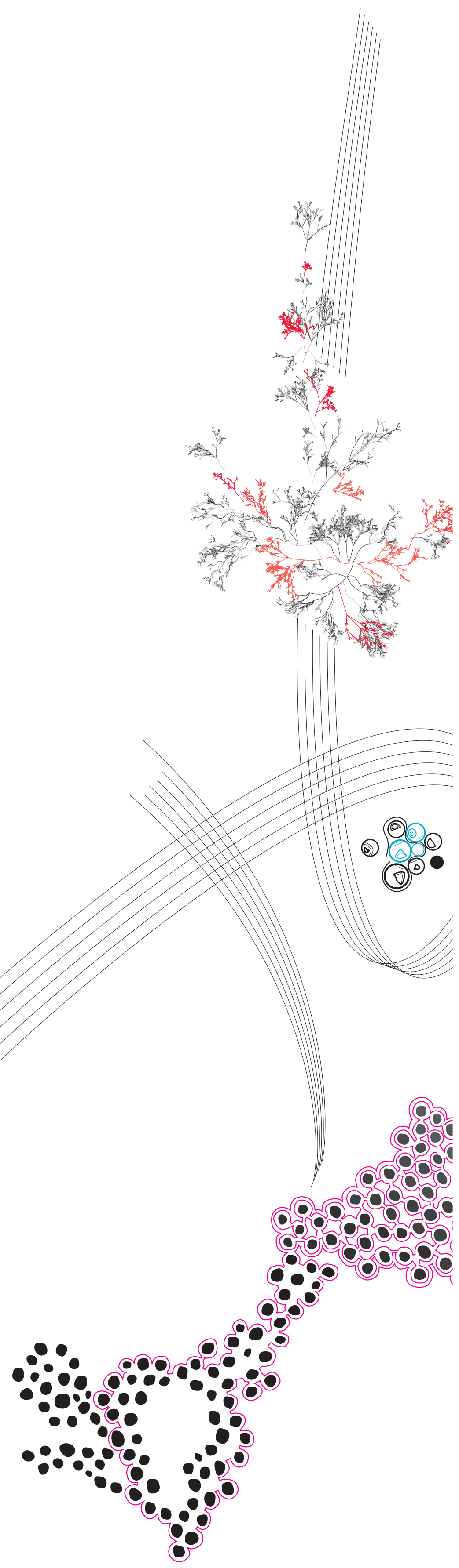
BSc Computer Science
Design Project Report



Cubewear Interactive Game Mindhash

Faisal Nizamudeen
Bram Otte
Mustafa Bilgin
Toghrul Garalov
Vladimir Halchenko

Supervisor: Nacir Bouali

April, 2023

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Chapter 1

Introduction

1.1 Module Background

The Design Project is a crucial module in the BSc Technical Computer Science program at the University of Twente, serving as a culmination of our studies. During this module, we were tasked with designing a technical system for a real company. The learning objectives of this module(taken from the project manual) are as follows:

- Collect functional and quality requirements in cooperation with a client and prioritize them
- Methodically design a system that meets the requirements, using relevant knowledge, techniques and tools
- Turn the design into a working prototype
- Formulate a test plan according to which the prototype is tested
- Document all phases in the design trajectory
- Justify choices and coordinate them with the client
- Write a project proposal and a project plan and organize the project accordingly
- Work in a team: plan activities, distribute responsibilities, interact in a constructive way
- Indicate consequences of system and design choices (ethical, societal, organizational)
- Indicate follow-up steps for the development of the system

Our team was fortunate to be assigned the project Cubewear Interactive Game, provided by Mindhash. This project was our top choice, and we were excited to have the opportunity to work on such an innovative and challenging assignment. We wanted to work on this project as we felt it had a nice scope with a lot of different interesting elements that we could work on.

1.2 Client Information

Mindhash is a small startup based in Hengelo which focuses on projects that combine hardware and software. Mindhash specializes in building custom software solutions for businesses of all sizes, from startups to large enterprises. Their services include an idea incubator where they use their technical knowledge to identify opportunities and challenges, UX discovery workshops to help establish customer value propositions, fast prototyping to validate ideas, and the development of internet-of-things products by utilizing innovative technologies. With a team of experienced developers, the company has worked on a range of software development projects, including IoT solutions, web applications, mobile applications, and complex software systems.

One of their current follow-up projects is called "Cubewear", which is a unique hardware and software system designed to display information and help navigate people around a building. Specifically, the cubes hang inside the hallway of the buildings, and their displays are currently used for two purposes: display basic information like the time, weather etc. and show navigation signs to help people find certain offices.

1.3 Goal

Mindhash has 12 LED Cubes spread throughout their office building in High Tech Systems Park. These cubes have five 64 by 64 bit pixel displays. Currently, these cubes are used for navigation throughout the building where Mindhash is based(there are over 50 companies with offices in this location). There is a touchscreen located at the entrance of the building with which you can select a company you would like to navigate to.

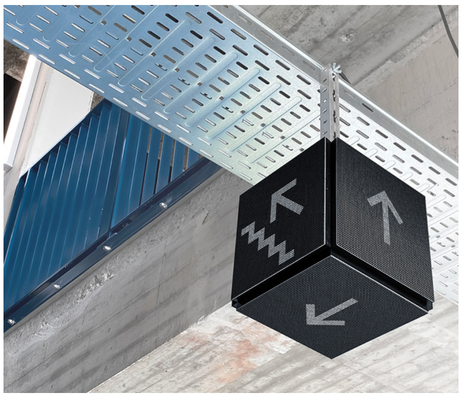


FIGURE 1.1: Cubes in Mindhash Office that are currently used for navigation

The company wants to give another purpose to these cubes. This is the basis for the project we will be working on. They want to implement several arcade-style mini-games that can be played on the cube and make use of its unique display layout. For example, think of how a regular snake game could be made more interesting by having the snake flow over to different sides of the cube. A mobile phone application should act as a controller for the game.

The motivation behind this project is primarily for the employees of the office space to have something fun to do during their free time away from their work. Due to this, the company also wants to introduce a competitive element where employees of different companies can compete with each other. To facilitate this, Mindhash would also like there to be a leaderboard page accessible which can be accessed by employees to see their personal high scores as well as overall high scores for the different games.

A formal list of requirements is elaborated on in the next section.

1.4 Stakeholders

There are several stakeholders involved in the development and eventual use of this product:

- Mindhash BV
- University of Twente Design Project students (us)
- Employees who work in High Tech Systems Park
- External visitors and guests to High Tech Systems Park

1.5 Report Outline

This report consists of 10 parts. The first section (current section) provides an overview of the project, the client, the goals of the project, and the stakeholders. The second section elaborates on all the functional and non-functional requirements of the project. In the third and fourth sections, design choices and technical information are mentioned. Specifically, the third section includes relevant diagrams to represent the overall architecture of the system and the fourth section describes the individual components of the system. The fifth is about communication between the team, supervisor, and client. In the sixth section, the final product and the implementations of individual components are described. The seventh section delves into testing for specific components and the overall system. In the eighth and ninth sections, information regarding the possible additions to the current project and a reflection on the module has been discussed. The final section serves as a conclusion to the report.

Chapter 2

Requirement Analysis

2.1 Functional Requirements

2.1.1 Must have

- The user must be able to choose any available game in the cube.
- An admin or user must be able to view all high scores.
- The user must be able to select a game to play on a specific cube.
- The user must be able to interact with the game using a mobile application on their phone.
- The user must be able to log in to have a username on the mobile app that will be associated with their high scores of the cube's mini-games, which are displayed on the leaderboard page.
- The cube must be able to provide minimum of 4 distinct mini-games that a user can select and play on it.

2.1.2 Should have

- The user should be able to connect to the cube with a mobile app via Bluetooth.
- The user should be able to look up the scores of his games on the leaderboards page.
- The user should be able to see an error message if there are any issues with connecting to a cube or starting a game.

2.1.3 Could have

- The admin could be able to edit (add/remove) any high score.
- An admin could be able to enable or disable any game on any cube.

2.2 Non-Functional Requirements

2.2.1 Functional suitability

- 90% of users should be able to infer that a game can be played on the cube after 15 minutes of interacting with the system.

- The game must always correctly and accurately compute the score of the player after playing a game

2.2.2 Performance efficiency

- The games on the cube should have a consistent framerate (depends on what game, but any game with smooth animations should run at least 30 fps).
- The games on the cube should have minimal latency on inputs, a maximum of 100ms.
- At Least 90% of users should not find it annoying to look at the cubes (for example due to flicker).

2.2.3 Availability

- The database must be reachable 99% of the time.

2.2.4 Compatibility

- The app should be able to run on Android.

2.2.5 Usability

- 95% of users must find the interface visually appealing, user-friendly and easy to use

2.2.6 Reliability

- The WebGUI and Cubes should be able to recover from a failure and resume normal operation within 2 minutes.
- The database must make backups at regular intervals

2.2.7 Scalability

- The backend server will be able to support at least 12 cubes simultaneously without significant degradation of performance.

2.2.8 Security

- The backend should be resistant to Command Injections.
- The admin credentials should be hashed with salt and pepper.

2.2.9 Maintainability

- The codebase for the system shall be well-documented and modular with a clear separation of concerns to allow easy maintenance and updates.

In conclusion, the functional and non-functional requirements outlined in this chapter provide clear guidelines for the development of components for the Cubewear project. The functional requirements detail the necessary features and functionalities that the project must possess to meet the needs of its users. Meanwhile, the non-functional requirements highlight the performance, usability, reliability, and security standards, among others of the system components involved in the project. In the next chapters, we will discuss in more detail how the requirements are met and what steps are taken to ensure that all the system components (the cube, the mobile app, and the web page) are functional and user-friendly.

Chapter 3

System design

This section presents the design documents of the system. In the diagrams, the emphasis is on how communication with the games that are in the cube takes place.

3.1 Data Flow Diagram

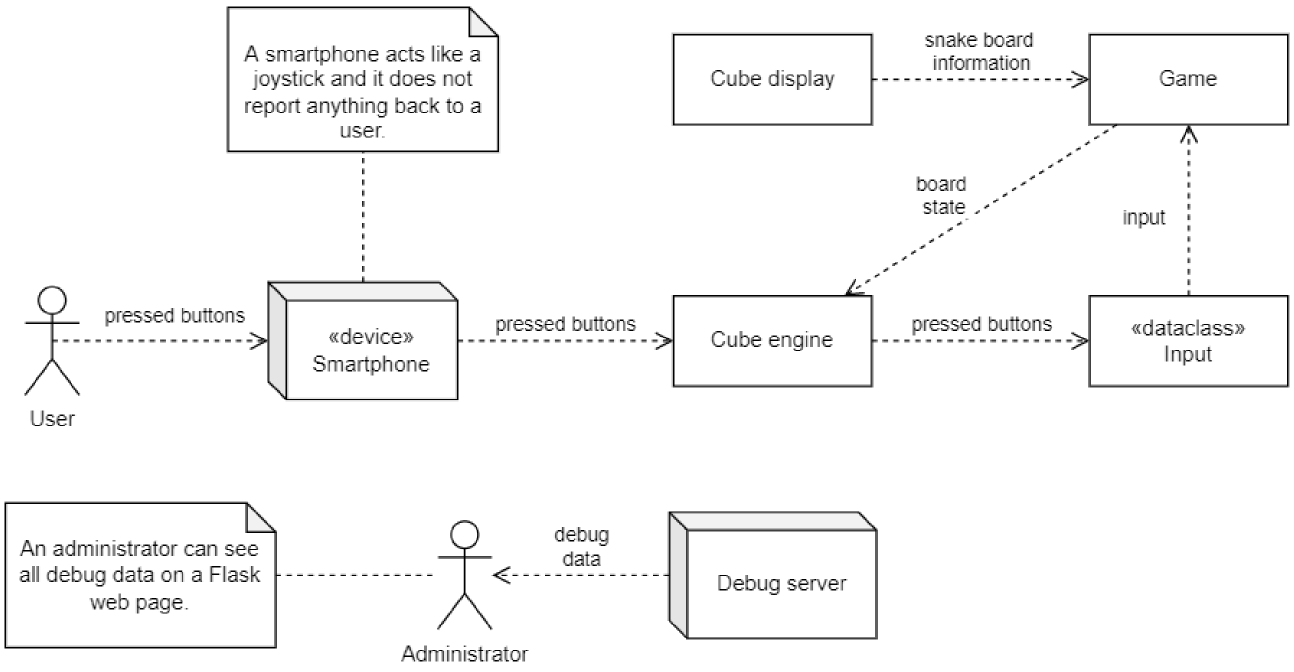


FIGURE 3.1: Data flow diagram

The accent in the data flow diagram is done on the flow of various data components between various devices of the system. The data flows from the smartphone to the cube in one way; there is no need to make any explicit answer to a smartphone since all actions of a user are immediately observable on the cube. The availability of the cube is monitored with acknowledgement packages.

There is a need to consider how the user input is processed in the cube in more detail. The software on the cube has a specific "Input" data class which stores the following public Button fields: "up", "down", "left", "right", "action", "pause". Each Button field represents an object which simulates the behaviour of the mobile device which acts like a joystick. "Input" has the following public float fields: "joyx", "joyy". The Button variables are set to default values, and the float variables are set to 0. This data class is constantly monitored by all software on the cube. If a user makes some action, and the cube receives the request to execute this action, the app protocol puts appropriate values in the appropriate Button fields and, if the joystick is involved, values describing the joystick position. The software on the cube acts in the "busy wait" and it checks each clock cycle the presence of the commands in terms of Button changed states and non-zero joystick positions. If there are actions to execute, the software handles input in this clock cycle and resets Button fields to default values and float fields to zeros. In case there are multiple messages arriving in the cubes, the Bluetooth client on the cube side takes care of them and puts requests in the queue. Each clock cycles the Bluetooth client moves each request to the "Input" data class which is processed by the cube in the next clock cycle.

The design of the games is done in such a way that the cube engine does not have to monitor the game. Instead, each game reports to the cube whether the game is over or it continues. This design decision reduces the complexity of the "busy wait".

There is a very handy debug server which is used to test various aspects of the connectivity to the cube and report technical issues to an administrator. It helps significantly while resolving technical issues related to connectivity and processing user input on the cube. This is done on the local web page which is served by Flask.

3.2 Sequence Diagram

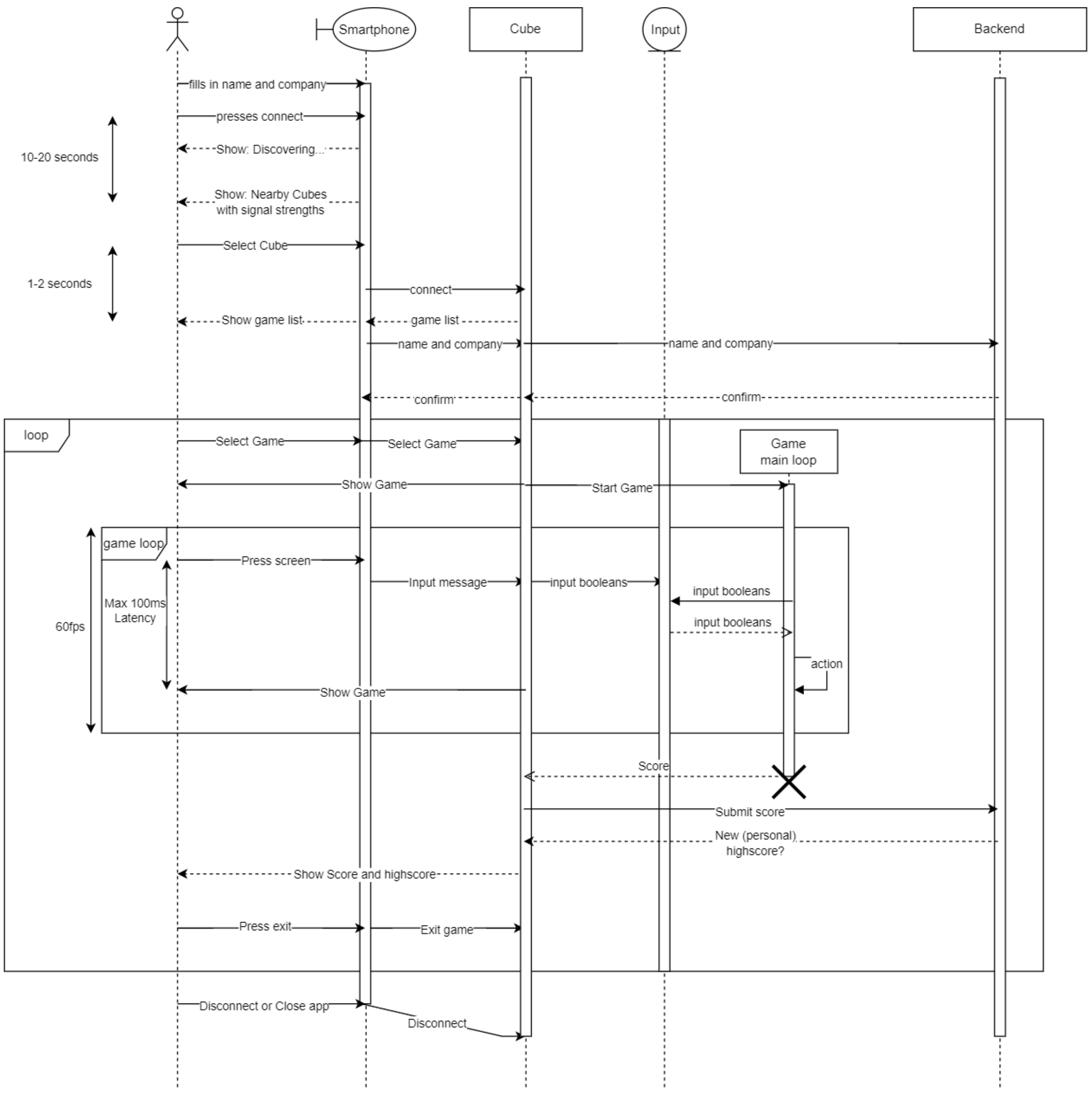


Figure 3.2: Sequence diagram

The accent in the sequence diagram is done on the communication between a cube and a user device via Bluetooth. An interval of 10-20 seconds during the stage of Bluetooth pairing with a cube is immediately thrown into the eyes. Such a long time interval is explained by the fact that the library we use does not display the available devices during the scan. In the beginning, it is necessary to scan all devices, and only when the scan is completed, the library returns a list of devices.

The next thing to note in the diagram is the game selection loop. It is indicated in the diagram that the game object is created after the selection of the game and then it is destroyed at the game end. The reason is that the game each time is created with default parameters, and it does not use any data from previous games. It means that the game does not adjust its behaviour based on user data, history, or other factors.

It is indicated as a time interval of 60 fps in the diagram. It is done because pygame clock speed depends on frames per second, and changing this value leads to a different game speed.

The game loop needs to be elaborated in more detail. As described above, the user input is read using the "Input" class. A smartphone sends a sequence of input messages which are moved to the "Input" class in terms of booleans. "Input" consists of Buttons, and each Button consists of multiple booleans describing its state. These booleans are read by a game to make an appropriate action. 100 ms latency is the performance efficiency requirement.

3.3 Class Diagrams - Game

3.3.1 Class diagrams structure

This section contains multiple class diagrams of the system. Before explaining the diagrams, we need to make a few comments.

In the classes, we did not include the self parameter since it is obvious that every method accepts the object by default. Getters and setters of objects are implemented in the pure Pythonic way with a @property decorator. If the attribute is labelled with «get, set» it means there exists a getter and a setter for this field.

Each game has a main file which contains a main loop and the properties which control in real time how the game is running in the main loop. Also, each game has a configuration file which controls how any game is running, and the properties in the configuration file are rarely changed.

All games are connected to the cube engine in a similar way. There is not much difference in the way how a particular game is connected to the cube. Instead of adopting a particular mechanic by changing the connection to the cube, we made changes in the games themselves.

3.3.2 Snake

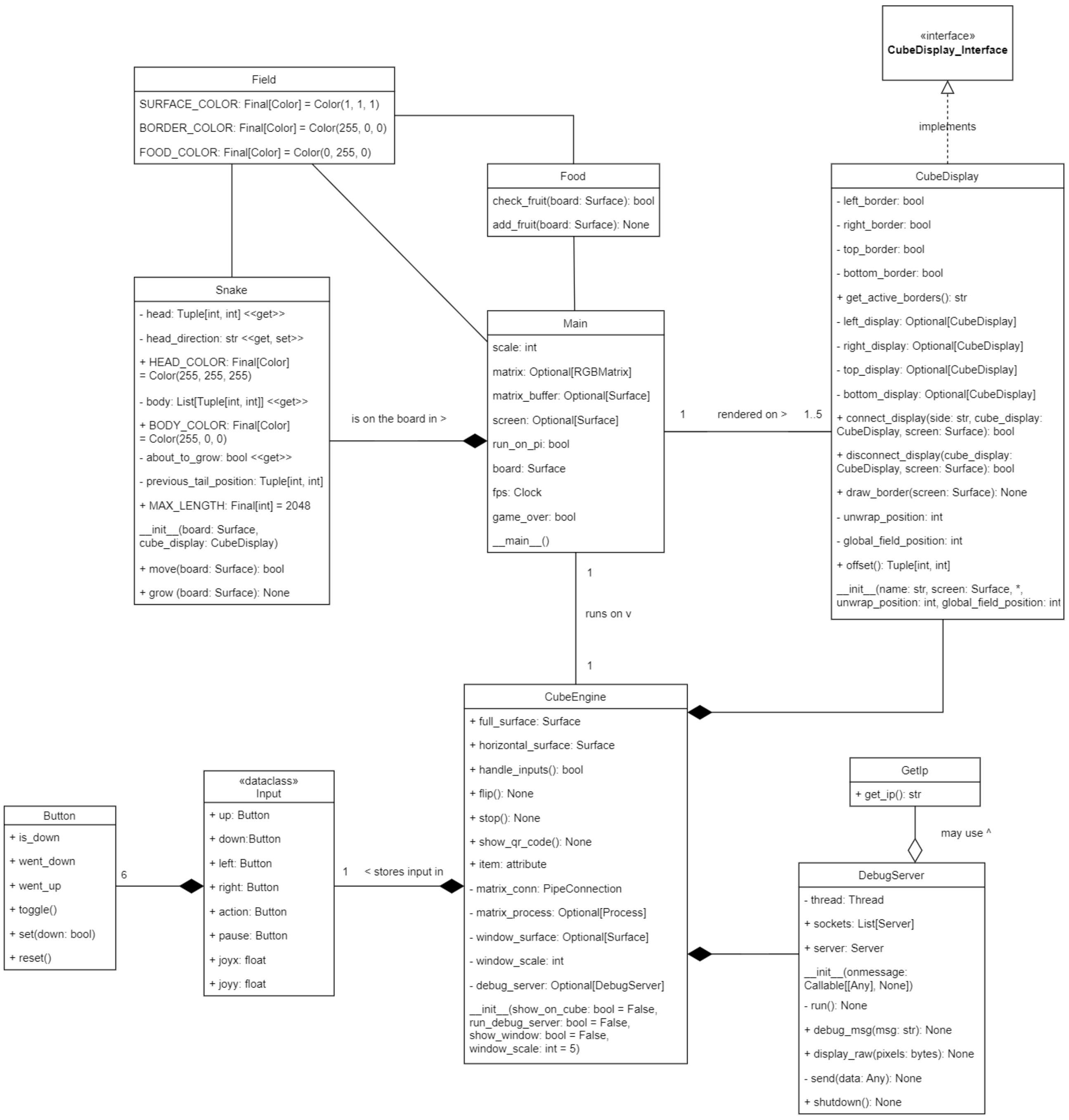

«interface»
CubeDisplay_Interface
implements
Field
SURFACE_COLOR: Final[Color] = Color(1, 1, 1)
BORDER_COLOR: Final[Color] = Color(255, 0, 0)
FOOD_COLOR: Final[Color] = Color(0, 255, 0)
Food
check_fruit(board: Surface): bool
add_fruit(board: Surface): None
CubeDisplay
- left_border: bool
- right_border: bool
- top_border: bool
- bottom_border: bool
+ get_active_borders(): str
- left_display: Optional[CubeDisplay]
- right_display: Optional[CubeDisplay]
- top_display: Optional[CubeDisplay]
- bottom_display: Optional[CubeDisplay]
+ connect_display(side: str, cube_display: CubeDisplay, screen: Surface): bool
+ disconnect_display(cube_display: CubeDisplay, screen: Surface): bool
+ draw_border(screen: Surface): None
- unwrap_position: int
- global_field_position: int
+ offset(): Tuple[int, int]
__init__(name: str, screen: Surface, *, unwrap_position: int, global_field_position: int
Snake
- head: Tuple[int, int] <<get>>
- head_direction: str <<get, set>>
+ HEAD_COLOR: Final[Color] = Color(255, 255, 255)
- body: List[Tuple[int, int]] <<get>>
+ BODY_COLOR: Final[Color] = Color(255, 0, 0)
- about_to_grow: bool <<get>>
- previous_tail_position: Tuple[int, int]
+ MAX_LENGTH: Final[int] = 2048
__init__(board: Surface, cube_display: CubeDisplay)
+ move(board: Surface): bool
+ grow (board: Surface): None
is on the board in >
Main
scale: int
matrix: Optional[RGBMatrix]
matrix_buffer: Optional[Surface]
screen: Optional[Surface]
run_on_pi: bool
board: Surface
fps: Clock
game_over: bool
__main__()
1
rendered on >
1..5
1
runs on v
1
CubeEngine
+ full_surface: Surface
+ horizontal_surface: Surface
+ handle_inputs(): bool
+ flip(): None
+ stop(): None
+ show_qr_code(): None
+ item: attribute
- matrix_conn: PipeConnection
- matrix_process: Optional[Process]
- window_surface: Optional[Surface]
- window_scale: int
- debug_server: Optional[DebugServer]
__init__(show_on_cube: bool = False, run_debug_server: bool = False, show_window: bool = False, window_scale: int = 5)
GetIp
+ get_ip(): str
may use ^
DebugServer
- thread: Thread
+ sockets: List[Server]
+ server: Server
__init__(onmessage: Callable[[Any], None])
- run(): None
+ debug_msg(msg: str): None
+ display_raw(pixels: bytes): None
- send(data: Any): None
+ shutdown(): None
Button
+ is_down
+ went_down
+ went_up
+ toggle()
+ set(down: bool)
+ reset()
6
«dataclass»
Input
+ up: Button
+ down:Button
+ left: Button
+ right: Button
+ action: Button
+ pause: Button
+ joyx: float
+ joyy: float
1
< stores input in

Figure 3.3: Snake class diagram

The snake game class diagram is very simple. The file main.py contains the main loop and the board which contains all game elements. There is a snake on the board which moves and reports about its state. There is no class for a food object because of the low complexity: the food just represents a fruit pixel on the board and it does nothing, so it has no properties. The game checks whether the food pixel exists with check_fruit(), and if it is consumed, it adds a new food pixel with add_fruit(). The configuration file field.py contains the settings about the colours of the objects used in the game.

