Lab 05: *JavaScript OOP + Phaser Framework* | **Module-2**

Phaser Game: Dodger

2D Graphical Browser Game - (JavaScript OOP + Phaser Framework)

Table of Content: *Dodger Game in JavaScript & Phaser Framework*

Topic	Description	Page
Introduction	Learning Objectives: JavaScript OOP, Phaser framework, Single Page Apps	3
OOP Design	Develop a Roadmap: Plan Full Featureset, Identify MVP, Divide Task into Dev Cycles, Goals, & Milestones	4
Dev Cycle 1: **Core Version** MVP Features	Goal 1: Set-up & Launch (Empty) Phaser Game Project Goal 2: Stub a (Empty) Phaser Game Scene Goal 3: Scene with a Background Goal 4: Add Player into Scene Goal 5: Move Player with Inputs & Physics Goal 6: Add Enemies into Scene Goal 7: Move Enemies in Scene with Physics Goal 8: Collision Detections & Game Over	8 10 12 14 17 20 23 25
Dev Cycle 2: **Full Version** Buildout Features	Enhancement 1: Customize Player Hitbox Enhancement 2: Scrolling Background Enhancement 3: Player/Enemy Walk Animation Enhancement 4: Top Score Challenge Enhancement 5: Player Projectiles Enhancement 6*: Enemy Projectiles Enhancement 7*: Power-ups Enhancement 8: Title Screen & Scene Management	29 31 33 37 43 52 56 64
Conclusion	Final Comments - Future Improvement - Submission - Module 2 Project	71

Lab Introduction

Introduction

In this lab, you will build a complete, fast-paced "Dodger" game from the ground up. This isn't just about learning game development; it's about mastering core software engineering principles in a fun, hands-on way.

You'll see firsthand how powerful concepts like Object-Oriented Programming (OOP) allow you to take a professional-grade framework like Phaser and extend it with your own custom, modular code. By the end, you'll have a fully functional game and a much deeper understanding of how modern JavaScript applications are built.

What You Will Build & Learn

Your goal is to **build a complete Dodger Game that runs in a browser**. Along the way, you will learn to:

- **Structure a Project**: Organize code and assets for a clean, manageable web application.
- **Master the Phaser Game Loop**: Understand and use the `preload`, `create`, and `update` methods that form the foundation of a Phaser game scene.
- **Define Custom Game Objects**: Build your `Player`, `Enemy`, `Projectile` and `PowerUp` as JavaScript classes, each with its own properties and behaviors.
- **Leverage Inheritance:** Extend core Phaser classes like Phaser.Scene and Phaser.Physics.Arcade.Sprite to make your custom objects work seamlessly within the game engine.
- **Manage Game State:** Handle dynamic events like spawning enemies, creating animations, and managing player power-ups.
- **Implement an Agile Workflow:** Build complex features in small, testable milestones, mirroring a modern agile development process.

Requirements

Knowledge:

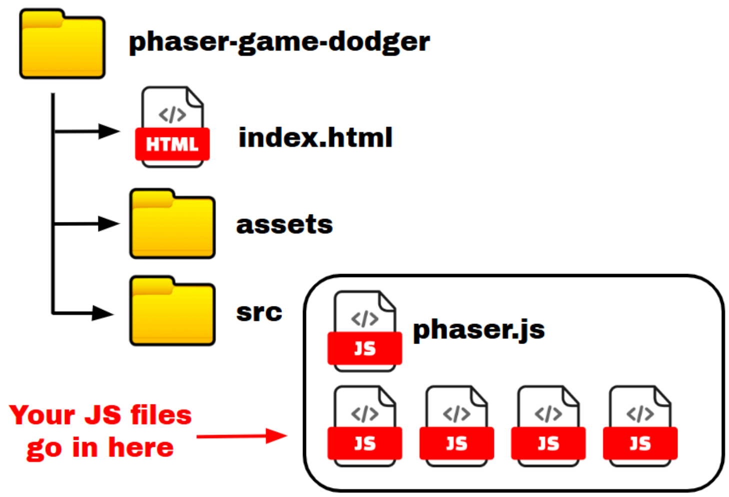
- A solid understanding of basic JavaScript.
- Familiarity with Object-Oriented Programming concepts, including:
  - Classes and Objects (instances)
  - Constructors
  - Properties and Methods
  - Inheritance (`extends`)

Tools:

- A modern web browser (Google Chrome is recommended).
- A local web server to run the project. A simple tool like `http-server` for Node.js is perfect.

Project Architecture:

Start this project by downloading the starter files from GitHub. See the project structure below.



Download Starter files:
https://github.com/scalemailted/phaser-game-dodger

Core Concepts: How a Phaser Game Works

Before we write any code, let's establish a simple mental model of the Phaser framework. Understanding this core structure is the key to everything we'll build.

The Big Picture

Phaser organizes your game into a clear hierarchy:

- **The Game:** This is the top-level object that manages the entire application window and launches your game. We will only create this once.
- **Scenes**: A game is made up of one or more Scenes. A scene can be a main menu, a level, a game-over screen, or the main gameplay area. Our dodger game will use a single `PlayScene` for all the action.
- **Game Objects**: These are the things that live inside a Scene, like the player, enemies, bullets, and power-ups.

The Scene Lifecycle: The Heart of Your Game

Every Scene in Phaser follows a critical, predictable lifecycle. You will write your own custom code for three main methods that Phaser calls automatically:

- `preload():` Phaser calls this first. Its only job is to load all the assets your Scene needs (images, sounds, etc.) into memory before the Scene begins.
- `create():` Once everything is preloaded, Phaser calls this method. Here, you will set up the initial state of your Scene by creating your Game Objects (like the player and enemies) and defining things like collision rules.
- `update():` This method is a loop that runs on every single frame (typically 60 times per second). All of your real-time game logic - like checking for keyboard input and moving characters - will go here.

Inheritance: Your Code works within the Framework

The "magic" of working with a framework like Phaser comes from inheritance. Instead of writing rendering and physics code from scratch, we will create our own custom classes (like `Player`) that extend built-in Phaser classes.

For example, our `Player` class will extend `Phaser's Physics.Arcade.Sprite`. By doing this, our Player object instantly "inherits" all the features of a sprite - it can be rendered, have physics applied to it, and work with the entire Phaser engine, all with just one line of code.

OOP Design in JS

'Approach' → Plan phase

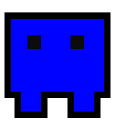
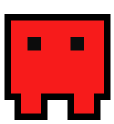
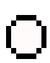
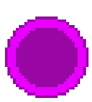
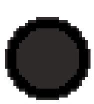
Game Objective:
Our mission is to build a fast-paced, high-score "Dodger" game. You will control a player who must dodge waves of enemies and their projectiles. We will build this game from the ground up using Object-Oriented principles to create clean, reusable code that integrates perfectly with the powerful Phaser 3 game engine.

'Apply' → Do phase

Step 1: Define the Core Gameplay & Feature

First, let's identify the key objects in our game from the player's point of view. We'll split the features for each object into a Minimal Viable Product (MVP) - the simplest playable version - and the Full feature set we'll build out later.

Game Objects (from Player Perspective):

	**Player character** (MVP) - Can move up, down, left, and right. (FULL) - Can fire projectiles and collect power-ups.
	**Enemy character** (MVP) - Spawns periodically and moves across the screen. (FULL) - Can fire projectiles at the player.
	**Projectile** (FULL) - A pellet fired by the Player or Enemiest
	**PowerUp - Projectile** (FULL) - A collectible that increases the size of the Player's projectiless
	**PowerUp - Slay** (FULL) - A collectible that destroys all enemies and their projectiles

Step 2: Map Gameplay to Code Classes

Next, let's translate those game concepts into the actual JavaScript classes and files we will create. This is our technical blueprint.

Overview of Classes/Files (from Developer's Perspective):

`Class/File`	Responsibility
`game`	*The entry point. This script creates the main Phaser Game configuration and tells it which Scene to launch first..*
`PlayScene`	*The heart of our game. This class will manage all the game objects, handle spawning, process collisions, and contain the main game loop (`preload`, `create`, `update`).*
`Player`	*Defines the `Player` character. This class will handle keyboard input for movement and firing projectiles.*
`Enemy`	*Defines a single `Enemy`. This class will manage its movement and, eventually, its attack patterns.*
`Projectile`	*Defines a `Projectile`. A simple class that moves in a straight line after being fired.*
`PowerUp`	*Defines a collectible `PowerUp`. It will have a specific type (e.g., 'slay') that determines its effect when the player touches it..*

Step 3: Define Our Development Roadmap

A key principle of modern software development is to build iteratively. We won't try to build everything at once. Instead, we'll first build the **MVP (Goals 1-8)** to get a simple, playable game. After that, we will add the remaining features as a series of **Enhancements.**

This roadmap will be our guide for the entire lab.

MVP Features - Core Build - Roadmap	**Buildout features - Full Build - Roadmap**
Goal 1: Set-up & Launch (Empty) Phaser Game Project Goal 2: Stub an Empty Phaser Scene Goal 3: Scene with a Background Goal 4: Add a Player into Scene Goal 5: Move Player with Inputs & Physics Goal 6: Add Enemies into Scene Goal 7: Enemies move with Physics Goal 8: Collision Detections & Game Over	Buildout Feature 1: Customize Player Hitbox Buildout Feature 2: Scrolling Background Buildout Feature 3: Player Walk Animation Buildout Feature 4: Top Score Challenge Buildout Feature 5: Player Projectiles Buildout Feature 6: Power-ups Buildout Feature 7: Enemy Projectiles Buildout Feature 8: Title Screen & Scene Management

Dev Cycle 1:
Core Version - MVP Features

Part 1: Building the Minimal Viable Product (MVP)

It's time to write some code. In this first major section of the lab, we will build a complete **Minimal Viable Product (MVP)** from scratch.

The goal here is to create a barebones but fully playable version of our game. This is a core principle of agile development: build a simple, working foundation first, then add the cool features (the "enhancements") later.

Your Objective for Part 1

By the end of this section, you will have a simple but operational dodger game. The final result will include:

- A game scene that renders a background and game objects.
- A player character that you can move in all four directions with keyboard input.
- Enemies that spawn periodically and move across the screen.
- A complete game loop: a collision between the player and an enemy will trigger a "Game Over" and restart the scene.

Table of Contents

Goal 1: Set-up & Launch (Empty) Phaser Game Project	8
Goal 2: Stub an Empty Phaser Scene	10
Goal 3: Scene with a Background	12
Goal 4: Add a Player into Scene	14
Goal 5: Move Player with Inputs & Physics	17
Goal 6: Add Enemies into Scene	20
Goal 7: Enemies move with Physics	23
Goal 8: Collision Detections & Game Over	25

Goal 1: Set-up & Launch *(Empty)* Phaser Game Project

'Approach' → Plan phase

Our first step is to create the basic file and folder structure for a simple, non-modular Phaser project.

The project root directory will contain.
- `assets/`: directory to hold all of the game's art
- `src/`: directory to hold all of the game's code
- `index.html`: file that launches your game in the browser

Goal: Successfully set up the project files and launch an empty, black 640x480 Phaser game window.

API References:
- https://photonstorm.github.io/phaser3-docs/Phaser.Game.html

'Apply' → Do phase

Step 1: Create the index.html File

Since JavaScript files can't be opened directly in a browser, we need an HTML file to load them. Create `index.html` in your project's root directory. This file will load the Phaser library (`phaser.js`) and our main game script (`game.js`).

Important: The order of the `<script>` tags matters. Since `game.js` will use code from the Phaser library, `phaser.js` must be loaded first.

index.html

```
<html>
    <head>
        <title> Phaser Game - Dodger </title>
        <script src="./src/phaser.js">    </script>
        <script src="./src/game.js">      </script>
    </head>
    <body></body>
</html>
```

Step 2: Create the game.js File

To initialize a Phaser game, we need to provide it with a configuration object. Create a `game.js` file inside the `src/` directory.

This script defines a `config` object that, at a minimum, specifies the `width` and `height` of the game screen. We then create a new instance of the `Phaser.Game` class, passing our configuration to it. This single line is what starts the game.

src/ → *game.js*

```
const config = new Object();

config.width    = 640;                                          //Width of viewport
config.height   = 480;                                          //Height of viewport

const game = new Phaser.Game(config);                           //New game with configs
```

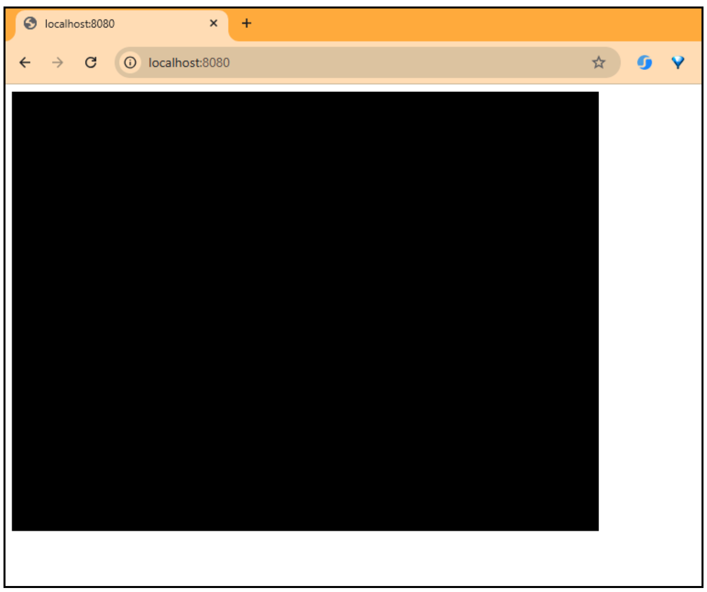
'Assess' → Test phase

Test Instructions:

1. Launch an HTTP server to host the index.html from the project via the HTTP protocol.
2. Open the browser to the specified URL with the port #, in my case it's localhost:8080

Expected Output (Viewport):

A 640x480 resolution black screen should render in the viewport, successfully launching an empty Phaser game



Expected Output (Devtools Console):

The console should display that Phaser is successfully launched with its version # and its A/V rendering settings

Goal 2: Stub a *(Empty)* Phaser Game Scene

'Approach' → Plan phase

Our empty game window isn't very useful yet. We need to add a **Scene**, which is the container for all of our game's objects, logic, and state.

We will use **inheritance** to create our own custom `PlayScene` class that extends Phaser's base `Phaser.Scene` class. This allows our custom scene to plug directly into the Phaser engine. As a best practice, our scene will include the four core lifecycle methods: `constructor`, `preload`, `create`, and `update`.

API References:

- https://photonstorm.github.io/phaser3-docs/Phaser.Scene.html

'Apply' → Do phase

Step 1: Create the PlayScene.js Class

Create a new file named `PlayScene.js` inside the `src/` directory. This file will define our main game scene. A Phaser Scene has four distinct phases:

- `constructor()`: Called when the scene is first created.
- `preload()`: Used to load all necessary assets (images, audio).
- `create()`: Used to set up the initial state of the scene (create the player, enemies, etc.).
- `update()`: The main game loop, called on every frame.

The `super('play')` call in the constructor is important; it registers this scene with Phaser's Scene Manager using the unique key `'play'`.

