Phaser.js: A Step-by-Step Tutorial On Making A Phaser 3 Game

#game development #JavaScript #Phaser

In this article, we are going to develop from scratch a game made with Phaser.js. You'll learn how to set up a build on webpack, load assets, create characters and animations, add keyboard controls, handle a powerful tool for creating maps that is

Tiled, and even how to implement a simple bot behavior. Let's go!

Table of contents

About Phaser

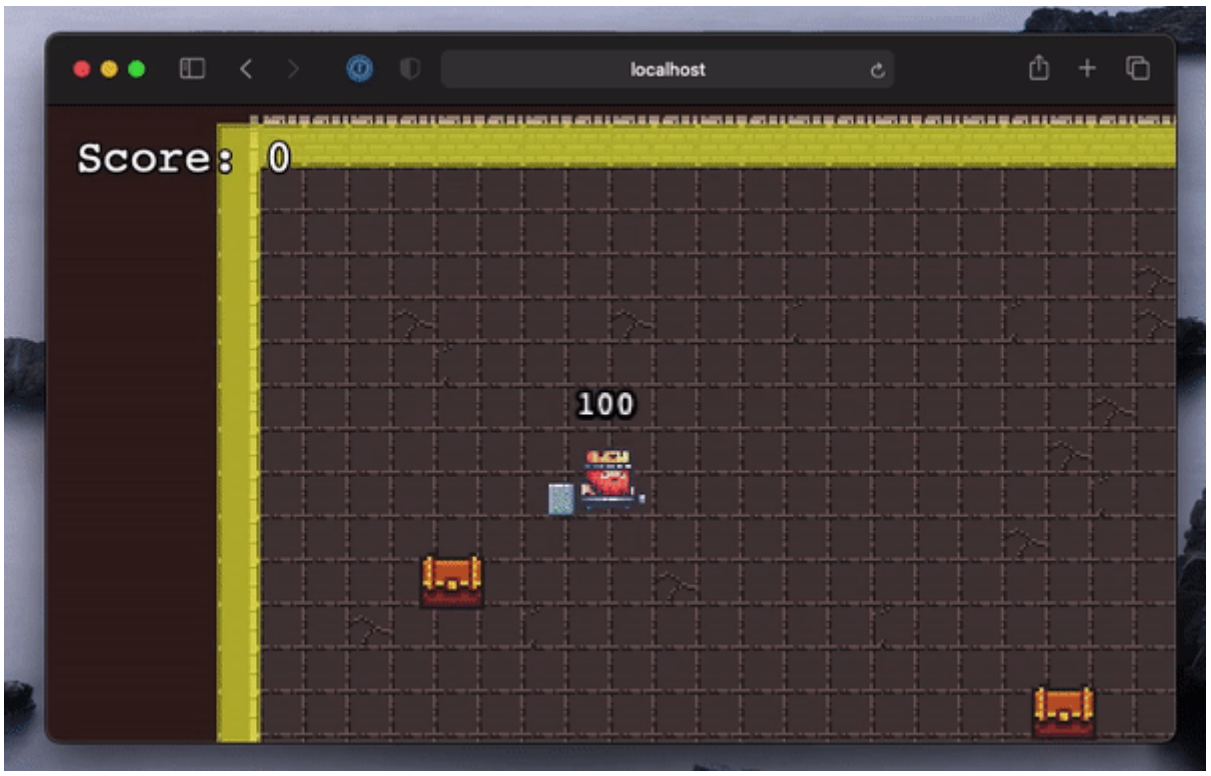
First, a little background. Phaser is an open-source JavaScript 2D game development framework developed by the folks at Photon Storm. It uses Canvas and WebGL renderers. You can play games developed with Phaser 3 in any modern web browser, and with tools like Apache Cordova, you can even turn them into mobile or native desktop apps. Phaser is open-source, easy to get started, and generally a great option for people who are looking to try JS for game development.

What you need to start

- Basic knowledge of JavaScript
- Basic knowledge of TypeScript to be able to get what all those "types" are
- Slight knowledge about webpack
- A code editor

- A package manager (**yarn** OR **npm**)

That's it! After completing the tutorial, you'll be able to create games like this:



Check it out and play the Demo version.
Here are the assets for this tutorial that you are going to need.
And this is where you can find the final code.
So since it looks like we are ready to start, let's turn to the initial stage which is about...

Part 1: **Installing packages and configuring webpack**

We begin by setting up the environment, the required packages, and setting up webpack.
For this tutorial, we'll be using the **yarn** package manager, but you can use the **npm** one as well since we only need it to install packages and launch the app build.

Preparing the file structure

```
mkdir game-example
cd game-example
mkdir src src/assets src/classes src/scenes
```

We get the following structure:

```
├── src
    ├── assets
    ├── classes
    └── scenes
```

`assets` — this is where we'll store all game assets: png sprites, sprite sheets, and JSON files.

`classes` — for classes (player, score meter, etc.).

`scenes` — a place to store the game scenes.

Let's add the `index.html` file to `src/` an make it an entry point of the application:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="user-scalable=no, width=device-width
  <style>
    html,
    body {
      margin: 0;
      padding: 0;
    }
  </style>
</head>
<body>
  <div id="game"></div>
</body>
```

```
</html>
```

Here we tell the game to not scale, remove the screen border padding, and specify the div `id="game"` element that'll be the parent block where the game will be rendered.

Initialization and packages

Initializing `package.json`:

```
yarn init
```

Filling our `package.json` with the `Phaser` itself:

```
yarn add phaser
```

And installing typescript and webpack with the necessary plugins and loaders:

```
yarn add -D typescript @types/node cross-env webpack webpack-cli webp
```

Note that we install **html-webpack-plugin@5.0.0-alpha.10** because at the time of this article's publication the plugin had an unresolved bug.
Add `eslint` and match it with `typescript` and `prettier` for linting and quick code formatting:

```
yarn add -D eslint eslint-config-prettier eslint-plugin-prettier prei
```

Done! We have installed all the necessary packages, now we can proceed to set up the configs.

Setting up configs

We'll create the configuration files at the root level of the file structure.

tsconfig.json

```
{
  "compilerOptions": {
    "sourceMap": true,
    "strict": true,
    "noImplicitReturns": true,
    "noImplicitAny": true,
    "module": "es6",
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "esModuleInterop": true,
    "target": "es5",
    "allowJs": true,
    "baseUrl": ".",
  },
  "include": [
    "./index.d.ts",
    "**/*.ts"
  ],
  "exclude": [
    "node_modules",
    "./dist/"
  ]
}
```

You can read more about the contents of `tsconfig.json` in the documentation.

.eslintrc.js

```
module.exports = {
  parser: '@typescript-eslint/parser',
  plugins: ['@typescript-eslint'],
  extends: [
    'plugin:@typescript-eslint/recommended',
    'plugin:prettier/recommended',
    'prettier/@typescript-eslint',
  ],
  parserOptions: {
    ecmaVersion: 2018,
    sourceType: 'module',
    ecmaFeatures: {
      jsx: true,
    },
  },
  rules: {
    '@typescript-eslint/camelcase': 0,
    '@typescript-eslint/explicit-function-return-type': 0,
    '@typescript-eslint/explicit-member-accessibility': 0,
    '@typescript-eslint/interface-name-prefix': 0,
    '@typescript-eslint/no-explicit-any': 0,
    '@typescript-eslint/no-object-literal-type-assertion': 0,
    'sort-imports': [
      'warn',
      {
        ignoreCase: true,
        ignoreDeclarationSort: true,
        ignoreMemberSort: false,
        memberSyntaxSortOrder: ['none', 'all', 'multiple', 'single'],
      },
    ],
  },
};
```

.prettierrs.js

```
module.exports = {
  printWidth: 100,
  semi: true,
  singleQuote: true,
```

```
  tabWidth: 2,
  trailingComma: 'all',
};
```

Optional, to automatically format files when saving to VSCode.

./vscode/settings.json

```
{
  "typescript.tsdk": "node_modules/typescript/lib",
  "eslint.autoFixOnSave": true,
  "eslint.validate": [
    "javascript",
    "javascriptreact",
    {
      "language": "typescript",
      "autoFix": true
    },
  ],
  "[javascript]": {
    "editor.formatOnSave": false
  },
  "[typescript]": {
    "editor.formatOnSave": false
  },
  "editor.codeActionsOnSave": {
    "source.fixAll.eslint": true
  },
}
```

Turning standard formatting off to avoid situations where it could get in conflict with eslint formatting.

You can customize rules for eslint, prettier, and vscode to better suit your needs, it won't affect working with Phaser in any way 😌

webpack.config.js

So that we can build our creation and run the dev-server, we need to first make a config for webpack:

```
/* eslint-disable @typescript-eslint/no-var-requires */
const path = require('path');
const { CleanWebpackPlugin } = require('clean-webpack-plugin');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const TerserPlugin = require('terser-webpack-plugin');
const isProd = process.env.NODE_ENV === 'production';
const babelOptions = {
  presets: [
    [
      '@babel/preset-env',
      {
        targets: 'last 2 versions, ie 11',
        modules: false,
      },
    ],
  ],
};
const config = {
  mode: isProd ? 'production' : 'development',
  context: path.resolve(__dirname, './src'),
  entry: './index.ts',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
  },
  module: {
    rules: [
      {
        test: /\.ts(x)?$/,
        exclude: /node_modules/,
        use: [
          {
            loader: 'babel-loader',
            options: babelOptions,
          },
          {
            loader: 'ts-loader',
          },
        ],
      },
      {
        test: /\.html$/,
```

```
      use: [
        {
          loader: 'html-loader',
        },
      ],
    },
    {
      test: /\.css$/,
      use: ['style-loader', 'css-loader'],
    },
  ],
},
resolve: {
  extensions: ['.ts', '.js'],
},
optimization: {
  minimize: true,
  minimizer: [
    new TerserPlugin({
      extractComments: false,
      terserOptions: {
        output: {
          comments: false,
        },
      },
    }),
  ],
},
plugins: [
  new CleanWebpackPlugin(),
  new HtmlWebpackPlugin({
    template: 'index.html',
    inject: true,
    title: 'Phaser Webpack Template',
    appMountId: 'app',
    filename: 'index.html',
    inlineSource: '.(js|css)$',
    minify: false,
  }),
],
devServer: {
  contentBase: path.join(__dirname, 'dist'),
```

```
    port: 5000,
    inline: true,
    hot: true,
    overlay: true,
  },
};
module.exports = config;
```

What we indicate in the config:

- what loaders to use for files
- what plugins to apply during the build
- that the minification is done using Terser
- dev-server settings.

You can find out more about each field of the webpack.config.js file on the official site.

Done!

We have described all the required configs. Now, all we need is to just write a couple of scripts to start dev-server and run the build.

Let's go to **package.json** and add this field:

```
...
"scripts": {
    "dev": "cross-env NODE_ENV=development webpack serve",
    "build": "cross-env NODE_ENV=production webpack --progress --hide
},
...
```

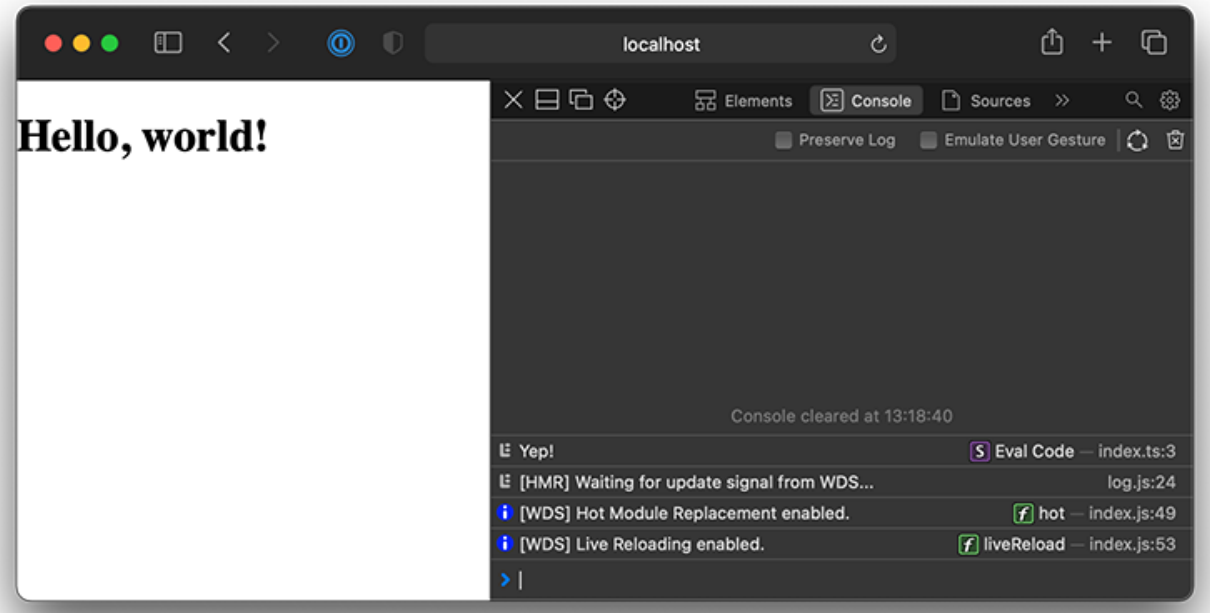
Awesome!

Now let's create a src/index.ts file that acts as an entry point for the bundle scripts and add some code snippets there to make sure everything works fine after starting dev-server.

src/index.ts

```
console.log('Hello world!');
```

Add a header to **src/index.html** just for testing purposes, and run dev-server to make sure everything's OK.
To start dev-server, execute the command `yarn dev`.



"Hello, world!" and the "Yep!" message in the console indicate that everything is set up alright and we can finally move on to Phaser itself 😎

Part 2: **The first scene, loading assets and showing a character on screen**

Setting up a game

In the previous part, we set up the entire development environment from scratch including webpack, TypeScript, linter, and formatter, which means it's time to move on to setting up the game and begin placing objects. Don't worry, it's not one massive file. In this part, we'll start small by showing our character.

First, we need to declare a Game object. It's the most important of the required objects since Phaser won't initialize without it. We'll initialize it at the "entry point", namely **src/index.ts**.

A Game must contain the second required object — a Scene, at least one. A Scene in the Phaser world is similar to a theater scene in the real world. It contains child elements like a real scene contains actors. These can be Sprites, Images, and Containers. For a start, this set of elements is enough for our purposes, since we'll be creating our own classes for child elements, inheriting their class properties.

To declare a game, we need to indicate what parameters we'll launch it with. So, describe the following in the parameters:

- `title` — game title
- `type` — render type, can be **CANVAS**, **WEBGL**, or **AUTO**. Many effects can be unavailable with the CANVAS type that are available with the WEBGL one. In this tutorial, we'll be using the latter
- `parent` — DOM element id of the page where we'll add a Game canvas element)
- `backgroundColor` — the background color of the canvas
- `scale` — setting for resizing the game canvas. Our choice is `mode`: `Phaser.Scale.ScaleModes.NONE` since we'll have our own sizing system. Read more about modes here
- `physics` — an object for setting the game physics
- `render` — additional properties of a game render
- `callbacks` — callbacks that will be triggered BEFORE (`preBoot`) or AFTER (`postBoot`) the game is initialized
- `canvasStyle` — CSS styles for the canvas element where the game will be rendered

- `autoFocus` — autofocus on the game canvas
- `audio` — sound system settings
- `scene` — a list of scenes to load and use in the game.

To declare a game, we need to indicate what parameters we'll launch it with. So, describe the following in the parameters:

src/index.ts

```
import { Game, Types } from 'phaser';
import { LoadingScene } from './scenes';
const gameConfig: Types.Core.GameConfig = {
    title: 'Phaser game tutorial',
  type: Phaser.WEBGL,
  parent: 'game',
  backgroundColor: '#351f1b',
  scale: {
    mode: Phaser.Scale.ScaleModes.NONE,
    width: window.innerWidth,
    height: window.innerHeight,
  },
  physics: {
    default: 'arcade',
    arcade: {
      debug: false,
    },
  },
  render: {
    antialiasGL: false,
    pixelArt: true,
  },
  callbacks: {
    postBoot: () => {
      window.sizeChanged();
    },
  },
  canvasStyle: `display: block; width: 100%; height: 100%;`,
  autoFocus: true,
  audio: {
```

```
    disableWebAudio: false,
  },
  scene: [LoadingScene],
};
```

Let's add to the same file a global function for resizing our game:

src/index.ts

```
...
window.sizeChanged = () => {
  if (window.game.isBooted) {
    setTimeout(() => {
      window.game.scale.resize(window.innerWidth, window.innerHeight)
      window.game.canvas.setAttribute(
        'style',
        `display: block; width: ${window.innerWidth}px; height: ${wir
      );
    }, 100);
  }
};
window.onresize = () => window.sizeChanged();
```

Finally, we can create a Game itself:

src/index.ts

