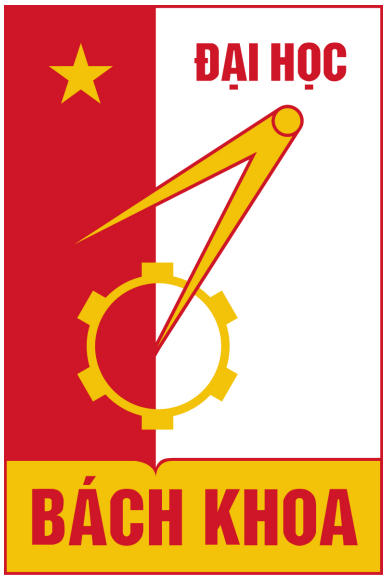
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

—————— o0o ——————



Voice Controlled Dinosaur Game

Embedded System Final Project IT4210E

Lecturer: Ngo Lam Trung
Team: SuperFastVoice
Students: Nguyen Huu Nam Hoa – 20226040
Tran Tung Duong – 20226033
Ngo Minh Duc – 20226028
Class: ICT-01 K67

Hanoi - 2025

Contents

Task assignment

Task Assignment for the project:

Member	**Student Code**	**Task Assignment**	**Percentage**
Nguyen Huu Nam Hoa	20226040	Set up microphone, design GUI, process sound input, implement obstacles in a game	40%
Ngo Minh Duc	20226028	Implement scoreboard, collision detection in a game, result display	30%
Tran Tung Duong	20226033	Implement level selection, character movement, live + score update in a game	30%

Table 1: Task assignment

Chapter 1

Introduction

1.1 Objectives

This project aims to design and implement a voice-controlled embedded game where the player controls a dinosaur character that jumps to avoid obstacles. Instead of using traditional button input, the game leverages voice signals to control vertical movement. The system analyzes the volume of the player's voice to determine the jump strength: a louder voice triggers a high jump, while a softer voice triggers a low jump.

This project showcases the integration of real-time voice processing and embedded systems, creating an interactive experience through natural user input.

1.2 Requirements

1.2.1 Functional requirements

Flow Chart

Figure 1.1 shows the main use cases of the system, including user interactions with key features such as starting a game and viewing the scoreboard.

Game rule

The game consists of three difficulty levels:

- Level 1: Only one type of cactus appears as an obstacle.
- Level 2: Two types of cacti are introduced — the second type is wider than the first, requiring longer jump timing.
- Level 3: In addition to both cactus types, a flying bird obstacle is added, which requires precise timing and jump height to avoid.

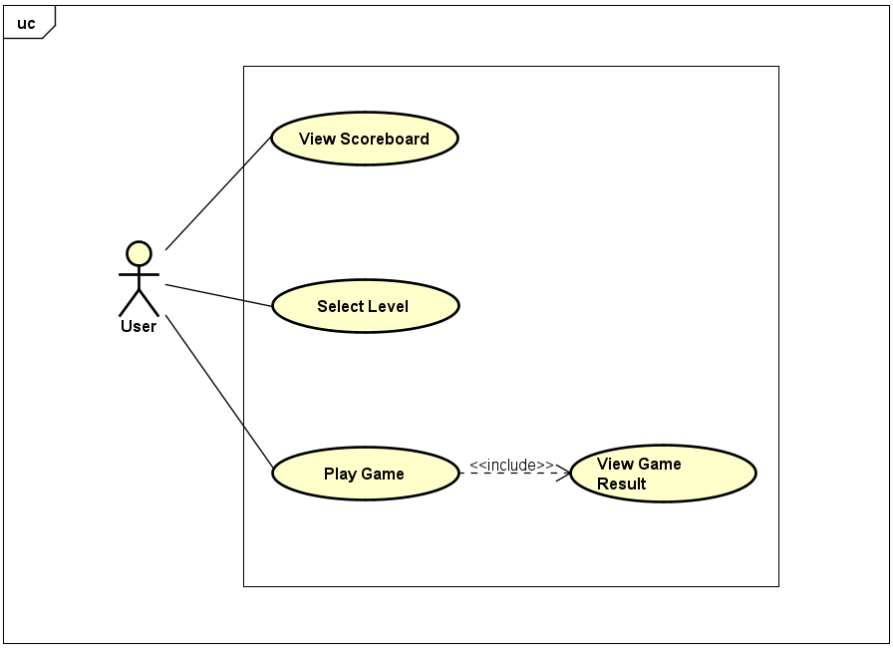


Figure 1.1: Use case diagram

The dinosaur jumps based on the player's voice:

- **Low voice volume** triggers a low jump.
- **High voice volume** triggers a high jump.