`src/` → *PlayScene.js*

```
class PlayScene extends Phaser.Scene {
    //construct new scene
    constructor() {
        super('play');      //set this scene's id within superclass constructor
    }

    //preload external game assets
    preload() {
    }

    //create game data
    create() {
    }

    //Update game data
    update() {
    }
}
```

Step 2:

Now that our `PlayScene` class is defined, we need to tell our main game instance to use it. Open `game.js` and add the `scene` property to the configuration object. The `scene` property should be an array of all the scenes your game will use. For now, it will just contain our `PlayScene`.

`src/` → *game.js*

```
const config = new Object();

config.width   = 640;                                  //Width of viewport
config.height  = 480;                                  //Height of viewport
config.scene   = [ PlayScene ];                        //Scenes for this game

const game = new Phaser.Game(config);                    //New game with configs
```

Step 3:

Every new JavaScript file must be loaded by our `index.html`. Open it and add a new `<script>` tag for `PlayScene.js`. The order is critical: `PlayScene.js` must be loaded after `phaser.js` (because it extends a Phaser class) but before game.js (because game.js references it).

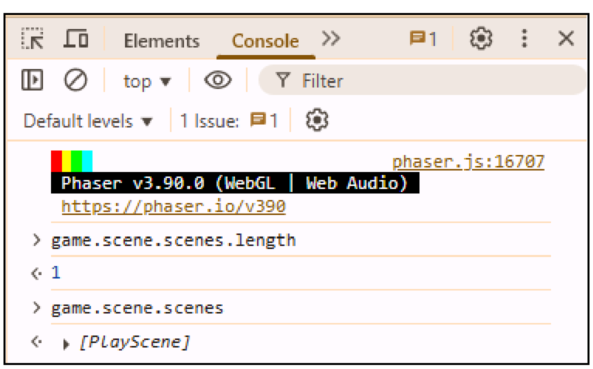
index.html

```
<html>
    <head>
        <script src="./src/phaser.js">    </script>
        <script src="./src/PlayScene.js"> </script>
        <script src="./src/game.js">      </script>
    </head>
    <body></body>
</html>
```

'Assess' → Test phase

Testing:

The Scene we've added has no visual elements yet, so the game will still appear as a black screen. However, we can use the browser's developer tools to verify that Phaser has successfully loaded our scene. Use the console logging from the screenshot to verify this goal is complete.



Test Instructions:

1. Open the devtools console
2. Access the Phaser game object in memory
3. Verify the length of the scenes list is 1
4. Verify PlayScene is defined in the scenes' list
    - The length of list is 1
    - The element is PlayScene

Goal 3: Scene with a Background

'Approach' → Plan phase

Now that our Scene is running, let's give it a visual backdrop. We'll refactor the `PlayScene` to load a background image and display it, which will be the first visible asset in our game.

To do this, we'll use two key features of a Phaser Scene:

1. **The Loader (`this.load`)**: Used in the `preload` method to load assets like images from our `assets/` folder.
2. **The GameObject Factory (`this.add`)**: Used in the `create` method to add game objects, like our background image, to the Scene.

API References:

- https://photonstorm.github.io/phaser3-docs/Phaser.Scene.html#add
- https://photonstorm.github.io/phaser3-docs/Phaser.GameObjects.GameObjectFactory.html#image

'Apply' → Do phase

Step 1: Preload the Background Image

First, we need to tell Phaser to load our image file into memory. We do this inside the `preload()` method. By setting `this.load.path`, we tell Phaser the default directory for all assets. Then, `this.load.image()` queues the image for loading. We give it a unique key (`'background'`) which we'll use to reference it later, and the filename (`'background.png'`).

Phaser will not proceed to the `create()` method until all assets in the `preload()` queue have finished loading.

PlayScene.js → preload()

```
preload() {
    this.load.path = 'assets/';                                     //Define file path
    this.load.image( 'background', 'background.png' );              //Load tile images
}
```

Step 2: Create a Helper Method for the Map

As our game grows, the `create()` method can become very cluttered. It's a good practice to organize creation logic into smaller, well-named helper methods. We'll create a `create_map()` helper to handle setting up our background. The main `create()` method's only job for now is to call this new helper.

PlayScene.js → create()

```
//Create Game World
create(){
    this.create_map();          // create level
}
```

Step 3: Add the Image to the Scene

Now, implement the `create_map()` helper method. Inside it, we use `this.add.image()` to create an image game object and add it to our scene's display list.

The three arguments are the **x-coordinate**, the **y-coordinate**, and the **asset key** for the image we loaded in preload(). By default, Phaser treats the x/y coordinates as the center point of the image, so `(640/2, 480/2)` will perfectly center our background in the game window.

PlayScene.js → create_map()

```
//Load level
create_map() {
        this.add.image(640/2, 480/2, 'background');
}
```

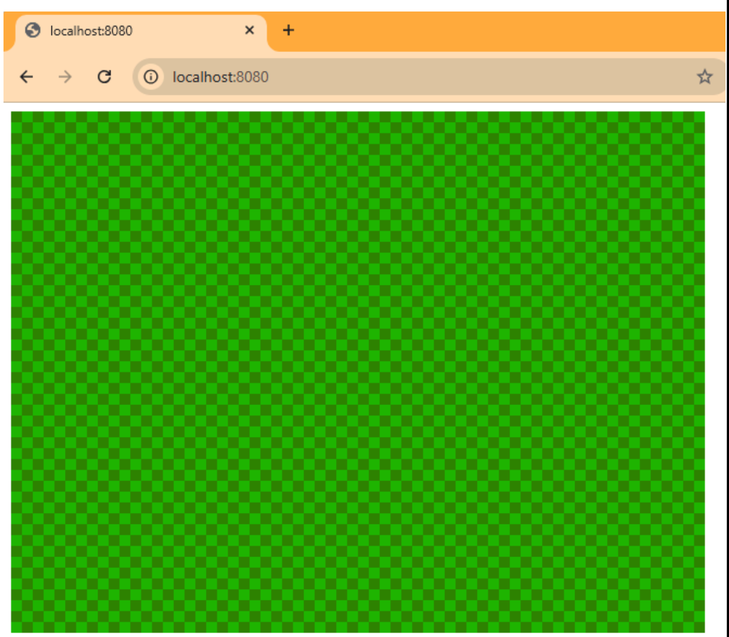
'Assess' → Test phase

Testing:

1. Make sure you have a `background.png` file inside your `assets/` directory.
2. Launch your local server and open the game in your browser.

Expected Output (Viewport):

Your background image should now be rendered, filling the entire 640x480 game screen.



Test Instructions:

1. Launch an HTTP server to host game
2. Open HTML in browser from Server

Expected Output (viewport):

- A background image should now render for the Scene

Goal 4: Add Player into Scene

'Approach' → Plan phase

Let's add our first interactive game object: the `Player`. The player will need to be a visible sprite that can also participate in Phaser's physics system for movement and collisions.

To achieve this, we will create a custom `Player` class that inherits from `Phaser.Physics.Arcade.Sprite`. This gives our player all the built-in rendering and physics capabilities of a standard Phaser sprite, which we can then extend with our own custom properties and behaviors.

'Apply' → Do phase

Step 1: Preload the Player Image

Just like with the background, the first step is to load the player's image asset in the `preload()` method of our `PlayScene`.

src/→ *PlayScene.js → preload()*

```
preload() {
    this.load.path = 'assets/';                                  //Define file path
    this.load.image( 'background', 'background.png' );          //Load background image
    this.load.image( 'player', 'player.png' );                   //Load player image
}
```

Step 2: Create the Player.js Class

Next, create a new file at `Player.js`. This class will define our player.

The constructor is the most important part:

- `extends Phaser.Physics.Arcade.Sprite`: This establishes the inheritance.
- `super(scene, 300, 200, 'player')`: This calls the parent sprite's constructor to set up the essential properties: the scene it belongs to, its starting x/y position, and the texture key to use.
- `this.depth = 2`: We set a custom depth property. A higher depth value means this sprite will be drawn on top of sprites with a lower value, ensuring the player appears in front of the background.
- `scene.add.existing(this)`: This is a crucial step. It tells the scene to add this newly created Player object to its display and update lists, making it visible and active in the game. It's a common pattern for custom game object classes to handle adding themselves to the scene.

`src/` → *Player.js*

```
class Player extends Phaser.Physics.Arcade.Sprite {
    constructor(scene) {
        super(scene, 300, 200, 'player');
        this.depth = 2;
        this.speed = 200;
        scene.add.existing(this);
    }
}
```

Step 3: Add a create_player Helper Method

In `PlayScene.js`, we'll continue our pattern of using helper methods. Add a call to `this.create_player()` within your `create()` method.

`src/` → *PlayScene.js* → *create()*

```
//Create Game World
create(){
    this.create_map();                          //helper method: create map
    this.create_player();                       //helper method: create player
}
```

Step 4: Instantiate the Player

Now, implement the `create_player()` helper method. Inside it, we create a new instance of our `Player` class. We store it in a scene property, `this.player`, so we can easily access it later. Notice we pass `this` (a reference to the `PlayScene` instance itself) to the `Player` constructor, which is required by its `constructor(scene)` signature.

`src/` → *PlayScene.js* → *create_player()*

```
//Create Game World
create_player() {
    this.player = new Player(this);             //create player
}
```

Step 5: Load the New Player.js File

Finally, we need to load our new `Player.js` file in `index.html`. The order is very important. Since `PlayScene.js` creates a `new Player()`, the `Player.js` file **must be loaded before** `PlayScene.js`.

index.html

```
<html>
    <head>
        <script src="./src/phaser.js">    </script>
        <script src="./src/Player.js">    </script>
        <script src="./src/PlayScene.js"> </script>
        <script src="./src/game.js">      </script>
    </head>
    <body></body>
</html>
```

'Assess' → Test phase

Test Instructions:

1. Make sure you have a `player.png` file inside your `assets/` directory.
2. Launch your local server and open the game in your browser.

Expected Output (Viewport):

The player sprite should now be rendered on the screen, appearing on top of the background image.

Goal 5: Move Player with Inputs & Physics

'Approach' → Plan phase

Let's make our game interactive by allowing the player to move. We'll use the arrow keys for input and leverage Phaser's built-in **Arcade Physics** engine to handle the movement.

First, we'll enable the physics system for our entire game. Then, we'll give our `Player` object a physics "body," which allows the engine to control its position via **velocity**. Velocity is a vector with `x` and `y` components. A positive x-velocity moves the object right, negative moves it left. A positive y-velocity moves it down, and negative moves it up. By changing the player's velocity based on keyboard input, we can bring it to life.

'Apply' → Do phase

Step 1: Enable Arcade Physics

In `src/game.js`, we need to add a `physics` property to our main game configuration. This tells Phaser to start the physics engine and make it available to all scenes.

src/ → *game.js*

```
const config = new Object();

config.width   = 640;                                             //Width of viewport
config.height  = 480;                                             //Height of viewport
config.scene   = [ PlayScene ];                                   //Scenes for this game
config.physics = { default:'arcade' };                            //Physics for collisions

const game = new Phaser.Game(config);                             //New game with configs
```

Step 2: Add a Physics Body to the Player

Now, let's update the `Player` constructor.

- `scene.physics.add.existing(this)`: This gives our Player sprite a physics body, allowing it to have properties like velocity.
- `this.setCollideWorldBounds(true)`: This method prevents the player from moving outside the boundaries of the game screen.
- `scene.input.keyboard.addKeys(...)`: We use the scene's input manager to create a cursor keys object that we can poll to check if the arrow keys are being pressed. We store this in this.buttons for easy access.

src/ → *Player.js* → *constructor()*

```
constructor(scene) {
    super(scene, 300, 200, 'player');
    this.depth = 2;
    this.speed = 200;

    scene.add.existing(this);
    scene.physics.add.existing(this);
    this.setCollideWorldBounds(true);      //don't go out of the map
    this.buttons = scene.input.keyboard.addKeys('up,down,left,right');
}
```

Step 3: Create the Player's move Method

This new method in the `Player` class will contain all of its movement logic. This is a great example of **encapsulation** - the `Player` object is responsible for knowing how it should move. The logic follows a simple "reset and check" pattern on every frame:

1. First, set the player's velocity to zero. This ensures the player stops moving if no key is pressed.
2. Then, check each arrow key. If a key `isDown`, set the velocity accordingly.

src/ → *Player.js* → *move()*

```
//move player
move()  {
    // reset velocity
    this.body.velocity.x = 0;
    this.body.velocity.y = 0;

    // take care of character movement
    if ( this.buttons.up.isDown ) {
        this.body.velocity.y = -this.speed;
     }
     if ( this.buttons.down.isDown ) {
        this.body.velocity.y = this.speed;
     }
     if ( this.buttons.left.isDown ) {
        this.body.velocity.x = -this.speed;
     }
     if ( this.buttons.right.isDown ) {
        this.body.velocity.x = this.speed;
     }
}
```

Step 4: Call the Player's Logic from the Scene's update Loop

The `PlayScene` is the orchestra conductor of our game. Its main `update` loop is responsible for telling all the game objects what to do on each frame. This is a design principle called **Separation of Concerns.**

First, add a call to a new helper method, `update_player()`, from the main `update()` loop.

,

src/ → *PlayScene.js* → *update()*

```
//update game state
update() {
    this.update_player();
}
```

Step 5: Implement the update_player Helper

Finally, implement the `update_player()` helper. Its only job is to call the `move()` method on our player instance. This delegates the responsibility of movement from the scene to the player itself.

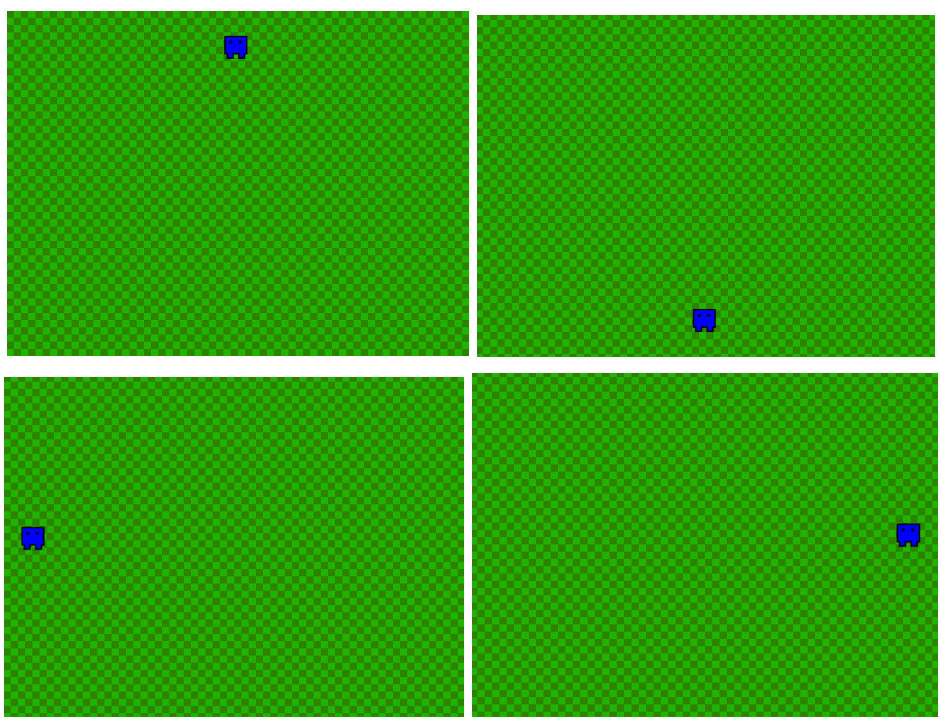
`src/` → *PlayScene.js* → *update()*

```
//update game state
update_player() {
    this.player.move();
}
```

'Assess' → Test phase

Testing:

- The player object should remain stationary when no arrow button is pressed
- The player object should move when any arrow button is pressed
- The player object should stop after the arrow button is stopped being pressed
- When you press the up-arrow butto,n the player object should move up
- When you press the down-arrow button, the player object should move down
- When you press the right-arrow button the player object should move right
- When you press the left-arrow button, the player object should move left
- When you press both a horizontal arrow and vertical arrow, the player object should move diagonally



Goal 6: Add Enemies into Scene

'Approach' → Plan phase

A dodger game isn't much fun without things to dodge. Let's add enemies to the scene. We will define an `Enemy` class that will serve as a blueprint for all enemy instances. Like our `Player`, the `Enemy` will need to be a visible, physics-enabled sprite, so it will also inherit from `Phaser.Physics.Arcade.Sprite`.

