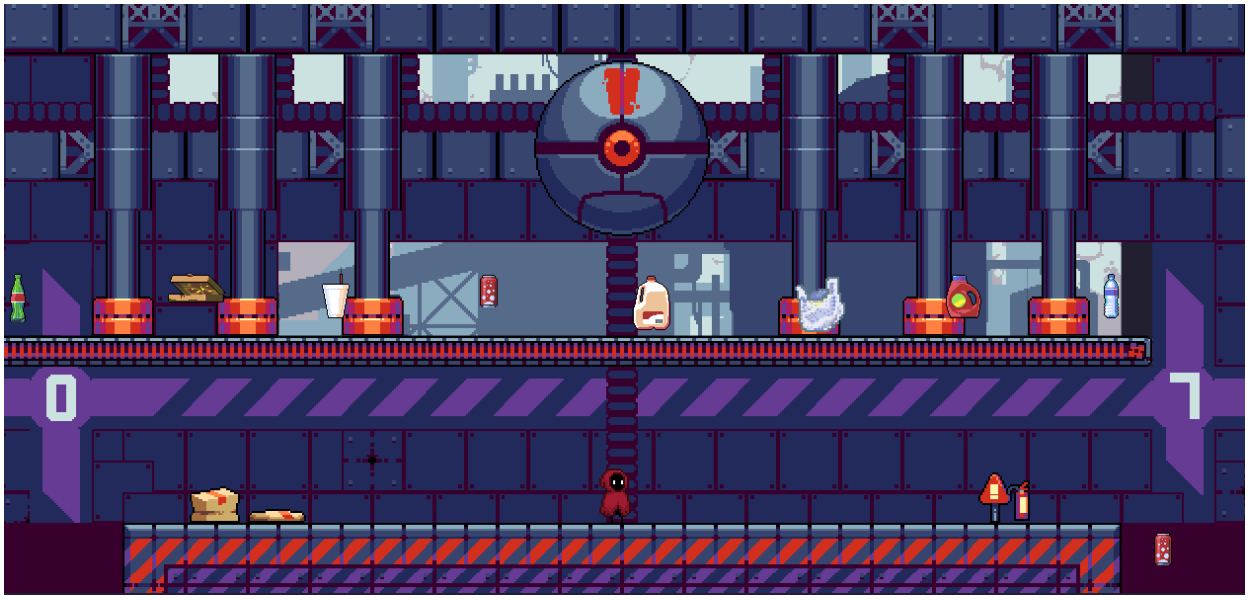
RECYCLE ADVENTRUE

Everything about the Game and Code



MARCH 8, 2024

GAEUN YOO

Recycle Adventure Tutorial

Contents

Game Overview

Working With Tiled

Applying Sprite Animations

Player Explained

Enemies Explained

Traps Explained

Boss Explained

Hud, Widgets, and Screens

Sound Effects & Background Music

Feedback & Issues to Fix

Game Overview

Introduction

The “Recycle Adventure” game is an educational game that teaches people to recycle. Recycling is essential nowadays since global warming and climate change issues are getting serious. Recycling also saves energy and reduces the extraction of raw materials, waste, and pollution. By reducing the wasted products, the amount of greenhouse gas emitted decreases and helps to tackle climate change. However, although the importance of recycling has been mentioned, recycling is not done correctly since most people don’t know much about what can be recycled. Hence, this game is aimed to teach the types of items that can be recycled in our daily lives.

Game Structure

The game has eight floors (game levels) and the final boss stage. The player should collect some recycled items to clear the stage and go to the next floor. There are 13 recycled items in total, and these items include plastic bottles, plastic bags, plastic laundry soap bottles, plastic milk jugs, newspapers, paper cardboard boxes, pizza boxes, metal tuna cans, metal red soda cans, glass green soda bottles, glass jar, mug cup, and polystyrene foam cup – everything we use in daily life. By consistently allowing players to collect recycled items, they will learn what to recycle and thus will practice recycling in their lives.

Game Story

The tiny, red-hooded protagonist is the main character of this game. The game begins in a dirty, polluted sewer where massive waste is dumped. Our protagonist picks up dumped, unproperly disposed of recycled items and navigates their way on a journey. In the middle of the journey, the protagonist meets mutated creatures due to human waste products. The slime, fish man (nicknamed cucumber), and contaminated whale all came from the ocean and lived under the sewer to avoid the waste in the sea. The protagonist eventually finds the exit outside and takes their first step into the city. The outside is worse than the sewer. The sky is dusty and contaminated due to the carbon gas emanating from cars. The protagonist continues their journey and arrives at the factory where all waste products are produced. The protagonist enters the factory and encounters the boss, the Machine Operator, to defeat it and stop further contamination.

Recycle Items List


green soda bottle (glass)

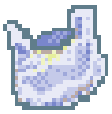

mason jar (glass)


red soda can (metal)


tuna can (metal)


cardboard box (paper)

newspaper (paper)


plastic bag (plastic)


plastic water bottle (plastic)


laundry soap bottle (plastic)


milk jug bottle (plastic)


pizza box (other)


polystyrene foam cup (other)


mug cup (other)

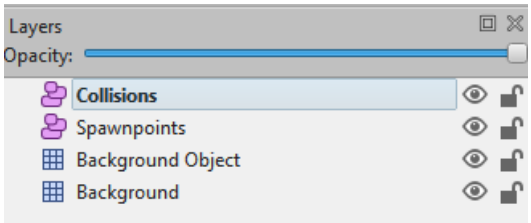
Working With Tiled

Tiled is a flexible level editor useful for creating a 2D pixel game. It allows you to easily design game maps, locate game objects, and edit platform collisions..

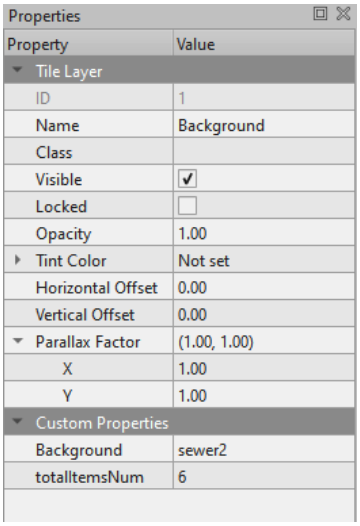
Layers in Tiled

In this game, three essential layers are required: Spawnpoints, Collisions, and Background.

The 'Spawnpoints' layer is where you set the player's starting point and locate game objects, items, and enemies. The 'Collisions' layer is where you add collisions to platforms. Lastly, the 'Background' layer is where you design a game map and add properties to a floor, such as the number of total items to collect and the background image.



*You can create other background layers to design game map.



*The properties of Background layer

Adding Collisions with Blocks & Loading Floors

Everything to connect between game and Tiled is done in floor.dart file; floor.dart works as a middleware.

Step 1. Get access to a floor's tile.

Once you create a floor (level) in Tiled, you will save it in tmx format. Then, load the Tile file into the code.

*Floor class extends `World`.

*`floorName` is the Tile file's name.

```
class Floor extends World with HasGameRef<RecycleAdventure> {
  final String floorName;
  final Player player;
  Floor({
    required this.floorName,
    required this.player,
  });
  late TiledComponent floor;
  List<CollisionBlock> collisionBlocks = [];
  late String backgroundName;
  late int totalItemsNum;

  @override
  FutureOr<void> onLoad() async {
    floor = await TiledComponent.load('$floorName.tmx', Vector2.all(16));
    add(floor);
```

*Use '`floor`' as a reference to the Tile file.

*`add(floor)` loads the floor on the game scene.

Step 2. Create CollisionBlock class and initialize a list of CollsionBlock in floor.dart.

```
class CollisionBlock extends PositionComponent {
  bool isPlatform;
  CollisionBlock({
    super.position,
    super.size,
    this.isPlatform = false, //Set it is not platform as default
  }) {
    debugMode = true;
  }
}
```

*This class extends `PositionCompnent`.

In floor.dart file:

```
List<CollisionBlock> collisionBlocks = [];
```

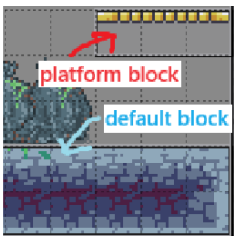
Step 3. Get Collision Blocks from Tiled.

```
void _addCollisions() {
  final collisionsLayer = floor.tileMap.getLayer<ObjectGroup>('Collisions');

  if (collisionsLayer != null) {
    for (final collision in collisionsLayer.objects) {
      switch (collision.class_) {
        case 'Platform':
          final platform = CollisionBlock(
              position: Vector2(collision.x, collision.y),
              size: Vector2(collision.width, collision.height),
              isPlatform: true); // CollisionBlock
          collisionBlocks.add(platform);
          add(platform);
          break;
        default:
          final block = CollisionBlock(
            position: Vector2(collision.x, collision.y),
            size: Vector2(collision.width, collision.height),
          ); // CollisionBlock
          collisionBlocks.add(block);
          add(block);
      }
    }
  }
  player.collisionBlocks = collisionBlocks;
}
```

Get a reference to Collisions layer in Tiled and use it to access the collision blocks.

Looping through them, add each block to the '`collisionBlocks`' list created above.

*Platform block vs default collision block:



Property	Value
Object	
ID	11
Template	
Name	
Class	Platform
Visible	✓

*Must specify class name to create platform block.

