Flappy Hitler

MAHI VIJAY

25BAI10333

# VITyarthi - Build Your Own Project

### General Project Instructions & Submission Guidelines

As part of the flipped course evaluation, each student must build their **own original project**, aligned with the concepts covered in the subject syllabus.

Instead of a fixed project, you now have the freedom to propose, design, and build a project of your choice, provided it meets the academic and technical expectations described below.

# INTRODUCTION

# This project, titled "Flappy Hitler vs. Stalin," is a simple side-scrolling game built using Python and the Pygame library. It serves as the project submission for the flipped course evaluation , applying subject concepts in a real-world context (game development/simulation). The project objective is to allow students to design, implement, and demonstrate a technical solution . The game involves controlling a character, "Flappy Hitler," to navigate through gaps in pipes, which are represented by images of "Joseph Stalin." The goal is to maximize the score by passing pipe pairs while avoiding all collisions.

# Project Objective

The core problem addressed is to **develop an engaging, working simulation/game** that effectively applies concepts covered in the subject (e.g., object-oriented programming, state machines, physics simulation, graphics/UI programming, etc., depending on the specific course).

**In the context of the game itself:** The problem is to **implement a stable, side-scrolling game environment** that challenges the player with continuous, randomly generated obstacles (pipes) requiring precise timing and control (flapping) to achieve a high score.

# Scope & Requirements

Your project must include:

## Functional Requirements

## The project must include at least three major functional modules.

## ->Game State Management (Core Module):

## Initialize and manage the game loop, transitioning between the start screen, active gameplay, and the 'Game Over' state.

## Handle game setup (asset loading, variable initialization).

## Restart the game upon user input after a collision.

## ->Player Control & Physics (Functional Module):

## Implement gravity to pull the player character ("Flappy Hitler") downwards.

## Implement flap control (Spacebar) to counteract gravity and gain height.

## Manage the player's position within the window bounds (collision with screen edges).

## ->Obstacle Generation & Scoring (Functional Module):

## Generate pipe obstacles ("Joseph Stalin") at regular intervals.

## Randomize the vertical gap position for each pipe pair.

## Implement side-scrolling movement for the pipes.

## Detect collision between the player and the pipes or screen edges.

## Increment the score when the player successfully passes a pipe gap

## Non-Functional Requirements

At least four non-functional requirements must be specified.

* **Performance:** The game must run at a consistent **60 FPS** to ensure smooth animation and responsive gameplay.
* **Usability:** The game must have **clear, intuitive controls** (Spacebar for flap) and a straightforward mechanism for starting and restarting.
* **Reliability:** The game must execute the collision detection logic accurately and reliably to ensure a fair 'Game Over' condition.
* **Error Handling Strategy:** The code should correctly handle cases where resource files (images, sounds) are missing or misnamed, preventing a fatal crash (e.g., using try-except blocks or clear initial asset checks).

# Technical Expectations

The system architecture follows a typical **Event-Driven/Game Loop** pattern using the Pygame framework.

1. **Input Handler:** Captures user inputs (e.g., Spacebar press for flap, any key for restart).
2. **Game Engine (Pygame):** The central control loop running at 60 FPS, managing the game state (run/pause/game over).
3. **Entity/Object Modules (Classes):**
   * **Player Class:** Manages the position, movement logic, and rendering of "Flappy Hitler."
   * **Pipe Class:** Manages the generation, movement, and rendering of the "Stalin" pipe obstacles.
   * **Score/UI Class:** Manages the score display and game over screen elements.
4. **Physics Module:** Handles gravity, vertical velocity, and collision detection.
5. **Asset Manager:** Loads and handles image, sound, and music resources.

**5. Design Diagrams**

Workflow Diagram

A workflow diagram illustrates the logical flow of the game's core loop:

1. **Start/Idle State:** Wait for user input (any key).
2. **Game Start:** Load assets, initialize score to 0, play background music.
3. **Game Loop (60 FPS):**
   * **Input Handling:** Check for Spacebar press (flap).
   * **Update Physics:** Apply gravity, update player position and pipe positions.
   * **Collision Check:** Check player vs. pipe/screen bounds.
   * **Scoring Check:** Check if the player passed a pipe.
   * **Rendering:** Draw background, pipes, player, and score.
4. **Collision Detected:**
   * Stop background music, play Game Over sound effect.
   * Display Game Over screen and meme image.
   * Transition to Game Over State.
5. **Game Over State:** Wait for user input to restart.

