# MIR/ROR/RIM

By: Louis-Antoine LeBel, Nicolas Gauthier and Mark-Anthony Moisescu-Pareja

Integrative Project in Computer Science and Mathematics

420-204-RE section 00001

Elaheh Mozaffari

May 20th, 2020

# Table of Contents

# INTRODUCTION

## Story of the Project

Initially, during the early stages of determining an idea for the integrative project, our group felt it was important, and necessary to agree upon an idea that would motivate each group member to work on. An enjoyable/motivating project idea was crucial to come up with during the first stages of our integrative project, as it removed the notion of the project being a "long, tiresome school assignment", but instead made it a fun, interactive project where each group member happily devoted a portion of their free-time to working on the project. Due to the positive mindset associated with an enjoyable project idea, it would improve the outcome of the final project, as our group would put in more time into fixing bugs, implementing different features that would enhance the user's experience, etc.

In terms of goals obtained through the completion of the project, our group would like to further improve our knowledge in the programming language of Java, a popular object-oriented programming language that has been taught to us throughout our two years in the Computer Science & Mathematics program in CEGEP. Likewise, with more in-depth knowledge in Java, we are able to pursue other advanced personal projects that could pose as a solution to a current problem faced in today's society. Moreover, working on larger-scale projects like this prepares us better for potential internships at university or for future programming jobs.

*What is the Project About?*

After brainstorming numerous ideas, such as a cooperation game or a fighting game, and evaluating each one, the final consensus amongst the group was to create a puzzle-themed video

game, thus leading to the creation of our group's final project: MIR/ROR/RIM. More specifically, the video game is 2-dimensional, consisting of various levels, each with their own distinct map design. On each level, based on the map layout, as well as the item(s) provided, the user must complete the level's respective puzzle, in order to advance to the next level. Due to the integrative project being an interactive video game, the user will control an in-game character, with commands such as jumping, moving, picking up items as well as dropping items.

*What is the Project's Importance?*

Being an interactive, puzzle-themed video game that implements certain aspects of physics, our game, 'MIR/ROR/RIM' trains the user's cognitive thinking/problem-solving skills, as certain aspects of the human mind, such as critical thinking and visual perception are challenged when trying to complete each level. Likewise, with the implementation of physics aspects (friction, gravity, etc.), the user must appropriately strategize on how they will exploit the aspect of physics present in the given level to their advantage, in order to clear the level.

*What is the Project's Objective?*

When looking at today's current teaching methods, the most popular being the traditional classroom setting, where one teacher gives a lecture on a given topic to a rather large group of students, the student's level of retention is rather low, as when the class is over, the students tend to forget more than 50% of the material taught. As a result, certain students tend to grasp and retain information through visual/interactive teaching methods.

Keeping the information pertaining to visual/interactive teaching methods in mind, our group decided to create a user-friendly, interactive video game where the user can manipulate aspects of physics, as well as solve different puzzles. By creating an interactive, puzzle-themed video game, it encourages and trains an individual's creativity and pre-planning (forces the user

to think of a strategy to complete the level before attempting it) skills, as well as it builds connections to real-world phenomenon, as the user is experimenting with physical aspects such as friction and gravity. As a result, the user is training their cognitive thinking, as well as learning about various aspects of physics, all of which are objectives we hope our integrative project touch upon.

## Description of the Project

*System Objectives:*

Our main objective was to create a polished-looking 2D physics-based puzzle with a few levels to test out the player's critical thinking skills.

*Project Constraints and Scope:*

We initially had a wide scope for the project with many puzzle objects and dynamic elements. For example, inverting the gravity is one of the main features of the game and we were originally planning on it inverting the palette of the entire game when the gravity was inverted. However, we found out that if every single asset changed in the game, it was easy to get disoriented and lose track of the player's position, so we had to reduce our scope on that and only changes the player's and the background's appearance. Time was a big constraint, so much so that we removed some of the features and puzzle elements we were planning to focus on getting the few we had left to be as good as possible. LibGDX, while it still has many positives, does not offer flexible level editing and modern features many other non-java engines have.

*Tools and Methodologies:*

- The first tool needed to begin programming our video game was the IntelliJ IDE in order to start projects in Java as well as to add the packages used. The IDE, IntelliJ, was chosen by our group, as it comes with numerous useful shortcuts and possesses a nice autocomplete feature.
- Likewise, we also installed libGDX, a free and open-source Java game-development application framework that facilitated the development of our project. LibGDX allows for the creation of games for multiple platforms, including Windows, which was our desired platform, as every group member was on that platform, while exporting to another platform would result in additional testing and debugging, which is time-consuming and unpleasant especially with a limited amount of time.
- In addition, in order to create the assets, we used Piskel, a free online editor for animated sprites and pixel art.
- We also used the pay-what-you-want software Tiled, which is a software to quickly create grid-based level. We created a "TileSet" which contains all the different visual objects in our levels and could place them in a big 100x100 block grid, almost in a "drag and drop" fashion.

*Critical Project Events:*

<u>Week 1 & 2</u>

- During this period, our group decided to exchange various ideas that could potentially be pursued in the form of an integrated project. After brainstorming different ideas, our group agreed upon the idea of creating an interactive video game, and thus began researching creative tools that could be used for our project. We also looked into which

engine or IDE we were going to use. We ruled out Javafx very quickly because of its limited base-functionalities. We considered JMonkeyEngine after seeing a group from a previous year develop a game in it, but after some research, we found that it was not ideal for a 2D project such as ours. Finally, we settled on LibGdx for its implementation of Box2D physics, which facilitates physics simulation.

Week 3

- During the third week, basic designs of the video game's characters and objects were made, as well as the design for the GUI.

Week 4

- For the fourth week, we had to write the first report and present our project to the class. This involved choosing all the features we wanted in the game and the design of the different classes and interfaces in the game as well as their relationship between them.

Week 5

- Almost all the assets and animations for the level objects were made, such as the playable character, and we started reading documentation and watching tutorials to understand LibGdx as best as we could.

Week 6

- The quarantine started and the school closed, meaning that we could no longer see each other every week. The production of the game slowed down to a halt due to uncertainty about the school situation and also due to laziness.

Week 11-15

- Throughout these weeks, various advancements pertaining to our video game were made. To begin, the map rendering algorithm was worked on, and later finished, as well as the video game's animations. Furthermore, distinct level designs were created for each of the five levels present in our project, as well as the implementation of sounds/music within the video game.

## PROJECT DESIGN

### Analyzing/Defining/Understanding the Problem

In terms of a 'problem', the video game our group developed does not aim to solve a direct problem, but rather serves as a form of entertainment, as well as a learning tool. Interestingly, a potential 'problem' that could arise is exposing certain aspects of physics to students who have not yet learnt about them, or who are currently learning. Revealing new concepts of physics to the user develops their level of curiosity, as they are tempted to learn more about the aspects of physics present in the game, potentially doing research on 'said' aspects during their free time.

In order to resolve this 'problem', our group decided to create meaningful game levels that challenge the user to utilize their surroundings, as well as the physics that is applied to their in-game character in order to find solutions to more complex tasks. The skill of problem-solving is heavily prominent in careers such as engineering, as they are responsible for analysing all potential components of a problem.