```
...
window.game = new Game(gameConfig);
```

To avoid getting the error about Window not having a `window.sizeChanged` method and a global `window.game` object, let's patch the `Window` interface:

```
interface Window {
  sizeChanged: () => void;
  game: Phaser.Game;
```

```
}
```

Now TypeScript understands what is behind `window.game` and `window.sizeChanged`.

Scene creation

If we run the game without a single scene, we'll get an error. Let's create the first scene then, which will later act as the main scene for loading assets and launching the rest of the scenes.
Create a scene Loading file to describe the Scene Class. In the constructor, indicate the Scene Key. We'll use it to select a specific scene among others. The Scene Key is a required parameter.
src/scenes/loading/index.ts

```
import { Scene } from 'phaser';
export class LoadingScene extends Scene {
  constructor() {
    super('loading-scene');
  }
  create(): void {
    console.log('Loading scene was created');
  }
}
```

As you can see, we are using the `create() {...}` method. It is one of the built-in scene methods and is the scene lifecycle method. There are several such methods:

- `init(data) {}` — kicks in when a scene is created. It accepts the Data Object that we can pass when we call game.scenes.add(dataForInit) or

game.scenes.start(dataForInit). For example, when we create a scene while being in some other scene (yes, you can do that). All scenes will be at the same hierarchy level, with no nested scenes.

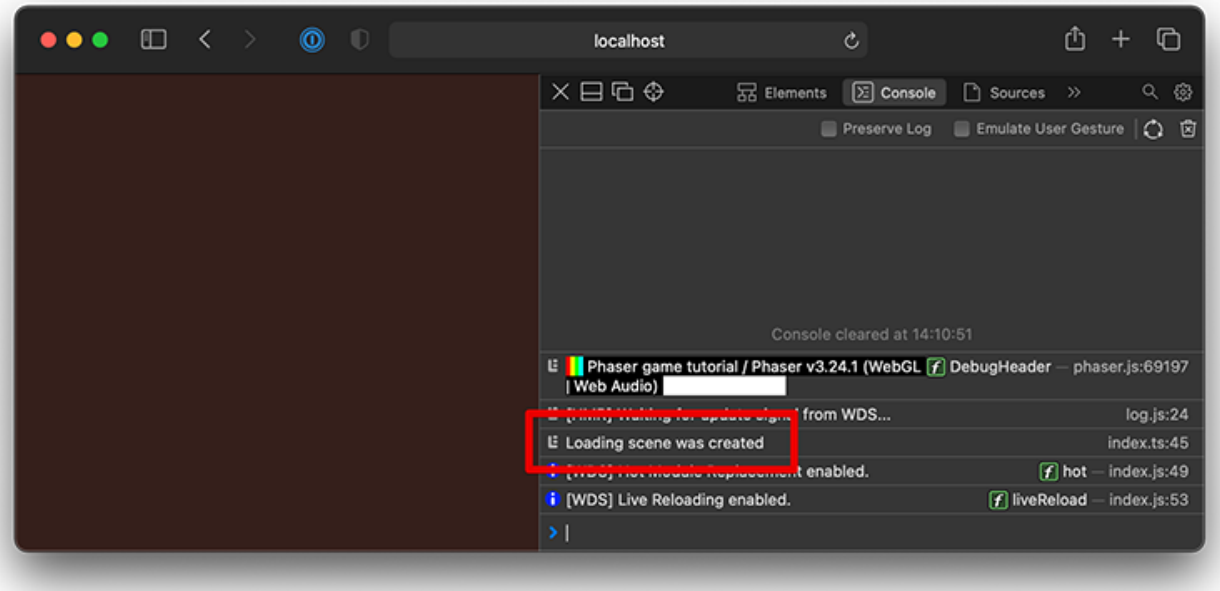
- `preload() {}` — a method that defines what we need to load before the scene and from where. We'll use it to load assets later on.
- `create(data) {}` — a method that gets triggered when a scene is created. In it, we'll specify positioning for such scene elements as Character and Enemies.
- `update(time, delta) {}` — a method that gets called with every render frame (on average, 60 times per second). It's a game loop in which redrawing, moving objects, etc. occurs.

Also, for convenient scene import, let's create a file with which we will "transmit" the exports in nested directories.

src/scenes/index.ts

```
export * from './loading';
```

Now we can start our game to see that the scene has been created and has triggered the message to the console, which we'd specified in the `create()` method. The brown of the background is the `backgroundColor` we specified in `gameConfig`.



Let's load a character for this background!

To do it, create a folder **src/assets/sprites/**, and add there a picture of our character — `king.png`. Next, in the `preload()` method of the scene class, add the following:

src/scenes/loading/index.ts

```
...
preload(): void {
    this.load.baseURL = 'assets/';
    // key: 'king'
    // path from baseURL to file: 'sprites/king.png'
    this.load.image('king', 'sprites/king.png');
}
...
```

Now to creating our character, let it be a simple sprite. Declare a character field, and write the following in the scene's `create()` method:

src/scenes/loading/index.ts

```
export class LoadingScene extends Scene {
  private king!: GameObjects.Sprite;
    constructor() {...}
    ...
    create(): void {
        this.king = this.add.sprite(100, 100, 'king');
    }
...
```

Here we indicate that we add a sprite (add.sprite), place it at such and such XY coordinates, and use a texture with the `'king'` key which we specified during loading in the `preload()` method.

Now, if we run the dev-server, instead of a character we'll get a black square with a green border. Squares like these signal that Phaser was unable to detect the texture

or sprite along a path.

But the path is right, so what gives?

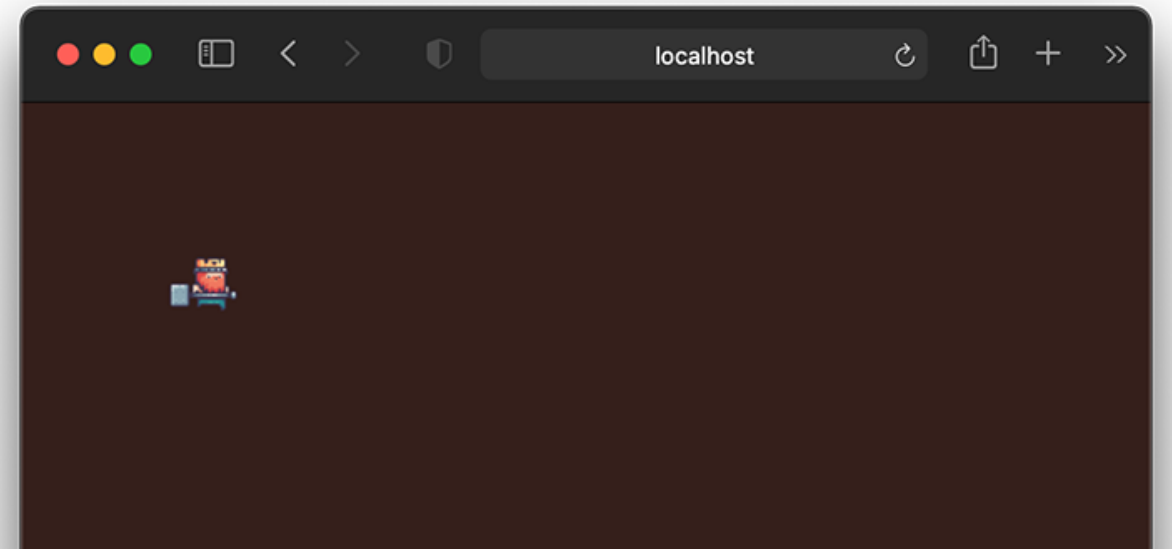
It has to do with webpack. We haven't specified the assets' movement when building or running in dev mode. Let's fix it.

Add plugin import and initialization to **webpack.config.js**.

webpack.config.js

```
...
const CopyWebpackPlugin = require('copy-webpack-plugin');
...
plugins: [
    ...,
    new CopyWebpackPlugin({
      patterns: [
        {
          from: 'assets',
          to: 'assets',
        },
      ],
    }),
    ...
]
```

Now, when running in dev mode, we can see our character 😎



Part 3: **Animating a character, adding the ability to move, keybinding**

Great! We now know how to load and place sprites. One thing left... Need more interactivity! Let's bring our character to life by giving him the ability to move. We'll start by creating a separate class for our character and a separate scene for the level and call it `level-1-scene`. We begin with the scene since we've already placed our character there in the previous part of this tutorial.
Let's create a scene folder with the index.ts scene file.

src/scenes/level-1/index.ts

```
import { Scene } from 'phaser';
export class Level1 extends Scene {
  constructor() {
    super('level-1-scene');
  }
  create(): void {
        ...
    }
}
```

Don't forget to add the "transmitter" to the scene.

src/scenes/index.ts

```
export * from './loading';
export * from './level-1';
```

Now we need to add our `level-1` scene to the scene array in `gameConfig`.

src/index.ts

```
import { Level1, LoadingScene } from './scenes';
...
const gameConfig = {
    ...
    scene: [LoadingScene, Level1],
    ...
};
```

And move our `king` from the Loading scene to the Level1 scene:

```
import { GameObjects, Scene } from 'phaser';
export class Level1 extends Scene {
  private king!: GameObjects.Sprite;
  constructor() {
    super('level-1-scene');
  }
  create(): void {
    this.king = this.add.sprite(100, 100, 'king');
  }
}
```

In the Loading scene, we set our scene-level to launch after all other loadings are finished:

src/scenes/loading/index.ts

```
...
preload(): void {
  this.load.baseURL = 'assets/';
  this.load.image('king', 'sprites/king.png');
}
...
create(): void {
```

```
  this.scene.start('level-1-scene');
}
...
```

In the `this.scene.start()` method, we specify the key of the scene we want to launch.

Let's run the dev server and make sure our character is displayed.

Awesome! Looks like he's in his proper place.

Creating an actor

Since in the future there will be enemies behaving somewhat similar to the player character, let's create an actor class to store general properties and methods, and then create a player class by extending the actor one.

src/classes/actor.ts

```
import { Physics } from 'phaser';
export class Actor extends Physics.Arcade.Sprite {
    protected hp = 100;
  constructor(scene: Phaser.Scene, x: number, y: number, texture: str
    super(scene, x, y, texture, frame);
    scene.add.existing(this);
    scene.physics.add.existing(this);
    this.getBody().setCollideWorldBounds(true);
  }
    public getDamage(value?: number): void {
    this.scene.tweens.add({
      targets: this,
      duration: 100,
      repeat: 3,
      yoyo: true,
      alpha: 0.5,
      onStart: () => {
        if (value) {
```

```
            this.hp = this.hp - value;
          }
        },
        onComplete: () => {
          this.setAlpha(1);
        },
      });
  }
    public getHPValue(): number {
    return this.hp;
  }
    protected checkFlip(): void {
    if (this.body.velocity.x < 0) {
      this.scaleX = -1;
    } else {
      this.scaleX = 1;
    }
  }
  protected getBody(): Physics.Arcade.Body {
    return this.body as Physics.Arcade.Body;
  }
}
```

Here we indicate that the `actor` class is not just some sprite but a physical one. This is to adjust its collisions and physical dimensions for contact with walls and other objects.

The `scene.add.existing(this)` and `scene.physics.add.existing(this)` methods tell us that we are adding our physical sprite to the scene and need it considered in terms of scene physics.

The `this.getBody().setCollideWorldBounds(true)` method tells the world to react to the physical model of the sprite, its “box”.

The protected `checkFlip()` method is designed to rotate an actor as it moves left or right.

The public `getDamage()` method is for attacking the actor. We can see a tween inside — something like an animation done by manipulating some properties of the target object. Here we describe that the `alpha` transparency will change 3 times `repeat: 3` within 100 ms `duration: 100`, each time returning to the original alpha

value `yoyo: true`. At the start of the blinking animation `onStart`, we change the actor's HP value in accordance with how much damage it received. At the end of the `onComplete` animation, we forcefully set the current character's opacity to `this.setAlpha(1)`. The value of the `alpha` property varies from 0 to 1. The lower the value, the more transparent the object.
The `getBody()` method was created due to errors in Phaser types and helps us get that very physical `body` model of a physical object.

Creating a player character

Now that the main actor class has been created, we can create the player class.
src/classes/player.ts

```
import { Actor } from './actor';
export class Player extends Actor {
  private keyW: Phaser.Input.Keyboard.Key;
  private keyA: Phaser.Input.Keyboard.Key;
  private keyS: Phaser.Input.Keyboard.Key;
  private keyD: Phaser.Input.Keyboard.Key;
  constructor(scene: Phaser.Scene, x: number, y: number) {
    super(scene, x, y, 'king');
    // KEYS
    this.keyW = this.scene.input.keyboard.addKey('W');
    this.keyA = this.scene.input.keyboard.addKey('A');
    this.keyS = this.scene.input.keyboard.addKey('S');
    this.keyD = this.scene.input.keyboard.addKey('D');
    // PHYSICS
    this.getBody().setSize(30, 30);
    this.getBody().setOffset(8, 0);
  }
  update(): void {
    this.getBody().setVelocity(0);
    if (this.keyW?.isDown) {
      this.body.velocity.y = -110;
    }
```

```
    if (this.keyA?.isDown) {
      this.body.velocity.x = -110;
      this.checkFlip();
      this.getBody().setOffset(48, 15);
    }
    if (this.keyS?.isDown) {
      this.body.velocity.y = 110;
    }
    if (this.keyD?.isDown) {
      this.body.velocity.x = 110;
      this.checkFlip();
      this.getBody().setOffset(15, 15);
    }
  }
}
```

Let's analyze the code above.

Here we describe the keys to track which one of them is being pressed:

```
private keyW: Phaser.Input.Keyboard.Key;
private keyA: Phaser.Input.Keyboard.Key;
private keyS: Phaser.Input.Keyboard.Key;
private keyD: Phaser.Input.Keyboard.Key;
...
this.keyW = this.scene.input.keyboard.addKey('W');
this.keyA = this.scene.input.keyboard.addKey('A');
this.keyS = this.scene.input.keyboard.addKey('S');
this.keyD = this.scene.input.keyboard.addKey('D');
```

In each frame, we check if any control key is pressed and change the XY movement speed depending on the direction. If a key is not pressed, set the speed = 0. Also, when moving left-right, we check whether we need to rotate the character.

```
update(): void {
    this.getBody().setVelocity(0);
    if (this.keyW?.isDown) {
```

```
      this.body.velocity.y = -110;
    }
    if (this.keyA?.isDown) {
      this.body.velocity.x = -110;
      this.checkFlip();
      this.getBody().setOffset(48, 15);
    }
    if (this.keyS?.isDown) {
      this.body.velocity.y = 110;
    }
    if (this.keyD?.isDown) {
      this.body.velocity.x = 110;
      this.checkFlip();
      this.getBody().setOffset(15, 15);
    }
  }
```

Sometimes character sprites are quite large or have white space. In this case, we can specify the size of the physical model (box) `setSize(width, height)` and set the point by XY coordinates to calculate the physical model.

