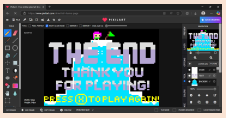
Group S3-14
Members: Lee Hao Zhe *A0272689J*, Isaac Saw Sheng Jie *A0272400W*, Neeraj Kumbar *A0271737X*, Mark Neo Qi Hao *A0281484X*

PERSONAL AND TEAM IMPROVEMENTS		
Student and Improvement Name	Improvement Description	Images/Photos
Team "Celeste"	<u>Our Game:</u> Our project is modelled after the game "Celeste", a 2D platformer game available on Steam. Our rendition features 4 playable levels, in which the user carefully navigates using the movement controls on the Xbox controller. The aesthetic look of the game aims to give users a smooth and immersive gaming experience. The choice of control was an Xbox controller, to make the game feel similar to the original. <u>Quick Start</u> • Power the Basys 3 board and connect the OLED. • Power on the ESP32 (preloaded with program to connect and read data from controller). • Pair your Xbox controller wirelessly. • Press Y (Start) to begin the game. <u>Controls</u> • **A** – Jump • **X** – Dash (8 directions via analog stick) • **Left Analog Joystick**– Move • **Y** – Start/Reset <u>Gameplay</u> • **Dash:** Usable once in mid-air; resets when grounded. LEDs show dash. • **Death:** Touching hazards triggers an explosion and respawn. • **Blink State:** Brief invulnerability after respawn. • **Level Completion:** Reach the far/right/top end of the screen. <u>💀 Death Counter</u> A 4-digit 7-segment display tracks how many times the player has died. The counter resets when entering Level 0 (Tutorial), letting players practice without penalty. <u>Developer Mode (SW[15] ON)</u> • SW[0–4]: Jump to Level 0–3 or End Screen • SW[14]: Return to Menu **Use only one switch at a time for smooth transitions.**	 *Full Celeste Gaming Setup*
Student A: Isaac "UI/UX Designer + Collision Detection"	Implemented core gameplay logic with 3 distinct functions (based on the player's current x,y position), • is_dead: check if player is **touching spikes** or falling into the **void** (out of screen) • is_level_done: determines **level completion** state of the player • is_obstructed: evaluates **player collision** in 8 cardinal/diagonal directions using an **8-bit flag** system, supporting detection in all four level maps. Functions are made to be **modular and reusable** for ease of implementation and debugging. **Designed pixel art** assets for static backgrounds and player sprites using *PixilArt*. **Optimized hardware resource usage** by converting PNG artwork into `.BMP` format and integrating them as source memory blocks during FPGA bitstream generation. This custom asset pipeline significantly **reduced LUT** (Look-Up Table) usage by utilising the Block Random Access Memory (BRAM) on the Basys3 board. This enhanced memory efficiency and made our project very scalable.	 *Example of collision detection*  *Usage of pixilart.com illustrator*
Student B: Hao Zhe "Frontend Engineer (Aesthetics)"	<u>Memory File Generation</u> **.BMP to .MEM Python Script:** The Python script **converts 96×64 BMP images** into a **.MEM file** suitable for driving the 16-bit color (RGB565) OLED display on the Basys3. <u>Sprite Animation</u> The sprite FSM is designed with three key states: **Normal**, where the sprite responds to player inputs and navigates the game world; **Explosion**, where the sprite displays an expanding explosion effect upon death; and **Blink**, where the sprite temporarily blinks, indicating either a **respawn after death** or **signaling a level transition.** **Level Transition & Sprite Respawn:** Both level transitions and sprite respawns share a similar sequence. First, movement is **briefly frozen**—either to allow the background to switch (level transition) or to display an explosion animation (sprite death). Following this freeze, the sprite is **reset to a predefined start position** appropriate for the next scenario. To convey a **clear visual cue**, the sprite alternates between **visible** and **invisible states**, signaling temporary invulnerability and granting the player a moment to orient. Once this short blink timer expires, the sprite resumes its normal operating state, restoring full player control and collision detection. **Sprite Death:** When a death condition is detected, the sprite switches from the **Normal** state into the **Explosion** state. In this state, an **explosion animation** is played by incrementally expanding a circle around the sprite's center and using a **pseudo-random** mix of red and white colors to simulate an explosion effect. After all explosion frames have been displayed, the FSM transitions into the **Blink** state, during which the sprite temporarily blinks before finally resetting to the starting position for a respawn. <u>Game Aesthetics</u> The overarching **Game FSM** controls transitions between major game states, such as the **main menu**, individual levels, and the **end screen**. It also manages special events like **fade-in/out effects** during transitions. The FSM tracks both the current and target background states, progressing through a structured sequence of **"animate"**, **"fade"**, and **"wait"** sub-states whenever a transition is triggered—either through	 *Raining Animation*  *Death/Explosion Animation (At the spikes)*

