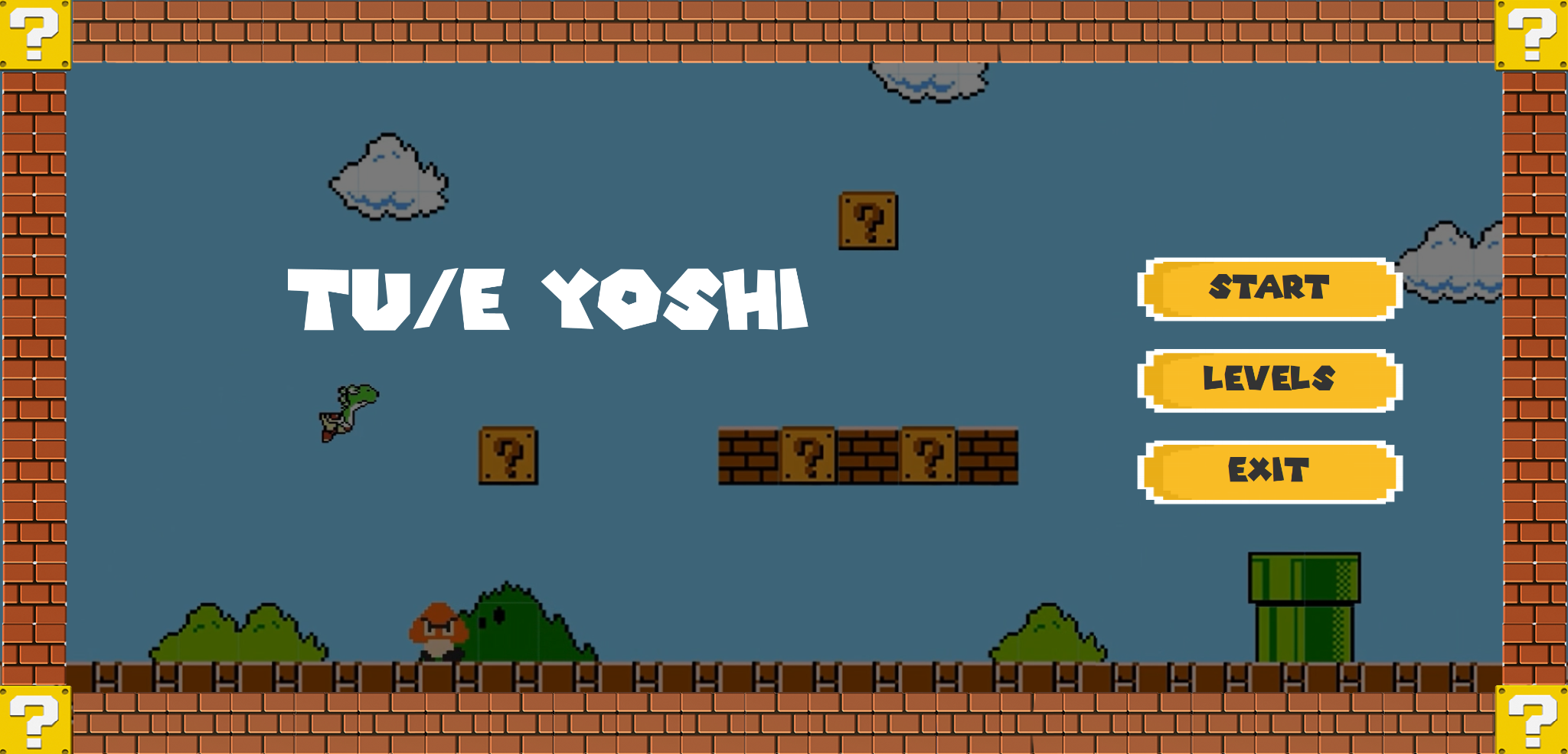
CBL **Game**

**TU/e** Yoshi

Group **26**

1. **Authors**:
   1. **Dani Stoyanova** (**1969021**)
   2. **Dimitar Vasilev** (**1974181**)
2. **Game Summary**:
   1. **Game objectives**:
      * **TU/e Yoshi** is a 2D game where the main character Yoshi jumps on platforms and enemies while avoiding their attacks. The player has to move to the right in order to complete the level and win.
   2. **Main stages of the development**:
      * Choosing the topic of the game.
      * Building the map.
      * Building the GUI.
      * Creating the different kinds of entities.
   3. **Level of complexity**

