ESLab Final Project
Interactive Dino Run

Yi-Kai, Li
department of Electrical Engineering.
National Taiwan University
Taipei, Taiwan
frxdyrz0718@gmail.com

Chun-Wei Chen
department of Electrical Engineering.
National Taiwan University
Taipei, Taiwan
wilson20020215second@gmail.com

Po-Wen, Huang
department of Electrical Engineering.
National Taiwan University
Taipei, Taiwan
kojilab13pi@gmail.com

ppt: https://docs.google.com/presentation/d/1DzRe66WZ8N1aPMDnpDfObGuNnjXvqBapqZuprVaNa3g/edit?usp=sharing

github: danielliyk/Embedded-System-Labs: This is for Embedded System Labs course of NTUEE 113-fall

Abstract

In this project, we attempt to combine modern game development with embedded system programming to create a unique gaming experience. Using the SDL2 library in C++, we'll design the game with smooth graphics, responsive controls, and engaging mechanics.

On the hardware side, we'll program the B-L475E-IOT01A development board with the STM32CubeIDE. The board will serve as the game controller, integrating sensors and buttons to enable interactive and immersive controls. This project bridges software and hardware, exploring innovative game mechanics like tilt navigation and sensor-based inputs for a fun and intuitive experience.

I. Motivation

When the internet went down, the Google dinosaur game was our go-to distraction. It's simple, addictive gameplay made it a fun way to pass the time, but over time, it began to feel repetitive. Dodging cacti and jumping over birds lost its charm after countless rounds.

This inspired us to create our own custom version of the game. We kept the core elements we loved — simplicity and endless running—while adding new challenges, obstacles, and customizable features to make it more engaging. It became a creative project where we learned game design and programming, resulting in a game that's not just nostalgic but uniquely ours, offering endless fun without ever feeling dull.

II. Game Introduction

This section provides an overview of our game, including the game rules, the visual design and appearance of the game, and the operational mechanics for interacting with and playing the game.

2.1 game rule

Initially, the game consists of five lanes, with the player starting with three lives and a score of zero. The primary objective is to increase the score by collecting sheep displayed on the screen. However, if the player's dinosaur collides with an obstacle, such as a tree or an arrow, one life is lost. Each time the player collects three sheep, a fire icon will appear in the bottom-left corner of the screen. Players can press a button to activate the fire ability, which destroys trees and provides a strategic advantage. Additionally, players have the option to trade three points from their score to regain one life. The game ends when the player's lives are reduced

to zero.

2.2 game operation

To control the dinosaur's movement up and down, players can tilt the STM32 board upward or downward. Additionally, the STM32 user button provides more functionality for gameplay. If the button is pressed for less than one second, and a fire icon is present in the bottom-left corner of the screen, the dinosaur emits fire to destroy tree obstacles. If the button is pressed for more than one second, and the player's scores are more than three, they can trade three points from their score to regain one life.

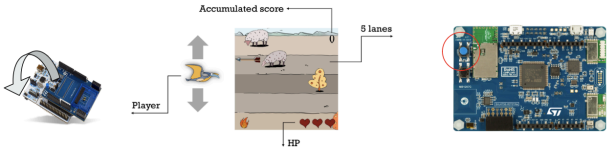

fig.1 game control

2.3 game art

In the diagram, the interface is divided into three sections: the left section represents the starting window, the middle section depicts the game processing window, and the right section illustrates the game ending window. Additional figures below the main interface showcase various game elements, including the dinosaur animation, fire animation, sheep, obstacles (trees or arrows), and the player's remaining lives.

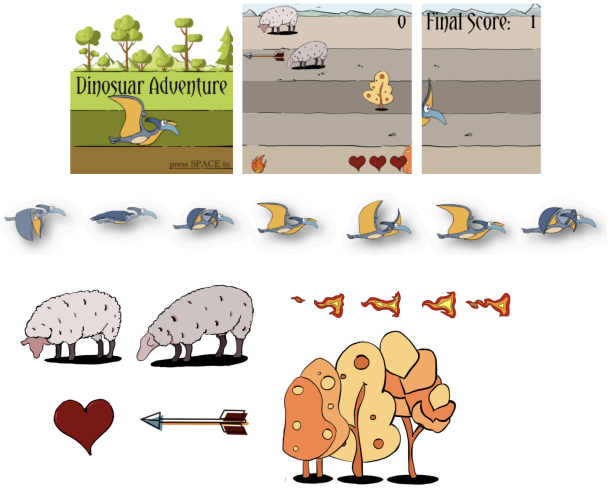

fig.2 game art collection

III. SDL2 Game Structure

Adopted libraries are listed and briefly explained as follows:

(1) **SDL2**: Create the game window and basic game functions like moving up and down (SDL_Event), scrolling background, collision events, etc.
(2) **SDL2_image**: Load required images and create animations, such as background images, obstacles, main character, and assigning them to the window.
(3) **SDL2_ttf**: Load the text and font package, displaying them on the window.
(4) **SDL2_mixer**: Mainly used for adding the background music and sound effects.

The following are some worth-noting classes:

(1) **AnimatedSprite**: This class handles objects requiring animation, rendering them to the window. We use the rendercopy function to display sprite sheet animations frame by frame, creating a smooth animation for the dinosaur with seven frames.

```
class AnimatedSprite{
    public:
        AnimatedSprite(SDL_Renderer*& renderer, std::string filepath);
        ~AnimatedSprite();
        void Draw(int x, int y, int w, int h);
        void PlayFrame(int x, int y,int w, int h, int frame);
        void Update();
        void Render(SDL_Renderer*& renderer);
        SDL_Rect getRect() { return m_dst; }

    private:
        SDL_Rect m_src;
        SDL_Rect m_dst;
        SDL_Texture* m_texture;
};
```

fig.3 Animated Sprite Class

```
void AnimatedSprite::Draw(int x, int y, int w, int h){
    // Create a rectangle
    m_dst.x = x;
    m_dst.y = y;
    m_dst.w = w;
    m_dst.h = h;
}

void AnimatedSprite::PlayFrame(int x,int y, int w, int h, int frame){
    m_src.x = x+w*frame;
    m_src.y = y;
    m_src.w = w;
    m_src.h = h;
}

void AnimatedSprite::Update(){

}

void AnimatedSprite::Render(SDL_Renderer*& renderer){
    SDL_RenderCopy(renderer,m_texture,&m_src,&m_dst);
}
```

fig.4 member functions

```
static int frameNumber =0;
AnimatedSprite animatedSprite(gRenderer,"/Users/pio/fi
animatedSprite.Draw(250,200,400,400);
animatedSprite.PlayFrame(-40,-30,300,250,frameNumber);
animatedSprite.Render(gRenderer);
frameNumber++;
if(frameNumber>6){
    frameNumber= 0;
}
```

fig.5 execution code block

(2) **LTexture**: Used for loading and initializing object textures, including color, rendering, etc. All objects requiring textures are declared globally.

Important helper functions in main.cpp:

(1) **Init():** Initializes the game, including all variables and objects, creates the window, and loads initial screen textures.
(2) **loadMedia():** Loads various game assets, including images, character animations, sound effects, strings, and fonts.
(3) **close():** Handles post-program tasks, such as resetting variables and freeing memory.
(4) **checkCollision():** Key function for detecting collisions. It assigns a rectangle to objects for collision detection and continues with the next command, such as scoring or losing health.
(5) **command():** Contains the main game loop for more convenient maintenance of our game logics, including executing key commands, looping the game background and character animations.

Here we elaborate how we deal with in-game objects' interactive responses:

To handle object scrolling We decrease the scrolling offset over time, representing the decrementing x coordinate of the object. When detecting the object has scrolled the length of the map, it scrolls again. Individual objects are assigned with their unique scrolling offsets, therefore, adjusting them allows us to achieve relative speed. For instance, relative to the scrolling background, the arrow's scrolling offset is designed to be subtracted by twice the speed (distance/frame). On the other hand, to cope with random obstacle generation, we need to generate random object positions. Yet to prevent objects from disappearing when scrolling to the left edge, we add the initial x-coordinate and object width from the last loop, ensuring objects complement each other and avoid leaving only the background visible, that is, the scene will never be empty.

```
//scroll tree
scrollingOffsetObj1 -= speed;
if ( scrollingOffsetObj1 < -gBGTexture.getWidth()-random_start_tree-140) {
    scrollingOffsetObj1 = 0;
    random_pos_tree = ((rand()%5 * 120)/10)*10;
    random_start_tree = ((rand()%6 * 140)/10)*10;
}
```