Each play session grants the player three lives. Every collision with an obstacle causes the player to lose one life. When all lives are lost, the game ends and displays the final score.

1.2.2 Non-functional requirements

- The system should respond to voice input with minimal latency.
- Game logic and rendering should be smooth and consistent on the embedded platform.
- The physical hardware (if present) should be durable and compact.
- The interface must be intuitive and require minimal learning effort.

Chapter 2

Design

2.1 Functional Division

The system is structured into two main domains: hardware and software, with each playing a crucial role in delivering an interactive and responsive game experience. The game is a custom variant of the popular "Dino Run", where a dinosaur must jump over randomly appearing obstacles. However, instead of using physical buttons, this project introduces an innovative sound-based control mechanism using a microphone.

2.1.1 Hardware Responsibilities

- **Main Processing Unit:** The system is built on a single STM32F429ZIT6 development board. This board is responsible for the entire operation of the game, including signal acquisition, game logic processing, and UI rendering.
- **Sound Input Module:** The MAX4466 microphone module is used to capture audio input from the player. The player controls the dinosaur's jump using sound:
  - A low-level sound triggers a short jump.
  - A high-level sound triggers a higher jump, approximately twice the height of a low-level jump.
- **Audio Signal Acquisition (ADC):** The analog signal from the microphone is sampled by the ADC peripheral of the STM32F429ZIT6. The sound intensity is then categorized into discrete levels (no input, low, high) by the software.
- **Display Interface:** The STM32F429ZIT6 development kit features an integrated LCD, which is directly used for rendering the game's UI. This screen displays the dinosaur, obstacles, score, and all animations in real-time.

2.1.2 Software Responsibilities

- **Game Logic Control:** The software defines and manages the full game flow: obstacle spawning (cacti and birds), movement simulation, collision detection, score calculation, and game states (start, in-game, game over).
- **UI Rendering with TouchGFX:** The user interface and all animations are implemented using the TouchGFX framework, a graphics library optimized for STM32 microcontrollers. TouchGFX manages rendering of moving sprites (dinosaur, obstacles), UI updates (score display), and transitions between game states.
- **Sound Processing:** The ADC continuously samples sound levels from the microphone. The software maps these values to jump actions using empirically determined thresholds to distinguish between low and high sound intensities. This module also includes filtering logic to ignore ambient noise and reduce false triggers.
- **Difficulty Management:** Based on the selected difficulty level, the game dynamically adjusts:
  - Level 1: Only one type of cactus appears as an obstacle.
  - Level 2: Two types of cacti are introduced — the second type is wider than the first, requiring longer jump timing.
  - Level 3: In addition to both cactus types, a flying bird obstacle is added, which requires precise timing and jump height to avoid.
- **Score Handling:** The software tracks the player's performance over time and updates the score in real-time and displayed on the screen.

2.2 Hardware Design

The hardware system of the project is designed to be both minimalistic and efficient, utilizing only essential components but maximizing the capabilities of the STM32F429ZIT6 microcontroller. The primary goal of the hardware is to accurately capture the player's vocal input, interpret it through analog signal processing, and render real-time gameplay on a built-in TFT LCD screen. The system is composed of the following major elements:

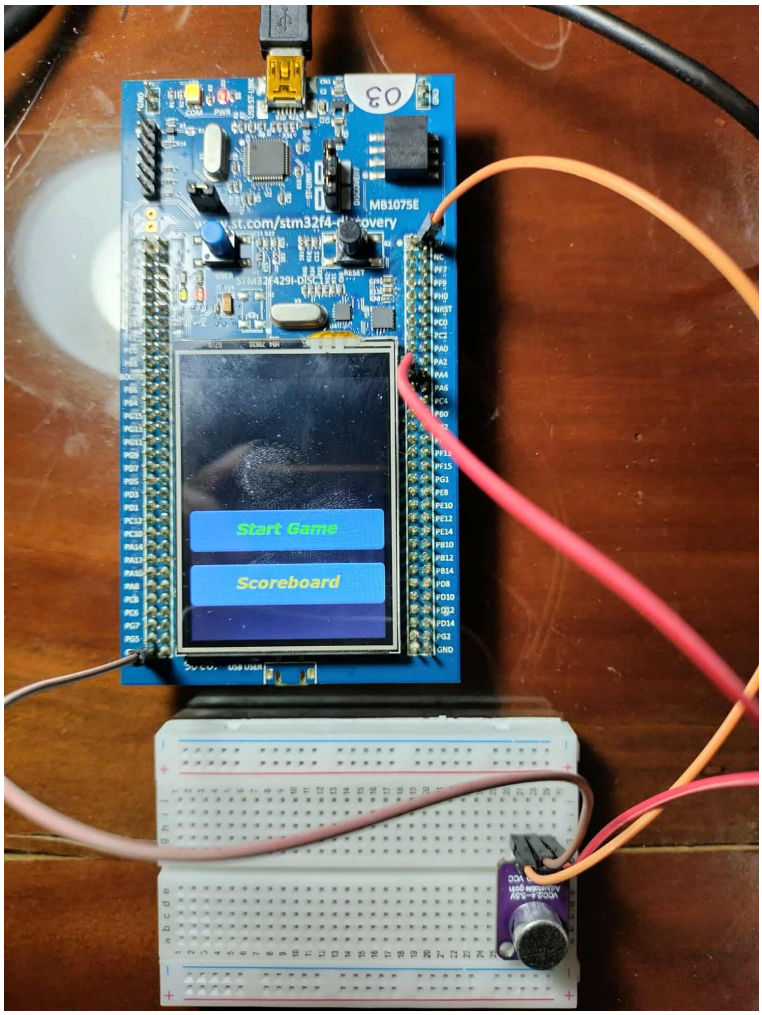


Figure 2.1: Circuit diagram

2.2.1 STM32F429ZIT6 Development Kit

This development board is the heart of the entire project. It integrates:

- A high-performance ARM Cortex-M4 core running at 180 MHz.
- Multiple ADC channels and DMA controllers.
- On-chip LCD controller and hardware graphics acceleration, fully compatible with TouchGFX.

The STM32F429ZIT6 handles:

- Sampling and processing of microphone data via ADC.

- Running game logic (obstacle generation, collision detection, scoring).
- Rendering all UI elements, animations, and player feedback via the onboard LCD.

By consolidating all processing into a single MCU, the system avoids the complexity of inter-board communication and minimizes latency between input and game response.

2.2.2 MAX4466 Microphone Module

The MAX4466 is a high-gain, low-noise analog microphone module with a built-in op-amp and adjustable gain potentiometer. It converts the player's voice into an analog voltage signal, which fluctuates in real time with sound pressure levels.

MAX4466	**STM32**
VCC	3.3V
GND	GND
OUT	PA5 (ADC1 IN5)

Table 2.1: MAX4466 Microphone Module Pin Connections

This microphone is chosen because:

- It has high sensitivity to human voice range (300 Hz – 3 kHz).
- It supports a sampling time of 84 ms and operates at a frequency of 12.5 MHz, which ensures adequate temporal resolution for voice signal acquisition.
- Its output is analog and can be directly connected to STM32's ADC input without additional circuitry.
- The onboard gain potentiometer allows fine-tuning to reject background noise and optimize for vocal detection.

2.2.3 Analog-to-Digital Converter (ADC) with DMA

The analog voltage output from the MAX4466 is read via STM32's internal ADC peripheral. The ADC digitizes the continuous analog signal into discrete numerical values (typically 12-bit resolution).

To avoid processor blocking and maintain real-time performance, the ADC operates in continuous conversion mode, and the results are transferred directly to memory using DMA (Direct Memory Access).

The reason why DMA is essential in this design:

- DMA allows the CPU to continue running game logic and graphics while audio data is collected in parallel.