## Algorithm

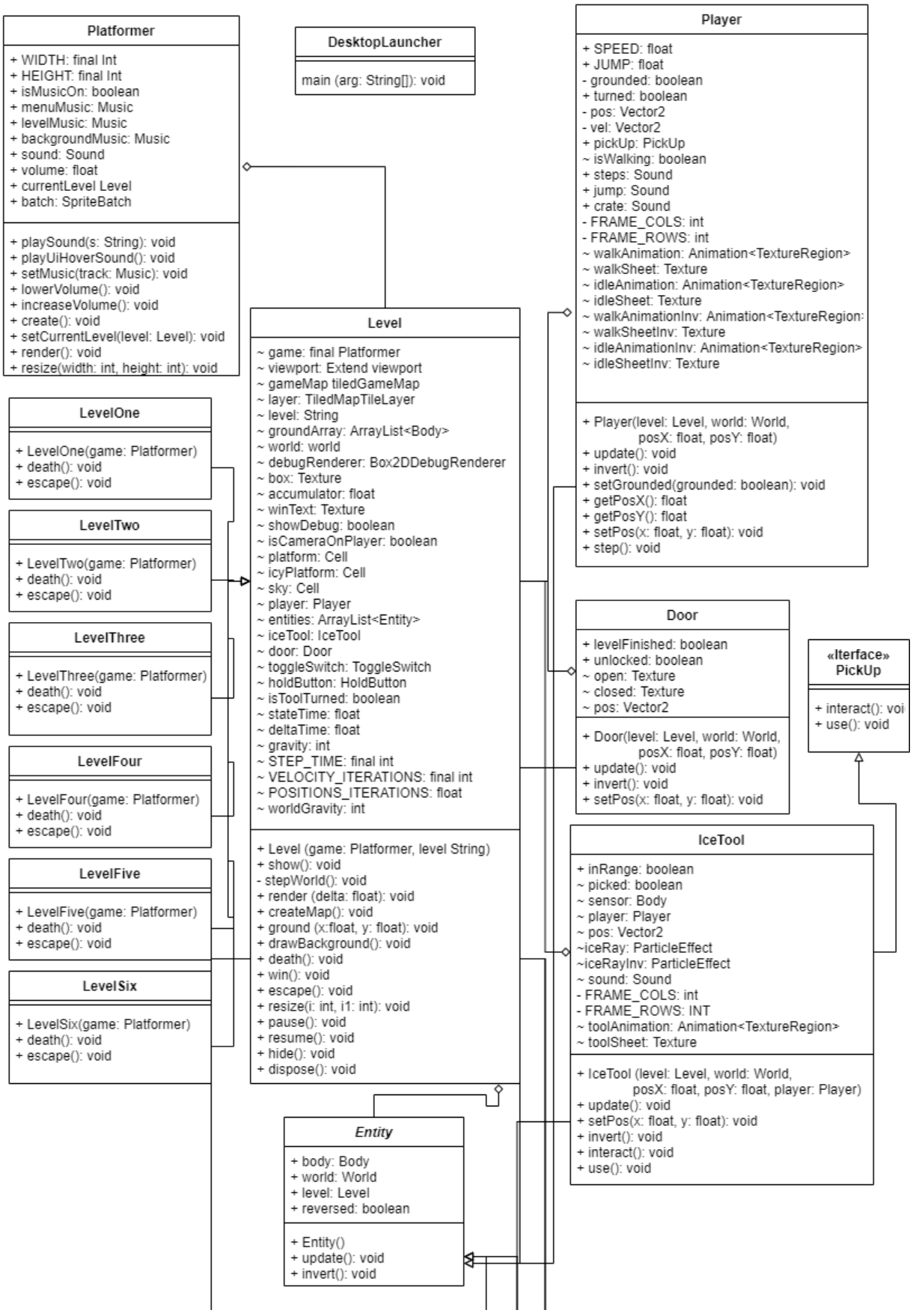
In a way, the code for the game is divided into three pillars:

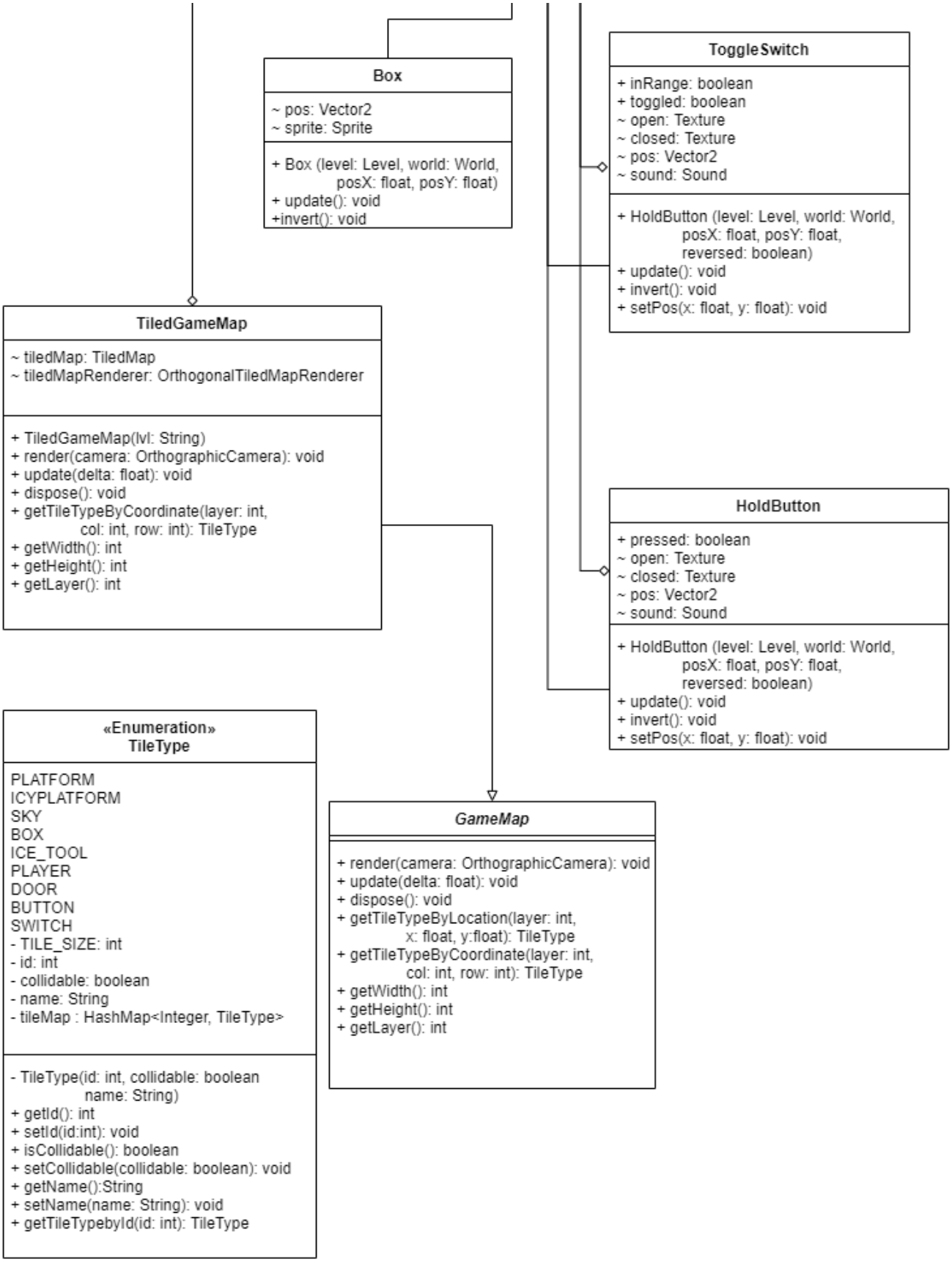
1. GUI/Menus
2. Game maps
3. Physics objects

The challenge was to get every pillar to work together seamlessly. For example, to make the maps the user will play on, we used Tiled. We chose this software to quickly design levels because otherwise we would have placed every single object on a map with precise coordinates, which is time consuming and generally inefficient. The problem with this approach though is that we end up with a 100x100 map of tiles that essentially acts as a static image, which meant that we had to find a way to implement physics into it. We ended up creating a tile type for each of the physics object, which we call entities. When loading a level, the program checks the tile of each and every tile in the Tiled file and if, for example, it finds the player tile, it places the Player entity at the tile's position and deletes the tile since it is just an image. The following is the UML diagram for pillars 1 and 2 followed by explanations for almost each of the methods and parameters.

