Building a platformer game using Phaser and Tiled

Tiled is a free map editor that can be easily integrated with Phaser. In this tutorial, we will build a simple platformer game using Phaser and Tiled. While the map will be created using Tiled, we will use Phaser to read this map and load all the objects into the level. In this tutorial I'll assume you're familiar with the following concepts

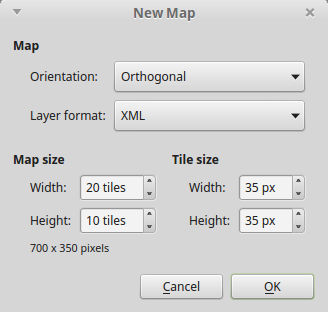
* javascript and object-oriented programming.
* basic Phaser concepts, such as: states, sprites, groups and arcade physics.

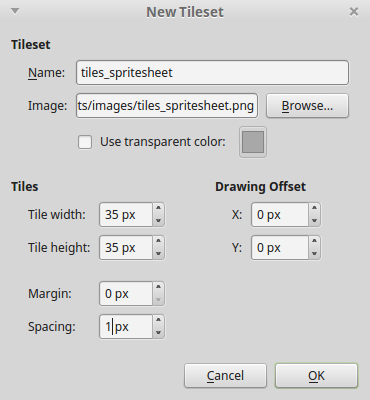
The following concepts will be covered in this tutorial:

* using Tiled to create a platformer level.
* writing a Phaser State that reads the Tiled map and instantiate all the game objects.
* writing the game logic of a simple platformer.

## Creating the map

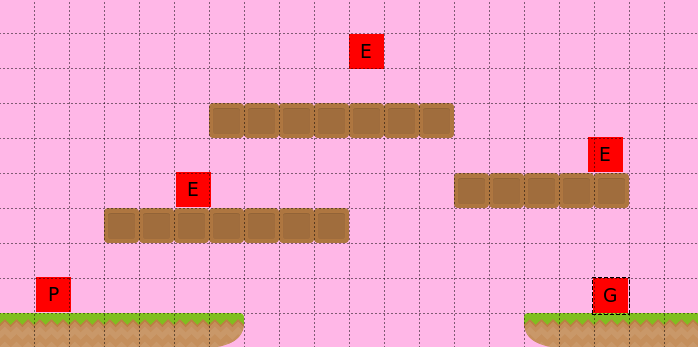
First, we have to create a new Tiled map. In this tutorial we will use an orthogonal view map with 20x10 tiles. Each tile being 35x35 pixels. For those not familiar with it, an orthogonal view means that the player views the game by a 90 degree angle (you can read more about the difference between orthogonal and isometric view here: http://gamedev.stackexchange.com/questions/22277/difference-between-orthogonal-map-and-isometric-map).



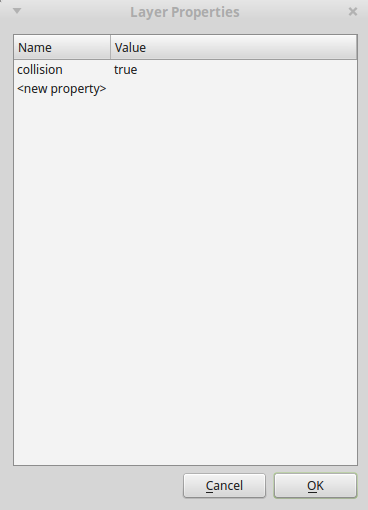
After creating the map, we have to add our tileset, which can be done clicking on "map -> new tileset". The tileset we're going to use has 35x35 pixels, and a 1 pixel spacing.

In tiled, there are two types of layers: tile layers or object layers. Tile layers are composed by the sprites of the loaded tileset and are aligned in the map grid. In this tutorial, we will have two tile layers: background and collision. Object layers are not aligned to a grid, and represent the objects of the game. In this tutorial, we will have only one object layer, containing all our game objects.

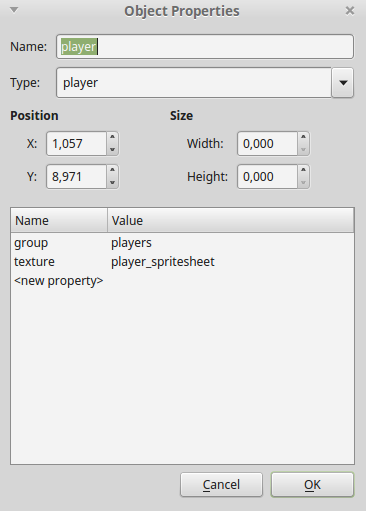
To create tile layers, just use the stamp brush tool to paint the layer with the desired tiles. For the object layer, select a tile image to use and put it in the desired position. Notice that the images used in the object layer don't represent the actual image that will be used by Phaser during the game. We will have to load the correct asset later in our code. Here is an example of a created map. Feel free to create your own!



Now, we're going to set some properties that will allow our game to load the map. First, in the collision layer we will add a property telling our game that this layer may collide with other objects, as shown below:



Finally, for each object, we set its properties. The name, type, group, and texture properties are required for our game, since we will use them to properly instantiate the game objects. Any other properties should be defined according to our game logic. For now, we will set only the required properties, after we code the game logic, we can go back to add each object properties.



With the map finished, export it to a json file, so our game can load it.

## Json level file

There are some informations our game needs to know before loading the map, like the game assets, groups and map information. We will keep all this information in a json file which will be read at the beginning of the game. Below is the json level file we will use in this game. Notice we have to define the assets we will need, the groups of sprites and the map information.