```
this.getBody().setSize(30, 30);
this.getBody().setOffset(8, 0);
```

Note that when rotating the character, we move the rendering point of the physical model. We have to do it because of the body miscalculation error of Phaser.

Adding a player character

Wonderful! We have created a character class, now we need to create a character on the first level scene.

src/scenes/level-1/index.ts

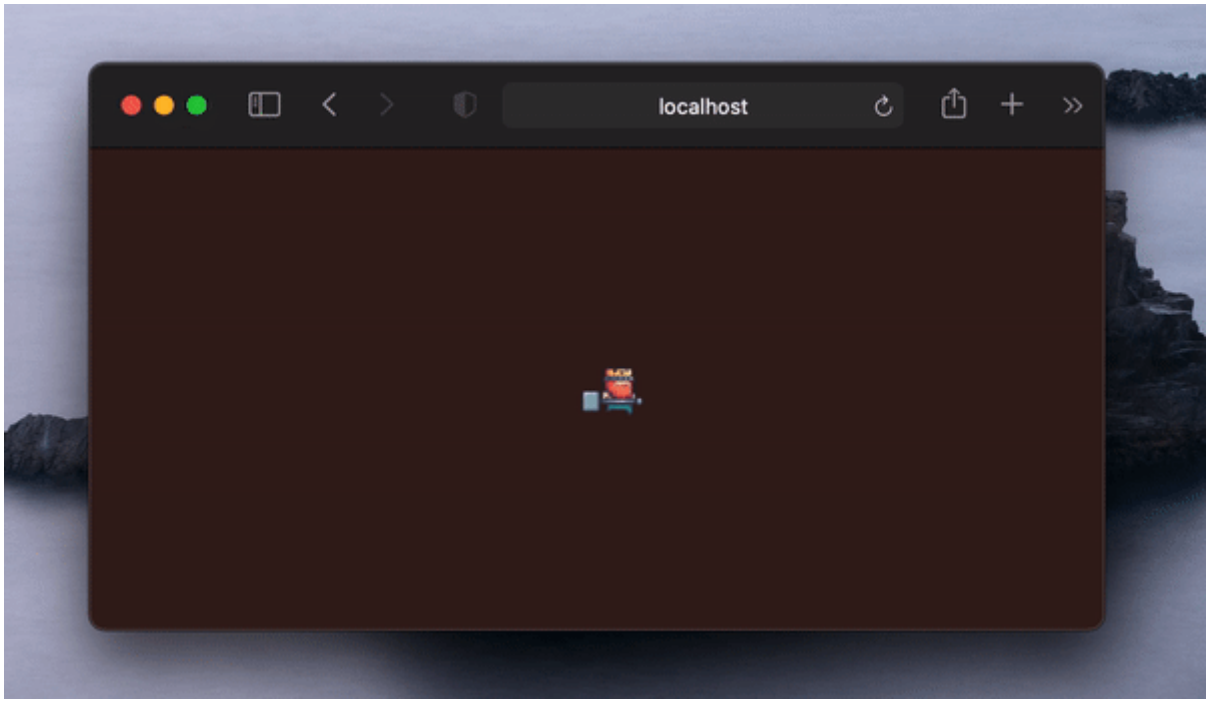
```
import { Scene } from 'phaser';
import { Player } from '../../classes/player';
export class Level1 extends Scene {
  private player!: Player;
  constructor() {
    super('level-1-scene');
  }
  create(): void {
    this.player = new Player(this, 100, 100);
  }
  update(): void {
    this.player.update();
  }
}
```

In the `update()` method of the scene, we specify for each frame to call the `update()` method on the player character, which in turn changes its position.

Time for a bit of running around

Run the dev server and try some running around with your character.

Excellent! The character moves and turns in the right direction 😎



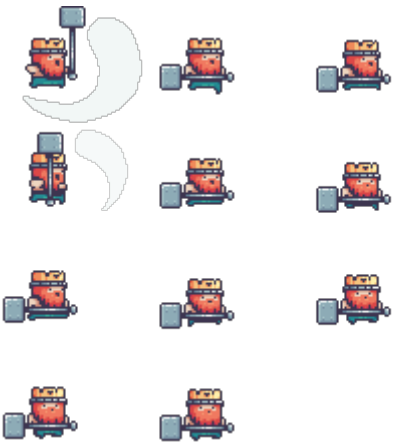
Part 4: **Sprite sheets and movement animation**

OK, our character can move now, but quite clumsily. Let's add a movement animation and figure out what sprite sheets are.

Sprite sheets and atlases

2D animation in Phaser is created using frames. Frames are sprites. **Sprite sheets (atlases)** are collections of sprites. The sprites in a sprite sheet are called **frames**. That is, you can create animation using frames from a sprite sheet.
For example, this is how our character's sprite looks like:

And this is the sprite sheet with frames which we'll use for animation:



But one sprite sheet is not enough. How can we indicate that such and such a frame is in such and such a position of the sprite sheet when a sprite sheet is just a picture? JSON comes to the rescue with information on each sprite: its name, width, height, and XY coordinates on the sprite sheet. Often, sprite sheets are generated automatically using online services or offline solutions, so you don't have to create JSON yourself and manually lay the sprites out on the grid. Also, some

generators add additional parameters to JSON like the `anchor` property, for example, which indicates the point at the origin of XY coordinates, where 0 is the beginning, 0.5 is the middle, and 1 is the end of the sprite. The `anchor` value can be negative, and not necessarily compliant with half sizes, that is, the value can be 1.23, or 0.99, or -3.33.

Let's take a look at the description of a frame:

src/assets/spritesheets/a-king_atlas.json

```
{
    "frames": [
        {
            "filename": "attack-0",
            "frame": {
                "w": 78,
                "h": 58,
                "x": 0,
                "y": 0
            },
            "anchor": {
                "x": 0.5,
                "y": 0.5
            }
        },
                ...
    ]
}
```

Here we can see that the frame called `attack-0` is located at 0-0 coordinates and has a width of 78px and a height of 58px.

Downloading the atlas

To use an atlas by its key it must be loaded like the rest of the assets.

Let's go to the Loading scene and load our atlas and its JSON file.

src/scenes/loading/index.ts

```
preload(): void {
    this.load.baseURL = 'assets/';
    // Our king texture
    this.load.image('king', 'sprites/king.png');
    // Our king atlas
    this.load.atlas('a-king', 'spritesheets/a-king.png', 'spritesheet
}
```

Now we can use our atlas in the right places with the `a-king` texture key.

Let's add running animation to our character! Go to the `player` class and describe the method for creating animation.

src/classes/player.ts

```
export class Player extends Actor {
...
    constructor(...) {
        ...
        this.initAnimations();
    }
...
    private initAnimations(): void {
    this.scene.anims.create({
      key: 'run',
      frames: this.scene.anims.generateFrameNames('a-king', {
        prefix: 'run-',
        end: 7,
      }),
      frameRate: 8,
    });
    this.scene.anims.create({
      key: 'attack',
      frames: this.scene.anims.generateFrameNames('a-king', {
        prefix: 'attack-',
        end: 2,
```

```
      }),
      frameRate: 8,
    });
    }
...
```

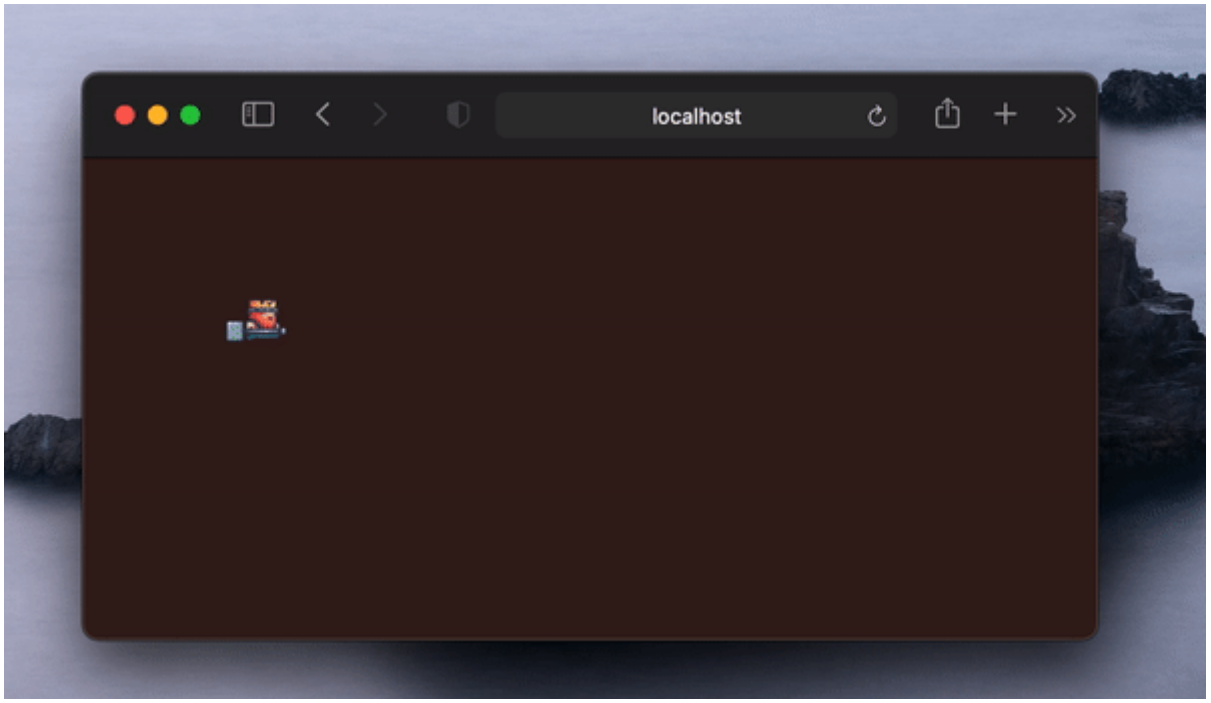
Here we indicate that we want to create an animation with such and such a key with the animation frames taken from the `a-king` atlas, by the prefix.
Great, we've created two animations: `run` and `attack`. Now let's add the running animation:

src/classes/player.ts