fig.6 execution code block

For the vital collision detection, where the character's life shall decrease upon colliding with arrows or trees, we detect and compare the horizontal or vertical lines of each object's edges, mimicking a bounding box. Parameters are tuned to present collisions ideally.

```
bool checkCollision( SDL_Rect a, SDL_Rect b )
{
    //The sides of the rectangles
    int leftB;
    int rightA, rightB;
    int topA, topB;
    int bottomA, bottomB;

    //Calculate the sides of rect A
    rightA = a.x + 200;
    topA = a.y + 50;
    bottomA = a.y + 100;

    //Calculate the sides of rect B
    leftB = b.x;
    rightB = b.x + b.w;
    topB = b.y;
    bottomB = b.y + b.h;
```

```
    if( topA > bottomB)
        return false;
    if( bottomA < topB)
        return false;
    if( rightA < leftB || leftB < 0)
    {
        return false;
    }

    //If none of the sides from A are outside B
    return true;
}
```

fig.7 collision checking

As for the character movements' constraint, since the track spacing is approximately 120. When the dinosaur moves, its Y coordinate changes by 120. We set boundary conditions to ensure the dinosaur does not exceed the window.

```
if( e.type == SDL_KEYDOWN && e.key.repeat == 0 )
{
    //Adjust the velocity
    switch( e.key.keysym.sym )
    {
        case SDLK_UP: {
            posy -= 120;
            if( ( posy < 0 ))
            {
                //Move back
                posy += 120;
            }

            break;}
        case SDLK_DOWN: {
            posy += 120;
            if(( posy > 480 ) )
            {
                //Move back
                posy -= 120;
            }

            break;}
    }
}
```

fig.8 character movement constraint

IV. Proposed Techniques & Data Flow

The STM32 board acts as the GATT server, while the computer serves as the GATT client. To ensure more immediate feedback, we adjusted the sampling rate of the BSP accelerometer sensor.

Regarding data processing, we resolved many user gesture recognition issues through trial and error. Techniques such as applying a weighted moving average and handling consecutive data points exceeding threshold values were employed to improve accuracy.

Fig.9 shows the data flow between the board and computer. Initially, we used Python's Bleak module to scan for Bluetooth devices and successfully established a BLE connection between the STM32 development board and the laptop. Once connected, the development board begins writing accelerometer data into a buffer on board. This buffer is necessary because our following data processing spans several small sample points.

To achieve edge computing, we process the data on the development board first. After recognizing the specific action, we write the result into the ACC characteristic. Meanwhile, the user can trigger an external interrupt by either a long press or short press of the user button. The recognized action from the button press is also written into the ACC characteristic.

Each action is treated as a state: 0 represents IDLE, and so on. On the computer side, once the development board's ACC characteristic notifies the current state, it triggers the corresponding action of the dinosaur in the SDL2 game.

Here, edge computing demonstrates its full potential and merits. Since raw data is being processed on the edge device, there is no concern in packet loss or precision issues due to wireless transmission. The sent data is merely integer values ranging from 0 to 4. Also, considering that implementing Bluetooth connections in Python is more convenient while the SDL2 game is written in C++, we use shared memory to allow both programs to access the state concurrently. While shared memory continuously updates with the latest received state, both executing program loops keep reading the stored value. Through our experiment, this approach significantly improves the delays in contrast to writing and reading to and from a text file.

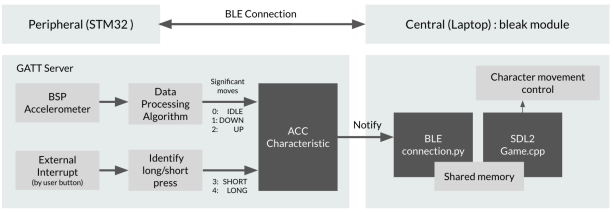

fig.9 data flow

V. Methodology

The system integrates several key components for efficient real-time processing of accelerometer data, specifically for movement detection and communication via Bluetooth. This includes the use of buffers, a weighted moving average algorithm for noise reduction, threshold detection logic for movement classification, and Bluetooth data transmission for interaction with external systems. The following section integrates the mathematical models and the system design in a comprehensive manner.

