VIETNAM NATIONAL UNIVERSITY OF HOCHIMINH CITY

THE INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

SELAPHOBIA GAME REPORT

Name	ID	Contribution
Khương Minh Triết	ITITDK22041	100%

Ho Chi Minh City, Vietnam

2022-2023

Table of Contents

1. Introduction

1.1. Topic motivation

- My first game development project involves creating a platformer game inspired by the globally popular Geometry Dash. This genre of games has captivated millions of players with its challenging gameplay that emphasizes overcoming obstacles with precise timing and musical rhythm. The objective is to utilize the elements of platformer games to create an experience for players.

- Geometry Dash, developed by Robert Topala and released in 2013, is known for its rhythmic levels, vibrant visuals and simple control scheme, using just a single key. The game has a large following and many fan-made levels. It offers endless creativity, as players can easily create their own levels using a variety of elements. These contents contribute to the game's longevity, as new levels and challenges keep players coming back for more, ensuring its relevance for years.

- This project provides an opportunity to explore the principles of game development, particularly focusing on object-oriented programming (OOP) using Java. By recreating and building upon the core gameplay elements of Geometry Dash, the aim is to meet the requirements of the university course while providing players with an engaging experience.

1.2. Developer team

Team name: Last man standing

Name	ID	Contribution
Khương Minh Triết https://github.com/triek	ITITDK22041	Report writing Game developing Graphic designing Presentation

1.3. References

Platformer Tutorial - Java by Kaarin Gaming

https://www.youtube.com/playlist?list=PL4rzdwizLaxYmltJQRjq18a9gsSyEQQ-0

2. Software requirements

2.1. Working tools, platforms:

- IntelliJ IDEA: Coding environment
- Adobe Photoshop: Game level map, background graphic
- Adobe Illustrator: Character, orb and block design

2.2. Use case scenario

State	Option	Description
Menu	Play	Entry point for the application. Initializes and starts the game.
	Options	Opens the pause menu, allowing the player to adjust sound settings.
	Quit	Exits the program.
Pause	Continue	Resumes the game from the point where it was paused.
	Replay	Restarts the current level and character from the beginning.
	Menu	Return to the main menu.
GameOver	Replay	Restarts the current level and character from the beginning.
	Menu	Return to the main menu.

Fig 2.2.1.: Use case scenario

3. Game Rule and Game Loop

3.1 Game Rule

- The core gameplay of Selaphobia is simple: The character will automatically move to the left. To finish the level, guide the character through a series of obstacles, reaching the end without crashing.

Basic Controls:

- Left-mouse press: This is the only control, it makes the character jump once every click.
- If the ground is made of blocks, the character can jump if it is on top of the block.

- The blue orb triggers jump action. The character can jump if touches the orbs and execute the action jump.

Obstacles:

- Levels are filled with blocks. If the character collides with one and cannot move further, it will result in game over.

Game buttons:

- Replay/Menu: Reset the level, character state, position and physics.
- Continue/play: Continue the level, character state, position and physics.

3.2 Game Loop

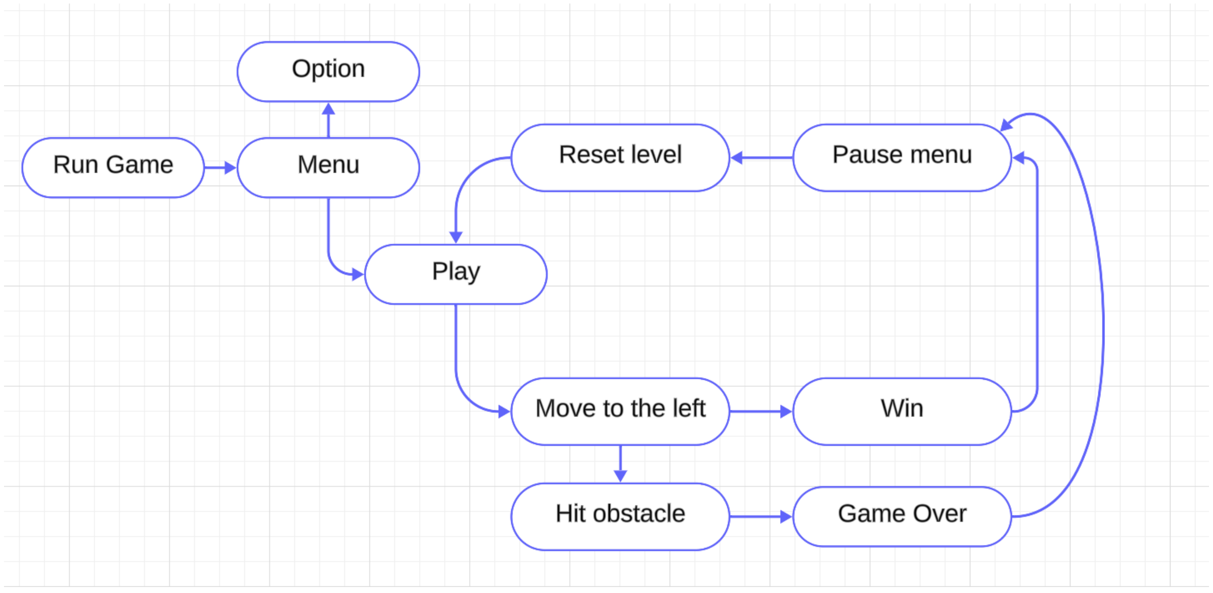


Fig 3.2.1.: Game loop

4. System Design

4.1. List of Classes and their Responsibilities:

Package	Class Name	Description
main	MainClass	Entry point for the application. Initializes and starts the game.
	Game	Manages the main game loop, game states, and game updates.
	GamePanel	Main panel for rendering the game. Listens for keyboard and mouse inputs.
	GameWindow	Manages the game window and handles window focus events.
utilz	Constants	Contains various game constants, such as dimensions, player actions, and UI element sizes.
	LoadSave	Utility class for loading and saving game resources, such as images and level data.
	HelpMethods	Utility class with helper methods for collision detection and movement logic.
gamestates	Gamestate	Enum representing different game states (PLAYING, MENU, OPTIONS, QUIT).
	State	Base class for game states, providing common functionality and utilities for game states.
	Statemethods	Interface defining common methods for game states, such as update, draw, and input handling methods.
	Menu	Represents the main menu state and handles menu-specific logic.
	Playing	Represents the playing state of the game and manages gameplay logic.

levels	LevelManager	Manages the levels, including loading, updating, and rendering level data.
	Level	Represents a single level in the game, containing level-specific data and logic.
	Entity	Abstract base class for all entities in the game, providing common properties and methods for entities.
	Player	Represents the player character and handles player-specific logic, such as movement, jumping, and attacks.
entities	Orb	Abstract base class for orbs, providing common properties and methods for different types of orbs.
	OrbManager	Manages the orbs (boost and reverse) in the game, including updating and rendering them.
	Boost	Represents a boost orb in the game, handling its specific behavior and animations.
	PauseOverlay	Handles the pause menu overlay, including rendering and updating pause menu buttons.
	GameOverOverlay	Handles the game over screen overlay, including rendering and updating game over buttons.
ui	MenuButton	Represents a button in the main menu, including rendering and updating the button state.
	PauseButton	Base class for buttons in the pause overlay, providing common properties and methods for pause menu buttons.
	SoundButton	Extends PauseButton to represent sound control buttons in the pause overlay, handling sound-specific behavior.
	VolumeButton	Extends PauseButton to represent volume control buttons in the

		pause overlay, handling volume-specific behavior.
	UrmButton	Extends PauseButton to represent "Urm" buttons (menu, replay, unpause) in the pause overlay.
input	KeyBoardInputs	Handles keyboard inputs and forwards them to the appropriate game state.
	MouseInputs	Handles mouse inputs and forwards them to the appropriate game state.