*`player.collisionBlocks` is for the player to interact with collision blocks.

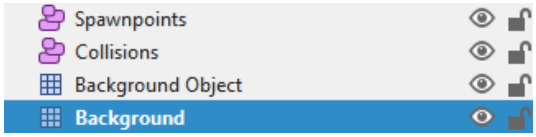
Step 4. Make sure it is all done on load:

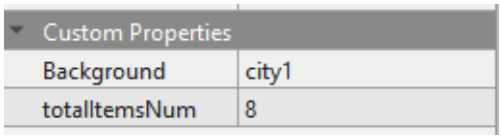
```
@override
FutureOr<void> onLoad() async {
  floor = await TiledComponent.load('$floorName.tmx', Vector2.all(16));
  add(floor);
  _addCollisions();
  return super.onLoad();
}
```

Loading Background Image

To apply your own authentic background image to the game scene, you need to work with the 'Background' layer in Tiled.

Step 1. Create a custom property, 'Background' (String), in the Background layer and specify the image's file name that you want to apply.





Step 2. Get a reference to the Background layer and get its properties in Floor.dart file.

```
void _addBackground() {
  final backgroundLayer = floor.tileMap.getLayer<TileLayer>('Background');
  if (backgroundLayer != null) {
    backgroundName = backgroundLayer.properties.getValue('Background');
    addAll([Background(backgroundName: backgroundName)]);
  }
}
```

`addAll()` will add the background to the game scene.

Step 3. Make sure it is included in the `onload`.

```
@override
FutureOr<void> onLoad() async {
  floor = await TiledComponent.load('$floorName.tmx', Vector2.all(16));
  add(floor);
  _addCollisions();
  _addBackground();
  return super.onLoad();
}
```

Accessing Game Variables

```dart
class RecycleAdventure extends FlameGame
    with
        HasKeyboardHandlerComponents,
        DragCallbacks,
        HasCollisionDetection,
        TapCallbacks {
  //sets default background color
  @override
  Color backgroundColor() => const Color(0xFF211F30);

  late CameraComponent cam;
  Player player = Player(character: 'Hood');
  final int maxHealth = 5;
  late int health; //player health
  int itemsCollected = 0;
  int totalItemsNum = 0;
  bool isOkToNextFloor = false;
  bool showControls = false; //turns on and off joysticks and other buttons
  bool isSoundEffectOn = true;
  bool isMusicOn = true;
  double soundEffectVolume = 1.0;
  double musicVolume = 1.0;
  List<String> floorNames = [
    'Floor-01',
    'Floor-02',
    'Floor-03',
    'Floor-04',
    'Floor-05',
    'Floor-06',
    'Floor-07',
    'Floor-08',
    'BossFight',
  ];
  int currentFloorIndex = 8; //Should initially set to be 0.
```

The player's health and floors handling are all done in recycle_adventure.dart file. You can access any of these variables in the picture in other files by referencing game:

`with HasGameRef<RecycleAdventure >`.

For example, you can interact with player's health by `game.health`:

```dart
if (game.health <= 0)
  dead = true;
```

Player Explained

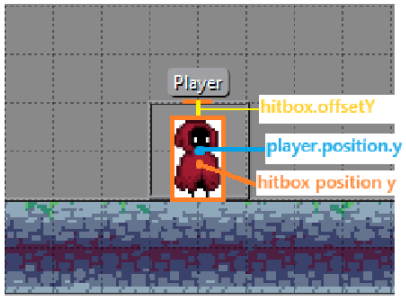
Vertical Collisions with Blocks

To enable the player to stand on and collide with game map terrain, a few calculations are required.

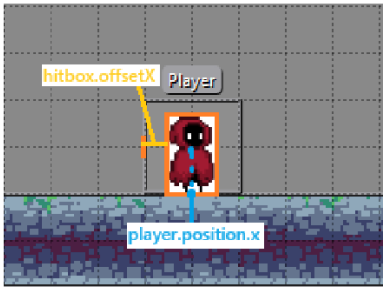
Step 1. Detect Collisions with Blocks

```
bool checkCollision(player, block) {
  final hitbox = player.hitboxSetting;
  final playerX = player.position.x + hitbox.offsetX;
  final playerY = player.position.y + hitbox.offsetY;
  final playerWidth = hitbox.width;
  final playerHeight = hitbox.height;

  final fixedX = player.scale.x < 0
      ? playerX - (hitbox.offsetX * 2) - playerWidth
      : playerX;
  final fixedY = block.isPlatform ? playerY + playerHeight : playerY;

  return (fixedY < block.y + block.height &&
      playerY + playerHeight > block.y &&
      fixedX < block.x + block.width &&
      fixedX + playerWidth > block.x);
}
```

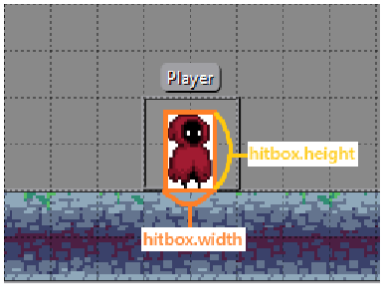
`bool checkCollisions()` detects the player's collisions with blocks in all directions.



playerY = player.position.y + hitbox.offsetY
= hitbox position y



playerX = player.position.x + hitbox.offsetX
= hitbox position X



playerWidth = hitbox.width
playerHeight = hitbox.height

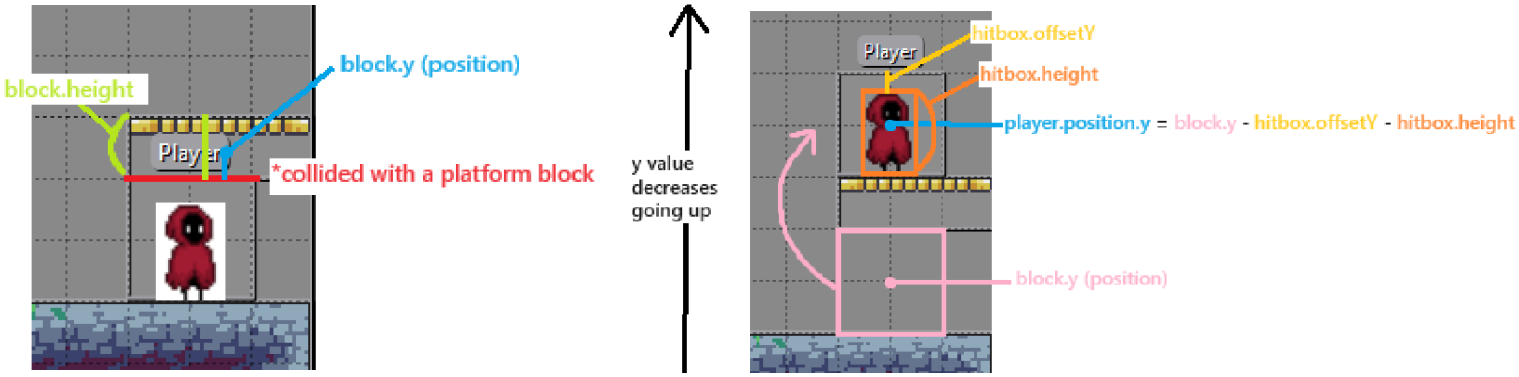
Step 2. Adjusts the Player's Position

<Collisions with Platform block>

```
//Checks collisions with blocks vertically.
void _checkVerticalCollisions() {
  for (final block in collisionBlocks) {
    if (block.isPlatform) {
      //handle collisions with platform.
      if (checkCollision(this, block)) {
        //if falling
        if (velocity.y > 0) {
          velocity.y = 0;
          position.y = block.y - hitboxSetting.height - hitboxSetting.offsetY;
          isOnGround = true;
          break;
        }
      }
```

The player is located on top of the Platform block if they collide with it and jump.

*Illustration:



<Collisions with default block>

```
else {
//handle collisions with any other blocks.
if (checkCollision(this, block)) {
  //if falling
  if (velocity.y > 0) {
    velocity.y = 0;
    position.y = block.y - hitboxSetting.height - hitboxSetting.offsetY;
    isOnGround = true;
    break;
  }
  //if jumping
  if (velocity.y < 0) {
    velocity.y = 0;
    position.y = block.y + block.height - hitboxSetting.offsetY;
    break;
  }
}
```

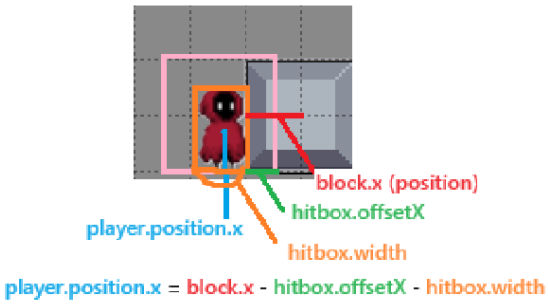
This makes the player able to stand on default blocks.

on a default block

Horizontal Collisions with Terrains

When the player is moving to the right (`player.velocity.x > 0`)

```
void _checkHorizontalCollisions() {
  for (final block in collisionBlocks) {
    if (!block.isPlatform) {
      if (checkCollision(this, block)) {
        if (velocity.x > 0) {
          //when directing to the right.
          velocity.x = 0;
          position.x = block.x - hitboxSetting.offsetX - hitboxSetting.width;
          break;
        }
```

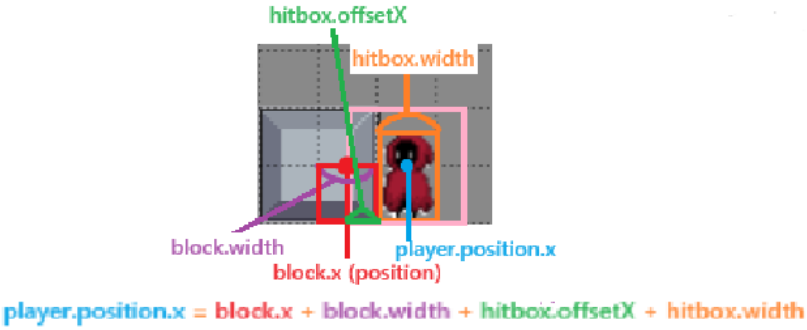


When the player's velocity is positive, this means the player is moving to the right.