```
...
update(): void {
  this.getBody().setVelocity(0);
  if (this.keyW?.isDown) {
    this.body.velocity.y = -110;
        !this.anims.isPlaying && this.anims.play('run', true);
  }
  if (this.keyA?.isDown) {
    this.body.velocity.x = -110;
    this.checkFlip();
    this.getBody().setOffset(48, 15);
        !this.anims.isPlaying && this.anims.play('run', true);
  }
  if (this.keyS?.isDown) {
    this.body.velocity.y = 110;
        !this.anims.isPlaying && this.anims.play('run', true);
  }
  if (this.keyD?.isDown) {
    this.body.velocity.x = 110;
    this.checkFlip();
    this.getBody().setOffset(15, 15);
        !this.anims.isPlaying && this.anims.play('run', true);
  }
}
...
```

Here we check for each button whether the animation is currently playing, and if not, we play it with such and such `this.anims.play ('key', ignoreIfPlaying)` key.

Now let's start the dev-server and try running😎



Part 5: **Creating and loading a map, enabling collisions**

So, we managed to create a development environment from scratch, load assets, and place them on the stage, and we also added a character with the ability to move. Now let's make a location for him to be actually able to do that.

A little theory before practice

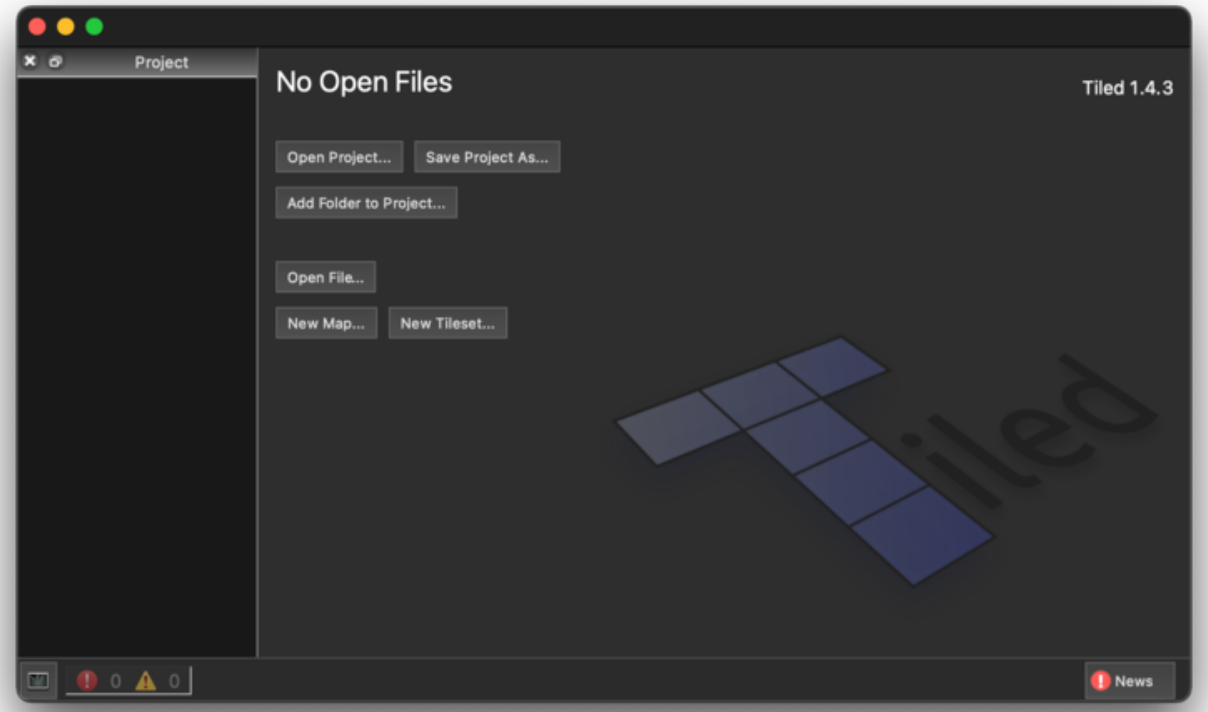
To create a location, we need the Tiled editor. It's free and its developers appreciate donations that help them make it bigger and better.

We also need assets — the so-called `tileset`. They are very similar to sprite sheets but are rigidly attached to the grid. Usually, tileset authors indicate the dimension of

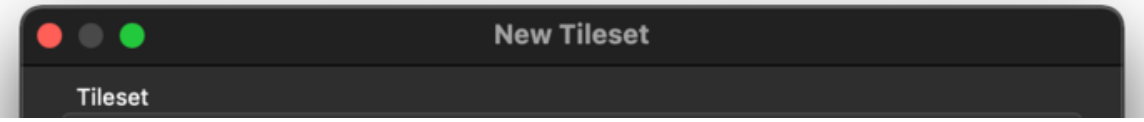
a grid. In the assets for this tutorial, we attached the necessary assets (**assets/tilemaps/tiles**), but you can also find some free packages available in the public domain, for example, on itch.io. Follow the example from the tutorial and create your own world.

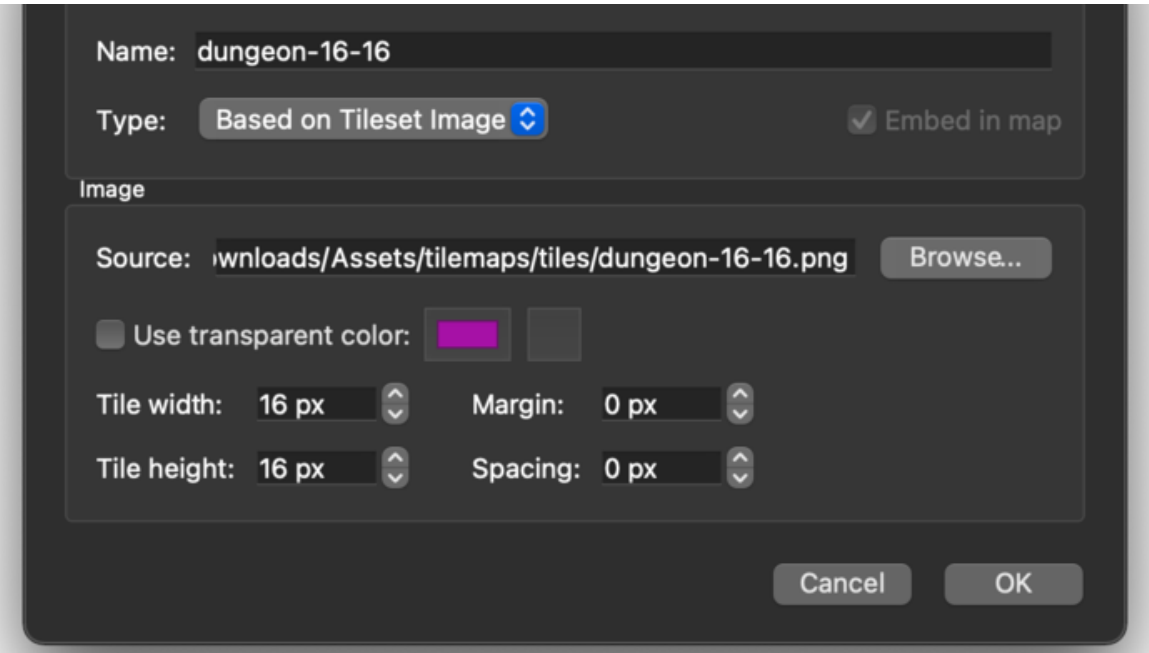
Preparing the editor

Open the Tiled editor and choose the **New Tileset** option.



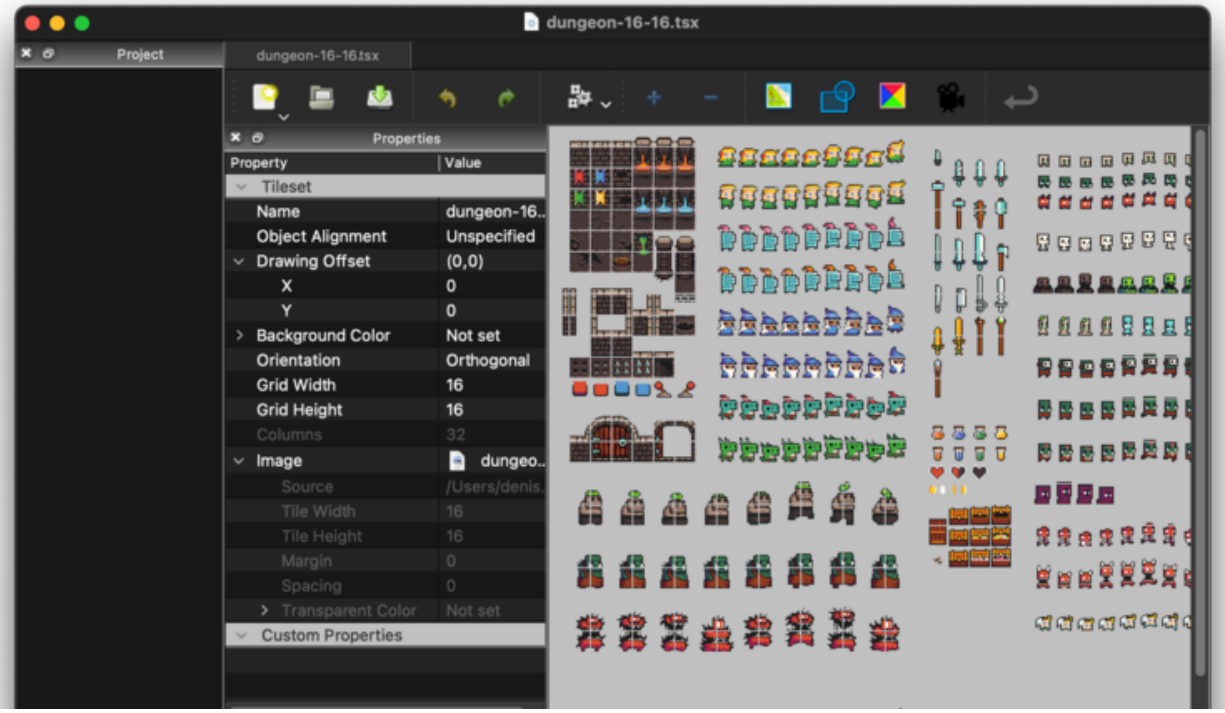
You'll see the menu for adding a tileset to a project. Select the tileset from the tutorial materials (**assets/tilemaps/tiles/dungeon-16-16)**. Now you need to specify the tile dimension (grid dimension): `Tile width` and `Tile height`. In our case, it's 16×16. Also, make sure that you check the **"Embed in map"** box. In case it cannot be installed or removed, it's okay, we'll return to it later.
Click OK.





Next, Tiled is going to ask you about the project file name and where to save it. The project file applies only to the Tiled editor, it's not used in the game, so we can save it wherever we want and under any name. Just don't forget to select the format of the saved file. We leave it to be the standard `.tsx`.

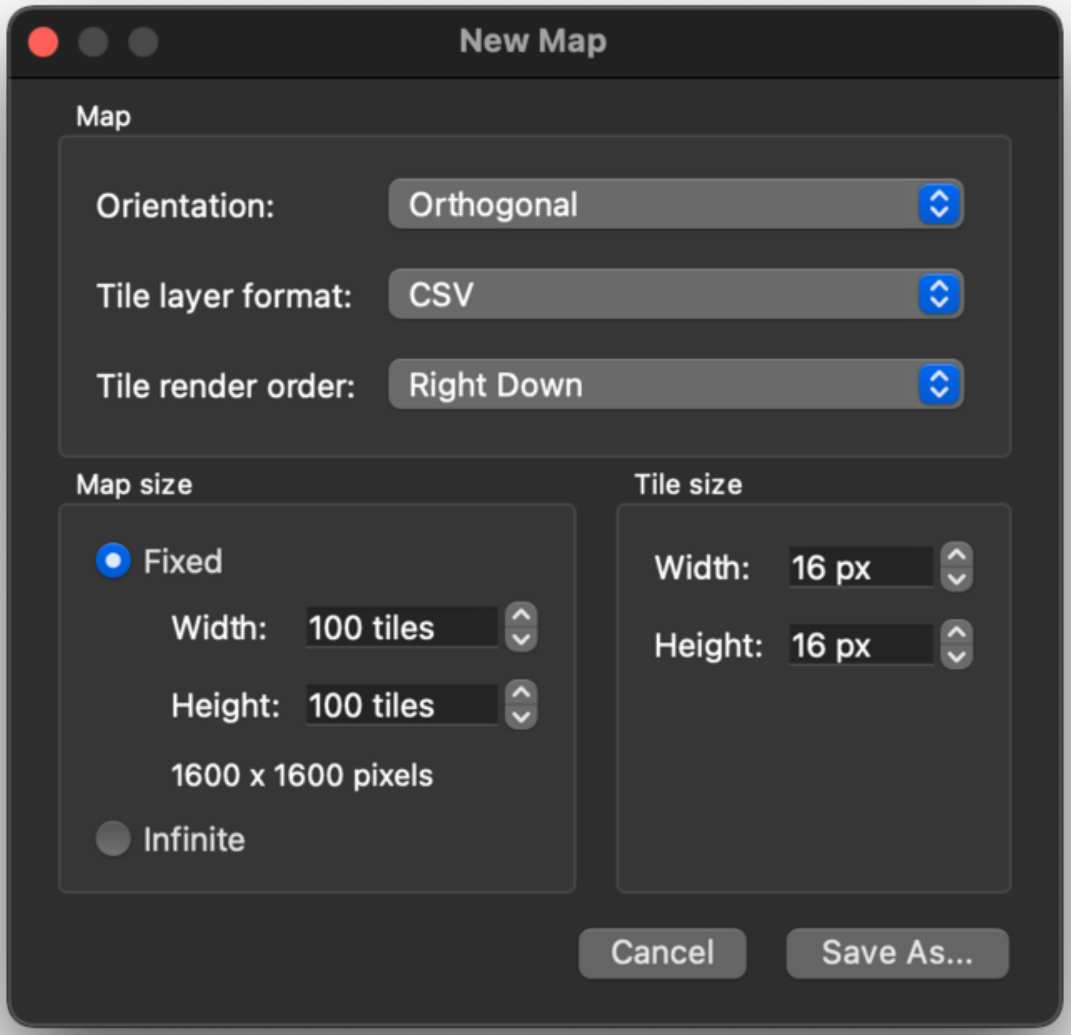
What we see before us now is the tile tab, where we can select any tile and edit it like adding any parameters, keys, values, names, etc. For now, let's leave it as it is, but we are definitely going to return here later.



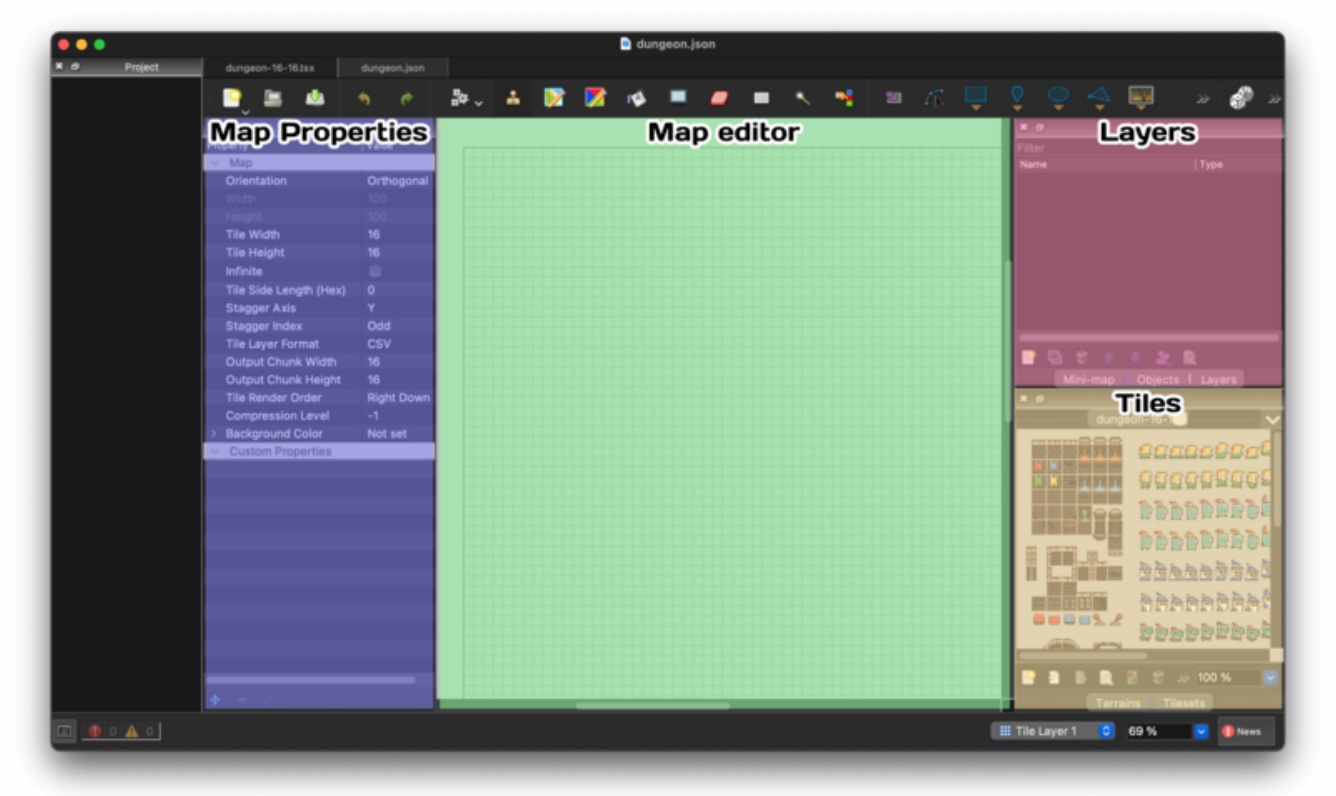
By adding a tileset — the material for creating the map — we can finally create the map itself!

To do this call the menu **File → New → New Map...**

We have a choice of map parameters before us. We are going to create a map with standard settings for the width and height (**Map size**), measured in tiles, and set the sizes of our tiles (**Tile size**) according to our tileset. As our tileset has a grid of 16×16 pixels, so in this window, we set the Tile size to 16×16 pixels.



Great! We now have a map editing tab:



Let's analyze the areas:

- `Map Properties` — these are map settings that you can edit in case you make a mistake in the **New Map...** menu.
- `Map editor` — it's the map grid where we'll place our tiles. That is, it's like a canvas for drawing a map with tiles🙂
- `Layers` — these are map layers. There can be many of them and they can be of different types. For example: on the Ground layer we'll mark the places where our character will move, and on the Walls layer we'll place walls and obstacles through which he won't be able to pass. Splitting into layers is a good practice since it is both more convenient for creating and editing a map, as well as for a more handy selection of the necessary layers for using some events in the game specifically with these layers.
- `Tiles` — this is the tileset window of the tileset we loaded earlier.

Creating a location

So let's create a location already!

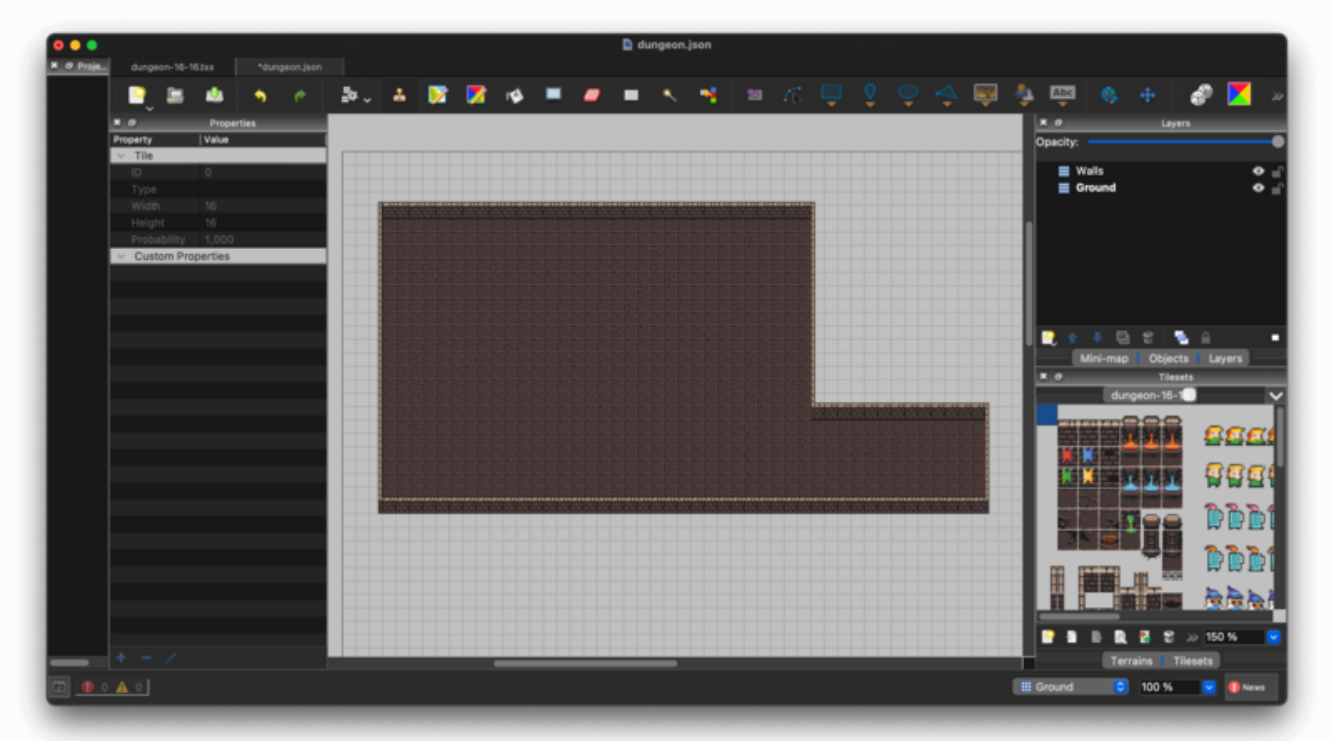
To do this, in the layers area, select the `Layers` tab, then select the standard layer and rename it. In this tutorial, we'll call it **Ground**, but you can choose any name.

Now select the tiles from the `Tiles` area and place them on the map, like putting brush to canvas.

You can use any of the tools from the toolbar above the `Map editor`. Stamp Brush, Shape Fill Tool, and Eraser seem to be the most popular ones.

Now let's add another layer with walls and name it **Walls**.

And here's our first map!



That's enough for a start. Let's transfer the map into the game.

Exporting and loading a map into a game

Save the Tiled project.

Then go to **File → Export As...**

You'll see the window for saving the map file. Select the **JSON map files** format and save. Let's do it under the project name — **dungeon**.
Create folders **src/assets/tilemaps/tiles/** for tilesets files (.png images) and **src/assets/tilemaps/json/** for json map files, where we'll transfer our map files.
As a result, our **src/assets/** folder will look like this:

```
├── sprites
│   └── king.png
├── spritesheets
│   ├── a-king.png
│   └── a-king_atlas.json
└── tilemaps
    ├── json
    │   └── dungeon.json
    └── tiles
        └── dungeon-16-16.png
```

Next, load the map using Phaser. To do this, go to the Loading scene, where we load all the assets, and add the following:

```
this.load.image({
  key: 'tiles',
  url: 'tilemaps/tiles/dungeon-16-16.png',
});
this.load.tilemapTiledJSON('dungeon', 'tilemaps/json/dungeon.json');
```

So, first, we load the texture, our tileset, and then load the JSON map file, which stores information about the location of each tile and the way it's divided into layers.
Go to the scene of the first level, and define our map to display it.
Let's create a function for initializing the map, and describe the property types:

```
private map!: Tilemaps.Tilemap;
private tileset!: Tilemaps.Tileset;
```

```
private wallsLayer!: Tilemaps.DynamicTilemapLayer;
private groundLayer!: Tilemaps.DynamicTilemapLayer;
...
private initMap(): void {
  this.map = this.make.tilemap({ key: 'dungeon', tileWidth: 16, tileH
  this.tileset = this.map.addTilesetImage('dungeon', 'tiles');
  this.groundLayer = this.map.createDynamicLayer('Ground', this.tiles
  this.wallsLayer = this.map.createDynamicLayer('Walls', this.tileset
  this.physics.world.setBounds(0, 0, this.wallsLayer.width, this.wall
}
```

Here, we define tilemap `this.map` with so-and-so a key and such-and-such width and height tile parameters. After that, we define the connection between `this.tileset` of the texture and the JSON file of the map (we indicate the keys that we used while loading on the Loading scene). Next, describe the `this.groundLayer` and `this.wallsLayes` layers where the first argument is the name of the layer, the second one is the map information, i.e. `this.tileset`, and the arguments number 3 and 4 are the XY coordinates relative to the world where we start drawing the map. After describing all the information about the map and layers, we set the size of the physical world to `this.physics.world.setBounds()`, where we state that we start counting at so-and-so XY coordinates, and set the width and height of the world to the similar to the wall layer.
It only remains to call our function at the very beginning of the `create()` method of the first level scene:

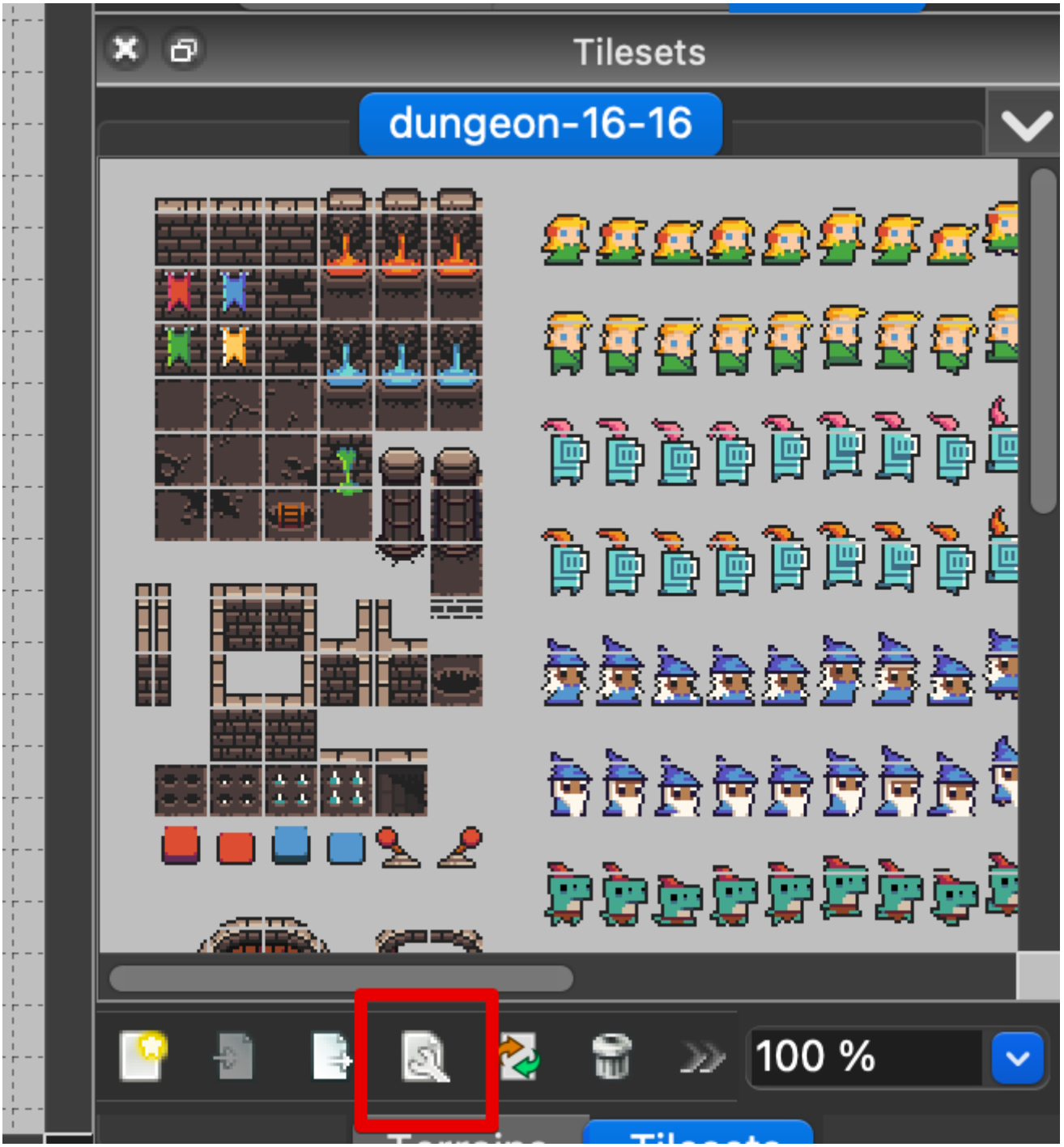
```
...
create(): void {
  this.initMap();
  this.player = new Player(this, 100, 100);
}
...
```

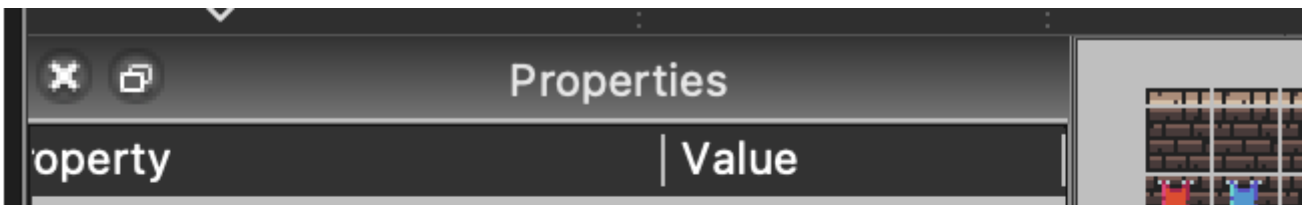
Let's start the dev-server to see that... Our location isn't there, and in the console, we see a warning from Phaser that the tileset "dungeon" could not be loaded. Don't

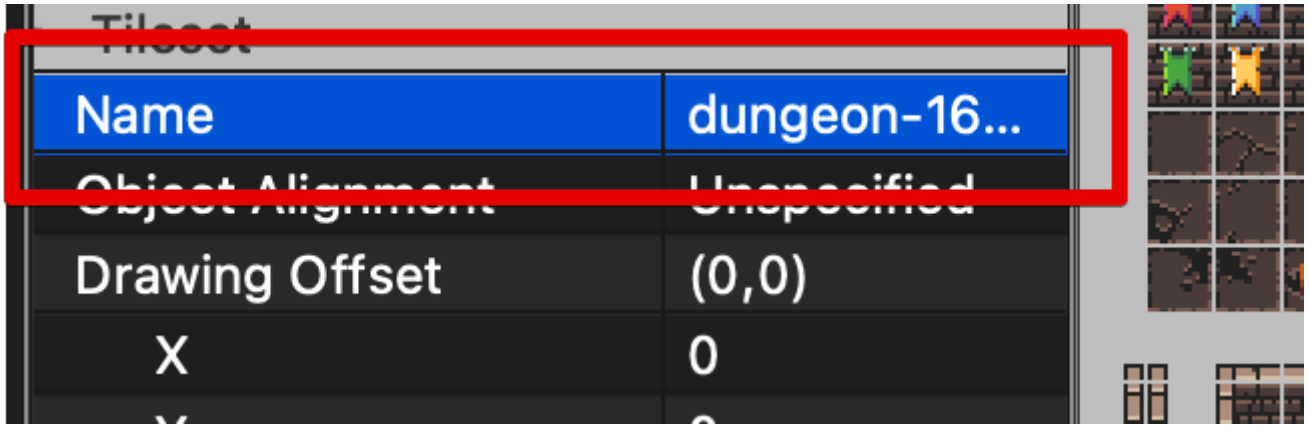
worry, everything is just as it should be!

This is because when loading the tileset into Tiled, we didn't change its name. And if you did everything according to the tutorial, now the map inside the JSON code is trying to associate itself with a tileset named **"dungeon-16-16"**. Let's fix it!

In the tilesets area, click on the settings icon for this tileset, and in the window that opens, change the **Name** property from **"dungeon-16-16"** to **"dungeon"**. Save.





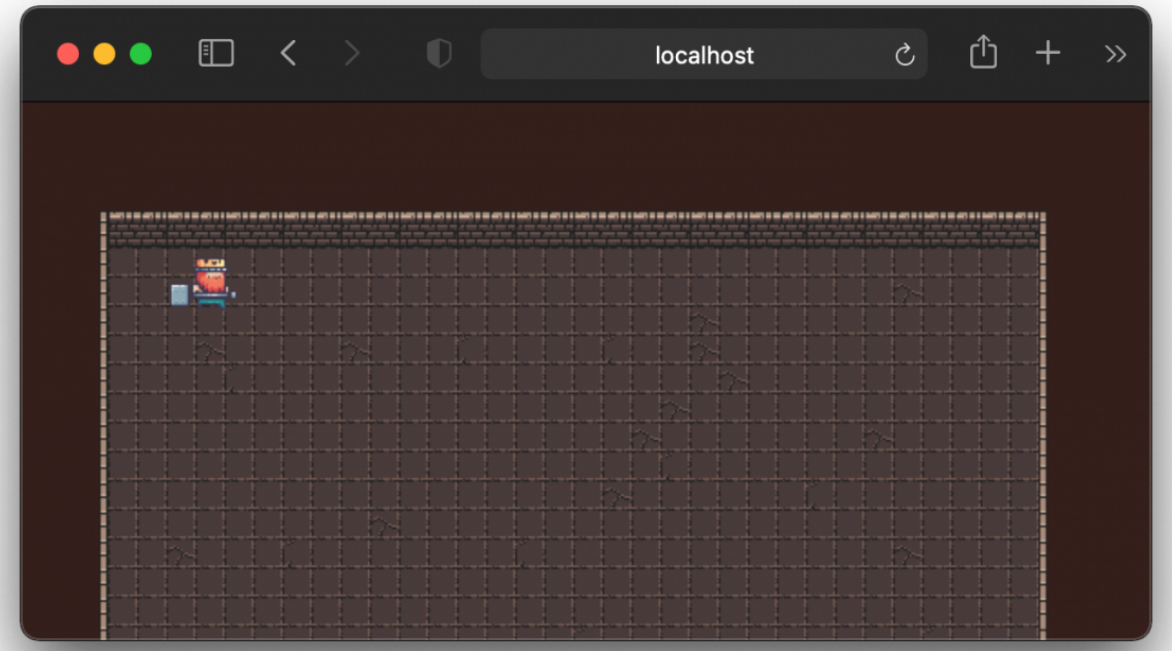


Also, make sure that in the tileset area and with the dungeon selected, the “Embed Tileset” button is not active. If it is active, click on it.

Now let’s re-export our JSON file. **File -> Export As ... -> File format .json.**

Transfer the new JSON file into the project and start the dev-server for verification.

Voila! We see the map!

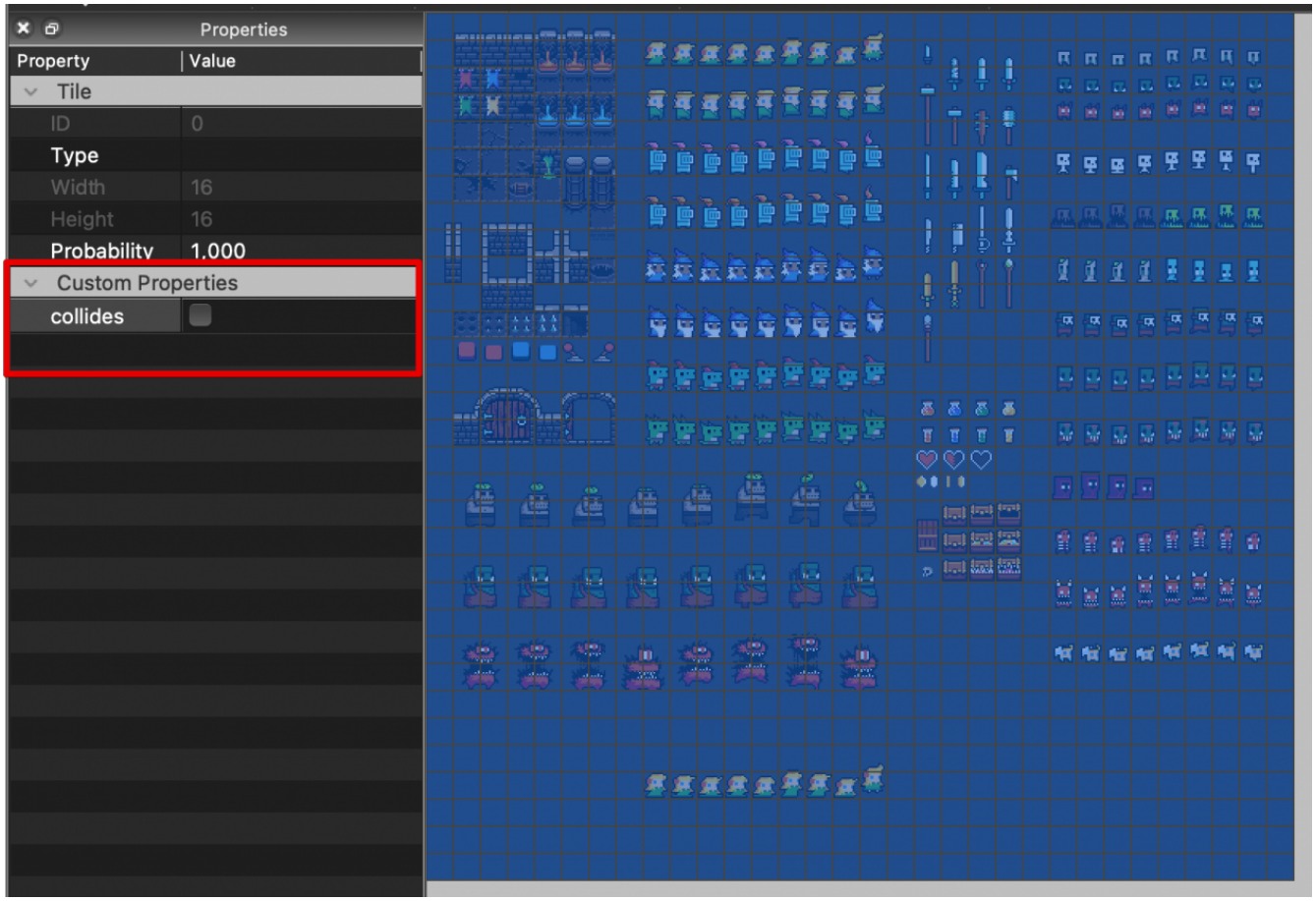


Collisions and walls

Let's try running around the map. As you can see, the character does not react to the walls, as if the whole map is just a background picture. To fix this, we need to set collisions on certain tiles.

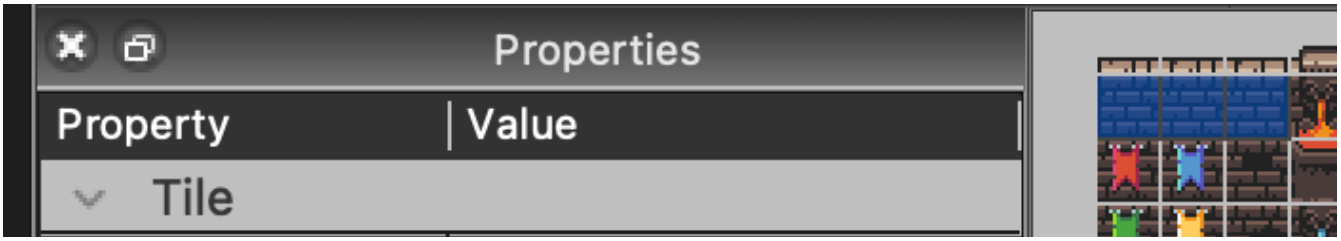
It's very convenient to do everything there in the Tiled editor.

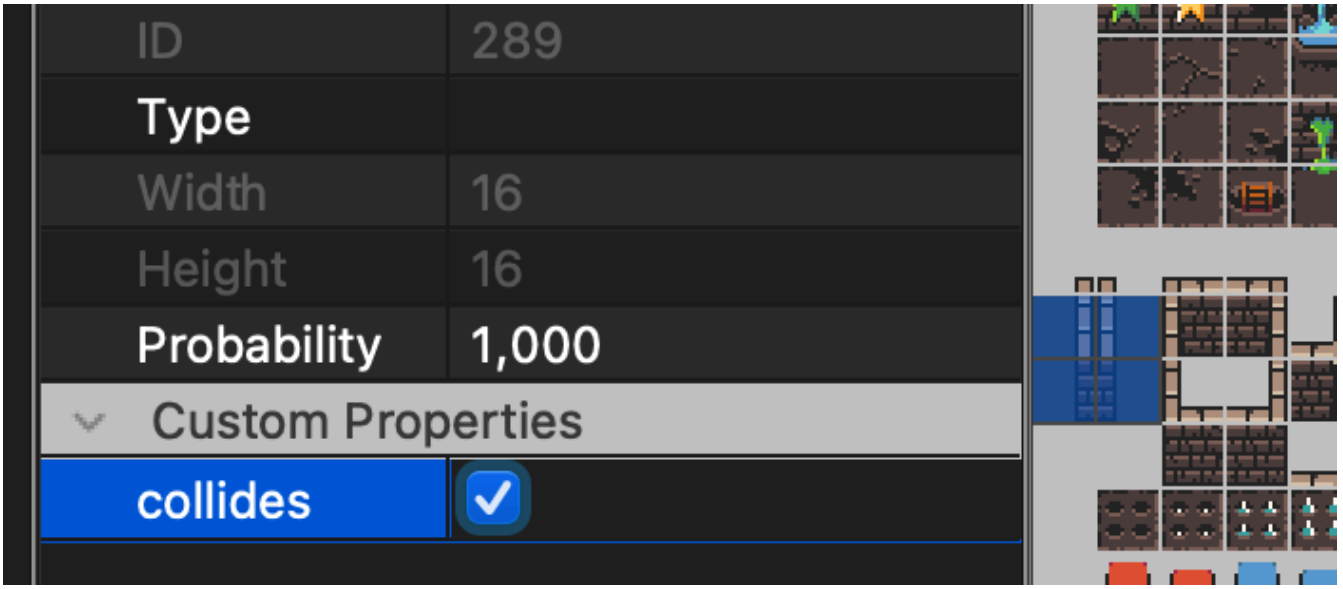
Open up our **"dungeon"** tileset and select all the tiles. Further, on the left, there are **Properties**. In the **"Custom Properties"** area, right-click → **Add Property** → choose any property name (we going to use **"collides"**), the data type is **bool** → OK.



There should appear a new field with the option to check the box. It might happen so that the created field is of the wrong type. In such a case, just delete it.

Now that all the tiles have the **collides** property, select only those tiles that cannot be passed through, and check the **collides** checkbox.





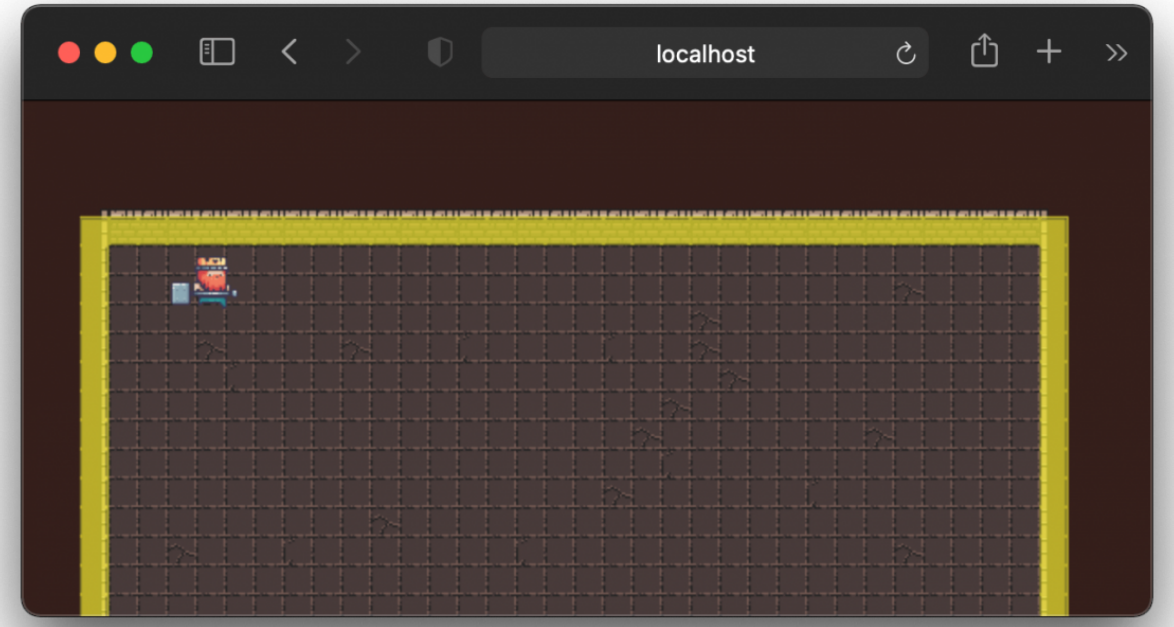
Save the project, export the **dungeon.json**, and transfer it into the game assets. Now, go to the scene of the first level and to the map creation function and enable collisions on tiles that we want to select by property:

```
...
this.wallsLayer.setCollisionByProperty({ collides: true });
...
```

To see if collisions have been applied OK to the walls, let's write a short helper function that will simply show us debug information and highlight the collision walls.

```
...
private initMap(): void {
    ...
    this.showDebugWalls();
}
...
private showDebugWalls(): void {
  const debugGraphics = this.add.graphics().setAlpha(0.7);
  this.wallsLayer.renderDebug(debugGraphics, {
    tileColor: null,
    collidingTileColor: new Phaser.Display.Color(243, 234, 48, 255),
  });
}
```

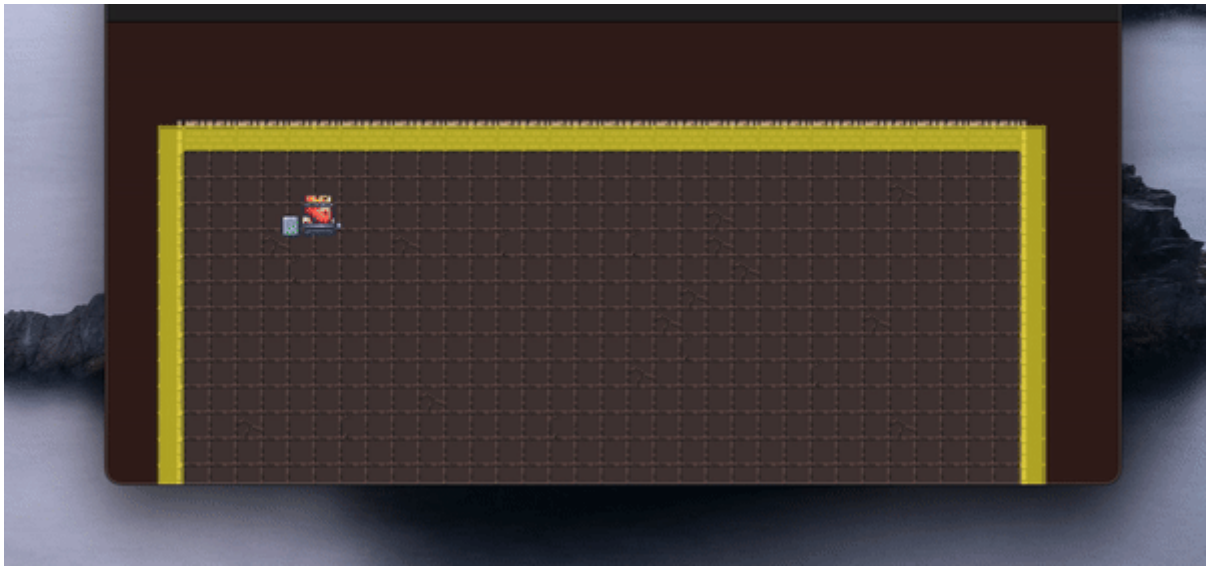
Start the dev-server and you'll now see exactly which walls have collisions.



But wait! Our character is still walking through the walls, what's the matter?
The fact is, we didn't indicate that the character should react to the walls in some way. Let's fix this!
On the first level stage, in the `create()` method, after initializing the map and the player, add the so-called "collider", a method that includes collisions between two objects:

```
...
create(): void {
  this.initMap();
  this.player = new Player(this, 100, 100);
  this.physics.add.collider(this.player, this.wallsLayer);
}
...
```

That's it, our character is no longer able to pass through the walls:



Now you can create your own locations, load them into the game, and identify impassable areas.
Try to improve your location, or expand it. Try different tiles or even choose a tileset for your own idea, the world is yours😎

Part 6: **Adding objects to the map. Camera**

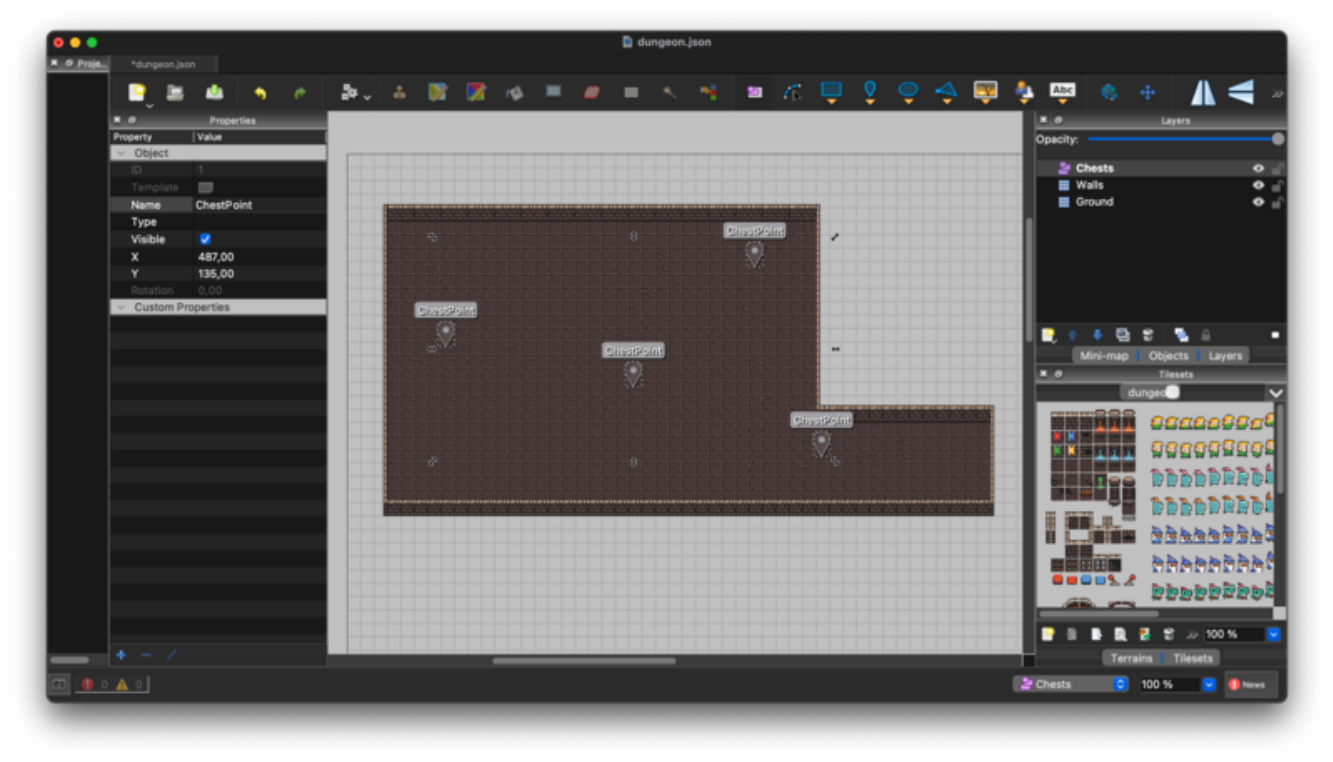
With the help of Tiled, we can not only build a map but also add points to show where to place the objects.
How about rewarding our character with chests? Let's create it all in the same old Tiled.

Putting object points on the map

To place objects in Tiled, we need to first create a separate layer for them. Go to the Layers area → Right-click on the empty space → **New → Object Layer**. Let's call it **"Chests"**.
Now select "Insert Point" on the toolbar and place points where you want the

objects to be.

Select all the points, and on the left side, in the object properties, set the **“Name”** property to **“ChestPoint”**.



Export the JSON and go to the first level scene.

Placing chests on points

Now we can place any object on any point. In our case, it’s chests.

Due to errors with Phaser types, it’s better to prepare a small helper adapter in **src/helpers/gameobject-to-object-point.ts**, with which we’ll translate the GameObject type to ObjectPoint.

```
export const gameObjectsToObjectPoints = (gameObjects: unknown[]): Ob
  return gameObjects.map(gameObject => gameObject as ObjectPoint);
};
```

Let’s define the ObjectPoint type in index.d.ts:

```
type ObjectPoint = {
  height: number;
  id: number;
  name: string;
  point: boolean;
  rotation: number;
  type: string;
  visible: boolean;
  width: number;
  x: number;
  y: number;
};
```

Also, we'll need to separately load our tileset as a sprite sheet so that we can take certain sprites from there (note, *the sprites*, not the tiles). To do this, go to the Loading scene and add the loading of the sprite sheet with the **tiles_spr** key (the name can be anything), specifying the dimension of the sprites:

```
...
this.load.spritesheet('tiles_spr', 'tilemaps/tiles/dungeon-16-16.png'
  frameWidth: 16,
  frameHeight: 16,
});
...
```

Now, on the first level stage, we'll define an array of chests, add a function that creates chests and call it after initialization of the character:

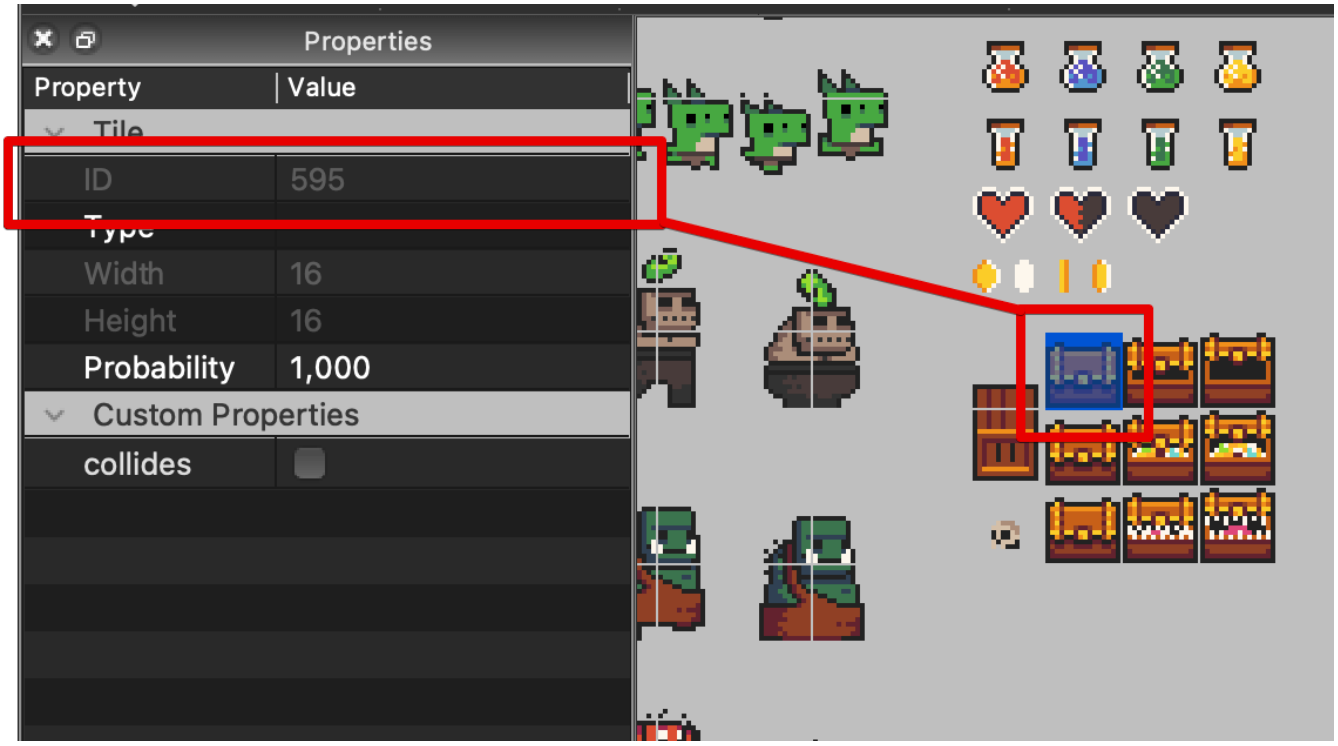
```
private chests!: Phaser.GameObjects.Sprite[];
...
...
private initChests(): void {
  const chestPoints = gameObjectsToObjectPoints(
    this.map.filterObjects('Chests', obj => obj.name === 'ChestPoint'
  );
  this.chests = chestPoints.map(chestPoint =>
```