Fig 4.1.1.: List of Classes and their Responsibilities

4.2. UML

Class	**Attributes**	**Constructors and Methods**
<<Java Class>> MainClass		+main(args: String[])
<<Java Class>> Game: Implements Runnable	-gameWindow: GameWindow -gamePanel: GamePanel -gameThread: Thread -FPS_SET: int -UPS_SET: int -playing: Playing -menu: Menu	+TILES_DEFAULT_SIZE: int +SCALE: float +TILES_IN_WIDTH: int +TILES_IN_HEIGHT: int +TILES_SIZE: int +GAME_WIDTH: int +GAME_HEIGHT: int +Game() (Constructor) -initClasses() -startGameLoop() +update() +render(g: Graphics) +run() +windowFocusLost()

		+getMenu(): Menu +getPlaying(): Playing
<<Java Class>> GamePanel: Extends JPanel	-mouseInputs: MouseInputs -game: Game	+GamePanel(game: Game) (Constructor) -setPanelSize() +updateGame() @Override +paintComponent(g: Graphics) +getGame(): Game
<<Java Class>> GameWindow	-jframe: JFrame	+GameWindow(gamePanel: GamePanel) (Constructor)
<<Java Class>> Constants	Nesting class	
<<Java Class>> OrbConstants: Nested within Constants	+BOOST: int +REVERSE: int +IDLE: int +HIT: int +BOOST_WIDTH_DEFAULT: int +BOOST_HEIGHT_DEFAULT: int +BOOST_WIDTH: int +BOOST_HEIGHT: int +BOOST_DRAWOFFSET_X: int +BOOST_DRAWOFFSET_Y: int	+GetSpriteAmount(orb_type: int, orb_state: int): int
<<Java Class>> Environment: Nested within Constants	+BIG_CLOUD_WIDTH_DEFAULT: int +BIG_CLOUD_HEIGHT_DEFAULT: int +SMALL_CLOUD_WIDTH_DEFAULT: int +SMALL_CLOUD_HEIGHT_DEFAULT: int +BIG_CLOUD_WIDTH: int	

	+BIG_CLOUD_HEIGHT: int +SMALL_CLOUD_WIDTH: int +SMALL_CLOUD_HEIGHT: int	
<<Java Class>> UI: Nested within Constants	Nesting class	
<<Java Class>> Buttons: Nested within UI	+B_WIDTH_DEFAULT: int +B_HEIGHT_DEFAULT: int +B_WIDTH: int +B_HEIGHT: int	
<<Java Class>> PauseButtons: Nested within UI	+SOUND_SIZE_DEFAULT: int +SOUND_SIZE: int	
<<Java Class>> URMButtons: Nested within UI	+URM_DEFAULT_SIZE: int +URM_SIZE: int	
<<Java Class>> VolumeButtons: Nested within UI	+VOLUME_DEFAULT_WIDTH: int +VOLUME_DEFAULT_HEIGHT: int +SLIDER_DEFAULT_WIDTH: int +VOLUME_WIDTH: int +VOLUME_HEIGHT: int +SLIDER_WIDTH: int	
<<Java Class>> Directions: Nested within	+LEFT: int +UP: int +RIGHT: int	

Constants	+DOWN: int	
<<Java Class>> PlayerConstants : Nested within Constants	+IDLE: int +RUNNING: int +JUMPING: int +DASHING: int +FALLING: int +DYING: int	+GetSpriteAmount(player_action: int): int
<<Java Class>> LoadSave	+PLAYER_ATLAS: String +LEVEL_ATLAS: String +LEVEL_ONE_DATA: String +MENU_BUTTONS: String +MENU_BACKGROUND: String +PAUSE_BACKGROUND: String +SOUND_BUTTONS: String +URM_BUTTONS: String +VOLUME_BUTTONS: String +MENU_BACKGROUND_IMG: String +PLAYING_BG_IMG: String +BIG_CLOUDS: String +SMALL_CLOUDS: String +BOOST_SPRITE: String +REVERSE_SPRITE: String +DEATH_SCREEN: String	+GetSpriteAtlas(fileName: String): BufferedImage +GetBoost(): ArrayList<Boost> +GetReverse(): ArrayList<Reverse> +GetLevelData(): int[][]
<<Java Class>> HelpMethods		+CanMoveHere(): boolean -IsSolid(): boolean +GetEntityXPosNextToWall(): float +GetEntityYPosUnderRoofOrAboveFloor (): float +IsEntityOnFloor(): boolean
<<Java	PLAYING	

Enum>> Gamestate	MENU OPTIONS QUIT +state: Gamestate	
<<Java Class>> State	-game: Game	+State(game: Game) +isIn(e: MouseEvent, mb: MenuButton): boolean +getGame(): Game
<<Java Interface>> Statemethods		+update(): void +draw(g: Graphics): void +mouseClicked(e: MouseEvent): void +mousePressed(e: MouseEvent): void +mouseReleased(e: MouseEvent): void +mouseMoved(e: MouseEvent): void +keyPressed(e: KeyEvent): void +keyReleased(e: KeyEvent): void

<<Java Class>>
Menu:
Implements
Statemethods
	-buttons: MenuButton[]
	-backgroundImg: BufferedImage (Private)
	-backgroundImgPink: BufferedImage
	-menuX: int (Private)
	-menuY: int (Private)
	-menuWidth: int (Private)
	-menuHeight: int (Private)
		+Menu(game: Game)
		-loadButtons(): void (Private)
		-loadBackground(): void (Private) (Commented out)
		+update(): void (Override)
		+draw(g: Graphics): void (Override)
		+mouseClicked(e: MouseEvent): void (Override) (Empty)
		+mousePressed(e: MouseEvent): void (Override)
		+mouseReleased(MouseEvent e): void (Override)
		-resetButtons(): void (Private)
		+mouseMoved(e: MouseEvent): void (Override)
		+keyPressed(e: KeyEvent): void (Override)

		+keyReleased(KeyEvent e): void (Override) (Empty)
<<Java Class>> Playing: extends State: implements	-player: Player -levelManager: LevelManager -orbManager: OrbManager -pauseOverlay: PauseOverlay -gameOverOverlay: GameOverOverlay -paused: boolean (Private) -xLvlOffset: int (Private) -leftBorder: int (Private) -rightBorder: int (Private) -lvlTilesWide: int (Private) -maxTilesOffset: int (Private) -maxLvlOffsetX: int (Private) -backgroundImg: BufferedImage (Private) -bigCloud: BufferedImage (Private) -smallCloud: BufferedImage (Private) -smallCloudsPos: int[] (Private) -rnd: Random (Private) -gameOver: boolean (Private) -playerDying: boolean (Private)	+Playing(game: Game) -initClasses(): void (Private) +update(): void (Override) -checkCloseToBorder(): void (Private) +draw(Graphics g): void (Override) -drawClouds(Graphics g): void (Private) +resetAll(): void +checkBoostHit(Rectangle2D.Float attackBox): void +checkReverseHit(Rectangle2D.Float attackBox): void +setGameOver(boolean gameOver): void +mouseClicked(MouseEvent e): void (Override) (Empty) +mouseDragged(MouseEvent e): void +mousePressed(MouseEvent e): void (Override) +mouseReleased(MouseEvent e): void (Override) +mouseMoved(MouseEvent e): void (Override) +keyPressed(KeyEvent e): void (Override) +keyReleased(KeyEvent e): void (Override) +unpauseGame(): void +windowFocusLost(): void +getPlayer(): Player +setPlayerDying(boolean playerDying): void
<<Java Class>>	-game: Game (Private)	+LevelManager(game: Game)

LevelManager	-levelSprite: BufferedImage[] (Private) -levelOne: Level (Private)	-importOutsideSprites(): void (Private) +draw(Graphics g, int lvlOffset): void +update(): void (Empty) +getCurrentLevel(): Level
<<Java Class>> Level	-lvlData: int[][] (Private)	+Level(lvlData: int[][]) +getSpriteIndex(x: int, y: int): int +getLevelData(): int[][]
<<Java Abstract Class>> Entity	- x: float (Protected) - y: float (Protected) - width: int (Protected) - height: int (Protected) - hitbox: Rectangle2D.Float (Protected)	+ Entity(float x, float y, int width, int height) - drawHitbox(Graphics g, int xLvlOffset): void (Protected) - initHitbox(float x, float y, int width, int height): void (Protected) + getHitbox(): Rectangle2D.Float
<<Java Class>> Player: Extends Entity	- animations: BufferedImage[][] (Private) - aniTick: int (Private) - aniIndex: int (Private) - aniSpeed: int (Private) - playerAction: int (Private) - moving: boolean (Private) - attacking: boolean (Private) - dying: boolean (Private) - left: boolean (Private) - right: boolean (Private) - jump: boolean (Private) - grav: boolean (Private) (unused in current code) - playerSpeed: float (Private) - lvlData: int[][] (Private) - xDrawOffset: float (Private) - yDrawOffset: float (Private)	+ Player(float x, float y, int width, int height, Playing playing, OrbManager orbManager) - initAttackBox(): void (Private) + update(): void - checkBoostAttack(): void (Private) - checkReverseAttack(): void (Private) - updateAttackBox(): void (Private) + render(Graphics g, int lvlOffset): void - drawAttackBox(Graphics g, int lvlOffsetX): void (Private) (Not called) - updateAnimationTick(): void (Private) - setAnimation(): void (Private) - resetAniTick(): void (Private) - updatePos(): void (Private) - jump(): void (Private) - changeGrav(): void (Private) - resetInAir(): void (Private)