3.3.3 Space Invaders

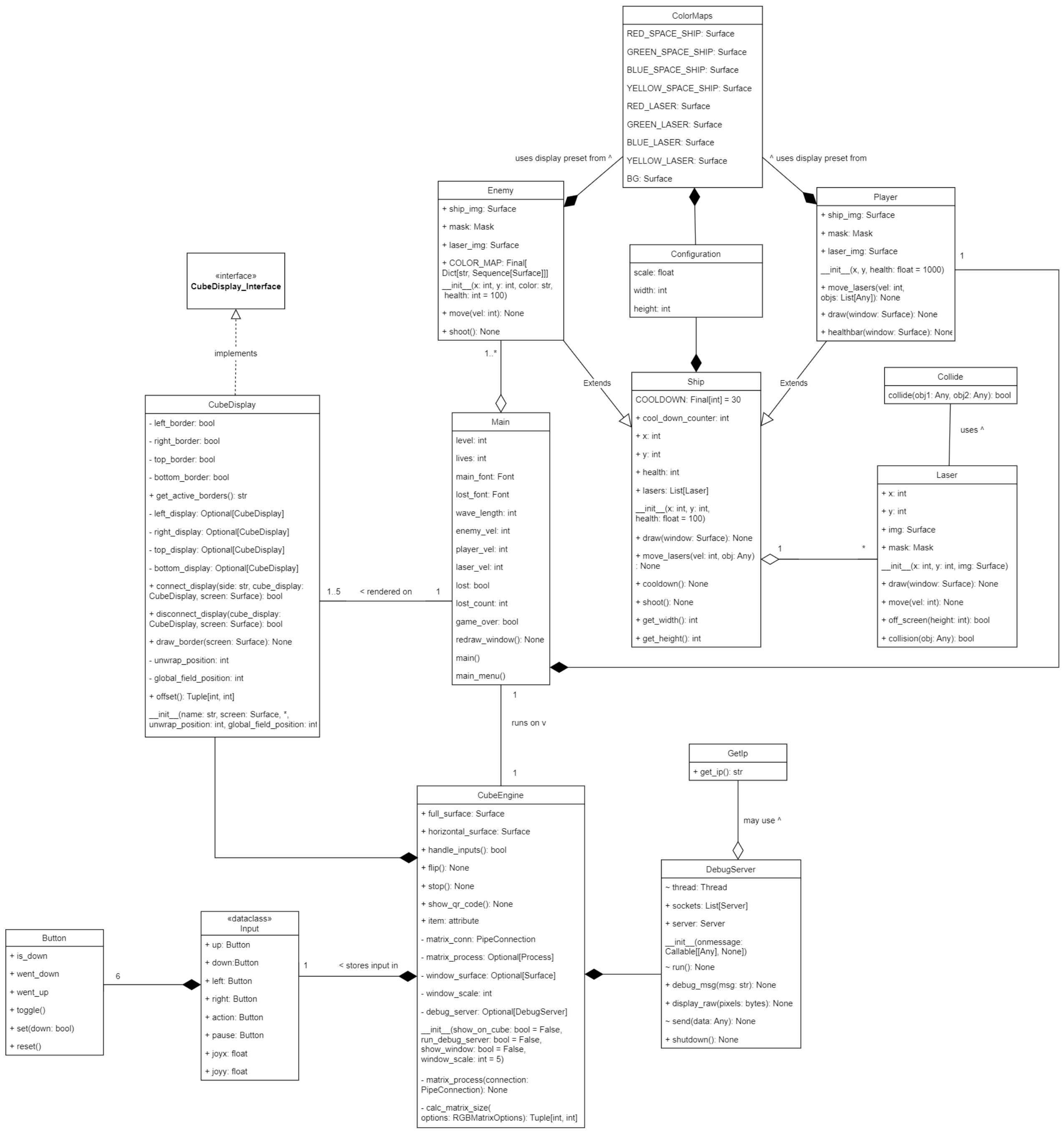

ColorMaps
RED_SPACE_SHIP: Surface
GREEN_SPACE_SHIP: Surface
BLUE_SPACE_SHIP: Surface
YELLOW_SPACE_SHIP: Surface
RED_LASER: Surface
GREEN_LASER: Surface
BLUE_LASER: Surface
YELLOW_LASER: Surface
BG: Surface
uses display preset from ^
^ uses display preset from
Enemy
+ ship_img: Surface
+ mask: Mask
+ laser_img: Surface
+ COLOR_MAP: Final[Dict[str, Sequence[Surface]]]
__init__(x: int, y: int, color: str, health: int = 100)
+ move(vel: int): None
+ shoot(): None
Configuration
scale: float
width: int
height: int
Player
+ ship_img: Surface
+ mask: Mask
+ laser_img: Surface
__init__(x, y, health: float = 1000)
+ move_lasers(vel: int, objs: List[Any]): None
+ draw(window: Surface): None
+ healthbar(window: Surface): None
1
«interface»
CubeDisplay_Interface
implements
1..*
Extends
Extends
Ship
COOLDOWN: Final[int] = 30
+ cool_down_counter: int
+ x: int
+ y: int
+ health: int
+ lasers: List[Laser]
__init__(x: int, y: int, health: float = 100)
+ draw(window: Surface): None
+ move_lasers(vel: int, obj: Any) : None
+ cooldown(): None
+ shoot(): None
+ get_width(): int
+ get_height(): int
Collide
collide(obj1: Any, obj2: Any): bool
uses ^
Laser
+ x: int
+ y: int
+ img: Surface
+ mask: Mask
__init__(x: int, y: int, img: Surface)
+ draw(window: Surface): None
+ move(vel: int): None
+ off_screen(height: int): bool
+ collision(obj: Any): bool
1
*
CubeDisplay
- left_border: bool
- right_border: bool
- top_border: bool
- bottom_border: bool
+ get_active_borders(): str
- left_display: Optional[CubeDisplay]
- right_display: Optional[CubeDisplay]
- top_display: Optional[CubeDisplay]
- bottom_display: Optional[CubeDisplay]
+ connect_display(side: str, cube_display: CubeDisplay, screen: Surface): bool
+ disconnect_display(cube_display: CubeDisplay, screen: Surface): bool
+ draw_border(screen: Surface): None
- unwrap_position: int
- global_field_position: int
+ offset(): Tuple[int, int]
__init__(name: str, screen: Surface, *, unwrap_position: int, global_field_position: int
1..5
< rendered on
1
Main
level: int
lives: int
main_font: Font
lost_font: Font
wave_length: int
enemy_vel: int
player_vel: int
laser_vel: int
lost: bool
lost_count: int
game_over: bool
redraw_window(): None
main()
main_menu()
1
runs on v
1
GetIp
+ get_ip(): str
may use ^
CubeEngine
+ full_surface: Surface
+ horizontal_surface: Surface
+ handle_inputs(): bool
+ flip(): None
+ stop(): None
+ show_qr_code(): None
+ item: attribute
- matrix_conn: PipeConnection
- matrix_process: Optional[Process]
- window_surface: Optional[Surface]
- window_scale: int
- debug_server: Optional[DebugServer]
__init__(show_on_cube: bool = False, run_debug_server: bool = False, show_window: bool = False, window_scale: int = 5)
- matrix_process(connection: PipeConnection): None
- calc_matrix_size(options: RGBMatrixOptions): Tuple[int, int]
DebugServer
~ thread: Thread
+ sockets: List[Server]
+ server: Server
__init__(onmessage: Callable[[Any], None])
~ run(): None
+ debug_msg(msg: str): None
+ display_raw(pixels: bytes): None
~ send(data: Any): None
+ shutdown(): None
Button
+ is_down
+ went_down
+ went_up
+ toggle()
+ set(down: bool)
+ reset()
6
«dataclass»
Input
+ up: Button
+ down:Button
+ left: Button
+ right: Button
+ action: Button
+ pause: Button
+ joyx: float
+ joyy: float
1
< stores input in

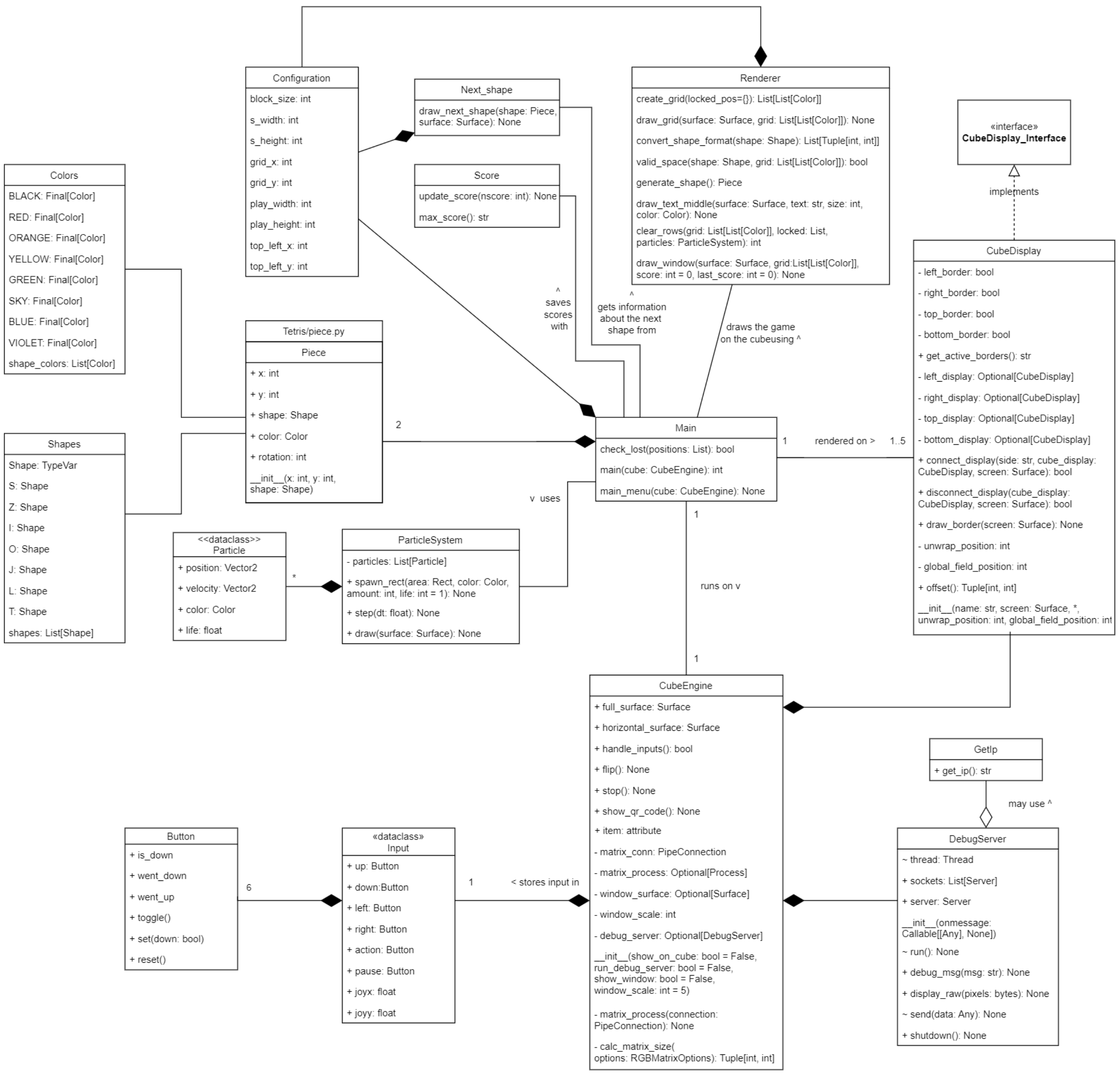
FIGURE 3.4: Space Invaders class diagram

The space invaders class diagram is more complicated. The file main.py contains the main loop, the lives of the space base and the speed with which objects move on the screen. There are two types of ships: a player ship and an enemy ship which have similarities and differences. We have implemented the general Ship class which is never used and the player ship class and the enemy ship class which extend it and implement the differences between a ship a player controls and alien ships controlled by a cube.

Objects in the game are using images defined in color_maps.py. Each object has a mask: objects collide if their non-transparent parts of the image collide which makes the game more realistic.

Every ship has zero to many Lasers. However, these are not laser cannons installed on the space ships, but actual lasers which fly and attack. A laser moves itself, without being directed by a ship.

3.3.4 Tetris



Configuration
block_size: int
s_width: int
s_height: int
grid_x: int
grid_y: int
play_width: int
play_height: int
top_left_x: int
top_left_y: int
Next_shape
draw_next_shape(shape: Piece, surface: Surface): None
Score
update_score(nscore: int): None
max_score(): str
Renderer
create_grid(locked_pos={}): List[List[Color]]
draw_grid(surface: Surface, grid: List[List[Color]]): None
convert_shape_format(shape: Shape): List[Tuple[int, int]]
valid_space(shape: Shape, grid: List[List[Color]]): bool
generate_shape(): Piece
draw_text_middle(surface: Surface, text: str, size: int, color: Color): None
clear_rows(grid: List[List[Color]], locked: List, particles: ParticleSystem): int
draw_window(surface: Surface, grid:List[List[Color]], score: int = 0, last_score: int = 0): None
«interface»
CubeDisplay_Interface
implements
Colors
BLACK: Final[Color]
RED: Final[Color]
ORANGE: Final[Color]
YELLOW: Final[Color]
GREEN: Final[Color]
SKY: Final[Color]
BLUE: Final[Color]
VIOLET: Final[Color]
shape_colors: List[Color]
^ saves scores with
^ gets information about the next shape from
draws the game on the cubeusing ^
Tetris/piece.py
Piece
+ x: int
+ y: int
+ shape: Shape
+ color: Color
+ rotation: int
__init__(x: int, y: int, shape: Shape)
2
Main
check_lost(positions: List): bool
main(cube: CubeEngine): int
main_menu(cube: CubeEngine): None
1
rendered on >
1..5
v uses
1
runs on v
1
CubeDisplay
- left_border: bool
- right_border: bool
- top_border: bool
- bottom_border: bool
+ get_active_borders(): str
- left_display: Optional[CubeDisplay]
- right_display: Optional[CubeDisplay]
- top_display: Optional[CubeDisplay]
- bottom_display: Optional[CubeDisplay]
+ connect_display(side: str, cube_display: CubeDisplay, screen: Surface): bool
+ disconnect_display(cube_display: CubeDisplay, screen: Surface): bool
+ draw_border(screen: Surface): None
- unwrap_position: int
- global_field_position: int
+ offset(): Tuple[int, int]
__init__(name: str, screen: Surface, *, unwrap_position: int, global_field_position: int
Shapes
Shape: TypeVar
S: Shape
Z: Shape
I: Shape
O: Shape
J: Shape
L: Shape
T: Shape
shapes: List[Shape]
<<dataclass>>
Particle
+ position: Vector2
+ velocity: Vector2
+ color: Color
+ life: float
*
ParticleSystem
- particles: List[Particle]
+ spawn_rect(area: Rect, color: Color, amount: int, life: int = 1): None
+ step(dt: float): None
+ draw(surface: Surface): None
CubeEngine
+ full_surface: Surface
+ horizontal_surface: Surface
+ handle_inputs(): bool
+ flip(): None
+ stop(): None
+ show_qr_code(): None
+ item: attribute
- matrix_conn: PipeConnection
- matrix_process: Optional[Process]
- window_surface: Optional[Surface]
- window_scale: int
- debug_server: Optional[DebugServer]
__init__(show_on_cube: bool = False, run_debug_server: bool = False, show_window: bool = False, window_scale: int = 5)
- matrix_process(connection: PipeConnection): None
- calc_matrix_size(options: RGBMatrixOptions): Tuple[int, int]
GetIp
+ get_ip(): str
may use ^
Button
+ is_down
+ went_down
+ went_up
+ toggle()
+ set(down: bool)
+ reset()
6
«dataclass»
Input
+ up: Button
+ down:Button
+ left: Button
+ right: Button
+ action: Button
+ pause: Button
+ joyx: float
+ joyy: float
1
< stores input in
DebugServer
~ thread: Thread
+ sockets: List[Server]
+ server: Server
__init__(onmessage: Callable[[Any], None])
~ run(): None
+ debug_msg(msg: str): None
+ display_raw(pixels: bytes): None
~ send(data: Any): None
+ shutdown(): None

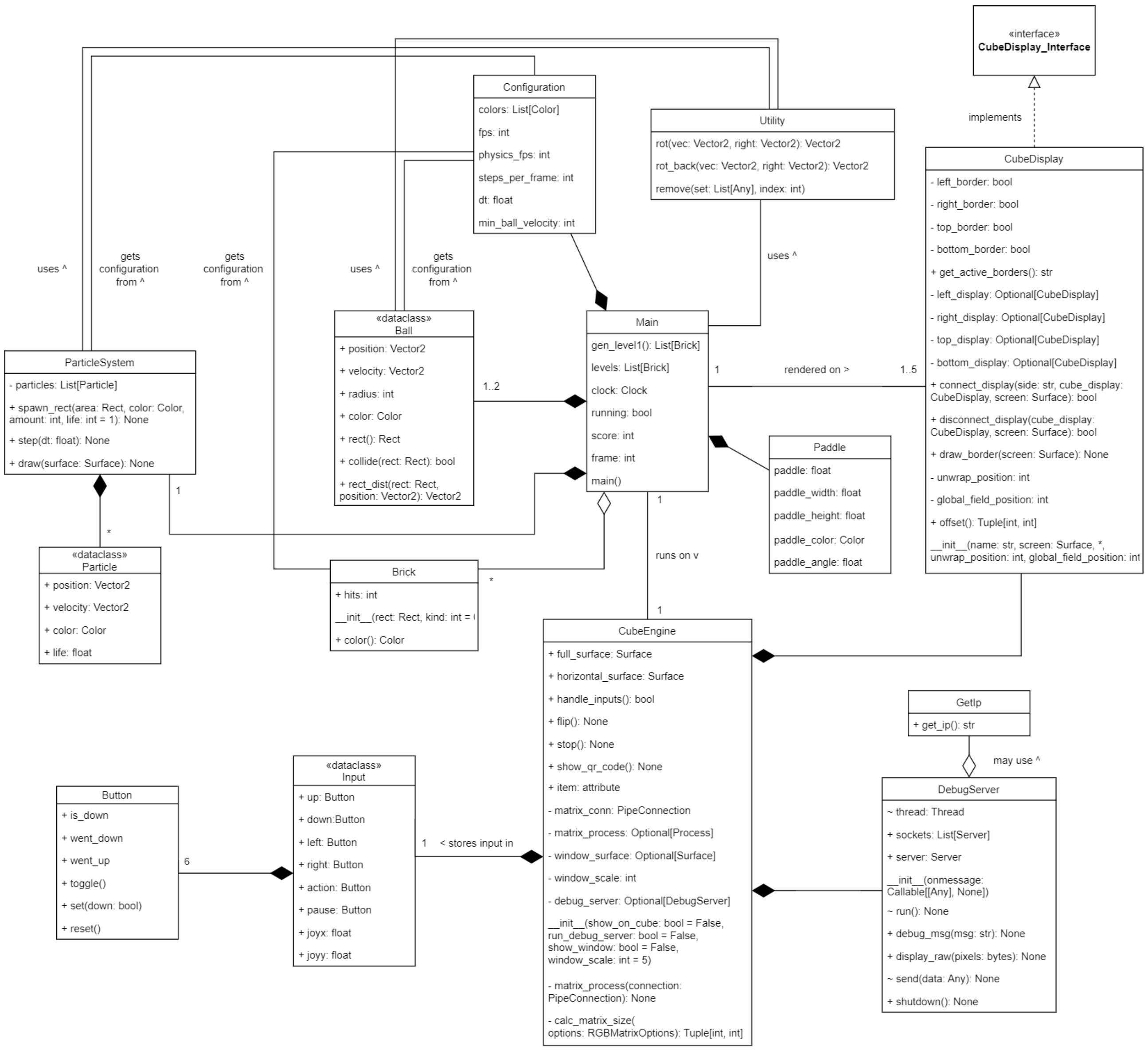
FIGURE 3.5: Tetris class diagram

The Tetris class diagram is big but simple. The main game happens in the main.py file after a user selects "Tetris" from the main menu. The game part can be visually divided into three parts: how the game is organized, how the game is rendered on the cube display, how the game saves scores.

One of the essential aspects of the game's logic involves the definition of shapes. The shapes are determined by a sequence of two-dimensional lists that specify how each shape looks in all rotations. This representation is visually convenient for debugging purposes as it allows for easy visual identification of shape structures. While it is easy to understand for a human, it is not obvious for a machine. That is why we convert this representation the shape in the coordinates of pygame with convert_shape_format() in renderer.py.

Notably, the game always has two pieces prepared for gameplay. While one piece falls onto the grid, the game has already generated the next piece, which is displayed on the game screen alongside the current score. The piece can take one of the shapes specified in shapes.py and one of the colors specified in the colors.py.

3.3.5 Arkanoid



«interface»
CubeDisplay_Interface
implements
Configuration
colors: List[Color]
fps: int
physics_fps: int
steps_per_frame: int
dt: float
min_ball_velocity: int
Utility
rot(vec: Vector2, right: Vector2): Vector2
rot_back(vec: Vector2, right: Vector2): Vector2
remove(set: List[Any], index: int)
CubeDisplay
- left_border: bool
- right_border: bool
- top_border: bool
- bottom_border: bool
+ get_active_borders(): str
- left_display: Optional[CubeDisplay]
- right_display: Optional[CubeDisplay]
- top_display: Optional[CubeDisplay]
- bottom_display: Optional[CubeDisplay]
+ connect_display(side: str, cube_display: CubeDisplay, screen: Surface): bool
+ disconnect_display(cube_display: CubeDisplay, screen: Surface): bool
+ draw_border(screen: Surface): None
- unwrap_position: int
- global_field_position: int
+ offset(): Tuple[int, int]
__init__(name: str, screen: Surface, *, unwrap_position: int, global_field_position: int
uses ^
gets configuration from ^
gets configuration from ^
uses ^
gets configuration from ^
uses ^
«dataclass»
Ball
+ position: Vector2
+ velocity: Vector2
+ radius: int
+ color: Color
+ rect(): Rect
+ collide(rect: Rect): bool
+ rect_dist(rect: Rect, position: Vector2): Vector2
1..2
Main
gen_level1(): List[Brick]
levels: List[Brick]
clock: Clock
running: bool
score: int
frame: int
main()
1
rendered on >
1..5
ParticleSystem
- particles: List[Particle]
+ spawn_rect(area: Rect, color: Color, amount: int, life: int = 1): None
+ step(dt: float): None
+ draw(surface: Surface): None
1
*
Paddle
paddle: float
paddle_width: float
paddle_height: float
paddle_color: Color
paddle_angle: float
1
runs on v
1
«dataclass»
Particle
+ position: Vector2
+ velocity: Vector2
+ color: Color
+ life: float
Brick
+ hits: int
__init__(rect: Rect, kind: int = (
+ color(): Color
*
CubeEngine
+ full_surface: Surface
+ horizontal_surface: Surface
+ handle_inputs(): bool
+ flip(): None
+ stop(): None
+ show_qr_code(): None
+ item: attribute
- matrix_conn: PipeConnection
- matrix_process: Optional[Process]
- window_surface: Optional[Surface]
- window_scale: int
- debug_server: Optional[DebugServer]
__init__(show_on_cube: bool = False, run_debug_server: bool = False, show_window: bool = False, window_scale: int = 5)
- matrix_process(connection: PipeConnection): None
- calc_matrix_size(options: RGBMatrixOptions): Tuple[int, int]
GetIp
+ get_ip(): str
may use ^
DebugServer
~ thread: Thread
+ sockets: List[Server]
+ server: Server
__init__(onmessage: Callable[[Any], None])
~ run(): None
+ debug_msg(msg: str): None
+ display_raw(pixels: bytes): None
~ send(data: Any): None
+ shutdown(): None
Button
+ is_down
+ went_down
+ went_up
+ toggle()
+ set(down: bool)
+ reset()
6
«dataclass»
Input
+ up: Button
+ down:Button
+ left: Button
+ right: Button
+ action: Button
+ pause: Button
+ joyx: float
+ joyy: float
1
< stores input in

Figure 3.6: Arkaniod class diagram

Arkaniod class diagram is simple. It has the utility with general-purpose functions which operate on all objects in the game. It has 1 or 2 balls, according to the game rules, which hit Bricks. After a hit, for better experience we added a particle system which displays a destruction effect.

3.4 Class Diagram - Backend

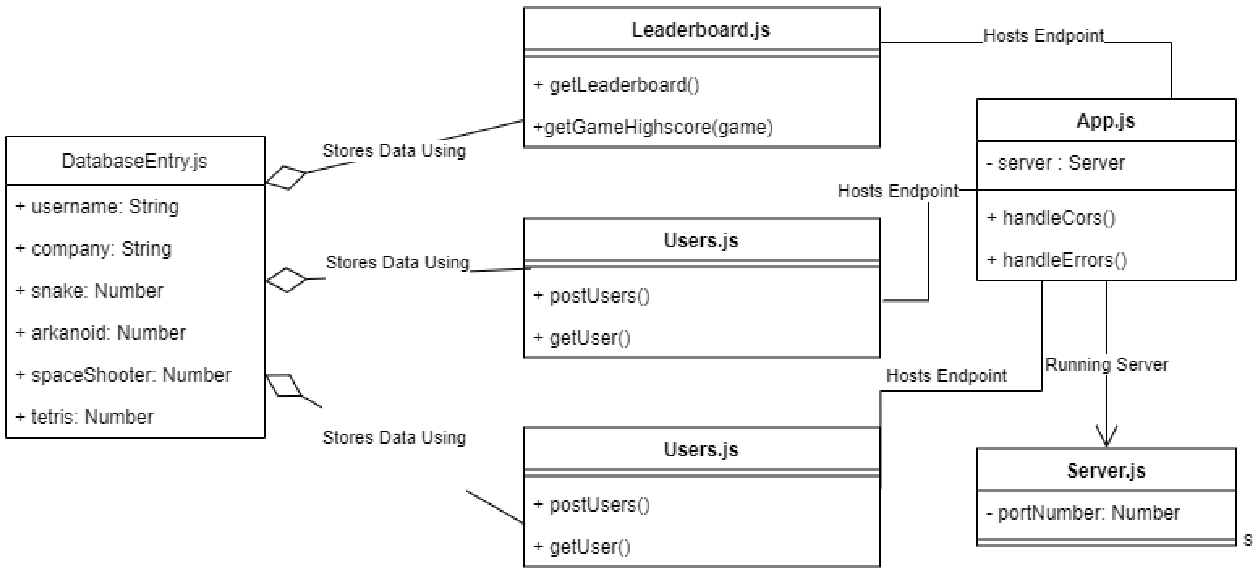


Figure 3.7: Class Diagram of Backend

The classes can be mainly split into 3 groups. The model which consists solely of databaseEntry.js. As stated in the previous chapter, all our data is stored in a single table(better known in MongoDB as a model). This model will store the username and company of each user who plays our game and the high score of the user for each of the games that we have to offer.

API endpoints which consist of leaderboards.js, users.js, scores.js. These endpoints are each associated with their specific URI's.

And lastly server code. This consists of app.js and server.js. These files make a call to expressJS and initialise a base version of the server.

3.5 REST API Calls

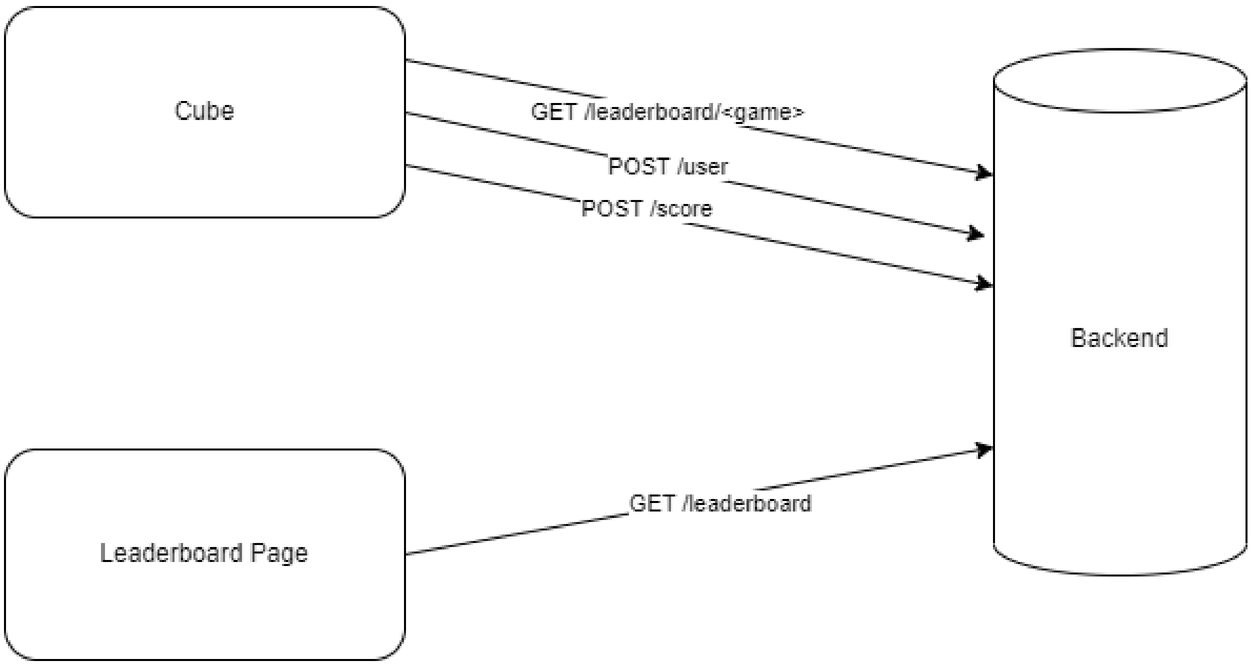


FIGURE 3.8: REST API Calls

Above you can see all the rest API calls that are made to the backend as well as the component which initiated this request. Here is a brief description of each request

GET /leaderboard: This call is made by the leaderboard page to load the leaderboard table.

GET /leaderboard/<game>: This call is made by the cube to get a high score for a specific game. This score is displayed on the cube so the user knows what score they need to beat

POST /user: This call is made by the cube to 'login' to the system as an existing user or to create a new user in the database

POST /score: This call is made by the cube to send the score of the user for a game that has been played. If the score is higher than their previous scores, the high score is updated.

Chapter 4

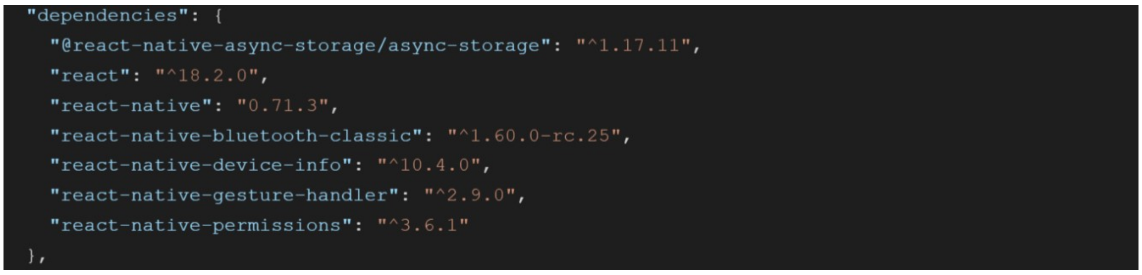
Technology Stack

4.1 Mobile App

The “CubeApp” mobile application is developed to be used as a controller for games on the raspberry pi cubes mentioned in the project. Specifically, it was built to control the following games: “snake, tetris, space shooter, and arkanoid”. The app itself is built using React Native, an open-source framework that allows for the creation of native mobile applications using JavaScript, React and Typescript JSX. In general, JSX syntax is a syntax extension that allows us to write HTML syntax in the code. However, in mobile app development JSX is used to write the user interface components in React Native, and it is similar to HTML but with added benefits of using inline coding with javascript.

In CubeApp, we are utilizing TSX, which is a combination of TypeScript and JSX. TSX helps us to utilize TypeScript with JSX and allows us to write React code with type-checking, which can prevent bugs and improve code quality.

Thus, TypeScript provides better code quality and error handling, making it a suitable choice for a project like CubeApp. In the below paragraphs we will explain all the dependencies and libraries we used in the development of the mobile app. The list of all libraries used in the mobile app can be seen in this code snippet taken from the “package.json” file of the react native project.



```
"dependencies": {
  "@react-native-async-storage/async-storage": "^1.17.11",
  "react": "^18.2.0",
  "react-native": "0.71.3",
  "react-native-bluetooth-classic": "^1.60.0-rc.25",
  "react-native-device-info": "^10.4.0",
  "react-native-gesture-handler": "^2.9.0",
  "react-native-permissions": "^3.6.1"
},
```

Figure 4.1: Mobile app dependencies

4.1.1 Built-in libraries in the React Native App

Firstly, React and React Native built-in initial libraries are used in the mobile app. They are both popular libraries used for building user interfaces and web/mobile applications respectively. Currently, for the project we are using React Native version “0.71.3” and regular React version `"^18.2.0"`(the latest versions at the time of writing the report). Both of these libraries have dependencies that can be used for various purposes.

In React, one of the most popular dependencies used is useState.

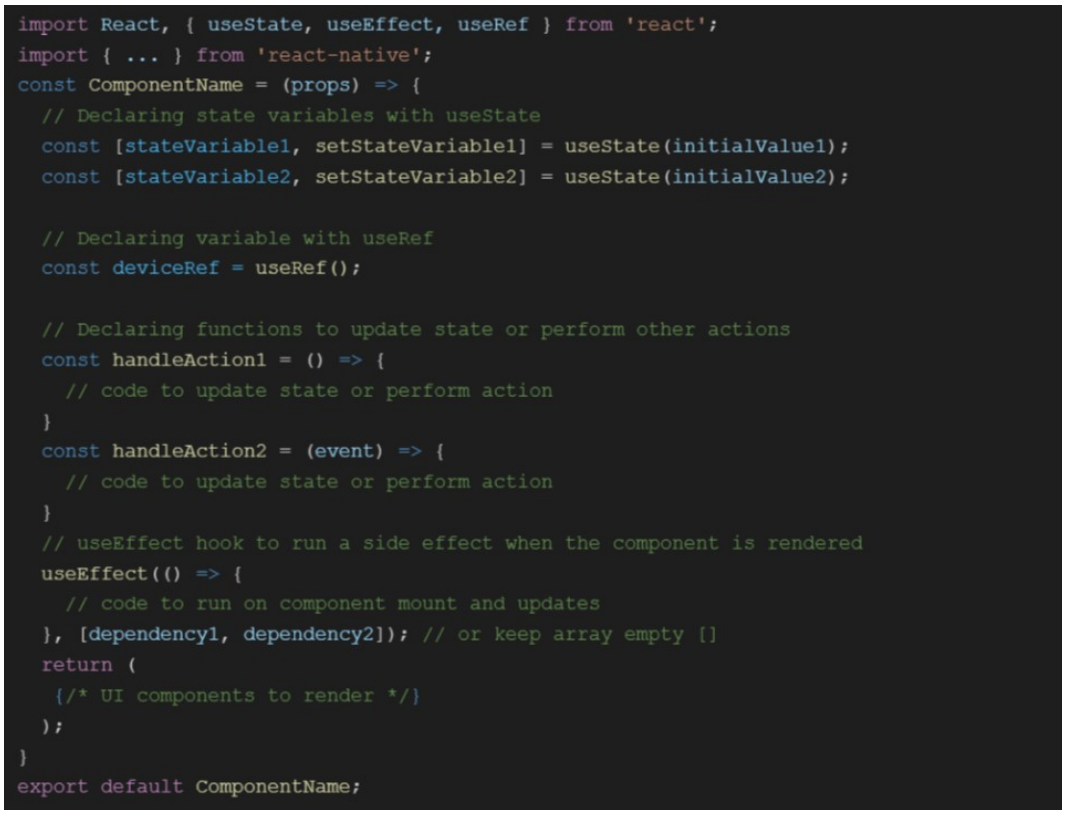
This hook allows you to have state variables in functional components in React, which can then be used to update the UI based on events or user actions.