'Apply' → Do phase

Step 1: Preload the Enemy Image

In `PlayScene.js`, add a line to the `preload()` method to load our enemy's image asset.

src/→ *PlayScene.js → preload()*

```
preload() {
    this.load.path = 'assets/';                                      //Define file path
    this.load.image( 'background', 'background.png' );               //Load background image
    this.load.image( 'player', 'player.png' );                       //Load player image
    this.load.image( 'enemy', 'enemy.png' );                         //Load enemy image
}
```

Step 2: Create the Enemy.js Class

Create a new file at `src/Enemy.js`. This class will be simpler than our `Player` class. Its constructor will take the `scene` and a `position` object (`{x, y}`) as arguments, create the sprite, and add itself to the scene.

`src/` → *Enemy.js*

```
class Enemy extends Phaser.Physics.Arcade.Sprite {
    constructor(scene, position) {
        super(scene, position.x, position.y, 'enemy');
        this.depth = 2;

        scene.add.existing(this);
    }
}
```

Step 3: Add the create_enemies Helper Method

In `PlayScene.js`, add a call to a new helper method, `create_enemies()`, from within the main `create()` method.

`src/` → *PlayScene.js* → *create()*

```
//create game data
create(){
    this.create_map();                       //helper method: create map
    this.create_player();                    //helper method: create player
    this.create_enemies();                   //helper method: create enemies
}
```

Step 4: Create a Timed Event to Spawn Enemies

Implement the `create_enemies()` helper. Instead of creating all enemies at once, we want them to spawn continuously. We'll use Phaser's **Time Manager** (`this.time`) to create a looping event.

First, we'll create an array, `this.enemies`, to keep track of all active enemy instances. Then, we'll configure a timed event to call a `spawn_enemy` function every 200 milliseconds.

`src/` → *PlayScene.js* → *create_enemies()*

```
create_enemies() {
    this.enemies = [];

    const event = new Object();
    event.delay         = 200;
    event.callback      = this.spawn_enemy;
    event.callbackScope = this;
    event.loop          = true;

    this.time.addEvent(event, this);
}
```

Step 5: Implement the spawn_enemy Callback

Now, create the `spawn_enemy()` function that the timer will call. This function's job is to:

1. Determine a random x/y position on the screen.
2. Create a `new Enemy()` at that position.
3. Add the new enemy instance to our `this.enemies` tracking array.

`src/` → *PlayScene.js* → *spawn_enemies()*

```
spawn_enemy() {
    const position = {};
    position.x = Phaser.Math.Between(0, 640);
    position.y = Phaser.Math.Between(0, 480);

    const monster =  new Enemy(this, position);
    this.enemies.push(monster);
}
```

Step 6: Load the New Enemy.js File

Finally, load our new `Enemy.js` file in `index.html`. Since `PlayScene.js` creates instances of `Enemy`, the `Enemy.js` file **must be loaded before** `PlayScene.js`.

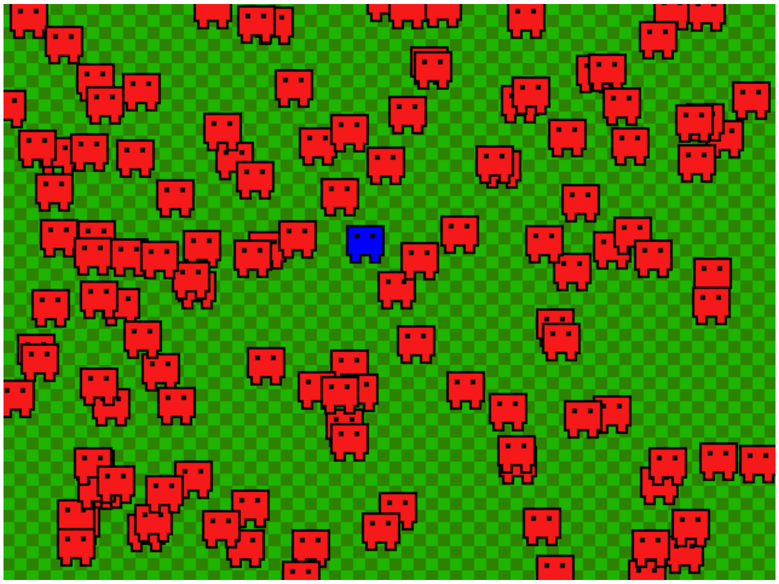
index.html

```
<html>
    <head>
        <script src="./src/phaser.js">    </script>
        <script src="./src/Player.js">    </script>
        <script src="./src/Enemy.js">     </script>
        <script src="./src/PlayScene.js"> </script>
        <script src="./src/game.js">      </script>
    </head>
    <body></body>
</html>
```

'Assess' → Test phase

Testing:

Make sure you have an `enemy.png` file in your `assets/` folder. Launch the game. Enemies should begin appearing randomly all over the screen at a rapid pace..



Goal 7: Move Enemies in Scene with Physics

'Approach' → Plan phase

Our enemies currently spawn randomly all over the screen and don't move, which isn't very challenging. Let's refactor our code so that all enemies spawn just off the right edge of the screen and move horizontally to the left at a random speed.

'Apply' → Do phase

Step 1: Encapsulate Movement Logic in the Enemy Class

A core principle of Object-Oriented Programming is **encapsulation** - an object should be responsible for its own data and behavior. We'll make our `Enemy` class responsible for its own movement.

In the Enemy constructor, we will:

1. Add the enemy to the physics system using `scene.physics.add.existing(this)`. This gives it a physics `body`.
2. Immediately set the `x` component of its body's velocity to a random negative number. This will cause the enemy to start moving left the instant it is created.

src/ → *Enemy.js*

```
class Enemy extends Phaser.Physics.Arcade.Sprite {
    constructor(scene, position) {
        super(scene, position.x, position.y, 'enemy');
        this.depth = 2;

        scene.add.existing(this);
        scene.physics.add.existing(this);
        this.body.velocity.x = -Phaser.Math.Between(120, 300);
    }
}
```

Step 2: Update the Spawning Position

Now, we need to change where the enemies are created. In `PlayScene.js`, refactor the `spawn_enemy` method so that it no longer spawns enemies at a random x-coordinate.

Instead, we will set a fixed `x` value of `640 + 32`. This will cause the enemy to spawn 32 pixels beyond the right edge of the screen, giving it a moment to appear before it flies into view. The `y` position will remain random.

src/ → *PlayScene.js* → *spawn_enemy()*

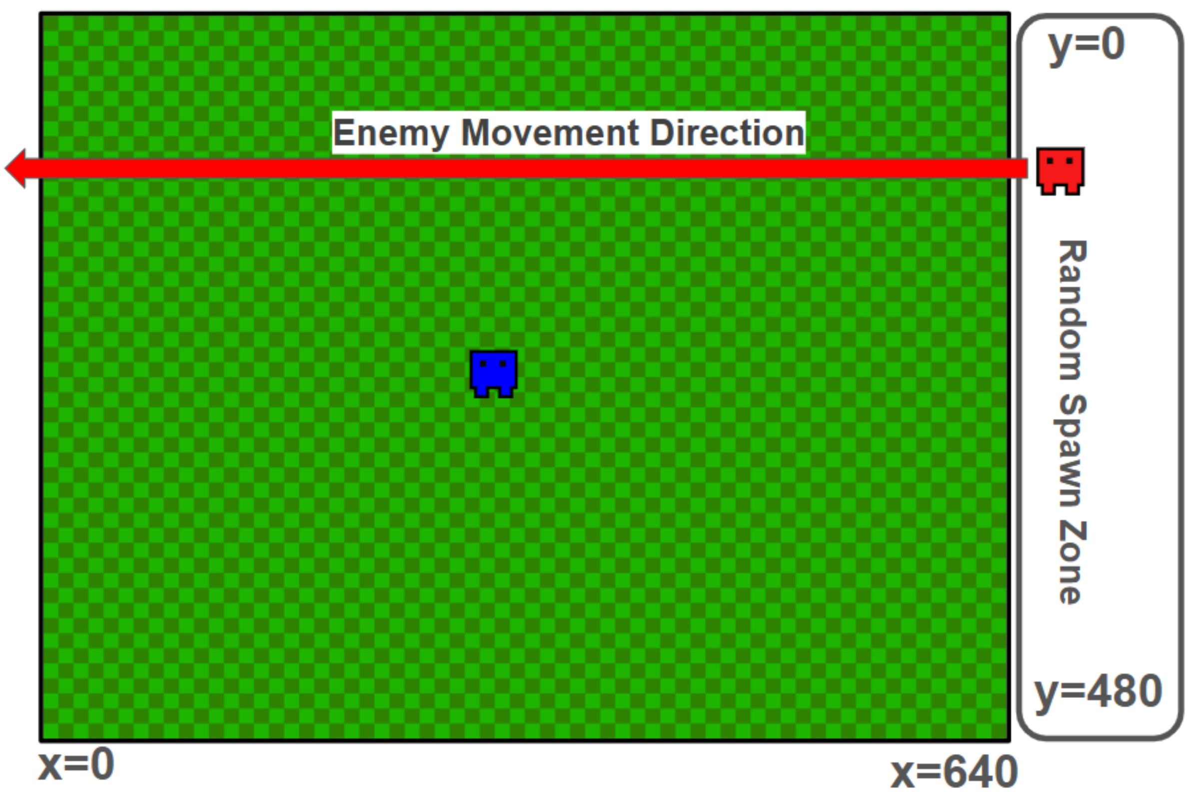
```
spawn_enemy() {
    const config = {};;
    config.x = 640 + 32;
    config.y = Phaser.Math.Between(0, 480)

    const monster =  new Enemy(this, config);
    this.enemies.push(monster);
}
```

'Assess' → Test phase

Testing:

Launch the game. All enemies should now spawn from a random vertical position just off the right side of the screen and travel horizontally to the left at varying speeds. For now, they will pass harmlessly over the player, as we have not yet implemented collision detection.



Goal 8: Collision Detections & Game Over

'Approach' → Plan phase

This is the final goal to complete our MVP. We will use Phaser's physics system to detect when the `Player` and an `Enemy` overlap. When a collision occurs, it will trigger a "Game Over" state. In our arcade-style game, the only lose condition is the player touching an enemy, which will cause the scene to flash and then restart, creating a complete and playable game loop.

'Apply' → Do phase

Step 1: Create a Collision Setup Method

In `PlayScene.js`, add a call to a new helper method, `create_collisions()`, from the main `create()` method. This continues our pattern of keeping the `create()` method clean and readable.

`src/` → *PlayScene.js* → *create()*

```
//create game data
create() {
    this.create_map();                              //create map
    this.create_player();                           //create player
    this.create_enemies();                          //create enemies
    this.create_collisions();                       //create physics-related behaviors
}
```

Step 2: Add a Physics Overlap Check

Now, implement the `create_collisions()` helper. Inside, we'll use this.`physics.add.overlap()` to tell Phaser to constantly check for collisions between two objects.

This method takes several arguments: `(object1, object2, callback, processCallback, context)`.

- `object1`: The first object/group to check. We'll use `this.player`.
- `object2`: The second object/group. We'll use `this.enemies`, our array of all active enemies.
- `callback`: The function to call when an overlap happens. We'll name it `game_over`.
- `processCallback`: An optional function to run first; we don't need it, so we pass `null`.
- `context`: The value of `this` inside the callback. We pass `this` so it refers to our `PlayScene`.

`src/` → *PlayScene.js* → *create_collisions()*

```
//sets up overlap collisions behaviors
create_collisions() {
    this.physics.add.overlap(this.player,this.enemies,this.game_over,null,this);
}
```

Step 3: Implement the game_over Callback Function

Finally, create the `game_over` method. This function will be executed the moment a player-enemy collision is detected.

- `this.cameras.main.flash()`: We'll use the scene's primary camera to create a brief white flash. This provides instant visual feedback to the player that something bad happened.
- `this.scene.restart()`: We'll use the scene's manager to restart the current scene. This effectively resets the game to its initial state, completing our game loop.

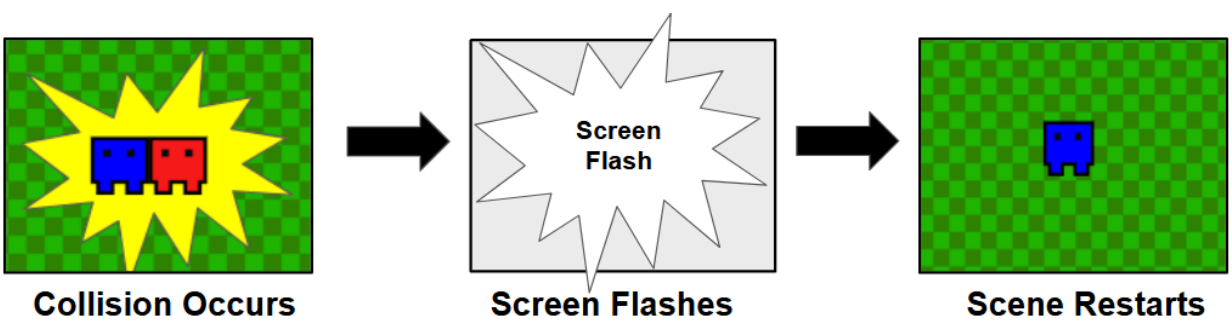
`src/` → *PlayScene.js* → *game_over()*

```
game_over() {
    this.cameras.main.flash();
    this.scene.restart();
}
```

'Assess' → Test phase

Testing:

Launch the game. Move the player into the path of an oncoming enemy. When the two sprites overlap, the screen should flash white for a moment, and then the scene should immediately restart, with the player back at their starting position and new enemies beginning to spawn.



Dev Cycle 2:
Full Version - Buildout Features

Full version Goal:

Congratulations on building a complete Minimal Viable Product! You now have a solid, working foundation. In this next section, we will transform that simple MVP into a polished and exciting game by adding a series of "enhancements."

These are the features that will add depth, challenge, and visual flair to make your game feel complete.

Our Enhancement Roadmap

We will add the following features, building upon the MVP you just completed. While many of these could be done in any order, we recommend following this sequence as they build upon each other logically.

