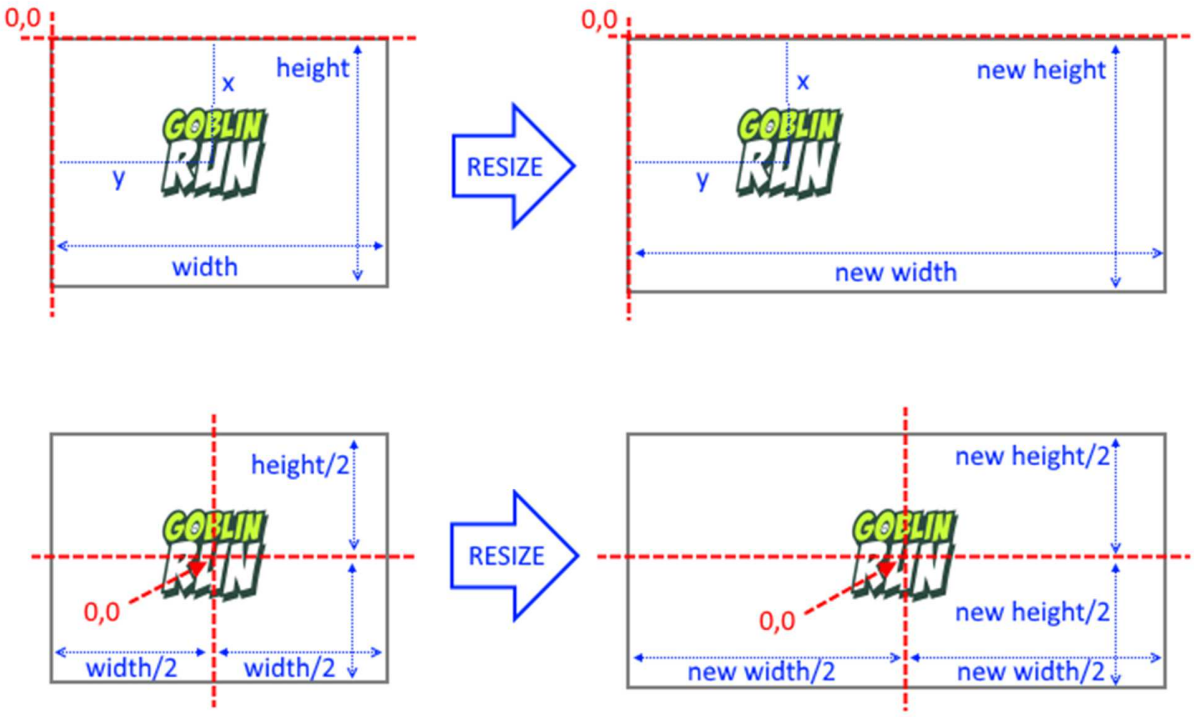
```
            // setup camera and world bounds
            this.setView(this.game.width, this.game.height);
```

This call is very important, so peek few lines below to see what this method does. Method itself is very short – it just sets bounds of world and camera focus, but its implications are serious. So far our coordinates origin (point 0,0) was in top left corner which is default for all Phaser games. When game

window was resized we did some calculations about new game size and scale parameters, but origin stayed in top left corner. It was perfectly ok for runner game. But menu screen is different. Remember I said in previous chapter that scaling strategy should come from game type. We can consider setting of coordinates origin as part of scaling strategy. Menu screen is centered around screen center. When window is resized, we still want title stay in the middle. I found it advantageous to change coordinates origin and put point 0,0 into center of screen and position as many objects as possible relative to it.

Look on following image. First row shows happens when you resize game with origin in top left corner. If your title image was centered previously, after increasing window width it stays on its original x,y position and is not centered anymore.

Second row shows what happen is origin is centered. Title image also stays on its original position, but as origin is kept in the center of new window it stays centered. All objects positioned relative to origin will stay at their positions after resize. Objects positioned against, let's say, top right corner need to be repositioned after window resize. With default origin in top left corner only those objects positioned against it do not need to be repositioned.

