HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Final Project Report

Space Invaders - STM32F429ZIT6

Group: 1

Course:	Embedded System
Instructor:	Ph.D. Ngo Lam Trung
Students:	Nguyen Thanh Tung – 20226071
	Trinh Luong Viet – 20235967
	Le Quoc Tuan – 20236008

Contents

Chapter 1

Introduction

1.1 Project Introduction and Motivation

The **Soldier vs Sheep** project is a sophisticated real-time embedded application that replicates the classic mechanics of the *Space Invaders* genre. Developed for the `STM32F429ZIT6` microcontroller, this project serves as a comprehensive case study in integrating high-performance graphical user interfaces (`GUI`) with low-level hardware control. The primary motivation is to demonstrate that modern micro-controllers are capable of delivering smooth, arcade-style experiences when programmed with efficient architectural patterns and hardware-accelerated rendering techniques.

1.2 Hardware Platform Strengths

The choice of the `STM32F429_Discovery` board is fundamental to the project's success. At its core is the `ARM Cortex-M4` processor, which operates at a high frequency of **180MHz**. This computational power allows the system to process complex game logic, such as collision detection and state management, within the strict timing constraints of a real-time display.

A critical feature of this hardware is the presence of dedicated graphical accelerators. The `LTDC` (LCD-TFT Display Controller) manages the raw pixel transmission to the 2.4-inch display, while the `DMA2D` (Chrom-ART Accelerator) offloads intensive tasks such as memory-to-memory pixel blending and area clearing. These hardware blocks are vital for maintaining a consistent 60 frame-per-second (`FPS`) experience. By offloading these tasks, the `CPU` is freed to focus on the game loop and peripheral interrupts, ensuring that the system remains responsive even during high-action sequences with multiple active sprites.

1.3 Software Stack and Methodology

The software integration is achieved through a robust toolchain. `STM32CubeIDE` is used for the underlying system configuration and `HAL` (Hardware Abstraction Layer) management. The `HAL` provides a consistent set of `APIs` to interact with the `GPIO` pins, `RNG` (Random Number Generator), and `FMC` (Flexible Memory Controller).

The graphical layer is built upon `TouchGFX`, an advanced `C++` framework that follows an object-oriented paradigm. This framework allows for the encapsulation of game entities—such as the `Soldier`, `Sheep`, and `Bullets`—as class objects. Using `C++` provides the benefits of clean inheritance and modular design, which are essential for implementing patterns like **Model-View-Presenter (MVP)**. This methodology avoids the "Spaghetti Code" common in simpler embedded projects and prepares the application for professional scaling and maintenance.

1.4 Project Objectives

The primary objective of this project is to demonstrate the convergence of high-level software engineering and low-level hardware optimization. Specifically, the project aims to:

- Implement a robust **MVP** architecture to decouple display logic from hardware signals.

- Utilize **Object Pooling** to achieve deterministic memory performance and prevent heap fragmentation.
- Integrate hardware peripherals like the `RNG` and `Buzzer` to enhance gameplay dynamics and user feedback.
- Maintain a rock-solid 60 `FPS` performance across all game states.

Through these goals, the *Soldier vs Sheep* project proves the viability of `STM32` platforms for professional-grade interactive graphical applications.

Chapter 2

Software Architecture

2.1 Overview of MVP in TouchGFX

The success of a sophisticated embedded application depends on its underlying architectural philosophy. In this project, we utilize the **Model-View-Presenter (MVP)** pattern, which is the native architectural standard for the `TouchGFX` framework. This section explores why this pattern is superior for resource-constrained systems and how it facilitates a clean data flow between hardware and software.

2.1.1 Why use MVP for Embedded Systems?

Developing a graphical application on a microcontroller like the `STM32F429` presents unique challenges. Without a formal architecture, many developers fall into the trap of creating "Spaghetti Code"—where hardware drivers, game logic, and display commands are all mixed within a single file.

Separation of Concerns

The core principle of **MVP** is the **Separation of Concerns**. It is dangerous to mix UI code with hardware driver code. For example, if a developer places a `HAL_Delay()` call directly inside a screen update function, the entire user interface will freeze, leading to a poor user experience. By separating these duties, the `Model` handles the slow hardware peripherals, while the `View` focuses entirely on high-speed rendering.

Maintainability and Testability

Using the **MVP** pattern significantly improves **Maintainability**. If we decide to change the display hardware or move from a 2.4-inch to a 7-inch screen, we only need to modify the `View` layer; the core game logic in the `Model` remains untouched. Furthermore, this decoupling enhances **Testability**. The computational game logic (such as score calculation or collision rules) can be tested independently of the physical `LCD`, saving significant development time during the debugging phase. Finally, this pattern ensures **Resource Efficiency** by allowing the `STM32` to update only the modified regions of the screen, reducing the load on the `DMA2D` and `LTDC` controllers.

2.1.2 Data Flow Diagram and Framework Mechanism

The **MVP** pattern in `TouchGFX` is structured as a triangular relationship. Each component has a specific communication path that prevents tight coupling between high-level and low-level classes.

The Push vs. Pull Mechanism

The communication between layers is governed by a **Push-Pull** mechanism:

- **The Push (Model to Presenter)**: When a hardware event occurs, such as a physical button press on `PA0`, the `Model` "pushes" this update to the `Presenter`. This is done through the `ModelListener` interface.
- **The Pull (View to Presenter)**: When the `View` needs information about the current state of the game (e.g., "Is the high score reached?"), it "pulls" or requests this data via the `Presenter`.

Technical Implementation via C++ Interfaces

Technically, the `TouchGFX_Engine` implements this bridge using C++ inheritance and virtual functions. The `Model` class holds a pointer to the `ModelListener` interface. Because the `Screen1Presenter` inherits from `ModelListener`, it can receive notifications from the `Model` without the `Model` knowing any details about the specific screen or its widgets. This use of **Polymorphism** is what allows the system to remain flexible and robust.

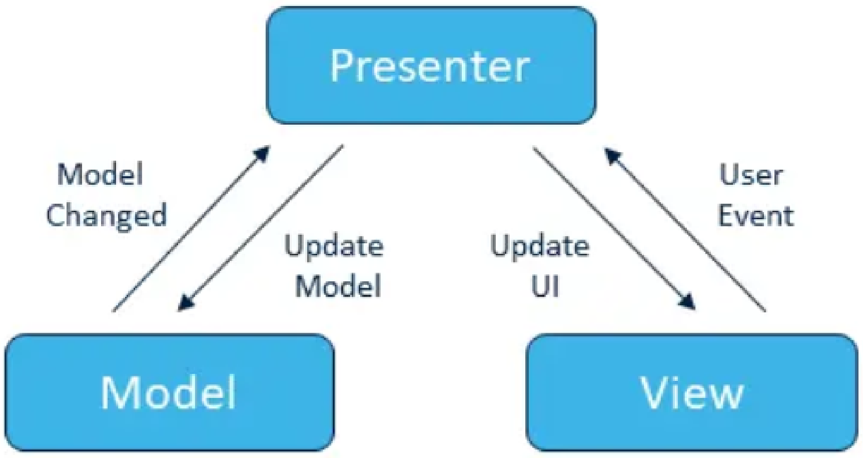

Figure 2.1: The relationship between Model, View, and Presenter in our project.