Enhancement 1: Customize Player Hitbox	29
Enhancement 2: Scrolling Background	31
Enhancement 3: Player/Enemy Walk Animation	33
Enhancement 4: Top Score Challenge	37
Enhancement 5: Player Projectiles	43
Enhancement 6: Power-ups	52
Enhancement 7: Enemy Projectiles	56
Enhancement 8: Title Screen & Scene Management	64

(Note: There is one key dependency: Enhancement 6 (Power-ups) requires the player to be able to fire projectiles, which are implemented in Enhancement 5.)

An Important Note on Applying Changes

Each enhancement guide will show you the new code to **add or change**, starting from the completed MVP. However, as you complete each enhancement, you are modifying your new, feature-rich codebase.
Be careful not to accidentally delete or overwrite the code you added in a previous enhancement. For example, when you start Enhancement 2, make sure you don't remove the hitbox code you added in Enhancement 1.

Pro Tip: This is why version control systems like Git are essential in professional development. A great habit is to save a backup of your project folder or make a Git commit after successfully completing each enhancement.

The Final Result *(Deliverable)*

By the end of this section, you will have a complete, feature-rich version of the game that realizes all of our initial design goals, including animations, scoring, power-ups, and more.

.

Enhancement 1: Customize Player Hitbox

'Approach' → Plan phase

A small but high-impact improvement we can make is to customize the player's **hitbox**. The hitbox is the invisible, physics-based shape used for collision detection. By default, it perfectly matches the size of the sprite's image.

We will make the hitbox slightly smaller than the visual art. This is a common technique in arcade-style games to make the movement feel more responsive and forgiving, as it reduces frustrating "unfair grazes" where the edge of a sprite seems to get clipped.

'Apply' → Do phase

Step 1: Shrink the Physics Body

In `Player.js`, find your constructor method. After the line that adds the physics body, add a new line to resize it.

We'll use the `this.body.setSize()` method. It takes three arguments: `(width, height, center)`.

- We'll set the new width and height to be 16 pixels smaller than the sprite's actual dimensions.
- We'll pass `true` as the third argument to automatically re-center the new, smaller hitbox within the sprite.

`src/` → *Player.js* → *constructor*

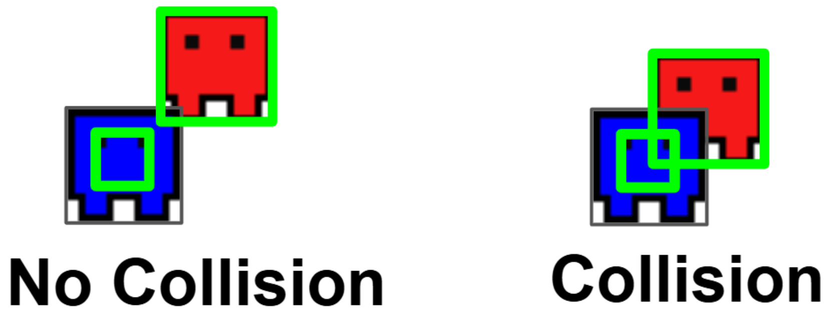
```
constructor(scene) {
    super(scene, 300, 200, 'player');
    this.depth = 2;
    this.speed = 200;

    scene.add.existing(this);
    scene.physics.add.existing(this);
    this.setCollideWorldBounds(true);                    //don't go out of the map
    this.body.setSize(this.width-16, this.height-16);    // 16x16 for 32x32

    this.buttons = scene.input.keyboard.addKeys('up,down,left,right');
}
```

'Assess' → Test phase

Testing:
Launch the game. You should now be able to fly the player so that the edge of its sprite slightly overlaps with an enemy sprite without triggering a "Game Over." The collision will only register when the smaller, invisible hitboxes touch.



Enhancement 2: Scrolling Background

'Approach' → Plan phase

In the MVP, our static background makes the player feel stationary. We'll introduce a scrolling background to create a sense of forward motion and travel. The goal is to give the game a constant feeling of movement, allowing the player to focus on dodging enemies. We will achieve this by replacing our static image with a seamlessly tileable texture that loops continuously.. (Optionally, multiple layers scrolling at different speeds - parallax - can add depth.)

'Apply' → Do phase

Step 1: Replace the Image with a TileSprite

In `PlayScene.js`, we need to refactor our `create_map` method. Instead of using a static `this.add.image`, we will use this.`add.tileSprite`.

A `TileSprite` is a special type of image object designed for creating scrolling backgrounds. Its key feature is that it repeats its texture seamlessly, as if the edges were glued together. This allows us to modify its `tilePosition` property later to create a continuous scroll. For this to work, your `background.png` file must be a **tileable texture**, meaning its left and right edges match up perfectly.

We'll store the created `TileSprite` in a new `this.background` property so we can access it in our `update` loop.

`src/` → *PlayScene.js* → *create_map()*

```
//Load level
create_map() {
    this.background = this.add.tileSprite(640/2, 480/2, 640, 480, 'background');
}
```

Step 2: Add a Helper Method to the update Loop

To maintain the clean structure of our core `update` method, we will add a call to a new helper method, `update_background()`. This continues our pattern of using `update` as a high-level "conductor" that delegates tasks to more specific functions.

`src/` → *PlayScene.js* → *update()*

```
update(){
    this.update_player();
    this.update_background();
}
```

Step 3: Animate the Background in the update Loop

Now, create the `update_background()` helper method. Inside this function, we will animate the background by accessing the `background` object and modifying its `tilePositionX` property on every frame.

By adding a small number to this property, we shift the texture, creating a smooth, leftward scrolling effect. The value `3` determines the speed in pixels per frame; feel free to change it to see how it affects the scroll rate.

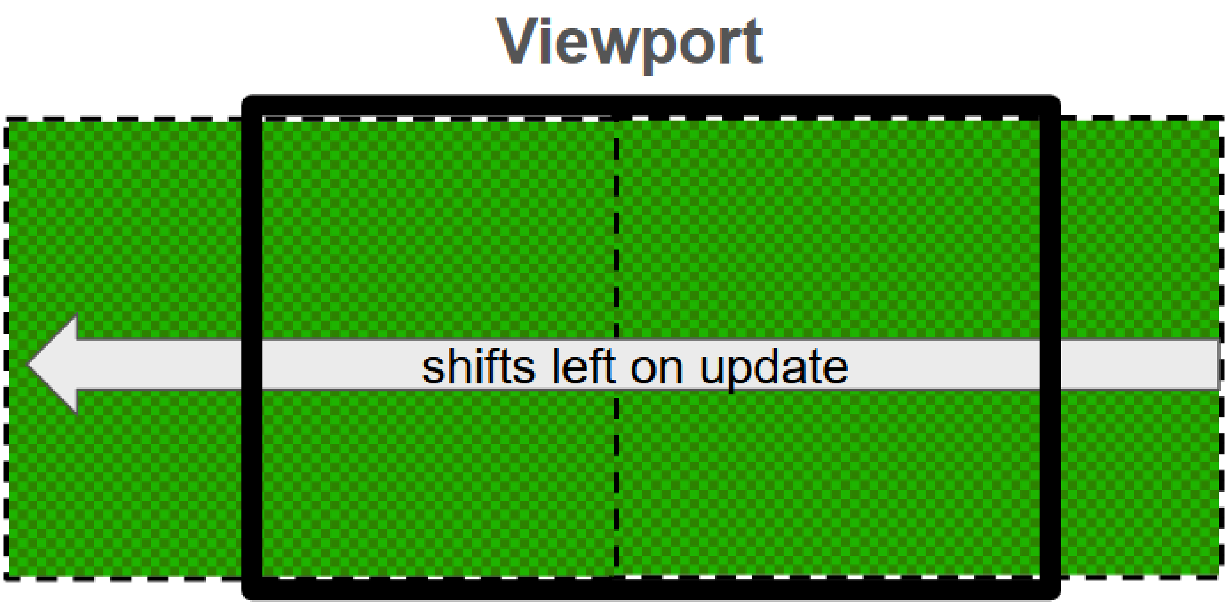
src/ → *PlayScene.js* → *update_background()*

```
update_backgtround(){
    this.background.tilePositionX += 3;
}
```

'Assess' → Test phase

Testing:

Launch the game. The background image should now be continuously scrolling from right to left at a steady speed. Because it's a `TileSprite` using a tileable texture, the pattern should repeat seamlessly with no visible gaps



Enhancement 3: Player/Enemy Walk Animation

'Approach' → Plan phase

Animating characters is a high-impact visual upgrade. We'll build a simple, two-frame looping walk animation for both the Player and the Enemy.

A key concept in Phaser is that its **Animation Manager is global**. This means we only need to **register** each animation once (in our `PlayScene`), and then any sprite instance (like a `Player` or `Enemy`) can simply play that pre-defined animation. We'll build this feature in two small, testable milestones.

'Apply' → Do phase

Milestone 1: Player Animation

Step 1: Preload the Animation Frames

In `PlayScene.js`, update the `preload()` method to load the two individual image frames that will make up our player's walk animation.

src/ → *PlayScene.js* → *preload()*

```
preload() {
    this.load.path = 'assets/';                                             //Define file path
    this.load.image( 'background', 'background.png' );                      //Load background image
    this.load.image( 'player', 'player.png' );                              //Load player image
    this.load.image( 'enemy', 'enemy.png' );                                //Load player image
    this.load.image( 'player-0', 'player-0.png' );                          //Load walk frame 0
    this.load.image( 'player-1', 'player-1.png' );                          //Load walk frame 1
}
```

Step 2: Create an Animation Helper Method

In the PlayScene's `create()` method, we'll add a call to a new helper, `create_animations()`. It's important to call this before we create the player or enemies so that the animations are registered and ready to be used.

src/ → *PlayScene.js* → *create()*

```
//create game data
create() {
    this.create_map();                          //create map
    this.create_animations();                   //create animations
    this.create_player();                       //create player
    this.create_enemies();                      //create enemies
    this.create_collisions();                   //create physics-related behaviors
}
```

Step 3: Register the Player Animation

Implement `create_animations()` in `PlayScene` and register animations with the Scene's Animation Manager (`this.anims`). Build a small config object with the standard fields—`key` (animation id), `frames` (ordered array of `{ key: <textureKey> }`), `frameRate` (frames per second), and `repeat` (use `-1` to loop)—then pass it to `this.anims.create(config)`. This defines the animation once globally; later, sprites just call `this.anims.play(key, true)` to use it without recreating it. We also safe-gaurd against adding the same animation key if it already exists in the animation manager.

`src/` → *PlayScene.js* → *create_animations*

```
//create animations
create_animations(scene){
    if ( !this.anims.exists('player-move') ){

        const anim_player_move = new Object();
        anim_player_move.key = 'player-move';                             //key to register into phaser
        anim_player_move.frames = [{key: 'player-0'}, {key: 'player-1'}]; //list of image keys for anim
        anim_player_move.frameRate = 6;                                   //speed to play animation
        anim_player_move.repeat = -1;                                     //-1 for infinite loop

        this.anims.create(anim_player_move);                              //facotory creates anim obj
    }
}
```

Step 4: Play the Animation on the Player

In Player `constructor` play the registered animation. The second arg (`true`) is `ignoreIfPlaying`. It tells Phaser not to restart the animation if it's already playing - handy when you call play repeatedly (e.g., in update) so the clip doesn't snap back to frame 0 every frame.

`src/` → *Player.js* → *constructor*

```
constructor(scene) {
    super(scene, 300, 200, 'player');
    this.depth = 2;
    this.speed = 200;

    scene.add.existing(this);
    scene.physics.add.existing(this);
    this.setCollideWorldBounds(true);      //don't go out of the map

    this.buttons = scene.input.keyboard.addKeys('up,down,left,right');
    this.anims.play('player-move',true);                          //tell anims manager to play move
}
```

(Milestone 1) Testing:

Make sure you have the player-0.png and player-1.png files in your assets/ folder. Launch the game and verify that the player sprite is now animated, cycling between the two frames.

Milestone 2: Enemy Animation

Now we'll repeat the same pattern for the Enemy.n

Step 1: Preload the Enemy Frames

Refactor the `preload` method to add the enemy frames into the load manager

`src/` → *PlayScene.js* → *preload()*

```
preload() {
    this.load.path = 'assets/';                                   //Define file path
    this.load.image( 'background', 'background.png' );            //Load background image
    this.load.image( 'player', 'player.png' );                    //Load player image
    this.load.image( 'enemy', 'enemy.png' );                      //Load player image
    this.load.image( 'player-0', 'player-0.png' );                //Load walk frame 0
    this.load.image( 'player-1', 'player-1.png' );                //Load walk frame 1
    this.load.image( 'enemy-0', 'enemy-0.png' );                  //Load walk frame 0
    this.load.image( 'enemy-1', 'enemy-1.png' );                  //Load walk frame 1
}
```

Step 2: Register the Enemy Animation

Refactor the `create_animations` method to configure the enemy move animation using similar properties and then register it into the global animation manager. We safe-gaurd against adding the same animation if it already exists.

`src/` → *PlayScene.js* → *create_animation*

```
//create animations
create_animations(scene){
    if ( !this.anims.exists('player-move') ){

        const anim_player_move = new Object();
        anim_player_move.key = 'player-move';                              //key to register into phaser
        anim_player_move.frames = [{key: 'player-0'}, {key: 'player-1'}]; //list of image keys for anim
        anim_player_move.frameRate = 6;                                    //speed to play animation
        anim_player_move.repeat = -1;                                      //-1 for infinite loop
        this.anims.create(anim_player_move);                               //facotory creates anim obj
    }
    if ( !this.anims.exists('enemy-move') ){
        const anim_enemy_move = new Object();
        anim_enemy_move.key = 'enemy-move';                               //key to register into phaser
        anim_enemy_move.frames = [{key: 'enemy-0'}, {key: 'enemy-1'}];   //list of image keys for anim
        anim_enemy_move.frameRate = 6;                                    //speed to play animation
        anim_enemy_move.repeat = -1;                                      //-1 for infinite loop
        this.anims.create(anim_enemy_move);                               //facotory creates anim obj
    }
}
```

Step 3: Play the Animation on the Enemy

In Enemy `constructor` play the registered animation. The second arg (`true`) is `ignoreIfPlaying`.

`src/` → *Enemy.js* → *constructor()*

```
class Enemy extends Phaser.Physics.Arcade.Sprite {
    constructor(scene, position) {
        super(scene, position.x, position.y, 'enemy');
        this.depth = 2;

        scene.add.existing(this);
        this.anims.play('enemy-move',true);                          //tell anims manager to play move
    }
}
```

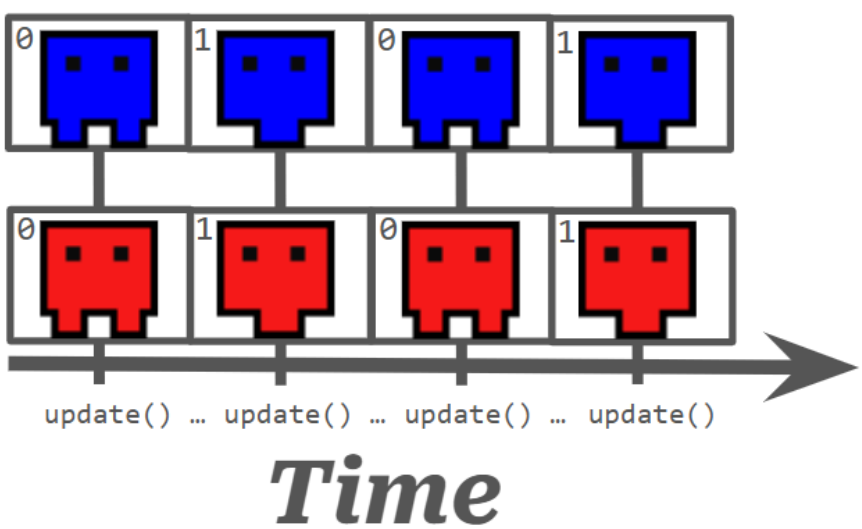
(Milestone 2) Testing:

Make sure you have the enemy-0.png and enemy-1.png files. Launch the game and verify that both the player and all spawned enemies are now animated.

