

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | |
|  | | SPACE INVADER GAME | | | | |  | |
|  |  | | | | | | |  |
|  | | | |  |  | | | |
|  | | | | BYFantastic 4 (F4)  * **Anas Norani** * **Hanan Majeed** * **Noor Ul Saba** * **Hamna Shah**   **BSDS-02-A** |  | | | |
|  | | | | DECEMBER 08, 2024—FUNDAMENTALS OF COMPUTER PROGRAMMING—PROF. ARHAM MUSLIM |  | | | |
|  | | |  | | |  | | |

#### ****Final and Detailed Recap of the Project Idea****

**Project Idea:**  
Space Invader" is a 2D arcade shooter game where the player controls a spaceship and must destroy a group of alien invaders positioned at the top of the screen when the game starts. The aliens might move in patterns, and the player must avoid their projectiles. Once all the aliens are destroyed, the game progresses to a new wave with increased difficulty.

**Objective:**

* Destroy all alien invaders in the specified wave while avoiding their attacks.
* Survive as long as possible to achieve a high score.
* The game ends when the player runs out of lives.

**Key Features:**

* **Player Controls:** Move the spaceship left, right, and shoot using keyboard inputs.
* **Enemy Waves (Optional):** Aliens might move in patterns, with increasing speed and complexity in later levels.
* **Scoring System:** Earn points for destroying enemies.
* **Level Progression:** Difficulty increases with each level.
* **Game Over:** Displays final score and allows restarting the game.

**Changes made:**

Instead of the enemies descending down to the bottom of screen and ultimately leading to termination of the game, we have considered it best using a fixed enemy grid having pre-defined locations. The reason for this modification is to make the logic more comprehensible for us because including enemy movement prompts to include further modifications in other modules that might be complex and difficult for us to handle.

#### ****2. Abstract Project Architecture:****

The game consists of several modules, each handling a specific aspect of the game's functionality:

**Key Modules and Responsibilities:**

### **1. Rendering Module:**

* **Role**: Handles the visual representation of all game objects and updates the screen each frame.
* **Key Components**:
  + **Spaceship Sprite**: Represents the player's ship graphically. Typically uses a ***sf::Sprite*** loaded with a texture.
  + **Enemy Sprites**: Represents enemy units with distinct designs for variety or difficulty differentiation.
  + **Bullet Shapes**: Simple geometric shapes (e.g., rectangles or circles) or textured sprites for player and enemy bullets.
  + **Background Image**: A static or scrolling background to enhance immersion.
  + **UI Elements**: Displays the score, remaining lives, and other game information (e.g., "Game Over" text).
* **Interaction**:
  + Fetches the current positions and states of all objects from the **Game State Module**.
  + Updates the display using SFML’s rendering pipeline.
  + Uses methods like ***sf::RenderWindow::draw()*** to render each object.

### **2. Player Input Module:**

* **Role**: Captures real-time player actions and translates them into game commands.
* **Key Inputs**:
  + **Arrow Keys**: Move the spaceship (up, down, left, right).
  + **Space Bar**: Fires bullets.
* **Interaction**:
  + Sends movement commands to the **Spaceship Module** to update its position.
  + Triggers bullet generation in the **Spaceship Module** when the firing key is pressed.
* **Implementation**:
  + Uses SFML's event system *(****sf::Event::KeyPressed*** or ***sf::Keyboard)*** for handling key presses.

### **3. Spaceship Module:**

* **Role**: Represents and manages the player's spaceship behavior.
* **Key Functions**:
  + **Movement**: Updates position based on player input, ensuring the spaceship stays within screen boundaries.
  + **Firing Bullets**: Spawns new bullet objects when the player presses the firing key.
* **Interaction**:
  + **Receives Input**: Commands from the **Player Input Module** to move or fire.
  + **Sends Bullets**: Creates and passes new bullet objects to the **Bullet Module**.
  + **Updates Position**: Provides updated coordinates to the **Game State Module**.
* **Implementation**:
  + Implements boundary checks to prevent the spaceship from leaving the screen (e.g., if ***(position.x < 0)).***