	- airSpeed: float (Private) - gravity: float (Private) - jumpSpeed: float (Private) - fallSpeedAfterCollision: float (Private) - inAir: boolean (Private) - reverseGrav: boolean (Private) - attackBox: Rectangle2D.Float (Private) - flipX: int (Private) - flipW: int (Private) - state: int (Private) (unused in current code) - attackChecked: boolean (Private) - playing: Playing (Private) - orbManager: OrbManager (Private)	- updateXPos(float xSpeed): void (Private) - loadAnimations(): void (Private) + loadLvlData(int[][] lvlData): void + resetDirBooleans(): void + setAttacking(boolean attacking): void + isLeft(): boolean + setLeft(boolean left): void + isRight(): boolean + setRight(boolean right): void + setJump(boolean jump): void + setChangeGravity(boolean grav): void (Unused) + resetAll(): void
<<Java Abstract Class>> Orb: Extends Entity	# aniIndex: int # orbState: int # orbType: int # aniTick: int # aniSpeed: int = 25 # firstUpdate: boolean = true # inAir: boolean	+ Orb(float x, float y, int width, int height, int orbType) # firstUpdateCheck(lvlData: int[][]): void # updateInAir(lvlData: int[][]): void # move(lvlData: int[][]): void # newState(orbState: int): void + hit(): void # updateAnimationTick(): void + resetOrb(): void + getAniIndex(): int + getOrbState(): int
<<Java Class>> OrbManager	- playing: Playing (Private) - boostArr: BufferedImage[][] (Private) - reverseArr: BufferedImage[][] (Private)	+ OrbManager(Playing playing) - addOrbs(): void (Private) + update(int[][] lvlData): void + draw(Graphics g, int xLvlOffset): void - drawBoosts(Graphics g, int xLvlOffset):

	- boosts: ArrayList<Boost> (Private) - reverses: ArrayList<Reverse> (Private)	void (Private) - drawReverses(Graphics g, int xLvlOffset): void (Private) + checkBoostHit(Rectangle2D.Float attackBox): void + checkReverseHit(Rectangle2D.Float attackBox): void + isBoostHit(Rectangle2D.Float attackBox): boolean + isReverseHit(Rectangle2D.Float attackBox): boolean - loadOrbImgs(): void (Private) + resetAllOrbs(): void

<<Java Class>>
Boost extends Orb
		+ Boost(float x, float y): void
		+ update(int[][] lvlData): void (Override)
		# updateMove(int[][] lvlData): void (Protected)
		# updateBehavior(int[][] lvlData, Player player): void (Protected)
		- updateAnimationTick(): void (Inherited from Orb) (Protected, not shown)

<<Java Class>> PauseOverlay	- playing: Playing (Private) - backgroundImg: BufferedImage (Private) - bgX, bgY, bgW, bgH: int (Private) - musicButton, sfxButton: SoundButton (Private) - menuB, replayB, unpauseB: UrmButton (Private) - volumeButton: VolumeButton (Private)	+ PauseOverlay(Playing playing) - loadBackground(): void (Private) - createSoundButtons(): void (Private) - createUrmButtons(): void (Private) - createVolumeButton(): void (Private) + update(): void + draw(Graphics g): void + mouseDragged(MouseEvent e): void + mousePressed(MouseEvent e): void + mouseReleased(MouseEvent e): void + mouseMoved(MouseEvent e): void - isIn(MouseEvent e, PauseButton b): boolean (Private)

<<Java Class>> GameOverOver lay	- playing: Playing (Private) - img: BufferedImage (Private) - imgX, imgY, imgW, imgH: int (Private) - menu, play: UrmButton (Private)	+ GameOverOverlay(Playing playing) - createImg(): void (Private) - createButtons(): void (Private) + draw(Graphics g): void + update(): void - isIn(UrmButton b, MouseEvent e): boolean (Private) + mouseMoved(MouseEvent e): void + mouseReleased(MouseEvent e): void + mousePressed(MouseEvent e): void + keyPressed(KeyEvent e): void
<<Java Class>> MenuButton	- xPos : int - yPos : int - rowIndex : int - xOffsetCenter : int - state : Gamestate - imgs : BufferedImage[] - currentState : ButtonState - bounds : Rectangle	- listener : ButtonClickListener + MenuButton(xPos : int, yPos : int, rowIndex : int, state : Gamestate, listener : ButtonClickListener) - initBounds() - loadImgs() + draw(g : Graphics) + update() + setMouseOver(mouseOver : boolean) + setMousePressed(mousePressed : boolean) + getBounds() : Rectangle + buttonClicked() + resetBools()
<<Java Class>> PauseButton	- x : int - y : int - width : int - height : int - bounds : Rectangle - mouseOver : boolean - mousePressed : boolean	+ PauseButton(x : int, y : int, width : int, height : int) - createBounds() + draw(g : Graphics) + update() + getX() : int + setX(x : int) + getY() : int + setY(y : int)

		+ getWidth() : int
		+ setWidth(width : int)
		+ getHeight() : int
		+ setHeight(height : int)
		+ getBounds() : Rectangle
		+ isMouseOver() : boolean
		+ setMouseOver(mouseOver : boolean)
		+ isMousePressed() : boolean
		+ setMousePressed(mousePressed : boolean)
		+ resetBools()

4.3. Program breakdown

1. Game Initialization and Main Loop:

- `MainClass` starts the game by creating a `Game` instance and running its `main` method.
- `Game` implements `Runnable`, meaning it can be run in a separate thread. This is crucial for smooth animation and game logic.
- The `Game` constructor initializes core components:
    - `GameWindow`: Creates the game window (using `JFrame`).
    - `GamePanel`: The drawing surface (extends `JPanel`).
    - `Playing`: The game's main gameplay logic.
    - `Menu`: The main menu.
- `Game` starts a game loop in a separate thread using `startGameLoop()`. This loop continuously:
    - `update()`: Updates game logic (player movement, enemy AI, etc.).

    - `render(g)`: Draws the game on the `GamePanel`.
- `GamePanel`'s `paintComponent(g)` method is called by the system to redraw the panel. It calls `game.render(g)` to draw the game.

2. Game States:

- `Gamestate` enum defines the possible game states (PLAYING, MENU, OPTIONS, QUIT).
- The `State` abstract class provides a base for game states like `Menu` and `Playing`.
- `Menu` handles the main menu logic (button clicks, drawing the menu).
- `Pause` state while playing, if the player triggers the pause action
- `Playing` handles the actual gameplay, including player movement, level loading, and enemy interactions.

3. Player and Level:

- `LevelManager` manages loading and drawing the game levels. It uses `Level` objects to store level data.
- `Player` extends `Entity` (which provides basic properties like position and hitbox) and handles player-specific logic (movement, animations, attacks).
- `HelpMethods` provides utility functions for collision detection and other game logic.

4. Entities and Objects:

- `Entity` is an abstract class that provides a base for all game objects that have a position, size, and hitbox (collision area).

- `Orb` (and its subclasses `Boost` and `Reverse`) are game objects that the player can interact with.
- `OrbManager` manages the orbs in the game.

5. UI Elements:

- `MenuButton`, `PauseButton`, `SoundButton`, `UrmButton`, and `VolumeButton` are UI elements used in the menus and overlays.
- `PauseOverlay` and `GameOverOverlay` are overlays that appear when the game is paused or over, respectively.

6. Input Handling:

- `MouseInputs` (in `GamePanel`) handles mouse input.
- `State` and its subclasses handle mouse and keyboard events.

7. Constants and Data Loading:

- `Constants` and its nested classes (`OrbConstants`, `Environment`, `UI`, `Directions`, `PlayerConstants`) define various constants used in the game.
- `LoadSave` handles loading game assets (images, level data).

Flow of Execution (Example: Starting the Game):

1. `MainClass.main()` creates a `Game` instance.
2. `Game` creates `GameWindow`, `GamePanel`, `Menu`, and `Playing`.
3. `Game` starts the game loop.
4. The game starts in the `MENU` state.
5. `Menu` draws the menu and handles user input.