'Assess' → Test phase

Testing:

Open the index.html file within the browser using an HTTP server. The player and enemy images should continually update with the animation sequence given the appearance of the characters moving their legs.



Enhancement 4: Top Score Challenge

'Approach' → Plan phase

Our game currently has a lose condition but no goal. We can significantly improve player engagement by adding a high-score challenge. This gives the player a clear objective: beat the top score.

We will build a Heads-Up Display (HUD) to show the player's **current score**, which will increase as they play. We will also track the **top score** that persists between playthroughs (but not page reloads) and even ask for the top-scoring player's name.

'Apply' → Do phase

Milestone 1: Add a Current Score Display

Add Score counter for the current game that continually increases, and reset score when the player dies.

Step 1: Create the HUD

Refactor the `create()` method to invoke a new helper method responsible for creating the scene's HUD.

src/ → *PlayScene.js → create()*

```
create() {
    this.create_map();                          //create map
    this.create_player()                        //create player
    this.create_enemies();                      //create enemies
    this.create_collisions();                   //create physics
    this.create_hud();                          //create hud
}
```

Step 2: Implement create_hud

Now, implement the `create_hud()` method. Here, we'll:

1. Initialize a `this.score` property to `0`. This will track the score for the current run.
2. Create a `Text` object using this.`add.text()` to display the score. We'll store it in `this.score_text`.
3. Set its `depth` to a high number to ensure it renders on top of all other game objects.

src/ → *PlayScene.js → create_hud()*

```
create_hud() {
    this.score = 0;
    this.score_text = this.add.text(32, 32, "");
    this.score_text.depth = 3;
    this.score_text.setColor( 'rgb(255,255,255)' );
}
```

Step 3: Add Scoring Logic

Let's give the player points. A simple way is to award one point for every enemy that spawns. In `spawn_enemy()`, add a line to increment `this.score`.

src/ → *PlayScene.js → spawn_enemy()*

```
spawn_enemy() {
    const config = {};
    config.x = 640 + 32;
    config.y = Phaser.Math.Between(0, 480)

    const monster =  new Enemy(this, config);
    this.enemies.push(monster);
    this.score +=1;
}
```

Step 4: Invoke helper update_score() in Scene's update()

To make the score on the screen update continuously, we need to update its text on every frame. We'll create a new helper method, `update_hud()`, and call it from our main `update()` loop.

src/ → *PlayScene.js → update()*

```
update() {
    this.update_player();
    this.update_score();
}
```

Step 5: Update the Score Display

Finally implement the `update_score` method by using the `setText` method of the `score_text` with the current numerical value from `score`.

src/ → *PlayScene.js → update_score()*

```
update_score() {
    this.score_text.setText("Score: " + this.score);
}
```

(Milestone 1) Testing:

Launch the game. You should see "Score: 0" in the top-left corner. The score should increase rapidly as enemies spawn. When you collide with an enemy, the game should restart and the score should reset to 0.

Milestone 2: Add a Persistent Top Score

Add a Top Score counter that tracks the highest score for all games played this session.

Step 1: Initialize Persistent State in the Constructor

Refactor the PlayScene's constructor method to intialize a `top_score` instance variable to `100`. The constructor for PlayScene is only called when the game is initially launched -- so this won't reset the `top_score` as the scene restarts between playthroughs. We default to 100 so that the player has a minimal threshold score they must first achieve to be awarded a top-score.

`src/` → *PlayScene* → *constructor*

```
constructor(){
    super('play');
    this.top_score = 100;
}
```

Step 2: Display the Top Score

Refactor the `create_hud()` method to add a game text object and initialize it into a `top_score_text` instance variable. The top score text's origin will be in the top-right corner of screen. Set the z-index depth to render above all other objects. We want to right-align the text so it stays anchored from right-most position. `setOrigin(x, y)` sets the object's anchor/pivot in normalized coordinates (0–1)—where (0,0) is top-left and (1,1) bottom-right—which determines how its position, rotation, and scaling are applied.

`src/` → *PlayScene.js* → *create_hud()*

```
create_hud(){
    this.score=0;
    this.score_text =this.add.text( 32, 32, "" );
    this.score_text.depth = 3;
    this.score_text.setColor( 'rgb(255,255,255)' );

    this.top_score_text = this.add.text( 600, 32, "" );                    //on a 640x480 size scene
    this.top_score_text.depth = 3;
    this.top_score_text.setOrigin(1,0);
}
```

Step 3: Update the Top Score Displa

Refactor the `update_score` helper method to update the `top_score_text` on each tick of the game loop.

`src/` → *PlayScene.js* → *update_score()*

```
update_score() {
    this.score_text.setText("Score: " + this.score);
    this.top_score_text.setText("Top Score: " + this.top_score);
}
```

Step 4: Check for a New High Score

Refactor the `game_over()` method to check if the current score is higher than the top score and set it if so.

src/ → *PlayScene.js* → *game_over()*

```
game_over() {
    if ( this.score >= this.top_score) {
        this.top_score = this.score;
    }
    this.cameras.main.flash();
    this.scene.restart();
}
```

(Milestone 2) Testing:

Play a game and get a score. When you die, the "Top Score" should update to reflect your score. Play again; the top score should remain, and only update if you beat it.

Milestone 3: Get Name of the Top Score Winner

When a new top score is awarded, prompt the player for their name and save it until either the page is refreshed or a new score beats it.

Step 1: Add a Persistent winner Property

Refactor the PlayScene `constructor` to include an instance variable to store the winner's name and default it to `'Top Score'`. This is placed in scene's construtor so we won't overwrite it between restarts.

src/ → *PlayScene* → *constructor*

```
constructor() {
    super('play');
    this.top_score = 0;
    this.winner = 'Top Score';
}
```

Step 2: Prompt the User for Their Name

Let's make our `game_over` logic more interactive. When a new high score is achieved, we will:

1. Pause the physics engine to freeze the game.
2. Use the browser's `prompt()` function to ask the user for their name.
3. Resume the game by restarting the scene.

Important Note: `prompt()` is a simple tool for tutorials, but it completely freezes the browser tab. This is why we pause the physics first.

`src/` → *PlayScene.js* → *game_over()*

```
game_over() {
    if ( this.score >= this.top_score) {
        this.top_score = this.score;
        this.physics.pause();                                        // freeze gameplay
        this.winner = prompt("Winner! Enter you name: ") ?? "Top Score"    // Use 'Top Score' if null
        this.input.keyboard.keys = []                                // reset phaser keys stream
    }
    this.scene.restart();
}
```

Step 3: Update the Persistent winner Property

Refactor the update_score method to set both the winner and the top score in the HUD text. We use a string template formatting to make this easier and cleaner to read.

`src/` → *PlayScene.js* → *update_score()*

```
update_score() {
    this.score_text.setText("Score: " + this.score);
    this.top_score_text.setText( `${this.winner}: ${this.top_score}` );
}
```

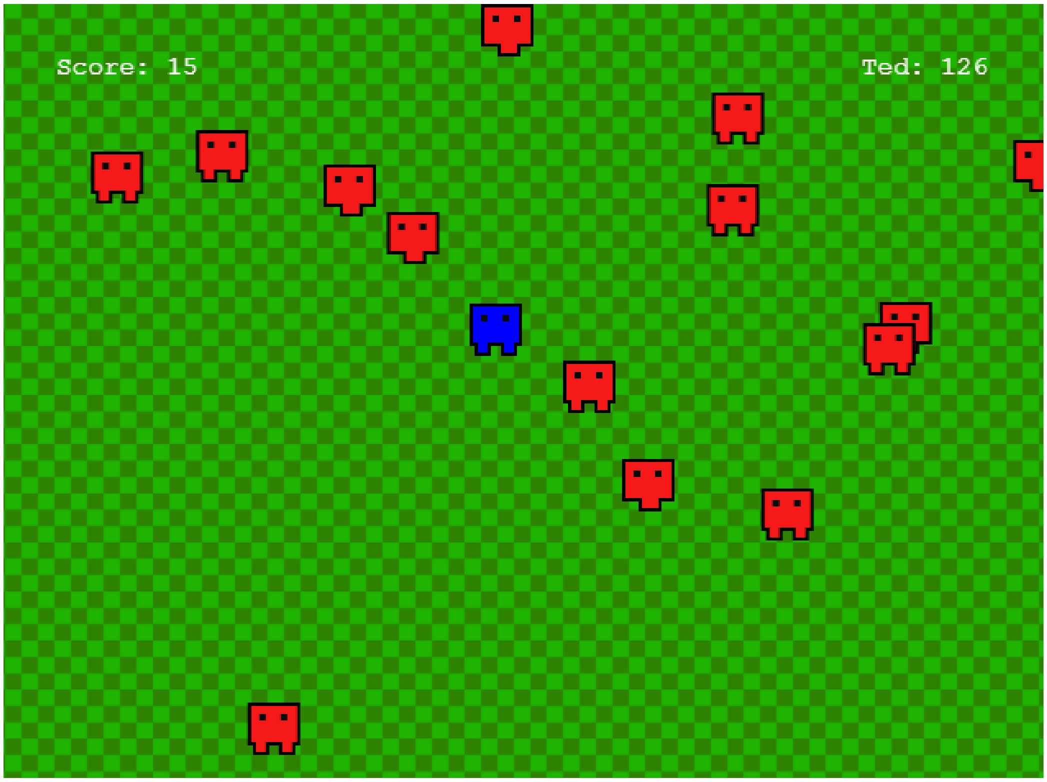
(Milestone 3) Testing:

Try running the game after these changes and verify that when a new top score is achieved that the game prompts the player for their name and it updates the HUD text along with their top score.

'Assess' → Test phase

Testing:

After completing all the milestones for this enhancement -- the game should have a HUD that displays the current score on the left and the top score along with the winner's name on the right.



Enhancement 5: Player Projectiles

'Approach' → Plan phase

A common upgrade to the dodger-styled game mechanics is to provide the player with additonal actions rather than simply to reactively dodge. Dodging is a reaction which makes the player feel powerless, constantly running for their lives without being able to act. In this enhancement, we will provide the player a new action-drive interaction which is to emit a projectile that can remove enemies. This enhancement will be fully realized across three milestones. The final goal is to create a new javascript class that represents projectiles and create one whenever the player presses or holds the spacebar. Both the projectile and an enemy are destroyed if they collide.

'Apply' → Do phase

Milestone 1: Fire projectiles from Player

The first task is to implement projectiles and trigger their creation based on a player action hitting the spacebar.

Step 1: Preload the Projectile Asset

As with all our other game objects, the first step is to load the required image asset. In the PlayScene's `preload()` method, add a line to load `projectile.png` and assign it the unique key `'projectile'`.

`src/` → *PlayScene.js* → *preload()*

```
preload() {
    this.load.path = 'assets/';                                     //Define file path
    this.load.image( 'background', 'background.png' );              //Load background image
    this.load.image( 'player', 'player.png' );                      //Load player image
    this.load.image( 'enemy', 'enemy.png' );                        //Load player image
    this.load.image( 'projectile', 'projectile.png' );              //Load projectile image
}
```

Step 2: Create the Projectile Class

Create a new file at `src/Projectile.js`. This class will serve as a blueprint for all bullets. The `constructor` is designed to be flexible: it takes a `scene`, a starting `position`, and a `velocity` as arguments. This allows us to spawn projectiles anywhere, moving in any direction. Inside the constructor, it adds itself to the scene and the physics system, then immediately sets its body's velocity.

`src/` → *Projectile.js*

```
class Projectile extends Phaser.Physics.Arcade.Sprite {
    constructor(scene, position, velocity)v{
        super(scene, position.x, position.y, 'projectile');
        this.depth = 1;

        scene.add.existing(this);
        scene.physics.add.existing(this);
        this.body.velocity.x = velocity.x;
        this.body.velocity.y = velocity.y;
    }
}
```

Step 3: Add a Projectile Manager to the Scene

Back in `PlayScene.js`, we need a central place to manage all active projectiles. Following our established pattern, add a call to a new helper method, `create_projectiles()`, from within the main `create()` method. It's important to call this before `create_player()` so that the projectile system is ready when the player is created.

`src/` → *PlayScene.js* → *create()*

```
//create game data
create() {
    this.create_map();                          //create map
    this.create_projectiles();                  //create projectiles
    this.create_player();                       //create player
    this.create_enemies();                      //create enemies
    this.create_collisions();                   //create physics-related behaviors
}
```

Step 4: Initialize the Projectile Array

Now, implement the `create_projectiles` helper. Its job is to initialize an empty array on the scene, `this.player_projectiles`. This array will serve as our master list to keep track of every active projectile the player has fired, which we will need for collision checks in the next milestone.

`src/` → *PlayScene.js* → *create_projectiles()*

```
create_projectiles(){
    this.player_projectiles = [];
}
```

Step 5: Connect the Player to the Projectile System

To allow the player to fire, we need to make two changes to its `constructor`. First, add `'space'` to the string passed to `addKeys` so we can listen for spacebar presses. Second, we'll give the player a reference to the scene's `player_projectiles` array so it knows which list to add new projectiles to.

`src/` → *Player* → *constructor()*

```
constructor(scene) {
    super(scene, 300, 200, 'player');
    this.depth = 2;
    this.speed = 200;
    this.projectiles = scene.player_projectiles;

    scene.add.existing(this);
    scene.physics.add.existing(this);
    this.setCollideWorldBounds(true);                                   //don't leave map
    this.body.setSize(this.width-16, this.height-16);                   //16x16 for 32x32
    this.buttons = scene.input.keyboard.addKeys('up,down,left,right,space');   //add space key
    this.anims.play('player-move',true);                                //play move animation
}
```

Step 6: Implement the Player's attack Method

This new `attack` method contains the core logic for firing and is a great example of **encapsulation**. It checks if the spacebar is currently held down. If it is, it creates a `new Projectile` instance at the player's current location with a hardcoded forward velocity. The newly created projectile is then added to the projectiles array we referenced in the constructor.

`src/` → *Player* → *attack()*

```
attack() {
    if( this.buttons.space.isDown ) {
        const position = {x:this.x, y:this.y};
        const velocity = {x:300, y:0};
        const projectile = new Projectile(this.scene, position, velocity);
        this.projectiles.push(projectile);
    }
}
```

Step 7: Call attack from the Scene's Update Loop

To make the player check for firing on every frame, we need to call its new `attack()` method from the game's main `update` loop. Following our principle of **Separation of Concerns**, we'll add this call inside the PlayScene's `update_player` helper method, right after the call to `this.player.move()`. The Scene "conducts" the update, while the Player handles its own logic.