The useState function returns an array with two elements. The first is the current state variable value, and the second element is a function that you can use to update the state variable's value. For example, one can employ useState to save user input in a form and subsequently adjust the UI based on the new results.

Another hook we used was useRef provided by React which can help you create a mutable reference that persists even after the re-renderings of the components. In contrast to the useState hook, modifications to the useRef value do not prompt a re-rendering of the component. This hook is useful for gaining access to and modifying values or DOM elements that would otherwise be hard to access via traditional React state or props. For example, useRef can be used to focus on an input element in response to a particular event. This can be achieved by making a reference to the input element and running its focus() method. Another use case for useRef is to store values that has to persist across component re-renders, such as previous state or props. Additionally, useRef can be useful to store a device reference and send constant keep-alive messages to it. In this example, you can use setInterval method to send the keep-alive messages at a regular interval. useRef is particularly useful in this case because it allows you to store a mutable value that persists across re-renders without triggering a re-render. If you were to use useState to store the device reference, calling setState on each interval would trigger a re-render, which would lead to unnecessary performance overhead. Through using useRef hook, updating the device reference can be accomplished without triggering a re-render of the component and this can lead to improved efficiency of the software.

Lastly, in the react library, we used useEffect dependency. This hook allows you to implement side effects within components, such as retrieving data from an external API or updating variables. useEffect can also be used to keep track of changes in your component's state, and then trigger an action based on those changes. This hook takes in two arguments, a callback function and a dependency array. The callback function is executed every time the component is rendered, while the dependency array specifies the values to be monitored for changes. When the dependency array is empty, the callback function will only be executed once, when the component's initial mounting. On the other hand, adding values to the dependency array will make the callback function run each time a change occurs in those values. This feature is useful for updating a component in response to changes in its dependencies. For instance, useEffect with an empty dependency array can be used for fetching data from an API when a component mounts. This makes sure that the data is retrieved only once, during the initial rendering of the component. If you need to re-fetch the data every time a certain value changes, you can include the variable in the dependency array. It is also worth noting that the dependency array can be left empty, in which case the callback function will execute with every component rendering, regardless of any value changes. This feature is good for executing actions that do not depend on any specific values.

Below you can find an example syntax of how we created components in react-native that use react's features.



```
import React, { useState, useEffect, useRef } from 'react';
import { ... } from 'react-native';
const ComponentName = (props) => {
  // Declaring state variables with useState
  const [stateVariable1, setStateVariable1] = useState(initialValue1);
  const [stateVariable2, setStateVariable2] = useState(initialValue2);

  // Declaring variable with useRef
  const deviceRef = useRef();

  // Declaring functions to update state or perform other actions
  const handleAction1 = () => {
    // code to update state or perform action
  }
  const handleAction2 = (event) => {
    // code to update state or perform action
  }
  // useEffect hook to run a side effect when the component is rendered
  useEffect(() => {
    // code to run on component mount and updates
  }, [dependency1, dependency2]); // or keep array empty []
  return (
   {/* UI components to render */}
  );
}
export default ComponentName;
```

FIGURE 4.2: Example component syntax

In the React Native library, the main purpose is to provide a wide range of UI components for building mobile applications. The components used in the App are:

- ActivityIndicator: A UI component that displays a spinning wheel to indicate that the application is loading.
- Alert: A UI component that displays an alert pop-up with a message and optional buttons.
- Dimensions: A library that can provide information about the device's screen size and orientation.
- FlatList: A UI component that efficiently renders large lists of data by rendering only the items that are visible on the screen.
- LayoutChangeEvent: An event that is triggered when the layout of a component changes.
- LayoutRectangle: An object that represents the position and size of a component's layout.
- ListRenderItemInfo: An object that provides information about a single item in a FlatList.
- Modal: A UI component that displays a modal dialog on top of the current screen (like a pop-up page).

- PermissionsAndroid: A module that provides methods for requesting and checking permissions on Android devices.
- Platform: A utility module that provides information about the current device platform, either iOS or Android.
- SafeAreaView: A UI component that ensures its children are not obscured by the device's status bar or other system UI.
- StyleSheet: A module that allows a way for developers to define styles for components in a structured and reusable way.
- Text: A UI component that displays a text string.
- TextInput: A UI component that provides a way for users to input text.
- ToastAndroid: A module that provides a way to display short-lived messages to the user.
- TouchableOpacity: A UI component that provides a touchable area that responds to user input. (responsive buttons)
- View: A UI component that serves as a container for other components. (similar to div in html)

By using these components, we built an interactive and dynamic user interface for the React Native game controller app. The UI components are similar to their HTML counterparts but have additional features and properties that are specific to mobile devices.

For example, TouchableOpacity is a React Native component that facilitates touchable functionality for its child components. This implies that when a user presses on a TouchableOpacity, the child components in it will obtain the touch event. One typical application of TouchableOpacity is to develop a clickable button or a link that submits a form or navigates to a different screen in the app when pressed. These components can also be customized using styles just like CSS, which can be defined using the StyleSheet module. This allows developers to achieve consistent designs and user interfaces across their app.

4.1.2 External installed libraries for the App

Besides built-in libraries we also utilized some external libraries for the features of the mobile app.

1. "react-native-permissions": This library provides a cross-platform API for requesting user permissions in React Native apps. It supports requesting permissions for several features such as location, camera and Bluetooth. In the App, this library was used to obtain Android Bluetooth and location permissions, which are required for Bluetooth device discovery.
2. "react-native-device-info": This library provides a way for retrieving device information such as device name, device model, and screen dimensions. In the App, this library was used to get the device dimensions and name. Dimensions of the mobile device were necessary for some controllers to make sure that the app looked good when the phone was rotated to horizontal view, and the device name was used to send

the username to the cube. Since we wanted to achieve some sort of nice uniqueness, the username sent to the server was always "<selected username>" + "- <device name>".

3. "@react-native-async-storage/async-storage": This library allows asynchronous storage system that allows for the storage and retrieval of data in the local storage of the mobile device. Essentially, It allows the app to store and retrieve data in the form of a key-value pair in the cache. In the App, this library was used to collect and store the user's name and company name in the application's local cache storage.

4. "react-native-gesture-handler": This library makes it easier to add gesture recognition to React Native apps, so developers can create custom UI components that respond to different types of touch interactions like taps, swipes, and pinches. In the App, this library was used to create a custom joystick to control some of the games. For spaceShooter we needed the joystick since the ship in the game can move to all directions. And for the arkanoid game we needed a slider that can go up and down, so we added an extension to our custom joystick making it a slider in the arkanoid controller and making it a regular joystick in spaceShooter game.

5. "react-native-bluetooth-classic": This library has an API that allows developers to communicate with Bluetooth devices that use classic Bluetooth technology. It supports connecting to Bluetooth devices via RFCOMM sockets, which is a Bluetooth protocol used for serial communication. In the App, this library was used to establish a connection between the Bluetooth cube RFCOMM socket server and the mobile device. This enabled smooth communication between the cube and the device, allowing users to enjoy games on the cube while using the mobile device as a wireless controller.

As a result, the overall features of the App were:

- Wireless controllers for games on the cube
- Custom joystick using "react-native-gesture-handler"
- Bluetooth connectivity using "react-native-bluetooth-classic"
- Cache storage of user name and company name using "@react-native-async-storage/async-storage"
- Obtaining Bluetooth and location permissions using "react-native-permissions"
- Obtaining Dimensions and name of the smartphone using "react-native-device-info"

4.2 Leaderboard Page

The leaderboard page is a web application that essentially allows users to view the high scores of all employees. The company requested an arcade-themed design for the web app, which influenced our decision to make it a single-paged application with a user-friendly and intuitive interface that incorporated arcade-style fonts and design choices. This web app is mainly built using the React and React Bootstrap library, which allows for the creation of dynamic and interactive user interfaces. Specifically, React was chosen for the leaderboard page due to its ability to create reusable and modular components. React's component-based architecture made it easy to break down our complex single page into smaller parts, each responsible for specific functionality. This modularity not only made the code easier to manage and maintain but also allowed for cleaner code and thus faster development times. Furthermore, while building our own components we also made use of React Bootstrap which provided a wide range of pre-built UI components, such as buttons, forms, and models. This helped us to rapidly develop the page while maintaining consistency and style across the application. By leveraging the power of both React and React Bootstrap, we were able to create a fully responsive and user-friendly web application that met the requirements of the project. In the following paragraphs, we will go through the dependencies and libraries that were used for building the web app.

4.2.1 React

Since the web app is built mainly using the React library, we made use of several dependencies throughout the project. Below is a small description of the used hooks (functions):

1. **useState:** This is a hook that enables functional components to manage state (similar to a variable). By using useState, you can define and update the state values within a component. Any update of a state will trigger the re-rendering of the specific component.
2. **useEffect:** Another hook that allows functional components to execute side effects, such as fetching data or updating certain components, when certain conditions are met. This is similar to a function basically.
3. **createContext:** A hook that creates a context object, which is used to share data across the component tree. This can be visualized as a global variable that can be accessed by different components.
4. **useContext:** Another hook that allows the context object, created by using createContext, to be modified.

For more information regarding these hooks, refer to the Mobile App section.

4.2.2 React Bootstrap and Bootstrap

The React Bootstrap library was mainly used to ease the process of creating our own complex components. The main components we relied on were button, table, modals, form, input and dropdown. Relying on these simple components allowed us to save time and focus on developing our own custom components while ensuring high responsiveness and efficient code. In addition to React Bootstrap, we also made use of the normal Bootstrap library for its CSS classes. Bootstrap is a popular front-end framework that provides a set of CSS classes for styling HTML elements. We made use of these CSS classes mainly for structuring our webpage.

4.2.3 Our Components

The majority of the components we built were the larger building blocks of the web app which consisted of small components from React Bootstrap. Below, we have outlined each component and its functionality:

1. **DropdownGame:** A component which is used to create a dropdown for selecting which game to display the leaderboard for.

2. **InputUser:** A large component which consists of an inner custom component. This component is essentially an input where users can enter a valid username to view their scores for each game. When a valid username is entered, a popup is created that displays the user's scores for each game. Additionally, the popup includes a table that allows users to compare their scores with another user. This table is dynamically generated based on the selected user, providing an interactive and user-friendly experience. Overall, the InputUser component serves the functionality for retrieving and comparing user scores.

3. **Leaderboard:** Another complex component which is used for displaying the high scores for all the available games on the cube. This component creates a dynamic table that shows the top scores for each game and updates the table based on a dropdown menu that allows users to select the game they want to view. When a game is selected, the table is automatically updated to show the top scores for that particular game.

4.3 Backend

This project required a Backend component as Mindhash requested the need for a competitive element. Thus we would have to somehow store and retrieve these scores. The backend will receive requests from both the Pi inside the cubes as well as the Leader board page. We decided to not send any requests from the Mobile App as we assume that no internet connection is needed to play the games.

We decided to use a REST API for the front end to interact with the backend. With regards to the programming language, we wanted to use a JavaScript framework to code the backend due to familiarity with it so we decided to go with NodeJS. One nice thing about Node was that it already has very good support for REST Backends through importing frameworks like ExpressJS.

Of course, we also needed a way to store data. We decided to use MongoDB to store our data. It is most suitable in this case as the database will only have to store basic information like usernames and high scores of users. In principle, we aimed to only have 1 'table' which will store all of this information. As such, there are no complex relations between different tables. Using a NoSQL option allows us to store this data in an unstructured or semi-structured manner which is suitable for our purpose.

4.3.1 Libraries Used

Apart from the built-in Node Libraries, there were several external libraries we used along with NodeJS to build the backend.

1. **ExpressJS:** ExpressJS simplifies the process of creating REST API. It is built on top of Node's existing HTTP module to give further abstraction when building APIs

2. **Mongoose:** Mongoose is used to connect and communicate with our MongoDB Database

3. **Body Parser:** Body Parser is used to easily parse the body of any POST requests that the server may receive

4. **Morgan:** Morgan is a logging library that automatically detects when a request has been made and prints this out to the console. This framework is mainly added for debugging purposes.

4.3.2 Requests

There are several requests that we want to make to the backend. In this section, We will not elaborate on the specific calls but simply on what we want to send and retrieve from the backend.
From the Cube:

- Send the score of a user for a specific game
- Send the details of a new user
- Retrieve high scores for a specific game

From the Leaderboard Page:

- Retrieve the leaderboard table for all games
- Retrieve the leaderboard table for a certain game

4.4 Cube Game Framework

We decided to build a small framework for interacting with the cube and app which we could use in each of the games. This framework covers displaying images on the cube, managing inputs and communicating with the app. We initially started making this framework in nodejs but later decided to use Python instead because it was better suited for making games and we had more experience making games in Python. Python was also equally as capable of communicating with the server and mobile app.

4.4.1 Libraries Used

1. **pygame** is used to draw the graphics for the games and to open up a window to test out the games without access to the cube
2. **rpi-rgb-led-matrix** is used to drive the displays on the cube
3. **pybluez** is used to communicate with the phone app over Bluetooth
4. **requests** is used to communicate with the leaderboard server
5. **flask** is used to host a debug server for testing how the games look on the cube without the cube using a 3d representation, it was also used to test out the inputs eventually used in the mobile app

4.5 Games

There are 4 games implemented for playing on the cubes: "Snake", "Space invaders", "Arkanoid", "Tetris". The choice of games was made taking into account the features of the displays on cubes and their current limitations. Our goal is to make the gaming experience unique, and therefore we try to use as many displays as possible for the game.

Also, we balance providing players with a fresh and exciting 360-degree gaming experience while ensuring that the gameplay is enjoyable without being too difficult, inconvenient or unsafe to play. This is going to be explained in detail in later sections.

The framework which is used in the games is pygame. The reason that pygame perfectly fits the purpose of our project is that its pixel-looking games are quite easily deployable on cubes and they look perfect on 64x64 displays.

4.5.1 Snake

"Snake" is a classic arcade video game in which a player controls a snake-like creature that moves around a rectangular playfield, consuming food items while avoiding obstacles and its own tail. The snake's tail grows longer as it consumes more food items, making it more challenging to avoid running into it. The game ends when a snake's head hits a wall on the side of the screen or its tail.

The game mechanics are implemented as follows. Four side displays of the cube are involved in the game, which delivers 360-degree experience. There are no left and right border; there exist only the top border and the bottom border. Specifically for this purpose, we have implemented an in-game algorithm for removing seams, which allows the player to freely move around 360 degrees. The food is implemented as a single green pixel which a player first has to find on one of the side displays and then move the snake to it and consume it. The most interesting part of the game is that most of the time a user does not know where the food is located, and they shall think ahead about how to guide a snake to the possible food location. Furthermore, the interest is added by the fact that at a late stage of the game, the snake occupies different displays, and the player needs to remember where the snake's body is in order not to run into himself sharply when switching to another display.

We have added the feature to win the game. If a player reaches a certain snake size, they win a game. Since there are no additional obstacles in the middle of the field, the maximum snake size is done to prevent people from seeking an algorithmic approach how to get most of the points. This mechanic lies in the idea of how to faster complete the game, not how to get the highest score.

4.5.2 Space Invaders

"Space Invaders" is a game where the player controls a spaceship which destroys incoming alien enemy ships descending from the top of the screen to the space base located at the bottom of the screen. The player must shoot down the aliens with the laser cannon located on the spaceship they control before they reach the bottom of the screen.

The game mechanics are implemented as follows. Four side displays of the cube are involved in the game, which delivers a 360-degree experience. Specifically for this purpose, we have implemented an in-game algorithm for removing seams, which allows the player spaceship to freely move around 360 degrees.

The game is divided into levels, and each level consists of multiple waves of alien ships. The game continues until the player loses all of their space bar lives or their spaceship is destroyed by an alien missile. The objective is to achieve the highest score possible by shooting down as many alien ships as possible while avoiding their attacks.

As the player progresses through the game, the waves of alien ships become longer and enemies shoot more aggressively, making it more difficult to avoid their attacks. Despite this, the player's health is gradually recovering, which motivates them to keep playing.

There are safety measures which were considered during the development of the game. Since the game is available on all side displays, and a user is unable to see the whole field of the game, is it vital to consider how a user is going to navigate through the game. There was a consideration that a user starts running around the cube in the late stage of the game in order to receive timely information about the state of the playing field and new enemies. That is why we applied a safety measure in the form of limiting the speed of enemies. Unlike the original game by Taito Corporation, in our version, enemies move slowly on cubes, and their speed does not increase with each level. This is done intentionally so that instead of running around the cube, risking injury the user can safely move around the cube.

4.5.3 Arkanoid

"Arkanoid" is a game where the player controls a paddle which is used to bounce a ball upwards to destroy a wall of bricks. The player must prevent the ball from falling off the screen by moving the paddle to keep the ball in play.

The game features various types of bricks, each with a different durability level and point value. Some bricks may require multiple hits to break. There is a small probability to receive one more ball while breaking bricks, and this mechanic gives an extra chance to a player to complete a certain level.

Unlike the original game by Taito Corporation, where the platform is placed on the bottom and the bricks are located on the top of the screen, in our implementation, the platform is located on one side and the bricks are located on another side. The game occupies two adjacent displays. There are several reasons why we implemented the game in such a way. We have tested different combinations of displays by multiple members of a team. The first reason is the optimal angle of view. We have found that the classical horizontal orientation of the game is not easily playable because of the bottom screen, and the usage of side displays provides the best experience. The second reason is the consideration of the balance between safety and involvement: since all sides of cubes are not visible to a player, and it is unsafe for them to run around the cube to see the whole playing field, we shall make some prevention measures. However, unlike in the case with Space Invaders, in Arkanoid decreasing speed makes it too inconvenient to move around cubes back and forth, too easy to control a platform, and, therefore, not interesting to play the game.

4.5.4 Tetris

"Tetris" is a classic puzzle video game known almost to everyone. The game features a rectangular playfield, consisting a grid of cells. The game pieces, known as "tetrominoes", are geometric shapes made up of four square blocks that fall from the top of the screen. The player must rotate and position the falling "tetrominoes" to fill complete horizontal lines at the bottom of the playfield without any gaps.

As the player clears lines, they earn points. If the stack of tetrominoes reaches the top of the screen, the game is over.

The game features seven "tetromino" shapes, each with a distinct color, and the game pieces fall in random order. The player can move the falling "tetrominoes" left or right, rotate them clockwise or counterclockwise, or drop them instantly to the bottom of the playfield.

While we were designing the game, we considered that Tetris is a game of skill and strategy which requires quick reflexes and spatial awareness to position the falling "tetrominoes" effectively. That is why we considered that the field must be clearly visible to a user. Although one display can accommodate 64x64 field, we decided to implement 16x16 field with 1 piece occupying 4 pixels. The game is implemented on two adjacent displays where the left display is allocated for the game and the right display displays the next piece and the high score. We believe that this design helps a user to quickly respond and allocate places for "tetrominoes" in advance.

Chapter 5

Process

5.1 Communication with Client

Throughout the project, we maintained clear and consistent communication with our client. We had weekly meetings with the client where we provided updates on our progress, discussed challenges we were facing, and requested help if needed. These meetings were very valuable as it gave the chance to receive immediate feedback and ensure that everyone was on the right track. In addition to these weekly meetings, we made use of Slack, a communication app, to quickly ask questions, share information, and stay connected with the client. These two communication methods helped us to maintain a high level of collaboration and transparency with the client, which was essential to the success of the project.

5.2 Communication with Supervisor

Throughout the project, we had weekly meetings with our supervisor. During these meetings, we provided updates on our progress and ensured that we were on track with the deliverable for the Design Project.

5.3 Peer Review Meetings

During weeks 3, 5, 7, and 9, we had Peer Review meetings where we had the chance to present our progress to our colleagues and receive feedback. These meetings were essential as we had the chance to see the work of other groups and assure that the quality of our work was on high standards and we were on right track.

Specifically, in the first meeting we focused on receiving feedback on our project proposal and planning. This way, we would guarantee that the overall design of the project was complete. In the second meeting, we made sure that our test plan was complete and had high coverage of all the components we built for the project. In the third meeting, special attention was given to receiving feedback on our design report. We had the chance to see the report structure of other groups and adjust our design structure accordingly. In the final meeting, we had the chance to see the posters of other groups and also receive feedback from people. Overall, these meetings were crucial throughout the development of our project.

5.4 Planning and GIT

Communication was crucial for the project's success, and we mainly relied on Discord for communication within the team. To keep the planning organized, we created specific channels for particular tasks. This way we ensured that a good division of tasks was achieved and attention was given to every task.

Regarding version control in our project, we made use of Gitlab which was provided by Mindhash. Gitlab allowed us to have all our code in one location which made it easy to manage and track changes. Moreover, by using Gitlab, we ensured that the code was transparent and the client could give feedback at any time.

5.5 Implementation Log

5.5.1 Week 1

Before the start of this week, we had a look at all the potential projects and as a group decided on our favourite ones. Once we got assigned the project we reached out to Mr Nacir who agreed to be our supervisor. During this week we also reached out to Mindhash and we scheduled a first meeting with them on the Monday of the following week. We spent the rest of the week going through the proposal document a few more times and coming up with questions that we could ask the company.

5.5.2 Week 2

At the start of this week, we went to the Mindhash Office to meet the company for the first time. After our meeting, we proceeded to start properly planning out our project and started writing the project proposal. We also got a better idea of all the different components, so we come up with a division within the team to tackle each task. After this, we started to brainstorm on all the languages and frameworks that we would use for each component of the project.

5.5.3 Week 3

During this week, we focused on finalising the proposal document as well as researching the frameworks/languages we had all decided to use. A few of us had not yet worked with the language that we would be using so we needed some time to learn it and get up to scratch. After sending our final proposal to Mindhash, they approved the proposal and said that we could come and pick up the cube in the coming week.

5.5.4 Holiday Week

During this Holiday Week, we continued to learn our required programming languages. In the middle of the week, we also travelled to Mindhash's office to pick up the cube.

5.5.5 Week 4

This is the week when we started implementing and working on the project. We started working on the leaderboard page, the back end, the mobile app and games for the cube. As we were coding, we also designed some Diagrams that we could include as part of the final report.

5.5.6 Week 5

By the end of week 5, we had completely finished implementing the leaderboard page as well as the back end and database. We had also implemented 2 games Snake and Arkanod in PyGame and were working on porting them over to the cube to work the LED Library. With regards to the mobile app, the initial interface and controllers were completed and we started to research potential Bluetooth Frameworks we could use.

5.5.7 Week 6

In week 6, we integrated the leaderboard page with the backend. We managed to get Snake and Arkanoid working on the cube display and started working on the next games which were Space Shooter and Tetris. We settled on a Bluetooth library and started working on the communication between the Pi and the Mobile application.

5.5.8 Week 7

This week, our main focus was on integrating all the components that we had implemented. We started by connecting our Pis in the cube with our backend and continued on working on the connection between the mobile app and Cube. We also finished developing Space Shooter and Tetris and had them working on the Cube Display. In this week we also started to work extensively on our report and already laid out an initial template that we would work on in the coming weeks.

5.5.9 Week 8

This week was devoted mainly to writing our final report. On the side, we finished integrating all the components and did some user testing and debugging. We tested our product for certain edge cases and added a few small additions that made the entire experience of playing the game a bit smoother and more enjoyable.

5.5.10 Week 9

This week we worked on finalising the draft version of our report. We also worked on the ethics group assignment that was part of the reflection component of this module. We also took time to work on the poster that we would use for our poster presentation this coming Wednesday.

5.5.11 Week 10

The last week of the module was spent completing the few tasks that were remaining and attending some final presentations. We spent time finalising our poster design and practising for our final presentation. We had the final presentation and poster presentation on Tuesday and Wednesday of this week respectively. Once those were over, we spent our time finalising our report based on the feedback that we received from our supervisor.

Chapter 6

Final Product Design

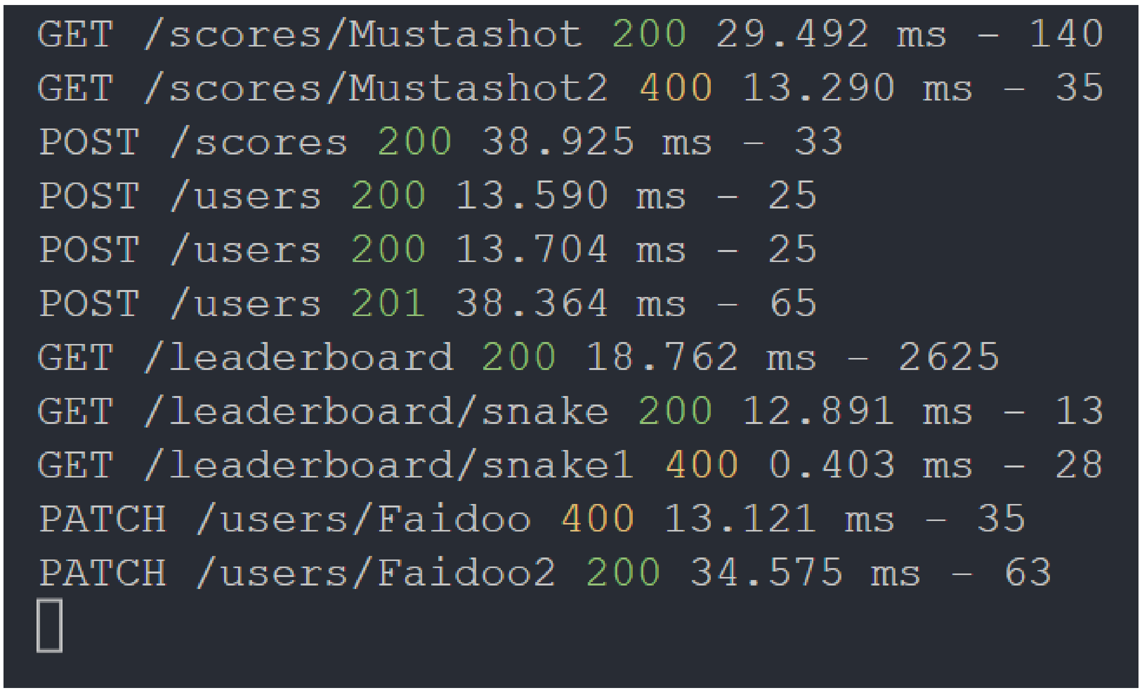
6.1 Back End

6.1.1 Logging

We set up our backend with debugging in mind. As such we made use of Morgan to print to console all the requests that are sent to the server.

For each request, Morgan will print out:

- Type or request (GET/POST etc.)
- URI
- Response Code
- Response Time



```
GET /scores/Mustashot 200 29.492 ms - 140
GET /scores/Mustashot2 400 13.290 ms - 35
POST /scores 200 38.925 ms - 33
POST /users 200 13.590 ms - 25
POST /users 200 13.704 ms - 25
POST /users 201 38.364 ms - 65
GET /leaderboard 200 18.762 ms - 2625
GET /leaderboard/snake 200 12.891 ms - 13
GET /leaderboard/snake1 400 0.403 ms - 28
PATCH /users/Faidoo 400 13.121 ms - 35
PATCH /users/Faidoo2 200 34.575 ms - 63
```

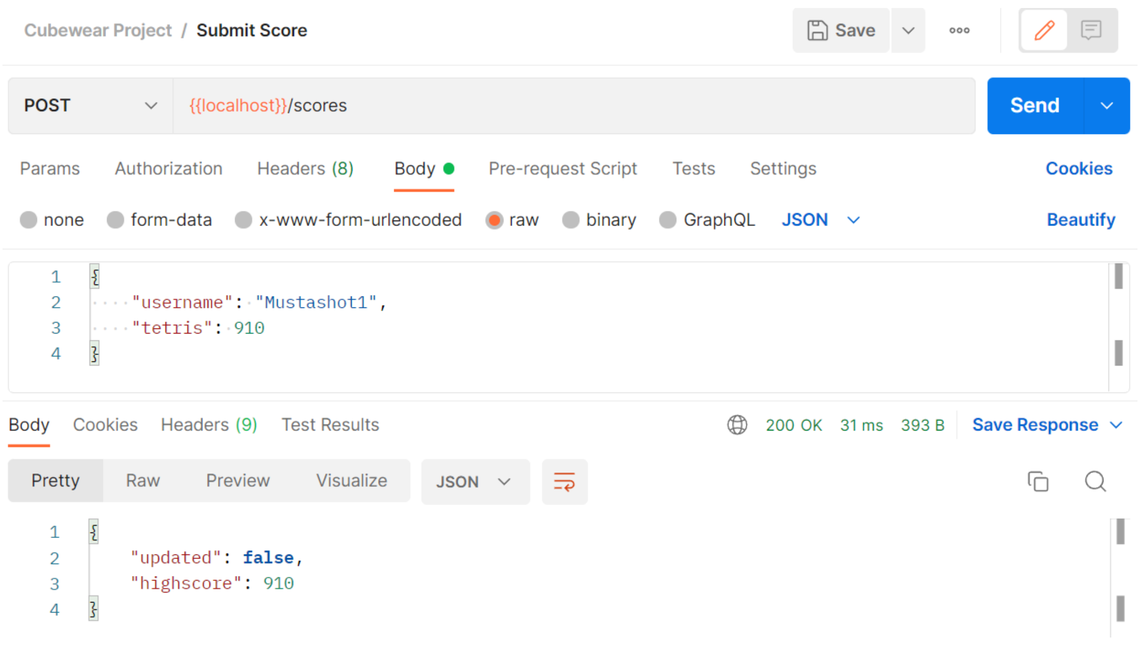
FIGURE 6.1: Morgan Log

6.1.2 Requests

In this subsection, we will show the response of the server on all of the endpoints we have designed. To see a more elaborate breakdown of the API refer to Appendix A.

These screenshots are from POSTMAN which shows the request body that is supplied as well as the response from the server(JSON Body and Status Code).