## Design of the Project

*UML:*


Platformer
+ WIDTH: final Int
+ HEIGHT: final Int
+ isMusicOn: boolean
+ menuMusic: Music
+ levelMusic: Music
+ backgroundMusic: Music
+ sound: Sound
+ volume: float
+ currentLevel Level
+ batch: SpriteBatch
+ playSound(s: String): void
+ playUiHoverSound(): void
+ setMusic(track: Music): void
+ lowerVolume(): void
+ increaseVolume(): void
+ create(): void
+ setCurrentLevel(level: Level): void
+ render(): void
+ resize(width: int, height: int): void
DesktopLauncher
main (arg: String[]): void
Player
+ SPEED: float
+ JUMP: float
- grounded: boolean
+ turned: boolean
- pos: Vector2
- vel: Vector2
+ pickUp: PickUp
~ isWalking: boolean
+ steps: Sound
+ jump: Sound
+ crate: Sound
- FRAME_COLS: int
- FRAME_ROWS: int
~ walkAnimation: Animation<TextureRegion>
~ walkSheet: Texture
~ idleAnimation: Animation<TextureRegion>
~ idleSheet: Texture
~ walkAnimationInv: Animation<TextureRegion>
~ walkSheetInv: Texture
~ idleAnimationInv: Animation<TextureRegion>
~ idleSheetInv: Texture
+ Player(level: Level, world: World, posX: float, posY: float)
+ update(): void
+ invert(): void
+ setGrounded(grounded: boolean): void
+ getPosX(): float
+ getPosY(): float
+ setPos(x: float, y: float): void
+ step(): void
Level
~ game: final Platformer
~ viewport: Extend viewport
~ gameMap tiledGameMap
~ layer: TiledMapTileLayer
~ level: String
~ groundArray: ArrayList<Body>
~ world: world
~ debugRenderer: Box2DDebugRenderer
~ box: Texture
~ accumulator: float
~ winText: Texture
~ showDebug: boolean
~ isCameraOnPlayer: boolean
~ platform: Cell
~ icyPlatform: Cell
~ sky: Cell
~ player: Player
~ entities: ArrayList<Entity>
~ iceTool: IceTool
~ door: Door
~ toggleSwitch: ToggleSwitch
~ holdButton: HoldButton
~ isToolTurned: boolean
~ stateTime: float
~ deltaTime: float
~ gravity: int
~ STEP_TIME: final int
~ VELOCITY_ITERATIONS: final int
~ POSITIONS_ITERATIONS: float
~ worldGravity: int
+ Level (game: Platformer, level String)
+ show(): void
- stepWorld(): void
+ render (delta: float): void
+ createMap(): void
+ ground (x:float, y: float): void
+ drawBackground(): void
+ death(): void
+ win(): void
+ escape(): void
+ resize(i: int, i1: int): void
+ pause(): void
+ resume(): void
+ hide(): void
+ dispose(): void
LevelOne
+ LevelOne(game: Platformer)
+ death(): void
+ escape(): void
LevelTwo
+ LevelTwo(game: Platformer)
+ death(): void
+ escape(): void
LevelThree
+ LevelThree(game: Platformer)
+ death(): void
+ escape(): void
LevelFour
+ LevelFour(game: Platformer)
+ death(): void
+ escape(): void
LevelFive
+ LevelFive(game: Platformer)
+ death(): void
+ escape(): void
LevelSix
+ LevelSix(game: Platformer)
+ death(): void
+ escape(): void
Door
+ levelFinished: boolean
+ unlocked: boolean
~ open: Texture
~ closed: Texture
~ pos: Vector2
+ Door(level: Level, world: World, posX: float, posY: float)
+ update(): void
+ invert(): void
+ setPos(x: float, y: float): void
«Iterface» PickUp
+ interact(): voi
+ use(): void
IceTool
+ inRange: boolean
~ picked: boolean
~ sensor: Body
~ player: Player
~ pos: Vector2
~iceRay: ParticleEffect
~iceRayInv: ParticleEffect
~ sound: Sound
- FRAME_COLS: int
- FRAME_ROWS: INT
~ toolAnimation: Animation<TextureRegion>
~ toolSheet: Texture
+ IceTool (level: Level, world: World, posX: float, posY: float, player: Player)
+ update(): void
+ setPos(x: float, y: float): void
+ invert(): void
+ interact(): void
+ use(): void
Entity
+ body: Body
+ world: World
+ level: Level
+ reversed: boolean
+ Entity()
+ update(): void
+ invert(): void


Box
~ pos: Vector2
~ sprite: Sprite
+ Box (level: Level, world: World, posX: float, posY: float)
+ update(): void
+invert(): void
ToggleSwitch
+ inRange: boolean
+ toggled: boolean
~ open: Texture
~ closed: Texture
~ pos: Vector2
~ sound: Sound
+ HoldButton (level: Level, world: World, posX: float, posY: float, reversed: boolean)
+ update(): void
+ invert(): void
+ setPos(x: float, y: float): void
TiledGameMap
~ tiledMap: TiledMap
~ tiledMapRenderer: OrthogonalTiledMapRenderer
+ TiledGameMap(lvl: String)
+ render(camera: OrthographicCamera): void
+ update(delta: float): void
+ dispose(): void
+ getTileTypeByCoordinate(layer: int, col: int, row: int): TileType
+ getWidth(): int
+ getHeight(): int
+ getLayer(): int
HoldButton
+ pressed: boolean
~ open: Texture
~ closed: Texture
~ pos: Vector2
~ sound: Sound
+ HoldButton (level: Level, world: World, posX: float, posY: float, reversed: boolean)
+ update(): void
+ invert(): void
+ setPos(x: float, y: float): void
«Enumeration»
TileType
PLATFORM
ICYPLATFORM
SKY
BOX
ICE_TOOL
PLAYER
DOOR
BUTTON
SWITCH
- TILE_SIZE: int
- id: int
- collidable: boolean
- name: String
- tileMap : HashMap<Integer, TileType>
- TileType(id: int, collidable: boolean name: String)
+ getId(): int
+ setId(id:int): void
+ isCollidable(): boolean
+ setCollidable(collidable: boolean): void
+ getName():String
+ setName(name: String): void
+ getTileTypebyId(id: int): TileType
GameMap
+ render(camera: OrthographicCamera): void
+ update(delta: float): void
+ dispose(): void
+ getTileTypeByLocation(layer: int, x: float, y:float): TileType
+ getTileTypeByCoordinate(layer: int, col: int, row: int): TileType
+ getWidth(): int
+ getHeight(): int
+ getLayer(): int

**DesktopLauncher**

This method is the project's main method. It sets the resolution, the maximum framerate, sets the window to not be resizable by users and makes the program run at full screen. It also creates a Platformer instance.

**Platformer**

This method extends the Game abstract class from Libgdx. It is used to transition between the multiple level screens and menu screens as well as playing the background music continuously between the screens.

*Field Summary*

- public static final int WIDTH : Width of the window, set to 1920 pixels.
- public static final int HEIGHT: Height of the window, set to 1080 pixels.
- public boolean isMusicOn: Defines if music is playing.
- public Music menuMusic: Music object for the menu music.
- public Music levelMusic: Music object for the level music.
- public Music backgroundMusic: Music object for music currently playing.
- private Sound sound: Sound object for UI click sound.
- public float volume: value for background music, minimum is 0, maximum is 1.
- Level currentLevel: defines what level is currently being played.
- public SpriteBatch batch: Spritebatch object from libgdx to draw textures and sprites on menu screens.

*Method summary*

- public void playSound(String s): loads the sound file with directory s and plays it once.
- public void playUiHoverSound(): plays the sound when user hovers over UI element.
- public void setMusic(Music track): stops music currently playing, starts track, loops it, sets the volume and sets isMusicOn to true.
- public void lowerVolume(): lowers the volume by 0.1f if volume is above 0.
- public void increaseVolume(): increases the volume by 0.1f if volume is below 1.
- public void create (): acts as a contrsuctor to Platformer, it runs when a Platformer is created in the DesktopLauncher. It initializes batch, loads the music files and sets the screen to MainMenu.
- public void render (): actually draws object on screen and creates a window. Runs once every frame, so 60 times a second.
- public void resize(int width, int height): Used to resize the window if allowed in DesktopLauncher.

**Level**

This class implements the Screen interface from Libgdx, which is used to show textures, sprites, animation, etc. drawn by a Batch or SpriteBatch objects from Libgdx. This class defines where the user plays the actual game and interacts with the different game entities.

*Field Summary*

- final Platformer game: Platformer object.
- OrthographicCamera cam: game camera, defines what the user will see at any given time. OrthographicCamera is a class from Libgdx.
- ExtendViewport viewport: related to cam object.

- TiledGameMap gameMap: the tiled map the player will play on.
- TiledMapTileLayer layer: to interact with the gameMap's tiles in real time on a certain layer of the map.
- String level: directory to load right level. Levels load files created in the Tiled software, saved as a .tmx extension.
- ArrayList<Body> groundArray: Array that stores all of the Box2D bodies that make up the floor the player will walk on.
- World world: Box2D world object for Box2D objects to interact with each other. Box2D bodies refer to this object for the strength of gravity.
- Box2DDebugRenderer debugRenderer: to show the "hitbox" of different Box2D objects.
- float accumulator: related to physics calculations.
- boolean showDebug: defines if debugRenderer should be on.
- boolean isCameraOnPlayer defines if cam object is on the player.
- Cell platform: stores the PLATFORM TileType
- Cell icyPlatform: stores the ICYPLATFORM TileType.
- Cell sky: stores the SKY TileType.
- Player player: Player object for the level.
- ArrayList<Entity> entities: ArrayList containing all entities in the level.
- IceTool iceTool: IceTool object for the level.
- Door door: Door object for the level.
- ToggleSwitch toggleSwitch: To ToggleSwitch object for the level.
- HoldButton holdButton: HoldButton object for the level
- float stateTime: stores time span between the current and last frame frame in seconds.

