Lappeenrannan teknillinen yliopisto

LUT UNIVERSITY (School of Technology)

Introduction to web development, Online course

Ekong Rejoice, 001494262

FEATURE	MAX POINT
Well written PDF report	3
Application is responsive	4
Application works on Firefox, Safari, Edge and Chrome	3
Clear directory structure	2
Clear plot	3
Two or more collectibles	2
Moving parts in the game	3
More than one map	3
Keyboard, No mouse	1
Physics engine	2
Enemies hurt player	3
Sound Effects	3
New feature : Camera movement*	2
New feature: More than one player sprite*	2
Total	36

***: this is because the player can feel like the game area is large enough to dodge various attacks by the enemy, and when it moves to another area, the camera will move to that area so that the player will never be out of focus in a case where the screen size of the player is small.**

***: the player can come up with various strategies because all player sprite have different speed and health status.**

REPORT

PLANS

I plan to make a small game with a lot of features that will make use of the the Phaser js game engine. The game will feature a player and a wide varieties of enemy that can take hit damage from the player and the player can take hit damage from them as well. The strategy of the gamer or user of the game is to dodge all incoming attacks from the enemy. While doing that the user will be able to shoot and then kill all visible enemy in its path. I have decided to use very few player options in which the player sprite will change anytime the user starts to play the game. I have materials in which this game can become possible and it is using the idea of jets shooting at enemy jets and vice versa. The game will also feature the phaser js physics engine which will make it even more fun to play. I have also decided to add collision detected where if the user cannot finish the enemy, the user can hit the enemy and the enemy will die but this will affect the stamina of the player as well. There will be collectibles that will equally affect the health and stamina of the player in a positive way, but this collectibles will only help at random, either speed, rate, health etc. This will make the game hard and fun to play.

RESOURCES AND CODE EXPLAINED

I started by installing the phaser js into my folder for the game website and then bringing in all my assets into my assets folder so that things will make easier to make reference for and in this section, i will explain all the files and resources i used to make this project a success. Of course, the first file that i began with was my index.html, main.css and main.js as well.

INDEX.html

In this file i link the css and also the phaser js folder so that i could be able to use all of it's resources in the future. I also link the main.js and this will also be where i will link all the future javascript files. In the body of this file i set the div tag with an id of the firsttry-game and a class of game which will serve as the container for the game canvas, where the game will be displayed.

MAIN.css

This one is going to be simple. I set a universal selector(*), which applies the following styles to all elements on the page. Then I put a user selector and

assigned it as none which prevents text selection in other modern browsers. This makes sure that users cannot accidentally select text while interacting with the game. I then put in the html and body selector which applies to the html and body elements in the html file. I set it width and height to be 100%, the margin to be 0 and the overflow to be hidden, I then put a background to be kind of black.

MAIN.js

window.onload function runs when the entire webpage has loaded, ensuring all resources (images, scripts, etc.) are available before the game starts. use strict directive enables strict mode, which helps catch common coding mistakes and unsafe actions. I declared a variable game that will hold the Phaser game instance, then Created a new Phaser game instance. CONFIG.GAME_WIDTH * CONFIG.PIXEL_RATIO sets the game width based on a configuration constant. CONFIG.GAME_HEIGHT * CONFIG.PIXEL_RATIO sets the game height based on a configuration constant. Phaser.AUTO automatically chooses the best renderer (Canvas or WebGL) for the device. game.CONFIG assigns the configuration object CONFIG to the game instance, making the configuration accessible throughout the game. I then added other game states and then started the game by transitioning to the boot state.

GAME.js

I created a Game function to initialize the game with default values such as score, player, last update time, and game states (preplay, play, postplay). I defined the prototype methods for the Game object which included create, createWorld, createGround, generateTerrain, createEnemies, createAudio, state transitions, update methods, collision handling, and GUI updates. In the create method, I initialized the game state to preplay, created the game world and ground, set the scroll speed, and initialized the bonus pool and player. I set up the camera to follow the player and added input handlers and GUI elements. I also called the methods to create the world, ground, enemies, audio, and GUI. I added bitmap text to the game to display messages such as "Get ready! FIRE!!!" and initialized the text positions and scaling. I initialized the game camera to follow the player. Specifically, I set the camera to track the player using the Phaser.Camera.FOLLOW_PLATFORMER mode. This ensures that the camera