Sending score



POST {{localhost}}/scores

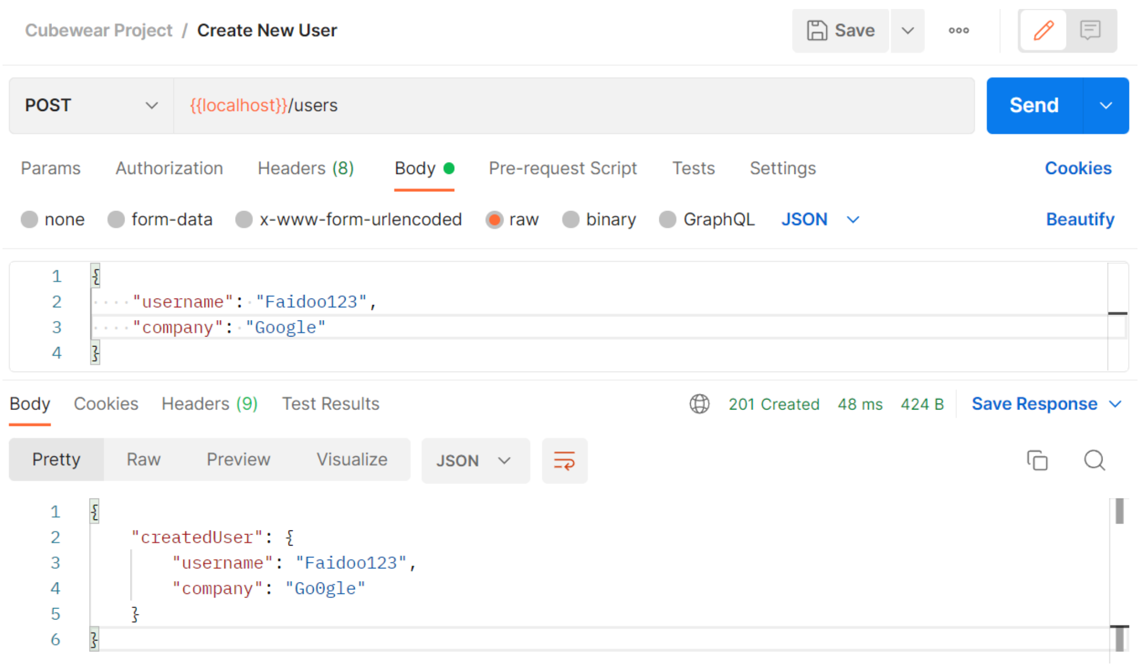
```
{
    "username": "Mustashot1",
    "tetris": 910
}
```

200 OK 31 ms 393 B

```
{
    "updated": false,
    "highscore": 910
}
```

FIGURE 6.2: POST /scores

Validating user

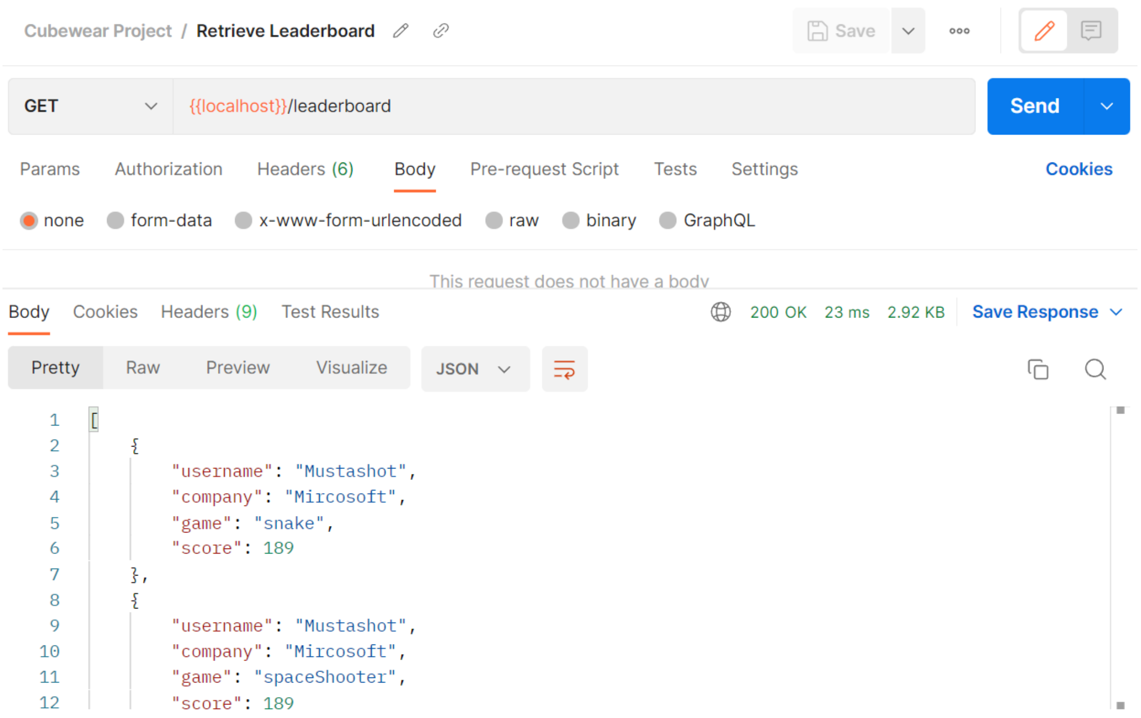


```
{
    "username": "Faidoo123",
    "company": "Google"
}
```

```
{
    "createdUser": {
        "username": "Faidoo123",
        "company": "Go0gle"
    }
}
```

FIGURE 6.3: POST /users

Retrieving leaderboard



Cubewear Project / Retrieve Leaderboard
Save
GET
Send
Params Authorization Headers (6) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL
Body Cookies Headers (9) Test Results
Save Response
Pretty Raw Preview Visualize JSON

{{localhost}}/leaderboard

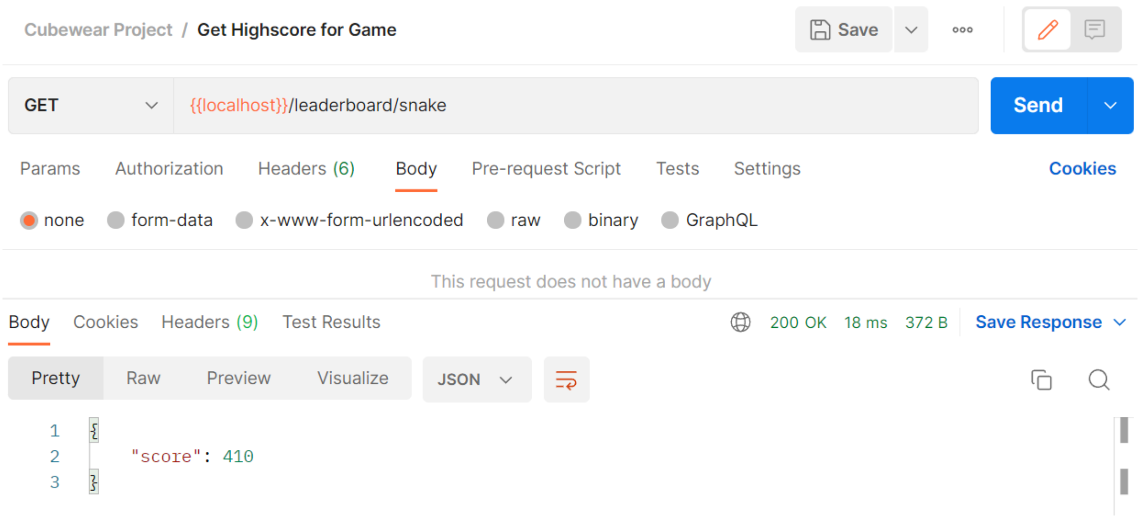
This request does not have a body

200 OK 23 ms 2.92 KB

```
[
    {
        "username": "Mustashot",
        "company": "Mircosoft",
        "game": "snake",
        "score": 189
    },
    {
        "username": "Mustashot",
        "company": "Mircosoft",
        "game": "spaceShooter",
        "score": 189
```

FIGURE 6.4: GET /leaderboard

Retrieving high score for given game



Cubewear Project / Get Highscore for Game
Save
GET
Send
Params Authorization Headers (6) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL
Body Cookies Headers (9) Test Results
Save Response
Pretty Raw Preview Visualize JSON

{{localhost}}/leaderboard/snake

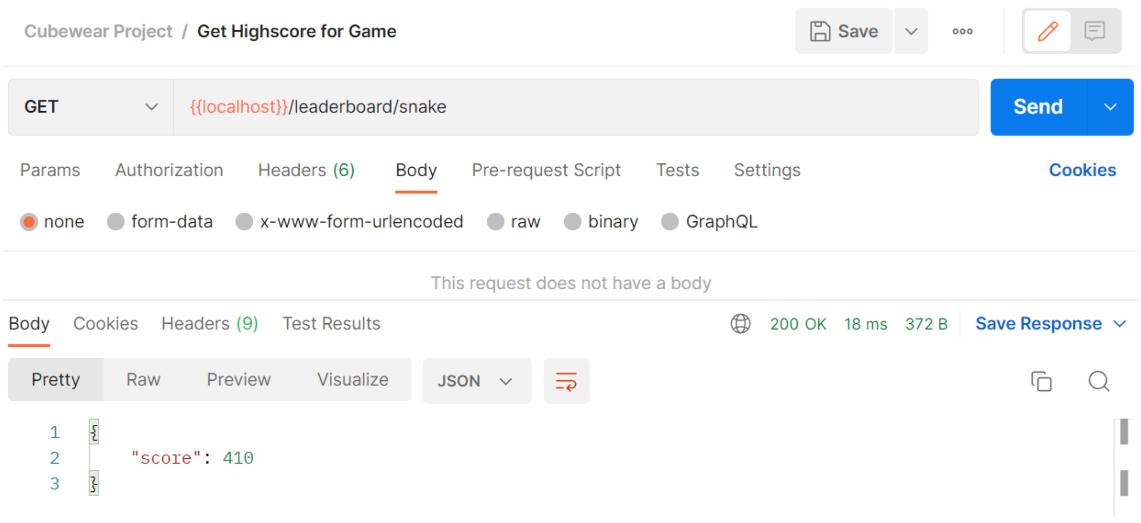
This request does not have a body

200 OK 18 ms 372 B

```
{
    "score": 410
}
```

FIGURE 6.5: GET /leaderboard/game

Update user details

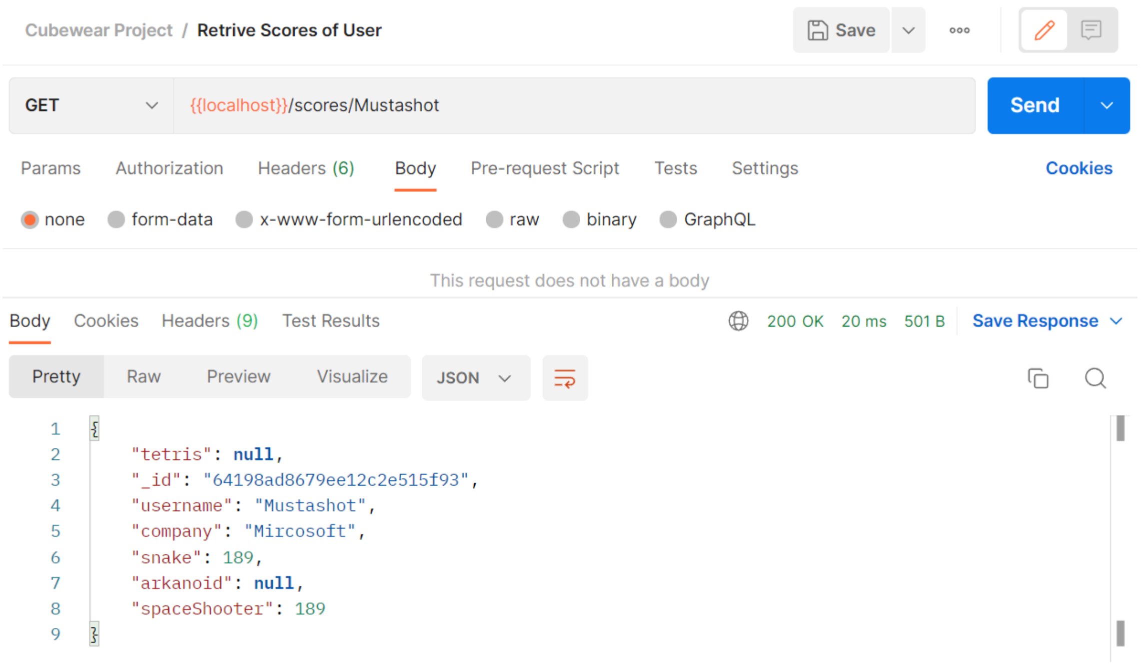


```
GET {{localhost}}/leaderboard/snake
```

```
{
    "score": 410
}
```

FIGURE 6.6: GET /leaderboard/game

Get scores of a given user



```
GET {{localhost}}/scores/Mustashot
```

```
{
    "tetris": null,
    "_id": "64198ad8679ee12c2e515f93",
    "username": "Mustashot",
    "company": "Mircosoft",
    "snake": 189,
    "arkanoid": null,
    "spaceShooter": 189
}
```

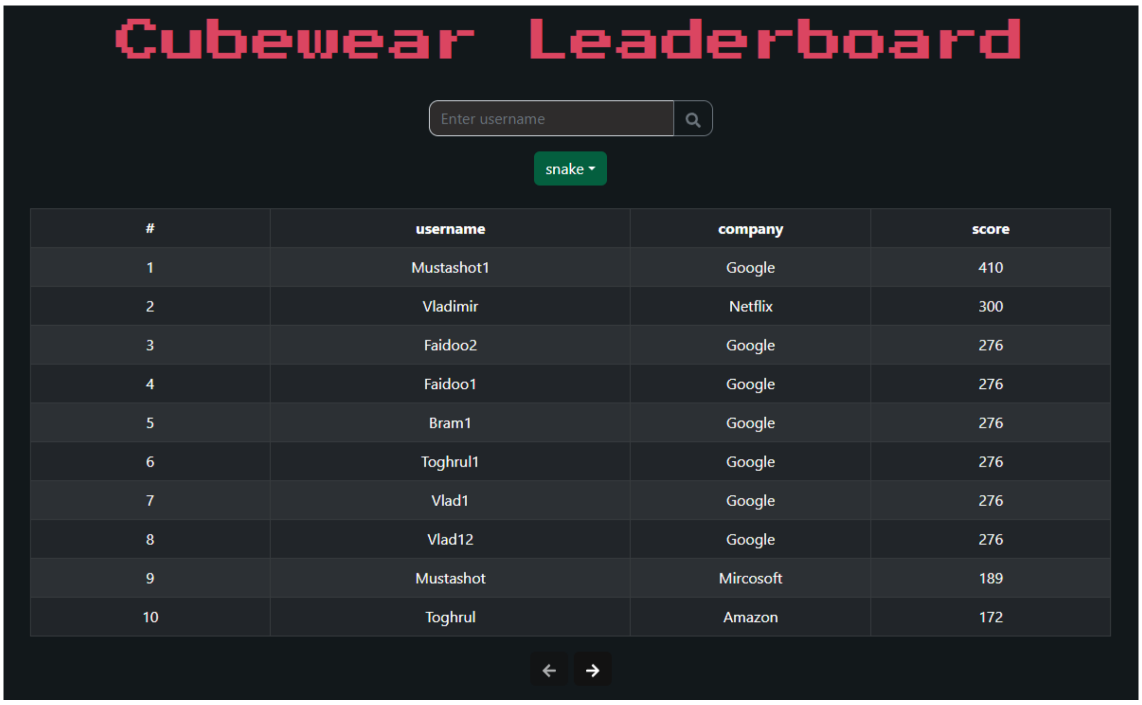
FIGURE 6.7: GET /lscores

6.2 Leaderboard Page

The final product for the leaderboard page includes all the functionalities that has been explained in the Technical Design section for the Leaderboard page. A summary of the components from the final product are below:

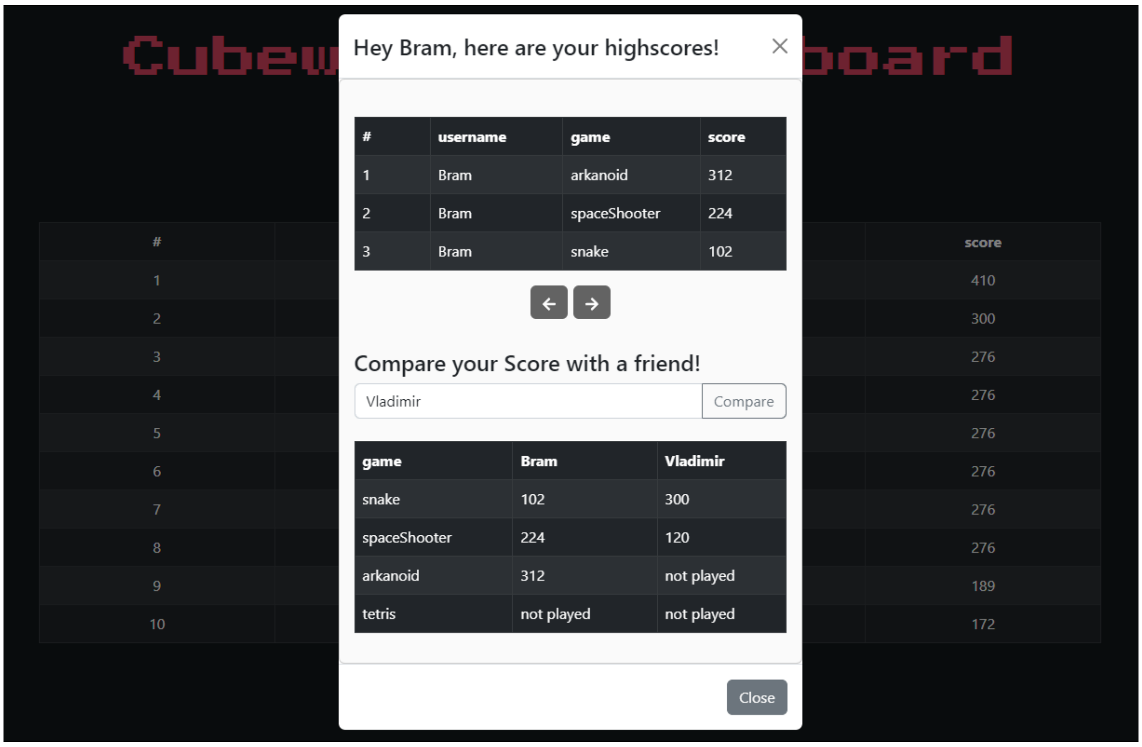
- Input field: Users can enter any valid username and once they click on the search button, a popup will shown.
- Popup: In this popup users can view their high scores for each game they played. They can also compare their scores with another player by typing their username respectively.
- Dropdown: Users can choose a game from the dropdown to view the high scores for that particular game.
- Leaderboard table: Table consisting of the high scores of players. This dynamic table updates based on the game selected from the dropdown.

All these components can be found in the screenshots of the application below.



#	username	company	score
1	Mustashot1	Google	410
2	Vladimir	Netflix	300
3	Faidoo2	Google	276
4	Faidoo1	Google	276
5	Bram1	Google	276
6	Toghrul1	Google	276
7	Vlad1	Google	276
8	Vlad12	Google	276
9	Mustashot	Mircosoft	189
10	Toghrul	Amazon	172

FIGURE 6.8: Leaderboard page



Hey Bram, here are your highscores!

#	username	game	score
1	Bram	arkanoid	312
2	Bram	spaceShooter	224
3	Bram	snake	102

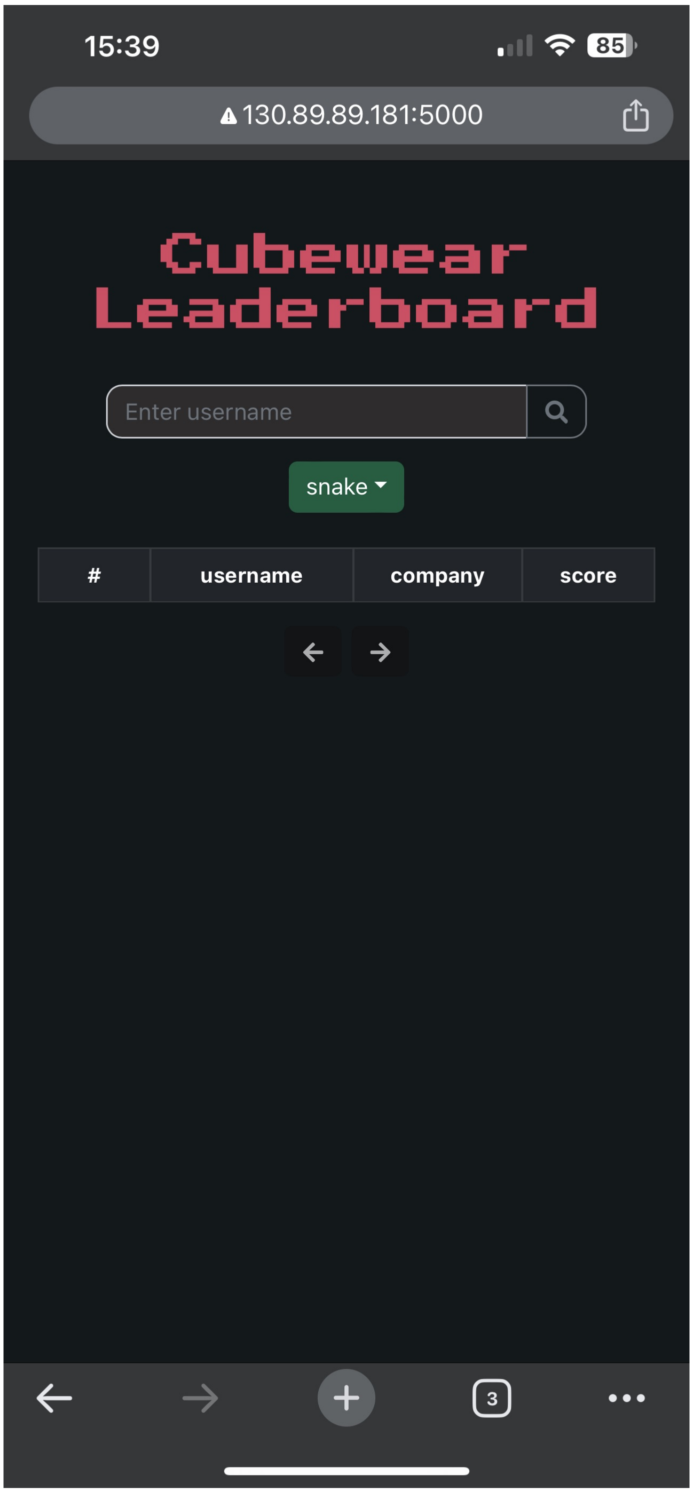
Compare your Score with a friend!

Vladimir Compare

game	Bram	Vladimir
snake	102	300
spaceShooter	224	120
arkanoid	312	not played
tetris	not played	not played

#	score
1	410
2	300
3	276
4	276
5	276
6	276
7	276
8	276
9	189
10	172

Figure 6.9: Popup



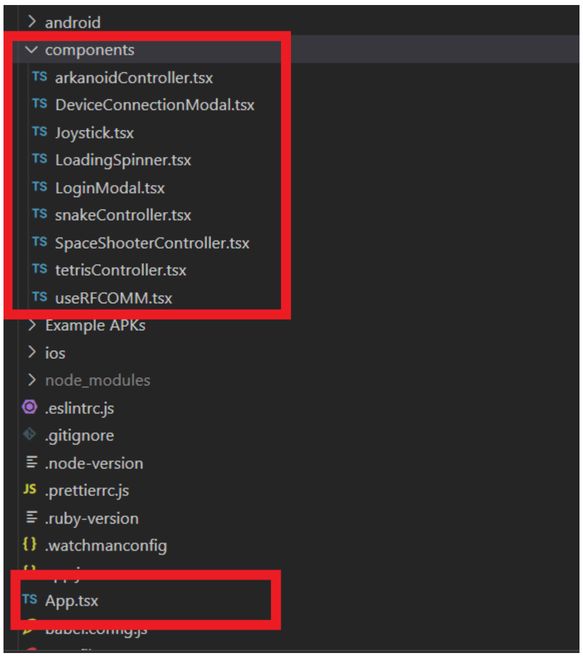
#	username	company	score

FIGURE 6.10: Leaderboard page Mobile View

6.3 Mobile Application development

In the development of the React Native game controller app the “CubeApp”, the main code logic is contained within the “App.tsx” file, which is the entry point for the application. The “App.tsx” file is responsible for rendering the various components that make up the user interface, as well as managing the application’s state and handling user input. To keep the code organized and maintainable, the various components used in the app are separated into their own files within the "components" folder. Each component has its own file that has the logic and styling specific to that component. This modular approach to development allows for easier maintenance and scalability of the codebase.

In this section of the report, we will dive deeper into the design and development part of the App and will talk about all the components we implemented in the App. One thing to note, all the sub-components that render a page are Modals. Modals are usually a pop-up window in web applications, however, since in mobile devices they cover the full screen, we can use them easily to imitate different pages. Usually for page routing and navigation people use external libraries, however, since this is a small controller app and since modals are already included in react-native’s built-in library, we decided to use modals for our App. Also, they are very easy to use and control, which is a good aspect of the maintenance and debugging of the codebase. Moreover, an overview of the components can be seen from the below picture.

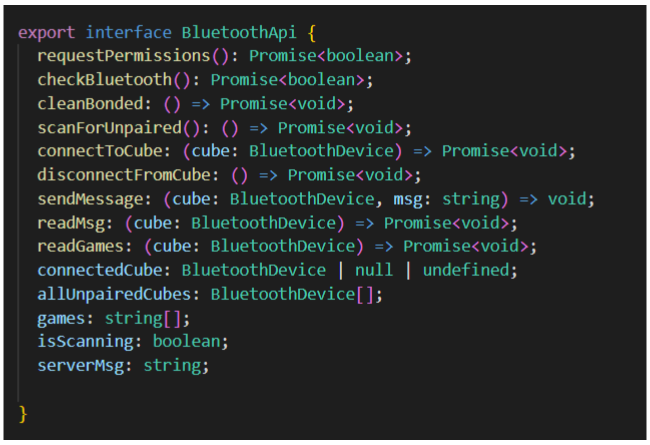


```
> android
v components
    arkanoidController.tsx
    DeviceConnectionModal.tsx
    Joystick.tsx
    LoadingSpinner.tsx
    LoginModal.tsx
    snakeController.tsx
    SpaceShooterController.tsx
    tetrisController.tsx
    useRFCOMM.tsx
> Example APKs
> ios
> node_modules
  .eslintrc.js
  .gitignore
  .node-version
  .prettierrc.js
  .ruby-version
  .watchmanconfig
  App.tsx
  babel.config.js
```

Figure 6.11: Overview of Mobile App dev files

6.3.1 Bluetooth RFCOMM CLIENT API

Before we go to the UI components of the App, we will start by explaining the custom API type hook we built to interact with the cube Bluetooth server. In this section, we will describe some of the important methods in the App. This component is a custom hook called useRFCOMM() which is a Client API we built for connecting and communicating with RFCOMM Bluetooth server devices in React Native. It uses mainly the "react-native-bluetooth-classic" library to scan for, check bluetooth availability, connect to Bluetooth devices, to receive and send messages to the cube's running Bluetooth server. The component has many self-explanatory functions necessary for our app. You can see the overview of it below.



```
export interface BluetoothApi {
  requestPermissions(): Promise<boolean>;
  checkBluetooth(): Promise<boolean>;
  cleanBonded: () => Promise<void>;
  scanForUnpaired(): () => Promise<void>;
  connectToCube: (cube: BluetoothDevice) => Promise<void>;
  disconnectFromCube: () => Promise<void>;
  sendMessage: (cube: BluetoothDevice, msg: string) => void;
  readMsg: (cube: BluetoothDevice) => Promise<void>;
  readGames: (cube: BluetoothDevice) => Promise<void>;
  connectedCube: BluetoothDevice | null | undefined;
  allUnpairedCubes: BluetoothDevice[];
  games: string[];
  isScanning: boolean;
  serverMsg: string;

}
```

FIGURE 6.12: Overview of methods in useRFCOMM()

The API provides several functions for interacting with the Bluetooth devices, such as requestPermissions, checkBluetooth, scanForUnpaired, connectToCube, disconnectFromCube, sendMessage, readMsg, and readGames. The API also includes several state variables configured with useState, such as allUnpairedCubes, connectedCube, games, isScanning, and serverMsg, which can be used to manage and monitor the Bluetooth connection. The meaning behind state variables are:

- allUnpairedCubes: an array of Bluetooth devices that have been discovered during scanning.
- connectedCube: the Bluetooth device that is currently connected, or null if there is no connection.
- games: an array of strings representing the names of games that have been received from the connected device.
- isScanning: a boolean indicating whether the app is currently scanning for Bluetooth devices.
- serverMsg: a string representing the server message received from the connected cube device.

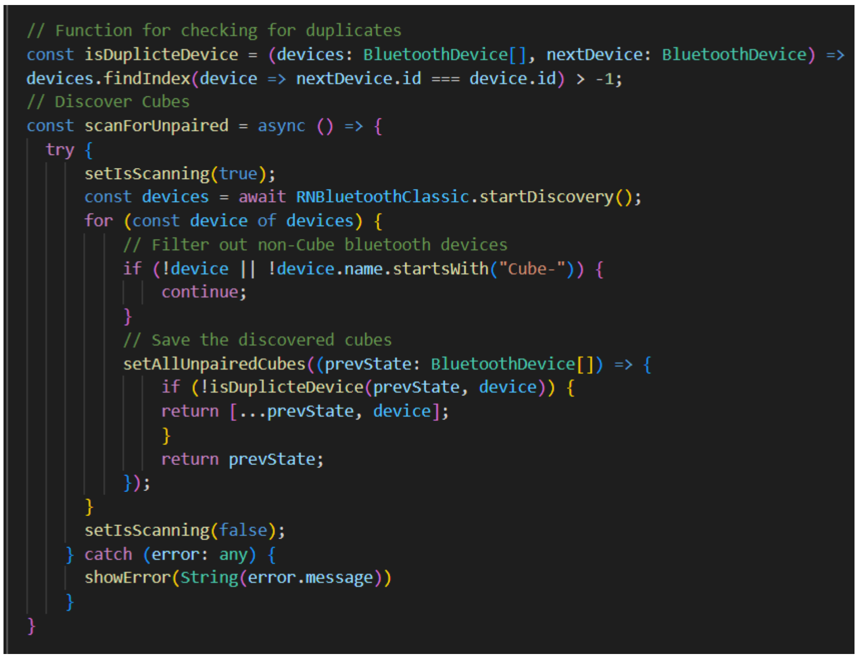
These variables are an important part of the App and used throughout the app to help manage connections.

Furthermore, before using the scanForUnpaired function, it is necessary to request permission to access Bluetooth and location services on the device using the requestPermissions function. Additionally, it is important to check whether Bluetooth is available and enabled on the device using the checkBluetooth function.