- float deltaTime: stores the current frame in seconds.
- int gravity: int value =-1 when gravity is downward and =1 when gravity is upwards.
- static final float STEP_TIME: time in seconds between each physics calculation.
- static final int VELOCITY_ITERATIONS: no clue
- static final int POSITION_ITERATIONS: no clue
- int worldGravity: strength of gravity for the world object.

*Method Summary*

- public Level(Platformer game, String level): Constructor for the Level class. Initializes variables, creates player, iceTool, door, toggleSwitch, holdButton entities and places them at temporary positions. It also sets the music to levelMusic in the game obect. Lastly it sets the world object's ContactListener, which is a Box2D class to register collisions between Box2D bodies and identify bodies at play. Let's look at the collision between the player and the door object in a level which will finish the level:

```
if(contact.getFixtureA().getBody().getUserData()== player &&
contact.getFixtureB().getBody().getUserData() == door && door.unlocked){
    door.levelFinished = true;
    System.out.println("yay");
    win();
}
```

At first the game checks if the first body in the collision (FixtureA) is a Player instance, if the second body (FixtureB) is a Door instance and if the door is unlocked. It that if statement is true, it triggers the win() method in the Level method, which ends the level.

- public void show(): This method is ran right after the constructor. It loads the gameMap and projects the camera unto it. It initializes the three Cell objects and calls the createMap() method.

- private void stepWorld(): This method calculates physics displacements between each frame and is called once in the render() method.
- public void render(float delta): This method runs every frame, so 60 times every second. It updates the camera position, updates the entities ArrayList, so that the entities continue moving according to every frame. This method also deals with keyboard inputs, like the escape button that calls the escape() method which in turn sets the screen to the PauseMenu:

```
if(Gdx.input.isKeyJustPressed(Input.Keys.ESCAPE)){
    //Stop any active sounds to prevent them playing in subsequent menus
    player.steps.stop();
    iceTool.sound.stop();

    escape(); //call escape method
}
```

The method also has a set keyboard inputs that only work when showDebug is True, such as creating Box objects on the cursor when pressing P, inverting the gravity when pressing G, setting the camera free from the player when pressing C, change the TileType of the tile the cursor is on when pressing T and drawing invisible solid boxes when pressing B. The debug mode can be turned on or off by pressing F3.

- public void createMap(): This method analyses the gameMap object and checks what tile every single tile is. If the TileType of the tile the method is currently looking is collidable, it will "draw" a rectangular box2D static body around the location of the tile. If the tile is not collidable, a switch case will analyze what entity it should draw at a the tile location.
- public void ground(float x, float y): this method creates a transparent box2D static box at position x and y of dimension 16x16 pixels.

- public void drawBackground(): draws a black background if the gravity is downward and white if the gravity is upwards.
- public  void death(): sets the screen to DeathScreen.
- public void win(): sets the screen to WinScreen.
- public void escape(): sets the Screen to PauseMenu.
- public void resize(int i, int i1): Would resize the window were the option available
- public void pause(): overriden from the Screen interface, does not do anything.
- public void resume(): overriden from the Screen interface, does not do anything.
- public void hide(): overriden from the Screen interface, does not do anything.
- public void dispose(): calls the world's dispose method. This method is called when the screen is changed.

**LevelOne**

Instead of looking at each subclass of Level, we'll limit ourselves to LevelOne since they are all quite identical.

*Method Summary*

• public LevelOne(Platformer game): This constructor invokes its superclass' constructor and places the directory for the first level, "map\\level_1.tmx" in this case.

• public void death(): This is an overridden method from Level. It ensures that when the player dies by going outside the map, the right level loads back.

• public void escape(): Similar to death(), but it just loads back the right level when the player comes back from the pause menu.

**Entity**

This abstract class describes objects that can be interacted with. It has a few subclasses: Player, IceTool, HoldButton, ToggleSwitch, Door and Box

*Field Summary*

- public Body body: This is the Box2D body the entity has so that entities can physically interact with eachother by collisions
- public World world: This is the World Box2D object attached to the entity so that it can look up its gravity value and therefore move in space.
- public Level level: Level object so that the entity can easily access information about its surroundings.
- public boolean reversed: if the entity has its gravity inversed.

*Method Summary*

- public Entity(): no-arg contructor.
- public abstract void update(): used to handle inputs and update physics on entity.
- public abstract void invert(): Called when gravity is inverted. Ended up only used for the Player class.

**Player**

The Player class allows the user to interact with the world by walking, jumping, inverting the gravity and using the IceTool.

*Field Summary*

- public static final float SPEED: this is the maximum speed value for the player, set at 50f.

- public static float JUMP: this is the maximum vertical velocity value for the player, set at 1000f.
- private boolean grounded: defines whether the player is touching the ground or not.
- public boolean turned: defines whether the player is facing right or not.
- private Vector2 pos: 2D vector storing the player's position in the world.
- private Vector2 vel: 2D vector storing the player's velocity.
- boolean isWalking: defines whether the player is walking or not.
- public Sound steps: walking sound.
- public Sound jump: jumping sound.
- public Sound crate: sound when the player hits a box.
- private static final int FRAME_COLS , FRAME_ROWS : dimensions of image containing the 16 different images for the player's animation.
- Animation<TextureRegion> walkAnimation: Libgdx animation class for the normal walking animation.
- Texture walkSheet: Texture containing the 16 different images for the player's animation.
- Animation<TextureRegion> idleAnimation: : Libgdx animation class for the normal idle animation.
- Texture idleSheet: Texture containing the 16 different images for the player's animation.
- Animation<TextureRegion> walkAnimationInv: Libgdx animation class for the walking animation when gravity is inversed.
- Texture walkSheetInv: Texture containing the 16 different images for the player's animation.

- Animation<TextureRegion> idleAnimationInv: Libgdx animation class for the idle animation when gravity is inversed.
- Texture idleSheetInv: Texture containing the 16 different images for the player's animation.

*Method Summary*

public Player(Level level, World world, float posX, float posY): Constructor for the Player class. It loads the three Sound objects, creates the Box2D body for the hitbox and initializes the four animations by taking an image such as this:

It then splits into 16 different images, places them in order into the Animation object and cycles through them when the animation starts playing.

The code to create initialize this animation looks like so:

```
walkSheet = new Texture(Gdx.files.internal("entity\\john.png")); //load texture
TextureRegion[][] tmp = TextureRegion.split(walkSheet,
        walkSheet.getWidth() / FRAME_COLS,
        walkSheet.getHeight() / FRAME_ROWS);//split iceTool picture into 16 distinct,
singular images
TextureRegion[] walkFrames = new TextureRegion[FRAME_COLS * FRAME_ROWS];
int index = 0;
//for loops to assign each index of the texture region to a frame of animation
for (int i = 0; i < FRAME_ROWS; i++) {
    for (int j = 0; j < FRAME_COLS; j++) {
        walkFrames[index++] = tmp[i][j];
```

```
    }
}
//create animation with the texture region
walkAnimation = new Animation<TextureRegion>(0.15f, walkFrames);
```

- public void update(): This method is called once in the Level class' render method, so it ends up being called once every frame. This method handles user input for movement.

```
• if(Gdx.input.isKeyPressed(Input.Keys.A) && vel.x > -SPEED){ //check if
  velocity is below maximum velocity
      turned = true; //set turned boolean to true
      body.setLinearVelocity(vel.x - 50f,vel.y); //remove 50f to the velocity in
  x
  }
```

This is the part to handle moving left. As long as the user keeps pressing A and that the velocity is above the minimum defined by SPEED, we decrease 50f to the velocity so that the player moves to the left. The same principle is used for moving right and jumping, but the velocity is changed in different directions for every case. This method also handles dynamic friction modification.

```
else if(((level.gameMap.getTileTypeByLocation(1, pos.x, pos.y -16f) ==
TileType.getTileTypeById(1) && !reversed)
        || (level.gameMap.getTileTypeByLocation(1, pos.x, pos.y +16f) ==
TileType.getTileTypeById(1) && reversed)) && grounded){
    body.setLinearDamping(60f);
}
```