- Achieves consistent sampling frequency, which is critical to detect brief voice signals.
- Frees the CPU from polling or handling ADC interrupts, ensuring smoother gameplay and UI responsiveness.
- DMA (in circular mode) continuously fills a memory buffer, which can be post-processed to determine the jump type (low/high) using peak detection or averaging methods.

This method ensures low-latency and accurate audio detection, which is crucial since jump height is directly tied to the intensity of the player's voice input.

2.3 Software Design

2.3.1 Flow chart

Figure 2.2 illustrates the main user interaction flow in the game. It begins at the Start screen, where the user is presented with two options: Start Game or Scoreboard. If the user selects the Scoreboard button, the system navigates to the Scoreboard Screen, where the highest scores of all levels are displayed. If the user chooses the Start Game button, the application proceeds to the Select Level screen, where the player can pick a difficulty level (1, 2, or 3). After selecting a level, the game enters the Play Game phase, where the voice input controls the dinosaur's jump to avoid obstacles. Upon game over, the system displays the Result screen, showing the score and selected level, with an option to return to the home screen.

2.3.2 Module description

The project includes the following modules:

- **Voice Input Processing Module**:
  - Captures audio signals from the microphone.
  - Analyzes voice volume and classifies it into two levels: *Low Jump* and *High Jump*.
  - Sends jump command to the game logic module based on classification.
- **Game Logic Module**:
  - Controls game flow based on current screen (start, level selection, game, result).
  - Manages obstacle generation and movement.
  - Detects collisions between dinosaur and obstacles.

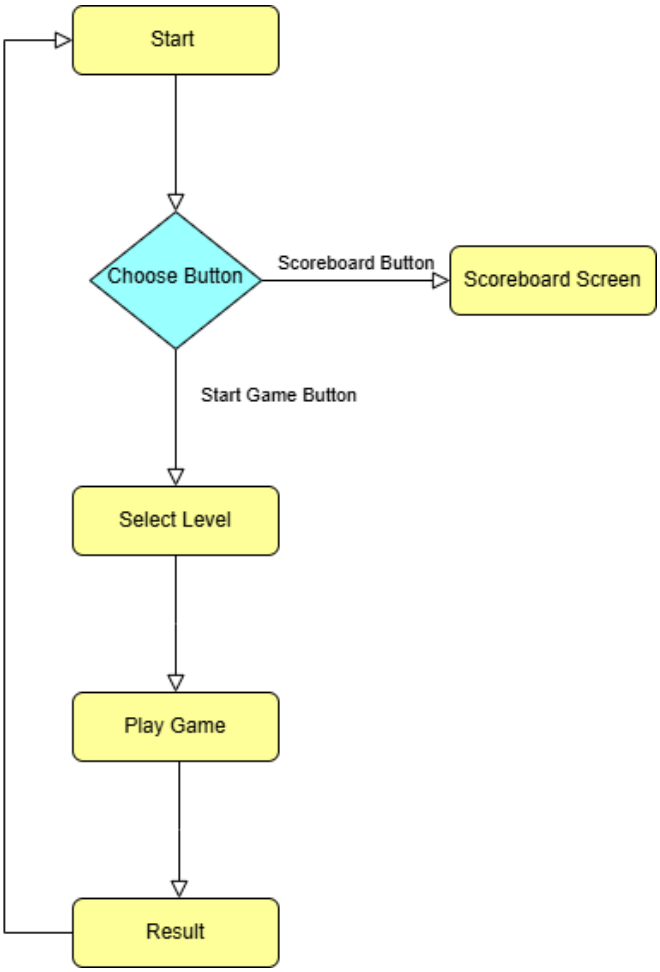


Figure 2.2: Flow chart

  - Updates score and life status accordingly.
  - Stores and updates the highest score per level.

- **Display Module**:
  - Renders game screens (Start, Scoreboard, Level Selection, Game, Result).
  - Draws character, obstacles, score, and life indicators in real time.
  - Provides screen transitions based on user input and game state.