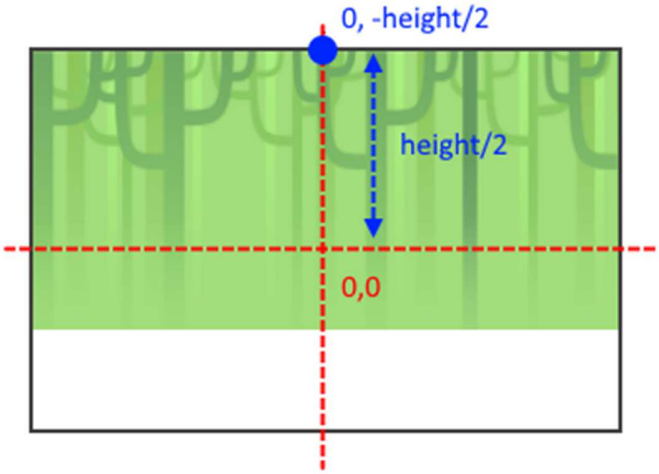


Now, back to create() method code:

```
            // trees bg
            let treesHeight = this.cache.getImage("TreesBg").height;
            this._treesBg = this.add.tileSprite(0, -this.game.height / 2, this.game.width,
treesHeight, "TreesBg");
            this._treesBg.anchor.x = 0.5;
```

Trees in background are our first object on menu screen. We will use Phaser.TileSprite for it. Position is set to 0, -this.game.height /2, so it is using our new coordinate settings. We need to set x anchor to 0.5 to

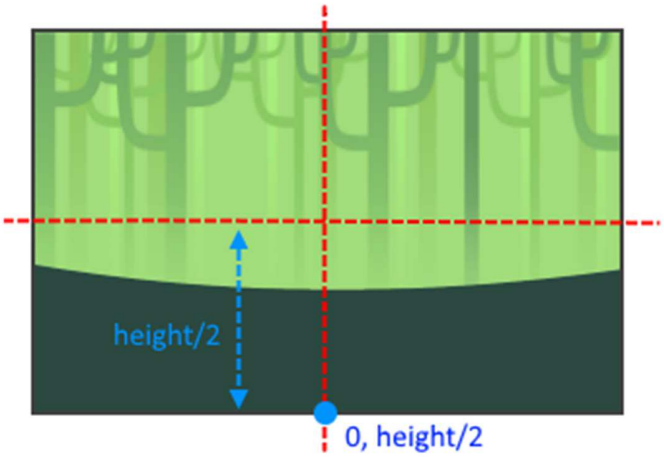
center it correctly around vertical screen center. Width is set to game width to cover screen from left to right:



Next object we will create is ground.

```
            // ground sprite
            this._ground = this.game.add.sprite(0, this.game.height / 2, this.generateGround());
            this._ground.anchor.set(0.5, 1);
```

Ground is single sprite. We do not have it in atlas or any loaded texture. Instead we are creating it dynamically in generateGround() method that will be introduced soon. This method renders it into texture that is returned and used for sprite. Ground is anchored in middle bottom of screen:



Rest of create() method are calls to individual methods that will create more complex objects. We will describe them soon.

```
            // objects on menu screen
            this.createGoblin();
            this.createTitle();
            this.createStartButton();
```

```
            // sound button
            this.createSoundButton();

            // set sound and start music
            this.sound.mute = !Preferences.instance.sound;
            Sounds.musicMenu.play();
        }
```

In the end of create() method we set engine mute property to value loaded in preferences. After that we start playing of menu music. If sound is muted, nothing will be heard.

setView() method was already described when discussing placing of origin.

```
        private setView(width: number, height: number): void {

            // set bounds
            this.world.setBounds(-width / 2, -height / 2, width / 2, height / 2);

            // focus on game center
            this.camera.focusOnXY(0, 0);
        }
```

Our world for menu state is single screen and we set its bounds so point 0, 0 is in the middle of it.