If the player is directing to the right and collides with a block, `velocity.x` is set to be zero to stop the player moving any further and adjust their x position.

When player is moving to the left (`player.velocity.x < 0`)

```
if (velocity.x < 0) {
  //when directing to the left.
  velocity.x = 0;
  position.x = block.x +
      block.width +
      hitboxSetting.width +
      hitboxSetting.offsetX;
  break;
}
```



Updating Character Sprite Direction

You can make a character's sprite flip horizontally depending on its direction by using `flipHorizontallyAroundCenter()`.

```
//if going to the left.
if (velocity.x < 0 && scale.x > 0) {
  flipHorizontallyAroundCenter();
  //if going to the right.
} else if (velocity.x > 0 && scale.x < 0) {
  flipHorizontallyAroundCenter();
}
```

When directing right

When directing left

*Update this in `update()`.

Collisions with Other Objects

Other objects include checkpoints, items, traps, and enemies. To detect collisions of player with these objects, using hitbox is one of the easiest ways.

Step 1. Specify `CollisionCallbacks` mixin in any object's class you want to detect collisions.

```
class Player extends SpriteAnimationGroupComponent
    with HasGameRef<RecycleAdventure>, KeyboardHandler, CollisionCallbacks {
```

```
class Item extends SpriteAnimationComponent
    with HasGameRef<RecycleAdventure>, CollisionCallbacks {
```

Step 2. Create `onCollisionStart()` in player.dart

```
@override
void onCollisionStart(
    Set<Vector2> intersectionPoints, PositionComponent other) {
    if (other is Item) other.collidedWithPlayer();

  super.onCollisionStart(intersectionPoints, other);
}
```

If the player collides with an object and if it is Item, it will trigger `collidedWithPlayer()` in the Item class.

So, in item.dart,

```
void collidedWithPlayer() async {
  //write your codes here...
}
```

**`onCollsionStart()` detects the collision only once. It is recommend to use `onCollisionStart()` since it has less potential to cause errors. For multiple times of collisions, `onCollision()` can be used instead.

*Public and Private Methods in Dart:

use '_' to make a method private. Don't if you want it to be public.

```
void _privateMethod() {
  //only accesible within its class.
}

void publicMethod() {
  //can be used in other classes.
}
```

Bullet Shooting on Key Pressed

You can make the player shoot bullets whenever a certain key or a button is pressed.

Step 1. Set up `onKey` event.

```
@override
bool onKeyEvent(RawKeyEvent event, Set<LogicalKeyboardKey> keysPressed) {
  //if Q key was pressed, the player shoots bullet.
  hasShooted = keysPressed.contains(LogicalKeyboardKey.keyQ) && !event.repeat;

  return super.onKeyEvent(event, keysPressed);
}
```

In this code, if the Q key is pressed, `hasShooted` becomes `true`, allowing the player to shoot bullets.

`*!event.repeat` prevents a key from being pressed successively.

Step 2. In `onUpdate`, shoot bullet if `hasShooted` is true.

```
@override
void update(double dt)
    if (hasShooted) {
      _shootBullet();
    }

  super.update(dt);
}
```

At a moment when `hasShooted` becomes true, it triggers `_shootBullet()` which creates a `Bullet`.

Step 3. Create a bullet and add it to the game scene.

```
void _shootBullet() {
  Bullet bullet = Bullet(
    moveVertically: false,
    position: (bulletHorizontalDirection == 1)
        ? Vector2(position.x + 20, position.y + 20)
        : Vector2(position.x - 20, position.y + 20),
    moveDirection: bulletHorizontalDirection,
  ); // Bullet
  add(bullet);

  hasShooted = false;
}
```

`moveVertically`: it makes a bullet move vertically when it is true.

A bullet's position is adjusted depending on which direction it is spawned.

`add(bullet)` adds the bullet to the game scene.

Sets `hasShooted` to be false at the end. This prevents bullets from spawning infinitely.

Bullet Class Example

```
class Bullet extends SpriteAnimationComponent
    with HasGameRef<RecycleAdventure>, CollisionCallbacks {
  bool moveVertically;
  double moveDirection;
  Bullet({
    super.position,
    this.moveVertically = false,
    required this.moveDirection,
  });
```

```
@override
FutureOr<void> onLoad() {
  debugMode = false;
  player = game.player;

  //updates bullet sprite's direction
  updateBulletDirection();

  //bullet hitbox
  add(RectangleHitbox(
    collisionType: CollisionType.passive,
    position: Vector2(2, 9),
    size: Vector2(22, 10),
  )); // RectangleHitbox

  //bullet sprite animation
  animation = SpriteAnimation.fromFrameData(
    game.images.fromCache('Bullet.png'),
    SpriteAnimationData.sequenced(
      amount: 1,
      stepTime: 0.2,
      textureSize: Vector2(16, 16),
    ), // SpriteAnimationData.sequenced
  ); // SpriteAnimation.fromFrameData
  return super.onLoad();
}
```

*`onload`: updates bullet's direction, adds bullet's hitbox, and load sprite animation.

*`update`: updates movement and handles removements

```
@override
void update(double dt) {
  movement(dt);
  super.update(dt);
}

void movement(dt) {
  if (moveVertically) {
    //vertically moves
    position.y -= speed * dt;
  } else {
    //horizontally moves
    position.x += moveDirection * speed * dt;
  }

  //removes bullet
  if (position.y < -game.size.y ||
      position.x < 0 ||
      position.x > game.size.x) {
    removeFromParent();
  }
}
```

```
void updateBulletDirection() {
  if (moveDirection == 1) {
    flipHorizontallyAroundCenter();
  }
  if (moveVertically) {
    angle = -90; //makes bullets direct upward
  }
}
```

*Flips bullet sprite horizontally depending on the direction it is spawned.

Player Respawn (Creating Successive Animations)

The player's respawn has successive animations. This can be achieved in multiple ways.

The first way to do this is using `animationTicker?.completed` to make sure the following animation is played after the prior one is completed.

```
void respawn() async {
  gotHit = true;
  current = PlayerState.hit;

  await animationTicker?.completed;
  animationTicker?.reset();

  scale.x = 1; //makes player face to the right.
  position = startingPosition - Vector2.all(32);
  current = PlayerState.appearing;

  await animationTicker?.completed;
  animationTicker?.reset();

  velocity = Vector2.zero();
  position = startingPosition;
  _updatePlayerState();
  Future.delayed(
    const Duration(microseconds: 400),
    () => gotHit = false,
  ); // Future.delayed
}
```

Use `animationTicker?.completed` and `animationTicker?.reset()` to make the flow of animations smooth.

Another way is giving a time delay between animations instead of using `animationTicker`.

`Future.delayed()` receives a duration time in the first parameter, and a function to trigger after a certain time in the second parameter.

The duration times can be in microseconds, milliseconds, and seconds etc.

Applying Sprite Animations

Step 1. Declare Enum States

To dynamically change game character's animation, '`SpriteAnimationGroupComponent`' can be used.
In the game character class, you can declare its states with `enum` and assign animations to each state.

* In the case of `Player` class:

```
enum PlayerState {
  idle,
  running,
  jumping,
  falling,
  hit,
  attack,
  dead,
  appearing,
  disappearing
}

class Player extends SpriteAnimationGroupComponent
    with HasGameRef<RecycleAdventure>, KeyboardHandler, CollisionCallbacks {
  String character;
  Player({
    super.position,
    required this.character,
  });

  late final Player player;
  late final SpriteAnimation idleAnimation;
  late final SpriteAnimation runningAnimation;
  late final SpriteAnimation jumpingAnimation;
  late final SpriteAnimation fallingAnimation;
  late final SpriteAnimation hitAnimation;
  late final SpriteAnimation appearingAnimation;
  late final SpriteAnimation disappearingAnimation;
  late final SpriteAnimation attackAnimation;
  late final SpriteAnimation deadAnimation;
```

You should first declare the sprite animation's name with `late` since the animation will be assigned later '`on load`'.

Step 2. Calling Sprite Animations

```
SpriteAnimation _spriteAnimation(String state, int amount, double stepTime) {
  return SpriteAnimation.fromFrameData(
    game.images.fromCache('Main Characters/$character/$state (32x32).png'),
    SpriteAnimationData.sequenced(
      amount: amount,
      stepTime: stepTime,
      textureSize: Vector2.all(32),
    ), // SpriteAnimationData.sequenced
  ); // SpriteAnimation.fromFrameData
}
```

This is the general structure of sprite animation application. Creating a method is useful to create multiple animations.