Chapter 3

Implementation and Results

3.1 Implementation

The full source code is available on GitHub at:
`https://github.com/Codeximus2k4/VoiceControlledDinosaurGameSTM32.git`

3.1.1 Display Module

The Display Module is responsible for rendering the graphical user interface (GUI) and visual feedback during gameplay. For this project, the GUI is developed using **TouchGFX Designer**, a powerful graphical tool for designing embedded interfaces on STM32-based systems.

GUI Screens and Components

The following screens are implemented in the GUI:

- **Start Screen**: Contains two buttons — `Start Game` and `Scoreboard`. User interactions on this screen determine the game flow.
- **Level Selection Screen**: Provides buttons to select from three levels (Level 1, 2, 3), each with increasing difficulty.
- **Game Screen**: Displays the dinosaur character, obstacles (cactus and bird), current score, and life status. It also updates in real time based on jump commands from the game logic.
- **Result Screen**: Shows the player's score and selected level after a game ends, with a `Home` button to return to the start screen.
- **Scoreboard Screen**: Displays the highest recorded scores for each level.

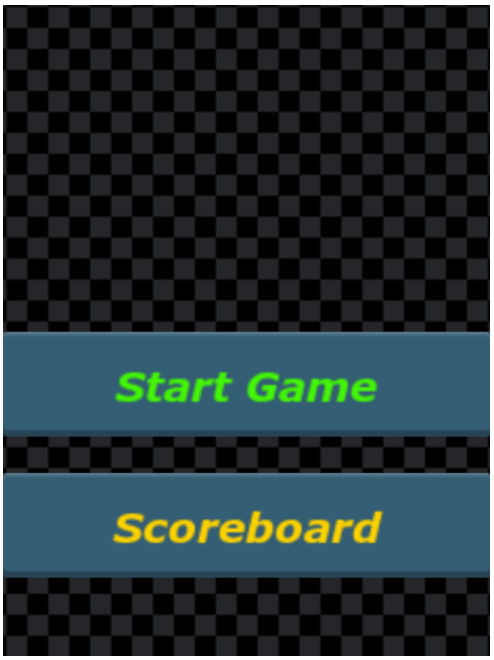


Figure 3.1: Start screen

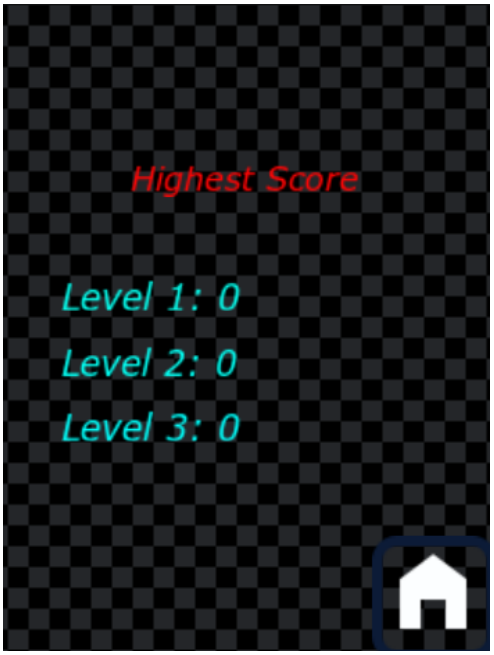


Figure 3.2: Scoreboard screen

Figure 3.3: Level selection screen

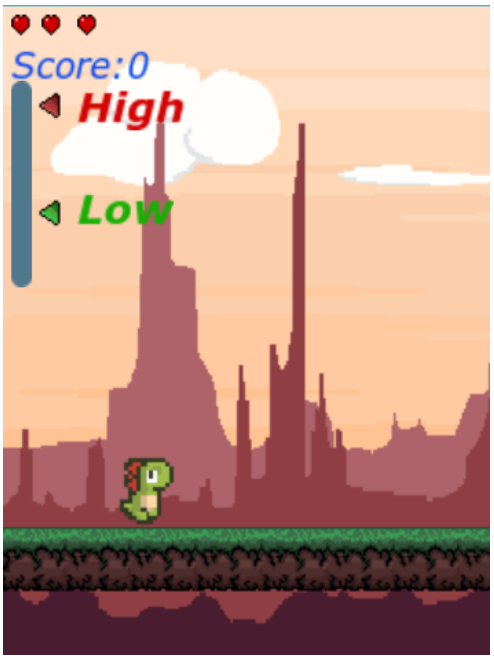


Figure 3.4: Game screen

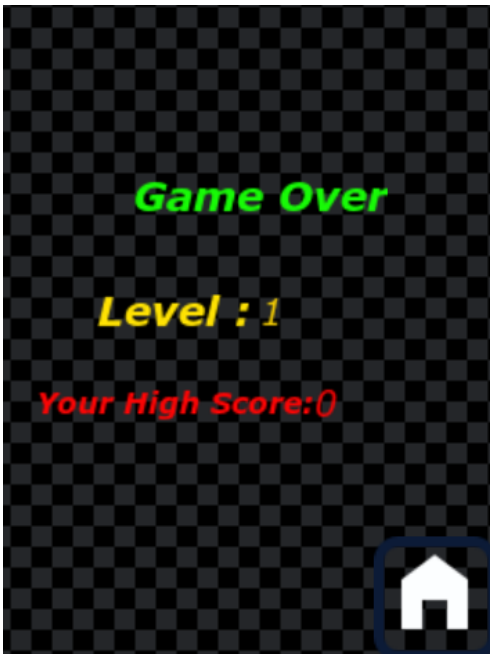


Figure 3.5: Gameover screen

Integration with Game Logic

TouchGFX components such as buttons, text areas, and images are connected to backend logic using `Model` and `Presenter` classes generated by TouchGFX. Button click events trigger state changes in the game logic module, and game states (e.g., score, lives, collision) update the display elements in real time.

Graphics Assets

Graphical assets such as background images, obstacle sprites, and dinosaur animation frames are imported into the TouchGFX Designer project.

3.1.2 Voice Input Processing Module

This module is responsible for capturing, processing, and classifying sound levels from the user's voice input to determine the dinosaur's jump height in the game. The voice control replaces traditional physical buttons, providing a hands-free and engaging gameplay experience.

Signal Flow Overview

- The **MAX4466** microphone outputs an analog signal corresponding to sound intensity.

- The **STM32F429ZIT6** samples this signal continuously using its internal ADC configured with DMA.