movement is directly linked to the player's position and actions. I configured the world boundaries by setting the game world size to specific dimensions based on the configuration values. This setup establishes the playable area of the game and ensures the camera operates within these limits. The dimensions are set using this.game.world.setBounds, where I provided the width and height of the world. I managed the position of the ground in relation to the camera's scrolling behavior. The ground layer's position is adjusted dynamically based on the scrolling speed and direction. This allows the ground to move in sync with the camera, creating a continuous background effect.I started the physics system and set the world bounds. The ground creation involved setting up the tilemap, determining the current background, and setting the ground's real and logical sizes. I also generated terrain data and called a method to draw the ground based on this data. I implemented a method to generate terrain using a noise function to create a randomized map. This includes calculating the appropriate tile positions based on the camera's scroll position and redrawing the ground accordingly. The terrain was classified into different types (forest, earth, water, deepwater) and smoothed out for a realistic look. For the second map it was a simple map so it followed the same function as the first map with the tile map and noise function to create a randomized map. The final map was created by assigning appropriate tile values based on the generated data. I created pools for different types of enemies and bullets to manage their existence and reuse them efficiently. This included creating various enemy types like turrets, planes, vessels, and flagships. I added audio elements for different actions in the game such as shooting, explosions, and collecting bonuses. I implemented methods to handle state transitions from preplay to play and from play to postplay, updating the game state and resetting enemy spawn timers accordingly. I added an update method to handle the game loop, updating the delta time, handling enemy spawning, collisions, and background updates. I implemented methods to spawn enemies and bonuses based on the current game time and predefined delays. I added collision detection between player bullets and enemies, player and enemies, player and enemy bullets, and player and bonuses. These methods managed the interactions and updated the game state and score accordingly. I created an explosion animation for visual effects when entities were destroyed and updated the GUI to reflect changes in the game state, such as the score and player health. I adjusted the camera's position relative to the game state and player actions. By managing

the camera's follow behavior and updating the ground's position based on player movement and game state changes, I ensured that the camera always provides an appropriate view of the gameplay area.

MENU.js

I created a Menu constructor function. This constructor initializes the properties titleTxt and startTxt to null. I then extended the Menu prototype to include methods necessary for creating and updating the menu. I calculated the center coordinates of the game screen and stored them in variables x and y. I added a title text using this.add.bitmapText at the center of the screen with the font 'minecraftia' and the text 'Jet Shooter'. I aligned the title text to the center and adjusted its position horizontally and vertically. I incremented the y coordinate to position the start text below the title text. I added start instructions using this.add.bitmapText with details on controls and game start instructions. I aligned the start text to the center and adjusted its position horizontally. I disabled mouse input by setting this.input.mouse.enabled to false, this is because I was not planning to use a mouse with this game because mouse movements will affect the players movement in the wrong way and using the clicking ability of mouse will require the user to use the mouse in their left hand and the controls in the right hand and may not be too comfortable for the user. I accessed the keyboard input using this.input.keyboard. I checked if the ENTER key is pressed using keyboard.isDown(Phaser.Keyboard.ENTER). If the ENTER key is pressed, I transitioned to the 'game' state using this.game.state.start('game'). I ensured the Menu function is available under the window['firsttry'] namespace by assigning it appropriately.

BOOT.js

I created a configuration object named CONFIG. This object contains various game settings such as dimensions, pool sizes, speed settings, audio levels, and class statistics for different types of game entities. I extended the Array prototype with a remove function. This function allows for removing a range of elements from an array by specifying the starting and ending indices. I defined a function that is executed immediately to encapsulate the boot process of the game. Within this function, I created a Boot constructor function to initialize the boot state. I added a preload method to the Boot.prototype to load the initial assets required for the

preloader, specifically a preloader image. I then added a create method to the Boot.prototype. I set the maximum number of input pointers to 1. I configured the game's scaling options using Phaser.ScaleManager.SHOW_ALL to scale the game while maintaining aspect ratio and aligning it horizontally and vertically on the page. I refreshed the scale settings to apply them. Finally, I started the 'preloader' state of the game. I ensured that the Boot function is accessible globally by assigning it to the window['firsttry'].Boot object, creating the firsttry namespace if it doesn't already exist.