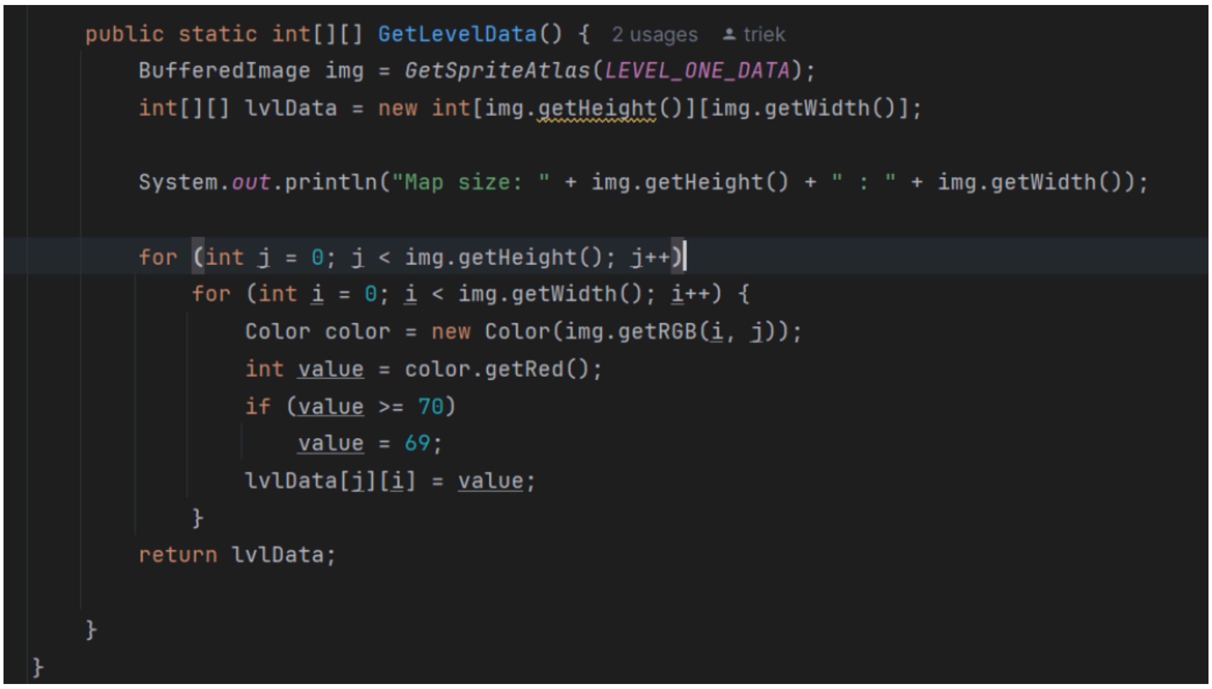
6. If the player clicks the "Play" button, the game state changes to `PLAYING`.
7. `Playing` takes over, loading the level, creating the player, and starting the gameplay.
8. The game loop continues, updating and rendering the game in the `PLAYING` state.

Key Relationships:

- `Game` owns `GameWindow`, `GamePanel`, `Menu`, and `Playing`.
- `GamePanel` has a reference to `Game`.
- `Playing` has references to `Player`, `LevelManager`, `OrbManager`, `PauseOverlay`, and `GameOverOverlay`.
- `LevelManager` has a reference to `Game` and uses `Level` objects.
- `Player` extends `Entity`.
- `Orb` extends `Entity`.
- `OrbManager` manages `Boost` and `Reverse` objects.
- `Menu` and `Playing` implement `Statemethods`.

4.4. Game methods

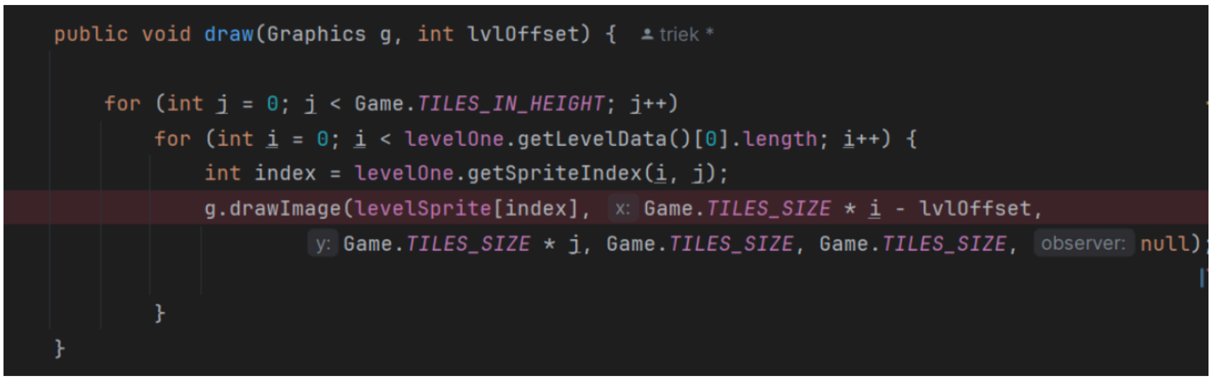
1. Draw map method



```
    public static int[][] GetLevelData() {  2 usages  triek
        BufferedImage img = GetSpriteAtlas(LEVEL_ONE_DATA);
        int[][] lvlData = new int[img.getHeight()][img.getWidth()];

        System.out.println("Map size: " + img.getHeight() + " : " + img.getWidth());

        for (int j = 0; j < img.getHeight(); j++)
            for (int i = 0; i < img.getWidth(); i++) {
                Color color = new Color(img.getRGB(i, j));
                int value = color.getRed();
                if (value >= 70)
                    value = 69;
                lvlData[j][i] = value;
            }
        return lvlData;

    }
}
```

Fig 4.4.1.1.: GetLevelData method

This method will get the level data such as size, RGB color codes of the pixels.



```
public void draw(Graphics g, int lvlOffset) {  triek *

    for (int j = 0; j < Game.TILES_IN_HEIGHT; j++)
        for (int i = 0; i < levelOne.getLevelData()[0].length; i++) {
            int index = levelOne.getSpriteIndex(i, j);
            g.drawImage(levelSprite[index],  x: Game.TILES_SIZE * i - lvlOffset,
                     y: Game.TILES_SIZE * j, Game.TILES_SIZE, Game.TILES_SIZE,  observer: null);

        }
}
```

Fig 4.4.1.2.: draw method

This method will draw the map tiles by tiles using the data from the GetLevelData method

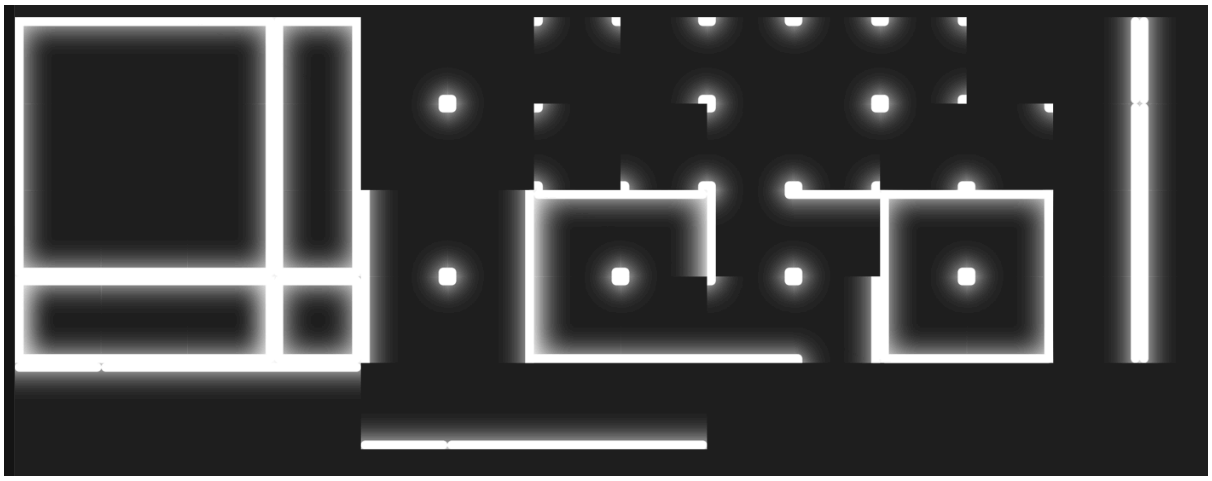


Fig 4.4.1.3.: Blocks graphics

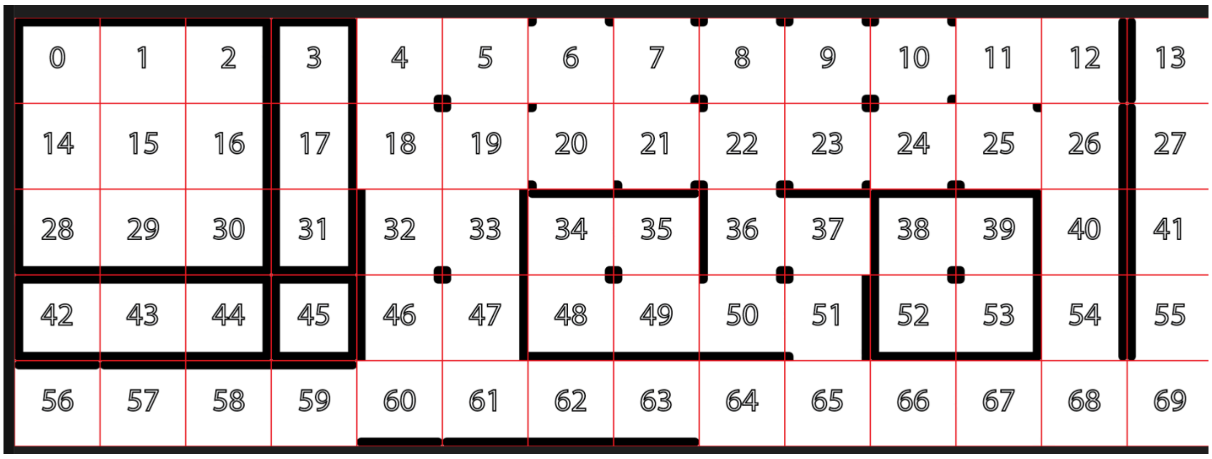


Fig 4.4.1.4.: Block drawing guide

Each tile is specified by the value of Red color. For instance, the tile 10 has a value of R:10, G:any, B:any with "any" being any number from 0-255.