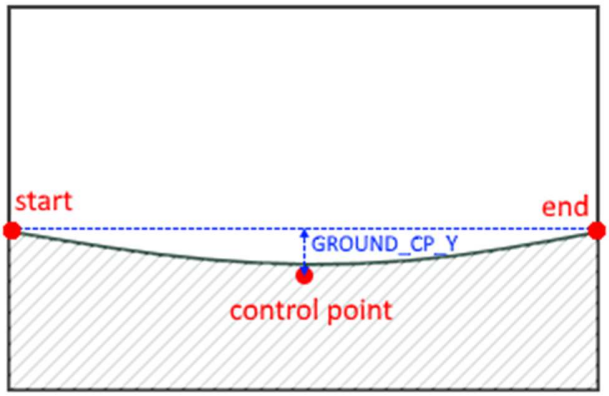
19.3 Menu objects

generateGround() method produces texture used for ground sprite.

```
        private generateGround(): PIXI.RenderTexture {

            let g = new Phaser.Graphics(this.game);

            // fill color
            g.beginFill(0x2B4940);

            // draw a shape
            g.moveTo(0, 0);
            g.quadraticCurveTo(this.game.width / 2, Menu.GROUND_CP_Y, this.game.width, 0);
            g.lineTo(this.game.width, Menu.GROUND_HEIGHT);
            g.lineTo(0, Menu.GROUND_HEIGHT);
            g.endFill();

            // generate texture from ggraphics
            let texture = g.generateTexture();

            // we do not need graphics anymore
            g.destroy();

            return texture;
        }
```

We start with Phaser.Graphics object that allows us to draw primitives (lines, circles, rectangles, polygons, curves, …). Drawing of complex shapes can be expensive. If renderer used is Canvas, HTML operations for drawing on canvas are used. If renderer is WebGL, shapes are decomposed into series of triangles. In case drawn object does not change, Phaser documentation recommends to bake it into texture with generateTexture() call. Then we can destroy Phaser.Graphics object as we do not need it anymore and method returns created texture.

Shape we draw is rectangle with top curved inside. For curve we use Bézier quadratic curve that has one control point. This point is shifted by Menu.GROUND_CP_Y pixels down in the middle of width. Bézier curves do not pass through control points. Instead, control points attract it:



Next menu object is goblin running along curved ground surface. He runs from left to the right and back. First direction is chosen randomly. For character we use skeletal animation we already used in game.

```
        private createGoblin(): void {
            // random direction
            this._runDirection = this.rnd.sign();

            // create Spriter loader - class that can change Spriter file into internal structure
            let spriterLoader = new Spriter.Loader();
            let spriterFile = new Spriter.SpriterXml(this.cache.getXML("GoblinAnim"));
            let spriterData = spriterLoader.load(spriterFile);

            this._goblin = new Spriter.SpriterGroup(this.game, spriterData, "Sprites", "Goblin",
"run", 100);
            this._goblin.scale.x = this._runDirection;

            // set position size
            this._goblin.position.set((this.game.width / 2 + 200) * this._runDirection,
Menu.GOBLIN_Y);

            // adds SpriterGroup to Phaser.World to appear on screen
            this.world.add(this._goblin);
        }
```

For random direction we use Phaser's method RandomDataGenerator.sign() which returns randomly 1 or -1. Initial position of goblin is set to be 200 pixels out of the screen horizontally in chosen direction.

We are not moving the character yet. It will be done in update() method later. If you want to check that this piece of code is working, set temporarily goblin's x position to visible area.

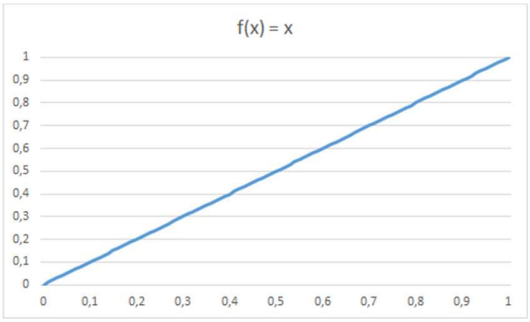
For main game title we use this piece of code:

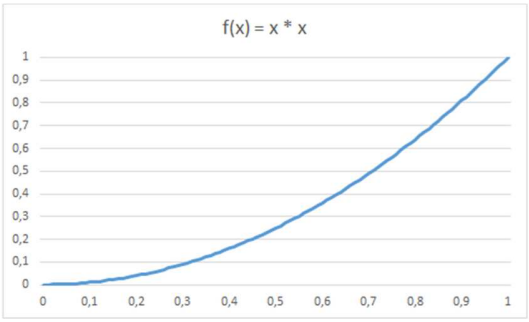
```
        private createTitle(): void {
            // title
            let title = this.add.sprite(0, -40, "Sprites", "Logo");
            title.anchor.set(0.5, 0.5);
```

```
            // title tweens
            this.add.tween(title).to({ angle: 3 }, 2500, function (k: number) {
                return Math.sin(k * 2 * Math.PI);
            }, true, 0, -1);
            this.add.tween(title.scale).to({ x: 1.02, y: 1.02 }, 1250, function (k: number) {
                return Math.sin(k * 2 * Math.PI);
            }, true, 0, -1);
        }
```