2.2 Detailed Component Analysis

The success of the *Soldier vs Sheep* project relies on a clear division of responsibilities across the three architectural layers. By analyzing the source code of the `Model`, `View`, and `Presenter`, we can observe how each component manages a specific aspect of the embedded system, from low-level `GPIO` signals to high-level sprite animations.

2.2.1 The Model (The Data Layer) - Analysis of `Model.cpp`

The `Model` class acts as the persistent backbone of the application. In the `TouchGFX` framework, the `Model` is a singleton-like structure that is instantiated when the system boots and remains active throughout the entire application lifecycle. This ensures that global data, such as the `highScore`, is preserved even when navigating between different screens.

The System Heartbeat: `tick()`

The most critical function in the `Model` is the `tick()` method. This function is triggered 60 times per second by the `TouchGFX` engine, synchronized with the `LTDC` V-Sync interrupt. Because of this timing, `tick()` serve as the reliable "System Heartbeat". Any logic placed inside this function is guaranteed to execute at a precise interval of 16.67 milliseconds.

Low-Level Hardware Interface

The `Model` is the only component permitted to interact with the **Hardware Abstraction Layer (HAL)**. It uses `extern` declarations to access C-based structures defined in `main.c`. For example, it utilizes the `HAL_GPIO_ReadPin` function to monitor the physical movement buttons and `HAL_GPIO_WritePin` to drive the sound `Buzzer`. This encapsulation keeps the rest of the application protected from changes in hardware pin assignments.

```cpp
void Model::tick()
{
    // The heartbeat counter increments every 16.6ms
    modelTickCount++;

    #ifndef SIMULATOR
    // Direct interaction with STM32 HAL library
    GPIO_PinState left_pin = HAL_GPIO_ReadPin(GPIOE, GPIO_PIN_2);

    // Edge detection logic to identify a valid button press
    if (prev_left_state == GPIO_PIN_SET && left_pin == GPIO_PIN_RESET)
    {
        // Signal the Presenter via the Listener interface
        modelListener->onLeftPressed();
    }
    prev_left_state = (left_pin == GPIO_PIN_SET);
    #endif
}
```

Listing 2.1: Hardware Polling and Heartbeat in Model.cpp

2.2.2 The View (The UI Layer) - Analysis of `Screen1View.cpp`

While the `Model` manages the data, the `View` is responsible for the visual representation on the 2.4-inch `LCD`. It is tightly coupled with the `TouchGFX` rendering engine and dictates how objects appear to the player.

Visual Game Logic: `handleTickEvent()`

The `View` contains its own loop called `handleTickEvent()`, which is executed immediately after the `Model::tick()`. This is where the "Visual Game Logic" occurs. This includes updating the coordinates of the `Soldier`, moving the `Sheep` down the Y-axis, and calculating the paths of `Bullets`. By keeping these visual updates in the `View`, we ensure that movements appear fluid and responsive at 60 `FPS`.

Widget and Memory Management

The `View` holds references to all UI widgets defined in the `TouchGFX Designer`, such as `soldier`, the `enemyPool`, and the `scoreText` labels. One of the most important methods used here is `invalidate()`. When a sprite moves, the `View` must call `invalidate()` on both its old and new positions. This marks the area as "dirty", commanding the `DMA2D` (Chrom-ART Accelerator) to redraw only the necessary pixels. This optimization is vital for maintaining high performance on a microcontroller with limited memory bandwidth.

```cpp
void Screen1View::handleTickEvent()
{
    // Skip updates if the game state is not 'Playing'
    if (is\_paused || is\_game\_over) return;

    // Iterate through the Object Pool to move active Sheep
    for (int i = 0; i < ENEMY_POOL_SIZE; i++)
    {
        if (enemyPool[i]->isVisible())
        {
            // Update Y-coordinate for downward movement
            int current_y = enemyPool[i]->getY();
            enemyPool[i]->setY(current_y + 3);

            // Mark the widget for redrawing by the DMA2D
            enemyPool[i]->invalidate();
        }
    }
}
```

Listing 2.2: Visual update loop in Screen1View.cpp

2.2.3 The Presenter (The Bridge Layer) - Analysis of `Screen1Presenter.cpp`

The `Presenter` is the intermediary that prevents the `View` and `Model` from communicating directly. This "Bridge" architecture is vital for maintaining a clean and testable codebase.

Implementing the Listener Pattern

The `Presenter` implements the `ModelListener` interface. When the `Model` detects a hardware event (like the button edge detection in Section 2.2.1), it does not know which screen is active. It simply calls `modelListener->onLeftPressed()`. Because the `Screen1Presenter` currently implements this listener, it receives the notification and decides how to route the information.

Smart Data Routing

The `Presenter`'s primary role is data routing. It takes ”Raw” data from the `Model` and translates it into ”Visual” commands for the `View`. For example, when the `Model` signals a button press, the `Presenter` commands the `View` by calling `view.moveSoldierLeft()`. This keeps the `View` ”dumb”—meaning it only knows how to move images when told—and the `Model` ”pure”—meaning it only knows about hardware states and data. This separation of concerns allows developers to change the hardware (Model) or the UI (View) independently without breaking the entire project.

Chapter 3

Game Logic Design

3.1 Hardware Random Number Generator (RNG)

The ability to generate unpredictable events is fundamental to the game mechanics of *Soldier vs Sheep*. In this system, randomness is required to determine the lane assignment for each newly spawned `Sheep` entity. To ensure a truly non-deterministic experience, we utilize the dedicated **Hardware Random Number Generator (RNG)** peripheral integrated into the `STM32F429ZIT6` microcontroller.

3.1.1 Physical Entropy and RNG Architecture

The `STM32F429 RNG` is a true hardware random number generator that leverages physical entropy. Unlike software-based algorithms such as the standard C `rand()` function, which are **Pseudo-Random Number Generators (PRNG)**, the hardware `RNG` provides non-deterministic output.

A `PRNG` relies on a mathematical formula and a starting value called a ”seed”. If the seed is known or remains constant—which is often the case upon a hardware reset in embedded systems—the resulting sequence will be identical every time. In contrast, the `STM32 RNG` captures analog noise from the internal semiconductor circuitry. This analog signal is sampled and processed through a linear-feedback shift register (LFSR) to produce a 32-bit value. This ensures that the `Sheep` spawn in unique patterns during every game session, providing high replay value and fairness.