- `amount` (int): the number of sprites in an animation.
- `stepTime` (double): how fast the animation will be played.
- `textureSize` (Vector2): the size of the animation sprite

*Character animation sprite sheet example:
This is the player's idle animation sprite. Its amount is 4 and `textureSize` is Vector2(32, 32) (*32x32 pixels).

Step 3. Load All Animations

```
void _loadAllAnimations() {
  idleAnimation = _spriteAnimation('Idle', 4, 0.27);
  runningAnimation = _spriteAnimation('Run', 8, 0.05);
  jumpingAnimation = _spriteAnimation('Jump', 1, 0.27);
  fallingAnimation = _spriteAnimation('Fall', 1, 0.27);
  hitAnimation = _spriteAnimation('disappear', 3, 0.25)..loop = false;
  attackAnimation = _spriteAnimation('attack', 8, 0.2)..loop = false;
  deadAnimation = _spriteAnimation('dead', 8, 0.27)..loop = false;
  appearingAnimation = _specialSpriteAnimation('Appearing', 7)..loop = false;
  disappearingAnimation = _specialSpriteAnimation('Disappearing', 7)
    ..loop = false;

  animations = {
    PlayerState.idle: idleAnimation,
    PlayerState.running: runningAnimation,
    PlayerState.jumping: jumpingAnimation,
    PlayerState.falling: fallingAnimation,
    PlayerState.hit: hitAnimation,
    PlayerState.attack: attackAnimation,
    PlayerState.dead: deadAnimation,
    PlayerState.appearing: appearingAnimation,
    PlayerState.disappearing: disappearingAnimation,
  };

  current = PlayerState.idle;
}
```

SprtieAnimationGroupComponent allows you to declare multiple animations to the states. ‘current’ sets the current player’s state.

Then, load the animations in `onload`:

```
@override
FutureOr<void> onLoad() {
  _loadAllAnimations();
  debugMode = false;
```

Enemies Explained

Enemy Class

All enemies (except boss) extend Enemy class. This is to improve code efficiency and avoid repetitions of code.

```
class Enemy extends SpriteAnimationGroupComponent
    with HasGameRef<RecycleAdventure>, CollisionCallbacks {
  final double offsetPositive;
  final double offsetNegative;
  int lives;
  Enemy({
    super.position,
    super.size,
    this.offsetPositive = 0,
    this.offsetNegative = 0,
    this.lives = 1, //default lives
  });

  double stepTime = 0.05;
  double tileSize = 16;
  double moveSpeed = 55;

  double rangeNegative = 0;
  double rangePositive = 0;
  Vector2 velocity = Vector2.zero();
  Vector2 moveDirection = Vector2(1, 0);
  Vector2 targetDirection = Vector2(-1, 0);
```

To create an enemy, you must initialize its **positive and negative offsets**. These numbers decide how large an enemy's detection range. You can adjust an enemy's detection range with these two values. The offsets are the properties of a spawn point in Tiled, so they should be initialized in Tiled.

Other properties that you can include to reuse include animation step time, running speed, velocity, and moving direction, and so on.

The '`rangeNegative`' and '`rangePositive`' will be calculated with the given offsets and changed later.

<Methods included in Enemy class>

- `void calculateRange()` : calculates the range of an enemy's player detection.
- `bool isPlayerInRange()` : returns true if the player is within the enemy's range.
- `void movement(dt)` : moves enemy. This should be called in `onUpdate()`.

```
void calculateRange() {
  rangeNegative = position.x - offsetNegative * tileSize;
  rangePositive = position.x + offsetPositive * tileSize;
}
```

`rangeNegative`: the left side of an enemy to detect.

`rangePositive`:the right side of an enemy to detect.

*The tile size is multiplied for accuracy. The size of a tile in this game is 16 x 16 pixels.

```
bool isPlayerInRange() {
  double playerOffsetX = (player.scale.x > 0) ? 0 : -player.width;

  return (player.x + playerOffsetX >= rangeNegative &&
      player.x + playerOffsetX <= rangePositive);
}
```

`playerOffsetX`: player's horizontal offset that is adjusted depending on the player's direction.

You can also include `player.y + player.height > position.y` to make enemies stop detecting player when the player is not on the ground.

If the player's horizontal position is larger than `rangeNegative` or less than `rangePositive`, the player is considered within the enemy's detection range.

```
//default movement
  void movement(dt) {
    velocity.x = 0;

    double playerOffset = (player.scale.x > 0) ? 0 : -player.width;
    double enemyOffset = (scale.x > 0) ? 0 : -width;

    if (isPlayerInRange()) {
      targetDirection.x =
          (player.x + playerOffset < position.x + enemyOffset) ? -1 : 1;
      velocity.x = targetDirection.x * runSpeed;
    }

    //Changes enemy's direction when player changes direction.
    moveDirection.x = lerpDouble(moveDirection.x, targetDirection.x, 0.1) ?? 1;

    position.x += velocity.x * dt;
  }
```

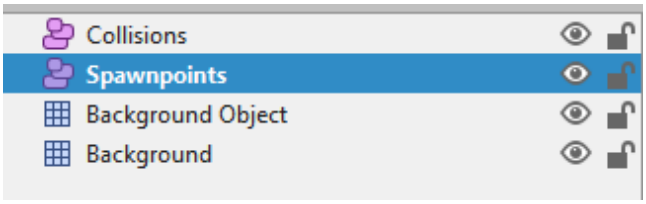
Once `isPlayerRange()` returns true, the enemy starts chasing the player until the player is out of their range.

*Methods that shouldn't be included in Enemy class:

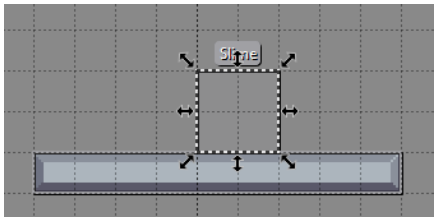
Never create `enum` states in Enemy class to override in other classes. This will make the game unproperly work and not load the game scene, displaying a blank screen. Be sure to make separate states for each enemy when working with sprite animations.

Adding Enemies to the Game Scene

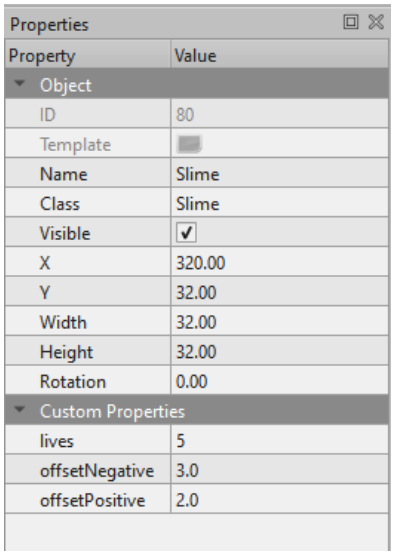
The following page are the instructions of enemies. Be sure to include the required properties at 'Spawnpoints' layer in Tiled to create enemies.



1.Create and click 'Spawnpoints' layer



2.Create a box on the map.



Property	Value
Object	
ID	80
Template	
Name	Slime
Class	Slime
Visible	✓
X	320.00
Y	32.00
Width	32.00
Height	32.00
Rotation	0.00
Custom Properties	
lives	5
offsetNegative	3.0
offsetPositive	2.0

3.Click the box and add properties and class name in its properties tab.

* This part is important! If it is not done properly, this will cause an error (The main factor of game screen not loaded).

4. Add enemies to the game scene (This is done in floor.dart file).

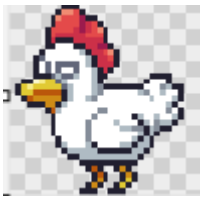
```
void _spawningObjects() {
  //gets spawnPoints layer
  final spawnPointsLayer = floor.tileMap.getLayer<ObjectGroup>('Spawnpoints');
  //gets properties from Tiled and adds the enemy to the game scene.
  if (spawnPointsLayer != null) {
    for (final spawnPoint in spawnPointsLayer.objects) {
      switch (spawnPoint.class_) {
        case 'Slime':
          final slime = Slime(
            position: Vector2(spawnPoint.x, spawnPoint.y),
            size: Vector2(spawnPoint.width, spawnPoint.height),
            offsetPositive: spawnPoint.properties.getValue('offsetPositive'),
            offsetNegative: spawnPoint.properties.getValue('offsetNegative'),
            lives: spawnPoint.properties.getValue('lives'),
          ); // Slime
          add(slime);
          break;
```

Slime Enemy

- required properties in Tiled:
- `offsetNegative (double), offsetPositive (double), lives (int)`
- Class name in Tiled: Slime
- Lives: 3
- Projectile spawning: X
- Specialty: Player can stomp and kill it. / Becomes an unharmful particle chasing player once it dies.

*the particle state (hitbox is removed)

Chicken Enemy

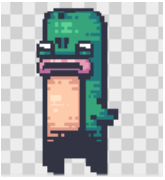


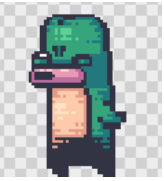
- required properties in Tiled:
- `offsetNegativ(double), offsetPositive(double), lives(int)`
- Class name in Tiled: Chicken
- Lives: 3
- Projectile spawning: X
- Specialty: Player can stomp and kill it.