`src/` → *PlayScene.js* → *player_update()*

```
player_update() {
    this.player.move();
    this.player.attack();
}
```

Step 8: Load the New Projectile.js File

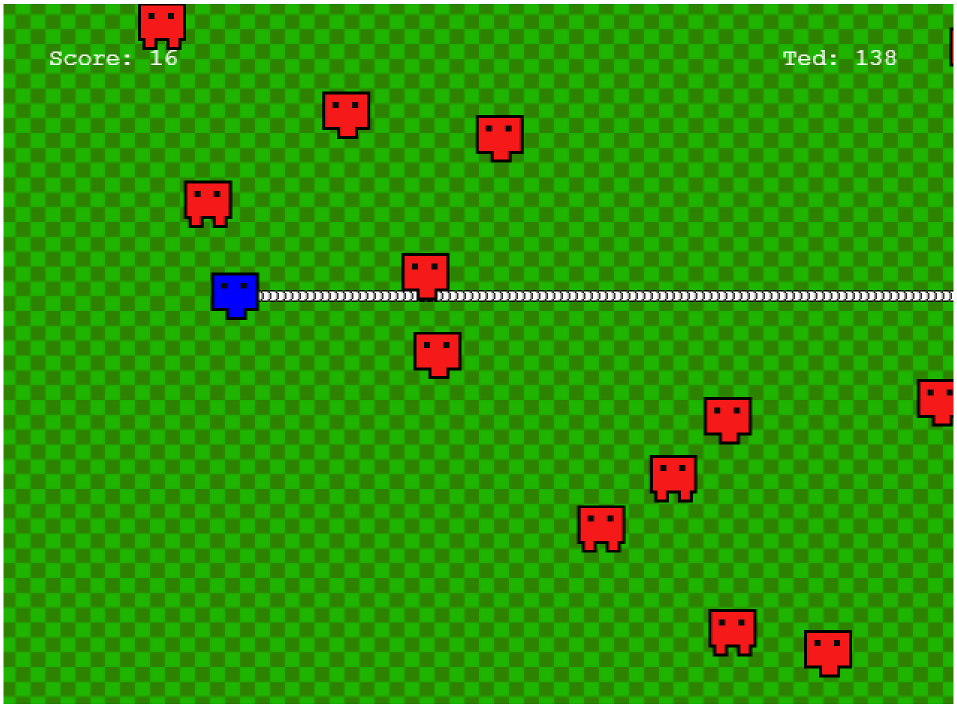
Finally, we must load our new `Projectile.js` file by adding a `<script>` tag to `index.html`. The order is critical: since the `Player` class creates instances of `Projectile`, `Projectile.js` must be included before `Player.js` to prevent a "class not defined" error.

index.html

```
<html>
    <head>
        <script src="./src/phaser.js">    </script>
        <script src="./src/Projectile.js"></script>
        <script src="./src/Player.js">    </script>
        <script src="./src/Enemy.js">     </script>
        <script src="./src/PlayScene.js"> </script>
        <script src="./src/game.js">      </script>
    </head>
    <body></body>
</html>
```

(Milestone 1) Testing:

Try running the game after these changes and verify that player triggers projectiles. However, it generates projectiles for every tick of the game loop.



Milestone 2: Collisions between Projectiles and Enemies

Milestone Goal

Now that projectiles are firing, we need to make them interact with enemies. We'll add a physics overlap check that triggers an action whenever a projectile touches an enemy, destroying both objects upon impact.

Step 1: Create the Collision Callback Function

First, we need to define the logic that will run *when* a collision occurs. We do this by creating a **collision callback** function. Phaser will automatically call this function for us, passing in the two specific game objects that collided (in this case, a `projectile` and an `enemy`). Inside this function, we simply call `.destroy()` on both objects to remove them from the scene.

`src/` → *PlayScene.js* → *slay_enemy()*

```
slay_enemy(projectile, enemy) {
    enemy.destroy();
    projectile.destroy();
}
```

Step 2: Add the Physics Overlap Check

Now that we have our callback logic, we need to tell Phaser's physics engine to use it. In the `create_collisions` method, we add another `this.physics.add.overlap()` check. This is the "glue" that connects our game objects to our logic. We tell Phaser to constantly watch for overlaps between the `this.player_projectiles` array and the `this.enemies` array, and to execute our `slay_enemy` function whenever it finds one.

`src/` → *PlayScene.js* → *create_collisions()*

```
//create physics and collisions
create_collisions(){
    this.physics.add.overlap(this.player,this.enemies,this.game_over,null,this);
    this.physics.add.overlap(this.player_projectiles,this.enemies,this.slay_enemy,null,this);
}
```

(Milestone 2) Testing:

Test Instructions:
Launch the game and fire at the enemies.

Expected Outcome:
When one of your projectiles hits an enemy, both the projectile and that specific enemy should instantly disappear from the screen.

Key Observation & What's Next:
Notice that you can fire an endless stream of projectiles, which is very overpowered and doesn't feel balanced. This is a "game feel" problem. In the next milestone, we will address this by implementing a firing rate cooldown.

Milestone 3: Set a Timed Firing Rate for Projectiles

Milestone Goal:

Currently, the player can fire a continuous stream of projectiles, which is unbalanced. To fix this, we'll implement a **cooldown** to limit the firing rate. We'll achieve this by tracking the time between shots using a high-precision timestamp that Phaser provides in its `update` loop.

Step 1: Pass the Timestamp to the update_player Method

Phaser's `update(time)` method gives us access to a constantly running clock (in milliseconds). To use this clock in our `Player`'s `attack` method, we need to pass the `time` value down through the chain of function calls: from `update()` to `update_player()`, and finally to `player.attack()`.

`src/` → *PlayScene.js* → *update()*

```
update(time) {
    this.update_player(time);
}
```

Step 2: Pass the Timestamp to the attack Method

Add parameter into update so that phaser passes us the timestamp for this call on the update method, then pass that value to the player attack method.

`src/` → *PlayScene.js* → *update_player()*

```
update_player(time) {
    this.player.move();
    this.player.attack(time);
}
```

Step 3: Initialize a Cooldown Timer

To track the cooldown, the `Player` needs a state variable to remember when it last fired. In the Player's constructor, create a new property called `this.last_fired`. We initialize it to `0` to ensure the player can fire their very first shot without any delay.

`src/` → *Player.js* → *constructor*

```
constructor(scene) {
    super(scene, 300, 200, 'player');
    this.depth = 1;
    this.speed = 200;
    this.last_fired = 0;

    scene.add.existing(this);
    scene.physics.add.existing(this);
    this.setCollideWorldBounds(true);      //don't go out of the map

   this.buttons = scene.input.keyboard.addKeys('up,down,left,right,space');
}
```

Step 4: Implement the Cooldown Logic

Now, refactor the `attack(time)` method to use our new timer. We add a second condition to the if statement: the expression `time - this.last_fired > 400` checks if at least 400 milliseconds have passed since the last shot. If both conditions (spacebar down AND cooldown passed) are `true`, we fire a projectile and then immediately update `this.last_fired = time;` to record the timestamp of the new shot, effectively resetting the timer for the next one.

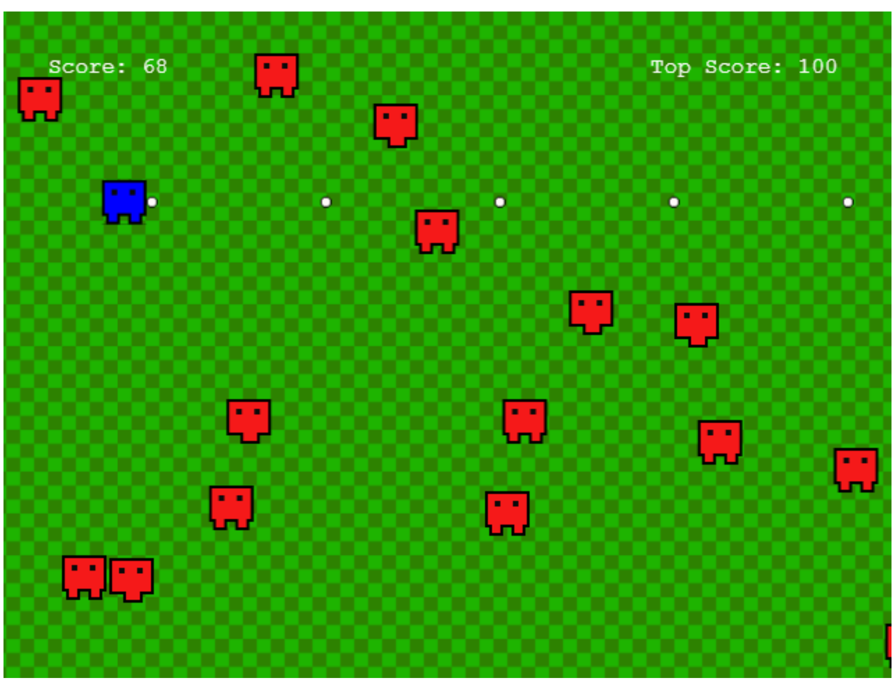
`src/` → *Player.js* → *attack()*

```
attack((time) {
    if( this.buttons.space.isDown && time - this.last_fired > 400 ) {
        const position = {x:this.x, y:this.y};
        const velocity = {x:300, y:0};
        const projectile = new Projectile(this.scene, position, velocity);
        this.projectiles.push(projectile);
        this.last_fired = time;
    }
}
```

(Milestone 3) Testing:

Test Instructions:
Launch the game and hold down the spacebar.

Expected Outcome:
Projectiles should now fire at a steady, controlled rate (roughly two shots per second) instead of a continuous stream.



Milestone 4: Rapid Fire on Spacebar Clicks

Milestone Goal:

Our current cooldown works well for holding down the spacebar, but it feels unresponsive to rapid tapping. We will now refine the logic to ensure that the first press of the spacebar after a release always fires a projectile instantly, rewarding players with quick reflexes.

Step 1: Reset the Cooldown Timer on Key Release

The solution is to add a second if statement to our `attack(time)` method. This new block checks if the spacebar `isUp`. When the player releases the key, we reset the `last_fired` timer back to `0`.

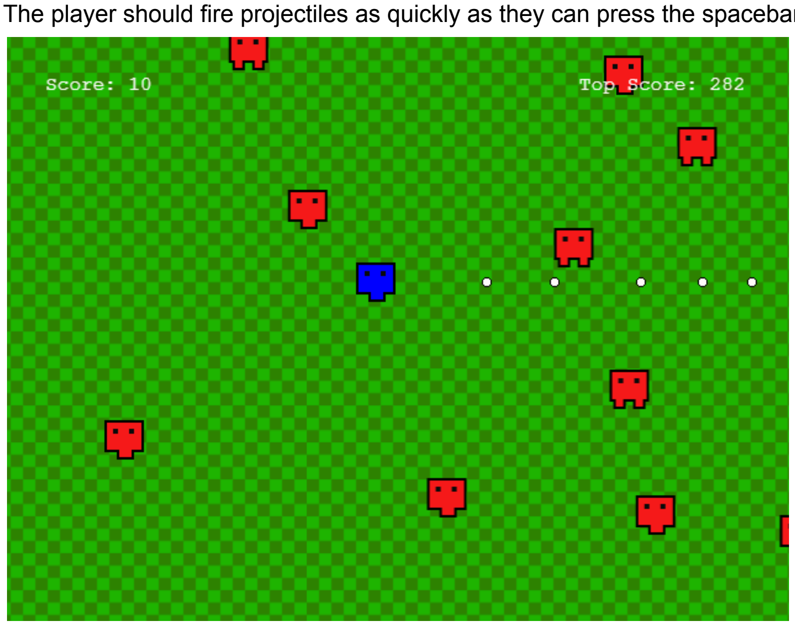
This means that the next time the player presses the spacebar, the cooldown check `(time - this.last_fired > 400)` will always be `true`, guaranteeing that the first shot of a new press is always instant. This gives us the best of both worlds: a controlled rate for holding the button down, and instant responsiveness for single taps.

`src/` → *Player.js* → *attack()*

```
attack((time) {
    if( this.buttons.space.isDown && time - this.last_fired > 400 ) {
        const position = {x:this.x, y:this.y};
        const velocity = {x:300, y:0};
        const projectile = new Projectile(this.scene, position, velocity);
        this.projectiles.push(projectile);
        this.last_fired = time;
    }
    if( this.buttons.space.isUp ) {
        this.last_fired = 0;
    }
}
```

(Milestone 4) Testing:

The player should fire projectiles as quickly as they can press the spacebar.



'Assess' → Test phase

Testing:

Test Instructions:
Launch the game and try both firing styles:

- Hold the spacebar down.
- Tap the spacebar as rapidly as you can.

Expected Outcome:
Holding the spacebar should still result in a controlled, timed firing rate. Tapping the spacebar, however, should now feel much more responsive, with each distinct press firing a projectile instantly.

Enhancement 6*: Enemy Projectiles

'Approach' → Plan phase

Dodger games become more challenging when the enemies can shoot projectiles towards the player. If the player is given the capability of projectiles then enemies should as well otherwise the challenge factor of the game is lopsided to the player since they have the advantage of distance attacks. To make the game more challenging and balanced its important that enemies also have that capability. Note if you do not give the player projectiles, then giving enemies projectiles might make the game too difficult and unbalanced. The key to good game design is finding the right balance. With this enhancement, enemies will be able to attack, but at a much slower pace than the player.

NOTE: this enhancment assumes you have included the player projectiles -- you can implement without it -- but then there are instructions in enhancement 5 that would need to be included in this implementation.

'Apply' → Do phase

Milestone 1: Fire projectiles from Enemies

Milestone Goal:

In this milestone, we will give our enemies the ability to fire projectiles. We will follow a similar pattern to how we implemented the player's attack: we'll give each enemy its own attack logic and cooldown timer, and then orchestrate the attacks from the main `PlayScene`.

Step 1: Create a Projectile List for Enemies

First, we need a way to manage all the projectiles fired by enemies. In the `PlayScene`'s `create_projectiles` method, initialize a new empty array called `this.enemy_projectiles`. This will keep all enemy projectiles separate from the player's, which is essential for setting up the correct collision rules later.

`src/` → *PlayScene.js* → *create_projectiles()*

```
create_projectiles(){
    this.player_projectiles = [];
    this.enemy_projectiles = [];
}
```

Step 2: Give Enemies an Attack Cooldown Timer

To make the enemies fire, we need to add several state variables to the `Enemy` class constructor. This is a great example of **encapsulation**, where each enemy instance will manage its own, independent firing behavior.

- `this.last_fired`: Tracks the timestamp of the last shot to manage the cooldown.
- `this.projectiles`: A reference to the scene's `enemy_projectiles` array, so the enemy knows where to add its new projectiles.
- `this.attack_duration`: A **randomized** cooldown for each enemy. This is a key game design choice that creates more varied and less predictable attack patterns from the enemy horde.