This is a part of an "if statement" to check the ground below the player. When the gravity is not reversed, the program checks the tile at the position 16 pixels below the player's position on the game map. It checks 16 pixels because each tile is 16x16 pixels. Here, it retrieves the ID of the tile and if it is equal to 1, an icy platform, the program sets the "linear dampening" on the player to 60f. This is not the same as changing the friction, but it gives a similar-enough feeling. We opted for this solution because changing the friction like this in real time involves a lot more

lines of code and creating different physics bodies, which was too complicated for the instantaneous nature of the task. Now, back to the code, the “or” part of the statement checks the tile above the player if the gravity is reversed so that the friction is modified even when the gravity is reversed. This method also decides which animation it should draw based on the player’s velocity and whether it should vertically flip the animation or not based on the gravity.

```
//if player is between these velocities draw walking animation
if(vel.x > 10f || vel.x < -10f){
    step(); //play step sounds
    isWalking = true;
    //draw inverted animation if gravity is inverted
    if(reversed) {
        TextureRegion currentFrame = walkAnimationInv.getKeyFrame(level.stateTime,
true);
        level.gameMap.tiledMapRenderer.getBatch().draw(currentFrame, pos.x - 6, pos.y
+ 15 / 2, currentFrame.getRegionWidth(), currentFrame.getRegionHeight() *-1);
    }
    else{
        TextureRegion currentFrame = walkAnimation.getKeyFrame(level.stateTime,
true);
        level.gameMap.tiledMapRenderer.getBatch().draw(currentFrame,pos.x -6, pos.y -
15/2);
    }



}
else{ //idle animation
    isWalking = false;
    steps.stop();
    if(reversed) {
        TextureRegion currentFrame = idleAnimationInv.getKeyFrame(level.stateTime,
true);
        level.gameMap.tiledMapRenderer.getBatch().draw(currentFrame, pos.x - 6, pos.y
+ 15 / 2, currentFrame.getRegionWidth(), currentFrame.getRegionHeight() *-1);
    }
    else{
        TextureRegion currentFrame = idleAnimation.getKeyFrame(level.stateTime,
true);
        level.gameMap.tiledMapRenderer.getBatch().draw(currentFrame,pos.x -6, pos.y -
15/2);
    }
}
```

Every velocity in between [-10f, 10f] is considered as idle and as such the isWalking boolean is set to false. Otherwise, isWalking is set to true and the program draws the animation on screen

using the level's SpriteBatch. The program also finds which frame in a second the level is at and finds the right frame the animation should be at. This way, transitioning between each animation is seamless and frictionless.

- Invert(): This method inverts the level's gravity and can only be called in the player's update() method when the player is grounded.
- Set methods for the grounded boolean, position and get methods for the player's position in x and y.
- public void step(): This method is called when the use just pressed A or D and starts playing and loops the walking sound if the player was not already walking.

**IceTool**

This is the tool the player can pick up and use to freeze the ground around them so that the "friction" on the player will be altered.

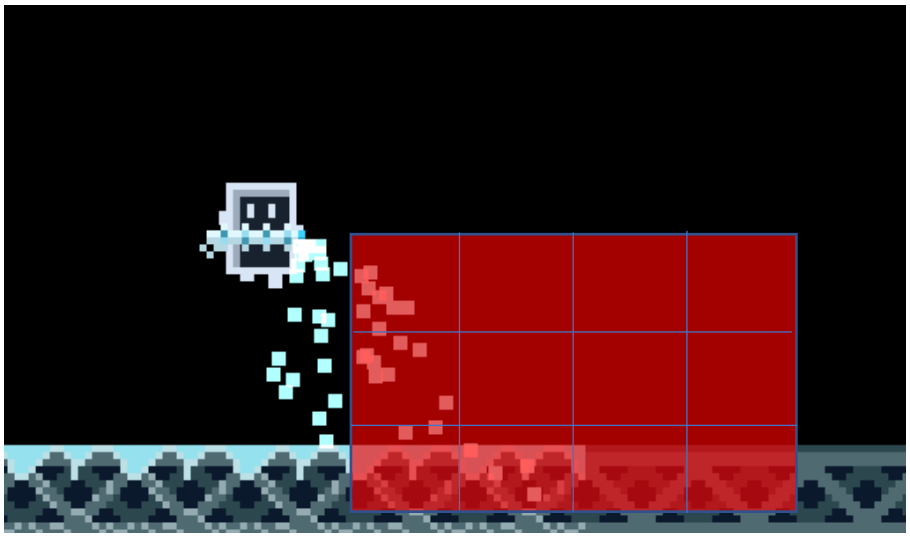
*Field Summary*

- public boolean inRange: This boolean is set to true when the player is in range from the tool and false if not. The player can only pick up the tool if inRange is set to true.
- boolean picked: If the tool has been picked up by the player.
- Body sensor: The Box2D body that will check for player proximity.
- Player player: the player object associated with the tool so that the tool can access its parameters.
- Vector2 pos: Tool's position vector.
- ParticleEffect iceRay: Visual effect for the ray of "ice" that comes out of the tool.
- ParticleEffect iceRayInv: Same as iceRay, but inverted along the Y-axis.

- Sound sound: Sound effect for using the tool.
- private static final int FRAME_COLS, FRAME_ROWS: dimensions of image containing the 4 different images for the tool's animation.
- Animation<TextureRegion> toolAnimation: Animation class for the tool.
- Texture toolSheet: Texture with the four different frames of the animation.

*Method Summary*

- public IceTool(Level level, World world, float posX, float posY, Player player): This is the constructor that loads the sound, creates the Box2D bodies for the body of the tool and the sensor, initializes the animation the same way as the player and initiates both particle effects.
- public void update(): Update method for the IceTool. Checks if the user presses E and if the player is in range to pick or let go of the tool. Also draws the animation based on the player's orientation. For example, if the gravity is reversed and the player is walking to the left, the animation will be flipped both in the X and Y axis. The method also checks if the player presses Spacebar to use the tool, which will trigger the particle effect to shoot out of the tool. The program also checks in a 4x3 tile area from the tool's position the tiletypes in the level.



With two loops, the program checks every tile in the red area to determine the tiletype of the tile it is looking at. If the is PLATFORM it will turn it to ICYPLATFORM. Lastly, the update() plays and loop the sound when using the tool.

- public void setPos(float x, float y): sets the tool's position to x and y.
- public void interact(): set the picked Boolean to true when called.
- public void use(): prints a message in the IntelliJ terminal.

**HoldButton & ToggleSwicth**

These two classes are very similar to each other and are used to "open" the Door of a level. They only differ in how they are activated: the HoldButton needs constant force on it to remain activated while the ToggleSwitch is on an on/off mechanism.

*Field Summary*

- public boolean inRange (only for ToggleSwitch): This boolean is set to true when the switch is in range from the player and false if not. The player can only activate the switch if inRange is set to true.

- public boolean toggled (Swicth), public boolean pressed (HoldButton): if they are activated.
- Texture open: Texture when activated.
- Texture closed: Texture when deactivated.
- Vector2 pos: Position vector.
- Sound sound: Sound on activation or deactivation.
- public HoldButton(Level level, World world, float posX, float posY, boolean reversed): loads the sound, creates the Box2D body, load both texture. Exact same constructor ToggleSwitch.
- public void update(): Draws the right Texture based on toggled or pressed boolean. For the ToggleSwitch, the method also handles the player pressing Q to activate it.
- public void invert(): sets the reversed boolean from Entity to true.
- public void setPos(float x, float y): sets the position to x and y.

**Door**

This is what the player needs to go through to complete the level. Its field and methods to those of HoldButton, except for an extra levelFinished boolean that is true when the door is triggered by a switch or button.

**Box**

This is a box that the player can push around and jump on.

*Field Summary*

- Vector2 pos: position vector for the box.

- Sprite sprite: this entity exclusively uses a sprite instead of a texture to ease sprite rotation. The Sprite class is from Libgdx and acts rather similar to the Texture we are using for the rest of the game. The main difference is when drawing the sprite using a SpriteBatch. For a texture, we draw using the draw() method in the SpriteBatch of the level, but with the Sprite class, the sprite itself has a draw() method with the level SpriteBatch as an argument.