Bat Enemy

- required properties in Tiled:
  `offsetNegative(double),offsetPositive(double), lives(int)`
- Class name in Tiled: Slime
- Lives: 3
- Projectile spawning: X
- Specialty: It chases the player horizontally and vertically.

Cucumber Enemy





- required properties in Tiled:
  `offsetNegative(double), offsetPositive(double), lives (int)`
- Class name in Tiled: Cucumber
- Lives: 5 / lives for dead ground: 3
- Projectile spawning: O
- Specialty: shoots projectiles / randomly jump over the player.

Whale Enemy

- required properties in Tiled:
  `offsetNegative (double), offsetPositive (double), lives (int)`
- Class name in Tiled: Whale
- Lives: 10
- Projectile spawning: O
- Specialty: shoots projectiles

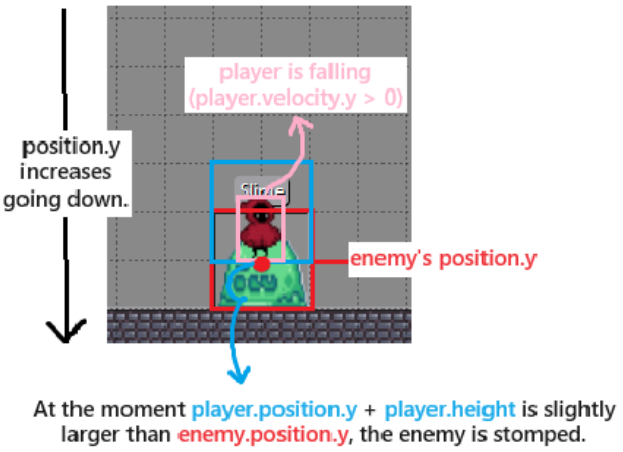
Stomping Enemy

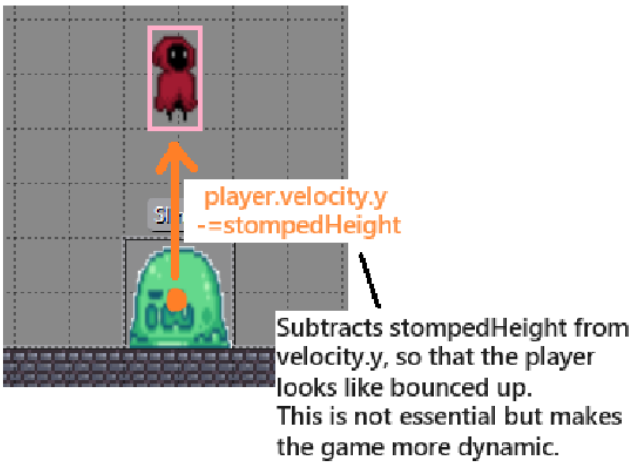
The player can stomp some enemies on their head and defeat them. To achieve this, the enemy's height and player's bottom position are required for the calculation of collisions.

For example, in the case of slime enemy,

```
void _collidedWithPlayer() async {
  if (player.velocity.y > 0 && player.y + player.height > position.y) {
    if (game.playSounds) {
      FlameAudio.play('enemyKilled.wav', volume: game.soundVolume);
    }
    dead = true;
    current = State.hit;
    player.velocity.y = -_stompedHeight;
    await animationTicker?.completed;
    onDead();
  } else {
    game.health--;
    player.respawn();
  }
}
```

A slime is considered stomped by the player if `player.velocity.y > 0` (the player is falling) and `player.y + player.height >` (this slime's) `position.y`.





Enemy Projectile Spawn Manager

To make an enemy shoot projectile in intervals, using Timer is one of the ways to achieve this. This is done by creating a projectile spawn manager that produces projectile using timer. The created projectile spawn manager can be implemented to an enemy to make it shoot projectiles.

Step 1. Create Projectile

```
class Projectile extends SpriteAnimationComponent
    with HasGameRef<RecycleAdventure>, CollisionCallbacks {

  Projectile({
    super.position,
  });

  @override
  FutureOr<void> onLoad() {
    //load sprite animations and hitbox.
    return super.onLoad();
  }

  @override
  void update(double dt) {
    updateBulletDirection();
    movement(dt);
    super.update(dt);
  }
```

Step 2. Create Projectile Spawn Manager

```
class EnemyProjectileManager extends Component {
  late Timer _timer;
  late Vector2 position;
  late double limit;
  EnemyProjectileManager({
    required this.position,
    required this.limit,
  }) : super() {
    _timer = Timer(limit, onTick: _spawnProjectiles, repeat: true);
  }

  @override
  void onMount() {
    super.onMount();
    _timer.start();
  }

  @override
  void onRemove() {
    super.onRemove();
    _timer.stop();
  }

  @override
  void update(double dt) {
    super.update(dt);
    _timer.update(dt);
  }
void _spawnProjectiles() {
  Projectile projectile = Projectile(
    position: position,
    moveDirection: -1,
    hitbox: RectangleHitbox(
      collisionType: CollisionType.passive,
      position: Vector2(2, 9),
      size: Vector2(22, 10),
    ), // RectangleHitbox
  ); // Projectile
  add(projectile);
}
```

Step 3. Implement the manager

```
class Whale extends Enemy {
  Whale({
    super.position,
    super.size,
    super.offsetPositive,
    super.offsetNegative,
    super.lives,
  });

  EnemyProjectileManager? _projectileManager;

  @override
  FutureOr<void> onLoad() {
    _projectileManager = EnemyProjectileManager(
      position: Vector2(
        position.x + 9,
        position.y + 18,
      ), // Vector2
      limit: 1,
    ); // EnemyProjectileManager
    parent?.add(_projectileManager!);

    return super.onLoad();
  }
```

Traps Explained

Saw Trap

Custom Properties	
initial direction	1
isVertical	☑
offsetNegative	2.0
offsetPositive	2.0

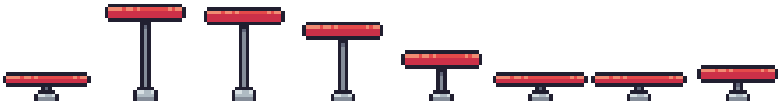
- Required Properties: `offsetNegative` (double), `offsetPositive` (double), `isVertical` (bool), `initialDirection` (int)
- Class Name in Tiled: Saw

`isVertical` determines if a saw will move vertically or horizontally.

```
//Sets movement range.
if (isVertical) {
  //vertical movement
  rangeNegative = position.y - offsetNegative * tileSize;
  rangePositive = position.y + offsetPositive * tileSize;
} else {
  //horizontal movement
  rangeNegative = position.x - offsetNegative * tileSize;
  rangePositive = position.x + offsetPositive * tileSize;
}
```

Traps' offsets work the same way as the enemies.

Trampoline



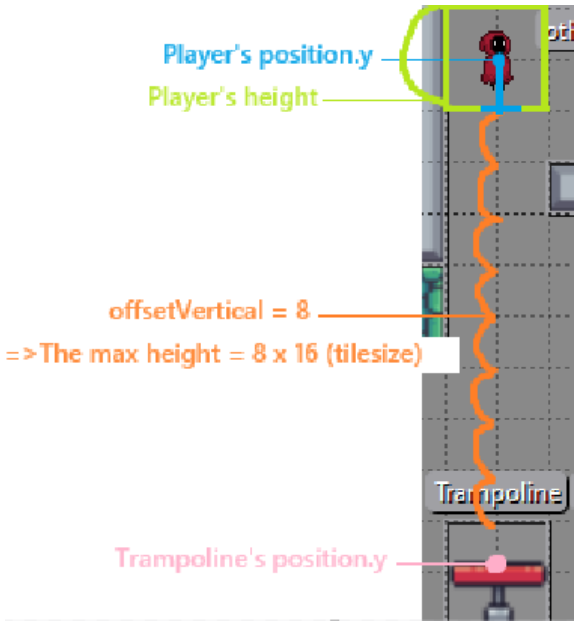
Custom Properties	
offsetVertical	13

Trampoline bounces the player once it detects a collision with them.

- Required Property: `offsetVertical` (int)
- Class name in Tiled: Trampoline

```
bounceHeight = position.y +
    offsetVertical * tileSize -
    player.position.y +
    player.height;
```

`offsetVertical` is used to calculate the maximum height of the player being bounced.



Once the player detects a collision with a trampoline, it changes the player's vertical movement direction and bounces them up.

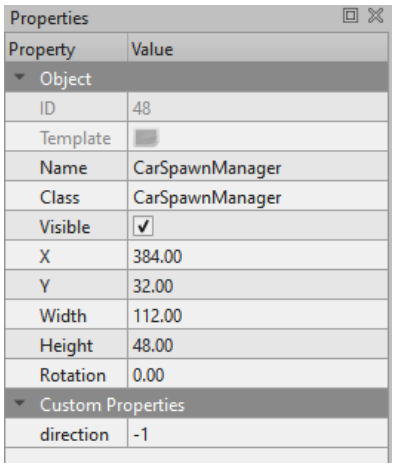
```
void _bouncePlayer(dt) {
  player.velocity.y = -bounceHeight;
  player.position.y += player.velocity.y * dt;
  player.hasJumped = true;
  player.isOnGround = false;
}
```

By setting `player.hasJumped = true` and `player.isOnGround = false`, the player will fall once they reach the maximum height.

Car Trap & Car Spawn Manager

Car Spawn Manager produces cars randomly colored in certain intervals.