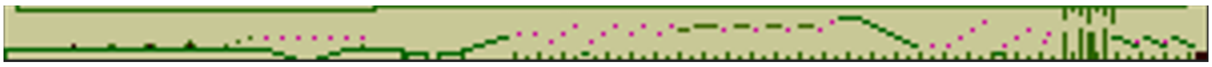


Fig 4.4.1.1.: Game level design

Each pixel in this map will have the color value corresponding to its block.

2. Orb method

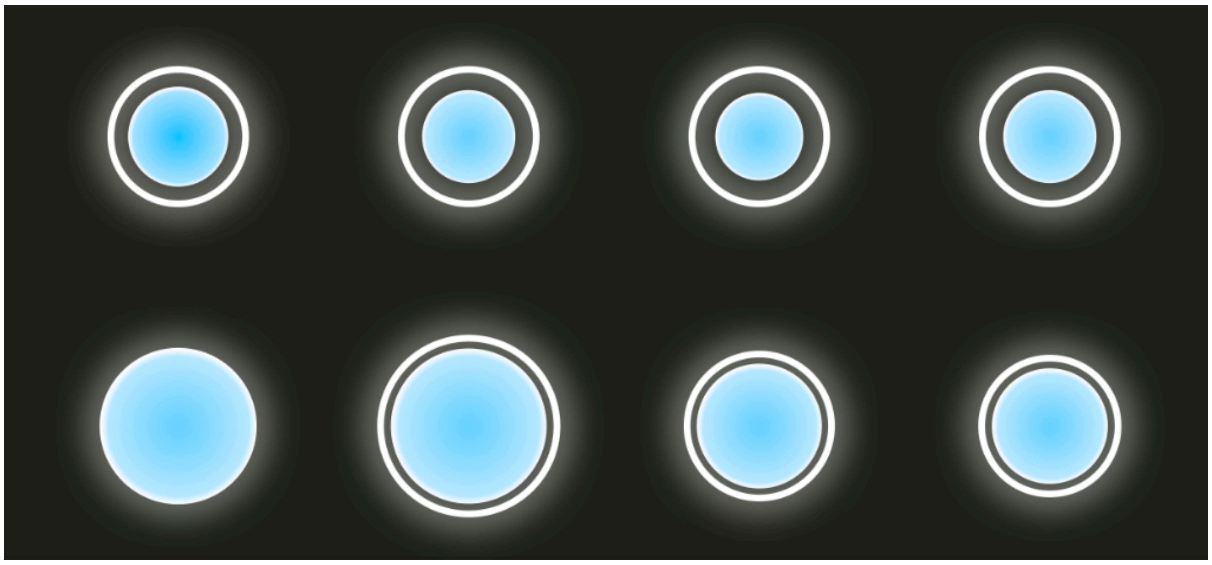
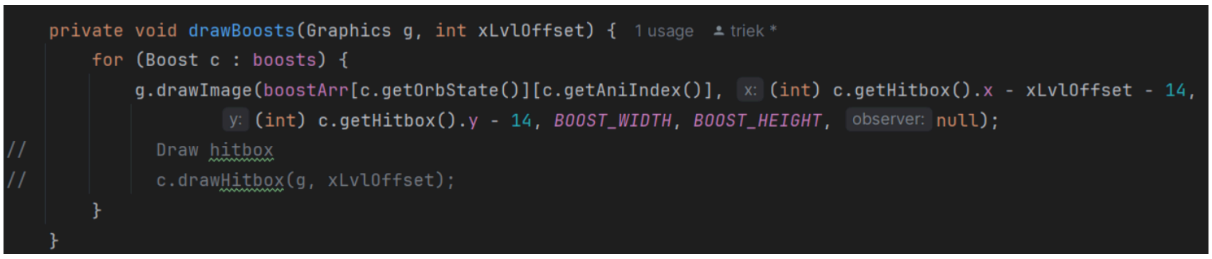


Fig 4.4.2.1.: Boost orb animation



```
    private void drawBoosts(Graphics g, int xLvlOffset) {  1 usage  triek *
        for (Boost c : boosts) {
            g.drawImage(boostArr[c.getOrbState()][c.getAniIndex()], x: (int) c.getHitbox().x - xLvlOffset - 14,
                    y: (int) c.getHitbox().y - 14, BOOST_WIDTH, BOOST_HEIGHT, observer: null);
//              Draw hitbox
//              c.drawHitbox(g, xLvlOffset);
        }
    }
```

Fig 4.4.2.2.: Draw orb method

3. Character method

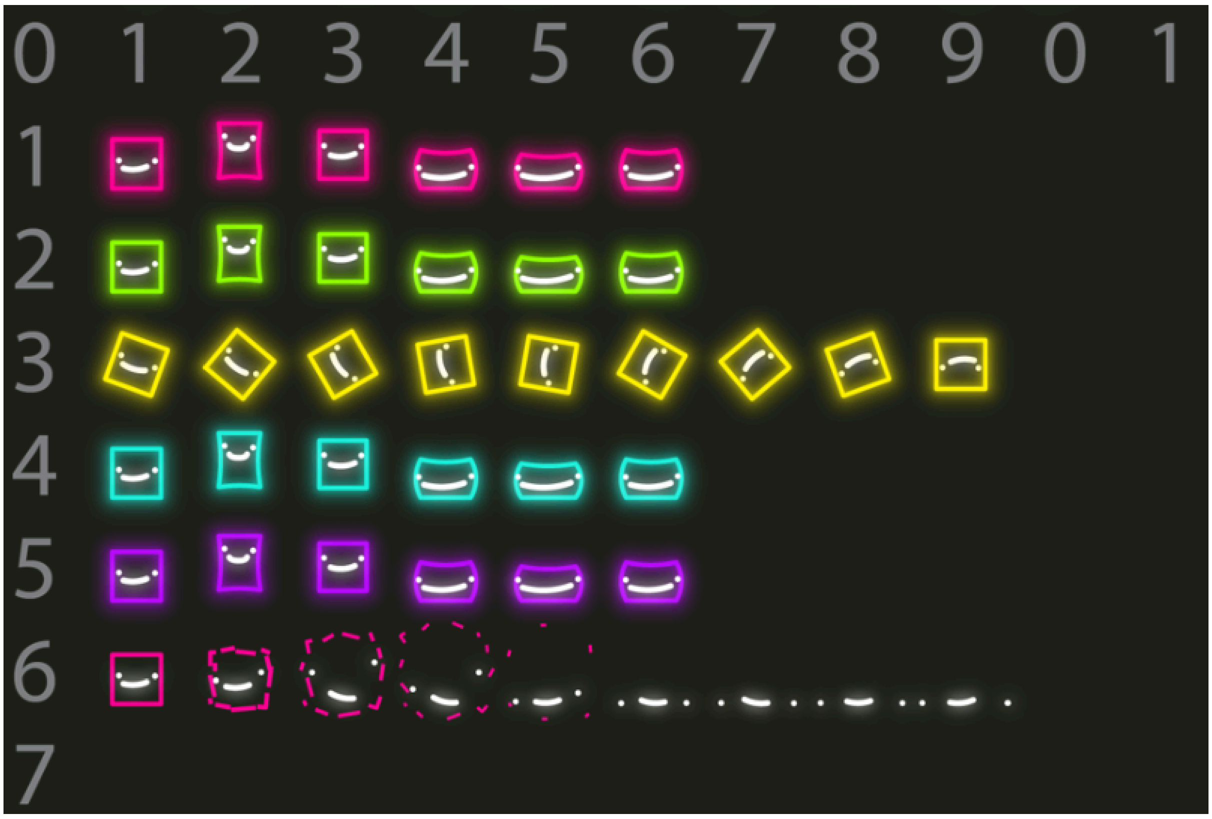
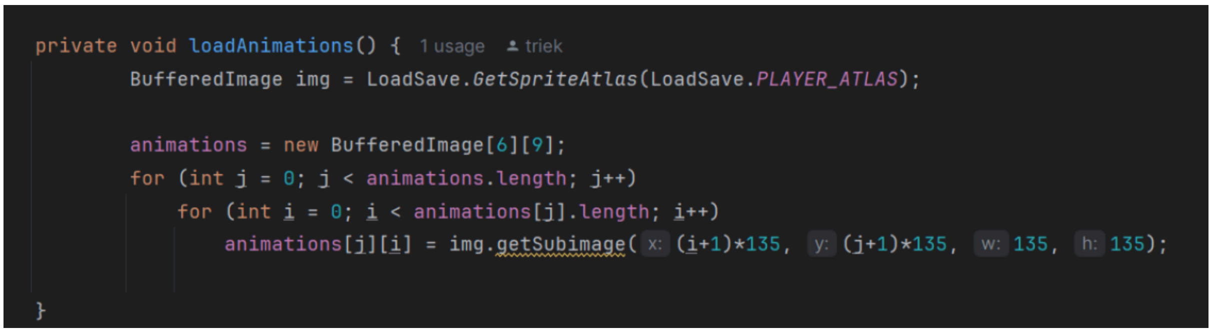