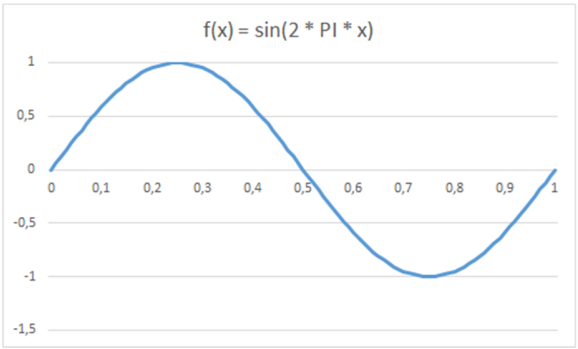
We put logo image slightly above screen center (-40 pixels in y direction) and use two tweens to animate it. First tween rotates it left and right with custom easing function in 2500 milliseconds and second one scales it up and down with the same easing function but two times faster (in 1250 milliseconds).

How easing functions work? It takes number in 0 to 1 interval and most often it returns number again in 0 to 1. This is how Phaser's default easing functions work. What is different is how slowly or fast is returned value growing from 0 to 1. For linear easing you get for some input value the same output (f(x) = x), for Quadratic.In function f(x) = x * x, etc. Shortly, when tween ends, object's property is set to target value.

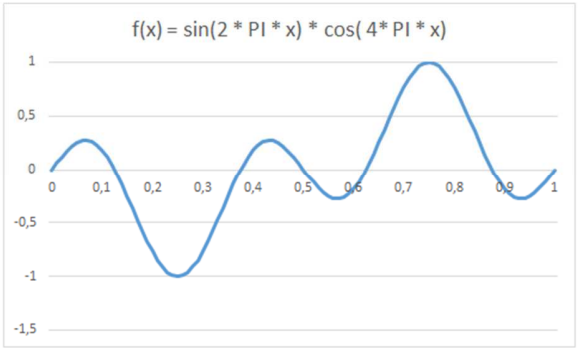


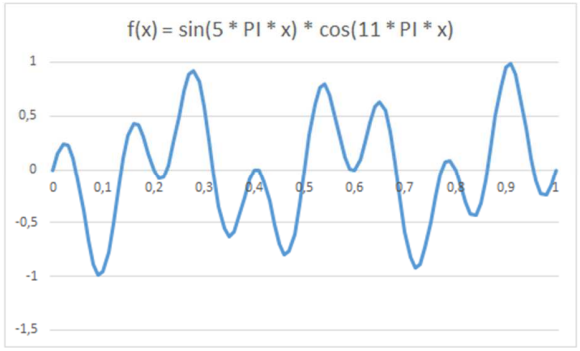


But I found it useful if, in some cases, object's property ends with value it had before tween started. This is handy especially for some swinging or wiggling things. All you need is to model some easing function that will return zero in the beginning as well as in the end. Sin is good candidate. Either half circle (PI) or full circle (2 * PI) does it. Target property value used in Tween.to() method then works as amplitude rather than target value.



You can even design more crazy easing functions:





What it is good for? If you, for example, create different function like this for x and y coordinate and set tween duration to some longer time span, you can get some image flying over screen in old 8-bit fashion.

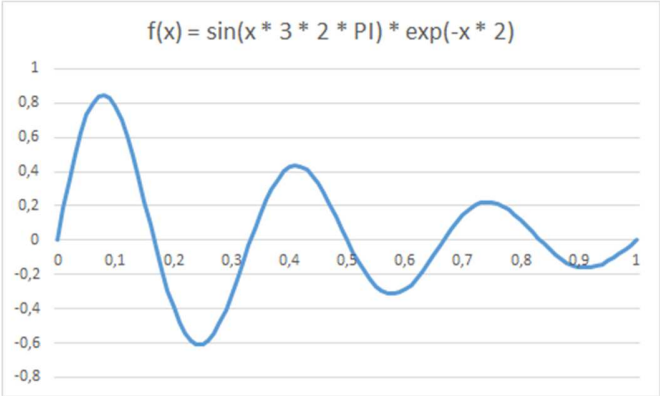
Our title image is rotating to 3 degrees in tween with full circle easing function, it means it will rotate from 0 to 3 to 0 to -3 to 0 degrees with nice sinusoidal deceleration near amplitude. The same trick is used for scaling. In method call we request to tween scale to 1.02, which in fact means that scale will go from 1 to 1.02 to 1 to 0.98 to 1.