`src/` → *Enemy* → *constructor()*

```
constructor(scene, position) {
    super(scene, position.x, position.y, 'enemy');
    this.depth = 1;
    this.last_fired = 0;
    this.projectiles = scene.enemy_projectiles;

    scene.add.existing(this);
    scene.physics.add.existing(this);
    this.body.velocity.x = -Phaser.Math.Between(120, 300);
    this.attack_duration = Phaser.Math.Between(2000, 4000);
}
```

Step 3: Implement the Enemy's attack Method

Now, create an `attack` method inside the `Enemy` class. This method contains the core firing logic. The first line is a "guard clause" to ensure that inactive or destroyed enemies don't try to fire. The main `if` statement checks if the randomized `attack_duration` has passed. If it has, the enemy creates a new `Projectile`, adds it to the list, and then resets its own timers, including rolling a *new* random duration for its next shot.

`src/` → *Enemy* → *attack()*

```
attack(time) {
    if (!this.active || !this.body || !this.scene) return;

    if( time - this.last_fired > this.attack_duration ) {
        const position = { x:this.x, y:this.y };
        const velocity = { x:this.body.velocity.x-100, y:0  };
        const projectile = new Projectile(this.scene, position, velocity);
        this.projectiles.push(projectile);
        this.last_fired = time;
        this.attack_duration = Phaser.Math.Between(2000, 4000);
    }
}
```

Step 4: Create a Scene-Level Enemy Update Loop

The `PlayScene`, acting as the "conductor," needs a way to tell every enemy to check if it should attack on every frame. We'll create a new helper method, `update_enemies`, for this purpose. This function simply iterates through our `this.enemies` array and calls the `attack` method on each active enemy, passing along the `time` value needed for the cooldown logic.

`src/` → *PlayScene* → *update_enemies()*

```
update_enemies(time){
    this.enemies.forEach(enemy => enemy.attack(time));
}
```

Step 5: Call the Enemy Update Loop from the Scene

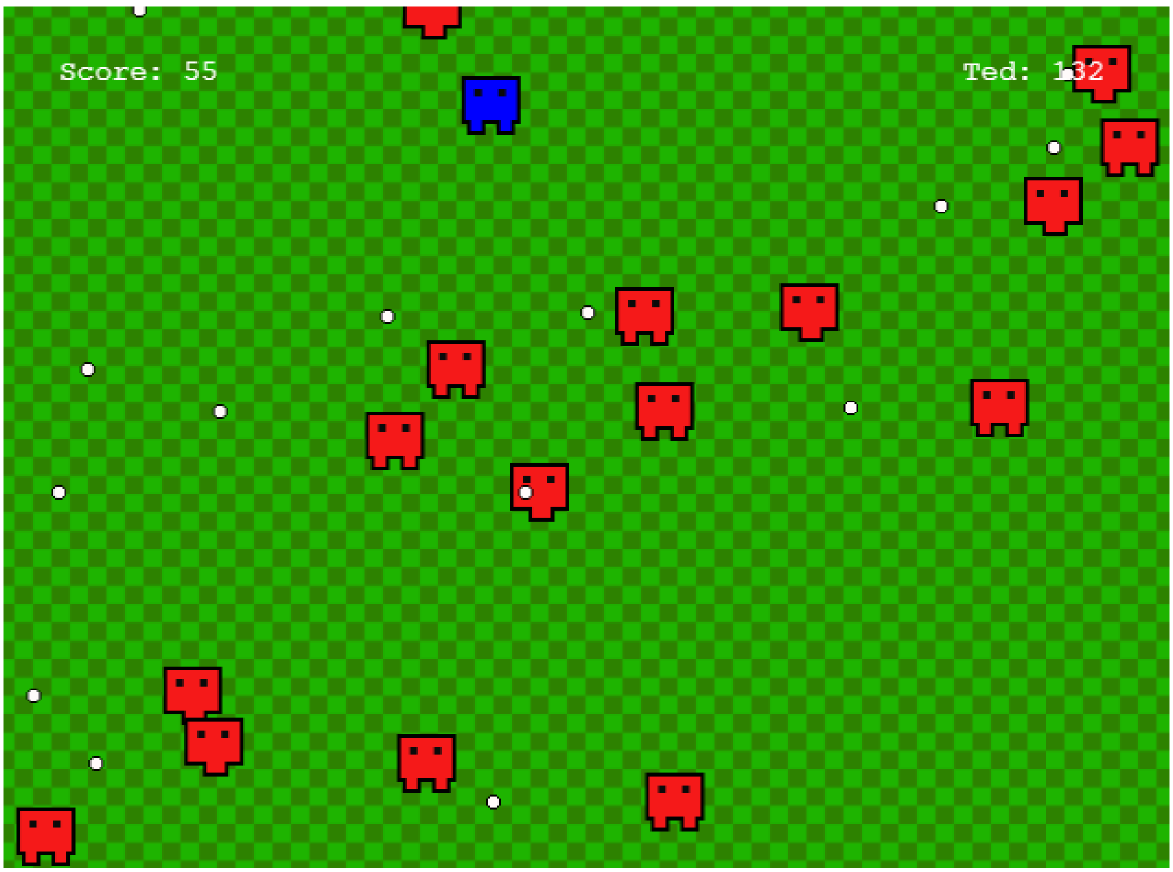
Finally, we wire everything up. In the `PlayScene`'s main `update` loop, add a call to your new `update_enemies(time)` helper method. This completes the logic chain, ensuring every enemy's attack logic is checked on every single frame of the game.

`src/` → *PlayScene* → *update()*

```
//Update game data
update(time) {
        this.update_player(time);
        this.update_enemies(time);
}
```

(Milestone 1) Testing:

Try running the game after these changes and verify that enemies emit projectiles. However, projectiles do not have collision behaviors yet.



Milestone 2: Collisions between Projectiles and Player

Milestone Goal:

Add a collider in the scene's physics system that triggers an action whenever a projectile touches the player. We can invoke the `game_over` method same as enemy overlaps

Step 1: Make Enemy Projectiles End the Game

Add overlap detection in physics system between `projectiles` & `player`, if an overlap occurs, invoke the the `game_over` method.

`src/` → *PlayScene.js* → *create_collisions()*

```
//create physics and collisions
create_collisions(){
    this.physics.add.overlap(this.player,this.enemies,this.game_over,null,this);
    this.physics.add.overlap(this.player_projectiles,this.enemies,this.slay_enemy,null,this);
    this.physics.add.overlap(this.enemy_projectiles,this.player,this.game_over,null,this);
}
```

(Milestone 2) Testing:

Try running the game after these changes and verify that enemies projectiles trigger a game over

'Assess' → Test phase

Testing:

Verify that the culmination of all milestones translates into a full realization for this enhancement feature goal.

Enhancement 7*: Power-ups

'Approach' → Plan phase

In game design, a sense of **player progression** ("leveling up") keeps play engaging by rewarding the player with new powers or efficiency over time. In dodger-style games, that progression is typically delivered through **power-ups** - temporary or stackable boosts the player collects mid-run.

We'll add two collectible power-ups:

- **Slay** - wipes all enemies (and their bullets) on screen.
- **Projectile** - increases the player's projectile size up to a cap

We will implement this using an Object-Oriented approach. We'll create a generic base `PowerUp` class that handles shared logic (like movement). Then, we will create specific subclasses (`SlayPowerUp`, `ProjectilePowerUp`) that **inherit** from the base class and provide their own unique `applyEffect` logic. This is a clean, scalable design that mirrors professional game development.

'Apply' → Do phase

Milestone 1: Create and Spawn a Generic Power-up

Milestone Goal:

First, we'll build the foundational `PowerUp` base class and get a generic, non-functional version of it spawning on a timer. This ensures our core spawning system is working before we add specific logic.

Step 1: Preload the Power-up Assets

In the PlayScene's `preload()` method, add the lines to load the images for both of our power-up types.

`src/` → *PlayScene.js* → *preload()*

```
preload() {
    this.load.path = 'assets/';                                         //Define file path
    this.load.image( 'background', 'background.png' );                  //Load background image
    this.load.image( 'player', 'player.png' );                          //Load player image
    this.load.image( 'enemy', 'enemy.png' );                            //Load player image
    this.load.image( 'powerup-projectile', 'powerup-1.png' );           //Load projectile image
    this.load.image( 'powerup-slay', 'powerup-2.png' );                 //Load projectile image
}
```

Step 2: Create the Base PowerUp Class

Create a new file at `src/PowerUp.js`. This will be our "abstract" base class. It handles all the logic that is common to *all* power-ups: being a physics sprite and moving to the left. Most importantly, it defines an `applyEffect` method. This creates a "contract," establishing that every power-up subclass we create in the future is expected to have this method.

`src/` → *PowerUp.js*

```
class PowerUp extends Phaser.Physics.Arcade.Sprite {
    constructor(scene, x, y, texture) {
        super(scene, x, y, texture);
        this.depth = 1;
        scene.add.existing(this);
        scene.physics.add.existing(this);
        this.body.velocity.x = -300;
    }

    // This is the key: a placeholder method for subclasses to override.
    applyEffect(player) {
        console.warn('applyEffect not implemented for this power-up type.');
    }
}
```

Step 3: Add the Power-up Manager to the Scene

Now, let's wire up our spawning system. In `PlayScene.js`, add a call to a new helper method, `create_powerups()`, from within the main `create()` method. This continues our pattern of keeping the `create()` method organized.

`src/` → *PlayScene.js* → *create()*

```
create(){
        this.create_map();
        this.create_player()
        this.create_enemies();
        this.create_powerups();
        this.create_collisions();
}
```

Step 4: Create a Timed Spawner

Implement the `create_powerups` helper. Its job is to set up a system that will spawn power-ups periodically. We'll create two new arrays to track each type of power-up. Then, we use Phaser's Time Manager (`this.time`) to create a looping timer that will call a `spawn_powerup` function every 3 seconds.

`src/` → *PlayScene.js* → *create_powerups()*

```
create_powerups() {
    this.powerups = [];

    const event = new Object();
    event.delay = 3000;
    event.callback = this.spawn_powerup;
    event.callbackScope = this;
    event.loop = true;

    this.time.addEvent(event, this);
}
```

Step 5: Implement a Placeholder Spawner

For this milestone, our `spawn_powerup` function will be very simple. It will have a 1-in-5 chance of spawning a generic `PowerUp` instance. We'll make it more dynamic in the next milestones.

`src/` → *PlayScene.js* → *spawn_powerups()*

```
// The powerup spawner
spawn_powerup() {
    if (Phaser.Math.Between(0, 4) !== 0) return;

    // 1. Pick a PowerUp CLASS
    const PowerUpClass = PowerUp;

    // 2. Define the spawn position
    const position = {
        x: 640 + 32,
        y: Phaser.Math.Between(50, 430)
    };

    // 3. Instantiate the chosen class and add it to a SINGLE group/array
    const powerup = new PowerUpClass(this, position.x, position.y, 'powerup-slay');
    this.powerups.push(powerup);
}
```

Step 6: Load the New PowerUp.js File

Add the new `PowerUp.js` file to your `index.html`, ensuring it is loaded before `PlayScene.js`.

index.html

```
<html>
    <head>
        <script src="./src/phaser.js">    </script>
        <script src="./src/Projectile.js"></script>
        <script src="./src/Player.js">    </script>
        <script src="./src/Enemy.js">     </script>
        <script src="./src/PowerUp.js"></script>
        <script src="./src/PlayScene.js"> </script>
        <script src="./src/game.js">      </script>
    </head>
    <body></body>
</html>
```

(Milestone 1) Testing:

Launch the game. Every few seconds, a power-up icon should spawn and move across the screen. Collecting it will do nothing yet.

Milestone 2: Implement the SlayPowerUp Subclass

Milestone Goal:

Now we will create our first *concrete* implementation. We'll build the `SlayPowerUp` subclass, make our spawner use it, and wire up its collision effect.

Step 1: Create the SlayPowerUp Subclass

Open `src/PowerUp.js`. At the bottom of the file, **underneath the base `PowerUp` class**, define the `SlayPowerUp` subclass. It `extends PowerUp` and **overrides** the `applyEffect` method with its own unique logic: destroying all enemies and flashing the camera.

`src/` → *PowerUp.js* → *class SlayPowerUp*

```
class PowerUp extends Phaser.Physics.Arcade.Sprite {
   ...
}


// SlayPowerUp subclass underneath superclass
class SlayPowerUp extends PowerUp {
    constructor(scene, x, y) {
        super(scene, x, y, 'powerup-slay');
    }

    applyEffect(player) {
        // We need the scene to access the enemies list
        const scene = this.scene;
        scene.enemies.forEach(monster => monster.destroy());
        scene.enemy_projectiles.forEach(bullet => bullet.destroy());
        scene.cameras.main.flash();
    }
}
```

Step 2: Update the Spawner to Use the New Class

In `PlayScene.js`, modify the `spawn_powerup` function to instantiate our new, specific `SlayPowerUp` class instead of the generic base class.

`src/` → *PlayScene.js* → *spawn_powerups()*

```
// The powerup spawner
spawn_powerup() {
    if (Phaser.Math.Between(0, 4) !== 0) return;

    // 1. Pick a random PowerUp CLASS from our array
    const PowerUpClass = SlayPowerUp;

    // 2. Define the spawn position
    const position = {
        x: 640 + 32,
        y: Phaser.Math.Between(50, 430)
    };

    // 3. Instantiate the chosen class and add it to a SINGLE group/array
    const powerup = new PowerUpClass(this, position.x, position.y);
    this.powerups.push(powerup);
}
```

Step 3: Create the Polymorphic Collision Callback

In `PlayScene.js`, create the single, simple callback that will handle *all* power-up collisions. This function, `collectPowerUp`, doesn't need to know what *kind* of power-up it is. It just tells the power-up to apply its effect and then destroys it. This is **polymorphism** in action.

PlayScene.js → create_collisions()

```
// The beautifully simple, polymorphic callback
collect_powerup(player, powerup) {
    // Tell the power-up to do its thing. The scene doesn't care what it is.
    powerup.applyEffect(player);
    powerup.destroy();
}
```

Step 4: Add the Physics Overlap

In `create_collisions()`, add the single overlap check between the `player` and the entire `powerups` group, linking it to your new `collectPowerUp` callback.

PlayScene.js → create_collisions()

```
create_collisions(){
   this.physics.add.overlap(this.player, this.enemies, this.game_over, null, this);
   this.physics.add.overlap(this.player, this.powerups, this.collectPowerUp, null, this);
}
```

(Milestone 2) Testing:

Launch the game. Wait for a 'Slay' power-up to appear. Verify that collecting it immediately clears the screen of all enemies. Do not proceed until this works.

Milestone 3: Implement ProjectilePowerUp & Randomize Spawning

Milestone Goal:

With the pattern established, we'll now add our second power-up, `ProjectilePowerUp`, and update our spawner to randomly choose which power-up to create.

Step 1: Update the Player to Track Projectile Size

This power-up requires the Player to track its projectile size. First, add a projectileScale property to the Player's constructor, initializing it to 1.

Player.js → constructor()

```
constructor(scene) {
    super(scene, 300, 200, 'player');
    this.depth = 2;
    this.speed = 200;
    this.last_fired = 0;
    this.projectiles = [];
    this.projectileScale = 1;

    scene.add.existing(this);
    scene.physics.add.existing(this);
    this.setCollideWorldBounds(true);      //don't go out of the map
    this.body.setSize(this.width-16, this.height-16);

    this.buttons = scene.input.keyboard.addKeys('up,down,left,right,space');
    this.setup_animations(scene);
}
```

Step 2: Sync Visual Scale with Physics Hitbox

Next, refactor the `attack` method. After creating a projectile, use `setScale()` to modify its visual size based on `this.projectileScale`. Critically, you must also update the physics body with `body.setSize()` to match the new visual scale.