3.1.2 Clock Configuration and Initialization

The `RNG` peripheral requires a stable and specific clock source to function. Based on the analysis of the `STM32F429I_DISCO_REV_D01.ioc` configuration file, the `RNG` is supplied by the **PLLQ** clock output.

In the clock tree configuration, the High-Speed External (HSE) oscillator (8MHz) is used as the system source. This is multiplied by the main PLL to reach the system clock of 180MHz (HCLK). However, the `RNG` requires a 48MHz clock frequency to satisfy the USB and RNG standards. As seen in the `.ioc` parameters:

- `RCC.PLLM = 4`
- `RCC.PLLN = 72`
- `RCC.PLLQ = 3`
- `RCC.PLLQCLKFreq_Value = 48000000` (48 MHz)

This dedicated 48MHz clock ensures that the analog noise sampling rate is consistent and meets the hardware requirements for entropy generation. The initialization is performed via the `MX_RNG_Init()` function, which initializes the handle and confirms the hardware occupancy.

```c
static void MX_RNG_Init(void)
{
  /* Initialize the RNG handle */
  hrng.Instance = RNG;
  if (HAL_RNG_Init(&hrng) != HAL_OK)
  {
    /* If the RNG fails to initialize, the system enters an Error Handler */
    Error_Handler();
  }
}
```

Listing 3.1: Hardware RNG Initialization in main.c

3.1.3 Implementation and Lane Calculation Logic

Within our **MVP** architecture, the `Model` class serves as the interface to the hardware. The `Model::GetRandomNumber()` function encapsulates the `HAL` call, providing a safe abstraction for the `Presenter` and `View` layers.

```
uint32_t Model::GetRandomNumber()
{
#ifndef SIMULATOR
    uint32_t randomnumber = 0;
    /* HAL_RNG_GenerateRandomNumber blocks until a new 32-bit
       number is ready in the RNG data register */
    if (HAL_RNG_GenerateRandomNumber(&hrng, &randomnumber) == HAL_OK)
    {
        return randomnumber;
    }
    return 0; // Fallback to 0 if hardware health check fails
#else
    return (uint32_t)std::rand();
#endif
}
```

Listing 3.2: HAL_RNG usage in Model.cpp

A critical aspect of the implementation is the **Status Check**. Since the `RNG` uses physical noise, it may occasionally fail if the environmental conditions are unstable (e.g., voltage drops). By checking the `HAL_OK` status, we ensure that the game does not use uninitialized or ”stale” data.

Transformation to Lane Indices

The raw output of the `RNG` is a 32-bit integer ranging from 0 to 4,294,967,295. To map this enormous value to our specific 6-lane game world, we employ the **Modulo Operation**. By taking `random_val % 6`, we effectively ”wrap” the value into a remainder between 0 and 5.

```
// Retrieve the 32-bit hardware value via Presenter
uint32_t raw_val = presenter->GetRandomNumber();

// Map the value to one of the 6 lanes (Indices 0 to 5)
int target_lane = raw_val % 6;
```

Listing 3.3: Lane index calculation in Screen1View.cpp

This mathematical transformation guarantees that even if the raw bitstream is uniformly distributed, the resulting lane distribution remains balanced, preventing any lane from becoming objectively more difficult or predictable than others.

3.2 Object Pooling Mechanism (Memory Optimization)

In high-performance real-time applications, particularly those running on resource-constrained embedded systems like the `STM32F429`, memory management is a primary concern. The *Soldier vs Sheep* project relies on the efficient handling of multiple moving entities, such as `Sheep` (enemies) and `Bullets`. To ensure the long-term stability and responsiveness of the game, we implemented an **Object Pooling** mechanism.

3.2.1 Embedded Constraints and Dynamic Memory Dangers

Standard software development often relies on dynamic memory allocation via functions such as `malloc()` or the C++ `new` operator. However, in an embedded game loop that executes 60 times per second, dynamic allocation presents significant risks. The primary danger is **Heap Fragmentation**. When objects of varying sizes are repeatedly allocated and freed, the heap becomes ”checkerboarded” with small, unusable gaps.

Over a continuous gaming session, the heap may reach a state where it cannot satisfy an allocation request despite having enough total free memory. On a microcontroller lacking a complex memory management unit (`MMU`), this results in a system crash or a hard fault. Furthermore, the time required for `malloc()` to search for a free block is non-deterministic, which can cause frame-rate stutters (jitter). To achieve consistent **Deterministic Performance**, our project uses static memory allocation, where all game objects are reserved in memory at compile time.

3.2.2 Static Array Implementation

We implemented the object pools as static member arrays within the `Screen1View` class. This approach ensures that the total memory footprint for game entities is fixed and known before the program starts. As shown in the header configuration below, we define a maximum capacity for each entity type.

```cpp
class Screen1View : public Screen1ViewBase
{
protected:
    // Pool for enemies (Sheep images)
    static const int ENEMY_POOL_SIZE = 6;
    touchgfx::Image* enemyPool[ENEMY_POOL_SIZE];

    // Pool for projectiles (Bullet images)
    static const int BULLET_POOL_SIZE = 1;
    touchgfx::Image bulletPool[BULLET_POOL_SIZE];
};
```

Listing 3.4: Definition of Object Pools in Screen1View.hpp

The `enemyPool` contains pointers to predefined images for the `Sheep`, while the `bulletPool` is a direct array of `touchgfx::Image` objects. By limiting the number of total active objects, we prevent the system from ever exceeding its physical `RAM` limits.

3.2.3 Object Lifecycle Management

The intelligence of the pooling system lies in the management of the object lifecycle. Instead of destroying an object when it is no longer needed, we simply change its state and hide it from the rendering engine. The lifecycle consists of four distinct stages:

1. **Inactive (Idle)**: Objects are initialized with `setVisible(false)`. They exist in memory but are ignored by the `TouchGFX` draw logic. This state consumes no CPU cycles for rendering.
2. **Spawning (Activation)**: When the `spawnTimer` reaches its threshold, the system iterates through the array to find the first object where `isVisible()` returns false. Once found, the object's X and Y coordinates are reset to the top of the screen in a lane determined by the `RNG`.
3. **Active (Rendering)**: The object is set to `setVisible(true)`. We then call the `invalidate()` function on the object. This tells the `TouchGFX` engine that this specific screen area must be redrawn in the next frame.
4. **Recycling (Deactivation)**: When a `Sheep` is hit by a `Bullet` or moves beyond the bottom screen-boundary (`Y > 320`), we call `setVisible(false)` and `invalidate()`. The object immediately returns to the *Inactive* state, becoming available for the next spawn event.

```cpp
void Screen1View::handleTickEvent()
{
    if (spawnTimer >= 60)
    {
        for (int i = 0; i < ENEMY_POOL_SIZE; i++)
        {
            // Search for an available (inactive) sheep
            if (!enemyPool[i]->isVisible())
            {
                // Reset entity parameters
                enemyPool[i]->setY(-enemyPool[i]->getHeight());
                enemyPool[i]->setVisible(true);
                enemyPool[i]->invalidate();

                spawnTimer = 0; // Reset spawn interval
                break; // Stop after activating one sheep
            }
        }
    }
}
```