*Method Summary*

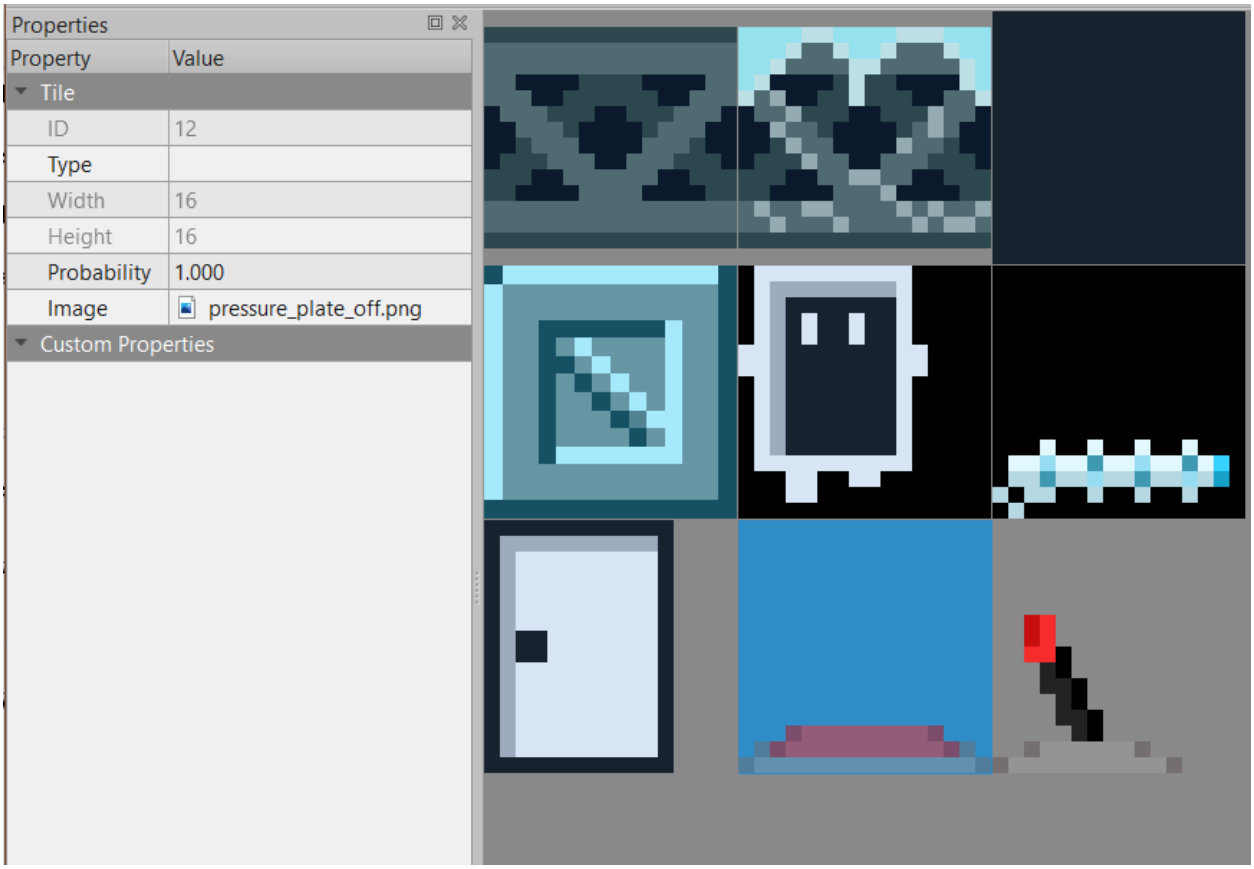
- public Box(Level level, World world, float posX, float posY): constructor for the box. It created the Box2D body and load the sprite.
- public void update(): updates the position Vector, and draws the Sprite at the correct position on the body with the body's rotation.

**TileType**

This enum class is somewhat of a way to communicate between the Tiled level files and the rest of the game.

*Field Summary*

PLATFORM, ICYPLATFORM, SKY, BOX, ICE_TOOL, PLAYER, DOOR, BUTTON, SWITCH : These are all the different tiles from the tileset:




Each tile is a .png image created beforehand in Piskel that is loaded in Tiled

- public static final int TILE_SIZE: size in pixels of each tiletype, as can be seen in the "property" tab under width and height.
- private int id: ID of the tile type, as can be seen in the "property" tab. When creating a new TileType, it is important to add one to the ID seen in Tiled when setting the id variable. The createMap() method from the level checks the tile ID and creates the right entity based on the ID.
- private boolean collidable: If the tile is collidable. The createMap() method from the level check if the tile is collidable to know if it should create a floor at the tile location.

- private String name: name of the tile.

*Method Summary*

- private TileType (int id, boolean collidable, String name): creates a TileType with the parameters in the constructor. Called several times at the top of the class to create the different tile types:

```
PLATFORM(1, true, "Platform"),
ICYPLATFORM(2, true, "Icy_Platform"),
SKY(3, false, "sky"),
BOX(4, false, "box"),
ICE_TOOL(8, false, "ice_tool"),
PLAYER(7, false, "player"),
DOOR(12, false, "door"),
BUTTON(13, false, "button"),
SWITCH(14, false, "switch");
```

- Sets and gets for id, collidable and name.
- public static TileType getTileTypeById (int id): return the TileType based on the tile's ID.

**GameMap and TiledGameMap**

GameMap is the superclass of TiledGameMap. These classes actually load the Tiled map files with methods built into LibGdx.

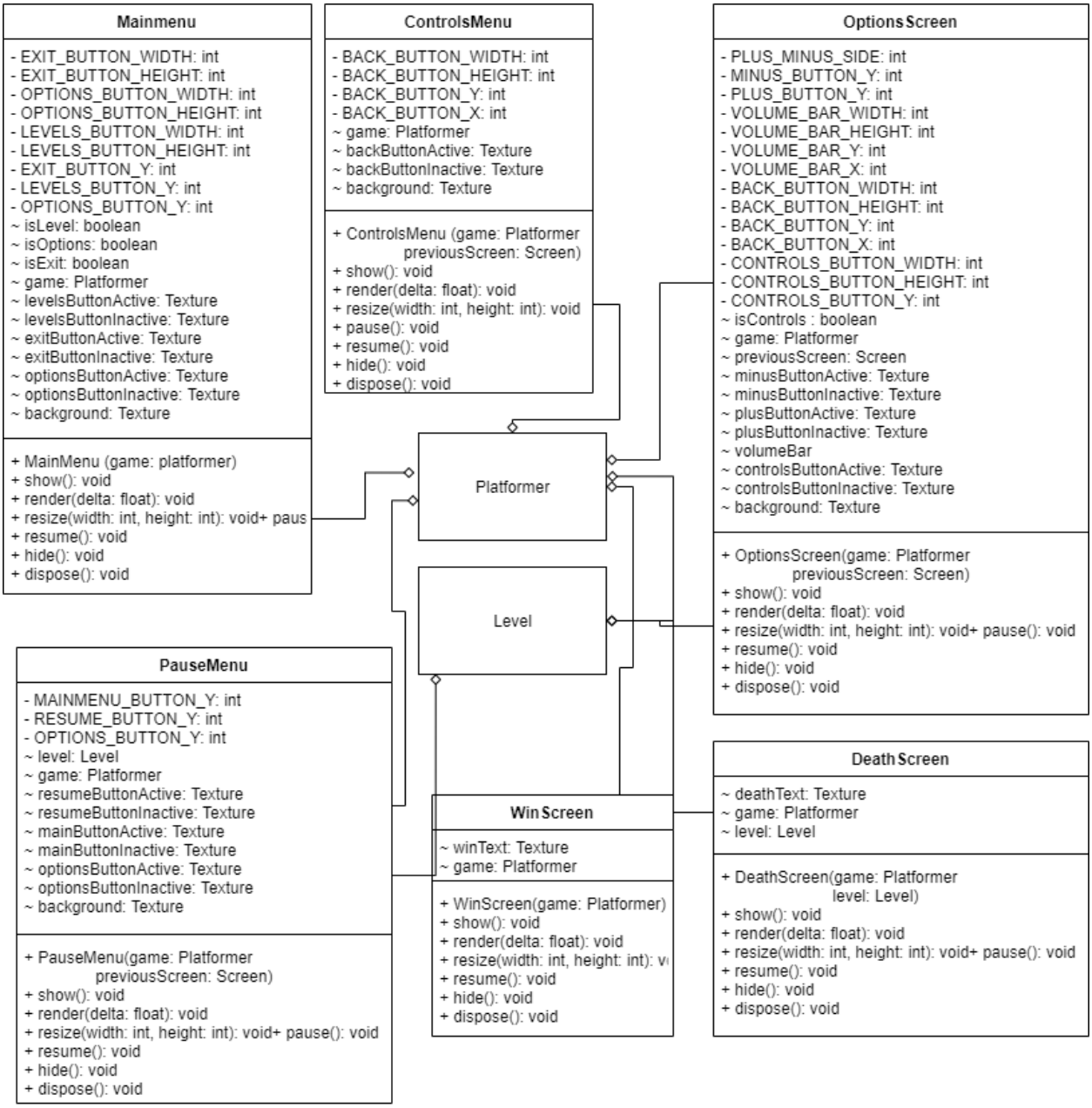
*Field Summary*

- TiledMap tiledMap: TiledMap is a class from LibGdx that hold a Tiled file for a map.
- OrthogonalTiledMapRenderer tiledMapRenderer: OrthogonalTiledMapRenderer is a class from LibGdx to render a TiledMap object on screen.