level completion, **developer switch input**, or reaching the game's conclusion.

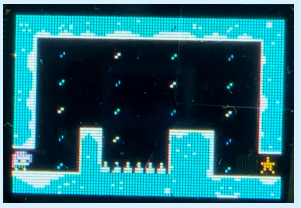
Game Levels Transition & Animation: To transition between levels, the FSM uses a **fade factor** that linearly interpolates between the source and target images in RGB565 format. This creates a smooth visual effect, blending the old background with the new one. The FSM cycles through states to increment the fade factor until the transition is complete, then briefly waits to prevent accidental rapid skips. If developer mode is active, users can also manually switch levels at will, with the same fade animation ensuring consistency.

Rain Animation: A procedural rain effect adds additional movement and depth. For each pixel coordinate, the logic calculates whether it should display a raindrop. Drops are positioned by offsetting the pixel's coordinate with a **running counter**, and some are drawn with a small "splash" pattern. By mixing different color values (blue, cyan and white) and cycling the offset each frame, the design simulates falling rain.

Developer Mode
User can **enter Developer Mode** by switching **SW[15] ON**. This switch must remain ON throughout to access any developer features. The following **one-hot encoded switches** allow direct level transitions:

- **SW[0]** – Jump to **Level 0 (Tutorial)**
- **SW[1]** – Jump to **Level 1**
- **SW[2]** – Jump to **Level 2**
- **SW[3]** – Jump to **Level 3**
- **SW[4]** – Jump to **End Screen**
- **SW[14]** – Return to **Home Screen**

These switches can be toggled **at any time during gameplay** to test specific levels instantly.



Sprite Blinking at Spawn Position

Level Transition Animation (Fade In/Out)

Student C: Mark Neo

"Backend Engineer (Movement)"

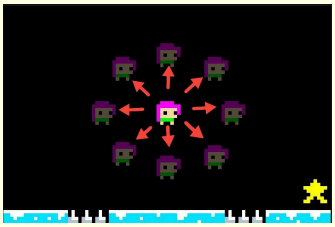
Horizontal Movement
The logic for horizontal movement was implemented by continuously checking the state of the joystick directions (mapped to btnL for left and btnR for right). When a directional input is detected, the code increments or decrements the player's x-coordinate by one pixel per update cycle. The design leverages **conditional checks against collision flags**, ensuring movement only occurs when no obstacle (e.g., walls) is present in that direction.

Vertical Movement and Gravity
Vertical movement is governed by a **gravity mechanism** that accelerates the character downwards over time. The code maintains a **fixed-point representation** of the y-position (using an 8-bit integer combined with a 4-bit fraction) to achieve sub-pixel precision. On each clock cycle where the character isn't on a platform, a constant gravity value is added to the vertical velocity (vy), simulating a natural fall.

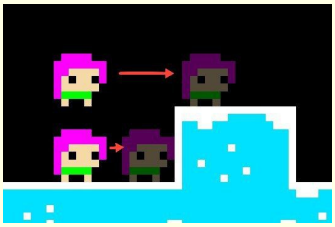
Jumping
The jumping mechanic is activated when the player uses the Xbox controller's A button. The jump is permitted only when the character is on solid ground, determined through collision checking. Once activated, the jump imparts an **initial upward velocity** to the character. The code then **continues to apply gravity on subsequent cycles**, thus producing a natural arc.