```
    this.physics.add.sprite(chestPoint.x, chestPoint.y, 'tiles_spr',
  );
  this.chests.forEach(chest => {
    this.physics.add.overlap(this.player, chest, (obj1, obj2) => {
      obj2.destroy();
      this.cameras.main.flash();
    });
  });
}
```

Using the `this.map.filterObjects()` function, select the required objects from the required layer. The first argument is the layer's name, the second one is the callback function for filtering. In our case, if the object has the name **"ChestPoint"**, then it's suitable, and will fall into the array of the same **chestPoints** objects.

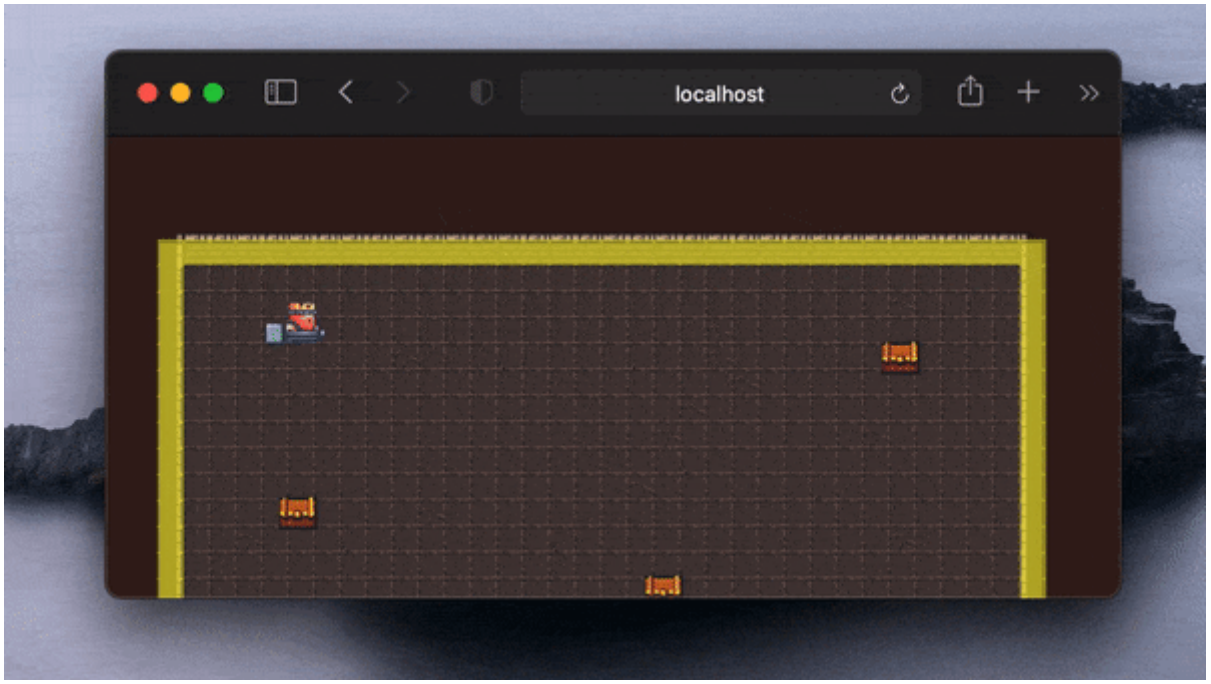
Next, let's create a sprite with a physical model for each point from the **chestPoints** array.

595 is the sprite's ID in the sprite sheet, but how to find it? It's simple! Open the tileset in the Tiled editor, select the chest, and on the left side, you'll see its ID:



Also, we made sure that when a character "steps" on a chest, there will be a flash, and the chest will disappear as if it had been taken.

Let's start the dev server and check it:



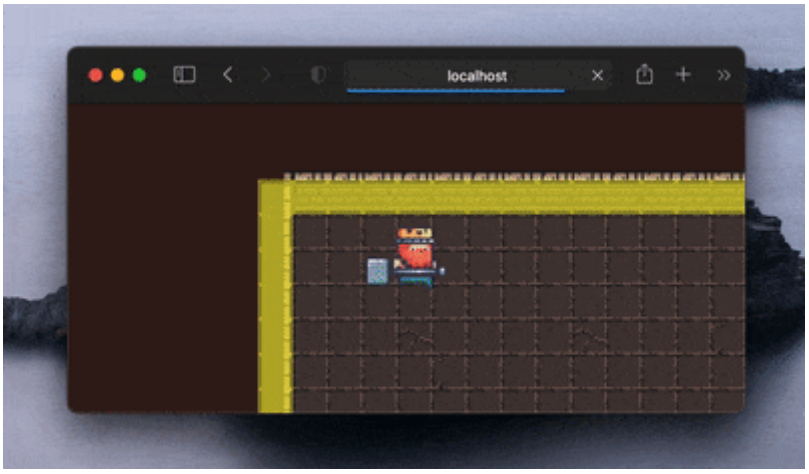
Cameras and following the player character

Cool, we can now collect chests! But what to do with those chests that are on the map, but that we don't see? Time to work with the map.
Each scene has a main camera already, and this is how it is being accessed — through `this.cameras.main`. Also, we can create our own cameras. For this tutorial, the main camera is enough, but if you want to learn more about cameras, be sure to look into these examples.
Let's add a function, where we'll call several methods for the camera and call the function after all initializations in the `create()` method:

```
private initCamera(): void {
  this.cameras.main.setSize(this.game.scale.width, this.game.scale.he
  this.cameras.main.startFollow(this.player, true, 0.09, 0.09);
  this.cameras.main.setZoom(2);
}
```

Here we have set the camera size, zoom, and the target that the camera will follow.

`0.09` is the value for the smooth movement of the camera towards the target position, thus we made the movement of the camera smoother.



Part 7: **Text, event system, counter**

We have already added chests to the map, but so far they don't give out anything, they only disappear. Let's create a score counter and link it to the moment of the chest collection.

Let's start small — with the text. So, create a `Text` class, which will be common for all text labels.

src/classes/text.ts

```
import { GameObjects, Scene } from 'phaser';
export class Text extends GameObjects.Text {
  constructor(scene: Scene, x: number, y: number, text: string) {
    super(scene, x, y, text, {
      fontSize: 'calc(100vw / 25)',
      color: '#fff',
      stroke: '#000',
      strokeThickness: 4,
    });
    this.setOrigin(0, 0);
    scene.add.existing(this);
  }
}
```

Now we'll create a `Score` class for the counter, which will inherit from our main `Text` class, where we'll define a method for changing the value depending on the operation.

src/classes/score.ts

```
import { Text } from './text';
export enum ScoreOperations {
  INCREASE,
  DECREASE,
  SET_VALUE,
}
export class Score extends Text {
  private scoreValue: number;
  constructor(scene: Phaser.Scene, x: number, y: number, initScore =
    super(scene, x, y, `Score: ${initScore}`);
    scene.add.existing(this);
    this.scoreValue = initScore;
  }
  public changeValue(operation: ScoreOperations, value: number): voic
    switch (operation) {
      case ScoreOperations.INCREASE:
        this.scoreValue += value;
        break;
      case ScoreOperations.DECREASE:
        this.scoreValue -= value;
        break;
      case ScoreOperations.SET_VALUE:
        this.scoreValue = value;
        break;
      default:
        break;
    }
    this.setText(`Score: ${this.scoreValue}`);
  }
  public getValue(): number {
    return this.scoreValue;
  }
}
```

Now we need to create a separate scene for the UI elements, which will be on top of the main game and levels.

And immediately add there our counter:

src/scenes/ui/index.ts

```
import { Scene } from 'phaser';
import { Score, ScoreOperations } from '../../classes/score';
export class UIScene extends Scene {
  private score!: Score;
  constructor() {
    super('ui-scene');
  }
  create(): void {
    this.score = new Score(this, 20, 20, 0);
  }
}
```

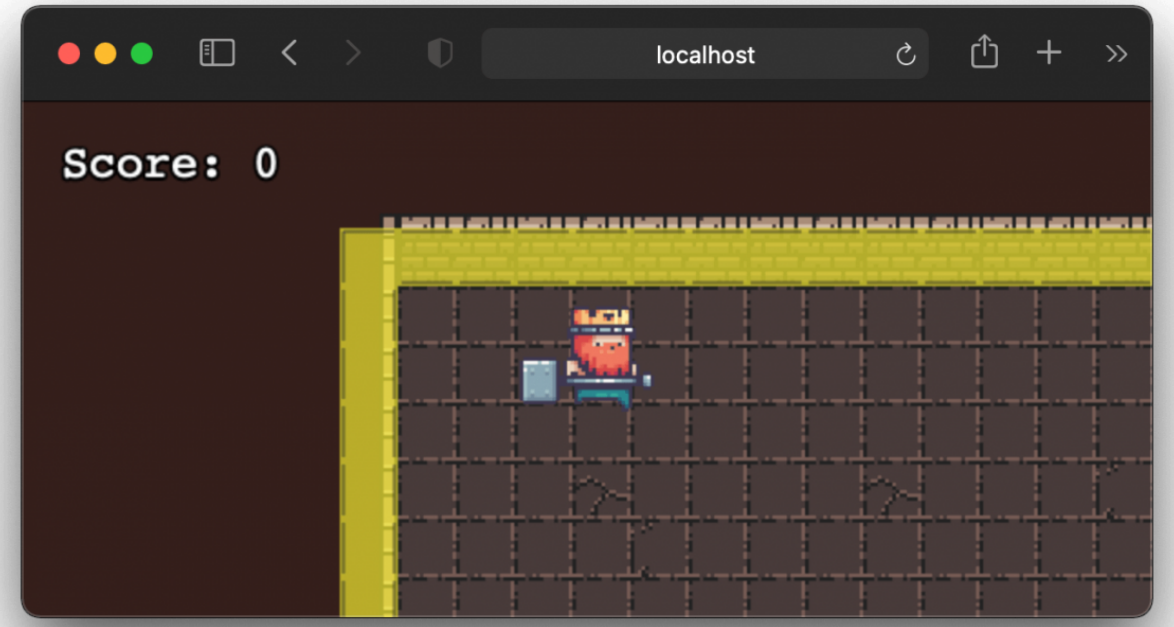
Also, don't forget to add the UI scene to the `gameConfig`. We'll launch it together with the level from the Loading scene.

src/index.ts

```
...
scene: [LoadingScene, Level1, UIScene],
...
```

src/scenes/loading/index.ts

```
...
create(): void {
    this.scene.start('level-1-scene');
  this.scene.start('ui-scene');
}
...
```



Event system

Great, we've added a counter, but right now it doesn't respond to collecting chests. Let's establish this connection using the event system, and send the corresponding event from one scene to another, at the moment when we pick up the chest.
In this tutorial, we'll not have many events, but in order not to remember the names of events, it is considered a good practice to store events in constants.
Let's create a file for these constants, and add only one event there so far — the looting.

src/consts.ts

```
export enum EVENTS_NAME {
  chestLoot = 'chest-loot',
}
```

To work with an event, we must trigger it and describe its handler somewhere. Let's

add this event trigger to the moment when the game character comes into contact with the chest. At the moment, this action triggers the destruction of the chest and the flash of the camera.

src/scenes/level-1/index.ts

```
import { EVENTS_NAME } from '../../consts';
...
private initChest(): void {
    ...
    this.chests.forEach(chest => {
    this.physics.add.overlap(this.player, chest, (obj1, obj2) => {
      this.game.events.emit(EVENTS_NAME.chestLoot);
      obj2.destroy();
      this.cameras.main.flash();
    });
  });
    ...
}
...
```

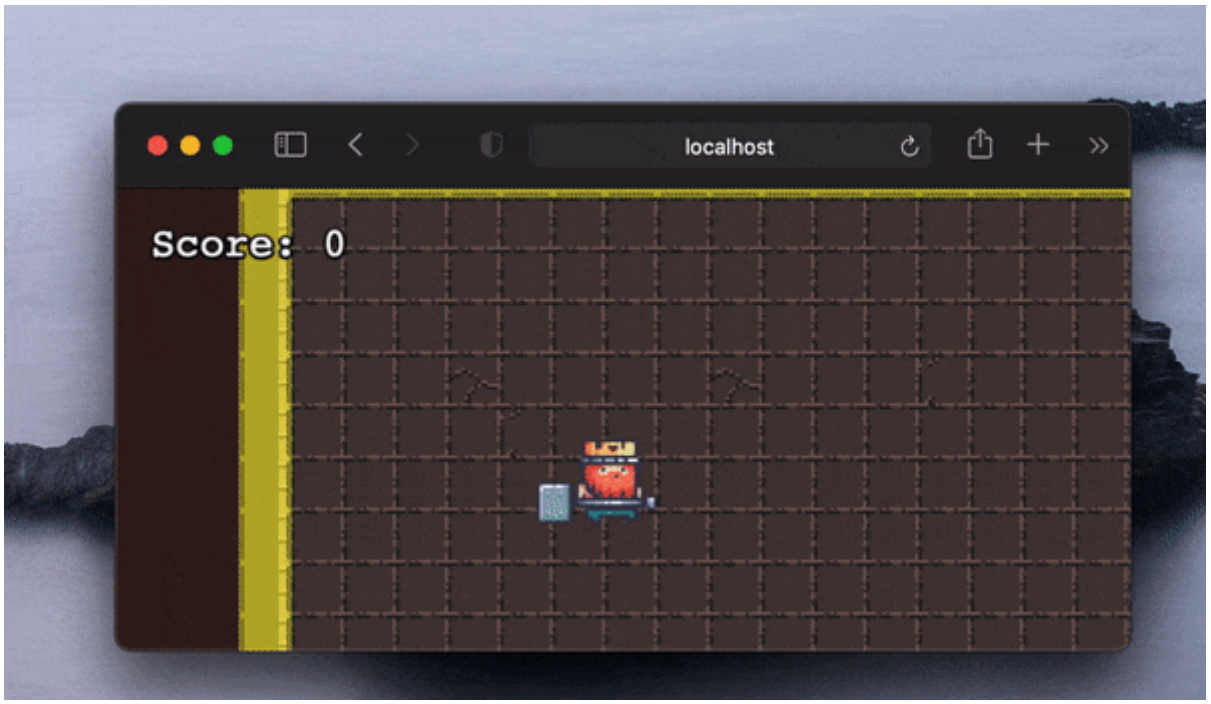
Now let's go to the UI scene, declare a listener for the looting event, and also describe its handler, which will change the counter value when we pick up the chest.

src/scenes/ui/index.ts

```
import { EVENTS_NAME } from '../../consts';
...
private chestLootHandler: () => void;
...
constructor() {
    ...
    this.chestLootHandler = () => {
    this.score.changeValue(ScoreOperations.INCREASE, 10);
  };
}
...
create(): void {
```

```
    ...
    this.initListeners();
}
...
private initListeners(): void {
  this.game.events.on(EVENTS_NAME.chestLoot, this.chestLootHandler, t
}
```

Now the counter responds to the collection of chests and increases the counter.



Part 8: **Bots, game end screen**

So we are now in the final part of our Phase 3 game tutorial. We already have a character to explore the locations, and chests that we get game points for. But there's not enough action, there's not enough of what could create a desire to win — **challenge**. At the moment, our main character lives in greenhouse conditions and aimlessly collects chests. To create a more enjoyable gaming experience, you need to set goals and some obstacles on the way — enemies.

In this part, we'll place enemies on the map and add logic to them so that they attack our hero if he gets too close.

Also, we'll add the win/lose logic, and depending on the status of the game completion, show the corresponding screen with the corresponding message.

Enemies

To begin with, let's create a class of enemies, which we'll use to "populate" the location with enemies. The enemy class will be similar to the player's class but without keyboard control.

src/classes/enemy.ts

```
import { Math, Scene } from 'phaser';
import { Actor } from './actor';
import { Player } from './player';
export class Enemy extends Actor {
  private target: Player;
  private AGRESSOR_RADIUS = 100;
  constructor(
    scene: Phaser.Scene,
    x: number,
    y: number,
    texture: string,
    target: Player,
    frame?: string | number,
  ) {
    super(scene, x, y, texture, frame);
    this.target = target;
    // ADD TO SCENE
    scene.add.existing(this);
    scene.physics.add.existing(this);
    // PHYSICS MODEL
    this.getBody().setSize(16, 16);
    this.getBody().setOffset(0, 0);
  }
  preUpdate(): void {
    if (
      Phaser.Math.Distance.BetweenPoints(
```

```
        { x: this.x, y: this.y },
        { x: this.target.x, y: this.target.y },
      ) < this.AGRESSOR_RADIUS
    ) {
      this.getBody().setVelocityX(this.target.x - this.x);
      this.getBody().setVelocityY(this.target.y - this.y);
    } else {
      this.getBody().setVelocity(0);
    }
  }
  public setTarget(target: Player): void {
    this.target = target;
  }
}
```

Define a constant value of the radius for the enemy, i.e. at what distance the target must be for the enemy to start pursuing it.