- requestPermissions(): a function that requests permission to access Bluetooth and location services on the device with the help from the "react-native-permissions" library. It returns a promise that resolves to a boolean value indicating whether the permissions were granted.
- checkBluetooth(): a function that checks whether Bluetooth is available and enabled on the device. It returns a promise that resolves to a boolean value indicating whether Bluetooth is enabled.

These two functions are usually called before attempting to discover cube devices with scanForUnpaired method. This is the method used in the Devices Discovery component. scanForUnpaired() is a function that uses Bluetooth Classic to scan for nearby Bluetooth devices that have "Cube-" in their name. It then filters out any devices that do not meet this criteria, and checks for duplicates using the isDuplicateDevice function. If the device is not a duplicate, it is added to the allUnpairedCubes state variable using setAllUnpairedCubes().

- scanForUnpaired() is a function that uses Bluetooth Classic to scan for nearby Bluetooth devices that have "Cube-" in their name. It then filters out any devices that do not meet this criteria, and checks for duplicates using the isDuplicateDevice function. If the device is not a duplicate, it is added to the allUnpairedCubes state variable using setAllUnpairedCubes().

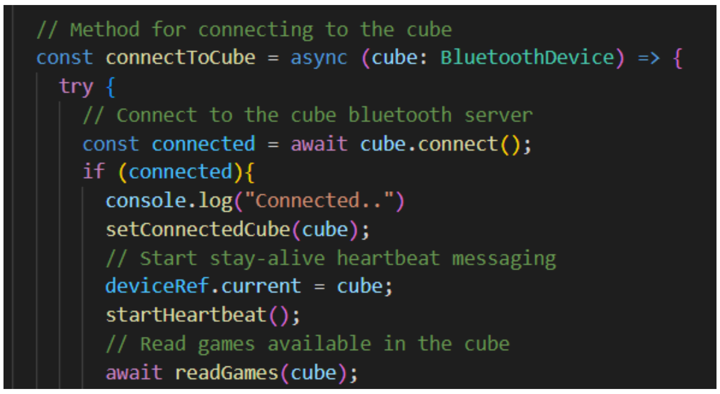


```
// Function for checking for duplicates
const isDuplicteDevice = (devices: BluetoothDevice[], nextDevice: BluetoothDevice) =>
devices.findIndex(device => nextDevice.id === device.id) > -1;
// Discover Cubes
const scanForUnpaired = async () => {
  try {
      setIsScanning(true);
      const devices = await RNBluetoothClassic.startDiscovery();
      for (const device of devices) {
          // Filter out non-Cube bluetooth devices
          if (!device || !device.name.startsWith("Cube-")) {
              continue;
          }
          // Save the discovered cubes
          setAllUnpairedCubes((prevState: BluetoothDevice[]) => {
              if (!isDuplicteDevice(prevState, device)) {
              return [...prevState, device];
              }
              return prevState;
          });
      }
      setIsScanning(false);
    } catch (error: any) {
      showError(String(error.message))
    }
}
```

FIGURE 6.13: Cube device discovery method

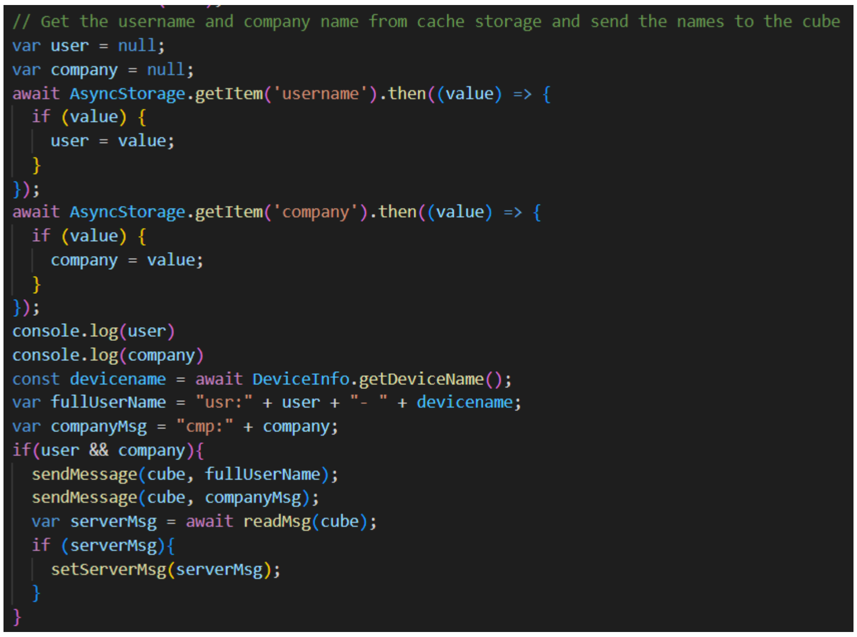
After scanning for Bluetooth cube devices, the application typically displays a list of available devices for the user to choose from and select which cube they want to connect to. For this, we have the connectToCube() method. Quite a few things happen here.

- The connectToCube() function attempts to establish a connection with the specified Bluetooth device by calling the connect method from the Bluetooth classic library on it. If the connection is successful, the function sets the connectedCube state variable to the specified device. It then starts a stay-alive heartbeat messaging process by setting a reference to the device and calling the startHeartbeat function. The startHeartbeat function sets an interval to send a heartbeat message every second using the sendMessage() function. The connectToCube() function also reads the games available in the cube by calling the readGames function, and retrieves the username and company name from cache storage. It then sends the names to the cube, along with the device name obtained using the getDeviceName method of the DeviceInfo library. If both the username and company name are available, the function sends the names to the cube with "usr: <username> - <mobile device name>" and "cmp: <company name>" messages and reads a message from the server. The Server sends the message "EXISTING USER" or "NEW USER" based on whether the logged username is in the database or not.



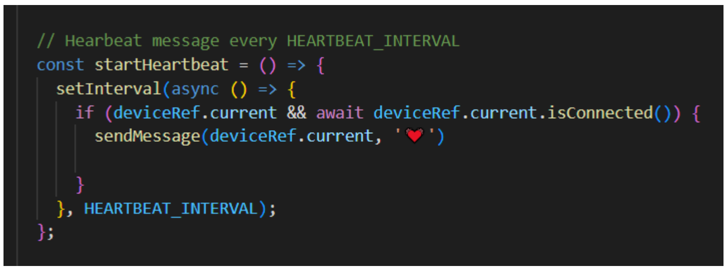
```
// Method for connecting to the cube
const connectToCube = async (cube: BluetoothDevice) => {
  try {
    // Connect to the cube bluetooth server
    const connected = await cube.connect();
    if (connected){
      console.log("Connected..")
      setConnectedCube(cube);
      // Start stay-alive heartbeat messaging
      deviceRef.current = cube;
      startHeartbeat();
      // Read games available in the cube
      await readGames(cube);
```

FIGURE 6.14: connectToCube() method first part



```
// Get the username and company name from cache storage and send the names to the cube
var user = null;
var company = null;
await AsyncStorage.getItem('username').then((value) => {
  if (value) {
    user = value;
  }
});
await AsyncStorage.getItem('company').then((value) => {
  if (value) {
    company = value;
  }
});
console.log(user)
console.log(company)
const devicename = await DeviceInfo.getDeviceName();
var fullUserName = "usr:" + user + "- " + devicename;
var companyMsg = "cmp:" + company;
if(user && company){
  sendMessage(cube, fullUserName);
  sendMessage(cube, companyMsg);
  var serverMsg = await readMsg(cube);
  if (serverMsg){
    setServerMsg(serverMsg);
  }
}
```

FIGURE 6.15: connectToCube() method second part



```
// Hearbeat message every HEARTBEAT_INTERVAL
const startHeartbeat = () => {
  setInterval(async () => {
    if (deviceRef.current && await deviceRef.current.isConnected()) {
      sendMessage(deviceRef.current, '❤')

    }
  }, HEARTBEAT_INTERVAL);
};
```

FIGURE 6.16: stay-alive hearbeat method

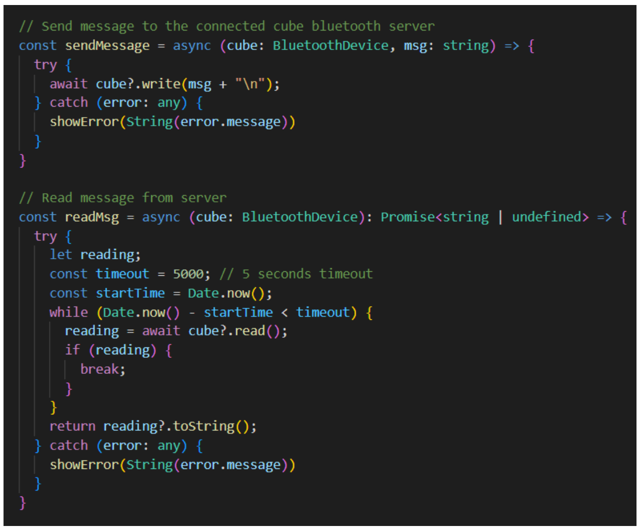
To view available games on the Cube, the connectToCube() method calls readGames().

- readGames() is a simple function that sends a "*get_games*" message to the server, and the server replies with a list of games in the cube.

Moreover, One of the most important functions in the Bluetooth classic library is being able to send and receive messages from the endpoint the device is connected to. With the help of the write and read methods from the library we created our own send and receive message functions.

- The sendMessage() function is responsible for sending a message to the Bluetooth server of the connected cube. It takes in two arguments, cube which is the BluetoothDevice to send the message to, and msg which is the message to be sent. The function uses the write method of the BluetoothDevice object to send the message and appends a newline character at the end of the message.

- The readMsg() function reads a message from the Bluetooth server of the connected cube. It takes in one argument, cube, which is the BluetoothDevice object to read the message from. The function uses a while loop to continuously read for messages until one is found or a timeout of 5 seconds is reached. We do this, since the server is a socket, and sometimes we have to wait a few seconds to receive the message we are waiting to get. Next, It uses the read method of the BluetoothDevice object to read the message and if the message is received it returns the message.



```
// Send message to the connected cube bluetooth server
const sendMessage = async (cube: BluetoothDevice, msg: string) => {
  try {
    await cube?.write(msg + "\n");
  } catch (error: any) {
    showError(String(error.message))
  }
}

// Read message from server
const readMsg = async (cube: BluetoothDevice): Promise<string | undefined> => {
  try {
    let reading;
    const timeout = 5000; // 5 seconds timeout
    const startTime = Date.now();
    while (Date.now() - startTime < timeout) {
      reading = await cube?.read();
      if (reading) {
        break;
      }
    }
    return reading?.toString();
  } catch (error: any) {
    showError(String(error.message))
  }
}
```

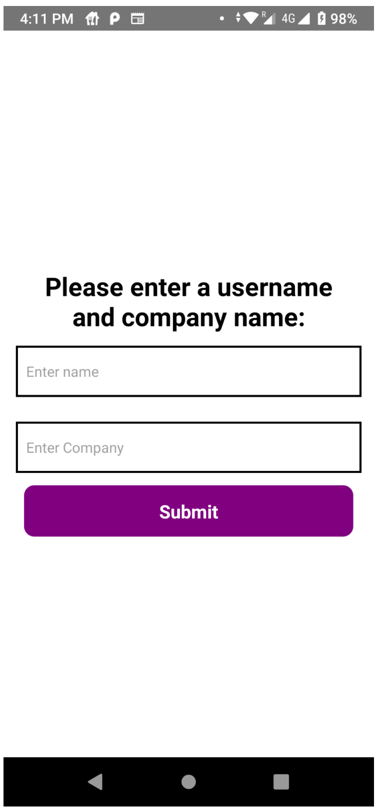
FIGURE 6.17: Send and receive message from RFCOMM server methods

Lastly, to disconnect from the cube we call disconnectFromCube().

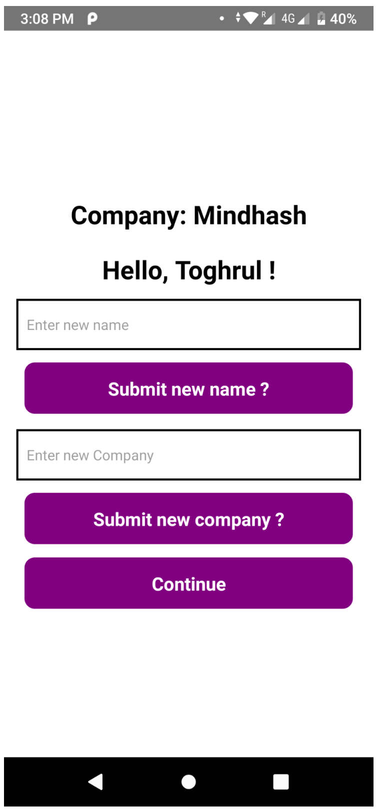
- disconnectFromCube(): This function just disconnects the cube with the Bluetooth device disconnect method from the Bluetooth library, which also stops the heartbeat stay alive messaging. Next, it resets all the necessary state variables to be ready to connect again with a fresh start.

6.3.2 Login component

In the App, we have a simple user system. When a new user installs the app, it will require them to enter a name and a company name. And thus, This component renders a modal with two input fields for a user's name and company and some error handling for input length. The component also stores the name and company in AsyncStorage (Local App Cache) and retrieves them on subsequent renders. Once both the name and company have been entered, a little different login page appears and a continue button is displayed to close the modal. You can see the pictures of the component below.



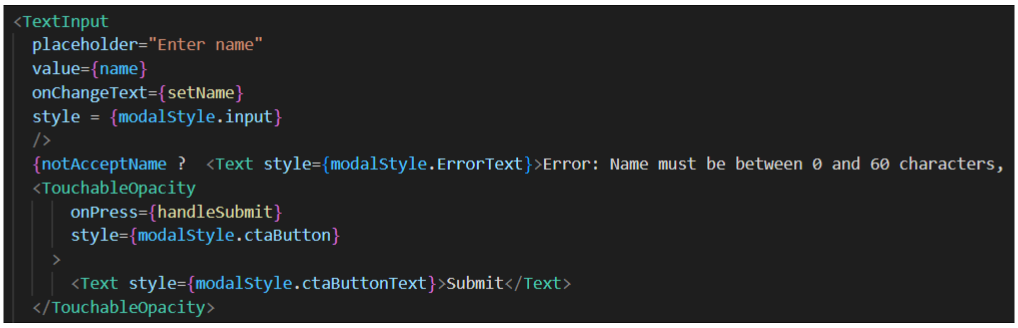
(A) Initial Login page



(B) After you have logged in before

Figure 6.18: Login Modal Component pages

From the above pictures, you can see that we have 2 kinds of Login pages. We use the "Text input" UI component to render the input fields and use "TouchableOpacity" to render the buttons.



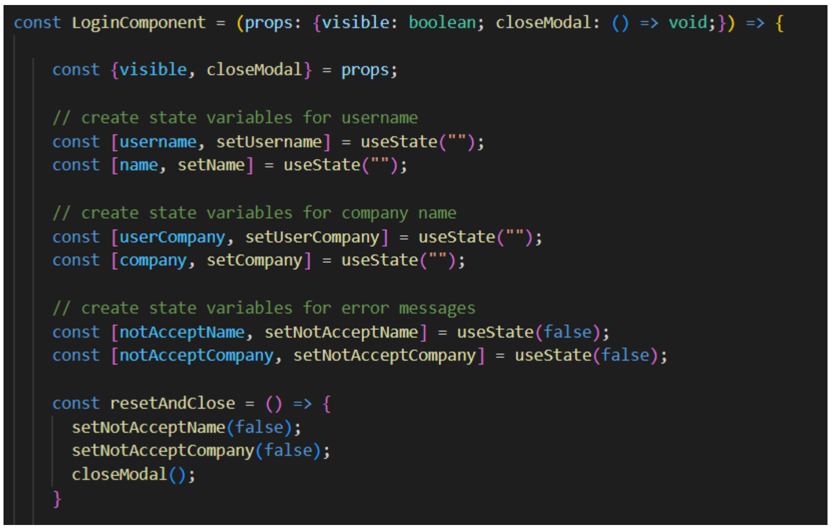
```
<TextInput
  placeholder="Enter name"
  value={name}
  onChangeText={setName}
  style = {modalStyle.input}
  />
  {notAcceptName ?  <Text style={modalStyle.ErrorText}>Error: Name must be between 0 and 60 characters,
  <TouchableOpacity
      onPress={handleSubmit}
      style={modalStyle.ctaButton}
    >
      <Text style={modalStyle.ctaButtonText}>Submit</Text>
  </TouchableOpacity>
```

FIGURE 6.19: Text input form

With help of the onChangeText and value properties of TextInput UI we store the entered names on state variables.

First picture (figure 7.18A) page is the one that appears when the user newly installed the app and hasn't entered name and company name. The user is required to have a name and company name on the app to be able to use its features, once they fill in the input forms they will be able to go to the main page. The second picture (figure 7.18B) page is the one that appears when a user opens the app when they have already filled in their names previously. So, the App remembers the names stored in the cache and opens the corresponding page with a welcome message.

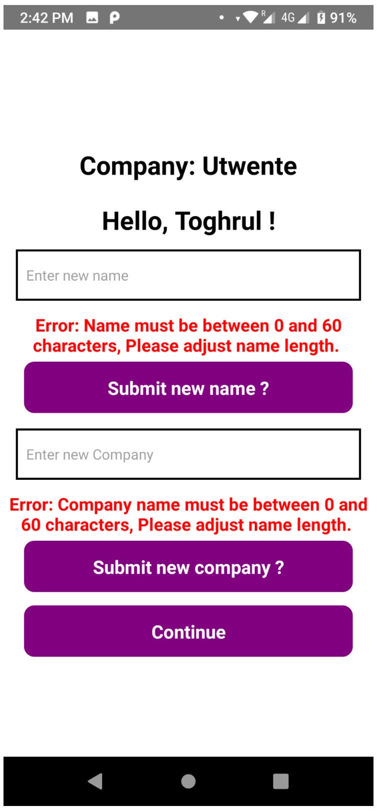
Furthermore, the components use the useState hook to store important state variables. Below, you will see that we have 2 variables for usernames and 2 for company names. For each, one variable is for entered texts in the input fields and another is for saved variables for cache storage. We also have small error handling messages with the help of the error state variables. The component also has a resetAndClose function that sets the notAcceptName and notAcceptCompany state variables to false and calls the closeModal function that is passed in as a prop. This is used when we are finished with this page and want to close and continue to the main page.



```
const LoginComponent = (props: {visible: boolean; closeModal: () => void;}) => {

    const {visible, closeModal} = props;

    // create state variables for username
    const [username, setUsername] = useState("");
    const [name, setName] = useState("");

    // create state variables for company name
    const [userCompany, setUserCompany] = useState("");
    const [company, setCompany] = useState("");

    // create state variables for error messages
    const [notAcceptName, setNotAcceptName] = useState(false);
    const [notAcceptCompany, setNotAcceptCompany] = useState(false);

    const resetAndClose = () => {
      setNotAcceptName(false);
      setNotAcceptCompany(false);
      closeModal();
    }
```

FIGURE 6.20: Login component state variables

The component also has two main functions handleSubmit() and handleSubmitCompany() to handle form submission for the name and company input fields, respectively. Both functions first check if the input is too long (more than 60 characters) and if it is, a state variable notAcceptName or notAcceptCompany is set to true which activates the error messages to be shown under the input fields. If the input is not too long, the corresponding state variable is set to false and the input values are stored in local cache.



2:42 PM 91%

Company: Utwente

Hello, Toghrul !

Enter new name

Error: Name must be between 0 and 60 characters, Please adjust name length.

Submit new name ?

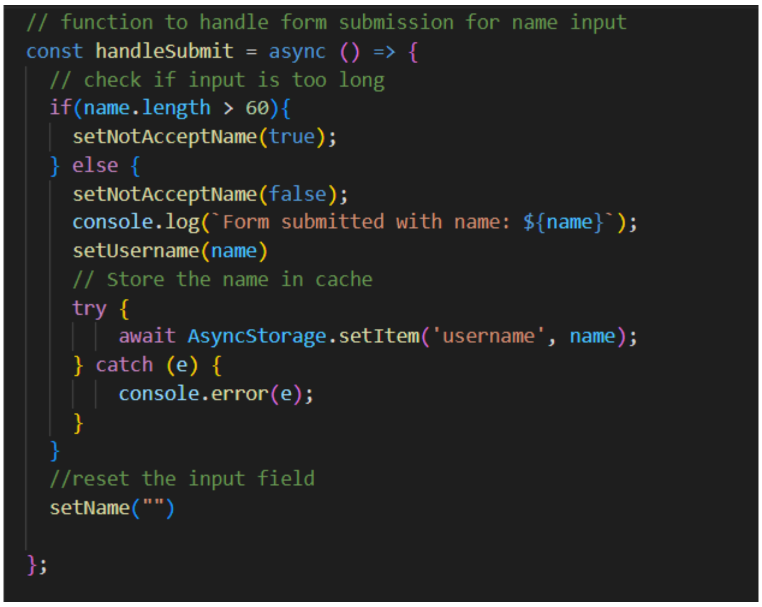
Enter new Company

Error: Company name must be between 0 and 60 characters, Please adjust name length.

Submit new company ?

Continue

FIGURE 6.21: Example login page with error showing

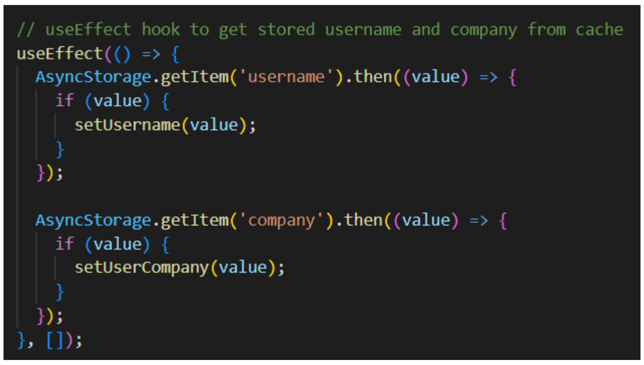


```
// function to handle form submission for name input
const handleSubmit = async () => {
  // check if input is too long
  if(name.length > 60){
    setNotAcceptName(true);
  } else {
    setNotAcceptName(false);
    console.log(`Form submitted with name: ${name}`);
    setUsername(name)
    // Store the name in cache
    try {
        await AsyncStorage.setItem('username', name);
    } catch (e) {
        console.error(e);
    }
  }
  //reset the input field
  setName("")

};
```

FIGURE 6.22: Method for submitting the username

Lastly, we use the useEffect hook to retrieve the stored name and company from Async-

Storage on initial render and store them in state variables username and userCompany.



```
// useEffect hook to get stored username and company from cache
useEffect(() => {
  AsyncStorage.getItem('username').then((value) => {
    if (value) {
      setUsername(value);
    }
  });

  AsyncStorage.getItem('company').then((value) => {
    if (value) {
      setUserCompany(value);
    }
  });
}, []);
```

FIGURE 6.23: Method for retrieving the names from cache on login page

Therefore, the component modal is conditionally rendered based on whether both username and userCompany have values. We use inline coding such as, "username !== "" && userCompany !== "" ? (render initial) : (render regular)" to render the 2 login pages. If the app has both values, the latter login page with continue button is displayed, otherwise the initial input fields and submit button are displayed.

After this, the main page appears, which is just a simple page that asks the user to press the connect button and go to the device discovery modal page.

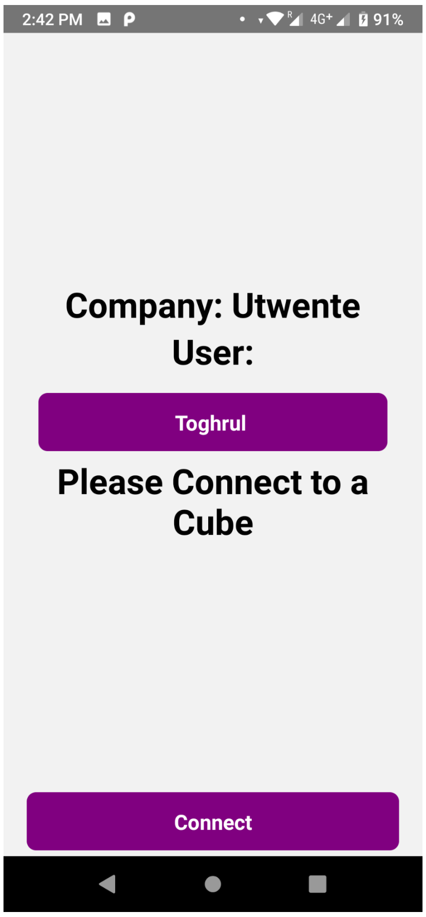
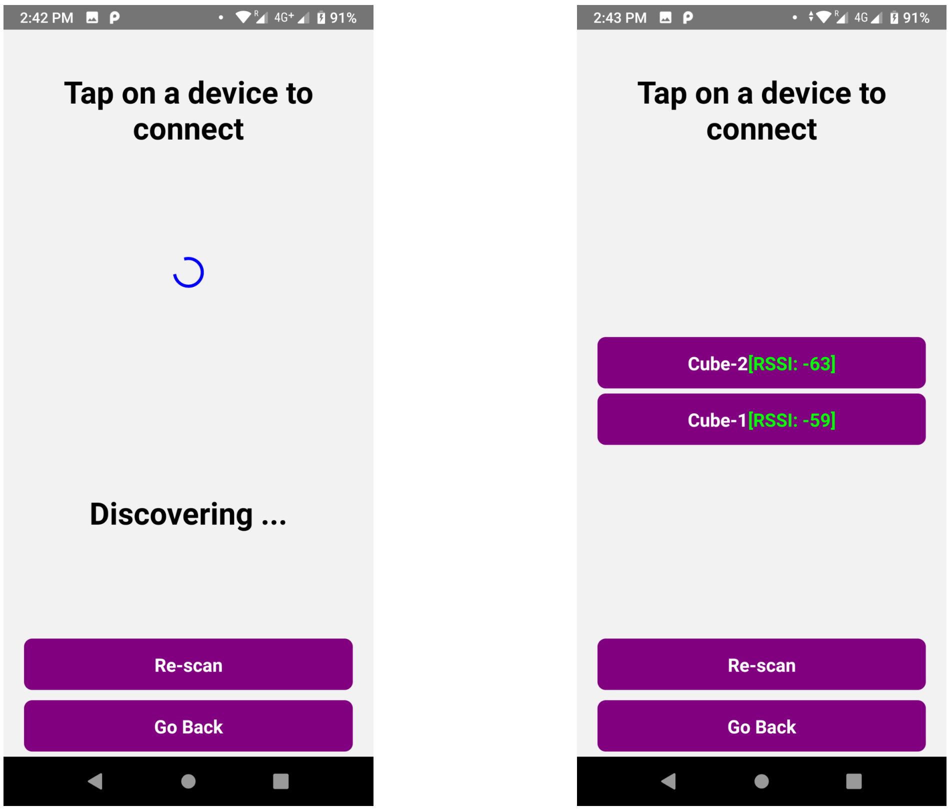


FIGURE 6.24: Main page

6.3.3 Device discovery component



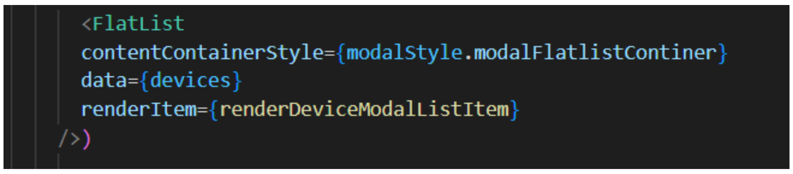
(A) Discovery component while scanning (B) While found devices

FIGURE 6.25: Device discovery component

After a connect button is pressed in the main page we can see the cube device discovery page. This is a functional component that lists nearby Bluetooth devices. The component has a couple of features that define its behavior and appearance. It has 2 functions inside, a main "DeviceModal" function component and a "DeviceModalListItem" helper function.

The DeviceModal component is the main component responsible for rendering the modal dialog. It receives several properties such as an array of BluetoothDevice objects (available cube devices in the area), a boolean indicating whether scanning for devices is in progress. It also has some functions from our useRFCOMM custom hook to scan for devices and connect to a device. Additionally there is open and close modal dialog functions and a spinner.

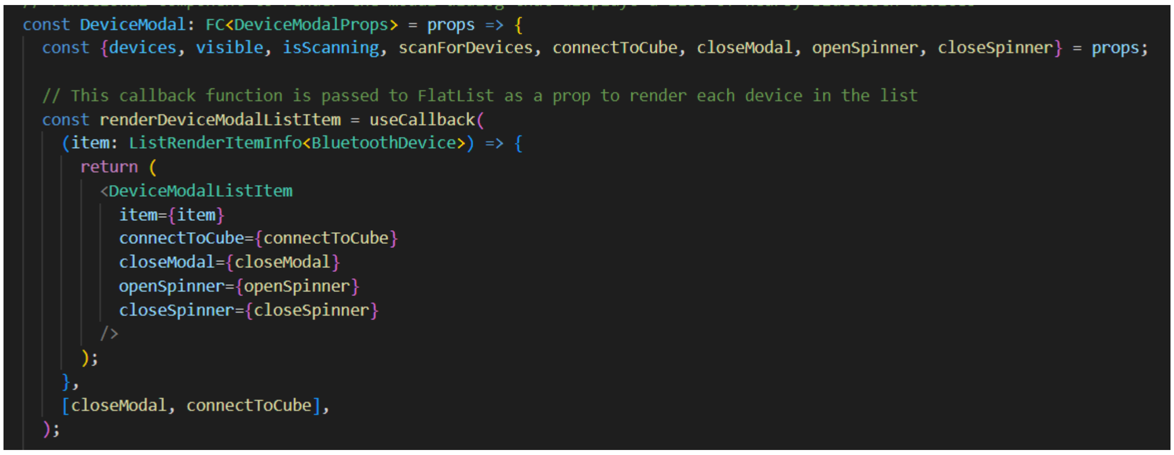
The list of devices is rendered with the help of the FlatList UI component.



```
    <FlatList
    contentContainerStyle={modalStyle.modalFlatlistContiner}
    data={devices}
    renderItem={renderDeviceModalListItem}
  />)
```

FIGURE 6.26: Example defining of FlatList UI

While the discovery function is scanning for devices it displays a spinner with a message "Discovering...", and displays a message "No device found, Please re-scan" when no devices are found after scanning. If devices are found, the FlatList component is rendered with each device shown using the DeviceModalListItem ListItemcomponent.



```
const DeviceModal: FC<DeviceModalProps> = props => {
  const {devices, visible, isScanning, scanForDevices, connectToCube, closeModal, openSpinner, closeSpinner} = props;

  // This callback function is passed to FlatList as a prop to render each device in the list
  const renderDeviceModalListItem = useCallback(
    (item: ListRenderItemInfo<BluetoothDevice>) => {
      return (
        <DeviceModalListItem
          item={item}
          connectToCube={connectToCube}
          closeModal={closeModal}
          openSpinner={openSpinner}
          closeSpinner={closeSpinner}
        />
      );
    },
    [closeModal, connectToCube],
  );
```

FIGURE 6.27: Render device modal item code snippet

The DeviceModalListItem component is responsible for rendering each item in the list of Bluetooth devices in FlatList UI. It receives several objects and functions, including the BluetoothDevice object, a function to connect to the selected device, and functions to open and close a spinner and the modal dialog. It returns a clickable Button for each found device. In the list, we also display RSSI (Received Signal Strength Indicator) for each device to find out which one is the closest cube. When a person chooses and clicks the selected cube, a loading page will appear to forward you to the cube's main page.