### **4. Enemy Module:**

* **Role**: Represents enemies and their behaviors, including movement and attacks.
* **Key Functions**:
  1. **Initialization**:
     + Enemies are created and positioned in a grid when the game starts.
     + Each enemy is placed at predefined locations based on a fixed grid structure.
  2. **Movement(Optional)**:
     + Moves all enemies in predefined patterns (e.g., horizontal side-to-side or downward progression).
     + Adjusts movement based on the game's difficulty level or progression.
  3. **Behavior**:
     + No new enemies are spawned during the game.
     + Enemies fire bullets based on their proximity to the player’s spaceship or at timed intervals.
* **Interaction**:
  1. Updates enemy positions in the **Game State Module**.
  2. Sends bullets to the **Bullet Module** when firing conditions are met.

### **5. Bullet Module:**

* **Role**: Manages all bullet objects, including movement, collisions, and screen boundaries.
* **Key Functions**:
  + **Update Positions**: Moves bullets forward each frame.
  + **Boundary Checks**: Removes bullets that leave the screen to optimize performance.
  + **Collision Events**: Detects collisions between bullets and other objects.
  + **Bullet speed**: Speed of bullets fired at the player’s spaceship might increase with difficulty level.
* **Interaction**:
  + Receives bullets from the **Spaceship Module** (player bullets) and **Enemy Module** (enemy bullets).
  + Sends collision events to the **Collision Detection Module**.
* **Implementation**:
  + Uses a container (e.g***., std::vector***) to store and iterate through active bullets.

### **6. Collision Detection Module:**

* **Role**: Detects and handles collisions between game objects.
* **Key Functions**:
  + **Collision Checks**:
    - **Bullet-Enemy**: Destroys enemies and adds to the player's score.
    - **Bullet-Spaceship**: Reduces the player's lives.
  + **Trigger Actions**: Signals appropriate responses (e.g., removing destroyed objects or ending the game).
* **Interaction**:
  + Notifies the **Game State Module** of collision outcomes.
  + Communicates with the **Rendering Module** to update visuals (e.g., explosion effects).
* **Implementation**:
  + Uses bounding box or circle-based collision detection techniques (e.g., ***sf::FloatRect::intersects****()).*