Class Diagram / Component Diagram

A Class Diagram would show the primary object-oriented components:

* **Game Class:** Manages initialization, the main loop, game state, and rendering.
* **Player Class (Hitler):** Attributes: position (x, y), velocity (y), score. Methods: flap(), update(), draw(), check\_collision().
* **Pipe Class (Stalin):** Attributes: position (x, y), gap height, speed. Methods: update(), draw(), is\_offscreen().
* **AssetManager Class:** Handles loading and accessing the ww2.jpg, adolf hitler.png, joseph stalin.png, etc.

**6.Design Decisions & Rationale**

* **Pygame Selection:** Pygame was chosen for its relative simplicity and effectiveness in quickly developing 2D games, which aligns with the project's scope.
* **60 FPS:** Running the game at 60 FPS (frames per second) was chosen to ensure **smooth animation** and predictable physics updates, improving the player experience (Usability/Performance).
* **Modular Design:** The use of separate classes/modules for the Player, Pipes, and Game logic ensures a **modular and clean implementation**, making the code easier to maintain, debug, and expand (Maintainability/Technical Expectation).
* **Randomized Gaps:** Randomizing the vertical position of the pipe gaps was chosen to maintain a high level of difficulty and **replayability**.

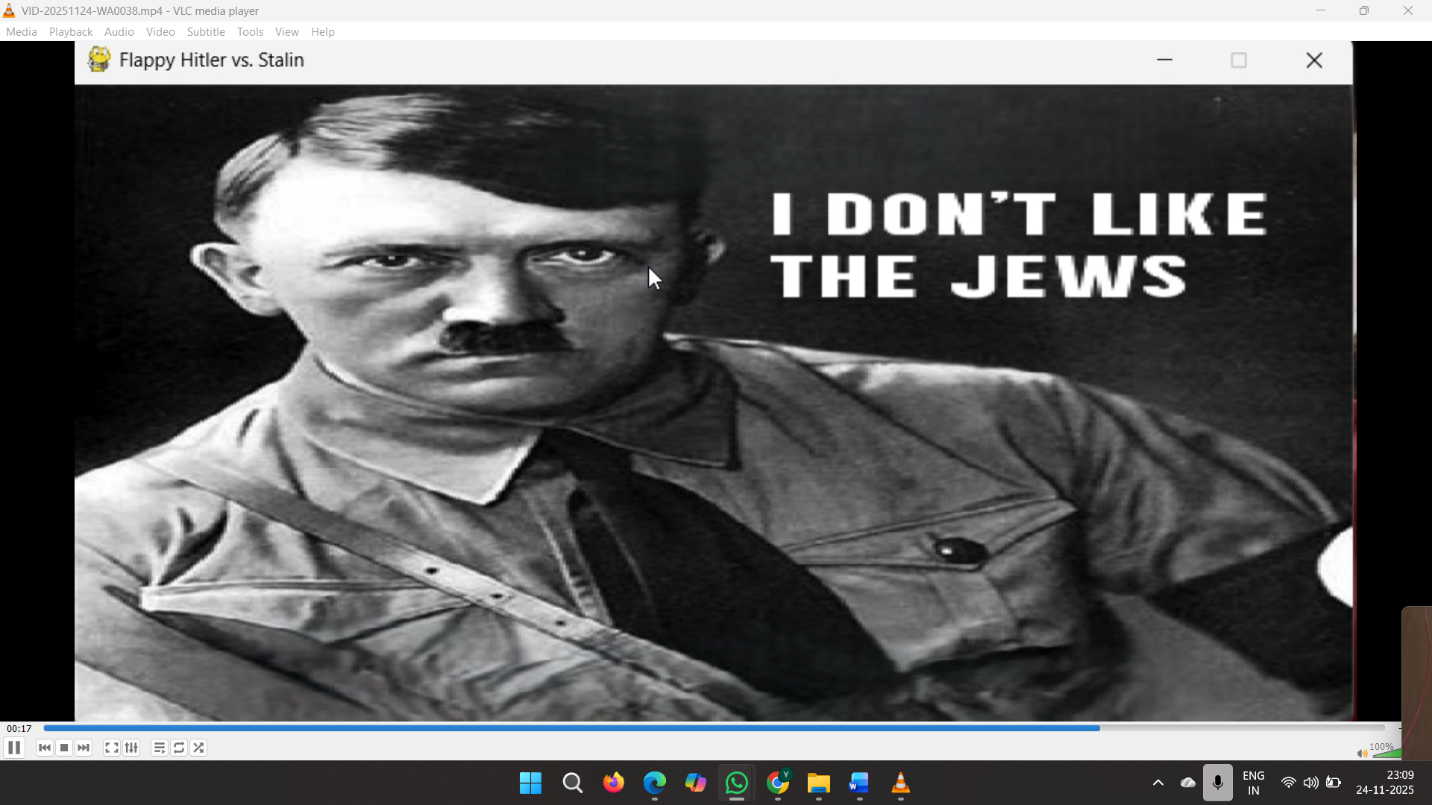
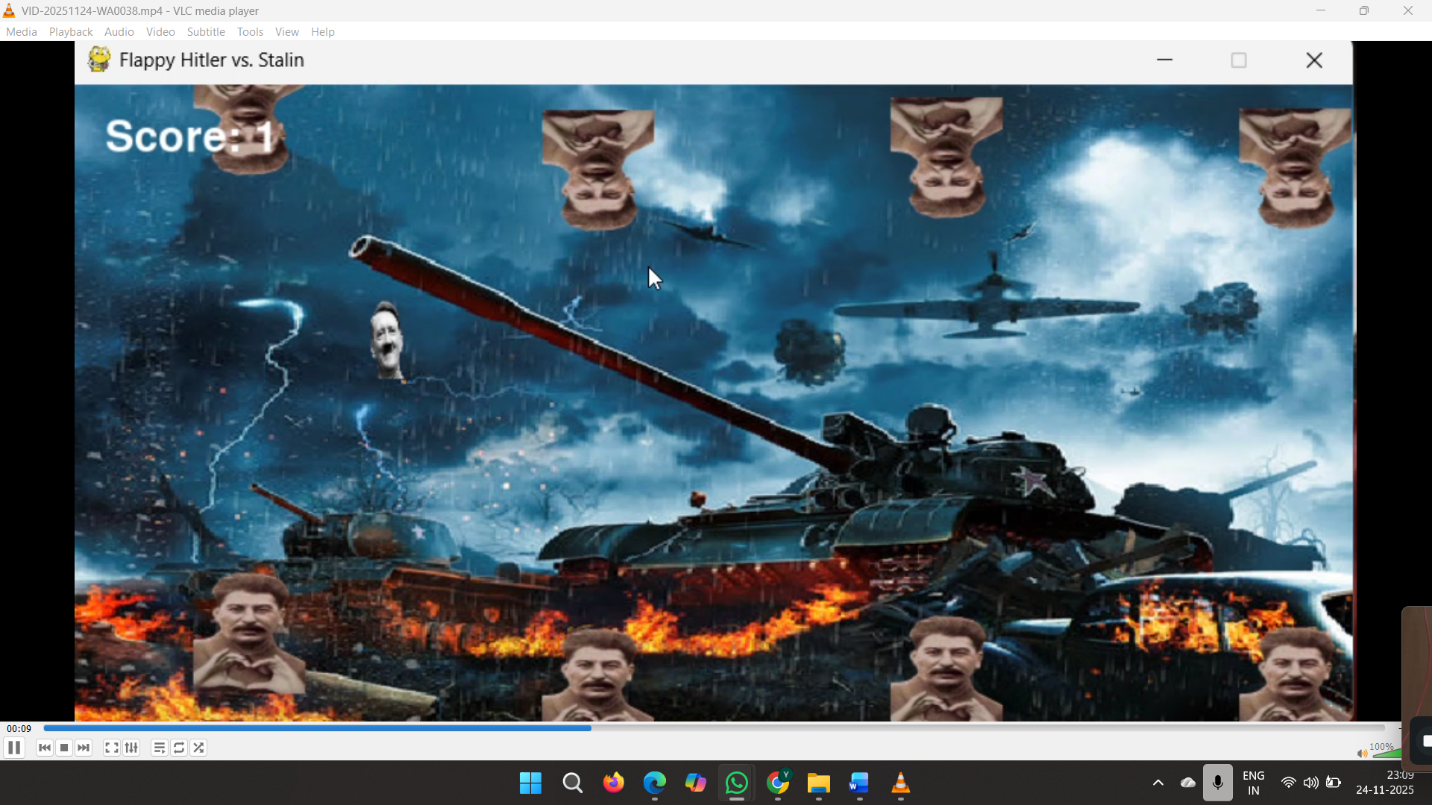
**7. Implementation Details**

