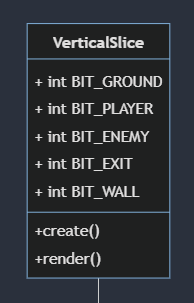
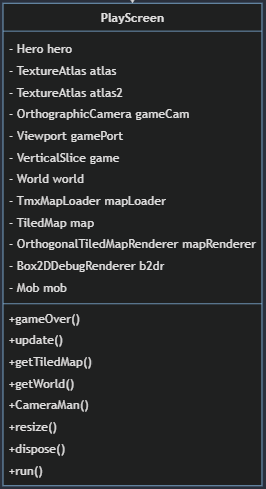
**Hypnotic Beef – High Level Implementation Manual**



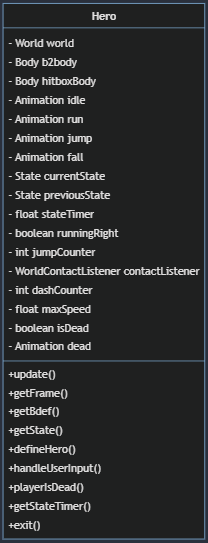
VerticalSlice is a class that defines a vertical slice of a game level. It contains several constant values that represent different types of objects that can exist within the slice, such as ground, players, enemies, exits, and walls. The class also includes methods for creating and rendering the slice. These BITS are a way to determine collision by identifying them through bit numbers. They also known known as ‘layers’ to differentiate game objects.



The PlayScreen class is responsible for managing the game screen where all the components of the game are assembled into one area. It contains several instance variables such as the Hero character, TextureAtlas objects for managing game textures, an OrthographicCamera for rendering the game view, a Viewport for managing the screen size, a VerticalSlice object for managing the game logic, a World object for managing the Box2D physics simulation, a TmxMapLoader for loading game maps, a TiledMap for rendering game maps, an OrthogonalTiledMapRenderer for rendering the maps, a Box2DDebugRenderer for rendering Box2D debug data, and a Mob object for managing enemy characters.

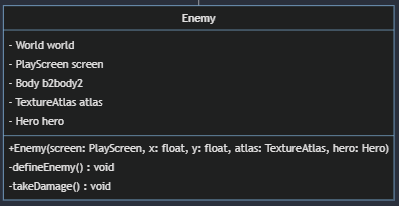
The PlayScreen class also contains several methods such as gameOver() for ending the game when the hero character dies, update() for updating game logic and rendering the game, getTiledMap() for retrieving the current TiledMap, getWorld() for retrieving the current Box2D world, CameraMan() for managing the game camera, resize() for handling screen resizing events, dispose() for freeing up memory when the game is closed, and run() for starting the game loop.

Implementing the PlayScreen class will provide a central location for managing all aspects of the game screen, including game logic, physics simulation, rendering, and camera management.



The Hero class represents the player's character in the game. It has a World object for physics simulation, a Body object for physics collision, and an additional Body object for hitbox collision. It has Animation<TextureRegion> objects for idle, run, jump, fall, and dead states, which are played according to the currentState and previousState of the hero. The stateTimer tracks the duration of the current state. The runningRight boolean indicates whether the hero is moving right or left, and the jumpCounter keeps track of how many times the hero has jumped in the air. The contactListener handles collision events. The maxSpeed determines the maximum speed that the hero can move, and the dashCounter keeps track of the number of times the hero has dashed. The isDead boolean indicates whether the hero is currently dead, and the exit() method is called when the hero reaches the exit of the level.

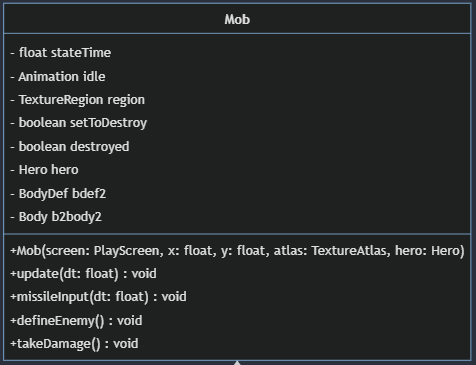
The update() method updates the hero's position, state, and animation based on user input and physics simulation. The getFrame() method returns the current frame of the hero's animation. The getBdef() method returns the BodyDef object used to create the hero's physics body. The getState() method returns the current state of the hero. The defineHero() method creates the hero's physics body and hitbox body. The handleUserInput() method handles user input to move, jump, and dash the hero. The playerIsDead() method is called when the hero dies, which sets the isDead boolean to true and plays the dead animation. The getStateTimer() method returns the stateTimer.



The Enemy class is a superclass of the Mob class and is responsible for creating and controlling enemies in the game. It contains a World object, a PlayScreen object, a Body object, a TextureAtlas object, and a Hero object.

The Enemy class has a constructor that takes in a PlayScreen object, x and y coordinates, a TextureAtlas object, and a Hero object as parameters. It also has a defineEnemy() method that creates the enemy's physical body and sets up its fixture, as well as a takeDamage() method that handles the enemy taking damage.