Now, equipped with knowledge about easing functions we can move to start button.

```
        private createStartButton(): void {
            // start button
            let start = this.add.button(0, 220, "Sprites", function () {
                this.game.state.start("Play");
            }, this, "Start", "Start", "Start", "Start");

            start.anchor.set(0.5, 0.5);

            // input down callback
            start.onInputDown.add(function () {
                start.scale.set(0.9, 0.9);
                Sounds.sfx.play("select");
            }, this);

            // start button tween
            this.add.tween(start.scale).to({ x: 1.2, y: 0.9 }, 750, function (k: number) {
                let period = k * 3;
                let decay = -k * 2;
                return Math.sin(period * Math.PI * 2) * Math.exp(decay);
            }, true, 2000, -1).repeatDelay(2000);
        }
```

Start button is offset 220 pixels down from screen center. We have only one image (Start) for all its states. Clicking on it will start Play state and player will be thrown into game. Beside this it will scale down a little on button press (unless button tween is currently running).

To make start button attract player, we add tween with more complex easing function. This tween will repeat every 2 seconds and this is shape of easing function:



As target scale for x is set to value above 1 and target scale for y is set to value under 1, scale oscillating for x and y goes in opposite direction. Amplitude is damped to zero for both.

Last object on menu screen is sound toggle button in top right corner.

```
        private createSoundButton(): void {
            let frameName = Preferences.instance.sound ? "Sound_on" : "Sound_off";

            let sound = this.add.button(this.game.width / 2 - Menu.SOUND_BUTTON_OFFSET,
                -this.game.height / 2 + Menu.SOUND_BUTTON_OFFSET,
                "Sprites", function () {
                    let prefs = Preferences.instance;

                    // toggle sound setting
                    prefs.sound = !prefs.sound;
                    this.sound.mute = !prefs.sound;

                    // change button icon
                    let frameName = prefs.sound ? "Sound_on" : "Sound_off";
                    sound.setFrames(frameName, frameName, frameName, frameName);

                    Sounds.sfx.play("select");

                    prefs.save();
                }, this, frameName, frameName, frameName, frameName);

            sound.anchor.set(0.5, 0.5);

            this._sound = sound;
        }
```

First, we check what is current state of sound – muted or unmuted? We choose frame for button according to it. Sound button's position is offset by 50 pixels from top right corner.

Most of the work is done inside click callback. First we toggle game sound mute state in engine and in preferences. Note, in preferences we have variable sound where true means sound is on, but engine has property mute where true means sound is off. This is reason for negating value taken from preferences before mute property is set. According to new setting we change frames for all button states. To give some response to player, we play short sound and finally save preferences.

19.4 Updating menu screen

Part of menu screen objects is already animated with tweens. But trees are still and goblin is waiting out of screen for his time. update() method will change it:

```
        public update() {

            // elapsed time in seconds
            let delta = this.time.elapsed / 1000;

            // move bg
            this._treesBg.tilePosition.x -= Menu.BG_SPEED_X * delta;
```

First we calculate delta time in seconds. Then we move trees inside tilesprite with defined speed. Speed is defined in Menu.BG_SPEED_X static variable in pixels per second.

```
            // update goblin anim
            this._goblin.updateAnimation();
```

This call updates skeletal animation of goblin, but does not move him so he is running on place. To move him we have to do more as we want him to copy curved surface of ground. Let's solve movement in x and y direction separately.

First, movement in x direction.

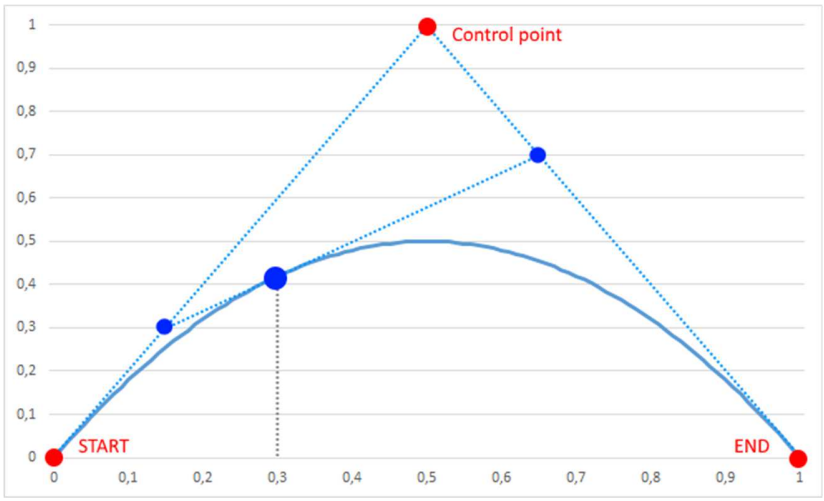
```
            // update goblin x poisiton
            this._goblin.x += Menu.GOBLIN_SPEED * delta * this._runDirection;
```