**<add file level1.json in source/assets/levels/>**

## Game states

We will use the following states to run our game:

* Boot State: loads a json file with the level information and starts the Loading State.
* Loading Sate: loads all the game assets, and starts the Level State.
* Tiled State: creates the map and all game objects.

The code for Boot State is shown below. The Boot State loads the json file described above, so the assets can be loaded in the Loading State.

**<add file BootState.js in source/js/states/>**

As shown below, the Loading State loads all the game assets in the preload method, and when it is finished, it starts the Tiled State in the create method. Notice that, since we specify the asset type in the json file, it is straightforward to load them, and we can load different kinds of assets.

**<add file LoadingState.js in source/js/states/>**

Finally, the Tiled State reads the map data and creates the game objects. We'll go through this state in more details, since it's the most important state of our game. I recommend that you take a look in the json file generated by Tiled, to have an idea of its structure. If you're confused about the Phaser map properties and methods, check Phaser documentation (<http://phaser.io/docs/2.4.3/Phaser.Tilemap.html>).

**<add file TiledState.js in source/js/states/>**

First, we have to tell Phaser what image represents each tileset we used in Tiled (the Tiled tilesets are in this.map.tilesets). Since we have only one tileset, and we have the image name from our json file, we can easily do that.

Next, we have to create the map layers. The map object has a layers array that we will iterate. If a layer has the property collision that we added in Tiled, we have to make it available for collision. To do this, we need to tell Phaser which tiles can collide, so we iterate through layer.data, which contains all the layer tiles and add them to a list. At the end, we set the collision for all these tiles. After creating all layers, we resize the world to be the size of the current layer. Since all our layers have the same size, we don't care which one is the current layer.

The next step is to create the groups of our game. This can be easily done by iterating the groups array of our json file and adding a new group for each one of them. However, two things are important in this step: 1) the order of the groups define the order they are drawn on the screen; 2) groups must be created after layers, otherwise the layers would be drawn above them.

Finally, we go through all object layers (in our case, only one) and create the game objects. Since in our map we defined the object type, it is easy to instantiate the correct Prefab. Notice that our prefab position is not the same position of the Tiled object. That happens because Tiled coordinates start at the bottom left corner, while Phaser coordinates start at the top left corner. Also, we want our prefabs anchor point to be 0.5, so we have to set the position to be the center of our prefab.

## Prefabs

In Phaser, prefabs are objects that extend Phaser.Sprite, acting as objects in our game. In our platformer game, we will need four prefabs: Player, Enemy, FlyingEnemy and Goal, which will all be explained now. This is our Prefab class:

**<add file Prefab.js in source/js/prefabs/>**

## Player

Our player will be able to walk, jump and kill enemies. For that we will need the following properties: walking speed, jumping speed and bouncing, which are all initialized in the constructor.

In the update method, we check all the player collision (collision layer and enemies) and do the walking and jumping logic. There are some important details to notice regarding player walking. First, we don't want to change the player direction while he's already moving. For example, if he's moving left and the right arrow is pressed, we want the player to keep moving left until the left arrow is released. So, we change the player velocity only when the correct key is pressed and the player isn't moving in the opposite direction. Second, we have to change the sprite direction accordingly. To do this, we use the scale attribute of the sprite, which will invert the sprite direction.

To allow the player to jump, we can just check for the up arrow key in the update method and change the velocity accordingly. The only important thing to notice here is that we only want to allow the player to jump when it is touching the ground. Since the ground is a tile, we have to use the blocked property of body, not touching (check the documentation in http://phaser.io/docs/2.4.3/Phaser.Physics.Arcade.Body.html for more information).

Another thing we have to check is if the player had fallen. For that, we check if the player bottom y is equal to the world height. If so, the player dies.

Finaly, the hit enemy method checks is called when the player collides with an enemy and checks if the player is on top of the enemy. If that's the case, the enemy is killed, otherwise the player dies.

**<add file Player.js in source/js/prefabs/>**

## Enemy

Our enemy will be simple, it will only walk up to a maximum distance and then switch direction. For this, the properties we need are: walking speed, walking distance and direction. Notice that, in the constructor we set the initial velocity and scale according to the direction property. Also, we save the previous x position, which in the beginning is the sprite x.

In the update method we check if the walked distance (this.x - this.previous\_x) is greater or equal to the maximum walking distance. If that's the case we switch the direction, updating the velocity, previous x and scale.

**<add file Enemy.js in source/js/prefabs/>**

# Flying enemy

Now that we have our regular enemy, creating a flying enemy is very easy. We will just create another prefab that extends Enemy, and that isn't affected by the gravity. Also, since your flying enemy asset is different, we have an animation too.

**<add file FlyingEnemy.js in source/js/prefabs/>**

# Goal

Our goal is simple. It has a next level property, and overlap with the player. If the player reaches the goal, the next level should be started.

Notice that, to load the next level we only need to start the Boot State sending as parameter the path of the next level json file. In this tutorial, we will have only one level, but this structure makes it simple to make it a multilevel game. Try creating different levels, and see how it works.

**<add file Goal.js in source/js/prefabs/>**

## Finishing the game

Now that we have our game logic, we know what properties are necessary to be defined in the Tiled map. We can just set them on our map and our game is working! Let me know in the comments section your opinion and what you would like to see in the next tutorials.