### **7. Game State Module:**

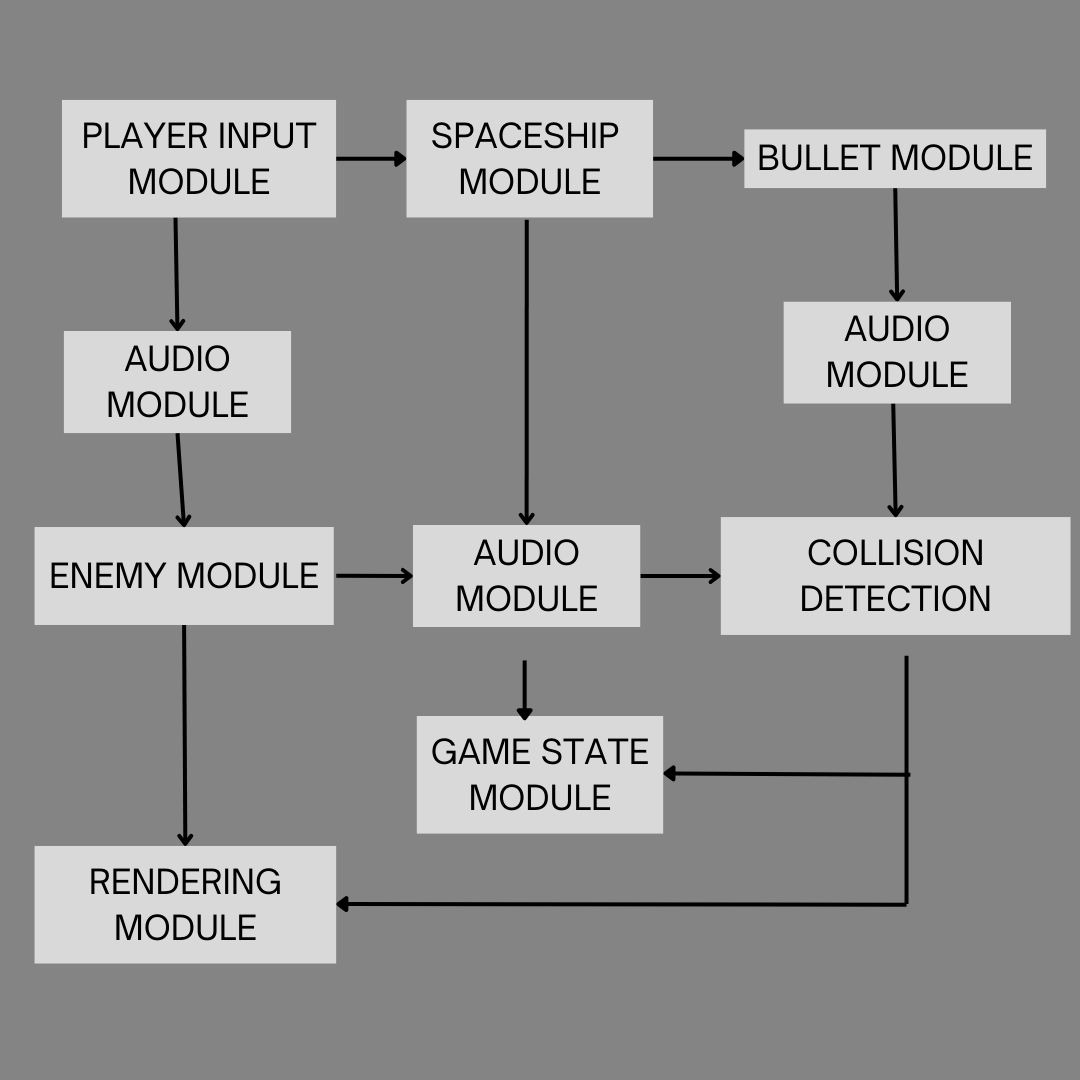
* **Role**: Maintains the overall state of the game, including score, lives, and progression.
* **Key Functions**:
  + **Score Management**: Increments the player's score when enemies are destroyed.
  + **Life Tracking**: Deducts lives when the player is hit, ending the game if lives reach zero.
  + **Level Progression**: Advances levels and adjusts difficulty as enemies are cleared.
* **Interaction**:
  + Receives collision events from the **Collision Detection Module**.
  + Sends updated positions and game status to the **Rendering Module**.
* **Implementation**:
  + Keeps track of all key metrics using variables or data structures (e.g., ***int score, int lives***).