*Method Summary*

- public TiledGameMap(String lvl): loads the Tiled file using the TmxMapLoader class in LibGdx and starts rendering it.
- public void render(OrthographicCamera camera): called in the Level class. It sets the level's camera to show the tiled map.
- public void update(float delta): unused.
- public void dispose(): disposes of the tiled map when it gets unloaded.
- public TileType getTileTypeByCoordinate(int layer, int col, int row): returns the TileType of tile based on the colum and row of the tile. In a Tiled map, (0,0) is at top-left corner.
- public TileType getTileTypeByLocation(int layer, float x, float y): Takes float x and y, divides both values by the Tiled map's tilesize (16 in our game) and calls getTileTypeByCoodinate to return the TileType.
- public int getWidth(): Returns the horizontal number of tiles.
- public int getHeight(): Returns the horizontal number of tiles.
- public int getLayer(): return the number of layers in the Tiled Map.

*GUI:*




All the menu implement the Screen interface from LibGdx. This way, we can easily switch between each menu screen. All the screen function in a very similar way to each other, so we will only be looking at the Controls menu.

**ControlsMenu**

This menu supplies the user with the control he needs to control to control the player. Each menu screen has two types of element: a background and buttons. Here is the background for the ControlsMenu:


This menu also has only one button, the back button to return to return to the previous menu screen. Each button has an active and inactive state that switches if the cursor is on the button: <Back <Back

The first image is drawn when the cursor is away from the button and the second, when the cursor is on the button. Either image is drawn at the top-left of the screen at all times.

*Field Summary*

- private static final int BACK_BUTTON_WIDTH: Width of the back button
- private static final int BACK_BUTTON_HEIGHT: Height of the back button
- private static final int BACK_BUTTON_Y: Y-position of the button

- private static final int BACK_BUTTON_X: X-position of the button
- final Platformer game: Platformer object
- Screen previousScreen: To store the previous screen the user was on.
- Texture backButtonActive: Texture for when the button is active.
- Texture backButtonInactive: Texture for when the button is inactive.
- Texture background: Texture for the background.

*Method Summary*

- public ControlsMenu (final Platformer game, Screen previousScreen): loads the textures and initializes parameters.
- public void render (float delta): Runs every frame and draws all the textures. Detects if the mouse if on a button:

```
if(Gdx.input.getX() < x + BACK_BUTTON_WIDTH && Gdx.input.getX() > x &&
Platformer.HEIGHT - Gdx.input.getY() < BACK_BUTTON_Y + BACK_BUTTON_HEIGHT &&
Platformer.HEIGHT - Gdx.input.getY() > BACK_BUTTON_Y) {
    game.batch.draw(backButtonActive, x, BACK_BUTTON_Y, BACK_BUTTON_WIDTH,
BACK_BUTTON_HEIGHT);

    //detect mouse input
    if(Gdx.input.isButtonJustPressed(Input.Buttons.LEFT)){
        game.setScreen(new OptionsScreen(game, previousScreen));
        game.playSound("sounds\\click.wav");
    }
}
else{
    game.batch.draw(backButtonInactive, x, BACK_BUTTON_Y, BACK_BUTTON_WIDTH,
BACK_BUTTON_HEIGHT);
}
```

Here if the x-position of the mouse (gotten from Gdx.input.getX()) is within the bounds defined by the button's width and x-position and the same in the y-direction, then the method draws "backButtonActive" at the button's coordinates with the right dimension. Also, if the user actually clicks on the button, then it will set the screen to the previous and play the clicking sound.

All the other menus run on the same principle, only they have more buttons.

## METHODS OF EVALUATION

In order to assure that our project matched our high standards for it and be completed in the limited time given to us, our group used a few methods of evaluation throughout the months we worked on the project. For instance:

- After deciding upon the idea of creating a video game as our integrative project, we had to verify if our idea was achievable. Using a formative evaluation, we checked if the group could realistically complete the project given a set amount of time, as well as having limited knowledge in resources we planned to use such as libGDX. After creating a detailed schedule pertaining to the tasks to be completed, as well as educating ourselves with libGDX, our group concluded that completing the project was attainable.
- During the early stages of our project, we decided to create a schedule that possessed the various tasks to be done, accompanied by the time period they should be completed by. Assessing our group's final project, we can agree upon the fact that we followed the schedule thoroughly up until the project's due date. Our group used process evaluation to test features/implementations present in our video game throughout the period we worked on the project.

# RESULTS: SYSTEM QUALITY

## Developer Perception

In the end, our group heavily enjoyed working on the integrative project, and felt somewhat fulfilled once it was completed. Initially, in the early stages of development, our group questioned if we would be capable of producing a finished product implementing all of the features we had anticipated to implement, however, being fuelled by determination and enjoyment, our group managed to complete the project in the allotted time given to us.

While working on the project, a good portion of our time was devoted to research, more specifically watching tutorials on how to use libGDX and learn about all its features, as it was a game development framework that we did not have much prior knowledge in. Using libGDX and getting to know how to use it more profoundly aided us enormously, as its built-in features saved us lots of time (we did not have to build everything from scratch in a typical IDE) and allowed ourselves to make something that looked somewhat professional, unlike the amateur look often given by Javafx programs.

Unfortunately, as mentioned previously, the quarantine drastically slowed down the development of our project. As a result, there was a period of about a week where we rushed to complete the project and while coding the game remained fun, it was a stressful time. We were able to implement a majority of the features we had anticipated, albeit with some cuts. We had originally planned to have an additional “light” object that could reflect on mirrors and activate doors, but we removed that object due to lack of time. Moreover, only the player is affected by friction modification instead of all physical objects in the level. There are also a few bugs, one of which can lead to a crash(turning down the music volume to 0 and then loading a level) that

make the game feel less polished than it could have been. Lastly, the level design could have used more work since the final levels now feel more like tech demos than actual puzzles. Overall though, we are happy with what we achieved and the final result.

## Objective Measure

To measure the quality of the product we simply tested it as much as possible, trying to break something to find a flaw in the code and to fix it if it was realistically possible. One such bug we fixed had to due with looping sound. If, for example, the walking sound was playing and the user pressed escape to access the pause menu, the walking sound would still be looping in the background, even though there was no Player object. So, we had to make every single sound stopped when loading a different screen. With that said, the game still has many bugs since fixing them would require re-working huge parts of the logic and we simply did not have time for that. For example, the player is considered to be in a “grounded” state when it in contact with a static body, such as those around platforms. We did not anticipate for the code to consider the Player to be grounded when it was only touching the side of a platform, i.e. a wall. The Player can only jump when grounded, but now the player can just infinitely jump on a wall to get height. The solution would have been to create four separate rigid bodies around a platform tile to differentiate between wall and floor, but our map rendering algorithm was already done, and time was running short.

# PROJECT MANAGEMENT

| Task | Plan Date | Assigned Person | Notes |
|---|---|---|---|
| Discuss project idea | Week 1 | Everyone | Successful |
| Refine project idea | Week 2 | Everyone | Successful |
| Research Development & Creative Tools | Week 2 | Everyone | Successful |
| Basic Designs of Characters & Objects | Week 3 | Louis-Antoine | Successful |
| GUI Design | Week 3 | Nicola | Successful |
| Write Report & Create Presentation | Week 4 | Everyone | Successful |
| Present First Part | Week 5 | Everyone | Successful |
| Draw Assets in Piskel | Week 6 | Louis-Antoine | Successful |
| Research Box2D physics | Week 6 | Nicola | Successful |
| Halt Production | Week 6-10 | Covid-19 | Successful |
| Finalize GUI | Week 10-11 | Louis-Antoine | Successful |
| Implement Box2D physics | Week 11-12 | Nicola | Successful |
| Create Map Render Algorithm | Week 11-12 | Louis-Antoine | Successful |
| Animations | Week 11-12 | Mark | Successful |
| Level Design | Week 12 | Louis-Antoine and Mark | Successful |
| Objects | Week 13 | Nicola | Successful |
| Sounds/Music | Week 14 | Louis-Antoine | Successful |

| | | | |
|---|---|---|---|
| Debug/Playtest | Week 15 | Everyone | Successful |
| Export Finished Game | Week 15 | Everyone | Successful |
| Present Final Project | Week 15 | Everyone | Successful |

## CONCLUSION

All in all, when looking back at the early planning stages of our integrative project, and the final product assembled (i.e. MIR/ROR/RIM), our group believes massive progress was achieved, and that the overall outcome of the project was a success, given the limited amount of time to complete the project.