PRELOADER.js

I initiated the process by defining an anonymous function which immediately gets executed. Within this function, I enabled 'use strict' mode to enforce stricter parsing and error handling. I created a Preloader constructor function. This constructor initializes the properties asset and ready, setting them to null and false respectively. I then extended the Preloader prototype to include several methods necessary for preloading assets. In the preload method I: Added a sprite asset centered at coordinates (320, 240) using this.add.sprite, Set the anchor point of the sprite to the center using this.asset.anchor.setTo(0.5, 0.5), Added an event listener for onLoadComplete which triggers once the loading process is finished using this.load.onLoadComplete.addOnce(this.onLoadComplete, this), Designated the sprite as the loading indicator with this.load.setPreloadSprite(this.asset), I loaded various assets such as images, spritesheets, bitmap fonts, and audio files using the respective this.load.image, this.load.spritesheet, this.load.bitmapFont, and this.load.audio methods. In the create method, I disabled the cropping of the asset sprite by setting this.asset.cropEnabled to false. In the update method, I checked if the ready property is true. If it is, I transitioned to the 'menu' state of the game using this.game.state.start('menu'). I defined the onLoadComplete method to set the ready property to true once all assets have been loaded. I ensured the Preloader function is available under the window['firsttry'] namespace by assigning it appropriately.

From here on out, I will explain the class folder and the file that are located in it which contains the functions of all the actors to the player that the game obeys.

ACTOR.js

I defined an anonymous function and invoked it immediately to create a local scope and avoid polluting the global namespace. I defined a constructor function named Actor that takes two parameters: state and image. I assigned the state parameter to this.state and state.game to this.game. I called the parent class constructor Spriter with the state and image parameters by using window['firsttry'].Spriter.call(this, state, image). I initialized an isPinnedToGround property to false. I set up the inheritance so that Actor inherits from Spriter. I achieved this by creating an object with Spriter.prototype as its prototype and assigning it to Actor.prototype. I set the Actor prototype's constructor back to Actor to ensure the correct constructor reference. I added a method named getAngleTo to Actor.prototype that calculates the angle to a target. I checked if the target has x or y properties, and if so, I calculated the angle using Math.atan2(target.x - this.x, target.y - this.y). I ensured that the Actor constructor is accessible globally by assigning it to window['firsttry'].Actor.

BULLET.js

I defined a function and invoked it immediately to create a local scope and avoid polluting the global namespace. I defined a constructor function named Bullet that takes two parameters: state and type. I assigned the state parameter to this.state and state.game to this.game. I called the parent class constructor Spriter with the state and an image key derived from type by using window['firsttry'].Spriter.call(this, state, 'mob_bullet_' + (type + 1)). I initialized several properties: type to 0, energy to 30, speed to 120, and shooter to undefined. I set up the inheritance so that Bullet inherits from Spriter. I achieved this by creating an object with Spriter.prototype as its prototype and assigning it to Bullet.prototype. I set the Bullet prototype's constructor back to Bullet to ensure the correct constructor reference. I added a method named revive to Bullet.prototype that revives the bullet and sets its velocity based on the shooter's position and an angle. I assigned the shooter parameter to this.shooter. I reset the bullet's position to the shooter's position using this.reset(shooter.x, shooter.y). I calculated the bullet's velocity based on its speed and the provided angle, scaled by the CONFIG.PIXEL_RATIO. I added an update method to Bullet.prototype that updates the bullet's state. I checked if the bullet is alive, and

if so, I called the parent class's update method using window['firsttry'].Spriter.prototype.update.call(this). I defined a safeRange and checked if the bullet is outside the game boundaries, considering the safeRange and CONFIG.PIXEL_RATIO. If the bullet is outside, I called this.kill() to deactivate it.

COLLECTIBLE.js

The code defines a class named Collectible that extends the Actor class, setting up properties, methods, and animations to manage collectible objects in the game. I defined a constructor function named Collectible that takes state and image parameters. I called the parent class constructor Actor using window['firsttry'].Actor.call(this, state, image) to initialize the Collectible with the state and image provided. I set the alive property to true to indicate that the collectible is currently active. I invoked the updateClass method to initialize the collectible's class and animations. I set up inheritance so that Collectible inherits from Actor. I achieved this by creating an object with Actor.prototype as its prototype and assigning it to Collectible.prototype. I set Collectible.prototype.constructor to Collectible to ensure that the constructor reference is correct. I added an update method to Collectible.prototype which: Calls the parent class's update method using window['firsttry'].Actor.prototype.update.call(this). Checks if the y coordinate of the collectible is greater than a threshold (CONFIG.GAME_HEIGHT * CONFIG.PIXEL_RATIO + 200). If the threshold is exceeded, the kill method is called to deactivate the collectible and exit the method. I added an updateClass method to Collectible.prototype which: Randomly sets the bonusClass property using this.state.rnd.integerInRange(0, 3), Adjusts the bonusClass to a different value if it is 3, setting it to 4, Computes an offset based on the bonusClass, Adds an idle animation with frames [0 + offset, 1 + offset, 2 + offset, 1 + offset], a frame rate of 15, and sets it to loop, Plays the idle animation to visually represent the collectible.