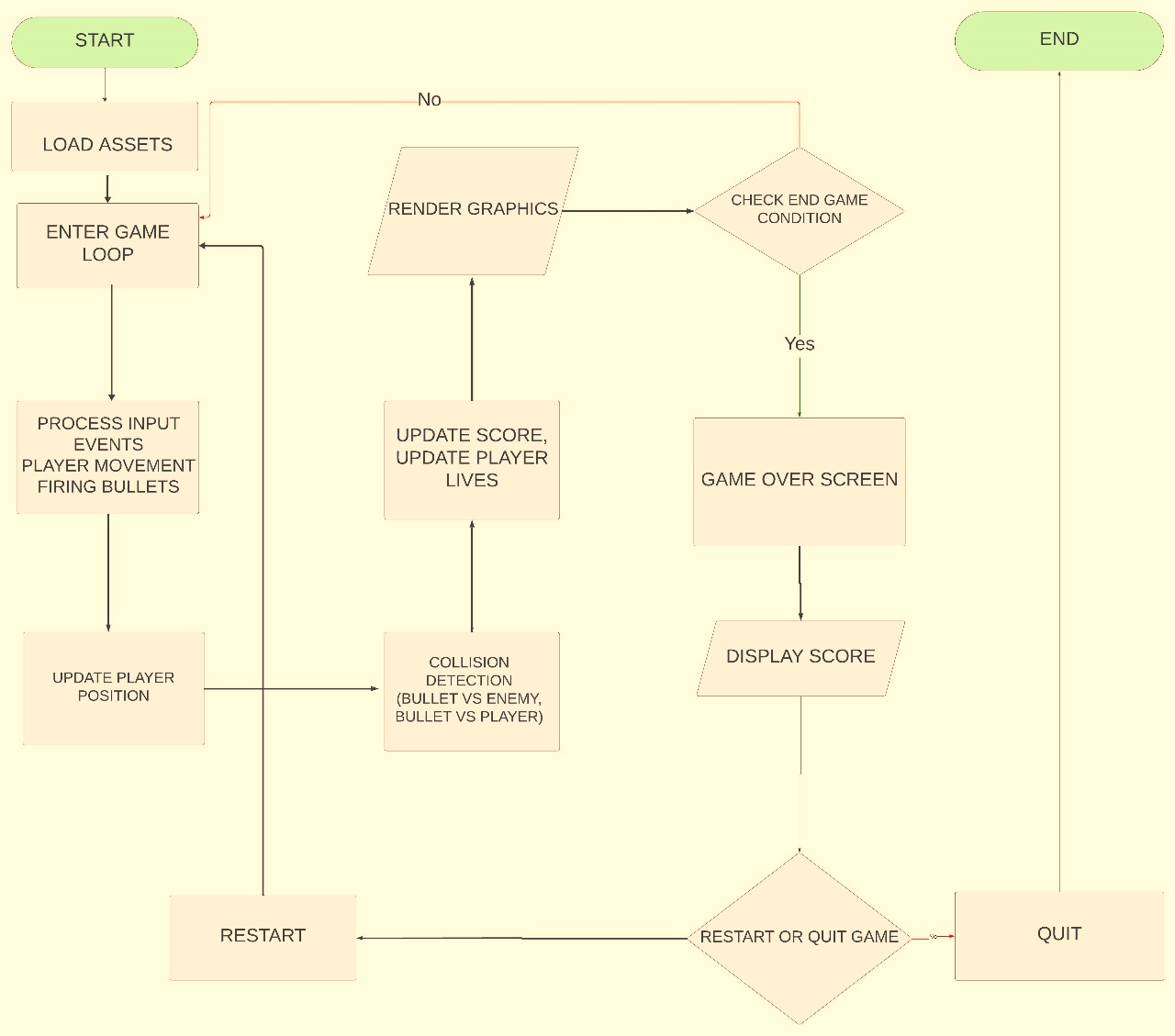
**8. Audio Module:**

* **Role: Manages sound effects and background music to enhance the gaming experience.**
* **Key Components:**
* **Sound Effects: Might include sounds for shooting, explosions, and level transitions.**
* **Background Music:** **Loops ambient music during gameplay to create immersion.**
* **Interaction:**  **Plays background music during active gameplay. Might trigger appropriate sounds based on in-game events (e.g., firing, collisions).**
* **Implementation:** **Uses SFML’s *sf::Sound* classes for loading and playing audio files.**

#### ****Architecture Diagram:****

Below is the abstract project architecture diagram:



* **Detailed Flowchart:**
* **Flowchart Steps with Technical Details:**

**1. Game Initialization:**

* **Purpose**: Initialize all required game elements and prepare the game loop.
* **Details**:
  + Set up the window using sf::RenderWindow with desired width and height.
  + Load resources (textures, fonts, sounds) using sf::Texture, sf::Font, and sf::SoundBuffer.
  + Initialize entities: player spaceship, enemies, bullets, boundaries, score, and lives.

**2. Input Handling:**

* **Purpose**: Process user input to control the player spaceship and shooting.
* **Details**:
  + Use sf::Keyboard::isKeyPressed() to detect key presses:
    - Arrow keys (Up, Down, Left, Right) for movement.
    - Spacebar (Space) to fire bullets.
  + Store bullets fired by the player in a **vector** (e.g., std::vector<sf::CircleShape>).