Listing 3.5: Iterative searching and activation logic in Screen1View.cpp

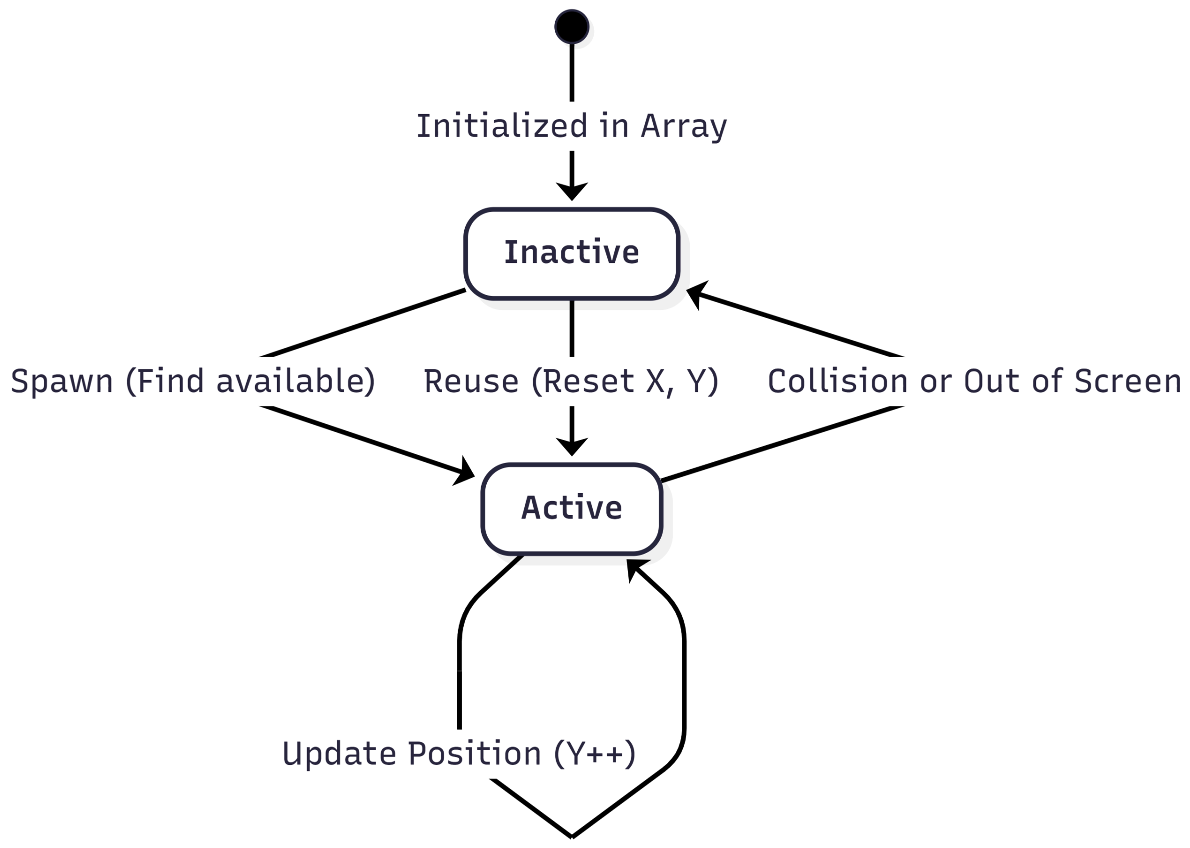

Figure 3.1: Object Pooling Mechanism

3.2.4 Efficiency and Stability Analysis

The primary advantage of this mechanism is that the **RAM usage remains constant** regardless of the game intensity. Whether there are zero sheep or six sheep on the screen, the memory allocated remains exactly the same.

From a performance standpoint, the complexity of finding a free object is `O(N)`, where `N` is the pool size. Since our pool sizes are small (`N=6`), this operation is extremely fast and occurs within the `GUI` task without causing frame drops. This stability allows the *Soldier vs Sheep* game to run for hours without any performance degradation, making it a robust solution for a production-level embedded device.

3.3 Collision Detection

Collision detection is a critical computational task in any real-time game. It determines the interactions between the player's projectiles and the enemy entities. In the *Soldier vs Sheep* project, we must efficiently verify if a `Bullet` has hit a `Sheep` within a 16.6ms window (60 FPS) to maintain a smooth user experience.

3.3.1 Axis-Aligned Bounding Box (AABB) Principle

The theoretical foundation for our collision system is the **Axis-Aligned Bounding Box (AABB)** algorithm. This method treats every graphical object as a non-rotated rectangle aligned with the X and

Y axes of the screen coordinate system.

Mathematically, two rectangles R_1 and R_2 overlap if and only if they satisfy the following four conditions simultaneously:

$$(R_1.x < R_2.x + R_2.width) \wedge (R_1.x + R_1.width > R_2.x) \wedge (R_1.y < R_2.y + R_2.height) \wedge (R_1.y + R_1.height > R_2.y)$$

If any of these conditions are false, the objects are separated. While this math is simple, performing it repeatedly in a nested loop can become a performance bottleneck. The `TouchGFX` framework provides a highly optimized implementation of this logic via the `touchgfx::Rect::intersect()` function, which we utilize for high-precision detection.

3.3.2 The Computational Bottleneck

In a naive implementation, a game would check every active `Bullet` in the `bulletPool` against every active `Sheep` in the `enemyPool`. For pools of size N and M, this results in an algorithmic complexity of $O(N \times M)$ per frame. While our current pool sizes are small (`N=1`, `M=6`), expanding the game to include dozens of objects would eventually strain the `STM32F429` CPU, especially since it must also handle gravity, input, and complex UI rendering tasks.

3.3.3 Performance Optimization: Lane Filtering

To mitigate this bottleneck, we implemented a **Lane-based Filtering** strategy. Both the `Soldier` and the `Sheep` move within 6 fixed horizontal positions (Lanes). Because a hit can only occurring if two objects are vertically aligned, we can use the `X` coordinate as a high-speed filter.