- The sampled data is processed in a dedicated RTOS thread to extract peak values.
- Based on the peak amplitude, the system classifies the sound into three levels: silence, low, and high.
- The result is then posted to a message queue and used by the main game logic to trigger appropriate jump actions.

Sound Detection Function

The following C function processes the ADC buffer to detect the peak sound value:

```
uint16_t detect_sound(uint16_t* data, uint16_t len)
{
    const uint16_t center = 2048;  // Assuming 12-bit ADC and 3.3V Vref
    const uint16_t clap_threshold = 400;
    uint16_t peak = 0;
    uint32_t average = 0;

    for (size_t i = 0; i < len; ++i)
    {
        if (data[i] > peak) {
            peak = data[i];
        }
        average += data[i];
    }

    return peak;
}
```

This function returns the peak value from a buffer of ADC samples. Thresholds are then applied to determine the jump level:

- **Peak below threshold:** No jump.
- **Peak within low-level range:** Short jump.
- **Peak exceeds high threshold:** High jump.

Real-Time Audio Processing with RTOS

The firmware runs two concurrent tasks under CMSIS-RTOS2:

- **soundInputTask (Priority: Above Normal)**
  Continuously processes the DMA-filled ADC buffer, detects the sound level, and pushes the result to a message queue.
- **GUI_Task (Priority: Normal)**
  Runs the TouchGFX graphics loop to update the game interface in real-time.

Thread and Queue Definitions

```
/* Definitions for soundInputTask */
osThreadId_t soundInputTaskHandle;
const osThreadAttr_t soundInputTask_attributes = {
  .name = "soundInputTask",
  .stack_size = 128 * 4,
  .priority = (osPriority_t) osPriorityAboveNormal,
};

/* Definitions for GUI_Task */
osThreadId_t GUI_TaskHandle;
const osThreadAttr_t GUI_Task_attributes = {
  .name = "GUI_Task",
  .stack_size = 8192 * 4,
  .priority = (osPriority_t) osPriorityNormal,
};

/* Definitions for soundInputQueue */
osMessageQueueId_t soundInputQueueHandle;
const osMessageQueueAttr_t soundInputQueue_attributes = {
  .name = "soundInputQueue"
};
```

Design Considerations

- **DMA** allows for non-blocking, high-speed sampling of the microphone without CPU intervention.
- **RTOS scheduling** ensures sound detection (higher priority) does not interfere with GUI rendering.
- **Message queue** provides safe communication between concurrent tasks without blocking.
- The architecture supports both responsiveness and UI performance, ensuring a smooth game experience.