**3. Update Game Entities:**

* **Purpose**: Update the position and states of all game elements.
* **Details**:
  + **Player**: Move based on input and ensure it stays within the window boundary.
    - Use an sf::FloatRect for boundaries to restrict movement.
  + **Enemies**: Move randomly or in a predefined pattern. If near the player, fire bullets straight down.
    - Store bullets fired by enemies in a separate **vector**.
  + **Bullets**:
    - Update positions (player bullets move upward, enemy bullets downward).
    - Remove bullets if they exit the screen using erase-remove idiom.
    - Data Structures: **Vectors** are used to store bullets for both the player and enemies.
  + **Collision Detection**:
    - Check if:
      * Player bullets collide with enemies.
      * Enemy bullets collide with the player.
    - Use sf::FloatRect::intersects() for collision detection.
    - On collision:
      * Remove the collided entity (e.g., enemy or bullet).
      * Update score or lives.

**4. Rendering:**

* **Purpose**: Draw all updated game entities onto the screen.
* **Details**:
  + Use sf::RenderWindow::clear() to clear the window at the beginning of each frame.
  + Draw each entity:
    - Player spaceship (sf::Sprite or sf::ConvexShape).
    - Enemies (sf::RectangleShape).
    - Bullets (sf::CircleShape).
    - Score and lives (sf::Text).
  + Use sf::RenderWindow::display() to present the updated frame.

**5. Collision Checking:**

* **Purpose**: Manage interactions between game elements.
* **Details**:
  + Player bullet vs. enemy: Increment score and remove enemy/bullet.
  + Enemy bullet vs. player: Decrease player lives and remove bullet.
  + Use algorithms optimized for collision detection:
    - Iterate through vectors of bullets and enemies.
    - Check intersections using bounding boxes (sf::FloatRect).

**6. Game State Updates:**

* **Purpose**: Manage game progression, score, and lives.
* **Details**:
  + Update **score**:
    - Increment by predefined points (e.g., 10 points per enemy).
  + Update **lives**:
    - Decrease when the player gets hit.
    - If lives reach zero, set the **Game Over** flag.

**7. Game Over Handling:**

* **Purpose**: Stop the game and display the final score.
* **Details**:
  + Check if player lives reach 0.
  + Pause the game loop.
  + Display "Game Over" and final score using sf::Text.

**Algorithms and Data Structures:**

1. **Data Structures**:
   * **Vectors**:
     + Store bullets for both player and enemies (std::vector<sf::CircleShape>).
     + Maintain dynamic entities (e.g., active enemies or bullets).
   * **sf::FloatRect**:
     + Used for boundary enforcement and collision detection.
   * **sf::Text**:
     + Used to display the score, lives, and "Game Over" message.
2. **Collision Detection Algorithm**:
   * Iterate through each player bullet and check for collisions with all active enemies using bounding box checks (sf::FloatRect::intersects()).
   * Iterate through each enemy bullet and check for collisions with the player.
3. **Random Enemy Movement**:
   * Use std::rand() or std::uniform\_int\_distribution to generate random directions or speeds for enemies.
   * Ensure they stay within boundaries.
4. **Game Loop Logic**:
   * Standard SFML game loop:

while (window.isOpen()) {

handleInput(); // Process player input

updateEntities(); // Update positions, collisions, score

render(); // Draw everything on screen

}

#### ****4. Breakdown of Group Work (4 Members)****

**Task Distribution:**

1. **Input and Rendering Module & Audio (Hamna Shah):**
   * Develop the Input Module to handle keyboard inputs for player movement and shooting.
   * Create and render player and enemy sprites using SFML.
   * Display UI components, including score and player lives.
   * Add sound effects for shooting, explosions, and other events using SFML.
   * Implement background music for the game.
2. **Game Logic Module (Anas Norani & Hanan Majeed):**
   * Implement enemy movement patterns, level progression, and difficulty scaling.
   * Manage collision detection between projectiles, enemies, and the player.
   * Define win/loss conditions and ensure proper game flow.
3. **Data Module and High Score Management (Noor Ul Saba):**
   * Create the system to track and save high scores.
   * Manage persistent data for settings and configurations.
   * Test and debug the overall system for performance optimization.