We just update x position with increment based on speed and direction. Then we check if goblin is already out of the screen (more than 200 pixels beyond screen border):

```
            // check if too far on right or left.
            if (this._goblin.x * this._runDirection > this.game.width / 2 + 200) {
                this._runDirection *= -1;
                this._goblin.x = -(this.game.width / 2 + 200) * this._runDirection;
                this._goblin.scale.x = this._runDirection;
            }
```

If out of the screen, we turn his run direction and also turn sprite with setting scale x to correct value.

Movement in y direction is more complicated. We know that surface is curved with Bézier quadratic curve that takes start and end point and one control point. Point on Bézier quadratic curve is calculated as distance from start point. Let's call this distance t. t is in interval from 0 to 1. If t = 0.5 we are calculating point in the middle of curve, if t = 1 we are in end point. To calculate point at t distance we do linear interpolation between linear interpolation between start and control point and linear interpolation between control point and end point. Sounds crazy, but look at picture below. It is drawn for t = 0.3. First we do linear interpolation between start and control point. Then we do linear interpolation between control point and end point. As last step we do linear interpolation between these two results.



In fact, because our control point is right in the middle in x direction, we need to calculate only y position.

```
            // calculate goblin Y position
            // three points for quadratic bezier curve (start point - control point - end point)
            // we need to calculate only y position so we can omit x coordinates
            let y0 = 0;
            let cy = Menu.GROUND_CP_Y;
            let y1 = 0;

            // map current goblin's position on screen into interval 0..1
            let t = Phaser.Math.clamp(
                Phaser.Math.mapLinear(this._goblin.x, -this.game.width / 2, this.game.width / 2, 0,
1),
                0, 1);

            // calculate y position
            this._goblin.y = Menu.GOBLIN_Y + Phaser.Math.linear(
                Phaser.Math.linear(y0, cy, t),
                Phaser.Math.linear(cy, y1, t),
                t);
        }
```

We start from known x position on screen. We map it into interval from zero to one with Phaser.Math.mapLinear() method to get t value. We have to clamp result to keep it in desired range as mapLinear() just says how to map from one range into another but does not check bounds.

With t calculated we can calculate goblin's new y position with nested linear interpolations as described earlier.

To stop menu music when leaving state add shutdown() method. This is one of methods Phaser recognizes, and if it finds it in your state class, it will call it when leaving state.

```
        public shutdown() {
            // stop music when leaving this state
            Sounds.musicMenu.stop();
        }
```

19.5 Scaling

To close menu state code, we have to add last method. It will react on screen size changes. Thanks to setting coordinates origin into center of screen it will be easy.

```
        public onResize(width: number, height: number): void {
            // resize vamera position and world bounds
            this.setView(width, height);

            // recreate ground texture
            this._ground.setTexture(this.generateGround());

            // change tilesprite width
            this._treesBg.width = width;

            // reposition sound button
            this._sound.position.set(this.game.width / 2 - Menu.SOUND_BUTTON_OFFSET,
                -this.game.height / 2 + Menu.SOUND_BUTTON_OFFSET);
        }
    }
}
```

First we update world bounds to take into account new size of window. In other words, we are keeping 0,0 coordinates origin in the middle of screen with it.

Then we recreate ground to prevent its deformation while scaling. We create new texture that is drawn with regard to new screen width. We also adjust width of tilesprite with trees to cover whole window width.

As last, we adjust position of sound toggle button to keep it 50 pixels offset from top right corner.

Now, compile and run game. It will start with menu screen and if you die during game, it will return there again.