ENEMY.js

This class extends the Mob class and includes properties, methods, and behaviors specific to enemy objects in the game. I created a constructor function named Enemy that takes state and image parameters. I called the parent class constructor window['firsttry'].Mob.call(this, state, image) to initialize the Enemy

with the provided state and image. I initialized several properties specific to the Enemy class: shootDelay, speed, points, bulletType, bulletSpeed, bulletCancel, lootProbability, lootType, shootConfig, shoote and in respect to the order that it was listed this was for time delay between shots, movement speed of the enemy, points awarded when the enemy is defeated, type of bullets the enemy uses, speed of the bullets, flag to determine if bullets should be canceled on death, chance of dropping loot when defeated, type of loot dropped, configuration for shooting behavior, array to keep track of ongoing shoot actions. I set up inheritance by creating an object with Mob.prototype as its prototype and assigning it to Enemy.prototype. I updated Enemy.prototype.constructor to refer to Enemy to maintain proper constructor references. I added an update method to Enemy.prototype which calls the parent class's update method using window['firsttry'].Mob.prototype.update.call(this). Checks if the enemy is alive, if the player is alive, if the enemy is positioned sufficiently above the player, if the current time is greater than nextShotAt, and if there are available bullets in the pool. If all conditions are met, the shoot method is invoked with the shootConfig. I added a shoot method to Enemy.prototype which creates a new Shoot object with the state, this (the enemy), and shootConfig. Adds this new Shoot object to the shoote array. Updates nextShotAt to schedule the next shot. I added a die method to Enemy.prototype which calls the parent class's die method using window['firsttry'].Mob.prototype.die.call(this). Iterates over all active shoots and calls their die method with bulletCancel. Determines if loot should be dropped based on lootProbability, and if so, invokes the loot method. Plays an explosion sound based on the enemy's maximum health, with different sounds for different health ranges. I added a revive method to Enemy.prototype which resets the enemy's position and velocity based on whether it is pinned to the ground. Sets nextShotAt to a random integer within the shootDelay. Calls the parent class's revive method using window['firsttry'].Mob.prototype.revive.call(this). I added a loot method to Enemy.prototype which retrieves a bonus object from the bonusPool. Updates the bonus class, resets its position to the enemy's location, and sets its velocity and angular velocity. I ensured that the Enemy constructor is globally accessible by assigning it to window['firsttry'].Enemy.

FLYING MOBS.js

This code defines three types of enemy classes in a game: Plane, Vessel, and Flagship. Each class extends the Enemy class and includes properties and methods specific to each enemy type. I defined a constructor function named Plane that inherits from window['firsttry'].Enemy. I initialized the Plane with specific properties:

- maxHealth: Set to 30.
- speed: Set to 60.
- shootDelay: Time between shots, set to 3000 milliseconds.
- bulletSpeed: Speed of bullets, set to 125.
- points: Points awarded for defeating this enemy, set to 100.
- lootProbability: Chance of dropping loot, set to 0.1.

I set up the shootConfig for the Plane with specific shooting parameters. I generated a random planeClass value and used it to configure animations which added 'idle', 'left', and 'right' animations with different frames and playback speeds. And also set the initial animation to 'idle'. I set up inheritance by creating Plane.prototype with window['firsttry'].Enemy.prototype as its prototype. I updated Plane.prototype.constructor to refer to Plane. added an update method to Vessel.prototype, It calls the update method of the parent class using window['firsttry'].Enemy.prototype.update.call(this). I defined a constructor function named Vessel that also inherits from window['firsttry'].Enemy and then I initialized the Vessel with properties. I set up the shootConfig for the Vessel with specific shooting parameters. I added an 'idle' animation and set it as the initial animation. I set up inheritance by creating Vessel.prototype with window['firsttry'].Enemy.prototype as its prototype. I updated Vessel.prototype.constructor to refer to Vessel. I defined a constructor function named Flagship that inherits from window['firsttry'].Enemy. I initialized the Flagship with properties like maxHealth, speed, shootDelay, points, lootProbability, bulletCancel.I set up the shootConfig for the Flagship with specific shooting parameters.I added an 'idle' animation and set it as the initial animation. I set up inheritance by creating Flagship.prototype and I updated Flagship.prototype.constructor to refer to Flagship. With an update method for it.