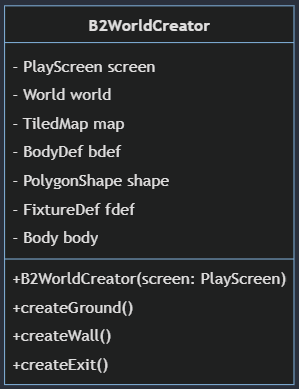
This class can be used as a base for creating different types of enemies in the game, each with their own unique properties and behaviors.



The Mob class is responsible for defining and updating enemies in the game. It has properties such as stateTime to keep track of the time, idle to define the idle animation of the enemy, region to store the current texture region of the enemy, setToDestroy and destroyed to indicate if the enemy has been defeated, hero to reference the main character in the game, bdef2 to define the body of the enemy, and b2body2 to store the physics body of the enemy.

The constructor of Mob takes in a PlayScreen object, x and y coordinates, a TextureAtlas object to define the texture of the enemy, and a Hero object to reference the player character.

The class has several methods, such as update to update the state of the enemy, missileInput to handle the input for the enemy's attacks, defineEnemy to define the enemy's body and fixture, and takeDamage to reduce the enemy's health and destroy it when it reaches zero.



The B2WorldCreator class is responsible for creating the physical boundaries and objects within the game world using Box2D physics engine. It has access to the PlayScreen object, World object, and TiledMap object, as well as various other Box2D related classes.

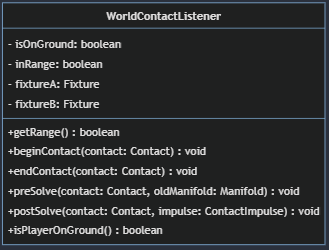
The class has a constructor that takes a PlayScreen object as an argument. It also has private fields for the World, TiledMap, BodyDef, PolygonShape, FixtureDef, and Body objects.

The class contains three methods that create the ground, walls, and exit objects of the game. These methods use the Box2D physics engine to create the physical bodies and fixtures of each object, which are then added to the game world.

The createGround() method creates the ground object, which is a static body with a rectangular shape that is created using the PolygonShape class.

The createWall() method creates the walls of the game world. These walls are also static bodies with rectangular shapes and are created using the PolygonShape class.

The createExit() method creates the exit object of the game. This object is also a static body with a rectangular shape and is created using the PolygonShape class. The method also sets a user data for the exit object, which can be used later to detect when the player has reached the exit.

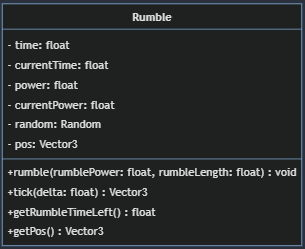
In the grand scheme, this code is responsible for detecting collisions between different fixtures in a physics simulation. It does this by implementing the ContactListener interface provided by the Box2D physics engine.

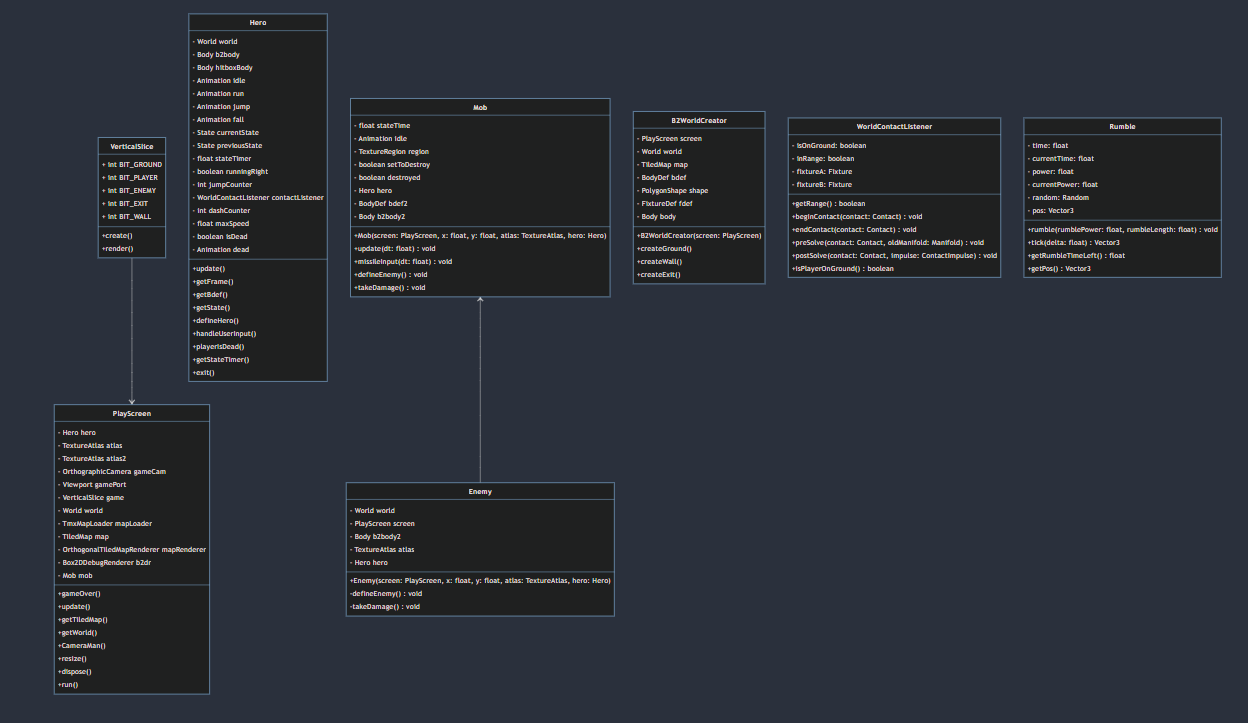
Whenever two fixtures come into contact, the beginContact() method of the ContactListener is called, and the fixtureA and fixtureB objects are set to the two fixtures that made contact.