3.1.3 Game Logic Module

Level Difficulty

The game offers three difficulty levels, each affecting the frequency and variety of obstacles the player encounters. This design ensures increasing challenge and maintains engagement across a wide range of player skill levels.

- **Level 1 – Easy:** Only one type of cactus is used as an obstacle. The spacing between obstacles is generous, providing more reaction time.
- **Level 2 – Medium:** Two types of cactus are introduced. The second type is wider than the first, requiring more precise jump timing. Obstacle generation becomes more frequent.
- **Level 3 – Hard:** In addition to the two cactus types, a flying bird appears as an aerial obstacle. The bird requires high-jump accuracy, and obstacles are generated at a higher rate.

Dinosaur movement

The dinosaur character's vertical movement is controlled by a simple physics-inspired state machine consisting of three animation states: `walkState`, `jumpState`, and `fallState` in `GameScreenView` class. The transitions and position updates are based on a vertical velocity variable `y_velocity`, which is modified each frame to simulate jumping and falling behavior.

State Definitions:

- `walkState`**:** The default idle state where the dinosaur is on the ground.
- `jumpState`**:** Triggered when a valid sound input is detected (either low or high). The dinosaur moves upward each frame based on the current `y_velocity`.
- `fallState`**:** Activated after the upward movement is complete. The dinosaur begins descending gradually due to increasing gravity.

Vertical Position Update: During each game tick, the dinosaur's Y position is updated depending on the current state:

```
// Position update
if (animation_state == jumpState) {
    dinosaur.moveTo(dinosaur.getX(), dinosaur.getY() - y_velocity);
}
else if (animation_state == fallState) {
```

```
    uint16_t future_pos = dinosaur.getY() + y_velocity;
    if (ground_pos > future_pos) {
        dinosaur.moveTo(dinosaur.getX(), future_pos);
    } else {
        dinosaur.moveTo(dinosaur.getX(), ground_pos);
        animation_state = walkState;
        animation_frame = 0;
    }
}
```

Velocity Update: After each position update, the velocity is adjusted based on the state:

```
// y_velocity update
if (animation_state == jumpState) {
    if (y_velocity == 0) {
        animation_state = fallState;
    } else {
        y_velocity--;
    }
}
else if (animation_state == fallState) {
    if (tickCount % 2 == 0) {
        if (y_velocity < 3) y_velocity++;
    }
}
```

Explanation:

- While in `jumpState`, the dinosaur ascends, and the vertical speed (`y_velocity`) decreases each frame until it reaches 0, simulating the top of a jump.
- Once `y_velocity` becomes 0, the state switches to `fallState`.
- In `fallState`, gravity is applied by gradually increasing `y_velocity` (every 2 ticks), causing the dinosaur to descend.
- When the dinosaur reaches the ground (`ground_pos`), it snaps back to its default walking state.

This logic provides a simple yet effective physics behavior without using floating point arithmetic, making it suitable for real-time embedded systems like STM32.

Obstacles generation

The obstacle generation is implemented inside the `GameScreenView` class and dynamically adjusts the complexity of obstacles based on the current game level. This behavior is controlled using the `obstacleCount` variable, which defines the range of obstacle types to randomly choose from.