During the implementation of the game, we dealt with the following problems:

* + - Loading the map – learning how to deserialize JSON data.
    - Colliding the player with different objects – drawing hitboxes around visible objects and pre-calculating if the next movement of objects will cause collision before rendering the next frame. Learning about collision in game development.
    - Creating gravity – learning how to implement gravity in game development.
    - Creating the jumping motion of the player – learning how to work with the velocity vector of the player and synchronously change it depending on the gravity vector.
    - Creating the player movement animation – in the Player class store the different sprite stages and iterate, then depending on time. Learning about animating in game development.
    - Creating the camera movement – creating a Camera class with a position vector, and shift the map according to camera position. Learn about scene translation.
  1. **Logical and functional description**

When the application is run the Start menu is visible. If the Start button is selected the game begins with the first level. The player moves with the keyboard buttons W, A, D, Space. The player can collect coins also by hitting blocks when jumping and by killing enemies when jumping on their heads. If the player is touched by an enemy or he falls off the ground, the player dies. If the player manages to complete a level, he wins and can continue to the next level.

Everything that has to be rendered in the game is represented by the class *AbstractEntity*. There are several initialized parameters of the future objects - position, image, width, height and origin of the specified position. Then the class *AbstractCollidable* extends the latter class and adds to the list of parameters a hitbox(created using *Vector2* - a class in the app representing a two-dimensional vector), a checker whether an object is collidable and whether it is solid. From there the objects divide into two different types - objects that extend the class *AbstractCharacter* and *BasicCollisionObject.* Objects from the type *BasicCollisionObject* areobjects that need to have the physical properties of a solid object, but don't need to move or have some serious interaction with the world. The other ones also divide into three types of objects - *Player*, *Enemy* and *Coin*. All of these classes extend the class *AbstractCharacter*. There are the methods for the movement of the characters, the correction of the characters’ position if a collision has occurred, and the selection of the sprites of the characters which happens according to their current movement state in order to get motion animation. The

Collisions class is responsible for managing collisions between collidable objects, and it notifies objects whenever they interact in a certain way, so that they can take actions based on the type of collision. The Camera class is used to create an imaginary camera object which helps to translate the scene according to its position.

The GUI is handled in the package *game*. There are several panels which are used to create the different functionalities of the application. The *MarioGame* class is responsible for building the objects in the application. *MarioPanel* is responsible for rendering the entities and setting the camera object. *StartMenuPanel*, *LevelMenuPanel*, *DiePanel* and *WinPanel* hold different buttons which make the connections between the different panels.

In the package *types* are stored different classes which describe types of objects used in the project mechanics. The *HitBox* class represents the hit box of an entity and checks for collisions. The *Level* class is responsible for loading the level and managing the resources of that specific level. The classes *MapDescriptor, MapLayer, MapLayerObject, TileSetDescriptor, TilesetEntry* are responsible for reading the JSON files which contain the set of tiles for the map and the map itself. The class *MarioFont* initializes the font used in the game. The class *Sound* contains a number of different sounds used in the game and handles their loading. It is initialized as a singleton class with the same instance across all usages.

* 1. **Implementation**

The game is implemented using Java for the mechanics, movements and collisions, Swing for the GUI, Tiled for the map creation and Gradle for project automation.

* 1. **Application description**

To run the application you simply need to run Main.java. From there the application should start normally.

A GitHub repository can be found [here](https://github.com/DimitarKV/Mario)

* 1. **Conclusion**

The application is a helpful method of learning new techniques in coding and developing a space orientational thinking about a problem that is close to the real world. It is partially developed to real world physics parameters such as gravity and velocity.

1. **Learning goals – topics of choice**:
   * + Learning how to use Gradle and build the application with it.
       - During the development of the application we faced the need to use some third-party libraries for deserializing JSON documents to POJOs. For this reason we turned to the GSON library developed by Google. By using Gradle, importing the library was as simple as just writing a single line of code in the Gradle configuration file.
     + Game design, how to design game mechanics and render objects on the screen, based on user input and in-game physics laws, combining the two, we achieved a smooth running application with a real-world feel.
       - While we were developing the application the main stopping factor were collisions. We tried implementing multiple types of collision detection, but figured out the most effective is the AABB collision detection algorithm.