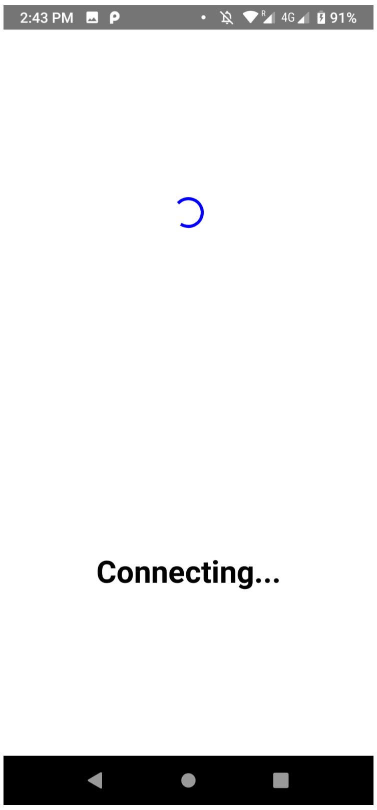
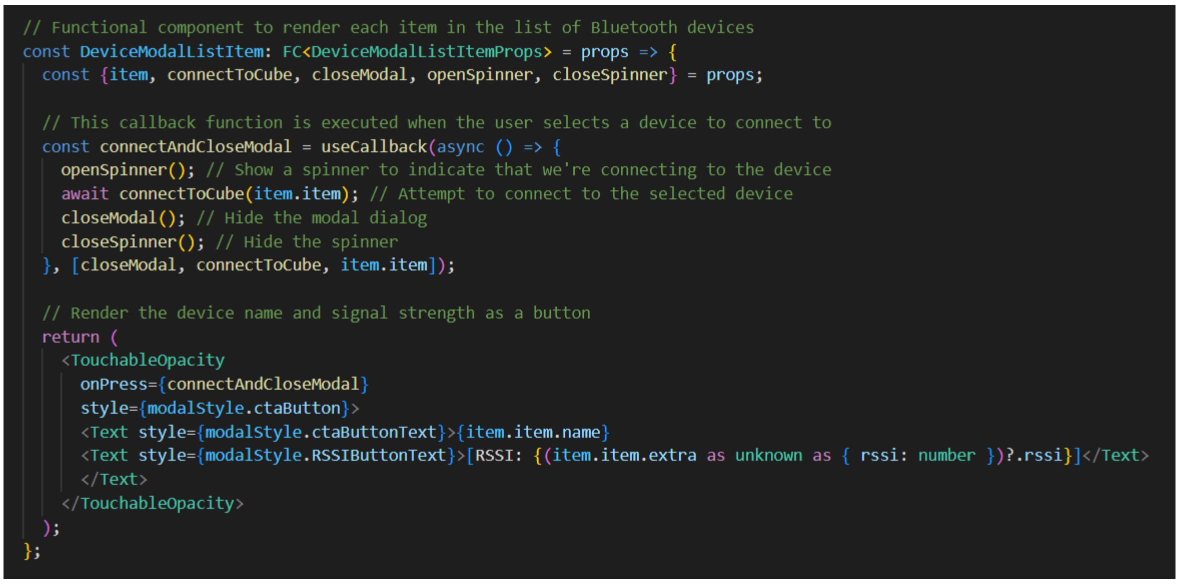


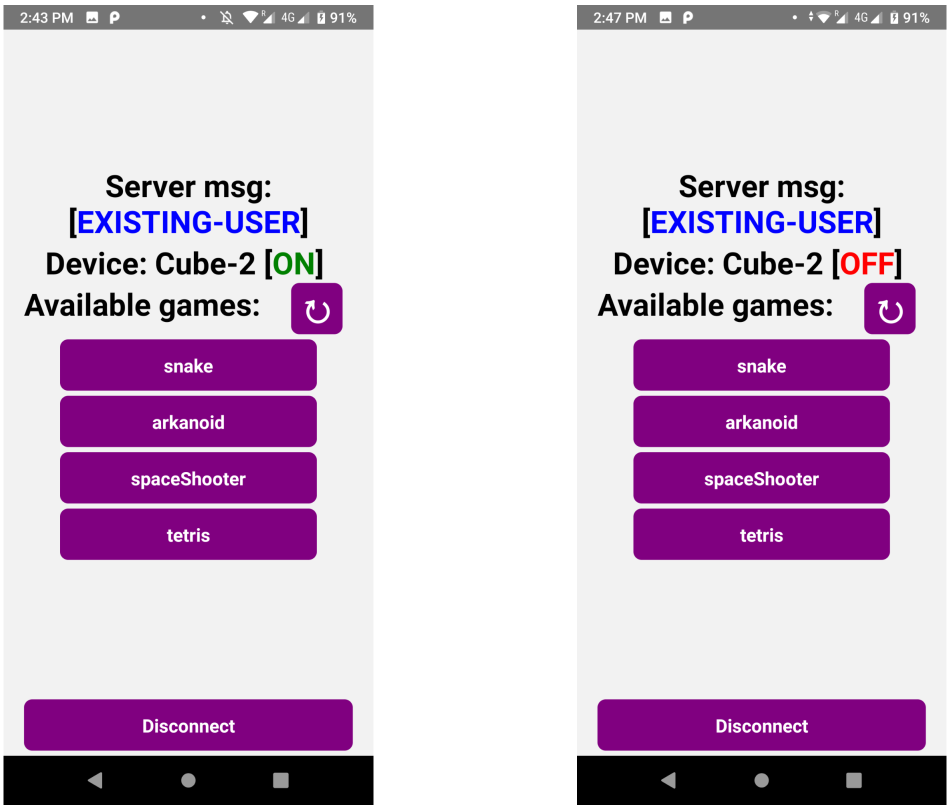
FIGURE 6.28: Loading page



```
// Functional component to render each item in the list of Bluetooth devices
const DeviceModalListItem: FC<DeviceModalListItemProps> = props => {
  const {item, connectToCube, closeModal, openSpinner, closeSpinner} = props;

  // This callback function is executed when the user selects a device to connect to
  const connectAndCloseModal = useCallback(async () => {
    openSpinner(); // Show a spinner to indicate that we're connecting to the device
    await connectToCube(item.item); // Attempt to connect to the selected device
    closeModal(); // Hide the modal dialog
    closeSpinner(); // Hide the spinner
  }, [closeModal, connectToCube, item.item]);

  // Render the device name and signal strength as a button
  return (
    <TouchableOpacity
      onPress={connectAndCloseModal}
      style={modalStyle.ctaButton}>
      <Text style={modalStyle.ctaButtonText}>{item.item.name}
      <Text style={modalStyle.RSSIButtonText}>[RSSI: {(item.item.extra as unknown as { rssi: number })?.rssi}]</Text>
      </Text>
    </TouchableOpacity>
  );
};
```

FIGURE 6.29: Device modal list item component

Overall, the DeviceModal component is responsible for managing the list of Bluetooth devices, and handling user interactions such as selecting a device to connect to and rescanning for devices. The next page we see after we connect is the cube's main page.



(A) when cube ONLINE (B) when cube suddenly goes OFFLINE

FIGURE 6.30: Cube's main page, when it is usually online and when it suddenly disconnects

There are several important fields here. First we see a Server message field. This field is responsible for telling us if we are playing with an existing user in the database or not. As you can see our user is already in the database so it shows an existing user message. If we were to have a newly created user it would show a new user message. It can also show error messages if anything happens wrong in the cube's bluetooth server.

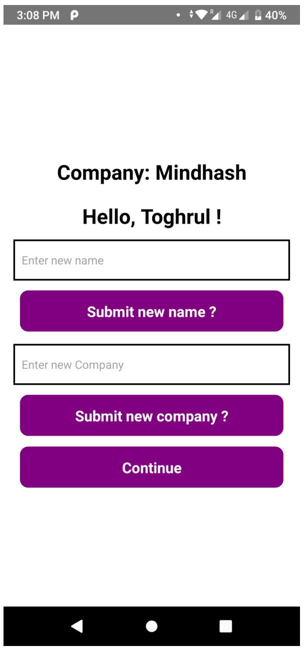
Next, we see a cube availability indicator showing if the Cube is ONLINE or OFFLINE. If the cube suddenly shuts down and the server closes, the app will detect it with the help of the event listener we setted up in the app that listens to the cube and figures out if it is disconnected or not. And will show indication if the cube is offline. You can see an example of this in the above picture.

Furthermore, below those, we have options to choose which game we want to start on the cube. If the app shows no options, we can also hit the refresh button to load the available games again. From the above pictures you can see that we have 4 main games available to play. When a person hits the button to play one of the games it sends the "start game: <game name>" to the server to launch that game.

In the next sections we will show the main app component and how we load game controllers.

6.3.4 The Main App component

In this component all the features we built are combined together. This is the component that renders the regular and main game pages. So let's have an overview of which pages load when a regular user opens the app.


3:08 PM

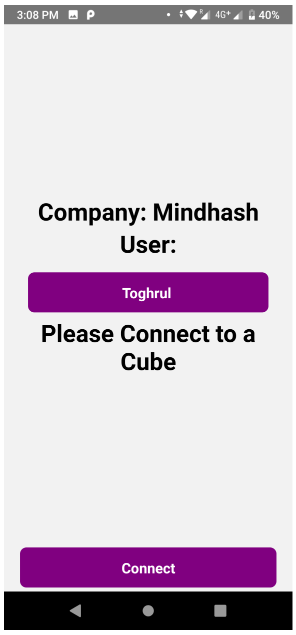
Company: Mindhash

Hello, Toghrul !

Enter new name

Enter new Company

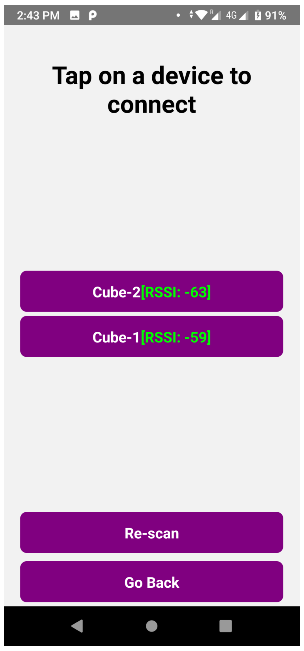
(A) First page


3:08 PM

Company: Mindhash
User:

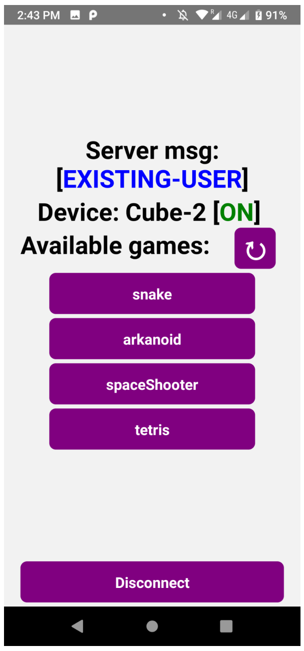
Please Connect to a Cube

(B) Main page


2:43 PM

Tap on a device to connect

(C) Discovery page page


2:43 PM

Server msg: [EXISTING-USER]
Device: Cube-2 [ON]
Available games:

(D) Cube Main page

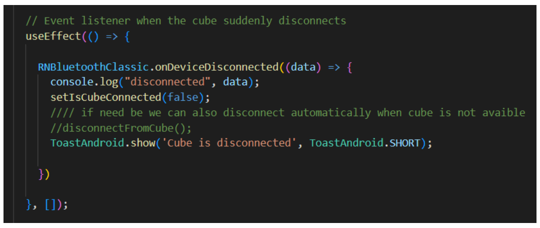
FIGURE 6.31: CubeApp small overview

The functional component named App contains various sub-components, hooks, and state variables.

The main component imports the useState and useEffect hooks from the React library and uses them to define state variables and lifecycle methods, respectively. It also imports several custom hooks from an our custom external module named "useRFCOMM" that handles Bluetooth communication with the Cube device.

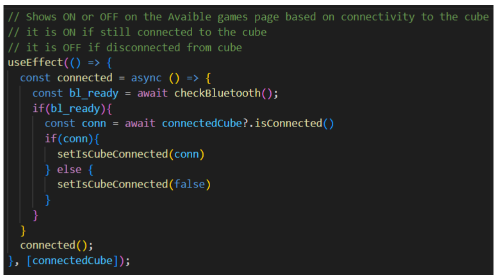
The component contains and puts together its several sub-components that are responsible for handling different functionalities of the application. These sub-components include a login component, a device selection modal, and various game controllers for different games. The login component allows users to log in to the application using their username and company name. The device selection modal allows users to scan for and connect to Cubes via Bluetooth. The game controllers enable users to play different games within the Cube.

Furthermore, It also defines several event listeners using the useEffect hook. These listeners monitor the Bluetooth connection status between the mobile device and the Cube device. They also listen for disconnection events, which are triggered when the Cube device is disconnected from the mobile device.



```
// Event listener when the cube suddenly disconnects
useEffect(() => {

  RNBluetoothClassic.onDeviceDisconnected((data) => {
    console.log("disconnected", data);
    setIsCubeConnected(false);
    //// if need be we can also disconnect automatically when cube is not avaible
    //disconnectFromCube();
    ToastAndroid.show('Cube is disconnected', ToastAndroid.SHORT);

  })

}, []);
```

(A) Listen to disconnected cube

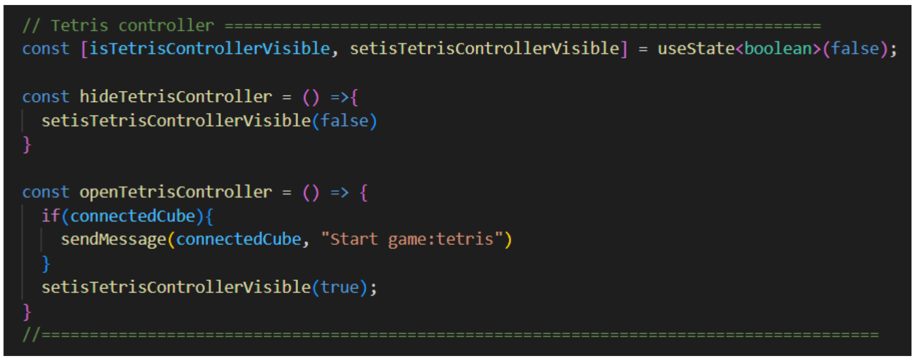


```
// Shows ON or OFF on the Avaible games page based on connectivity to the cube
// it is ON if still connected to the cube
// it is OFF if disconnected from cube
useEffect(() => {
  const connected = async () => {
    const bl_ready = await checkBluetooth();
    if(bl_ready){
      const conn = await connectedCube?.isConnected()
      if(conn){
        setIsCubeConnected(conn)
      } else {
        setIsCubeConnected(false)
      }
    }
  }
  connected();
}, [connectedCube]);
```

(B) Listen to ONLINE or OFFLINE

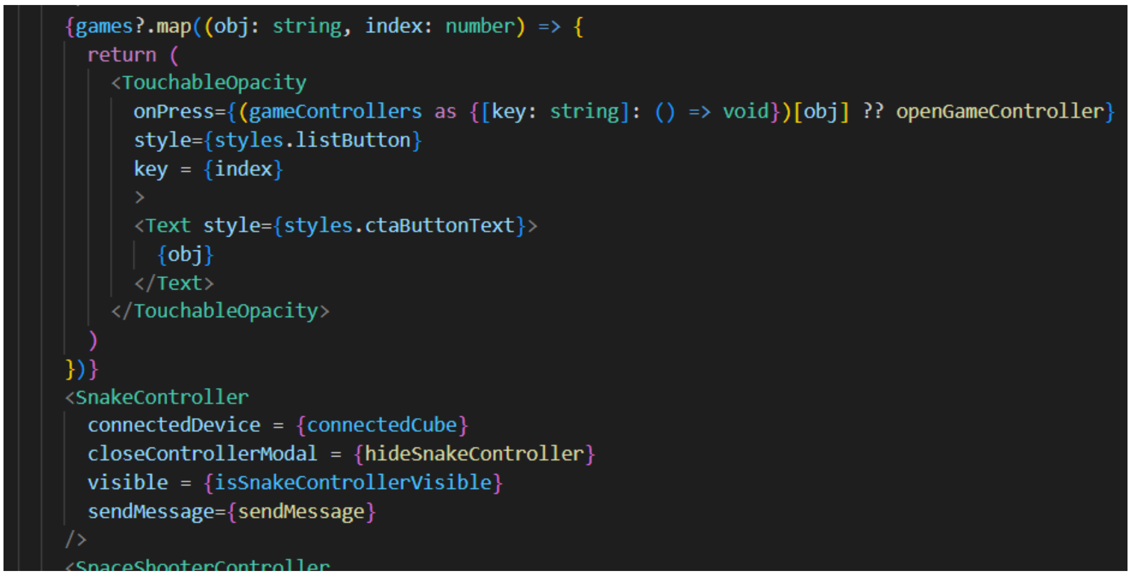
FIGURE 6.32: CubeApp Event listeners

Finally, it defines several functions that handle different functionalities of the application. These functions include scanning for unpaired Cubes, hiding and showing different controller sub-components, and sending messages to start the game in the Cube. An example of how this is handled is also seen below.



```
// Tetris controller ============================================================
const [isTetrisControllerVisible, setisTetrisControllerVisible] = useState<boolean>(false);

const hideTetrisController = () =>{
  setisTetrisControllerVisible(false)
}

const openTetrisController = () => {
  if(connectedCube){
    sendMessage(connectedCube, "Start game:tetris")
  }
  setisTetrisControllerVisible(true);
}
//===============================================================================
```

FIGURE 6.33: Example of how game related functions are in App.tsx

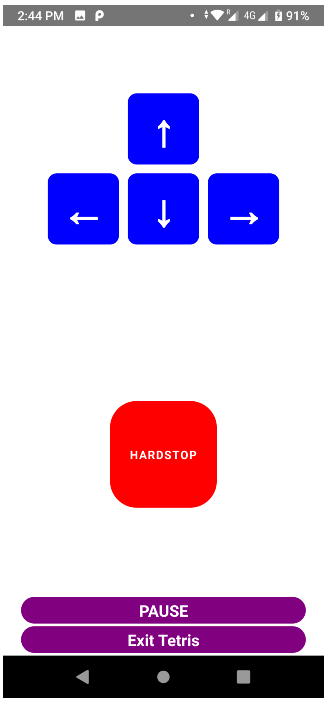


```
{games?.map((obj: string, index: number) => {
  return (
    <TouchableOpacity
      onPress={(gameControllers as {[key: string]: () => void})[obj] ?? openGameController}
      style={styles.listButton}
      key = {index}
      >
      <Text style={styles.ctaButtonText}>
        {obj}
      </Text>
    </TouchableOpacity>
  )
})}
<SnakeController
  connectedDevice = {connectedCube}
  closeControllerModal = {hideSnakeController}
  visible = {isSnakeControllerVisible}
  sendMessage={sendMessage}
/>
```

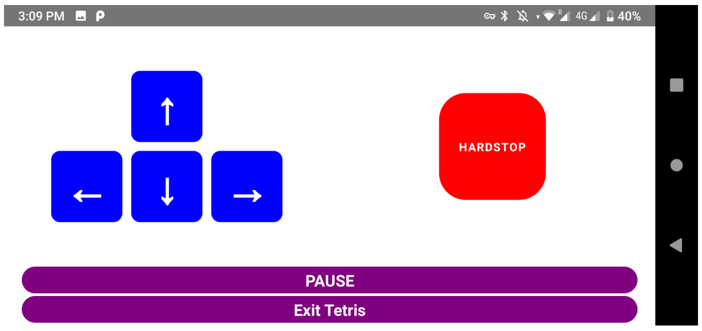
FIGURE 6.34: This is example of how controller modals are loaded in App.tsx

In the next section we will show the game controllers.

6.3.5 Game controllers

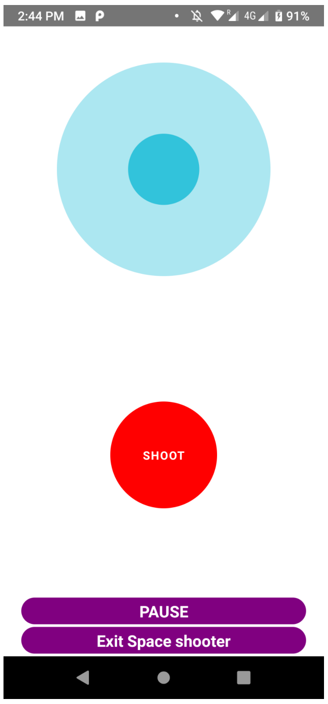


(A) Vertical view

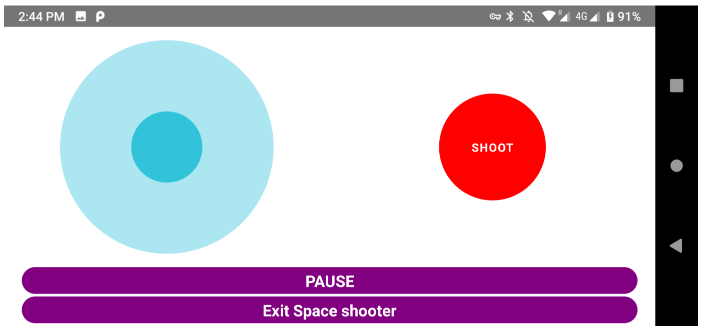


(B) Horizontal view

FIGURE 6.35: Tetris controller

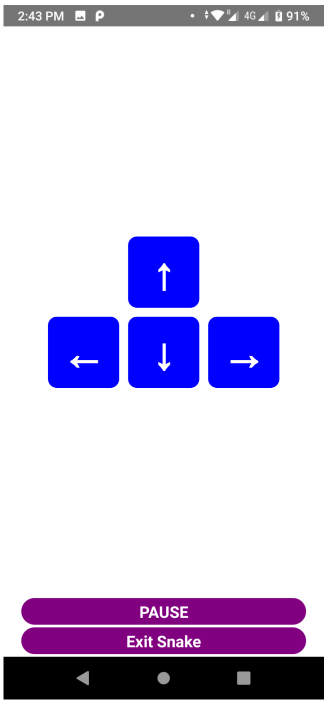


(A) Vertical view

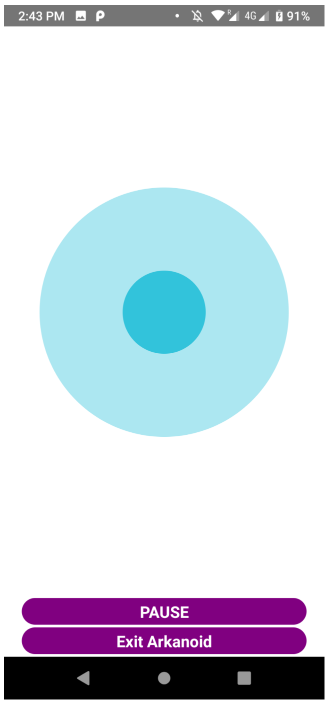


(B) Horizontal view

FIGURE 6.36: Space Shooter controller



(A) Snake controller

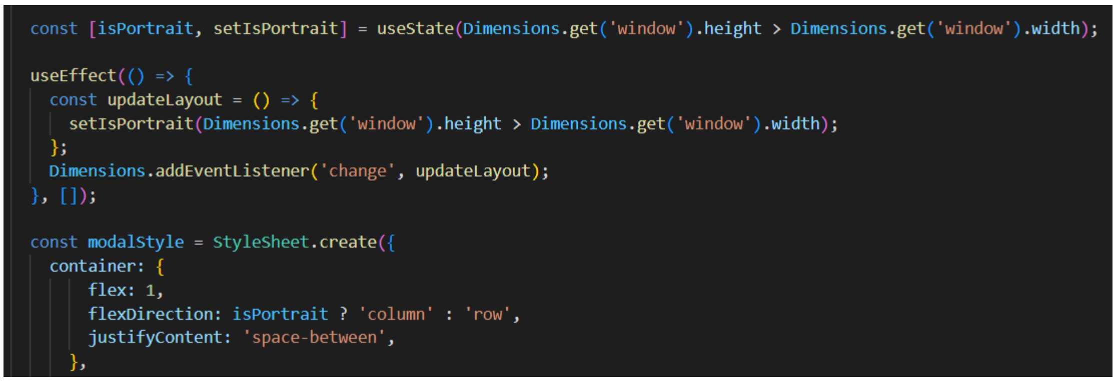


(B) Arkanoid slider controller

FIGURE 6.37: Snake and Arkanoid controllers

All controllers have some similar characteristics with each other. These components send messages to the server where these messages are handled and an action is taken in the corresponding games. We have specific message protocols to handle the controls for the games. We use the sendMessage() function from the useRFCOMM API to send the Messages.

Some of the protocols are similar across other controllers. For example, All the controllers have pause and exit buttons. When the corresponding buttons are pressed it can send the server “PAUSE” or “EXIT” buttons which can pause or exit the game and go back to the games list menu. Another similarity is in between spaceShooter and tetrisController. They both have an extra button which does some action in their corresponding games. So for this we have defined a “ACTION” message protocol for these buttons in the controllers. This same message is sent to the server when the user presses the extra red button, and the server makes the action it should do for the corresponding game the user is playing. Additionally, there is also another small similar feature in tetris and spaceShooter. They both should be able to be rotated to horizontal view. Because otherwise, it is hard for the users to control the games with a vertical view. We achieve a nicer horizontal view by getting Dimensions of the phone and checking if it is in portrait mode or not. When the window dimensions change, the "updateLayout" function is called to update the "isPortrait" state variable based on the new dimensions and we adjust the style of the page based on that. The code snippet for this is in both controllers and can be seen below:

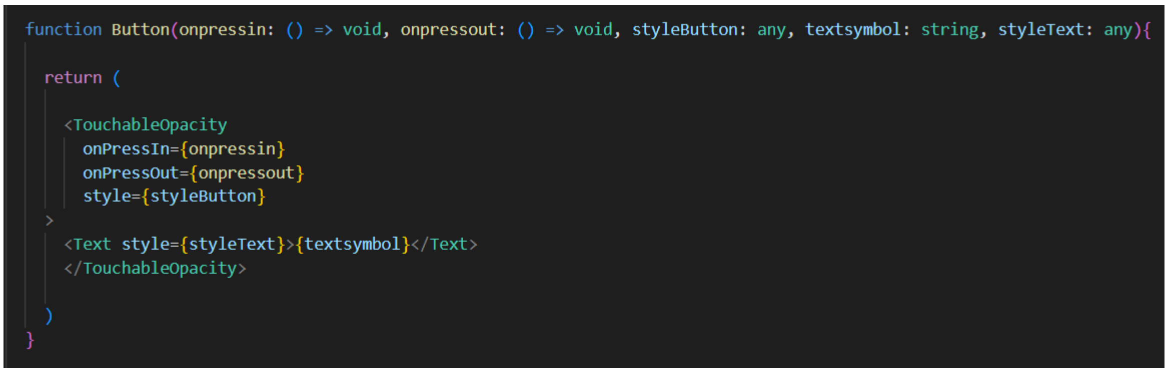


```
const [isPortrait, setIsPortrait] = useState(Dimensions.get('window').height > Dimensions.get('window').width);

useEffect(() => {
  const updateLayout = () => {
    setIsPortrait(Dimensions.get('window').height > Dimensions.get('window').width);
  };
  Dimensions.addEventListener('change', updateLayout);
}, []);

const modalStyle = StyleSheet.create({
  container: {
      flex: 1,
      flexDirection: isPortrait ? 'column' : 'row',
      justifyContent: 'space-between',
    },
```

FIGURE 6.38: Code for handling horizontal view

Furthermore, The SnakeController component allows the user to control the snake game using Bluetooth Classic communication. The component renders a modal that contains six buttons: up, down, left, right, pause, and exit. When the user presses any of these buttons, the corresponding function is called, which sends a message to the connected device using the sendMessage. The message protocol for snake game is simple. When the user presses a button, the component sends a string message to the connected device. The message is a single word indicating which button was pressed: "UP", "DOWN", "LEFT", "RIGHT". The connected device is expected to interpret these messages and control the snake game accordingly.

The tetrisController component allows the user to control the tetris game on the cube with Bluetooth. Here, The component also renders a modal that contains six buttons: up, down, left, right, pause, exit and an additional HardStop button. In case you are familiar with Tetris, the classic game can be played on a computer using four arrow keys and a spacebar. The Up key is used to rotate the falling shape, while the other keys - left, right, and down - are employed to move the shape to the desired position. Additionally, players can press and hold the left, right, or down keys to make the shape go in the corresponding direction more rapidly. Moreover, there is a hardstop button that can be activated by pressing the spacebar in the PC version of the game, causing the shape to drop instantly to the bottom. In the App we have simulated all these controls. The component defines functions for handling button presses, such as handleUp, handleLeft, handleDown, handleRight, handleHardStop, handleLeftOut, handleRightOut, and handleDownOut. In the TouchableOpacity mobile UI button component, besides the regular onPress prop, it also has onPressIn and onPressOut functionalities. These are basically props, for which we can assign actions for when we press the button and when we remove our finger from the button. This action simulates holding down the key in the keyboard in the App's controller. Below you can see the example function we made for this kind of button.

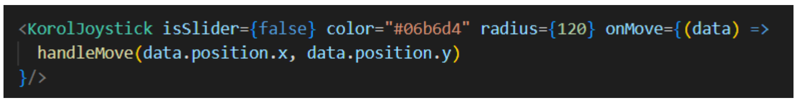


```
function Button(onpressin: () => void, onpressout: () => void, styleButton: any, textsymbol: string, styleText: any){

  return (

    <TouchableOpacity
      onPressIn={onpressin}
      onPressOut={onpressout}
      style={styleButton}
  >
    <Text style={styleText}>{textsymbol}</Text>
    </TouchableOpacity>

  )
}
```

FIGURE 6.39: Function for creating buttons which you can press hold

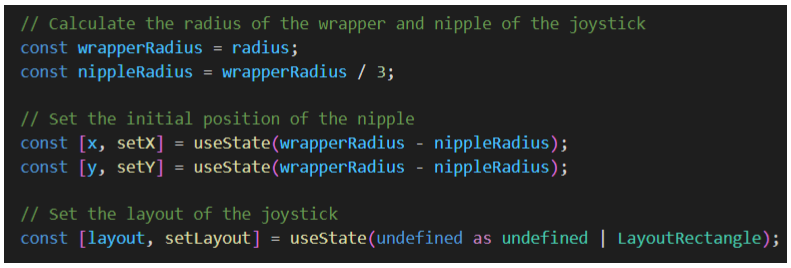
These functions mentioned above can send "UP", "LEFT", "LEFTOUT", "RIGHT", RIGHTOUT", "DOWN", "DOWNOUT" and an "ACTION" message which simulates hardstop in the game. These Bluetooth messages in the connected device server are appropriately handled for the tetris game.