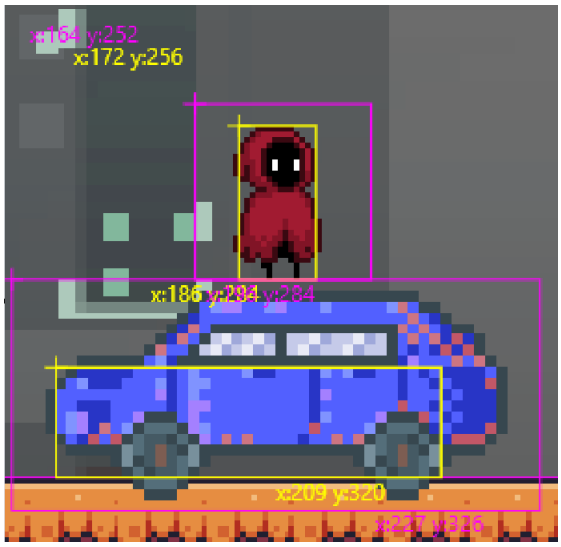
To add car traps, you should create `CarSpawnManager` in Tiled, not cars alone.



Property	Value
Object	
ID	48
Template	
Name	CarSpawnManager
Class	CarSpawnManager
Visible	✓
X	384.00
Y	32.00
Width	112.00
Height	48.00
Rotation	0.00
Custom Properties	
direction	-1

- Required Property: `direction` (int)
- Class Name in Tiled: CarSpawnManager

'`direction`' determines the direction of Car Spawn Manager spawning cars and their movements.



Car traps are one of difficult, easy-to-collide traps. The player should step on top of the cars to get over them.

To make the player able to step on a car, not hit by them, some calculations are required.

The yellow boxes are hitboxes that detect collisions between objects.

The player dies if they collide with other objects' hitboxes, so we should make the player not fall and collide with a car's hitbox when the player jumps and steps on the top of a car.

However, if the player collides with a car's hitbox horizontally, they are hit by the car.

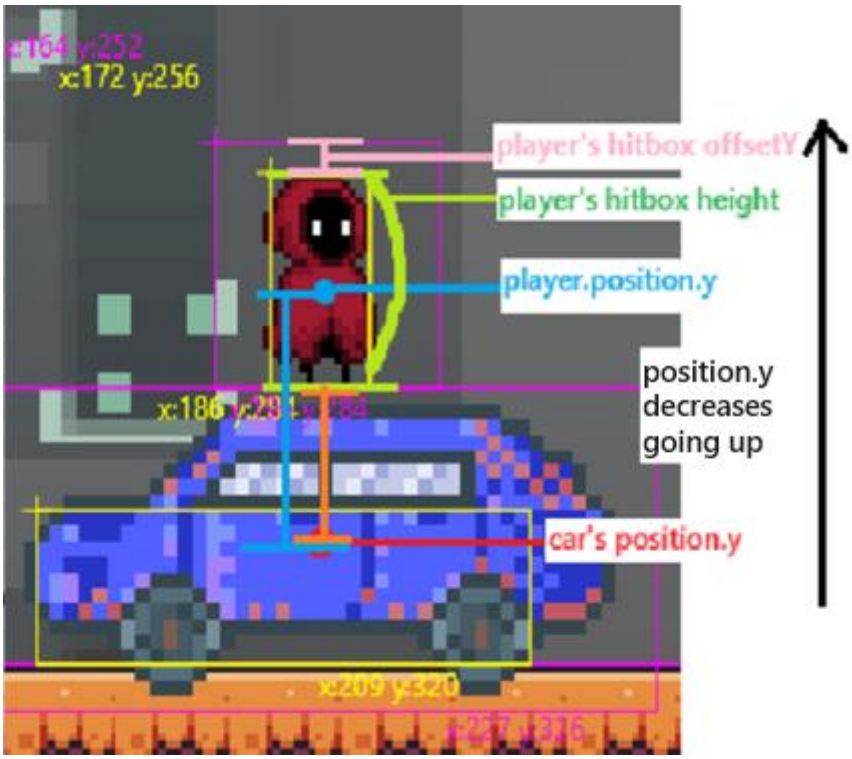
```
void _checkVerticalCollisions(dt) {
  if (checkCollision(player, this)) {
    player.position.x += velocity.x * dt;

    if (player.velocity.y > 0) {
      player.velocity.y = 0;
      player.position.y = position.y -
          player.hitboxSetting.height -
          player.hitboxSetting.offsetY;
      player.isOnGround = true;
    }

    if (player.velocity.y < 0) {
      player.isOnGround = false;
      player.velocity.y = 0;
      player.position.y = position.y + height - player.hitboxSetting.offsetY;
    }
  }
}
```

For a car's vertical collision detection with the player, I utilized `checkCollision(player, block)` that was used for player and game platform block collisions.

if `player.velocity.y` is positive, this means the player has jumped.

Once the player and a car collide and if the player is in the state of jump, it is considered that player has fallen on top of the car.

Then, `player.velocity.y` is set to be 0 to stop the player falling, and the player's y position is adjusted to make sure the player to locate on top of the car.

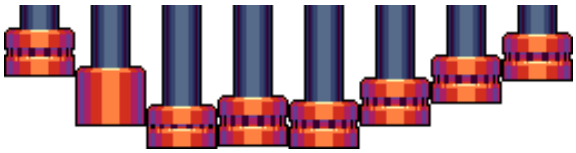


To handle when the player jumps on the top of the car, we should detect when the `player.velocity.y` value turns negative (`player.velocity.y < 0` means the player has jumped).

Once the player has jumped, sets `player.isOnGround` to be `false` so that the gravity is applied to the player and they fall down.

Then `player.position.y` is adjusted again to make the player jump from the top of the car.

Hammer Trap



- Required Properties: None
- Class Name in Tiled: Hammer

Hammers are the traps used in the factory stage. If the player collides with the bottom of a hammer, they die and respawn. To make a hammer's hitbox dynamic along with the sprite animation, I chose to make the hitbox extend and shrink consecutively.

It was difficult to adjust the speed of the hitbox transforming by the speed of the animation. To calculate this speed, I used the amount of sprite animation frames and stepTime.

*Still fixing

Rock Head Trap

Rock Head trap is a fascinating, motile trap that the player can stand on and move along with it together.

Its collision detection with the player works the same as Car trap.

Properties

Property	Value
Object	
ID	18
Template	
Name	RockHead
Class	RockHead
Visible	✓
X	528.00
Y	272.00
Width	48.00
Height	48.00
Rotation	0.00
Custom Properties	
offsetHorizontal	29
offsetVertical	14

The Rock Head trap gets offsetHorizontal (int) and offsetVertical(int) to define its customizable pathway.

```
void _movement(dt) async {
  if (directionY == 0) {
    //on horizontal movement
    if (directionX == -1) {
      //if moving to the left
      if (position.x >= 528 - offsetHorizontal * tileSize) {
        velocity.x = -speed;
      } else {
        //stop moving horizontally and start moving vertically.
        directionX = 0;
        velocity.x = 0;
        if (position.y >= 272) {
          directionY = -1; //goes up
        } else {
          directionY = 1; //goes down
        }
      }
    } else if (directionX == 1) {
      //if moving to the right
      if (position.x <= 528) {
```

Rock Head moves in a square shape. Its `offsetHorizontal` and `offsetVertical` determine how far it moves horizontally and vertically.

Boss Explained

The boss has three patterns, and to randomly pick up and generate the patterns in certain intervals, set up a timer that runs in certain intervals.

Triggering Patterns Using Timer

```
class Boss extends SpriteAnimationGroupComponent
    with HasGameRef<RecycleAdventure>, CollisionCallbacks {
  late Timer _timer;
  Boss({
    super.position,
    super.size,
  }) : super() {
    //randomly chooses pattern every second.
    _timer = Timer(1, onTick: _randomlyChoosePattern, repeat: true);
  }
```

The boss randomly chooses a pattern every one second. However, it will not choose a pattern when one pattern is on running.

```
final int maxLives = 120;
int lives = 120;
bool dead = false;
bool isHitOn = false;
Vector2 velocity = Vector2.zero();
double directionX = 0;
double moveSpeed = 80;

bool onPattern1 = false;
bool onPattern2 = false;
bool onPattern3 = false;
```

This is the setting of the boss.

isHitOn, onPattern1, 2, and 3 are used to manage patterns.

```
void _randomlyChoosePattern() {
  if (!onPattern1 && !onPattern2 && !onPattern3 && !isHitOn) {
    int rd = Random().nextInt(3);

    switch (rd) {
      case 0:
        onPattern1 = true;
        directionX = -1;
        hitbox.size = Vector2(32, 78);
        hitbox.position = Vector2(32, 0);
        _pattern1();
        break;
      case 1:
        onPattern2 = true;
        directionX = 1;
        _pattern2();
        break;
      case 2:
        onPattern3 = true;
        directionX = 1;
        _pattern3();
        break;
    }
  }
}
```

The timer triggers `_randomlyChoosePattern()` every one second but it will play a pattern only if onPattern1, onPattern2, onPattern3, and isHitOn are all false.