By checking `bullet.getX() == enemy.getX()` before calling the more complex `intersect()` function, we perform an ”Early Exit”. If the X-coordinates do not match, we know a collision is mathematically impossible, and we skip the boundary calculation entirely. This reduces the number of expensive geometric calls significantly.

```cpp
void Screen1View::handleTickEvent()
{
    // Iterate through Bullet Pool
    for (int i = 0; i < BULLET_POOL_SIZE; i++)
    {
        if (bulletPool[i].isVisible())
        {
            // Iterate through Enemy Pool
            for (int j = 0; j < ENEMY_POOL_SIZE; j++)
            {
                if (enemyPool[j]->isVisible())
                {
                    // STEP 1: Lane Optimization (Fast Integer Check)
                    if (bulletPool[i].getX() == enemyPool[j]->getX())
                    {
                        // STEP 2: Precise AABB Check using TouchGFX API
                        if (bulletPool[i].getRect().intersect(enemyPool[j]->getRect()))
                        {
                            // Collision handling logic
                            handleHit(i, j);
                        }
                    }
                }
            }
        }
    }
}
```

Listing 3.6: Optimized Collision Nested Loop in Screen1View.cpp

[Figure Placeholder: AABB Collision with Lane Filtering Diagram]

3.3.4 Post-Collision Lifecycle Operations

When a collision is confirmed, the system must synchronize the game state with the visual interface. This involves three critical steps:

1. **Deactivation**: Both the `Bullet` and the `Sheep` have their visibility set to false via `setVisible(false)`. This returns them to the *Inactive* state of the object pool.
2. **Invalidation**: The `invalidate()` function is called for both objects. This is vital because `TouchGFX` uses smart-drawing; it only updates areas that have changed. Without `invalidate()`, the image of the "destroyed" sheep would remain frozen on the screen.
3. **Event Propagation**: The `View` sends a signal to the `Presenter` (`presenter->playScoreSound()`), which then commands the `Model` to trigger the hardware buzzer and increment the scoreboard variables.

This optimized pipeline ensures that the `STM32F429` maintains a consistent 60 FPS even during intense gameplay moments with multiple simultaneous collisions.

3.4 Game State Machine

A finite state machine is an essential component for managing the complex behaviors and transitions within a real-time system. In the *Soldier vs Sheep* project, the state machine coordinates how the game reacts to user inputs and internal logic events, such as collisions or pauses. By defining distinct states, we ensure that the system remains predictable and that the user interface (`UI`) stays synchronized with the underlying game logic.

3.4.1 State Definitions and Flags

The system utilizes boolean flags to represent the three primary operational states. This approach allows for a simple yet robust transition logic within the highly constrained environment of the `STM32`.

- **Playing State**: This is the default active mode where all subsystems—movement, spawning, and collision detection—are fully functional. In this state, `is_paused` and `is_game_over` are both false.
- **Paused State**: Triggered by the user (typically via a `TouchGFX` button interaction), this state halts all entity updates. The game remains resident in memory, but the "heartbeat" of the logic loop is effectively suspended.
- **Game Over State**: This terminal state is triggered when a `Sheep` successfully reaches or collides with the `Soldier`. In this state, gameplay logic is permanently frozen until a hard reset occurs.

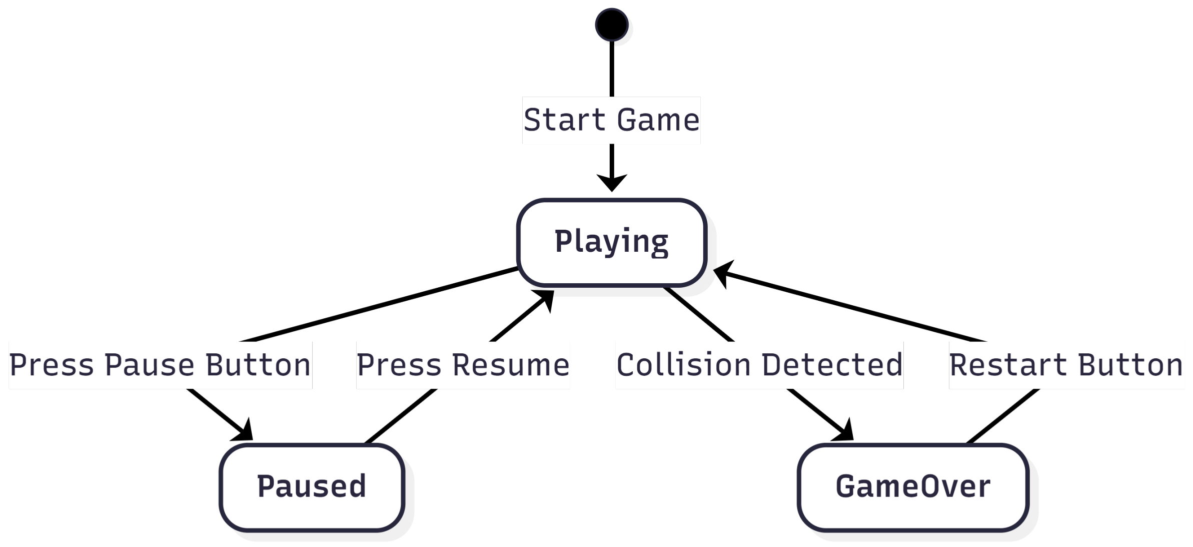

Figure 3.2: State Machine Diagram

3.4.2 Detailed State Analysis and Early Exit Strategy

The core logic loop resides within the `Screen1View::handleTickEvent()` function. To maintain high performance and prevent unnecessary calculations during non-active states, we implement an **Early Exit** (or **Early Return**) strategy.

At the beginning of every frame, the system checks the state flags. If either `is_paused` or `is_game_over` is true, the function returns immediately. This prevents the CPU from executing any movement or collision mathematical operations, effectively freezing all sprites in their current positions while the `TouchGFX` engine continues to render the static scene.

```
void Screen1View::handleTickEvent()
{
    // The framework calls this every 16.6ms (60 fps)
    Screen1ViewBase::handleTickEvent();

    // Logic-UI Synchronization check
    if (is\_game\_over) return; // Halt all operations
    if (is\_paused) return;    // Suspension mode

    // Proceed with dynamic game updates only if in "Playing" state
    tickCount++;
    updateEntities();
}
```

Listing 3.7: Early Exit logic in Screen1View.cpp

3.4.3 Logic-UI Synchronization (The MVP Link)

A critical requirement of the **MVP** pattern is that the `View` must never update itself autonomously based on hardware data. Instead, a strict sequence of event propagation must be followed. When a state change occurs in the `Model`, it must travel through the `Presenter` before reaching the screen.

Consider the transition to the **Game Over** state:

1. **Logic Detection**: The `handleTickEvent()` loop detects a coordinate overlap between an enemy and the player.
2. **State Flip**: The variable `is_game_over` is set to true.
3. **Presenter Signal**: The `View` or `Model` notifies the `Presenter`.
4. **UI Command**: The `Presenter` commands the `View` to show overlays.
5. **Rendering Update**: The `View` calls `box_game_over.setVisible(true)`. Crucially, it must then call `box_game_over.invalidate()`.