Besides the arrow keys we have built a custom Joystick component. The component imports several modules from the React and React Native libraries, including useState, View, and LayoutRectangle. It also imports Gesture, GestureDetector, GestureStateChangeEvent, GestureTouchEvent, and GestureHandlerRootView from the react-native-gesture-handler library. With these components it is able to create a circular view and listen to the touch and gesture events to simulate the joystick. In the Joystick component we have several important variables we keep. First, we define wrapper and joystick nipple radius sizes we get from when we create a joystick of specific size. Then we initialize the position of the joystick center element, the "nipple". Next, we define a variable to hold the dimensions of the rectangular UI component that encloses the joystick. By doing so, we can ensure that the joystick remains compatible with different mobile devices, regardless of their screen layouts. This approach avoids hardcoding the dimensions based on the screen size of the device, making the joystick adaptable to various screen sizes and resolutions.



```
<KorolJoystick isSlider={false} color="#06b6d4" radius={120} onMove={(data) =>
  handleMove(data.position.x, data.position.y)
}/>
```

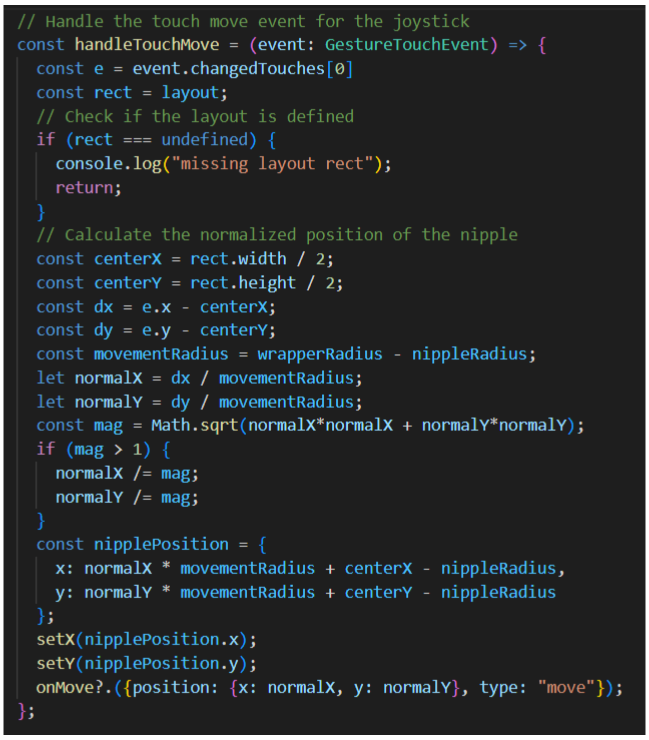
FIGURE 6.40: Example of how we define our custom Joystick in UI



```
// Calculate the radius of the wrapper and nipple of the joystick
const wrapperRadius = radius;
const nippleRadius = wrapperRadius / 3;

// Set the initial position of the nipple
const [x, setX] = useState(wrapperRadius - nippleRadius);
const [y, setY] = useState(wrapperRadius - nippleRadius);

// Set the layout of the joystick
const [layout, setLayout] = useState(undefined as undefined | LayoutRectangle);
```

FIGURE 6.41: Joystick state variables

Moreover, the most important function in the Joystick component is handleTouch-Move() function which takes an event object that contains information about the touch event and updates the position of the joystick nipple accordingly. When a touch movement event occurs on the joystick, the handleTouchMove function is called. It retrieves the touch event position and joystick layout from the state variable. The function then calculates the normalized position of the nipple relative to the center of the joystick wrapper by dividing the distance between the touch position and center by the maximum distance the nipple can move. This result is maintained to the range of -1 to 1. The function then calculates the magnitude of the normalized position vector and, if it is greater than 1, normalizes the vector by dividing its components by the magnitude. The nipple position is then calculated based on the normalized position and the radius of the joystick wrapper and nipple. Finally, the function updates the state variables for the nipple position and calls a callback function with the new position data.



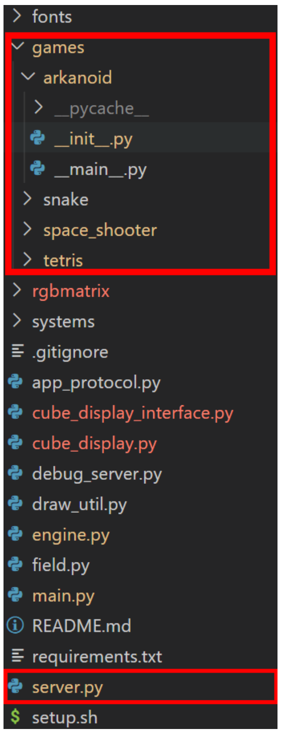
```
// Handle the touch move event for the joystick
const handleTouchMove = (event: GestureTouchEvent) => {
  const e = event.changedTouches[0]
  const rect = layout;
  // Check if the layout is defined
  if (rect === undefined) {
    console.log("missing layout rect");
    return;
  }
  // Calculate the normalized position of the nipple
  const centerX = rect.width / 2;
  const centerY = rect.height / 2;
  const dx = e.x - centerX;
  const dy = e.y - centerY;
  const movementRadius = wrapperRadius - nippleRadius;
  let normalX = dx / movementRadius;
  let normalY = dy / movementRadius;
  const mag = Math.sqrt(normalX*normalX + normalY*normalY);
  if (mag > 1) {
    normalX /= mag;
    normalY /= mag;
  }
  const nipplePosition = {
    x: normalX * movementRadius + centerX - nippleRadius,
    y: normalY * movementRadius + centerY - nippleRadius
  };
  setX(nipplePosition.x);
  setY(nipplePosition.y);
  onMove?.({position: {x: normalX, y: normalY}, type: "move"});
};
```

FIGURE 6.42: Joystick's main important method

The spaceShooter controller uses the custom built Joystick and a shoot button which sends the "ACTION" message to the server. The Arkanoid also has the same Joystick as spaceShooter with only difference being, the arkanoid's joystick is limited to go only up and down, since that's how an arkanoid controller should be, it should be a slider. Meanwhile the ship in spaceShooter can go into all directions, so it has the full potential of our custom built Joystick. The Message Protocol for the Joystick is also simple. When we move the joystick it generates x and y coordinate values in range between -1 and 1. When that happens we send the message "$J \sim< xvalue >\sim< yvalue >$" to the server, and the server makes the ship or the slider in arkanoid move according to it. In the arkanoid controller it is also the same message, it is just in arkanoid the x value is always 0 since it can only go up and down.

6.4 Software for the Cube

The software for the cube is organised as follows: the root of the project contains the entry point server.py and some commonly used files, and the different games each have a subfolder inside of the games folder. Each game folder contains atleast an __init__.py file which contains the main game loop and a __main__.py file which is used to launch the games individually for testing. One of the more important files in the root is engine.py as it contains the interface games use to poll for input and display to the LED displays on the cube.



```
> fonts
v games
  v arkanoid
    > __pycache__
      __init__.py
      __main__.py
  > snake
  > space_shooter
  > tetris
> rgbmatrix
> systems
  .gitignore
  app_protocol.py
  cube_display_interface.py
  cube_display.py
  debug_server.py
  draw_util.py
  engine.py
  field.py
  main.py
  README.md
  requirements.txt
  server.py
$ setup.sh
```

FIGURE 6.43: Cube software folder structure

6.4.1 Bluetooth server

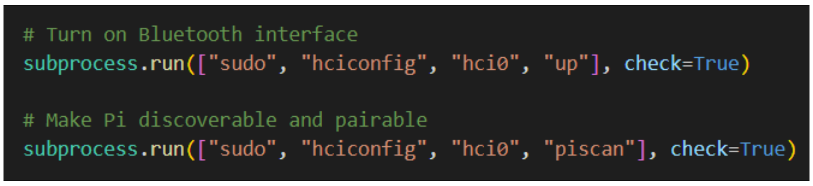
The Bluetooth server is a software component in the raspberry pi that operates the cube and it facilitates communication between Bluetooth-enabled mobile devices. The server is written in Python since it is very compatible with raspberry pi, and since all the games and LED display library has nice python support. For creating games there is a pygame library and for interacting with LED displays we have the "rgb-rpi-led-matrix" library python binding.

For the CubeApp, the server acts as a central hub, connecting users' mobile devices and transmitting game controller input commands to the game in the cube. Additionally, it initiates games, provides access to 4 arcade games that can be played on a LED cube display and tracks scores, sending them to the leaderboard backend server for display.

The server is responsible for managing various data types such as game data, user inputs, and scores. It processes this data in real-time, ensuring smooth gameplay for users. It is also responsible for handling the connection between the server and mobile devices, guaranteeing that the data is transmitted accurately and efficiently.

In the Cube the server script is the only code we run, it is the main python file that has access to all the other code in the cube (e.g games files). Here is an overview of the steps involved in the server script:

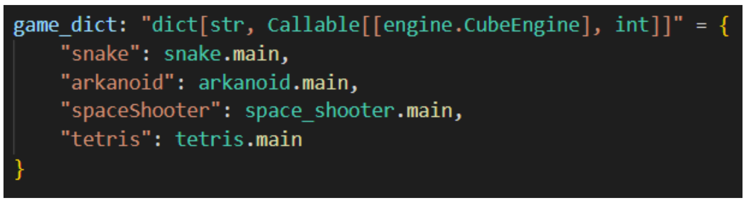
1. Importing libraries, such as subprocess for executing shell commands, pygame for drawing games, and bluetooth for creating a Bluetooth server. Since this is the initial file we run on the cube, before we run the bluetooth socket server, we have to make sure that the Pi has enabled bluetooth and it is in pairable and discoverable mode. We achieve this with the below figure example commands. After this we also set a name for the raspberry pi to be discovered for with the help of the "hciconfig hci0 name <Cube Name>" command.



```python
# Turn on Bluetooth interface
subprocess.run(["sudo", "hciconfig", "hci0", "up"], check=True)

# Make Pi discoverable and pairable
subprocess.run(["sudo", "hciconfig", "hci0", "piscan"], check=True)
```

FIGURE 6.44: subprocess library example where we make pi discoverable and pairable

2. Define a dictionary called "game_dict" that maps game names to their main functions, which are later used to launch the game.

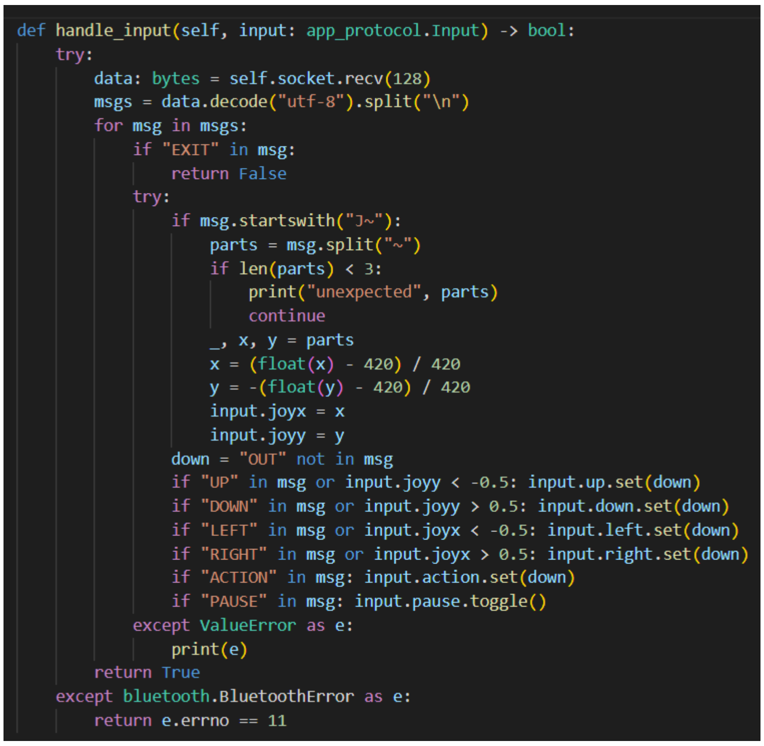


```python
game_dict: "dict[str, Callable[[engine.CubeEngine], int]]" = {
    "snake": snake.main,
    "arkanoid": arkanoid.main,
    "spaceShooter": space_shooter.main,
    "tetris": tetris.main
}
```

FIGURE 6.45: games dictionary that contains main method of the games as values

3. Create a BluetoothClient class that handles Bluetooth communication between the server and app inside our "server.py" file. Inside the Bluetooth client class, we have

the main handle_input function. This is where we handle all the game related inputs, and make the proper actions inside games.



```
def handle_input(self, input: app_protocol.Input) -> bool:
    try:
        data: bytes = self.socket.recv(128)
        msgs = data.decode("utf-8").split("\n")
        for msg in msgs:
            if "EXIT" in msg:
                return False
            try:
                if msg.startswith("J~"):
                    parts = msg.split("~")
                    if len(parts) < 3:
                        print("unexpected", parts)
                        continue
                    _, x, y = parts
                    x = (float(x) - 420) / 420
                    y = -(float(y) - 420) / 420
                    input.joyx = x
                    input.joyy = y
                down = "OUT" not in msg
                if "UP" in msg or input.joyy < -0.5: input.up.set(down)
                if "DOWN" in msg or input.joyy > 0.5: input.down.set(down)
                if "LEFT" in msg or input.joyx < -0.5: input.left.set(down)
                if "RIGHT" in msg or input.joyx > 0.5: input.right.set(down)
                if "ACTION" in msg: input.action.set(down)
                if "PAUSE" in msg: input.pause.toggle()
            except ValueError as e:
                print(e)
        return True
    except bluetooth.BluetoothError as e:
        return e.errno == 11
```

FIGURE 6.46: Bluetooth client class's main handle input method

The function first receives data from the server's Bluetooth socket with a maximum size of 128 bytes. It then splits the data into individual messages using the newline character ("\n") as a delimiter. For each message, the function checks if it contains the keyword "EXIT". If it does, the function returns False, indicating that the controller has exited the game.

If the message starts with "J ", the function assumes it contains joystick data and extracts the x and y values from the message. It then converts these values back to a range of -1 to 1. The reason why it does that is because in the App we send the x, and y values like this "x = Math.round(x * 420) + 420;, y = Math.round(y * 420) + 420;". Since this is a bluetooth communication, we have to make sure that data is scaled for easy transmission to the server. And on the server we convert it back. Later, we set the "joyx" and "joyy" properties of the input object accordingly.

The function then checks for other keywords such as "UP", "DOWN", "LEFT", "RIGHT", "OUT", "ACTION", and "PAUSE" in the message. If any of these keywords are present, the function sets the corresponding properties of the input object based on the message content.

If an error occurs while receiving data from the socket, the function checks if the error code is 11 (which indicates that the socket is temporarily unavailable) and returns True if it is. Otherwise, it returns False.

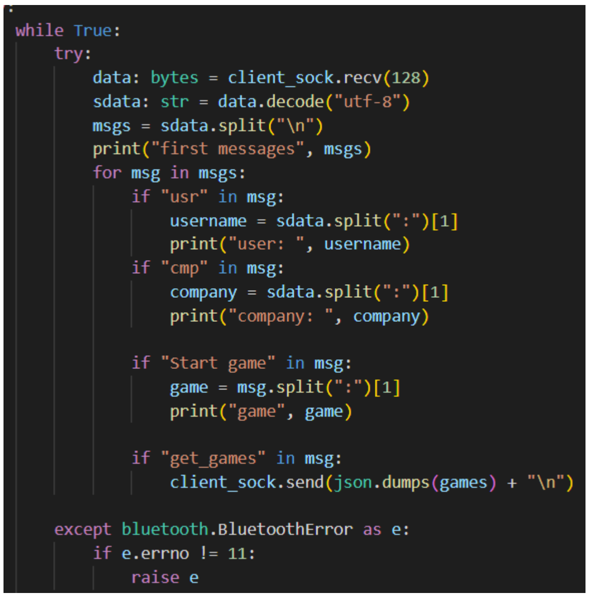
4. Create a cube variable with CubeEngine instance that manages the LED display and Bluetooth communication, and set the show_on_cube parameter to True to display

the game on the LED cube. CubeEngine is a custom class we build to help us display games on a LED matrix.

5. Enter a loop that listens for incoming Bluetooth connections.

6. After a client has connected, create a BluetoothClient instance to handle input from the client.

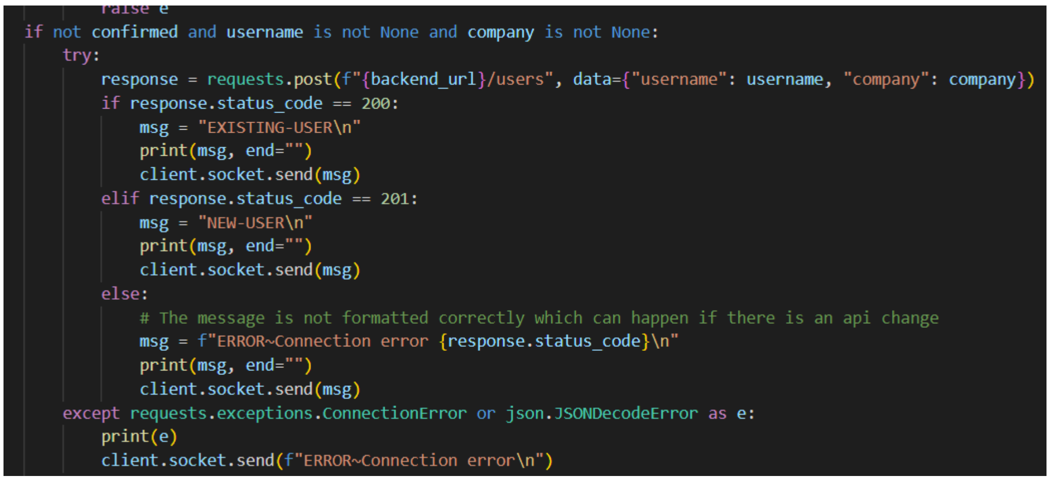
7. Enter a loop that waits for receiving variables like username, company and request for games list from the client app.



```
while True:
    try:
        data: bytes = client_sock.recv(128)
        sdata: str = data.decode("utf-8")
        msgs = sdata.split("\n")
        print("first messages", msgs)
        for msg in msgs:
            if "usr" in msg:
                username = sdata.split(":")[1]
                print("user: ", username)
            if "cmp" in msg:
                company = sdata.split(":")[1]
                print("company: ", company)

            if "Start game" in msg:
                game = msg.split(":")[1]
                print("game", game)

            if "get_games" in msg:
                client_sock.send(json.dumps(games) + "\n")

    except bluetooth.BluetoothError as e:
        if e.errno != 11:
            raise e
```

FIGURE 6.47: Handling initialization of important variables

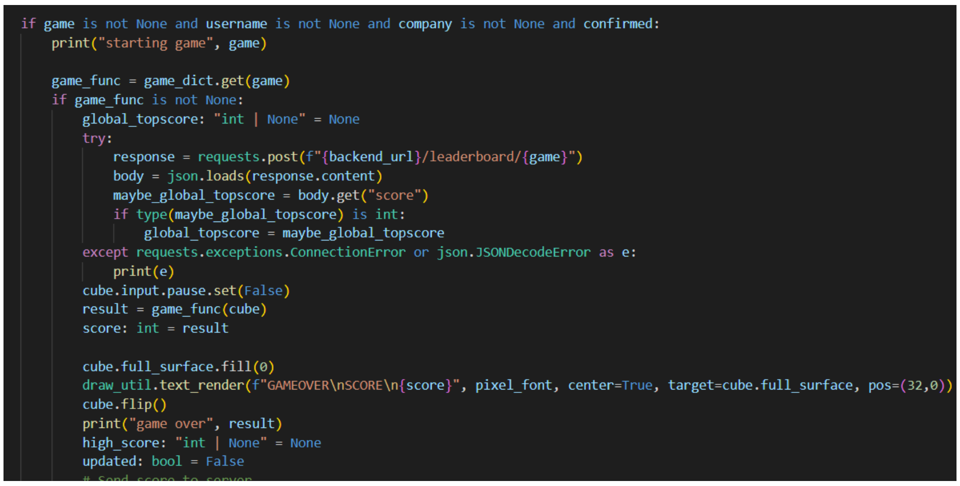
8. Once username and company name are received, these values are stored in username and company. After we have a username and company name set in the variables, the server checks to see if it needs to create a new user in the database or not.



```
        raise e
if not confirmed and username is not None and company is not None:
    try:
        response = requests.post(f"{backend_url}/users", data={"username": username, "company": company})
        if response.status_code == 200:
            msg = "EXISTING-USER\n"
            print(msg, end="")
            client.socket.send(msg)
        elif response.status_code == 201:
            msg = "NEW-USER\n"
            print(msg, end="")
            client.socket.send(msg)
        else:
            # The message is not formatted correctly which can happen if there is an api change
            msg = f"ERROR~Connection error {response.status_code}\n"
            print(msg, end="")
            client.socket.send(msg)
    except requests.exceptions.ConnectionError or json.JSONDecodeError as e:
        print(e)
        client.socket.send(f"ERROR~Connection error\n")
```

FIGURE 6.48: Handling new user and existing users in the server

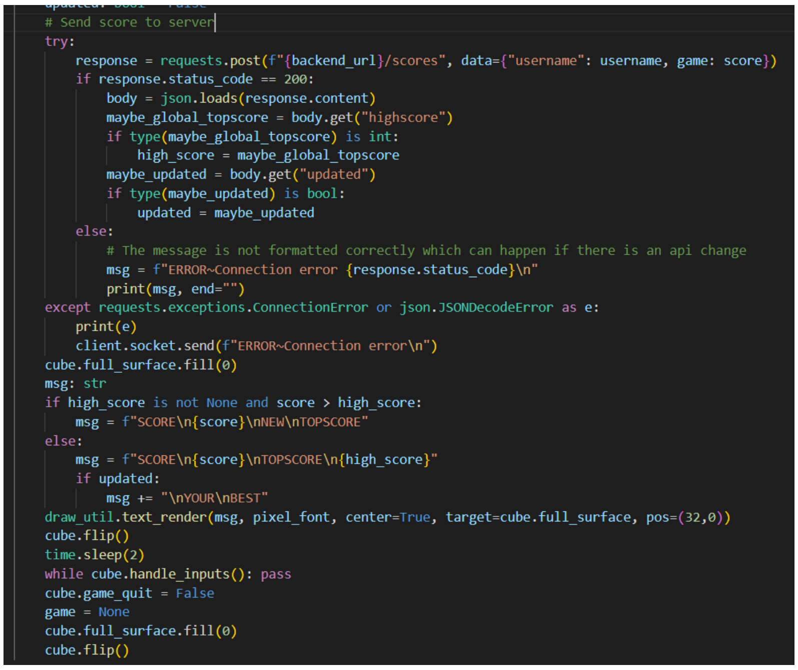
9. After a request like “get_games” is received, the server sends the list of available games to the client.
10. Wait for the client to select a game to play.
11. Launch the selected game by calling the corresponding function in game_dict.
12. While the game is running, wait for input from the client and pass it to the game. It also retrieves the high score for the game from the webpage’s backend server and displays it on the LED cube. The game runs until it is over, and the user’s score is calculated.



```python
if game is not None and username is not None and company is not None and confirmed:
    print("starting game", game)

    game_func = game_dict.get(game)
    if game_func is not None:
        global_topscore: "int | None" = None
        try:
            response = requests.post(f"{backend_url}/leaderboard/{game}")
            body = json.loads(response.content)
            maybe_global_topscore = body.get("score")
            if type(maybe_global_topscore) is int:
                global_topscore = maybe_global_topscore
        except requests.exceptions.ConnectionError or json.JSONDecodeError as e:
            print(e)
        cube.input.pause.set(False)
        result = game_func(cube)
        score: int = result

        cube.full_surface.fill(0)
        draw_util.text_render(f"GAMEOVER\nSCORE\n{score}", pixel_font, center=True, target=cube.full_surface, pos=(32,0))
        cube.flip()
        print("game over", result)
        high_score: "int | None" = None
        updated: bool = False
        # Send score to server
```

FIGURE 6.49: Game launch and getting highscore of the game from backend server

13. Once the game has ended, send the score to a central server and request the high score for that game.



```python
updated: bool = False
# Send score to server
try:
    response = requests.post(f"{backend_url}/scores", data={"username": username, game: score})
    if response.status_code == 200:
        body = json.loads(response.content)
        maybe_global_topscore = body.get("highscore")
        if type(maybe_global_topscore) is int:
            high_score = maybe_global_topscore
        maybe_updated = body.get("updated")
        if type(maybe_updated) is bool:
            updated = maybe_updated
    else:
        # The message is not formatted correctly which can happen if there is an api change
        msg = f"ERROR~Connection error {response.status_code}\n"
        print(msg, end="")
except requests.exceptions.ConnectionError or json.JSONDecodeError as e:
    print(e)
    client.socket.send(f"ERROR~Connection error\n")
cube.full_surface.fill(0)
msg: str
if high_score is not None and score > high_score:
    msg = f"SCORE\n{score}\nNEW\nTOPSCORE"
else:
    msg = f"SCORE\n{score}\nTOPSCORE\n{high_score}"
    if updated:
        msg += "\nYOUR\nBEST"
draw_util.text_render(msg, pixel_font, center=True, target=cube.full_surface, pos=(32,0))
cube.flip()
time.sleep(2)
while cube.handle_inputs(): pass
cube.game_quit = False
game = None
cube.full_surface.fill(0)
cube.flip()
```

FIGURE 6.50: Sending score to the server and displaying results on the cube

14. Display the score, global high score and user's personal best score on the LED cube.
15. The game is reset, and the LED cube is cleared for the next game.
16. Wait for the client app to launch another or the same game again.
17. When the client App disconnects from cube, the server disconnects the client and starts listening for new connections again.

Overall, this script demonstrates how to use a Raspberry Pi and LED cube display to create a server that incorporates a fun interactive gaming experience that can be controlled using a mobile app via Bluetooth.

Thus, A briefer overview of the steps:

1. cube starts
2. app connects
3. cube stops accepting connections
4. app submits username and company (will be cached in the app)
5. cube sends available games
6. app requests a game
7. cube requests highscore for game from server
8. cube displays current highscore (optional)
9. cube starts game
10. app sends inputs
11. cube detects game over
12. cube sends score to central server
13. server automatically update the highscore on the database
14. server notifies the webpage about the new score
15. cube shows score and highscore
16. cube sends game over message to app
17. user exits the game and goes back to game selection menu
18. app user disconnects after being done playing
19. cube starts accepting connections again

6.4.2 Debug Server

When running the software for the cube there is also an option to enable a debug server. This allows you to connect via a browser and see how the games would look on the cube and also test inputs without the need for the cube or bluetooth meaning it can also be run on a windows computer. When you first visit the page a websocket is setup to send inputs from the page and the frames back to the page.

1. At the top of the page a flat view of the different displays on the cube can be seen.
2. Underneath are bits and brightness options these are for seeing how the image might look with different settings for the led displays. This is important as the games are rendered in true color (24bit) but because the raspberrypi inside the cubes can't keep a sufficient framerate to prevent flicker on the LED displays with all the colors we reduce the amount of colors thus we have to ensure the games look good with this reduced color pallet. Reducing the brightness will also further reduce the color pallet.
3. The final option "Lock X" is to lock the the joystick in the horizontal direction this is used for the Arkanoid game as there you can only move up and down not left and right.
4. In the center of the page there is a 3d representation of the cube which you can rotate around by dragging left or right. The bottom display is always visible as the cubes don't have a top display.
5. At the bottom of the page there are controls for the games in the form of a joystick and an action button.

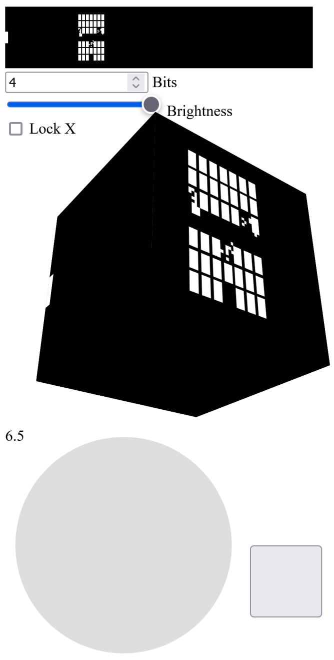


FIGURE 6.51: Debug Client

6.4.3 Games

Each game consists of a function with a regular pygame game loop but pygame's input and display functionalities are delegated to the CubeEngine class which is passed along to each game. At the end of the function, the final score is returned which can then be submitted to the leaderboard and shown to the player. Each game loop first processes input then updates the game, then draws the game and displays it on the cube and finally, it waits for the next update interval. The CubeEngine class will automatically pick between which sources of inputs and which outputs should be invoked depending on the platform the code is run on. For example on a Windows computer it will show a window and take input from the keyboard while on the cube it will output on the LED displays and get input from the mobile app. Input is handled by a special object which contains the current state of the buttons and joystick as well as whether it was pressed down between now and the last time the input was polled. This way it doesn't matter where the input comes from as they will always be mapped to this state object, this pattern is also useful because it prevents each game from having to keep track of this state individually.

```
class CubeEngine:
    full_surface: pygame.Surface
    input: Input

    def __init__(
        self, show_on_cube=False,
            run_debug_server=False,
            show_window=False,
            window_scale=5
    ): ...

    """Poll for input and returns whether the game should continue"""
    def handle_inputs(self) -> bool: ...

    """Submit self.full_surface to the displays on the cube and debug server"""
    def flip(self): ...

    """gracefully stop the debug server and driving of the LED displays"""
    def stop(self): ...
```

FIGURE 6.52: engine.py

```
def game(cube: engine.CubeEngine):
    fps = 60.
    clock = pygame.time.Clock()
    score = 0
    running = True
    display = cube.full_surface
    while running and cube.handle_inputs():
        # Input
        if cube.input.pause:
            clock.tick(fps)
            continue
        if cube.input.action.went_down: print("action button was pressed")
        if cube.input.up.is_down: print("d-pad-up is down")
        if cube.input.down.is_down: print("d-pad-down is down")
        if cube.input.left.is_down: print("d-pad-left is down")
        if cube.input.right.is_down: print("d-pad-right is down")
        cube.input.joyx # a value between -1 and 1
        cube.input.joyy # a value between -1 and 1

        # Update
        # movement and physics go here

        # Draw
        cube.flip()
        clock.tick(fps)

    return score
```

FIGURE 6.53: games/<game>/__init__.py

Arkanoid

Arkanoid is played on 2 of the displays on the cube. The paddle is directly linked to the joystick's y axis making it very snappy to control. In this game you have to break as many blocks as possible, there are different blocks which take a different amount of hits indicated with different colors. Additionaly some of the bricks will spawn new balls making it easier The game is played in the course of multiple levels each having a different arrangement of bricks. The game is over when all balls get behind the paddle 3 times in a row or all levels are cleared.