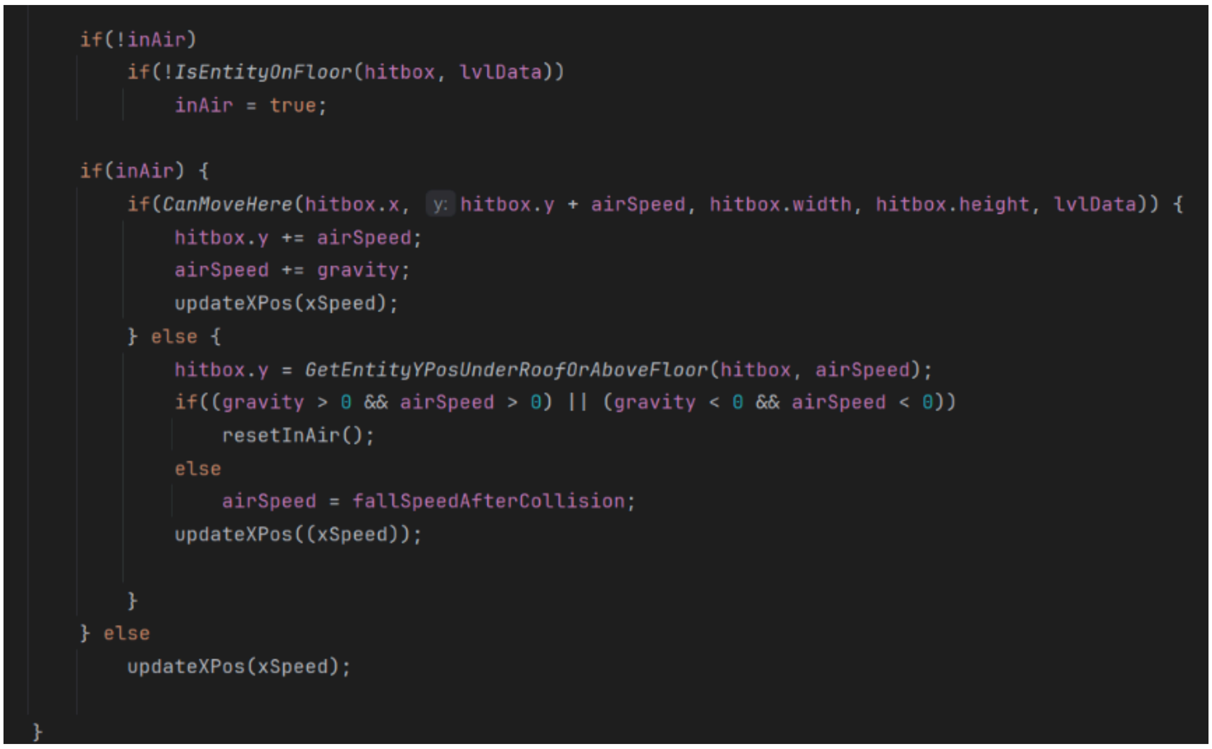
Fig 4.4.3.1.: Character animation



```
private void loadAnimations() {  1 usage  triek
        BufferedImage img = LoadSave.GetSpriteAtlas(LoadSave.PLAYER_ATLAS);

        animations = new BufferedImage[6][9];
        for (int j = 0; j < animations.length; j++)
            for (int i = 0; i < animations[j].length; i++)
                animations[j][i] = img.getSubimage( x: (i+1)*135,  y: (j+1)*135,  w: 135,  h: 135);

}
```

Fig 4.4.3.2.: Draw character method

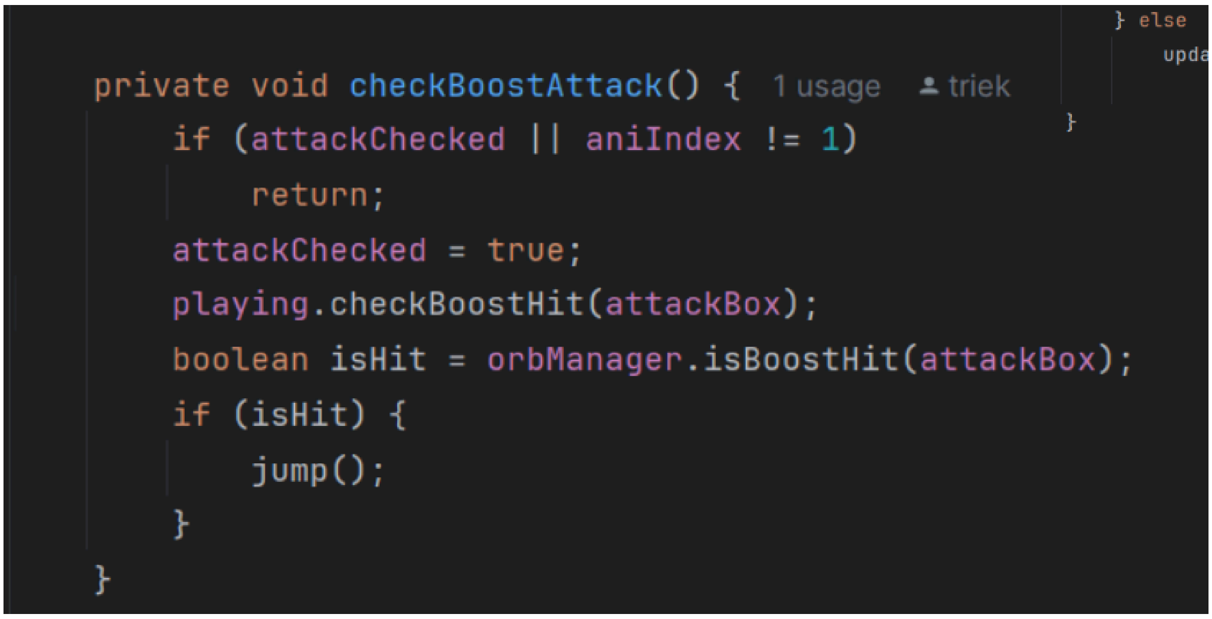
4. Physics methods



```
if(!inAir)
    if(!IsEntityOnFloor(hitbox, lvlData))
        inAir = true;

if(inAir) {
    if(CanMoveHere(hitbox.x, y: hitbox.y + airSpeed, hitbox.width, hitbox.height, lvlData)) {
        hitbox.y += airSpeed;
        airSpeed += gravity;
        updateXPos(xSpeed);
    } else {
        hitbox.y = GetEntityYPosUnderRoofOrAboveFloor(hitbox, airSpeed);
        if((gravity > 0 && airSpeed > 0) || (gravity < 0 && airSpeed < 0))
            resetInAir();
        else
            airSpeed = fallSpeedAfterCollision;
        updateXPos((xSpeed));

    }
} else
    updateXPos(xSpeed);

}
```