Summary

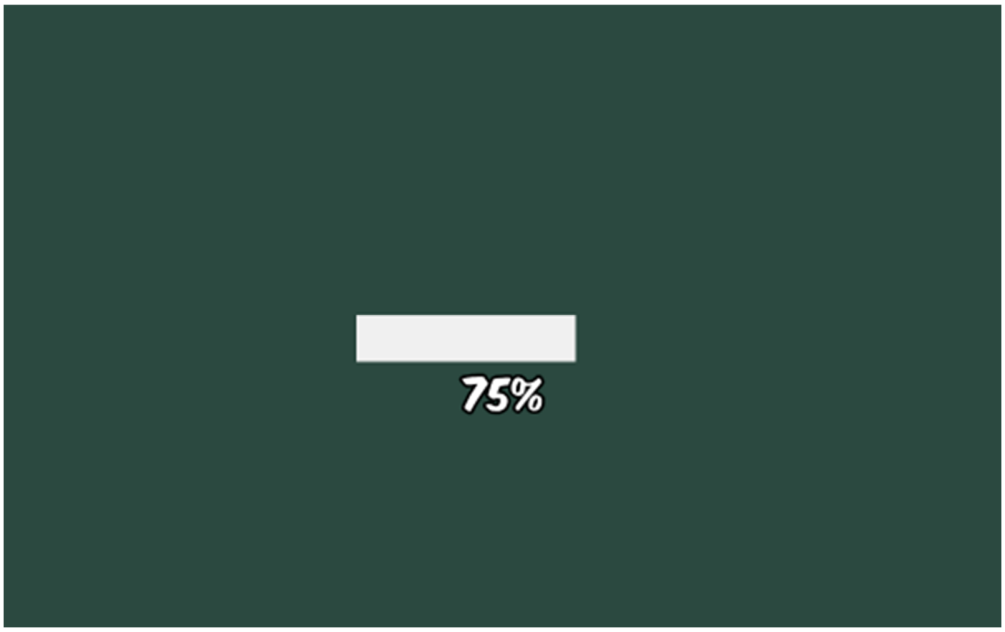
We are almost finished. Now you can start game repeatedly from menu screen without need of reloading page every time you die. We are missing only one last thing to make it look more professional – loading bar. This is what next chapter will be about.

20. Loading screen

Last missing piece in our game is loading screen. It is what we will build in this chapter. Its purpose is to indicate how much of game data already have been loaded. Even simple animation like growing bar can make waiting less boring for player. You can of course make some great complex loading screen, but keep in mind that assets for it also have to be loaded first.

20.1 Loading font

Our loading screen will be very simple – just loading bar and text saying how many percent of loading is done:



Font we will use is the same we use in game. We just need to load it already in Boot state.

Open Boot.ts and add preload() method into it:

```
        public preload(): void {
            // font
            this.load.bitmapFont("Font", "assets/Font.png", "assets/Font.xml");
        }
```

Later, when adjusting Preload.ts, we will comment out this line there.

In Phaser you do not have to load all your assets in Preload state. You can load assets in any state in which you implement preload() method. All loaded assets stay in cache until you clear it.

20.2 Constructing bar

Up to now, Preload state was just loading assets and did not give any visual feedback to player. We will change it now. First, we will add some variable to the top of it. Add new lines in bold:

```
    export class Preload extends Phaser.State {

        private static LOADING_BAR_WIDTH = 300;

        // music decoded, ready for game
        private _ready: boolean = false;

        // sprite for loading bar
        private _loadingBar: Phaser.Sprite;
        // loading text
        private _loadingText: Phaser.BitmapText;
```

First new line is bar width in pixels. Other two lines hold sprite representing loading bar and bitmap text for loading percent.

In previous part we loaded font in Boot state – it is now only asset in cache when we enter preload() method of Preload state. Sprite for bar will be created dynamically. We will construct whole loading screen arrangement before we start scheduling files for loading. Locate preload() method and on top of it add following lines:

```
            this.stage.backgroundColor = 0x2B4940;

            this.setView(this.game.width, this.game.height);
```

This sets background color for whole screen. setView() method will be introduced later. But it is exactly the same method we used for moving coordinates origin into center of screen in Menu state. Writing the same code twice is bad practice as it increases maintenance overhead if you want to make change in it later. If your game is larger you may consider to move it into some parent class or class with static utility methods. We are setting coordinates origin into center of screen as it will allow us to easily react on window size changes. Bar and percent text will be positioned relative to origin, so when window size is changed, it will stay in center.