The project implementation used **Python 3.x** and the **Pygame library**. The implementation includes a minimum of **5-10 meaningful modules/classes/files** (e.g., main.py, player.py, pipe.py, constants.py, assets.py—assumed folder structure).

Key implementation aspects:

* **Gravity Simulation:** The player's vertical velocity is continuously increased by a constant gravity value each frame.
* **Flap Mechanic:** Pressing the Spacebar instantaneously sets the vertical velocity to a negative value, simulating a jump/flap.
* **Collision Detection:** Simple bounding box collision detection is used between the player sprite and the pipe sprites.
* **Asset Management:** All image and audio assets are loaded once at startup and referenced during the game loop.

**8. Screenrecording/screenshots**

* **In-Game Screenshot:** (Conceptual placeholder for recommended screenshot)
* **Game Over Screen:** Display of the final score and the "funny meme display" upon collision. 

**9. Testing Approach**

The testing approach for this project is primarily focused on **validation tests**:

1. **Functional Testing:**
   * **Flap Control Test:** Verify that pressing the Spacebar reliably makes the player move up.
   * **Scoring Test:** Verify that the score increments exactly once when the player successfully passes a pipe gap.
   * **Collision Test:** Verify that hitting the top/bottom of the screen or any part of the pipe structure correctly triggers the 'Game Over' state.
2. **Performance Testing:** Check that the game consistently runs at 60 FPS under normal operating conditions.
3. **Usability Testing:** Confirm that the game restart mechanism works correctly after 'Game Over'.

**10. Challenges Faced**

* **Smooth Pipe Generation:** Initially, ensuring consistent and smooth generation of pipe pairs while managing their off-screen cleanup required careful timing and object management within the game loop.
* **Collision Accuracy:** Fine-tuning the size of the collision bounding box for both the player and the pipes was necessary to achieve a fair and predictable collision experience.

**11. Learnings & Key Takeaways**

* Reinforced understanding of the **Game Loop** design pattern and state management.
* Applied object-oriented principles by creating distinct classes for game entities (Player, Pipe).
* Gained practical experience with Pygame's drawing, event handling, and basic physics simulation (gravity/velocity).

**12. Future Enhancements**

1. **Difficulty Scaling:** Implement an increasing difficulty by speeding up the pipes or reducing the gap size over time.
2. **High Score Persistence:** Add a mechanism to save and display the all-time high score using a simple file or database.
3. **Power-ups:** Introduce temporary in-game power-ups (e.g., temporary shield or slow motion).

**13. References**

* Pygame Documentation (<https://www.pygame.org/>)
* Python 3.x Documentation (<https://www.python.org/>)
* [Any specific tutorials, code examples, or documentation used

**15.Evaluation Rubric**

|  |  |
| --- | --- |
| **Component** | **Weightage** |
| Problem Understanding & Requirements | 10% |
| Design & Documentation | 20% |
| Implementation Quality | 25% |
| Innovation, Depth & Complexity | 15% |
| GitHub Repository & Version Control | 10% |
| Project Report | 20% |
| **Total** | **100%** |