5.1. Global Variables and Buffers

To efficiently handle the accelerometer data, the system utilizes buffers for storing acceleration values along the x, y, and z axes. These buffers operate as circular arrays, with a fixed size determined by the BUFFER_SIZE constant. The circular nature of the buffers ensures that new data overwrites the oldest data once the buffer reaches its maximum capacity. The buffer_index variable tracks the current position for writing new data, ensuring that data is stored in the correct location. When the buffer index reaches the end of the buffer, it wraps around to the beginning, maintaining continuous operation without memory overflow.

The system also uses fags such as threshold _exceeded and last_exceeded_threshold. These flags are essential for managing state transitions related to movement detection. The threshold _exceeded flag indicates whether a significant movement has occurred, while the last_exceeded _threshold flag records the direction of the exceedance (positive or negative). Together, these flags ensure precise tracking of the movement event, preventing false positives or premature state resets.

5.2. Weighted Moving Average Implementation

The core of the noise reduction technique in this system is the weighted moving average (WMA), which smooths the y-axis acceleration data to minimize the impact of minor, unintended board movements. The WMA is computed using a window size of 2, with higher weight assigned to the most recent value. The mathematical formula for the weighted moving average is given as:

$$WMA_n = \frac{2 \cdot y_n + y_{n-1}}{3}$$

Here, y_n represents the most recent y-axis value, and y_{n-1} represents the previous value. The weight of 2 is assigned to the most recent value, while the previous value is given a weight of 1. The sum of the weights (3) normalizes the result. This weighted approach ensures that the most recent data point has a greater influence on the smoothed result, while still accounting for the previous value to reduce volatility in the data. If insufficient data points are available, such as when fewer than two samples are present, the function returns the most recent value, ensuring robustness even in early or sparse sampling scenarios.

5.3. Threshold Detection and Reset Logic

Threshold detection is a critical component for detecting significant movements, such as lifting or tilting the board. The threshold detection mechanism compares the smoothed y-axis acceleration value against predefined thresholds: the POSITIVE_THRESHOLD and NEGATIVE _THRESHOLD. If the smoothed value exceeds the positive threshold, it indicates an upward movement; if it falls below the negative threshold, it indicates a downward movement. The system maintains the state of movement using the threshold_exceeded flag, and the direction of the threshold exceedance is stored in last_exceeded _threshold.

Mathematically, this logic can be expressed as follows:

I. $WMA_n >$ POSITIVE_THRESHOLD
set threshold_exceeded = 1, and the direction to "up."

II. $WMA_n <$ NEGATIVE_THRESHOLD
set threshold_exceeded = 1 and the direction to

"down."
Once a threshold has been exceeded, the system enters a state where it continuously monitors recent acceleration data within a sub-window, which is defined by the constant SUB_WINDOW_SIZE. The purpose of the sub-window is to avoid premature resets of the threshold exceedance state due to transient fluctuations in the data. If all values within the sub-window return to normal (below the positive threshold or above the negative threshold), the system resets the exceedance state:

I. If yi ≤ POSITIVE_THRESHOLD for all yi in the sub-window, we reset threshold_exceeded

II. If yi ≥ NEGATIVE_THRESHOLD for all yi in the sub-window, we reset threshold_exceeded

5.4. Bluetooth Data Transmission

The system communicates movement events through Bluetooth by using the aci_gatt_update_char_value function. This function sends data to an external Bluetooth device, such as a smartphone or tablet, enabling real-time interaction with the system. The transmitted data consists of a single byte representing the current state of the system. The possible states are:

1. Idle: 0
2. Upward movement: 1
3. Downward movement: 2
4. Short duration user button press: 3
5. Long duration user button press: 4

The Bluetooth communication ensures seamless integration with external devices, making it possible to trigger appropriate actions based on the detected movements.

5.5. Accelerometer Data Handling

The function ACC_DATA_Update is responsible for managing the flow of accelerometer data. It retrieves the current x, y, and z axis values using the BSP_ACCELERO_AccGetXYZ function and stores these values in the respective buffers. The buffer_index is incremented in a circular manner to ensure continuous data storage without data overflow.

After storing the new data, the system computes the weighted moving average for the y-axis values using the weighted_moving_average function. This smoothed value is then passed to the check_thresholds_and_send function, which checks whether a significant movement has occurred and triggers the appropriate Bluetooth notification. The buffer index is updated to ensure that new data overwrites the oldest data once the buffer is full.