The game uses a lightweight pseudo-random number generator based on the XORShift algorithm. This generator is used to determine both obstacle types and spawn positions. It manipulates the seed using bitwise operations to produce a new pseudo-random number each time it is called. The `getRandomInt()` function takes a minimum and maximum bound and returns a uniformly distributed integer within that range.

```
// Global seed
uint32_t seed = 2463534242;

// XORShift32 PRNG
uint32_t xorshift32() {
    seed ^= seed << 13;
    seed ^= seed >> 17;
    seed ^= seed << 5;
    return seed;
}

// Get random integer in [min, max]
int getRandomInt(int min, int max) {
    return min + (xorshift32() % (max - min + 1));
}
```

Obstacles are spawned at randomized time intervals to keep the gameplay unpredictable. The main game loop, specifically the `tickEvent()` function, checks whether any obstacles are currently active. If not, it triggers a new obstacle generation based on a randomly selected `obstacleType`. Depending on the type, the game may spawn a cactus, a bird, or both. For certain obstacle types, the system also randomly decides whether to include a special cactus alongside the regular one.

Once spawned, obstacles begin moving leftward across the screen. The cactus and bird are independently managed with flags (`cactus_running`, `bird_flying`) that track their active status. If an obstacle moves out of view (past the left edge of the screen), it is marked as inactive and eligible for respawning.

Collision detection

Collision detection is a critical component in the gameplay loop, ensuring the game can respond when the player character comes into contact with obstacles. In this implementation, collision checks are performed every tick using bounding box comparisons between

the dinosaur character and each obstacle on screen.

The core logic relies on the `checkCollision()` function, which takes the coordinates and dimensions of two rectangles and returns `true` if they overlap. The dinosaur's bounding box is offset using a predefined hitbox array, while the obstacles use their current on-screen positions and dimensions.

```
bool checkCollision(int x1, int y1, int w1, int h1,
                    int x2, int y2, int w2, int h2) {
    return (x1 < x2 + w2) &&
           (x1 + w1 > x2) &&
           (y1 < y2 + h2) &&
           (y1 + h1 > y2);
}
```

Each tick, the game checks for collisions between the dinosaur and three types of obstacles: the regular cactus, the special cactus (if present), and the bird. If any collision is detected and the hit cooldown timer has expired, the player loses one life, and a flickering effect is activated to visually indicate damage. The cooldown mechanism prevents lives from being deducted too rapidly from continuous collisions.

Lives and Score Update

During each game tick, the system updates the player's score and remaining lives.

The score increases continuously based on the time the player survives. This increment is multiplied by a variable called `speed_multiplier`, which scales with performance to increase difficulty. Initially set to 1, the multiplier increases to 2 and 3 once the player reaches 2000 and 4000 points respectively.

- This dynamic scaling ensures that the game becomes more challenging as the player performs better.
- The current score is converted to a string using a buffer and displayed on the screen in real-time.
- This live feedback motivates players to stay engaged and aim for high scores.

The player starts with three lives, visually represented by health icons on the user interface. When a collision with an obstacle is detected and the cooldown period has passed, the player loses one life.

- A flickering visual effect is triggered after each hit to indicate temporary invincibility.
- For each lost life, a corresponding heart icon is updated to reflect the new health status.
- When all lives are depleted, the game ends and transitions to the game over screen.

- At that point, the current score is compared against the level-specific high score. If it exceeds the previous high score, the record is updated.

3.2 Result

A full gameplay demonstration video is available at the following link:

https://drive.google.com/drive/folders/18XhtlLegr15L4UNeDgrlWF9TXNxdcqQa

3.3 Evaluation

The game successfully meets the core functional requirements: generating a continuous side-scrolling experience with dynamic obstacle spawning, collision detection, real-time score tracking, and player feedback through visuals and sound input. The integration with hardware features such as sound level detection (for jumping) makes the game interactive and engaging.

Strengths

- The software runs stably and fulfills all major gameplay use cases.
- Core features such as obstacle spawning, score/life updates, and background movement are implemented smoothly.
- The game includes multiple levels with increasing difficulty, enhancing replayability.
- There is a sound-based control mechanism using a microphone, offering a fresh and engaging way to play.

Weaknesses

- The microphone input can be unreliable in noisy environments, leading to false or missed jump triggers. If the ambient noise level is too high, it may cause the player to jump continuously without intention.
- There are limited types of obstacles and animations, which may reduce visual variety in long play sessions.
- The game lacks pause and restart features, requiring a full reset after game over.

Overall, the game satisfies the intended design goals and runs effectively on the target hardware. Its voice-controlled gameplay mechanic sets it apart from traditional games.