Fig 4.4.4.1.: Check if in air

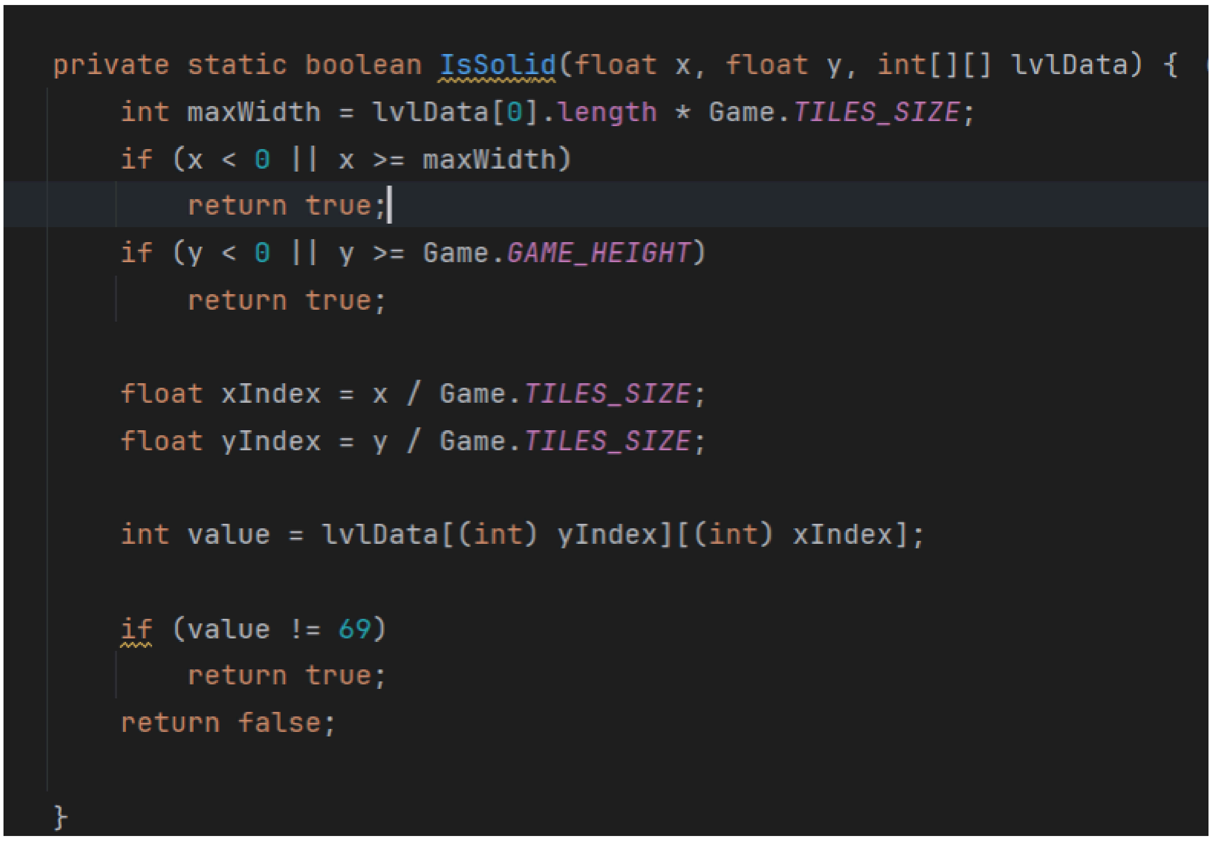
This method will check if the character is in the air to prevent jumping while in air. It will be true when the character is not on the floor or immediately set to true when the character jumps.



```
private void checkBoostAttack() {  1 usage  triek
    if (attackChecked || aniIndex != 1)
        return;
    attackChecked = true;
    playing.checkBoostHit(attackBox);
    boolean isHit = orbManager.isBoostHit(attackBox);
    if (isHit) {
        jump();
    }
}
```

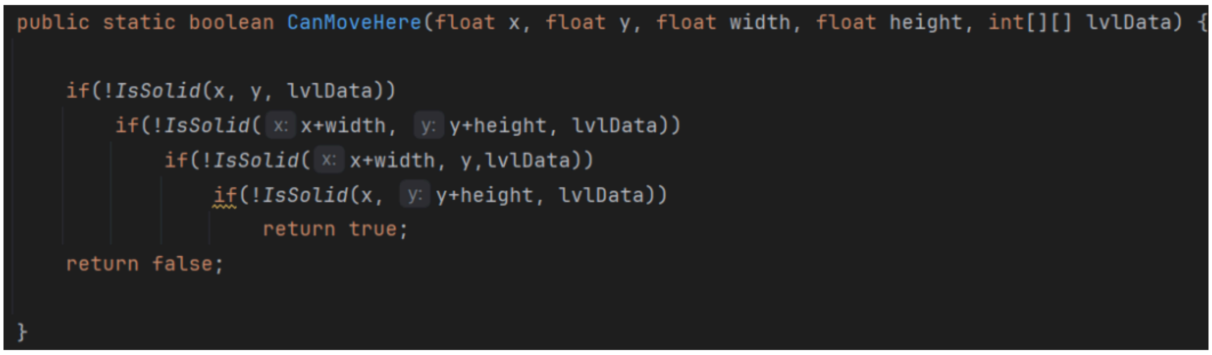
Fig 4.4.4.2.: Boost orb animation

This will check if the player executes attack action, which is also the jump button, while touching the orb. If it is true then the player jumps.
It will ignore the fact that the character is in the air.



```java
private static boolean IsSolid(float x, float y, int[][] lvlData) {
    int maxWidth = lvlData[0].length * Game.TILES_SIZE;
    if (x < 0 || x >= maxWidth)
        return true;
    if (y < 0 || y >= Game.GAME_HEIGHT)
        return true;

    float xIndex = x / Game.TILES_SIZE;
    float yIndex = y / Game.TILES_SIZE;

    int value = lvlData[(int) yIndex][(int) xIndex];

    if (value != 69)
        return true;
    return false;

}
```

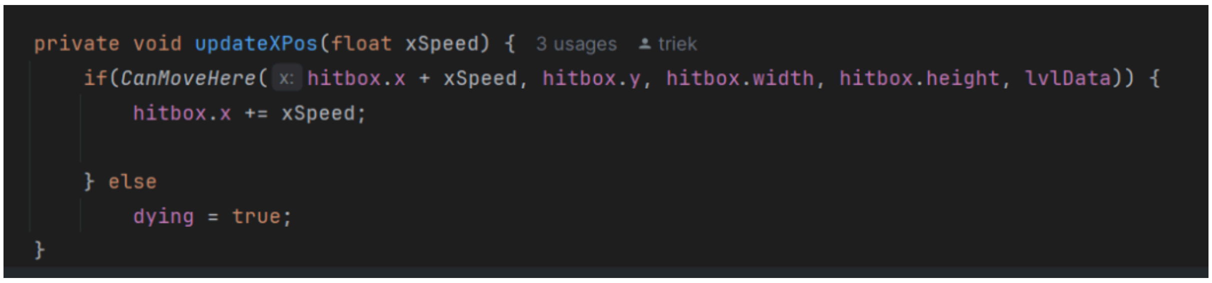
Fig 4.4.4.3.: Check if the ground is solid method



```java
public static boolean CanMoveHere(float x, float y, float width, float height, int[][] lvlData) {

    if(!IsSolid(x, y, lvlData))
        if(!IsSolid( x: x+width,  y: y+height, lvlData))
            if(!IsSolid( x: x+width, y,lvlData))
                if(!IsSolid(x,  y: y+height, lvlData))
                    return true;
    return false;

}
```

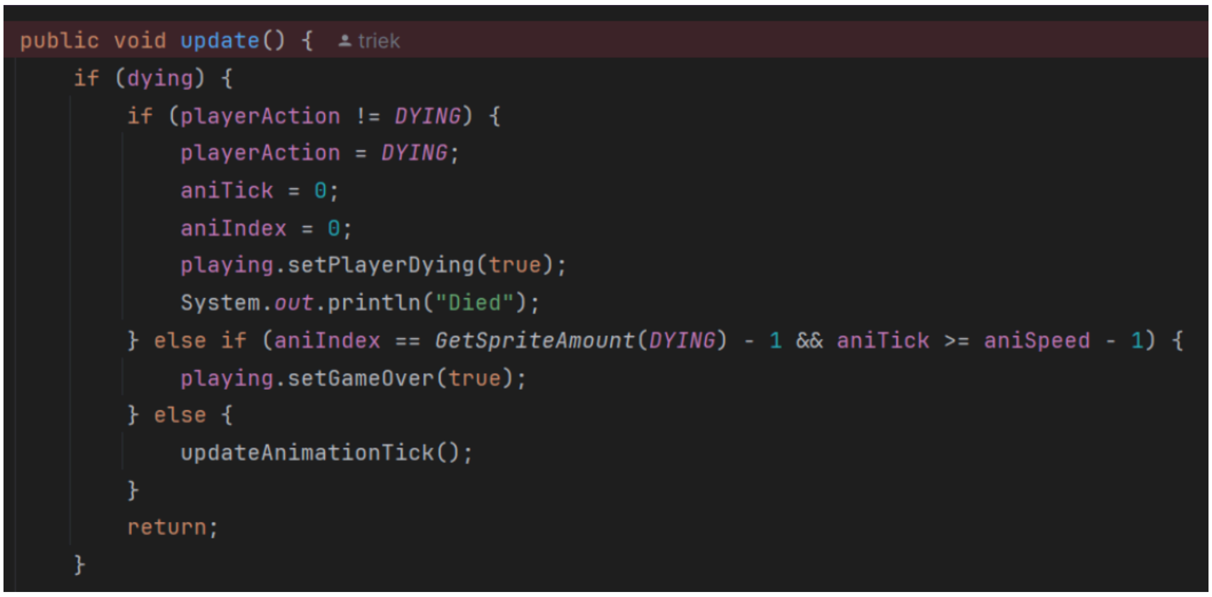
Fig 4.4.4.4.: Check if the player can move method