The `invalidate()` call is the most important technical detail here. `TouchGFX` does not redraw the whole screen to save power. Instead, it only redraws "dirty" regions. By calling `invalidate()`, we mark the **Game Over** box as dirty, forcing the `DMA2D` or `LTDC` to push the new pixels to the `LCD`.

```
void Screen1View::handleCollisionWithPlayer()
{
    is\_game\_over = true;

    // Synchronize UI widgets with internal logic state
    gameOverOverlay.setVisible(true); // Show red overlay
    buttonNew.setVisible(true);       // Enable replay button

    // Force the TouchGFX engine to redraw these specific widgets
    gameOverOverlay.invalidate();
    buttonNew.invalidate();
}
```

Listing 3.8: UI Update handling during Game Over state change

3.4.4 Transition and Reset Logic

Returning from a terminal state like **Game Over** to the active **Playing** state requires a full system reset. In our project, this is handled through the `resetGame()` or `newGame()` function. This function performs a "hard-clear" of the system: it resets the score to zero, moves the player back to the starting lane, hides all active enemies in the pools, and hides the UI overlays. Once the cleanup is complete, the state flags are cleared, allowing the `handleTickEvent()` loop to resume its normal 60Hz execution.

[Figure Placeholder: Game State Transition Diagram]

3.5 Scoring System Management

The scoring system is a vital feedback loop that measures player performance and provides a sense of achievement. In the *Soldier vs Sheep* project, we implemented a real-time scoring mechanism that manages volatile session data and persistent high-score records across the **MVP** architecture.

3.5.1 Real-time Score Calculation and Data Flow

The scoring process begins at the moment of collision detection (as described in Section 3.3). When the `View` identifies that a `Bullet` and a `Sheep` have intersected, it immediately increments a local integer variable called `currentScore`.

However, simply updating a local variable is insufficient for a robust application. The scoring data must travel through multiple layers:

1. **View**: Increments the score and updates the visual text widgets.
2. **Presenter**: Forwards the new score to the model to check for records.
3. **Model**: Stores the value and compares it against the historical best.

This data flow ensures that the game logic (`Model`) and the display (`View`) remain synchronized. By centralizing the score calculation in the `View` during the frame update, we achieve low latency between the physical explosion effect and the incrementing digits on the screen.

3.5.2 UI Rendering and Unicode Conversion

A significant technical challenge in embedded `GUI` development is the handling of numeric data within text-based widgets. `TouchGFX` uses **Unicode (UTF-16)** for all its internal text buffers to support multiple languages and character sets. However, the game score is stored as a 32-bit `int`.

To display the score, we must perform a string conversion. We utilize the `touchgfx::Unicode::snprintf()` function. This function is an optimized, thread-safe version of the standard C `sprintf()`. It handles the conversion of integers into a wide-character buffer that the `LCD` controller can interpret.

```cpp
void Screen1View::updateScore(int score)
{
    // Clear and format the Unicode buffer with the integer value
    // The format string "%d" is used to convert base-10 integers
    Unicode::snprintf(textArea1Buffer, TEXTAREA1\_SIZE, "%d", score);

    // Crucial step: Mark the textArea as dirty for redrawing
    textArea1.invalidate();
}
```

Listing 3.9: Unicode conversion and UI update logic in Screen1View.cpp

As noted in the code above, calling `textArea1.invalidate()` is mandatory after updating the buffer. Because `TouchGFX` uses a sophisticated invalidation system to save memory bandwidth, it will not redraw the digits unless the software explicitly signals that the content of the buffer has changed.

[Figure Placeholder: Score Update Data Flow (View-Presenter-Model)]

3.5.3 High-Score Persistence in the Model Layer

While the `currentScore` is relevant only for the active game session, the high-score is a persistent record. Following the **MVP** architectural rules, we store the high-score within the `Model` class. Storing it in the `Model` rather than the `View` ensures that the value is not lost if the user navigates to a menu or if the screen is re-initialized.

```cpp
void Model::saveHighScore(int score)
{
    // Compare the current session score with the stored maximum
    if (score > high\_score)
    {
        high\_score = score;

        // At this point, the Model could also command a flash-write
        // for permanent storage in non-volatile memory.
    }
}
```

Listing 3.10: High-score comparison logic in Model.cpp

This logic performs a simple "greater-than" check. If the player exceeds the record, the `Model` updates its internal state. In the current implementation, this data stays in the `SRAM`. While this is volatile (lost on power-off), it provides a consistent experience during a long power-on session.

3.5.4 Feedback and User Experience (UX)

To enhance the user experience, the scoring system is tightly integrated with the hardware feedback layers. Whenever a successful hit is recorded in `Screen1View`, the system doesn't just update the UI; it also sends a signal through the `Presenter` to the `Model` to trigger the **Buzzer**. This multi-modal feedback—visual (text change), interactive (sheep disappearing), and auditory (buzzer sound)—creates a satisfying "game-feel" that is essential for player engagement. This synchronization demonstrates bridge capabilities of the **MVP** pattern, where a high-level UI event directly influences low-level hardware registers.

Chapter 4

Peripheral and System Interaction

4.1 Input Handling (Physical Push Buttons)

The interaction between the physical world and the digital game logic is managed through a low-level hardware abstraction layer. This section details the technical bridge between mechanical button presses and the execution of game commands within the **MVP** architecture.

4.1.1 Hardware Connection Schema

The `STM32F429I-DISCO` board provides several physical inputs used for player control. We utilize both the on-board User Button and external mechanical switches connected to the `GPIO` headers.

Pin Mapping and Electrical Logic

To ensure stable logic levels, we configured the `GPIO` pins using internal resistors. This prevents "floating" signals that could cause random move or fire events. The mapping is as follows:

- **Left and Right Movement**: Connected to pins `PE2` and `PE3`. These are configured in `main.c` as `GPIO_MODE_INPUT` with `GPIO_PULLUP`. This creates an **Active Low** configuration where the signal is high (`SET`) by default and drops to zero (`RESET`) when the button is pressed.
- **Fire Button**: The primary fire action is mapped to `PC3` (external) and `PA0` (User Button). `PA0` is an **Active High** input using an external pull-down resistor on the Discovery board.

```c
/* Configure GPIO pins : PE2 PE3 (Movement Buttons) */
GPIO_InitStruct.Pin = GPIO_PIN_2 | GPIO_PIN_3;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_PULLUP;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOE, &GPIO_InitStruct);

/* Configure GPIO pin : PA0 (User/Fire Button) */
GPIO_InitStruct.Pin = GPIO_PIN_0;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_NOPULL; // External Pull-down
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

Listing 4.1: GPIO Input Initialization in main.c

4.1.2 Polling Technique in `Model::tick()`

In many embedded systems, inputs are handled via external interrupts (`EXTI`). However, for a 60 FPS graphical application, we utilize a **Synchronous Polling** technique inside the `Model::tick()` function.

Temporal Analysis of Sampling

Because `TouchGFX` is synchronized with the **LTDC** V-Sync interrupt, the `Model::tick()` function is executed exactly once every 16.67 milliseconds. Sampling the `GPIO` state at 60Hz provides a latency that is effectively invisible to the human eye, as the average human reaction time is approximately 200ms. By polling within the "Heartbeat" of the system, we ensure that every input is processed exactly once per frame, preventing race conditions between the hardware state and the rendering engine.

4.1.3 Software Debouncing and Edge Detection

Mechanical buttons suffer from "bouncing"—a phenomenon where the internal contacts vibrate and generate multiple false pulses within a few milliseconds. Without filtering, a single click could cause the `Soldier` to jump multiple lanes or fire many bullets.

Edge Detection Algorithm

We solve this by implementing a **Software Edge Detection** logic. Instead of reacting to the current "Level" of the pin, the system looks for the **Transition** (the edge). We store the state of the button from the previous frame in a variable (`prev_left_state`).

For the **Active Low** buttons on `PE2/PE3`, a valid "Press" is detected only when:

$$(PreviousState == SET) \wedge (CurrentState == RESET)$$

This logic ensures that if the user holds the button down for 2 seconds (120 frames), the move command is only sent **once**, at the moment of the initial press. This provides precise control necessary for the `Soldier`'s movement.