*directionX is used to handle the horizontal movement direction of the boss.

```
@override
void onCollisionStart(
    Set<Vector2> intersectionPoints, PositionComponent other) {
  super.onCollisionStart(intersectionPoints, other);
  if (other is Player) {
    game.health--;
    player.respawn();
  }
  if (isHitOn) {
    if (other is Bullet) {
      if (game.isSoundEffectOn) {
        FlameAudio.play('boss-damaged.mp3', volume: game.soundEffectVolume);
      }
      current = State.hit;
      Future.delayed(const Duration(milliseconds: 200), () {
        current = State.idle;
      }); // Future.delayed
      lives--;
      other.removeFromParent();
    }
  }
```

In collision detection, the boss is damaged when isHitOn is true.

This is to prevent the player from defeating the boss quickly and enable them to attack only when they are allowed.

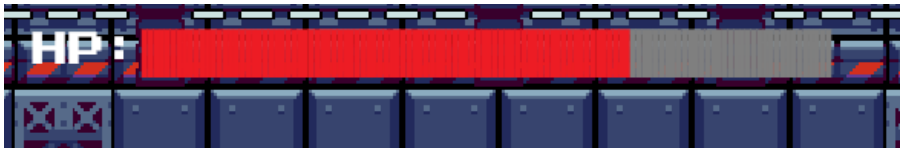
```
Future.delayed(const Duration(seconds: 500), () {
  isHitOn = true;
}); // Future.delayed

Future.delayed(const Duration(seconds: 4), () {
  isHitOn = false;
  onPattern3 = false;
}); // Future.delayed
```

At the end of every pattern, isHitOn is true for 4 seconds, giving time for the player to attack the boss.

```
void _checkLives() {
  if (lives <= 0) {
    if (lives == 0) {
      if (game.isSoundEffectOn) {
        FlameAudio.play('boss-dead.mp3', volume: game.soundEffectVolume);
        lives--;
      }
    }
    dead = true;
    isHitOn = false;
    current = State.dead;
    _timer.stop();
  }
}
```

On the boss' death, it stops timer and sets the current state to be 'dead'.

Boss HP Bar




```
enum State { available, unavailable }

class BossHealthBar extends SpriteGroupComponent
    with HasGameRef<RecycleAdventure> {
  final int barNumber;
  final Boss boss;
  BossHealthBar({
    super.position,
    super.size,
    super.priority,
    required this.barNumber,
    required this.boss,
  });
```

The boss' HP bar works the same as the player's health bar.

It requires Boss as a parameter so that all tiles have the same Boss and reflect its current health.

```
@override
void update(double dt) {
  if (boss.lives < barNumber) {
    current = State.unavailable;
  } else {
    current = State.available;
  }
  super.update(dt);
}
```

If the boss health is smaller than a bar's number (that reflects the order of the bar), the tile changes from the red (available) to the grey tile (unavailable).

The boss' health number of tiles are created and consist of the bar.

In floor.dart file:

```
case 'Boss':
  final boss = Boss(
    position: Vector2(spawnPoint.x, spawnPoint.y + 17),
    size: Vector2(spawnPoint.width, spawnPoint.height),
  ); // Boss
  add(boss);
  //adds boss health bar HUD
  for (int i = 1; i <= boss.maxLives; i++) {
    final positionX = 1.9 * i;
    await add(
      BossHealthBar(
        boss: boss,
        barNumber: i,
        position: Vector2(200 + positionX.toDouble(), 13),
        size: Vector2(2, 16),
      ), // BossHealthBar
    );
  }
```

The health bar is created when a boss is created. This process is done in floor.dart in order to make the boss accessible in every single tiles when a boss is created in a 'BossFight' floor.

A tile's index is assigned to its bar number.

Hud, Widgets, and Screens (Menus)

HUDs extend `PositionComponent`. HUDs are added on the foremost layer of the game, so its priority should be the highest.

Heart Bar Hud

To make a responsive heart bar, you should, of course, work with the player's current number of lives. You can access it from the game reference: `game.health`.

```
enum HeartState { available, unavailable }

class HeartHealthComponent extends SpriteGroupComponent
    with HasGameRef<RecycleAdventure> {
  final int heartNumber;
  HeartHealthComponent({
    required this.heartNumber,
    required super.position,
    required super.size,
    super.scale,
    super.angle,
    super.anchor,
    super.priority,
  });
```

Step 1. Create Heart class.

There is total two states required: available and unavailable.

The Heart HUDS extends `SpriteGroupComponent`, meaning it's not animated.

(It is like `SpriteAnimationGroupComponent` but without animation, so you should use `sprites` instead of `animations` to load multiple sprites.)

The `heartNumber` represents the maximum number of the player's lives.

```
@override
void update(double dt) {
  if (game.health < heartNumber) {
    current = HeartState.unavailable;
  } else {
    current = HeartState.available;
  }
  super.update(dt);
}
```

On `update()`, if current player's lives are less than the maximum number of lives, a heart's current state will be set to be unavailable, displaying an empty, grey heart.

Step 2. Add the heart bar HUD to the game scene.

In the hud.dart file,

```
void _addHeartHealthComponent() async {
  for (int i = 1; i <= maxHeartNum; i++) {
    final positionX = 25 * i;
    await add(
      HeartHealthComponent(
        heartNumber: i,
        position: Vector2(positionX.toDouble() - 10, 10),
        size: Vector2.all(16),
      ), // HeartHealthComponent
    );
  }
}
```

Creates and adds the maximum number of Hearts. You can make a gap between hearts with a loop.

Be sure to get this done `onload()`.

Pause Game Button Widget (Using Game Loop)

You can pause and resume the game by using the game loop.

Making widgets is done the same way as building a mobile application in Flutter.

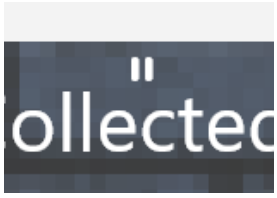
```
class PauseButton extends StatelessWidget {
  static const String ID = 'PauseButton';
  final RecycleAdventure gameRef;
  const PauseButton({
    super.key,
    required this.gameRef,
  });

  @override
  Widget build(Object context) {
    return Align(
      alignment: Alignment.topCenter,
      child: TextButton(
        child: const Icon(
          Icons.pause_rounded,
          color: Colors.white,
        ), // Icon
        onPressed: () {
          gameRef.pauseEngine();
          FlameAudio.bgm.pause();
          gameRef.overlays.add(PauseMenu.ID);
          gameRef.overlays.remove(PauseButton.ID);
        },
      ), // TextButton
    ); // Align
  }
}
```

`onPressed` is where you pause the game.

`gameRef.pauseEngine()` stops the game loop.

Then, by using overlays, you can add any other widgets on the screen: `gameRef.overlays.add(Widget.ID);`

*Every widget has their own unique ID, which is used to add and remove it from the overlays.



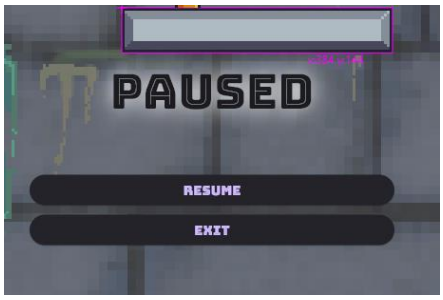
Pause Button

For resume button,

```
//Resume Button
SizedBox(
  width: MediaQuery.of(context).size.width / 5,
  child: ElevatedButton(
    onPressed: () {
      gameRef.resumeEngine();
      gameRef.overlays.remove(PauseMenu.ID);
      gameRef.overlays.add(PauseButton.ID);
    },
    child: const Text('Resume'),
  ), // ElevatedButton
), // SizedBox
```

You can resume the paused game loop by `gameRef.resumeEngine()`.

This is what is shown after clicking the Pause button:



*

Overlay Builder Map

```
class GamePlay extends StatelessWidget {
  const GamePlay({super.key});

  @override
  Widget build(Object context) {
    return Scaffold(
      body: PopScope(
        canPop: false,
        child: GameWidget(
          game: gameRef,
          initialActiveOverlays: const [PauseButton.ID],
          overlayBuilderMap: {
            PauseButton.ID: (BuildContext context, RecycleAdventure gameRef) =>
                PauseButton(
                  gameRef: gameRef,
                ), // PauseButton
            PauseMenu.ID: (BuildContext context, RecycleAdventure gameRef) =>
                PauseMenu(
                  gameRef: gameRef,
                ), // PauseMenu
            GameOverMenu.ID: (BuildContext context, RecycleAdventure gameRef) =>
                GameOverMenu(
                  gameRef: gameRef,
                ), // GameOverMenu
          },
        ), // GameWidget
      ), // PopScope
    ); // Scaffold
  }
}
```

To activate all widgets created above the last thing you need to do is to create an overlay builder map. This is where you activate all the widgets' ID to add and remove them from the game reference.

*gameRef is initialized in main.dart file for global use.

Game Menus (Screen)

You can make multiple screens within a game, such as the main menu.

Step 1. Create Menu Widget

```
class MainMenu extends StatelessWidget {
  const MainMenu({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Container(
        decoration: const BoxDecoration(
          image: DecorationImage(
            image: AssetImage("assets/images/Menu/main_menu.png"),
            fit: BoxFit.cover,
          ), // DecorationImage
        ), // BoxDecoration
        child: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              SizedBox(
                width: MediaQuery.of(context).size.width / 5,
                child: ElevatedButton(
                  onPressed: () {
                    Navigator.of(context).pushReplacement(
                      MaterialPageRoute(
                        builder: (context) => const GamePlay(),
                      ), // MaterialPageRoute
                    );
                  },
                  child: const Text('Play'),
                ), // ElevatedButton
              ), // SizedBox
```

This is the Play button on the main menu. If the button is pressed, it will navigate to the game.