5. Game over methods



```
private void updateXPos(float xSpeed) {  3 usages  triek
    if(CanMoveHere( x: hitbox.x + xSpeed, hitbox.y, hitbox.width, hitbox.height, lvlData)) {
        hitbox.x += xSpeed;

    } else
        dying = true;
}
```

Fig 4.4.5.1.: If the player can not move then game over



```
public void update() {  triek
    if (dying) {
        if (playerAction != DYING) {
            playerAction = DYING;
            aniTick = 0;
            aniIndex = 0;
            playing.setPlayerDying(true);
            System.out.println("Died");
        } else if (aniIndex == GetSpriteAmount(DYING) - 1 && aniTick >= aniSpeed - 1) {
            playing.setGameOver(true);
        } else {
            updateAnimationTick();
        }
        return;
    }
```

Fig 4.4.5.2.: Methods when the game is over

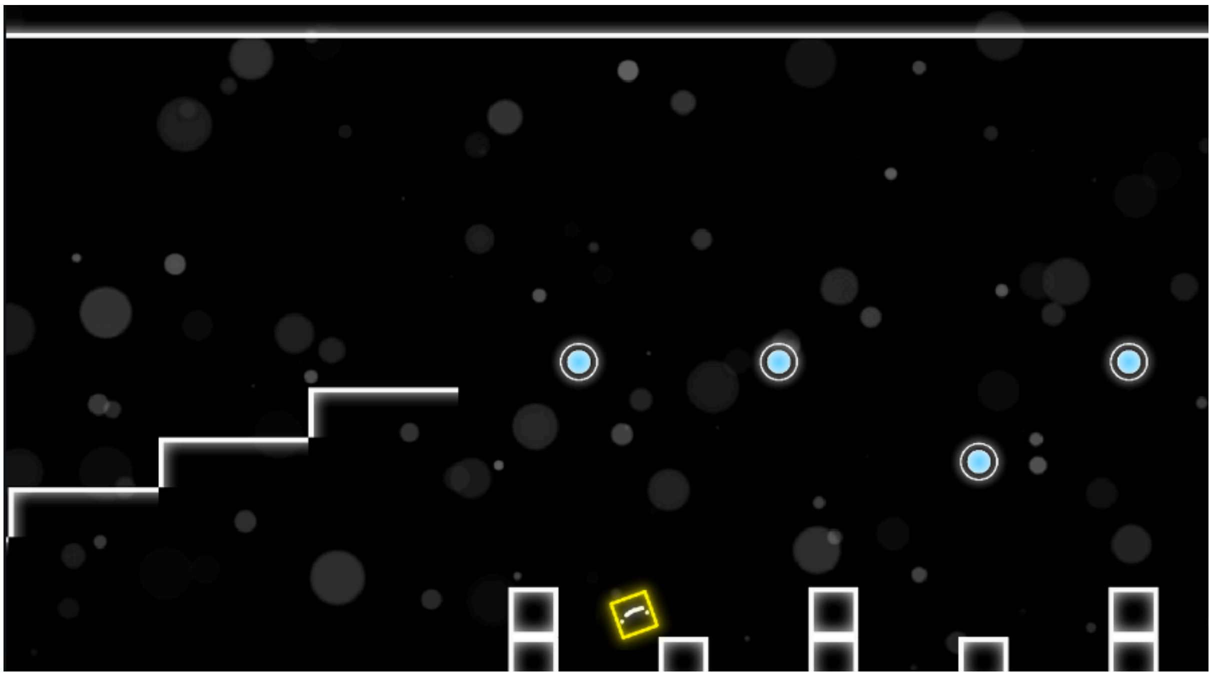


Fig 4.4.5.3.: Game over situation

In this situation, the player could have jumped if they pressed jump when touched the orb, but they missed and fell to the blocks below. Because the player approached the blocks from the left side, they could not move further to the left, therefore they would be game over.

6. Button methods

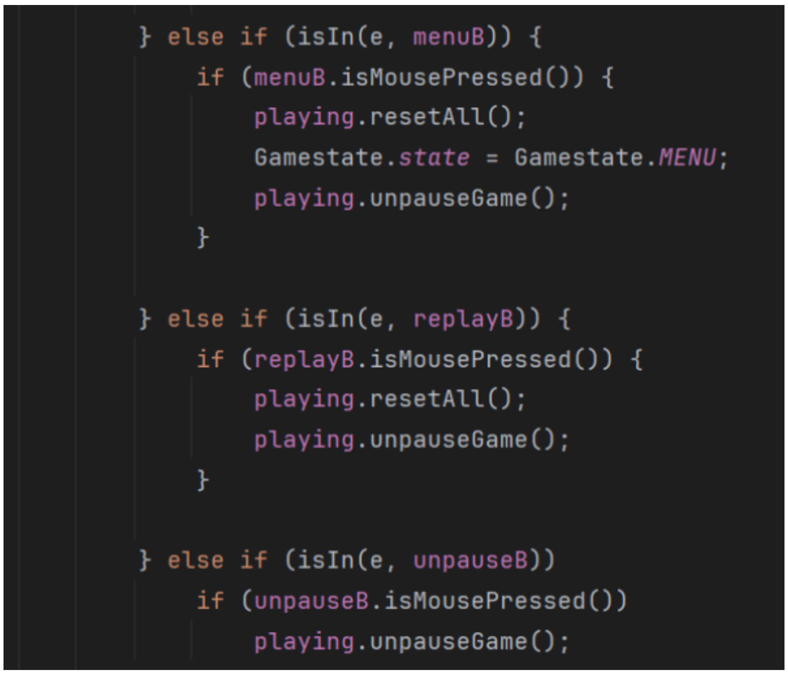

```
} else if (isIn(e, menuB)) {
    if (menuB.isMousePressed()) {
        playing.resetAll();
        Gamestate.state = Gamestate.MENU;
        playing.unpauseGame();
    }

} else if (isIn(e, replayB)) {
    if (replayB.isMousePressed()) {
        playing.resetAll();
        playing.unpauseGame();
    }

} else if (isIn(e, unpauseB))
    if (unpauseB.isMousePressed())
        playing.unpauseGame();
```

Fig 4.4.6.1.: Menu, restart and continue button methods

5. Conclusion

In this game, we have implemented a range of features, including a system for handling game states (MENU, PLAYING), player and level management, entity interactions (player, orbs), UI elements (buttons, overlays), and asset loading. We have also implemented collision detection using hitboxes and basic animation through sprite sheet handling. The game utilizes an object-oriented approach, with classes representing key game elements such as the player, levels, and UI components. This allows for clear separation of concerns and facilitates interactions between different parts of the game.

This project was primarily developed to enhance our understanding of object-oriented programming principles. We made a conscious effort to apply SOLID principles throughout the development process. While we acknowledge that our implementation may not be perfectly aligned with all aspects of SOLID, the design and structure of the game were guided by these principles. This provided valuable experience in designing more maintainable, extensible, and testable code.

5.1 What can be improved next time?

This game demonstrates a solid foundation, but there's significant room for expansion and refinement. Several key areas offer exciting possibilities for future development:

- **Transforming into a Rhythm Game:** Integrating music and rhythmic gameplay elements would add a new dimension to the experience. This could involve syncing orb appearances, player actions, and visual effects to the beat of the music.
- **Embracing the Selaphobia Theme:** Implementing flashing light effects is crucial to aligning the game with its name (Selaphobia - fear of

flashing lights). These effects should be carefully designed to be engaging and thematically relevant, while also considering accessibility for players with light sensitivity. Options to adjust or disable these effects would be important.

- **Expanding Orb Variety:** Introducing new orb types, such as gravity-changing, teleporting, and more powerful boost orbs, would significantly enhance gameplay variety and strategic depth. Careful balancing of these new mechanics would be essential.
- **Visual Enhancements:** Improving the graphics by adding more detailed backgrounds, diverse block designs, and potentially particle effects would greatly enhance the game's visual appeal. Consistent art style and high-resolution assets should be prioritized.
- **Map Editor Implementation:** Implementing a map editor would empower players to create and share their own levels, greatly extending the game's replayability. This is a complex feature, but exploring existing level editor tools and libraries could provide valuable guidance.
- **Collaborative Development:** Working with a team would bring diverse perspectives and skill sets to the project, accelerating development and improving the overall quality of the game. Collaboration would also provide opportunities for shared learning and problem-solving.