I ensured that the Plane, Vessel, and Flagship constructors are globally accessible by assigning them to window['firsttry'].

MOB.js

This class extends the Actor class and manages various aspects of a game character, including health, damage handling etc. I defined a constructor function named Mob that inherits from window['firsttry'].Actor. I initialized the Mob with several properties like alive, health, maxHealth, isDamaged, damageBlinklast, tint. I set up inheritance by creating Mob.prototype with window['firsttry'].Actor.prototype as its prototype. I updated Mob.prototype.constructor to refer to the Mob constructor function. I added an update method to Mob.prototype this method calls the update method of the parent class using window['firsttry'].Actor.prototype.update.call(this). I then checked if the Mob has moved off-screen and, if so, called the kill method to deactivate it. I also updated the mob's visual tint based on its damage state by calling this.updateTint(). I defined the updateTint method to handle the visual indication of damage if the mob is damaged, it reduces the damageBlinkLast timer. If the timer runs out, it resets the damage state. The tint is set to red when the mob is damaged and white otherwise. I added the takeDamage method to reduce the mob's health subtracts the given damage value from this.health. If health falls to 0 or below, the kill method is called to deactivate the mob. If health is still above 0, the mob starts blinking to indicate damage by calling this.blink(). I defined the blink method to initiate the damage blink effect sets isDamaged to true and initializes the damageBlinkLast timer based on CONFIG.BLINK_DAMAGE_TIME. I added the revive method to restore the mob's health to its maximum value resets health to this.maxHealth. I defined the die method to handle the mob's death calls the kill method to deactivate the mob. I ensured the Mob class is available globally by assigning it to window['firsttry'].

PLAYER.js

The Player class extends the Mob class and manages the player's behavior, including movement, shooting, and interactions.

I defined the Player constructor function that takes state as a parameter.this.state is assigned the game state. this.game is assigned the game instance from the state. this.playerClass is randomly selected between 1 and 4. this.playerStats is set from CONFIG.CLASS_STATS based on the selected player class. I called the Mob constructor using window['firsttry'].Mob.call(this, state, 'player_' + this.playerClass) to initialize the player as a type of mob. I adjusted the player's body size using this.body.setSize() for collision detection. I defined several animations for different player states (e.g., idle, left, right) and set the default animation to 'idle'. I set the player's health to match the player's stats and called this.updateStats() to configure player attributes.

spawn(): I set the player's initial position on the screen to the center horizontally and a third from the top vertically.

createBulletPool(): I created a group for bullets, enabling physics and setting up properties such as size, anchor, and out-of-bounds behavior. I also called this.updateBulletPool() to configure bullet animations.

update(): I overrode the update() method from the Mob class to include:

- this.updateInputs() to handle user input.
- this.updateSprite() to update the player’s animation based on movement.
- this.updateBullets() to manage bullets in the bullet pool.

updateStats(): I updated player attributes like speed, acceleration, strength, and shooting delay based on the player's stats.

updateInputs(): I handled player input based on the current game state, including movement and shooting controls. I updated movement direction and handled different game states (e.g., preplay and menu).

updateSprite(): I updated the player's sprite animation based on velocity.

Movement Functions: I implemented methods (moveLeft(), moveRight(), moveUp(), moveDown(), floatH(), floatV()) to control the player’s movement and acceleration.

fire(): I handled shooting mechanics by checking the time between shots, resetting bullet positions, and setting bullet velocity. I also played a shooting sound effect based on the player's strength.

updateBullets(): I managed the bullet pool by deactivating bullets that move out of bounds.

updateBulletPool(): I updated bullet animations based on player strength.

collectUpgrade(): I handled upgrades by modifying player stats based on the upgrade type and updating the bullet pool. I also played a sound effect upon collecting an upgrade.

SHOOT.js

The Shoot class is designed to handle shooting mechanics in a game, allowing for the configuration of shooting patterns and bullet management. I defined a constructor function named Shoot that takes state, shooter, and shootConfig as parameters.state provides access to the game state and related properties. shooter represents the entity initiating the shoot. shootConfig contains the configuration details for the shooting pattern. I initialized the Shoot instance with the following properties:

- this.state and this.game are assigned from the state parameter for later use.
- this.shooter is assigned to keep track of the shooter initiating the action.
- this.shootConfig is assigned to store the configuration details for shooting.
- this.bullets is initialized as an empty array to hold bullets that will be fired.
- this.t is initialized as an empty array to manage timeouts for shooting.