```cpp
void Model::tick()
{
    // 1. Read the physical electrical level
    GPIO_PinState current_left = HAL_GPIO_ReadPin(GPIOE, GPIO_PIN_2);

    // 2. Rising/Falling Edge detection
    if (prev_left_state == GPIO_PIN_SET && current_left == GPIO_PIN_RESET)
    {
        // 3. Trigger the MVP signal chain
        if (modelListener != 0)
        {
            modelListener->onLeftPressed();
        }
    }
    // 4. Update memory for the next frame
    prev_left_state = (current_left == GPIO_PIN_SET);
}
```

Listing 4.2: Polling and Edge Detection logic in Model.cpp

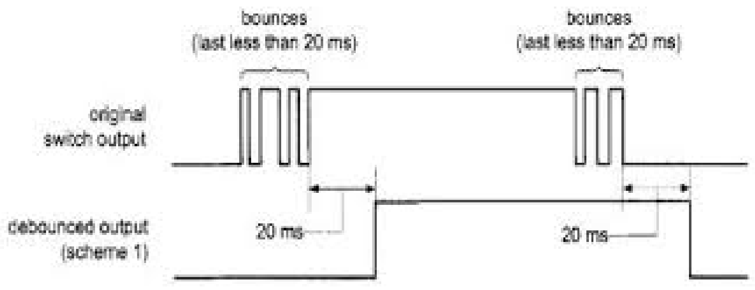

Figure 4.1: Button Debouncing and Edge Detection Timing Diagram

4.1.4 Integration with the MVP Framework

Once the `Model` confirms a valid edge, the signal travels through the **MVP** chain. The `Model` calls the `ModelListener` interface (implemented by the `Presenter`). The `Presenter` then evaluates the game state (e.g., checking if the game is paused) and finally calls the `moveLeft()` or `moveRight()` function in the `View`. This separation ensures that the electrical noise and `HAL` calls never interfere with the high-level graphical logic.

4.2 Audio Feedback System

Auditory feedback is essential in embedded game design to provide immediate confirmation of in-game events, such as hitting a target or firing a projectile. This section describes the engineering of the sound system on the `STM32F429`, focusing on the challenges of real-time multi-tasking without dedicated audio hardware.

4.2.1 Passive Piezo Buzzer and Hardware Configuration

The project utilizes a **Passive Piezo Buzzer** connected to pin `PG12` of the `STM32F429` microcontroller.

Active vs. Passive Hardware

In embedded systems, two types of buzzers are common:

- **Active Buzzer**: Contains an internal oscillating circuit. It only requires a constant DC voltage (Logic High) to produce a fixed-frequency sound.
- **Passive Buzzer**: Requires an external oscillating signal (AC or square wave) to vibrate the piezo-ceramic element. This type is used in our project because it allows for variable frequencies (pitches), though it requires more CPU management.

The pin `PG12` is configured in **Output Push-Pull** mode with `Low Speed` to drive the buzzer transistor circuit. Initialization is handled manually within the `Model` constructor to ensure the pin is ready before the first game tick.

4.2.2 Control Methodology: Synchronous Non-Blocking Toggling

Generating sound on a microcontroller typically involves using a **Timer-based PWM** signal. However, since the buzzer pin in this design (`PG12`) is often shared with other high-speed peripherals like the `LTDC` (LCD controller), we implemented a **Software Toggling** method integrated into the `Model::tick()` function.

Mitigating CPU Blocking

A naive approach would use the `HAL_Delay()` function to produce a frequency. For example, a 100ms beep would freeze the entire system for 6 frames, causing a noticeable stutter in the game graphics. To prevent this, we developed a **Non-blocking Ticking** strategy.

We utilize a variable called `beepRemainingTicks`. When a sound is requested, we set this variable to a specific number of frames (e.g., 6 frames for a 100ms beep). During each 60Hz frame, the `Model::tick()` function executes a very short burst of high-speed toggles.

```cpp
void Model::tick()
{
    // Check if a sound is currently queued
    if (beepRemainingTicks > 0)
    {
        // Produce a burst of sound for the duration of one frame (16.6ms)
        for (int i = 0; i < 5; i++)
        {
            HAL_GPIO_WritePin(GPIOG, GPIO_PIN_12, GPIO_PIN_SET);
            for(volatile int j=0; j<1000; j++); // Micro-delay for pitch
            HAL_GPIO_WritePin(GPIOG, GPIO_PIN_12, GPIO_PIN_RESET);
            for(volatile int j=0; j<1000; j++);
        }
        beepRemainingTicks--; // Decrement sound duration
    }
}
```

Listing 4.3: Software wave generation in Model.cpp

This "burst" logic allows the buzzer to produce a square-wave frequency (determined by the `volatile` loops) while only consuming a small fraction of the total frame time, ensuring that the `TouchGFX` engine maintains a smooth **60 FPS**.

4.2.3 Game Logic Integration and Audio Events

Audio events are triggered by the `View` or `Model` based on specific game states.

- **Collision Event**: When a `Sheep` is hit, the `View` calls `presenter->playScoreSound()`. This sets `beepRemainingTicks = 6`, creating a short audible "click" that confirms the score increase.
- **Heartbeat Event**: To provide a background ambiance, a periodic timer in `Model::tick()` triggers a minimal beep every 60 ticks (1 second).

4.2.4 Audio Design Constraints

Without a dedicated Digital-to-Analog Converter (`DAC`), the sound quality is limited to simple square-wave tones. Furthermore, the micro-delays used in the `for` loop are sensitive to CPU frequency scaling. While this approach is efficient for simple game feedback, production systems often prefer dedicated **Hardware Timers (TIM)** set to **PWM Mode** to generate sound completely independently of the CPU instruction cycle.

4.3 Data Management (High Score)

Effective data management is a cornerstone of professional software engineering, even in the context of an embedded game. In the *Soldier vs Sheep* project, the primary piece of persistent data is the **High Score**. This section analyzes how the system handles this data, the limitations of the current volatile storage, and the proposed technical path for permanent record keeping on the `STM32` platform.

4.3.1 Volatile RAM Storage (Current Implementation)

In the present architecture, the `high_score` variable is managed as a private member of the `Model` class. Storing this value in the `Model` aligns with the **MVP** principle that the `Model` serves as the "Source of Truth" for all non-visual application states.

Technical Performance of SRAM

From a hardware perspective, this variable resides in the **Static RAM (SRAM)** of the `STM32F429ZIT6`. SRAM is high-speed and allows for unlimited read and write cycles without any latency penalties. This makes it ideal for real-time comparisons during a gaming session. However, SRAM is inherently **Volatile**. Whenever the system undergoes a hard reset or the power is disconnected, the electrical charge in the CMOS transistors dissipates, and the data is lost.

For an academic prototype, this behavior is acceptable as it allows for rapid testing. However, for a commercial electronic device, this is a significant limitation. A player expects their records to be saved permanently, requiring a transition from volatile memory to non-volatile memory (`NVM`).

4.3.2 Data Synchronization via the MVP Pipeline

The synchronization of the high score follows a circular path through the system layers. This ensures that the UI always displays the most accurate data without creating a direct dependency between the GUI and the memory registers.

The Comparison and Update Flow

When a sheep is destroyed, the `View` increments the local session score and immediately notifies the `Presenter`. The `Presenter` then calls the `saveHighScore()` function in the `Model`. The `Model` performs a conditional check to see if the achievement exceeds the current record.