The method then checks the category bits of the two fixtures to determine what type of collision occurred. Based on the collision type, it performs different actions such as calling the hit() method of the Hero object if it collides with an enemy or updating the isOnGround variable if the player is on the ground.

The endContact() method is called when two fixtures stop touching each other. In this case, the code sets the isOnGround variable to false if the player is no longer touching the ground.

Overall, this code plays an essential role in determining the outcome of collisions in a physics simulation

The Rumble class simulates a camera shake effect by applying a random vibration to the camera's position for a specified duration and intensity. The rumble method initializes the rumble effect with a specified power and length, while the tick method returns the current position of the camera, which is randomly offset based on the rumble parameters. The getRumbleTimeLeft method returns the remaining time of the rumble effect, while the getPos method returns the current position of the camera after the rumble effect has been applied.



UML Diagram Code  
classDiagram

class VerticalSlice {

+ int BIT\_GROUND

+ int BIT\_PLAYER

+ int BIT\_ENEMY

+ int BIT\_EXIT

+ int BIT\_WALL

+ create()

+ render()

}

VerticalSlice --> PlayScreen

class PlayScreen {

- Hero hero

- TextureAtlas atlas

- TextureAtlas atlas2

- OrthographicCamera gameCam

- Viewport gamePort

- VerticalSlice game

- World world

- TmxMapLoader mapLoader

- TiledMap map

- OrthogonalTiledMapRenderer mapRenderer

- Box2DDebugRenderer b2dr

- Mob mob

+ gameOver()

+ update()

+ getTiledMap()

+ getWorld()

+ CameraMan()

+ resize()

+ dispose()

+ run()

}

class Hero {

- World world

- Body b2body

- Body hitboxBody

- Animation<TextureRegion> idle

- Animation<TextureRegion> run

- Animation<TextureRegion> jump

- Animation<TextureRegion> fall

- State currentState

- State previousState

- float stateTimer

- boolean runningRight

- int jumpCounter

- WorldContactListener contactListener

- int dashCounter

- float maxSpeed

- boolean isDead

- Animation<TextureRegion> dead

+ update()

+ getFrame()

+ getBdef()

+ getState()

+ defineHero()

+ handleUserInput()

+ playerIsDead()

+ getStateTimer()

+ exit()

}

class Mob {

- float stateTime

- Animation<TextureRegion> idle

- TextureRegion region

- boolean setToDestroy

- boolean destroyed

- Hero hero

- BodyDef bdef2

- Body b2body2

+ Mob(screen: PlayScreen, x: float, y: float, atlas: TextureAtlas, hero: Hero)

+ update(dt: float): void

+ missileInput(dt: float): void

+ defineEnemy(): void

+ takeDamage(): void

}

Mob <-- Enemy

class Enemy {

- World world

- PlayScreen screen

- Body b2body2

- TextureAtlas atlas

- Hero hero

+ Enemy(screen: PlayScreen, x: float, y: float, atlas: TextureAtlas, hero: Hero)

- defineEnemy(): void

- takeDamage(): void

}

class B2WorldCreator {

- PlayScreen screen

- World world

- TiledMap map

- BodyDef bdef

- PolygonShape shape

- FixtureDef fdef

- Body body

+ B2WorldCreator(screen: PlayScreen)

+ createGround()

+ createWall()

+ createExit()

}

class WorldContactListener {

- isOnGround: boolean

- inRange: boolean

- fixtureA: Fixture

- fixtureB: Fixture

+ getRange(): boolean

+ beginContact(contact: Contact): void

+ endContact(contact: Contact): void

+ preSolve(contact: Contact, oldManifold: Manifold): void

+ postSolve(contact: Contact, impulse: ContactImpulse): void

+ isPlayerOnGround(): boolean

}

class Rumble{

- time: float

- currentTime: float

- power: float

- currentPower: float

- random: Random

- pos: Vector3

+ rumble(rumblePower: float, rumbleLength: float): void

+ tick(delta: float): Vector3

+ getRumbleTimeLeft(): float

+ getPos(): Vector3

}