I set up shooting by looping through shootConfig.nShoots, creating a delayed shooting effect using setTimeout.For each iteration, I used a closure to capture the current shootConfig and j (the iteration index). Inside the timeout function,I calculated the shooting angle based on shootConfig.shootAngle, adjusting it if necessary to point towards the player or to a random direction. I calculated the angle step between bullets based on shootConfig.nBullets and

shootConfig.bulletSpread. I looped through config.nBullets to get a bullet from the pool and configure its angle. I then revived the bullet with the calculated angle. I defined a die method to handle the cleanup of bullets and timeouts when the shooting action is complete, If bulletCancel is true, I iterated over this.bullets and called kill() on each bullet to deactivate it. I cleared all timeouts stored in this.t using window.clearTimeout().

SPRITER.js

This class integrates Phaser's Sprite functionality with additional setup specific to the game. I defined a constructor function named Spriter that takes state and image as parameters.state is used to access the game state and related properties. image is used to set the sprite's image. I initialized the Spriter instance with the following steps:

- I assigned state and game to instance variables for later use.
- I called the Phaser.Sprite constructor with this.game, 0, 0, and image to create a new sprite and positioned it at coordinates (0, 0).
- I used this.game.add.existing(this) to add the sprite to the game world.
- I set the sprite's anchor point to the center (0.5, 0.5) to ensure scaling and rotation are centered around the sprite.
- I scaled the sprite using this.scale.setTo(CONFIG.PIXEL_RATIO, CONFIG.PIXEL_RATIO) to adjust the sprite size based on the configured pixel ratio.
- I enabled arcade physics on the sprite with this.game.physics.enable(this, Phaser.Physics.ARCADE) to allow physical interactions in the game.

I set up inheritance by creating Spriter.prototype with Phaser.Sprite.prototype as its prototype, allowing Spriter to inherit methods and properties from Phaser.Sprite. I updated Spriter.prototype.constructor to ensure that the constructor property correctly refers to the Spriter function.

TURRET.js

This file defines a class named Turret which extends another class Enemy, sets up its properties and behaviors, adds methods for shooting and reviving the turret,

and makes the Turret class globally accessible. I defined a constructor function named Turret that takes a state parameter. I called the parent class constructor Enemy with the state and an image key 'mob_turret_1' by using window['firsttry'].Enemy.call(this, state, 'mob_turret_1'). I initialized several properties: maxHealth to 150, speed to 0, isPinnedToGround to true, bulletType to 1, shootDelay to 5000, points to 5000, and lootProbability to 0.5. I defined a shootConfig object with various shooting configurations such as bulletType, nBullets, bulletDelay, bulletAngle, bulletSpread, nShoots, shootDelay, shootAngle, and shootRotationSpeed. I added an animation named pre-shoot with frames [0, 1, 2, 3, 4, 5, 6, 7, 8], a frame rate of 15, and set it to not loop. I assigned it to the preshoot variable. I attached an event handler to the preshoot animation's onComplete event. When the animation completes, it calls the shoot method from the parent Enemy class with the shootConfig, and then plays the shoot animation. I added an animation named shoot with frames [8, 7, 6, 5, 4, 3, 2, 1, 0], a frame rate of 15, and set it to not loop. I attached an event handler to the shoot animation's onComplete event. When the animation completes, it plays the idle animation. I added an idle animation with frame [0], a frame rate of 5, and set it to loop. I then played the idle animation. I set up the inheritance so that Turret inherits from Enemy. I achieved this by creating an object with Enemy.prototype as its prototype and assigning it to Turret.prototype. I set the Turret prototype's constructor back to Turret to ensure the correct constructor reference. I added an update method to Turret.prototype that calls the parent class's update method using window['firsttry'].Enemy.prototype.update.call(this). I added a shoot method to Turret.prototype that plays the pre-shoot animation. I added a revive method to Turret.prototype that takes i and j parameters. It resets the turret's position based on i and j, and the game configuration constants CONFIG.PIXEL_RATIO and CONFIG.WORLD_SWAP_HEIGHT. I set the turret's vertical velocity to the state's scroll speed scaled by CONFIG.PIXEL_RATIO. I set this.nextShotAt to a random integer between 0 and this.shootDelay. I called the parent class's revive method using window['firsttry'].Mob.prototype.revive.call(this).