5.6. Code Design and Benefits

This implementation is designed to efficiently handle real-time sensor data in an embedded system. The combination of noise reduction via the weighted moving average and stateful threshold detection allows for accurate and reliable movement detection, even in the presence of minor fluctuations. The use of circular buffers ensures efficient memory utilization, while the Bluetooth integration enables real-time communication with external devices. This design is both robust and extensible, making it suitable for a wide range of applications that require real-time processing of sensor data and interaction with external systems. The modularity of the system allows for easy adaptation to different sensor types, movement thresholds, and communication protocols.

5.7. User button settings

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{

    if (GPIO_Pin == BUTTON_EXTI13_Pin){
        //printf("hello \n");
        if (HAL_GPIO_ReadPin(BUTTON_EXTI13_GPIO_Port, BUTTON_EXTI13_Pin) == 0){
            buttonPressTime = xTaskGetTickCount();
            buttonState = 1;
            //printf("hello \n");

        }
        else {

           //printf("enter\n");
           if (buttonState == 1) {
               uint32_t pressDuration = xTaskGetTickCount() - buttonPressTime;
               buttonState = 0;

               if( pressDuration >= pdMS_TO_TICKS(1000) ){
                  button_pressed = 2;

               }
               else{
                  button_pressed = 1;
               }
           }
        }
    }
}
```

fig.10 User button settings code

When the GPIO pin is triggered (i.e., when the button is pressed or released), the interrupt callback function HAL_GPIO_EXTI_Callback is invoked. The logic proceeds as follows:

Button Press Detection:

I. If the button is pressed (i.e., the GPIO pin reads low, indicating a button press), the function records the current system tick count (buttonPressTime) using xTaskGetTickCount(). This marks the time at which the button was first pressed, and sets the buttonState variable to 1 to indicate that the button is in a pressed state.

Button Release Handling:

I. If the button is released (i.e., the GPIO pin reads high), the function checks if the button was previously pressed (buttonState == 1).

It calculates the duration of the button press by subtracting buttonPressTime from the current system tick count (xTaskGetTickCount()), yielding the press duration in ticks. Based on the duration of the press, the system determines whether the press was short or long:

I. If the press duration is greater than or equal to 1000 milliseconds (converted to ticks using pdMS_TO_TICKS(1000)), it categorizes the press as a long press, and sets button_pressed to 2.
II. If the press duration is shorter than 1000 milliseconds, it categorizes the press as a short press, setting button_pressed to 1.

This approach allows the system to differentiate between short and long button presses, enabling different actions or events based on the press duration. It provides an efficient mechanism for handling user input in embedded systems using GPIO interrupts.
Note that The GPIO pins defined in the original BLE project might interfere with the operation when using the HAL_GPIO_EXTI_Callback function. Therefore, we commented out all the related definitions of the predefined GPIO pins to avoid conflicts.

VI. Processing Experimental Results

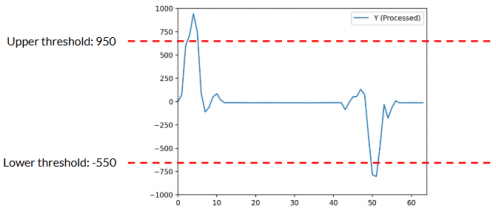

fig. Received y-axis acceleration data

The results indicate that flipping the board leads to a significant change in the received y-axis acceleration data, resembling a peak or trough in the waveform. By detecting this fluctuation pattern, we can determine the orientation of the board and send corresponding "up" or "down" commands to the main program, thereby enabling character movement.

VII. Work Distribution

STM32CubeIDE:
Data process: Yi-Kai, Li
Bluetooth connection: Po-Wen, Huang
Button press detection: Chun-Wei Chen

SDL2 package :
Game Logic: Yi-Kai, Li、Po-Wen, Huang
Scrolling Background & Text & Memory: Chun-Wei Chen
Sound Effects & Collision Detection: Yi-Kai, Li
Character Animation & Art: Po-Wen, Huang、Chun-Wei Chen

Report:
PPT: Yi-Kai, Li、Po-Wen, Huang
Demo Video: Chun-Wei Chen

VIII. References

https://lazyfoo.net/tutorials/SDL/01_hello_SDL/index.php

SDL2/Tutorials - SDL Wiki