```
...
private AGRESSOR_RADIUS = 100;
...
```

Also, add the ability to change the target. We'll not be using this function in this tutorial, but after completing it, you'll be able to upgrade the game by adding several extra characters.

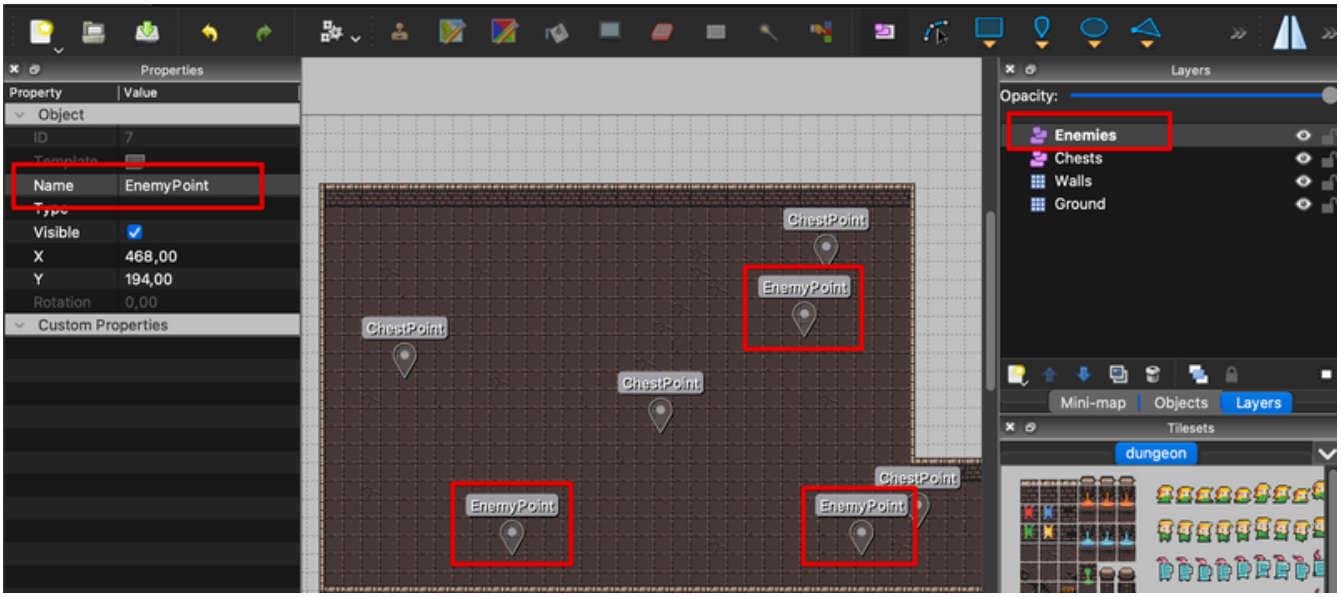
```
...
public setTarget(target: Player): void {
  this.target = target;
}
...
```

Pay attention to the `preUpdate()` function, this is a built-in function for classes like `GameObjects`, and it gets triggered every time before the game frames are re-

rendered. Here we describe the “brains” of our enemy. They will appear to be waiting until the target comes close enough to start moving towards it:

```
...
preUpdate(): void {
  if (
    Math.Distance.BetweenPoints(
      { x: this.x, y: this.y },
      { x: this.target.x, y: this.target.y },
    ) < this.AGRESSOR_RADIUS
  ) {
    this.getBody().setVelocityX(this.target.x - this.x);
    this.getBody().setVelocityY(this.target.y - this.y);
  } else {
    this.getBody().setVelocity(0);
  }
}
...
```

The enemy class is there, now let’s recall the information about adding chests to the location — on a separate **Enemies** layer. In the same way, add a couple of enemies to the first level scene.

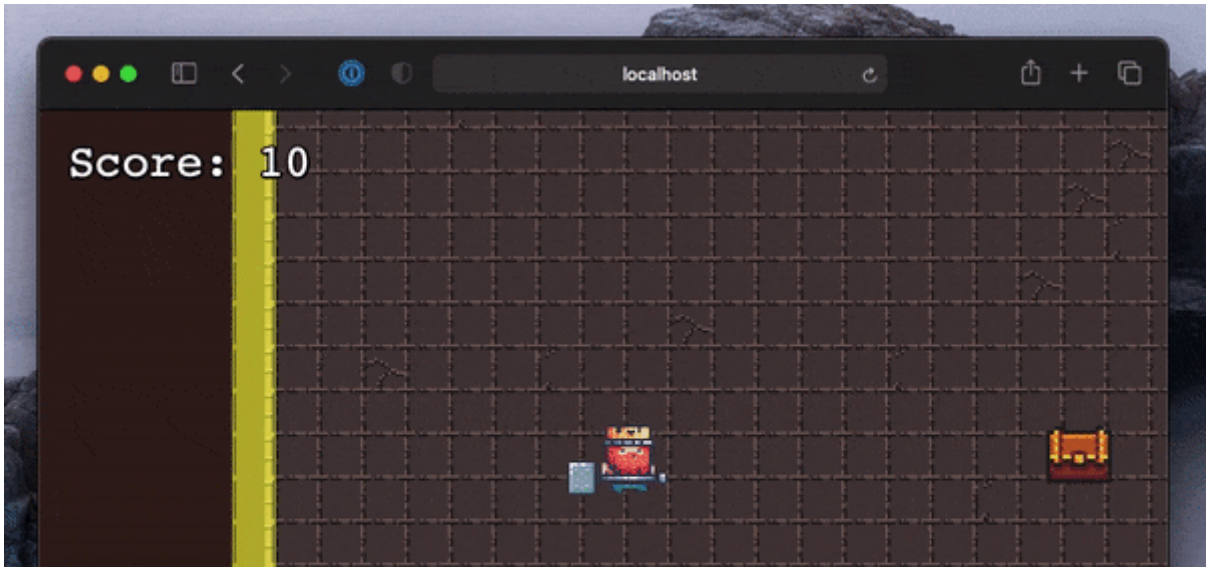


src/scenes/level-1/index.ts

```
...
create() {
    ...
    this.initEnemies();
    ...
}
...
private initEnemies(): void {
  const enemiesPoints = gameObjectsToObjectPoints(
    this.map.filterObjects('Enemies', (obj) => obj.name === 'EnemyPoi
  );
  this.enemies = enemiesPoints.map((enemyPoint) =>
    new Enemy(this, enemyPoint.x, enemyPoint.y, 'tiles_spr', this.pla
      .setName(enemyPoint.id.toString())
      .setScale(1.5),
  );
  this.physics.add.collider(this.enemies, this.wallsLayer);
  this.physics.add.collider(this.enemies, this.enemies);
  this.physics.add.collider(this.player, this.enemies, (obj1, obj2) =
    (obj1 as Player).getDamage(1);
  });
}
```

For each enemy, specify that they should not go through walls, the character, and through other enemies.

Also, describe that when in contact with the character, the enemy will inflict damage to the player in the amount of 1 HP (without additional restrictions, they will inflict damage very quickly).



Now let's display an HP value above the character's head. To do this, add the `Player` class.

src/classes/player.ts

```
import { Text } from './text';
...
private hpValue: Text;
...
this.hpValue = new Text(this.scene, this.x, this.y - this.height, thi
  .setFontSize(12)
  .setOrigin(0.8, 0.5);
...
update() {
    ...
    this.hpValue.setPosition(this.x, this.y - this.height * 0.4);
  this.hpValue.setOrigin(0.8, 0.5);
}
...
public getDamage(value?: number): void {
  super.getDamage(value);
  this.hpValue.setText(this.hp.toString());
}
```

Now the HP value will be shown above the game character, even when he moves, since in the `update()` method we update the position for the text value of HP. We've also extended the standard `getDamage()` method of the `Actor` superclass to change the HP value.

When the enemy starts attacking the character, we can see how quickly the damage is dealt. So far, we are satisfied with this result, but you can add additional conditions for taking damage.

Defense

Now the enemies can attack us, but we must also be able to defend. And, as they say, the best defense is a good offense. Let's add our character the ability to attack by pressing the spacebar.
Go to the `Player` class and add this little bit there:
src/classes/player.ts

```
import { Input, Scene } from 'phaser';
...
private keySpace: Input.Keyboard.Key;
...
constructor(...) {
    ...
    this.keySpace = this.scene.input.keyboard.addKey(32);
    this.keySpace.on('down', (event: KeyboardEvent) => {
      this.anims.play('attack', true);
      this.scene.game.events.emit(EVENTS_NAME.attack);
    });
    ...
}
...
```

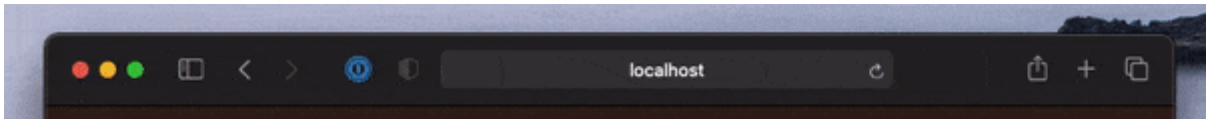
Excellent! Our character can now attack enemies. When we press the spacebar, we play the attack animation, which was described in one of the previous parts, and trigger the attack event. We'll declare listeners for this event in the `Enemy` class.
src/classes/enemy.ts

```
...
private attackHandler: () => void;
...
constructor(...) {
    ...
    this.attackHandler = () => {
    if (
      Phaser.Math.Distance.BetweenPoints(
        { x: this.x, y: this.y },
        { x: this.target.x, y: this.target.y },
      ) < this.target.width
    ) {
      this.getDamage();
            this.disableBody(true, false);
      this.scene.time.delayedCall(300, () => {
        this.destroy();
      });
    }
  };
    ...
    // EVENTS
  this.scene.game.events.on(EVENTS_NAME.attack, this.attackHandler, t
  this.on('destroy', () => {
    this.scene.game.events.removeListener(EVENTS_NAME.attack, this.at
  });
}
```

Here we describe the handler for the damage-taking event, and each time we make our player character swing the hammer, we check to see if the enemy is really close enough to get damaged.

When receiving damage, we turn off the physical model of the enemy so that they don't continue to attack the player character while in the process of disappearing, and after blinking, we remove them from the scene.

Also, we described that at the moment when the enemy is being removed from the scene, we must stop the damage reception. Otherwise, the enemy will still be able to take damage and will try to process the event.



Game completion

Great, we can now fight the enemies! Time to add a game completion screen to show if we won or lost.
Let's start with the **losing** option. In the `Player` class, when receiving damage, add a check for the HP value, and if it becomes 0, then it will trigger the end-of-the-game event with the status of a loss.
Determine the possible game statuses:
src/consts.ts

```
export enum GameStatus {
  WIN,
  LOSE,
}
```

src/classes/player.ts

```
import { EVENTS_NAME, GameStatus } from '../consts';
...
```

```
public getDamage(value?: number): void {
  super.getDamage(value);
  this.hpValue.setText(this.hp.toString());
  if (this.hp <= 0) {
    this.scene.game.events.emit(EVENTS_NAME.gameEnd, GaneStatus.LOSE)
  }
}
...
```

Next, let's add a listener for this event to the UI scene and define an event handler. Also, add a text output to the screen, which will contain our message about the game status.

src/scenes/ui/index.ts

```
import { EVENTS_NAME, GameStatus } from '../../consts';
...
private gameEndPhrase!: Text;
private gameEndHandler: (status: GameStatus) => void;
...
constructor(...) {
    ...
    this.gameEndHandler = (status) => {
    this.cameras.main.setBackgroundColor('rgba(0,0,0,0.6)');
    this.game.scene.pause('level-1-scene');
    this.gameEndPhrase = new Text(
      this,
      this.game.scale.width / 2,
      this.game.scale.height * 0.4,
      status === GameStatus.LOSE
        ? `WASTED!\nCLICK TO RESTART`
        : `YOU ARE ROCK!\nCLICK TO RESTART`,
    )
      .setAlign('center')
      .setColor(status === GameStatus.LOSE ? '#ff0000' : '#ffffff');
    this.gameEndPhrase.setPosition(
      this.game.scale.width / 2 - this.gameEndPhrase.width / 2,
      this.game.scale.height * 0.4,
    );
  };
```

```
}
...
private initListeners(): void {
  this.game.events.on(EVENTS_NAME.chestLoot, this.chestLootHandler, t
  this.game.events.once(EVENTS_NAME.gameEnd, this.gameEndHandler, thi
}
```

Now, when the main character runs out of HP, we are shown a screen with the corresponding inscription:

Great, we have the logic of losing ready, now let's implement the **winning** one. Remember that `gameConfig` object where we specified the parameters for the game? We can use this object to store some parameters, for example, the number of points needed to win.

Let's go to the file with `gameConfig` and add a field where we describe that you need to score 40 points to win, that is, collect 4 chests.

src/index.ts

```
type GameConfigExtended = Types.Core.GameConfig & {
  winScore: number;
};
export const gameConfig: GameConfigExtended = {
    ...
    winScore: 40,
```

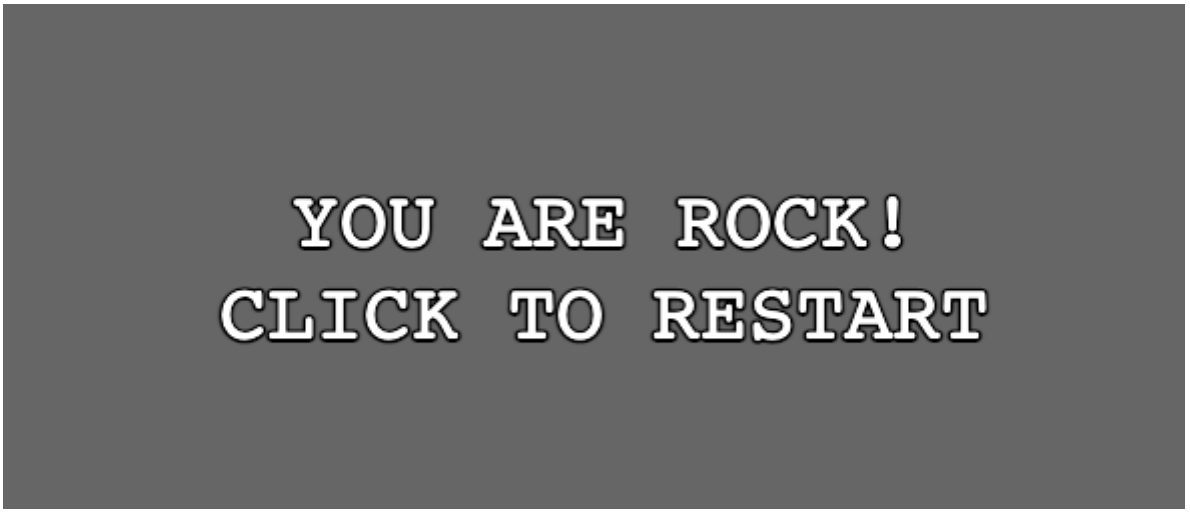
```
};
```

Now let's return to the UI scene, and in the handler for collecting the chest, we'll check if we have enough points to win.

src/scenes/ui/index.ts

```
import { gameConfig } from '../../';
...
this.chestLootHandler = () => {
  this.score.changeValue(ScoreOperations.INCREASE, 10);
  if (this.score.getValue() === gameConfig.winScore) {
        this.game.events.emit(EVENTS_NAME.gameEnd, 'win');
    }
};
...
```

Cool, now that we have 40 points, we see the victory screen!



Restart

You probably noticed that in the inscription there's something about restarting the game on click. Let's implement it, it's pretty simple.

On the UI scene, inside our end-of-the-game event handler, add a click listener.

Inside, we'll be clearing all current event listeners and restarting the scene:

src/scenes/ui/index.ts

```
...
this.gameEndHandler = (...) => {
    ...
    this.input.on('pointerdown', () => {
        this.game.events.off(EVENTS_NAME.chestLoot, this.chestLootHar
        this.game.events.off(EVENTS_NAME.gameEnd, this.gameEndHandler
        this.scene.get('level-1-scene').scene.restart();
        this.scene.restart();
    });
}
...
```

To make one scene from another, we use a record of getting a scene from the list of scenes by its key `this.scene.get(sceneKey).scene` and then we can work with the resulting one.

Awesome! Now we can restart the game😎

When we restart the scene, all objects there are destroyed (the `destroy()` method built into the scene class gets triggered), but all their listeners remain. Because of this, we can observe a bug when, after restarting, when we press the spacebar, an error is displayed in the console that such and such animation cannot be found. This is an attack animation. The error occurs due to the fact that the listener tied to the state of the player BEFORE the restart gets triggered, and is still trying to play an animation that no longer exists. Add the `Player` class to remove all listeners when the player character gets destroyed.

src/classes/player.ts

```
...
constructor(...) {
    ...
    this.on('destroy', () => {
    this.keySpace.removeAllListeners();
```

```
    });
}
```

Now there are no errors, and we can start the game again as many times as we like 😎

* * *

Thank you for being here!

This concludes our Phaser 3 tutorial. We have learned from scratch how to set up the environment for developing and building a web game project with Phaser.js. We created scenes and classes for the player character and his enemies. We downloaded and worked with sprites, sprite sheets, atlas, animations. We learned how to create a game map from scratch and place objects. We added keyboard controls for the player character and taught bots to hunt him. And finally, we described the logic for winning and losing.

So, everything's in your hands now. This tutorial is enough to get your feet wet so that you could dive deeper, and understand what they all are talking about in other tutorials.
Once again,

- Check the game out and play the Demo version.
- Here are the assets for this tutorial.
- And this is where you can find the final code.