`Navigator.of(context).pushReplacement()` replaces a screen to another one. You can route the game scene by calling it with `MaterialPageRoute()`.

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Container(
      decoration: const BoxDecoration(
        image: DecorationImage(
          image: AssetImage("assets/images/Menu/main_menu.png"),
          fit: BoxFit.cover,
        ), // DecorationImage
      ), // BoxDecoration
```

If you want to use a custom image for the background, you can specify this in `decoration` parameter.

`BoxDecoration` loads an image in box shape. You can also set its type to fit.

Step 2. Call the main menu on program start.

```
Run | Debug | Profile
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Flame.device.fullScreen();
  await Flame.device.setLandscape();

  runApp(MaterialApp(
    themeMode: ThemeMode.dark,
    darkTheme: ThemeData.dark().copyWith(
      textTheme: GoogleFonts.bungeeInlineTextTheme(),
      scaffoldBackgroundColor: Colors.black,
    ),
    home: const MainMenu(),
  )); // MaterialApp
}
```

By setting `MainMenu` to be home, this will display the main menu first.

This is the result of the main menu.

Sound Effects & Music (Flame Audio)

Sound Effects

You can add sound effects for a better game experience with Flame Audio.

`FlameAudio.play()` plays music and can adjust the volume.

For example,

```
void _playerJump(double dt) {
  if (game.playSounds) {
    FlameAudio.play('jump.wav', volume: game.soundVolume);
  }
```

Play 'jump.wav' when the player jumps.

*In recycle_adventure.dart:

```
bool playSounds = false; //turns on game audios
double soundVolume = 1.0;
```

Multipe Game Levels with Different BGM

`FlameAudio.bgm` is one way to play background music. To apply different music to multiple levels, I achieved this by changing and playing music whenever a floor is loaded.

In recycle_adventure.dart:

```
void playBackgroundMusic(String floorName) {
  if (isMusicOn) {
    switch (floorName) {
      case 'Floor-01':
        FlameAudio.bgm.play('tutorial-music.mp3', volume: musicVolume);
        break;
      case 'Floor-02':
      case 'Floor-03':
      case 'Floor-04':
        FlameAudio.bgm.play('sewer-music.mp3', volume: musicVolume);
        break;
      case 'Floor-05':
      case 'Floor-06':
        FlameAudio.bgm.play('city-music.mp3', volume: musicVolume);
        break;
      case 'Floor-07':
        FlameAudio.bgm.play('a-short-break-music.mp3', volume: musicVolume);
        break;
      case 'Floor-08':
        FlameAudio.bgm.play('factory-music.mp3', volume: musicVolume);
        break;
      case 'BossFight':
        FlameAudio.bgm.play('boss-fight-music.mp3', volume: musicVolume);
        break;
    }
  }
```

I made a public method that plays a bgm depending on the floor name.

*Most importantly, initialize gameRef in main.dart to create global gameRef and use it in other files.

```
RecycleAdventure gameRef = RecycleAdventure();

Run | Debug | Profile
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Flame.device.fullScreen();
  await Flame.device.setLandscape();
```

```
SizedBox(
  width: MediaQuery.of(context).size.width / 5,
  child: ElevatedButton(
    onPressed: () {
      Navigator.of(context).pushReplacement(
        MaterialPageRoute(
          builder: (context) => const GamePlay(),
        ), // MaterialPageRoute
      );
      gameRef.playBackgroundMusic(
          gameRef.floorNames[gameRef.currentFloorIndex]);
    },
    child: const Text('Play'),
  ), // ElevatedButton
), // SizedBox
```

Then I used the `playBackgrondMusic()` in the main screen's play button so as to change and update the bgm whenever it is clicked.

*Another possible solution is using `AudioPool`. However, I encountered several errors when working with this, so I substituted to use `FlameAudio.bgm` instead.

Joysticks on Mobile Screen

Step 1. Prepare joystick HUD in png.

Step 2. Add joysticks on the Screen.

`JoystickComponent` is a tool to create a joystick functioning with a knob.

```
void addJoystick() {
  joystick = JoystickComponent(
    priority: 10,
    knob: SpriteComponent(
      sprite: Sprite(
        images.fromCache('HUD/Knob.png'),
      ), // Sprite
    ), // SpriteComponent
    knobRadius: 64,
    background: SpriteComponent(
      sprite: Sprite(
        images.fromCache('HUD/Joystick.png'),
      ), // Sprite
    ), // SpriteComponent
    margin: const EdgeInsets.only(left: 32, bottom: 32),
  ); // JoystickComponent

  add(joystick);
}
```

This will combine the body.png and knob.png. Thus, the final joystick will look like:

Step 3. Make joysticks work.

```
void updateJoystick() {
  switch (joystick.direction) {
    case JoystickDirection.left:
    case JoystickDirection.upLeft:
    case JoystickDirection.downLeft:
      player.horizontalMovement = -1;
      break;
    case JoystickDirection.right:
    case JoystickDirection.upRight:
    case JoystickDirection.downRight:
      player.horizontalMovement = 1;
      break;
    default:
      player.horizontalMovement = 0;
      break;
  }
}
```

`JoystickDirection` makes the knob move on the joystick's body. Along with `joystick.direction`, makes the player horizontally move left or right.

Jump & Attack Buttons on Mobile Screen

Not only joystick, but you can also create jump or attack button that enables the player to jump/attack when it is clicked.

Creating jump/attack button is like the way creating joystick.

Flame provides `HudButtonComponent` that detects tap events, meaning it is possible to set its function when it is pressed or released.

In recycle_adventure.dart:

```
void addJumpButton() {
  jumpButton = HudButtonComponent(
    priority: 10,
    onPressed: () {
      player.hasJumped = true;
    },
    onReleased: () {
      player.hasJumped = false;
    },
    button: SpriteComponent(
      sprite: Sprite(
        images.fromCache('HUD/Jump Button.png'),
      ), // Sprite
      size: Vector2.all(64),
    ), // SpriteComponent
    margin: const EdgeInsets.only(right: 128, bottom: 32),
  ); // HudButtonComponent
  add(jumpButton);
}
```

On pressed, `player.hasJumped` set to be true that makes the player jump.

On released, `player.hasJumped` is set to be false, making the player fall by gravity.

```
@override
FutureOr<void> onLoad() async {
  health = maxHealth;
  if (!_isAlreadyLoaded) {
    await images.loadAllImages();

    _loadFloor();

    if (showControls) {
      addJoystick();
      addJumpButton();
      addAttackButton();
    }
    _isAlreadyLoaded = true;
  }

  return super.onLoad();
}
```

*Be sure to add joystick and buttons on game load.

Feedback & Issues that haven't been fixed.

1. Updating Object's Position

   In the boss, the patterns are continuously triggered in onUpdate() to update the boss' current position and change its velocity. However, this is not a good solution for the patterns that must work with timers. For instance, once pattern 2 is chosen, onPattern2 turns true and triggers _pattern2(). Then _pattern2() keeps triggered in onUpdate to make the boss move while spawning drone enemies; pattern 2 starts droneSpawnManager and stops it after specific seconds. Because _pattern2() is triggered too often onUpdate, droneSpawnManager is malfunctioning and stuck in an infinite loop. I managed to fix this issue by resuming droneSpawnManager only if droneSpawnManager.timer.isRunning is false, but I don't like the idea of triggering _pattern2() over many times because it has so much potential to cause errors. I also set isHitOn true at the end of every pattern to give time for the player to attack the boss, turning isHitOn and onPattern false after 4 seconds. But this doesn't work correctly for any reason, and I still haven't figured out why and fixed it. I assume it is because the patterns are triggered too many times.

2. Making a Hitbox Dynamic

   In the Hammer trap, I tried making its hitbox dynamic by moving its position or resizing along with the sprite animation over time. I used the amount of animation and step time to calculate the speed at which the hitbox should move. However, so many other factors, such as slight delay, were affecting the speed that the hitbox and the sprite animation kept misaligned.

3. Block Collision Error with Player

   The player can walk on and collide with blocks by calculating collisions between them with their position, height, width, and hitbox offset. It works well only when the player collides with blocks on one side; if the player moves or jumps when colliding with blocks horizontally and vertically at the same time, the player's position changes and sends the player to a weird spot. I assume there should be additional calculations to fix this, but another possible solution is applying a hitbox to the blocks and using collision callbacks.

4. Joystick and Button HUD Priority Issue

   Joystick and Jump/Attack HUD buttons are available on mobile platforms. However, the game scene hides them. Although I set their priority highest, it doesn't resolve the issue.

5. Resetting Game Loop

   I've been searching for how to reset the game loop so players can restart it if they die. I thought I could achieve this using the Game Engine, but it doesn't provide such a function.

6. Small Community

   Because the Flutter Flame community is small, it is difficult to find the issue and its solution, and debugging code takes more time.

7. Lack of Game Physic Engine

   I think it would be easier to develop games if Flame provided a game physics engine. I know there is one, Forge2D, but I don't think it can be used to create enemy AI; the enemy detects the player and chases them. I could make enemies detect the player by calculating their detection range, but I'm still figuring out how to make them chase the player. I think Bonfire has this feature.