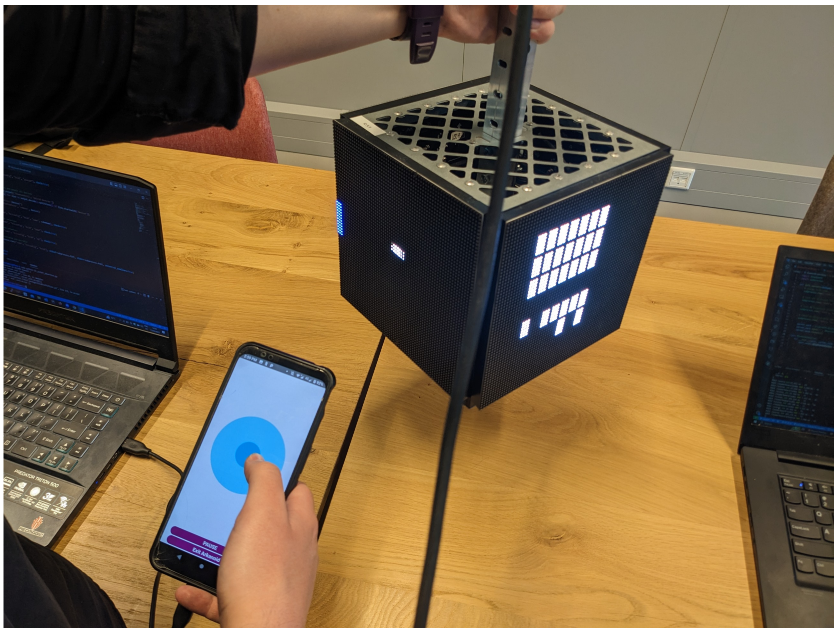


FIGURE 6.54: Arkanoid Playing on the cube

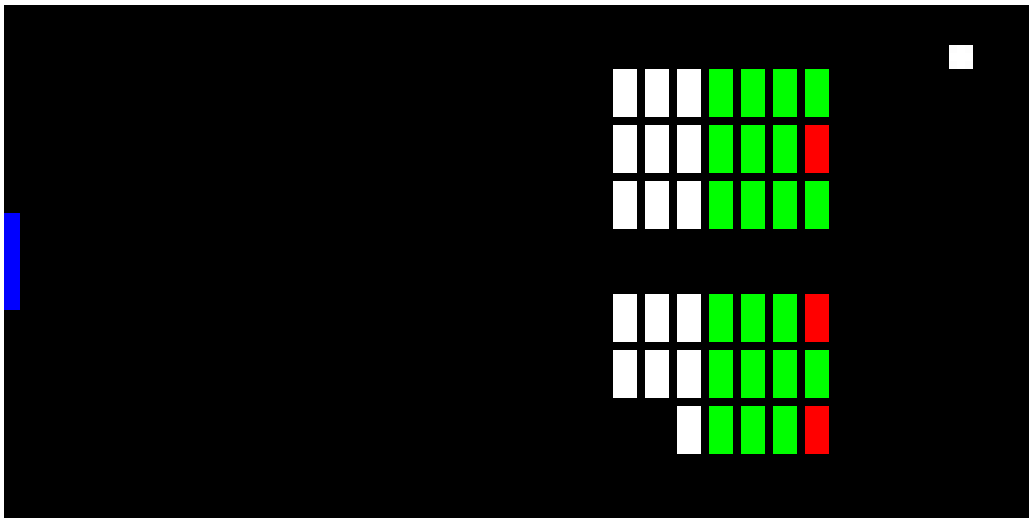


FIGURE 6.55: Arkanoid Playing on simulated cube

Snake

A simple snake game played around the parameter of the cube. To get the highest score make your snake the longest the quickest. The game is over when the snake can't grow anymore.

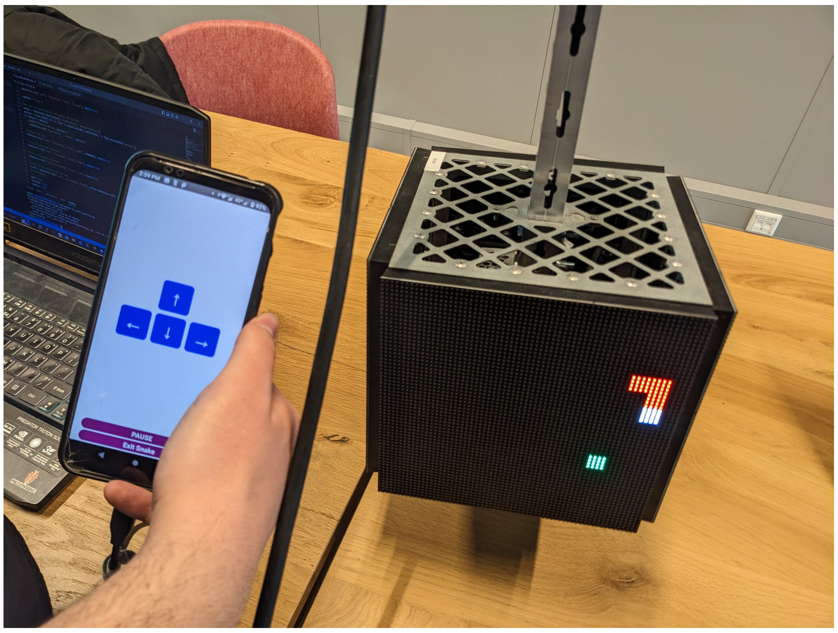


FIGURE 6.56: Snake Playing on the cube

Space Shooter

A game where you have to prevent incoming spaceships around the cube from reaching the bottom of the cube The game is played in multiple levels of increasing difficulty. The current lives and levels are shown at the bottom of the cube. The game is over when either your ship is destroyed or more than 5 ships get behind your defences.

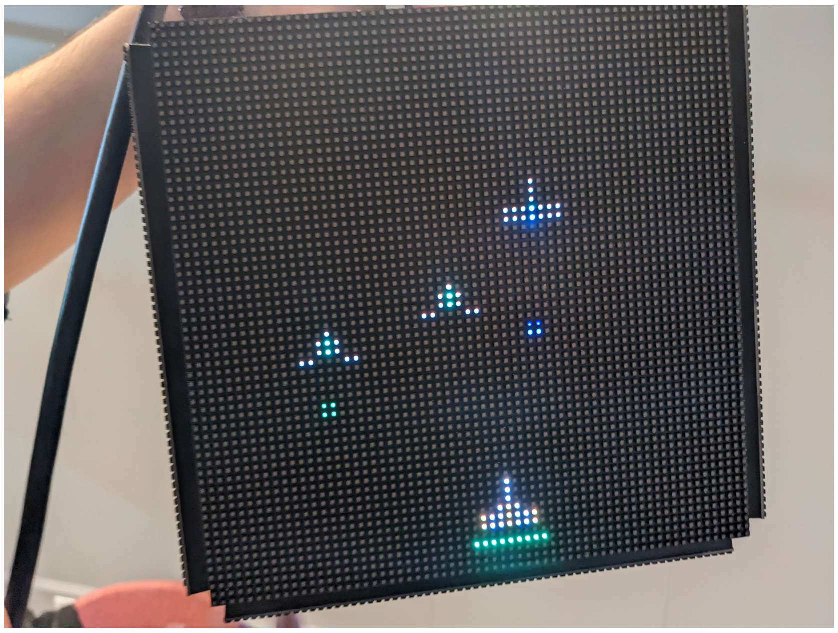


FIGURE 6.57: Space shooter Playing on the cube

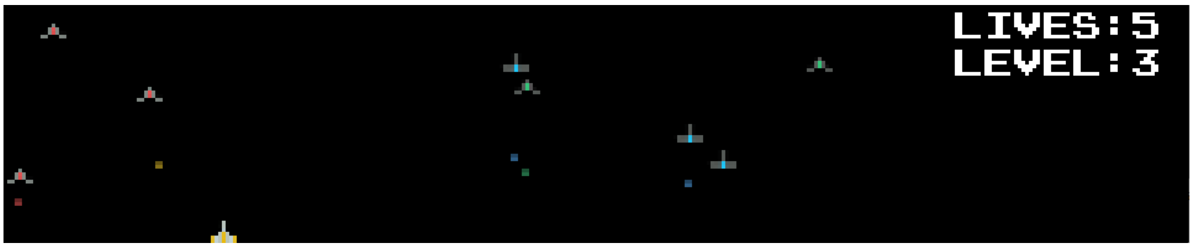


FIGURE 6.58: Space shooter playing on simulated cube

Tetris

Tetris played on one side of the cube with an additional side on the right used to show the score and the next piece. Points are awarded whenever a full line of blocks is complete and cleared. Extra points are awarded when multiple lines are cleared at once and when a piece is forced down.

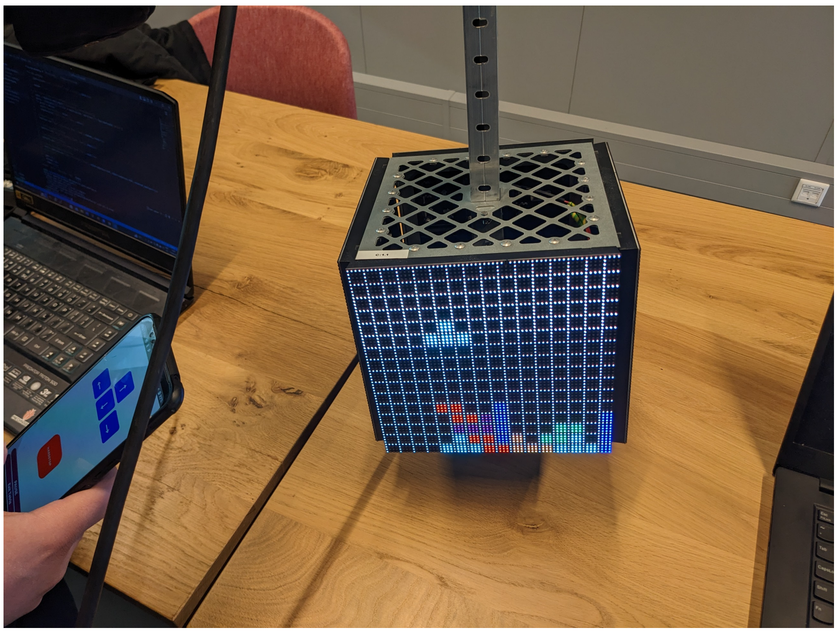


FIGURE 6.59: Tetris Playing on the cube

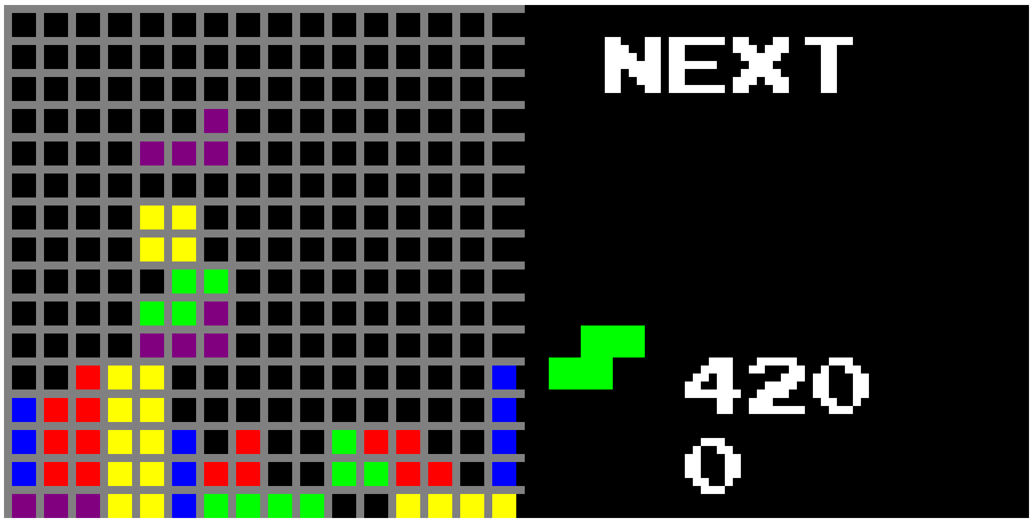


FIGURE 6.60: Tetris Playing on simulated cube

Chapter 7

Testing

7.1 Test Plan

Testing is being executed gradually throughout the development process.

Weeks 3, 4

We tested that the connection with the cube is functioning adequately via Bluetooth. The testing is performed on an Android device. This includes testing the moment of the connection itself and checking whether a device can detect all cubes located up to 4 metres from the user's device. If a cube is busy because someone is already playing a game on it, this cube shall not broadcast any signal and it shall not be recognised by other external devices during this time (this does not include other cubes).

Week 5

During this week, the database was designed and tested. Since there was not all games at this moment, the requests were manually created with Postman. First, we tested the interaction between the database and the game. This includes testing whether the score sent from the backend to the database is stored correctly. After that, the performance was checked. We aimed to receive a response within 4 seconds from the backend. We also ensured that there are no troubles with decimal values and verified that the web GUI displays the high scores accurately.

Weeks 6, 7, 8

During these weeks, we focused on implementing the minigames. Thus, most of the game testing was performed during this period. First, we ensured that the games were integrated into the cubes correctly. We checked whether a person, after selecting a cube, could see all available games on their device.

After that, we performed testing of the basic functionalities of each minigame. This includes but was not limited to instructions (game movement), score, and game logic. We tested the ability of the cube to handle losses in connection.

When the games were implemented, we tested the correctness of the interactions between the controller, the web application, and the database. We also verified that the Bluetooth broadcast resumed when the player exits the minigame on it. In addition to this, we verified that database requests were handled appropriately and that the high score from a certain game was stored in the correct game in the database.

7.2 Backend

Testing for the backend was mainly in the form of manual testing. We did this by issuing POSTMAN requests and verifying that the database was in fact updated in the way that we expected. In addition to the regular backend code, we developed certain Javascript files that would allow us to clear the database and view the contents of the database. This was used extensively during testing purposes.

Some of the tests that we did include:

- Creating a new user
- Posting a new high score for a user for a specific game
- Updating the company for a given user
- Retrieving the leaderboard data
- Retrieving the high score for a game

We also ran several tests with malformed URLs and incorrect JSON bodies to ensure the server would not crash and appropriately handle these specific edge cases.

7.3 Leaderboard Page

Regarding the leaderboard page, we have mainly used a combination of unit and manual testing to verify the functionality and completeness of the web app. Specifically, we made use of Jest and Selenium for automated testing. We also made use of manual testing to quickly verify the functionalities and also receive feedback regarding certain design choices.

Jest is a Javascript testing framework mainly used for writing unit tests and for mocking and asserting functions and component rendering. Our unit tests in Jest ensured that specific components were rendered properly. On the other hand, we made use of Selenium for integration testing. Selenium is an open-source testing framework used for web application testing. It automates browser interactions and can simulate user actions like clicking, typing etc. This way, we were able to test the interaction between components and verified that the app behaved as expected under certain types of input from the user.

In addition to manual and integration testing, special attention was given to manual testing. This type of testing is when a tester manually interacts with the product to verify its functionality. In this case, we went through and verified all the functionalities and features of the application, ensuring that the application worked correctly and without any bugs. We relied on manual testing to check the correctness of the communication between the web app and the backend. We also tested the application's responsiveness to different screen sizes and devices to ensure that the user experience was consistent across all platforms.

7.4 Cube

The best way to test games is to play them. However, it is not effective to play the whole game again and again after each change. That is why the "debug mode" is introduced. The "debug mode" can be activated within the game. This tool allows developers to access certain parts of the game, such as specific levels or mechanics, and quickly test changes without having to play through the entire game. Debug mode also provides additional information, such as the game state, variables, or logs, that can be useful for debugging and troubleshooting issues. The information is collected in the debug server which is installed and integrated into the cube engine, and all the necessary information is displayed on the Flask web page which is open at the machine debugging a particular cube. Also, the design of each game allows very quick changes of variables which are responsible for various parts of the game, and the developer can immediately test certain levels with a minimal change in the source code.

7.5 Integration Testing

In the integration testing, we mainly tested component interactions and actions with the server. Since, currently, there is no software that can emulate Bluetooth connectivity we tested integration between different components manually.

7.5.1 Mobile App -> Server

The mobile App tests mainly involved seeing how the communication between server and the app is handled. For example, we tested how the event listeners in the Mobile Application App component work. We did a couple tests like shutting down the Cube during the use of game controller to see if the App receives an Alert message saying the "Cube is disconnected" or not. We also did similar test, while the user is in CubeApp's cube main page to see whether it will instantly show "OFF" in the menu when the cube suddenly shuts down. In the end, the tests were successful and the result was, event listeners instantly detect and inform the user about the status of the Cube.

In the initial stages of Mobile App and Server development, we were a bit worried about the communication since Mobile App uses Typescript and Server uses Python. So, at first, we had no idea whether we would have smooth communication between a Typescript client and Python server. Through a trial and error, we were in the end able to find a compatible library, but it wasn't our initial choice. The initial Bluetooth libraries we were working with were "react-native-ble-plx" in the App and pyBleno in the Server side. Bluetooth Low Energy (BLE) libraries were our first choice for communicating. However, during the testing of the communication, we realized that these 2 libraries don't work fully well with each other as a Client and Server. The problem was how many messages it can transfer per minute. We tested that BLE could only transfer 2 messages per second smoothly. This wasn't good since the Joystick component was sending 60 messages maybe more per second. Thus, after this, we changed our plans to a different library.

Bluetooth Classic was our solution to the previous problem. We tested that in order to have smooth and fast gameplay we needed a bluetooth socket. PyBluez from the server side and react-native-bluetooth classic from the mobile app side worked perfectly for us.

The initial communication we tested was seeing how we can send and how the server can send the messages back to us. We first tested this by sending a request for retrieving the list of games from the server. Initially, we had problems, since we were not receiving the message back from the server. We later realized that the string text messages sent from the server should be delimited strings, otherwise, the react native client wouldn't be able to read it. Since in strings, the delimiter is a new line character (\n), we added it to every message we sent both from the client and back from the server. After this, with the help of the if-else statements in the Server, we were able to create a messaging protocol for the initialization of variables such as launching the game, username, company name and game controller inputs. We tested this by logging the responses on both sides to see if it works and it was successful.

One of the most important tests we did were about seeing how fast it can send and receive Joystick messages. We tested this by instantly moving the joystick centre around to generate messages. The results were good, we found that it could handle more than 70 messages per second with our custom-made joystick. However, there was still another issue we detected with the help of the test. We found out that when the server stops receiving messages for a while and starts receiving them again, it delays for 2-3 seconds and receives the messages as a bundle instead of one line at a time. The solution for this problem was a heartbeat stay-alive message to keep the socket connection alive. After this, we tested all the game controllers again, and this time we had no issues.

In the end, smooth and fast communication was our goal for the App and the server, and with the help of various tests, we did we were able to achieve it and were able to find our issues and find solutions for them in time.

7.5.2 Server -> Backend webpage server

Another component with the bluetooth server interacts is the backend webpage server. There are 3 main requests it gives. The first one is sending a post request to the backend to create a new user based on the logged-in client's name and company name. We created several tests around this request and also tested sending this request to the backend with a Python code with the help of the requests library. Our main tests revolved around the fact that when the user does not exist in the database, it should return status code 201, which means it created a new user in the database. And when the username we are sending does exist in the database, it should return status code 200, which returns a message that the user exists in the database. We tested this with post requests and example usernames we created in Python. There is also another feature of this post request, when the username does exist in the database and when the company name is different than in the database for that user, we made sure that in the post request, we update the company name with the company name we sent the post request. To test this, we did some Python post and get requests code to make sure that the company name is properly updatable from the "server.py" file.

The second request in the server is a very simple one. The second request is just getting a highscore value for a specific game. We need it to show the user at the end of the game as a global score that the user should beat. We mainly did small tests for this, we made sure to play 4 different games and go to game over all games to see if the request is able to retrieve the global highscore for each game correctly or not.

The third request was about sending the score of the user for a game to the backend server. There is a specific protocol we have for this in the server. Basically, the backend server remembers users' own personal highscore as well as global high scores. Thus, if a user achieves a higher score on some game that he played before, in the game over screen there will be "YOUR BEST" message indicating you beat your own previous score, and this is your new personal highscore. Based on this message we created test cases, and played the games to see if the server receives your scores and is able to process them correctly. In the tests, we played each game 2 times, one time deliberately scoring very low, and the next time scoring higher to see if we will see the "YOUR BEST" message or not.

To sum up, We conducted various gameplays and test cases to ensure the proper functionality of the Bluetooth server requests.

7.5.3 Server -> Games

Besides the requests, Server also interacts with the game components. The tests here were mainly about interactions with the game code and the Bluetooth server. We tested whether it is possible to do game loops from the server loop and at the same time also listen to inputs from the server loop. This would mean, First, we tested whether, with a press of a button from the App, we can start games through bluetooth server. After we did these launch tests for each game, we moved on to test the inputs. Testing inputs were mainly through gameplay with the mobile app. We just played a few rounds of each game to see if it is correctly able to interpret the proper game input commands. We also tested exit functionality. If you remember from the mobile app we had an exit button in the controllers. This test was to make sure that you can exit and choose different games to play and the server doesn't crash while we do this. We tested this by playing and exiting each game and launching other games to make sure that the exit feature is working. Lastly, we tested the pause implementation feature. Being able to pause games was a very important feature for the Company since they would use that feature of the game when they would want to use the cube for other purposes instead of games. For this, we also did similar tests with gameplay as we did for the exit feature. We just launched each game and did pause and unpaused to see if it is working for each game on the cube or not.

In conclusion, with the help of the gameplay through the mobile app, we were able to make sure that server correctly interprets and passes commands from the mobile App to the Games.

Chapter 8

Future Work

The main issue which is addressed in future work is the availability of the Bluetooth on the cubes. The reason is that the cubes shall be idle to answer the Bluetooth request. If a cube is displaying time or something else, it is not discoverable. The main focus shall be on the integration with the existing Mindhash software system (using signals) which will allow the Bluetooth server to stay up regardless of whether the cube is busy or not.

One exciting idea for future work on the Cubewear project is to make the games adjust to environmental hazards such as stairs, walls, and furniture. The design of the building in High Tech Park where Mindhash is located allows for the installation of additional sensors which may detect physical obstacles and report the locations to cubes. By using data gathered from sensors, the game could adjust the gameplay to account for these hazards and disable certain displays based on a certain cube location. This might be done by putting in the sensors the distance to the floor of the building which is measured in advance and the angle to which a sensor is installed. If a sensor measures the distance shorter than the distance given, then it concludes there exists an obstacle and sends the signal to the cube system. Each cube might have a mapping of which sensor is responsible for which display and, in case of a signal received, the ability to disable this display and change the game field.

Another idea for future work on Mindhash's Cubewear project is to expand the game library by adding more games that utilize the unique features of the Cubewear cubes. The team could explore different genres of games, such as puzzles, adventure, or sports games, that could leverage the capabilities of the Cubewear cubes, such as the multiple displays and the motion sensors. For example, a puzzle game could require players to rotate the cube to match patterns on different sides which requires them to clearly remember what is located on other displays. The game library may be expanded with the following games:

- Endless Runner Game (Similar to Dino Game on Chrome)
- Maze Arcade game
- Wordle

Another idea for future work on Mindhash's Cubewear project is to create a better account system. Currently, if a user wants to change their account name, they must create a new account and lose all their previous scores and achievements. With a better account system, users could change their usernames without losing their progress. Additionally, the team could explore other account management features, such as password recovery or two-factor authentication, that could further enhance the security and convenience of the system.

Currently, users do not have the ability to delete their accounts, which could be problematic from a privacy perspective. By enabling users to delete their accounts, the Cubewear project could demonstrate its commitment to data privacy (GDPR). To implement this feature, the team could explore different approaches, such as allowing users to delete their accounts through the mobile app or web interface or implementing an automated process for deleting inactive accounts. The team could also develop a data retention policy to ensure that user data is only kept for as long as necessary and is securely deleted when no longer needed. Additionally, the team could provide users with transparency and control over their data by allowing them to download or export their data in a standardized format, such as JSON or CSV.

Currently, the games can only be played on the cubes. Another idea for future work on Mindhash's Cubewear project is to enable users to play games on the web client: this feature could be particularly useful for users who have movement disabilities or difficulties and they are unable to normally go around a cube. Also, this feature is nice for people who do not have access to the cubewear cubes or who prefer to play games on their desktop or laptop computers. The team could also work on optimizing the games for different screen sizes.

Another idea for future work is to develop multiplayer games that could be played on the cubes. Currently, the games are intentionally designed for single players. Multiplayer games may promote teamwork and friendly competition, and also encourage users to invite friends or colleagues to play together, increasing user engagement and retention. To implement this feature, the team could explore different approaches, such as implementing a local network protocol that would allow multiple cubes to communicate with each other. The team could also develop new game mechanics or objectives that would be specifically designed for multiplayer gameplay, such as cooperative puzzles or team-based challenges.

Chapter 9

Evaluation

9.1 Task Division and Teamwork

Our project was one with many different components. As such distributing the different components amongst ourselves was needed. This is the division that we came up with:

- Cube LED Matrix sofware - Bram
- Cube Bluetooth server - Toghrul, Bram
- Mobile Application - Toghrul
- Backend - Faisal
- Leaderboard Page - Mustafa
- Game Development:
  - Snake: Vladimir
  - Space Shooter: Vladimir
  - Tetris: Vladimir
  - Arkanoid: Bram

This was the distribution that we created at the start of the module and stuck with till around Week 8. From week 8 onwards, the group was mainly focused on the report so everyone worked on it.

Overall, the team maintained good communication with each other throughout the module duration and everyone carried out their individual tasks as expected. The division that we created did manage to effectively balance the workload that we had.

9.2 Final Product

Throughout the development of the product, we worked diligently to ensure that all the requirements agreed upon by the client were met, and we conducted extensive testing to validate the functionality and various metrics of the project.

Based on the list of functional requirements mentioned in this report, we were able to achieve all "must have" and "should have" requirements. These requirements consisted of core functionalities that were necessary for the product to function as intended. Additionally, we had some "could have" requirements which were added to the project as stretch goals. We decided to only fulfil these requirements if we had extra time. Unfortunately, we did not meet these requirements due to time constraints.

Regarding non-functional requirements, we also made significant efforts to achieve them. These requirements mainly included factors like performance, security, accessibility etc. Through extensive testing of each component, we are confident that we have achieved all the non-functional requirements.

Chapter 10

Conclusion

Overall, the design project module was an excellent opportunity for us students to apply the knowledge and skills we have learnt till now in a practical project. We are grateful to Mindhash for providing us with such an interesting project as well as their constant support throughout the module duration.

The weekly meetings with Mindhash as well as our supervisor Nacir were invaluable as they provided us with feedback and ensured we were always on track to meet our goals. The Peer review meetings gave us the opportunity to share our progress with other students in the same position as us and hear some different perspectives. All the team members contributed a significant amount towards this goal and did a fair amount of work. The development process went smoothly and we as a team mostly had steady progress throughout the entire project duration.

All in all, we as a team are proud of what we have made. We have managed to develop a system that consists of multiple different components including a mobile application, leaderboard page and games that are playable on the sides of the cube.

We believe the system as a whole is of high quality and hope that Mindhash will make good use of the system moving forward.

Appendix A

REST API Specification

In this chapter, we will go through the endpoints of our REST API system as well as how to invoke them.

Retrieve Leaderboard

This call is made by the front-end web page to populate the table with relevant data.

Request

GET /leaderboard

Response

200 OK
JSON Body
An array of object with following fields:

- username: username of the user
- company: company of the user
- game: the game for which the score is being displayed (snake, tetris, arkanoid or spaceShooter)
- score: score for the game

```
[
  {
    "username": "Mustashot",
    "company": "Mircosoft",
    "game": "snake",
    "score": 189
  },
  {
    "username": "Mustashot",
    "company": "Mircosoft",
    "game": "spaceShooter",
    "score": 189
  },
  {
    "username": "Bram",
    "company": "Google",
    "game": "snake",
    "score": 102
  }
]
```

Retrieve Global High score for a Game

This call is made by the Pi in the Cube and displayed to the user on the screen.

Request

GET /leaderboard/<game>
URL Parameters:

- game: The name of the game for which the high score is to be retrieved (snake, tetris, arkanoid or spaceShooter)

Response

200 OK
JSON Body Fields:

- score: the high score for the game

```
{
  "score": 219
}
```

Send High score

This call is made by the Pi in the Cube to send the highscore to the backend.

Request

POST /scores
JSON Body Fields:

- username: username of the user
- score: the score for the game that the user has just played. The name of this field should be either snake, tetris, arkanoid or spaceShooter depending on the game.

```
{
  "username": "user1",
  "snake": 219
}
```

Response

200 OK
JSON Body Fields:

- updated: true if the personal highscore was updated, false if the highscore was not updated
- highscore: the global highscore for the game before the submission of the score

```
 {
    "updated": true,
    "highscore": 219
  }
```

Create New User

This call is made by the Cube when a new user is created or when a user tries to authenticate into the system.

Request

POST /user
JSON Body Fields:

- username: username of the user
- company: the company of the user

```
{
  "username": "user1",
  "company": "company1"
}
```

Response

201 Created
This is the response if the username was not yet in the database.
JSON Body Fields:

- createdUser: details of the user that was created

```
    "createdUser": {
        "username": "Faidoo",
        "company": "Google"
    }
```

200 OK
This is the response if the user was already in the database.
JSON Body Fields:

- detailsChanged: true if the details of the user were changed, false if the details were not changed. The company can be changed if a post request is sent for a user with the same username but a different company name.

- newCompany: the updated company name of the user. Only present if details changed is true.

```
{
  "detailsChaanged": true,
  "newCompany": "company5"
}
```

Get Scores of a User

This call can be made by the Mobile App or Leaderboard Page to retrieve the individual scores of a user for each game.

Request

GET scores/<username> URL Parameters:

- username: username of the user

Response

200 OK
JSON Body Fields:

- username: username of the user
- company: company of the user
- snake: score for the snake game
- tetris: score for the tetris game
- arkanoid: score for the arkanoid game
- spaceShooter: score for the spaceShooter game

```
{
  "username": "user1",
  "company": "company1",
  "snake": 219,
  "tetris": 357,
  "arkanoid": 234,
  "spaceShooter": 123
}
```

Update Details of a User

This call can be made by the Mobile App to update the details of a user.

Request

PATCH/users/<username> URL Parameters:

- username: username of the user

JSON Body Fields:

- username: username of the user
- company: the company of the user

```
{
  "username": "user1",
  "company": "company5"
}
```

NOTE: Both fields are optional. If only one field is provided, only that field will be updated. If both fields are provided, the username will be updated first and then the company will be updated.

Response

200 OK
JSON Body Fields:

- updatedFields: the fields that were updated and their new values

```
{
  "detailsChaanged": true,
  "newCompany": "company5"
}
```