In next piece of code, we will create sprite for loading bar.

```
            // create graphics for loading bar
            let g = new Phaser.Graphics(this.game);
            // fill color
            g.beginFill(0xF0F0F0);
            // draw a shape
            g.drawRect(0, 0, 8, 8);
            // loading bar sprite
            this._loadingBar = this.add.sprite(-Preload.LOADING_BAR_WIDTH / 2, 0,
g.generateTexture());
            this._loadingBar.width = 0;
            this._loadingBar.height = 48;
            // we do not need graphics anymore
            g.destroy();
```

We dynamically draw small (8 x 8) white square and generate texture from it. This texture is used by _loadingBar sprite. This sprite is located half of the total bar width to the left from center. Its height is set to 48 pixels and initial width to 0 pixels.

After sprite is created, we can destroy graphics object as we do not need it.

Next, we create bitmap text initially set to “0%”.

```
            // laoding text
            this._loadingText = this.add.bitmapText(0, 60, "Font", "0%", 40);
            this._loadingText.anchor.x = 0.5;
```

Text is anchored to center itself horizontally and is placed relatively to coordinates center – 60 pixels down from it.

Last change in preload() method is commenting out font loading as we already loaded it in Boot state:

```
            // font
            // this.load.bitmapFont("Font", "assets/Font.png", "assets/Font.xml");
```

20.3 Resizing

Even though loading bar is not so important part of game, and it may be visible on screen only for short time, we want it to correctly react to window size changes. Earlier in create() method we called setView(). We will add it now along with onResize() callback.

```
        public onResize(width: number, height: number): void {
            this.setView(width, height);
        }

        // ---------------------------------------------------------------------
        public setView(width: number, height: number): void {
            // set bounds
            this.world.setBounds(-width / 2, -height / 2, width / 2, height / 2);
            // focus on game center
            this.camera.focusOnXY(0, 0);
        }
```

There is nothing special here. We just keep point 0,0 in the middle of the screen. We do not need to adjust position of bar sprite or loading text because their positions are relative to coordinates center.

20.4 Updating bar

As files are one by one loaded, we need some way how to get notified about it so we can update loading bar width and percent number. Fortunately, there is another special Phaser function which will be called on load progress if defined in state – loadUpdate(). This method gets called whenever loader completes loading of file in loading queue. If you read value of progress property of Phaser.Loader class, you can get rounded percentage value of total files loaded.

```
        public loadUpdate(): void  {
            // update bar width
            this._loadingBar.width = Preload.LOADING_BAR_WIDTH * this.load.progress / 100;

            // update loading text percent
            this._loadingText.text = this.load.progress + "%";
        }
```

As last thing, we will make small change in update() method. We will add half second delay before we change from Preload to Menu state. Only reason for this is to gain some short time, so player can see full

bar and “100%” underneath it. Without delay it may sometimes look like loading bar did not get it to its full width.

```
                // small delay before changing state
                this.time.events.add(500, function () {
                    this.game.state.start("Menu");
                }, this);
```

To create short delay, we add one-time timer event with delay 500 milliseconds.

Summary

With loading screen in place our game is complete!

21. Conclusion

Our game is complete. It took quite a long way to get here, so let's summarize what we achieved.

After we set new project and created simple game skeleton (chapters 2 and 3), we delved into world of procedural generation. We calculated jump tables in chapter 4 and created first version of platforms generator in next one.

To have some fun, we added player in chapter 6. While it was just a blue rectangle placeholder we could play the game since. In next chapter we focused on generator again and made it produce more pleasing results from visual and aesthetics point of view.

Chapter 8 was first one in which we used final game graphics assets – for tiles that time. One chapter later we did the same for main goblin character. Beside this, we made our goblin nicely animated with skeletal animations and we learned how to use Spriter Player for Phaser. Adding graphics continued in chapter 10 in which we added scrolling background layers.

Chapters 11, 12 and 13 added new features into game – spikes, bonus jumps and gold nuggets to gather.

In chapter 14 and 15 we focused on details that make game nice. We added particle effects and shadow below running goblin. Chapter 16 added another small detail – displaying maximum reached distance along with code for saving and loading preferences.

In chapter 17 our game finally stopped to be silent. We added sound effects and music.

Chapter 18 focused on scaling. Our game now fills all available window space even if it has really strange dimensions.

In last two chapters we added new game states. First one was menu screen in chapter 19 and second loading screen in chapter 20.

Now, we have game with complete flow from loading screen to menu and game itself, that has flexible procedural generator separated from the rest of the game, which can be easily parametrized to tweak gameplay or even used for different game. Beside this our main character is presented with smooth skeletal animations.