Dashing
The dash function, triggered by the Xbox controller's X button. When the dash button is pressed, the code checks the directional inputs (via btnU, btnL, btnD, and btnR) to allow dashes in any of the 8 directions (up, down, left, right, and the four diagonals).
Before executing the dash, the system verifies whether dashing is allowed by checking the **dash_ready** flag. This flag ensures that the player can **only dash once while airborne**. When the character is on a platform, **dash_ready** is reset to allow a new dash. Upon executing a dash, the flag is cleared, ensuring that a dash cannot be performed again until the character lands and **dash_ready** is restored. Once a valid dash is initiated, the code computes per-cycle horizontal (**dash_dx**) and vertical (**dash_dy**) displacements based on a fixed dash distance divided over the dash duration. During the dash, normal physics, such as gravity, are **temporarily bypassed**, with the movement driven solely by these dash parameters. Additionally, even within the dash, **collision detection is maintained** to ensure that the player cannot dash through obstacles; if a collision is detected, the dash is canceled to prevent the character from moving into an obstructed area.

Jumping Mechanism



Dashing Mechanism (8 Directions)



Collision Detection during Dash

Student D: Neeraj

"Backend Engineer (Controller)"

Controlling Celeste via Xbox Controller through UART between ESP32 and FPGA
An Xbox controller is connected wirelessly to an ESP32 running Bluepad32. The ESP32 processes analog stick and button states, encodes them into an 8-bit `buttons` *byte*, and transmits it over UART to the Basys 3 board at a steady rate.

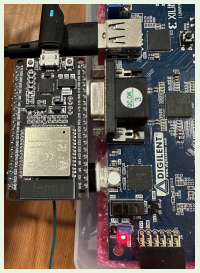
Input Mapping Format:
The controller's input is mapped into specific bits of the buttons byte:
Jump (A on Xbox Controller), Dash (X on Xbox Controller), Left/Right via analog stick X-axis, Up/Down via analog stick Y-axis, Start/Restart (Y on Xbox Controller)

Controller Reception Logic:
The `controller_input` Verilog module receives UART packets, decodes the `buttons` byte, and outputs clean button signals (e.g., `btnL`, `btnR`, `jumpBtn`).

Death Counter
I implemented the death counter, which tracks player deaths during gameplay and resets on returning to the tutorial. It updates in real time using a 100Hz clock and displays the count on the Basys3's 7-segment display, providing feedback and enhancing game challenge.

Dash FSM and Direction Handling
A specialized FSM detects rising edges of the dash button and synchronizes them to a 100Hz clock for consistent animation timing. During a dash, the player's facing direction (left or right) is tracked and used to control a sweep of LEDs (LED[15:0]) that turn OFF in sequence. Once the dash ends, all LEDs are restored to indicate dash availability. A short pulse extender ensures reliable dash activation and direction tracking.

Showcasing Controller, Death Counter and Dash Availability



ESP32 connected to FPGA (UART Connection)

Group S3-14
Members: Lee Hao Zhe *A0272689J*, Isaac Saw Sheng Jie *A0272400W*, Neeraj Kumbar *A0271737X*, Mark Neo Qi Hao *A0281484X*

References:

- Pixilart.com: Used for illustration of all sprites and background images.
- 8-bit Celeste Game: Used as inspiration.
- ESP-IDF Arduino Bluepad32 Template: Utilized for ESP32 integration with Xbox controller over UART.

Course Feedback:

- The practical exam experience could be improved by reducing the long reading period and using that time for coding instead.
- Bitstream generation times vary by PC performance, which disadvantages students with slower machines— consider making all students use the lab PC.
- PE difficulty also differs across lab groups— consider standardizing the PE difficulty, and all students should take a single PE paper at the same time.
- The PE instructions were too condensed with limited examples, making them hard to interpret under pressure— consider providing a sample video or GIF of the expected output would help clarify requirements, especially since the PE is open internet anyway.