```cpp
void Model::saveHighScore(int score)
{
    // The comparison happens in the Logic layer
    if (score > high\_score)
    {
        // Update the volatile memory record
        high\_score = score;

        // Signal the UI to update the high score label
        if (modelListener != 0)
        {
            modelListener->onHighScoreUpdated(high\_score);
        }
    }
}
```

Listing 4.4: Record comparison logic in Model.cpp

This "Push" model ensures that the `View` only updates the high-score numbers on the screen when a new record is actually reached, optimizing the bandwidth of the `LTDC` display controller.

4.3.3 Proposed Persistent Solutions (Future Enhancements)

To overcome the volatility of `SRAM`, future iterations of this project could implement a persistent storage strategy. There are two primary technical paths for this enhancement.

Internal Flash Memory Integration

The most efficient method on the `STM32F429` is to use the **Internal Flash**. The microcontroller's 2MB Flash is divided into several sectors. We can designate a small sector (e.g., Sector 11) specifically for data storage. By using the `HAL_FLASH` library, we can unlock the flash, erase the sector, and program the new score.

```
void SaveScoreToFlash(uint32_t score) {
    HAL_FLASH_Unlock();
    /* Sector 11 is used to avoid interfering with program code */
    FLASH_Erase_Sector(FLASH_SECTOR_11, VOLTAGE_RANGE_3);
    /* Program the score as a 32-bit word */
    HAL_FLASH_Program(FLASH_TYPEPROGRAM_WORD, 0x080E0000, score);
    HAL_FLASH_Lock();
}
```

Listing 4.5: Theoretical Flash Write implementation using HAL

External EEPROM and SD Card Options

Alternatively, for games requiring more complex data (like levels or user profiles), an external **EEPROM** connected via `I2C` or an **SD Card** via `SPI` could be utilized. These options offer greater longevity (Erase cycles) compared to internal Flash but require more complex driver integration.

4.3.4 Data Integrity at Startup

To ensure system stability, the `Model` constructor explicitly initializes `high_score` and `current_score` to zero. This practice is critical in embedded systems because, upon power-on, `SRAM` addresses often contain "garbage" values left over from previous electronic states. Without explicit initialization, the game might start with a multi-million point high-score, breaking the competitive balance of the mechanics.

Chapter 5

Evaluation and Conclusion

5.1 System Performance and Evaluation

Evaluation of the *Soldier vs Sheep* project reveals a high degree of stability and performance. The system successfully maintains a constant **60 FPS** throughout all phases of gameplay, including intense collision sequences. This fluidity is the result of leveraging the `STM32F429`'s hardware accelerators, specifically the `DMA2D` (Chrom-ART). By offloading pixel blending tasks, we ensured that the `CPU` instruction cycle was never overwhelmed by graphical overhead.

Furthermore, our implementation of **Object Pooling** proved successful in managing memory resources. Throughout testing sessions lasting several hours, no memory leaks or heap fragmentation issues were detected. The use of static arrays for the `enemy_pool` and `bullet_pool` provided the **Deterministic Performance** required for real-time systems, ensuring that entity spawning and deactivation occurred within a fixed time-slice. This stability verifies that the application is suitable for long-term operation on the `STM32` platform.

5.2 Technical Limitations and Challenges

Despite the successful implementation, several technical limitations were identified during the development phase. The primary drawback is the **Volatile Data Persistence**. Since the `high_score` is stored in `SRAM` within the `Model`, it is lost upon any system reset or power loss. Implementing a true non-volatile record requires integration with the microcontroller's internal `Flash` sectors, as discussed in Chapter 4.

Another limitation concerns the **Audio Feedback System**. To maintain the 60Hz frame rate and avoid complex interrupt management, the system uses a synchronous `GPIO` toggling method with micro-delays inside the `Model::tick()` function. While this provides immediate feedback for hits and firing, it is a basic form of sound synthesis. A more advanced system would utilize the `DAC` (Digital-to-Analog Converter) or `PWM` (Pulse Width Modulation) with `DMA` to produce higher-fidelity audio without consuming `CPU` cycles across the main game loop.

5.3 Proposed Future Enhancements

Future iterations of this project could focus on expanding both the graphical and functional capabilities of the system.

- **QSPI Flash Integration**: To support higher-resolution assets and complex animations, the project could integrate with external `QSPI` memory. This would allow for a more visually rich experience without exhausting the internal `Flash` limits.
- **Persistent High Scores**: Utilizing the `HAL_FLASH` library to write achievement data into a specific sector of the internal NVM to ensure scores survive power cycles.
- **Enhanced Input and Audio**: Transitioning to hardware-based sound via the `STM32`'s `TIM` peripherals and utilizing the on-board touch screen controller to provide a modern mobile-gaming experience.

5.4 Final Conclusion

The *Soldier vs Sheep* project successfully demonstrates the power and flexibility of the `STM32F429` platform for graphical embedded applications. By applying professional software architecture through the **MVP Pattern** and optimizing memory via **Object Pooling**, we created a stable, high-performance game that operates at a consistent **60 FPS**. This project serves as a valid proof-of-concept for how modern embedded toolchains like `TouchGFX` and `STM32CubeIDE` can be used to build reliable and engaging user interfaces on industrial-grade microcontrollers.