Through working on the project, we learned to work with the software, Piskel, as it was extensively used for creating the assets. Furthermore, the software supported animation creation, different layers to a sprite and can export sprites/animations in many different file formats. Luckily for us, the software was free and solely focused on pixel art aesthetics, which worked to our advantage, as we decided to pursue pixel art for our game's art design.

In addition, our group also became more familiarized with using libGDX, a Java game development framework that includes many features for graphics rendering, sound playback, particle creation and physics, all of which saved us a lot of time, as we did not have to build everything from scratch.

To conclude, our group believes we created a well-functioning project with clear and concise code, so that anyone who looks at the code can get an idea of how we created our videogame and can create their own version of it. More importantly, our group feels we have

created an enjoyable/educational, user-friendly game where people can experiment with the game's implemented physics aspects, in order to complete each level and in turn, win the game.

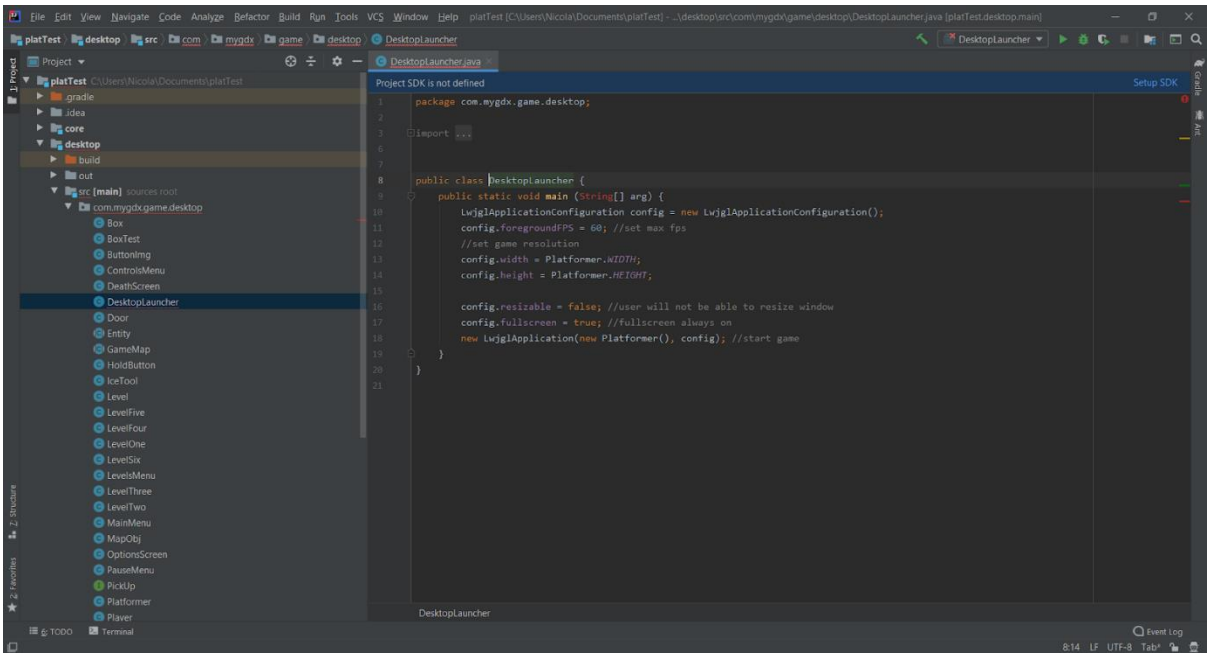
# MANUALS

---

## System Manual

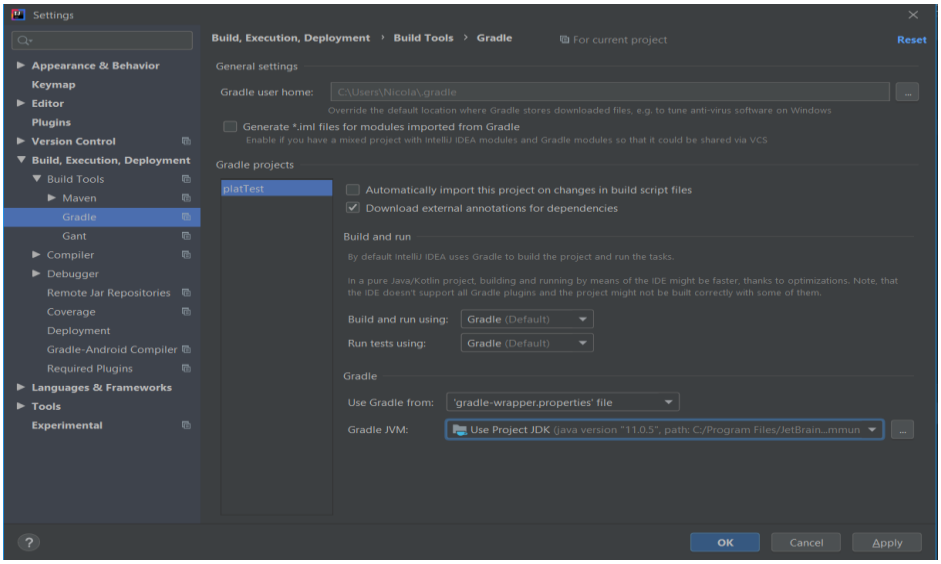
To install the project on the IntelliJ IDE

1. If not already installed, download the IntelliJ IDE community edition link
2. Download platTest folder and and extract .zip file
3. Open IntelliJ IDE and, on the top left corner, click File > Open > platTest
4. Now the project is imported in IntelliJ. To run, click desktop > src > com.mygdx.game.desktop > DesktopLauncher
5. This should give you a blue prompt at the top of the code: "Project SDK not defined"




6. Click "setup SDK" on the top right corner of the blue prompt and select 11.0.5

7. Click on File > Settings > Build, Execution, Deployment > Build Tools > Maven > Gradle and in “Gradle JVM” select “Use Project JDK” and click OK




8. Click the green triangle at the top right corner to run the project

You can now run the project and edit the classes in the project

To modify levels from the project using Tiled

1. If not already installed, download and install Tiled link
2. Open the software
3. Click “Open File” and navigate to the main “platTest”. Go to “platTest\core\assets\map” and open “TileTest.tmx”. You will now see all the tiles from the game.
4. To load a level, go to File > Open and all the level files are in the same directory as in step 3. Click Open to load a level and you can now edit the level. The controls are quite intuitive.

Notes: There are a few quirks/rules to the map-rendering algorithm that are needed to make functional levels.

1. Every single Type of tile should be present in the level.

2. The tile layer 2 should only be filled with the sky (Id=2) and should set to invisible by pressing the eye icon in the “Layers” menu.
3. The tile layer 1 is the one to be used to draw the actual map.
4. There is only one Player, IceTool, HoldButton, Switch and Door to a level, so just put one in the Tiled map editor. If there are more than one, its position in the level will move to its tile closest to the upper-right corner.
5. Always put a normal platform (id=0) at (0,0), an icy platform (id=1) to its right and leave the tile to its right blank to make sure the map rendering and IceTool function.

To modify the assets:

1. If not already installed, download and install Piskel link
2. Open Piskel, click the folder icon, select “Browse .piskel files”
3. Navigate to “platTest\core\assets\Piskel files”
4. There are four folders that organize all the types of assets, choose the asset you want to modify by opening a “.piskel” file.
5. Export the sprites at scale 1x in the appropriate folder “platTest\core\assets\”

Neat Debug Tools:

While in a level, press F3 to activate debug mode. The hitboxes on every Box2D bodies will now appear. Press C to free the camera from the Player and drag the mouse to move around the level, Press P to add a box at the mouse’s position, Press G to invert gravity at any time, Press T on a platform tile to change to freeze or unfreeze it, press B to draw an invisible static box.

## User Documentation

To run the game as an executable game file

1. If not already installed, download Java link
2. Download the Mirrorrim.jar file
3. Click on the file to run it

The different playable levels can be accessed from the “Levels” tab, the layout of the controls and the volume controls can be found in the “Options” tab, and to exit to the desktop, click the “Exit” tab.