Player.js → attack()

```
attack(time)  {
    if( this.buttons.space.isDown && time - this.last_fired > 400 ) {
        const position = { x:this.x, y:this.y };
        const velocity = { x:300 ,y:0 };
        const projectile = new Projectile(this.scene, position, velocity);
        // scale the sprite you see    #POWERUP
        projectile.setScale(this.projectileScale);
        // make the physics hitbox match what you see (minimal + effective) #POWERUP
        projectile.body.setSize(projectile.displayWidth, projectile.displayHeight, true);
        this.projectiles.push(projectile);
        this.last_fired = time;
    }
    if( this.buttons.space.isUp ) {
        this.last_fired = 0;
    }
}
```

Step 3: Create the ProjectilePowerUp Subclass

Create a new subclass `ProjectilePowerUp` in the file, `src/PowerUp.js`. Just like the `SlayPowerUp`, it extends `PowerUp` and overrides the `applyEffect` method, this time with the logic to increase the player's `projectileScale`.

`src/` → *PowerUp.js* → *class ProjectilePowerUp*

```
class PowerUp extends Phaser.Physics.Arcade.Sprite {
   ...
}

class SlayPowerUp extends PowerUp {
   ...
}

class ProjectilePowerUp extends PowerUp {
    constructor(scene, x, y) {
        // Pass the specific texture key to the parent
        super(scene, x, y, 'powerup-projectile');
    }

    // Override the base method with specific logic
    applyEffect(player) {
        player.projectileScale = Math.min(player.projectileScale + 1, 3);
    }
}
```

Step 4: Update the Spawner to be a "Factory"

Now, make the final change to `spawn_powerup`. It will now randomly pick a **Class** from the `powerupTypes` array and instantiate it. This is known as the **Factory Pattern**. Your spawner is now complete and scalable.

`src/` → *PlayScene.js* → *spawn_powerups()*

```
// The powerup spawner
spawn_powerup() {
    this.powerup_types = [ProjectilePowerUp, SlayPowerUp]
    if (Phaser.Math.Between(0, 4) !== 0) return;

    // 1. Pick a random PowerUp CLASS from our array
    const PowerUpClass = Phaser.Utils.Array.GetRandom(this.powerup_types);

    // 2. Define the spawn position
    const position = {
        x: 640 + 32,
        y: Phaser.Math.Between(50, 430)
    };

    // 3. Instantiate the chosen class and add it to a SINGLE group/array
    const powerup = new PowerUpClass(this, position.x, position.y);
    this.powerups.push(powerup);
}
```

(Milestone 3) Testing:

Launch the game. Both 'Slay' and 'Projectile' power-ups should now spawn randomly. Verify the 'Slay' power-up still works, and that collecting the 'Projectile' power-up makes your bullets larger.

'Assess' → Test phase

Testing:

This is the final integration test where we confirm that all previous milestones work together to deliver the complete power-up feature. Launch the game and perform a full playtest to verify the following behaviors:

Spawning:

- [ ] Do both 'Slay' and 'Projectile' power-ups appear periodically from the right side of the screen?

'Slay' Power-up Functionality:

- [ ] When you collect the 'Slay' power-up, are all enemies currently on screen instantly destroyed?
- [ ] Are all enemy *projectiles* also destroyed?
- [ ] Does the screen flash briefly upon collection?
- [ ] Does the power-up icon disappear after being collected?

'Projectile' Power-up Functionality:

- [ ] When you collect the 'Projectile' power-up, do your fired bullets become visibly larger?
- [ ] **Test the Cap:** Collect the 'Projectile' power-up multiple times. Confirm that the bullet size increases with the second and third collection, but **stops** increasing after the third.
- [ ] Does the power-up icon disappear after being collected?

If you can check all of these boxes, you have successfully implemented the entire power-up enhancement.

Enhancement 8: Title Screen & Scene Management

'Approach' → Plan phase

A polished game needs a proper introduction. Our final enhancement will be to add a **Title Screen**. This screen will serve as the game's main menu, displaying the title and instructions, and providing a clear starting point for the player.

This change transforms our project into a multi-scene application. To manage this, we will use two key Phaser systems:

1. **The Scene Manager (** A powerful tool that allows us to start, stop, and switch between different scenes.
2. **The Registry (** A global data manager. Since switching scenes destroys the old scene's data, we will use the Registry to store our `top_score` and `winner` name so this data can **persist** between playthroughs.

'Apply' → Do phase

Milestone 1: Create the Basic Title Scene

Milestone Goal:

First, we'll create a new `TitleScene` class, display the game title and instructions, and make it the new starting point for our game.

Step 1: Create the class TitleScene

Create a new file at `src/TitleScene.js`. This will be a simple scene whose only job is to display text. Its `constructor` registers itself with the key `'title'`. The `create()` method invokes a helper method

`src/` → *TitleScene.js*

```
// src/TitleScene.js
class TitleScene extends Phaser.Scene {
    constructor() {
        super('title'); // Register scene with key 'title'
    }

    create() {
        // We will add content here in the next step
       this.create_title()
    }
}
```

Step 2: Add Title and Instruction Text

In the `TitleScene`'s `create()` method, we'll use `this.add.text()` to display our game's UI. A useful trick for centering text is to use the game's width/height and set the text's **origin** to `(0.5, 0.5)`. The origin is the text's "anchor point" on a 0-to-1 scale.

src/ → *TitleScene.js → create_title()*

```
// src/TitleScene.js → create()
create_title() {
    const width = this.game.config.width;
    const height = this.game.config.height;

    // Game Title
    this.add.text(width / 2, height / 3, 'DODGER GAME', {
        fontSize: '48px',
        fill: '#FFFFFF'
    }).setOrigin(0.5);

    // Instructions
    this.add.text(width / 2, height / 2, 'Arrow Keys to Move\nSpacebar to Fire', {
        fontSize: '24px',
        fill: '#FFFFFF',
        align: 'center' // Center-align multi-line text
    }).setOrigin(0.5);

    // Start prompt
    this.add.text(width / 2, height * 2 / 3, 'Press SPACE to Start', {
        fontSize: '24px',
        fill: '#FFFF00'
    }).setOrigin(0.5);
}
```

Step 3: Make the Title Screen the Starting Scene

Now we need to tell Phaser to launch our new `TitleScene` first. Open `src/game.js` and modify the `scene` array in the game configuration. The first scene in this array is the one that loads on startup.

src/ → *game.js*

```
const config = new Object();

config.width   = 640;                                            //Width of viewport
config.height  = 480;                                            //Height of viewport
config.scene   = [ TitleScene, PlayScene ];                      //Scenes for this game
config.physics = { default:'arcade' };                           //Physics for collisions

const game = new Phaser.Game(config);
```

Step 4: Make the Title Screen the Starting Scene

Finally, add the `TitleScene.js` script to your `index.html`, ensuring it is loaded before `game.js`.

index.html

```html
<html>
    <head>
        <title>Phaser Game - Dodger</title>
        <script src="./src/phaser.js">    </script>
        <script src="./src/Projectile.js">    </script>
        <script src="./src/Player.js">    </script>
        <script src="./src/Enemy.js">     </script>
        <script src="./src/PowerUp.js"></script>
        <script src="./src/PlayScene.js"> </script>
        <script src="./src/TitleScene.js"> </script>
        <script src="./src/game.js">      </script>
    </head>
    <body></body>
</html>
```

(Milestone 1) Testing:

Launch the game. It should now open to your new Title Screen with the title and instructions, instead of jumping directly into the action.

Milestone 2: Implement Scene Switching

Milestone Goal:

Now we'll wire up the scene transitions: pressing `Space` on the title screen will start the game, and pressing `Escape` during gameplay will return to the title screen.

Step 1: Start the Game from the Title Screen

In `TitleScene.js`, we need to listen for a key press. We can use an event listener for this. In the `create()` method, add a listener that waits for the `SPACE` key to be pressed. The callback function will call `this.scene.start('play')`, which shuts down the current scene (`title`) and launches the one with the key `'play'`.

`src/` → *TitleScene.js* → *create()*

```
create() {
    this.create_title();
    this.input.keyboard.on('keydown-SPACE', () => { this.scene.start('play'); });

}
```

Step 2: Return to the Title Screen from the Play Scene

In `PlayScene.js`, we need to add a similar listener. In the `create()` method, add a listener for the `ESC` key. The callback will start the `'title'` scene.

`src/` → *PlayScene.js* → *create()*

```
// src/PlayScene.js → in create()
create() {
    // ... (this can go anywhere in the create method)
    this.input.keyboard.on('keydown-ESC', () => {  this.scene.start('title'); });
}
```

(Milestone 2) Testing:

Run the full flow. Start the game, you see the title. Press `Space`, the game starts. Play for a bit, then press `Escape`. You should return to the title screen. **Notice a problem:** if you got a high score, it resets! We'll fix this next.

Milestone 3: Sharing State Between Scenes with the Registry

Milestone Goal:

Our `top_score` and `winner` name currently live inside the `PlayScene`, but our `TitleScene` can't access them. To solve this, we will use Phaser's **Registry** as a global, shared data store. This will allow our scenes to communicate and ensures our high score persists between playthroughs.

Step 1: Establish the Registry as the "Single Source of Truth"

The most important step in managing global state is to initialize it in one, and only one, place. We will make the `TitleScene`, our game's entry point, responsible for setting up the default values in the Registry. This ensures that the global state is ready before any other scene needs it.

src/ → *TitleScene.js → create_game_data()*

```
// src/ → TitleScene.js → add this helper method
create_game_data() {
    // Set default values in the registry only if they don't already exist
    this.registry.set('top_score', this.registry.get('top_score') || 100);
    this.registry.set('winner', this.registry.get('winner') || 'Top Score');
}
```

Step 2: Display the Global State in the TitleScene

Now, we'll make the `TitleScene` read from the Registry to display the current high score. This demonstrates the "read" part of our inter-scene communication.

src/ → *TitleScene.js → create_topscore()*

```
create_topscore(){
    // Get the top score and winner from the registry
    const topScore = this.registry.get('top_score');
    const winner = this.registry.get('winner');

    // Display the top score
    const x = this.game.config.width / 2;
    const y = this.game.config.height - 50;
    this.add.text(x,y, `Leader: ${winner} - ${topScore}`, {
            fontSize: '20px',
            fill: '#FFFFFF'
    }).setOrigin(0.5);
}
```

Step 3: Invoke create_topscore in the TitleScene create method

In `TitleScene.js`, invoke the helper method to create the game data and display the top score from the registry.

`src/` → *TitleScene.js* → *create()*

```
// src/ → TitleScene.js → create()
create(){
    this.create_title();
    this.create_game_data();
    this.create_topscore();
    this.input.keyboard.on('keydown-SPACE', () => { this.scene.start('play'); });
}
```

Step 4: Refactor PlayScene's HUD Creation to Read from Registry

Next, we'll begin modifying the `PlayScene`. The first change is to update the `create_hud` method to read its initial values directly from the Registry instead of using any local properties.

`src/` → *PlayScene.js* → *create_hud()*

```
create_hud() {
    this.score = 0;
    this.score_text = this.add.text(32, 32, "");
    this.score_text.depth = 3;
    this.score_text.setColor( 'rgb(255,255,255)' );

    // Initialize persistent state by reading from the registry
    const {winner, top_score} = this.registry.values;
    this.top_score_text = this.add.text(600, 32, `${winner}: ${top_score}`);
    this.top_score_text.depth = 3;
    this.top_score_text.setOrigin(1,0);
}
```

Step 5: Refactor PlayScene's Score Update to Read from Registry

To keep the display in sync, we must also refactor the `update_score` helper. It will now read the latest `winner` and `top_score` from the Registry on every single frame.

`src/` → *PlayScene.js* → *update_score()*

```
update_score() {
    this.score_text.setText(`Score: ${this.score}`);
    const {winner, top_score} = this.registry.values;
    this.top_score_text.setText(`${winner}: ${top_score}`);
}
```

Step 6: Refactor game_over to Write to the Registry

Next, we'll update the `game_over` method. It will now read the current `top_score` from the Registry to check if a new high score was achieved. If it was, it will write the new score and winner name directly back into the Registry.

src/ → *PlayScene.js → game_over()*

```
game_over() {
    const {top_score, winner} = this.registry.values;
    if ( this.score >= top_score) {
        this.registry.set('top_score', this.score);
        this.physics.pause();                                    // freeze gameplay
        const winner = prompt(`New High Score! Enter your name:`);
        this.registry.set('winner', winner || 'Top Score');
        this.input.keyboard.keys = []                            // reset phaser keys stream
    }
    this.cameras.main.flash();
    this.scene.restart();
}
```

Step 7: The Final Cleanup - Remove Redundant State

The final step in any good refactor is to remove old, unused code. Now that our `PlayScene` relies entirely on the Registry for persistent data, its own `top_score` and `winner` properties in the `constructor` are no longer needed. Removing them makes our code cleaner and ensures the Registry is truly the single source of truth.

src/ → *PlayScene.js → construtor()*

```
// src/ → PlayScene.js → constructor()
constructor() {
    super('play');
    // REMOVE these lines from the constructor:
    // this.top_score = 100;
    // this.winner = 'Top Score';
}
```

(Milestone 3) Testing:

This is the final test of our complete game loop.

1. Launch the game. The title screen should show a top score of 0.
2. Play and get a new high score (e.g., 50). Enter your name. The game will restart.
3. Press `Escape` to return to the title screen. The title screen should now correctly display your name and the top score of 50.
4. Press `Space` to start a new game. The `PlayScene`'s HUD should also correctly show the top score.

This proves that our Registry is working as a shared data store between our two scenes.

Conclusions

Final Comments

This lab was a multifaceted journey through modern web and game development. You have successfully:

1. **Applied Object-Oriented JavaScript** to extend a popular, professional framework.
2. **Mastered the core concepts of the Phaser 3 library**, a transferable skill for learning any new library, framework or engine.
3. **Utilized software engineering techniques** (sprints, milestones) to iteratively build a complex project from a simple MVP to a feature-rich final product.
4. **Delivered a complete and polished Dodger game** with challenging gameplay, multiple player interactions, and a clear sense of progression.

Future Improvements *(Beyond the Scope of this Lab)*

- Use the browser's LocalStorage to save the highscore between sessions
- Add more enemy types with different behaviors and patterns
- Add multiple levels
- Add more powerups, gear, equipment
- Add collectibles
- Add boss battles.
- Triggered Animations that start and stop.
- Parallax scrolling.
- Optimize performance by remove enemies/projectiles when no longer active

Lab Submission

Push all of your lab files into your forked repository and close this Lab in your github issues. Mark it as complete.

Module 2 - Project:

Create your own front-end browser app with JavaScript & Canvas API / Phaser or modifies the DOM.

Project Bonus:

Showcase bonus. You can receive up to 20 bonus points if your project is outstanding and novel.
I'll publish all showcase projects as a demo for future students. You should reference such projects on your resume or CV.