Individual Project 1 – Jump King Clone

| Arnav Mehta  COMP 350 ON1 Winter 2025  300196365 |
| --- |

# BSTRACT

This report outlines the comprehensive process of designing, coding, and developing the gameplay features for Project 1 in COMP350 ON1**,** a Jump King-inspired 2D platformer. The analysis covers the entire development cycle, detailing the design methodology, system architecture, and object-oriented implementation strategies that underpin the game’s mechanics. Key components are encapsulated in specialized classes, including Player, Environment, GameState, and PlatformMaker, each managing critical functions such as realistic physics simulation, collision detection, and interactive user input. The gameplay is structured to challenge players with precision-based jumps, dynamic platform interactions, and progressive difficulty that advances from an introductory tutorial to a fully realized victory condition. **T**his report demonstrates how careful planning and modular programming techniques contribute to a scalable, maintainable, and engaging gaming experience.

# INTRODUCTION

The theme of this project is to create a challenging 2D platformer reminiscent of Jump King [1][2]. In this game, the player must navigate a series of precarious platforms and obstacles while dealing with realistic physics that affect each jump and landing. Inspired by research on difficult platformer mechanics (see references [1] and [2]), the game emphasizes precision, strategy, and timing. The narrative might follow a determined protagonist attempting to ascend a mysterious tower filled with traps and secrets.

The code structure follows an Object Oriented Design and uses the Processing 4 language and compiler to create a working sample of the proposed game.

# Project overview

The project is a 2D platformer where players must ascend through a series of platforms using controlled jumping mechanics. The game is designed to be challenging, requiring skillful timing and precision. A key gameplay feature is that missing a platform results in falling to a lower level, increasing the difficulty and requiring repeated attempts to progress.

## Game objective

The primary goal of the game is to reach the topmost platform by executing well-timed jumps. The jumps are influenced by gravity, momentum, and charge mechanics. Players must gauge their jump distance accurately to avoid falling and having to repeat sections of the level.

## Game flow

1. Startup Screen: Displays the game’s title, menu options, and navigation interface.
2. Gameplay Mechanics: Players utilize keyboard and mouse inputs to execute jumps between platforms.
3. Level Progression: Increased difficulty is introduced through platform spacing, elevation, and collision constraints.
4. Game Completion: The final platform serves as the victory condition, marking the conclusion of the level.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

# Gameplay Mechanics and Controls

The game is controlled using both the keyboard and mouse, creating an intuitive but challenging movement system. Players must carefully judge their jump trajectory and power to navigate the platforms effectively.

## Jumping Mechanics

The primary gameplay mechanic involves jumping between platforms. The jump is affected by two key factors:

1. Direction Control: The cursor is used to determine the direction of the jump. A line extending from the player character visually represents the intended trajectory.
2. Jump Power: Holding down the spacebar increases the jump power, represented by an on-screen charge indicator. Releasing the spacebar executes the jump with the accumulated power.

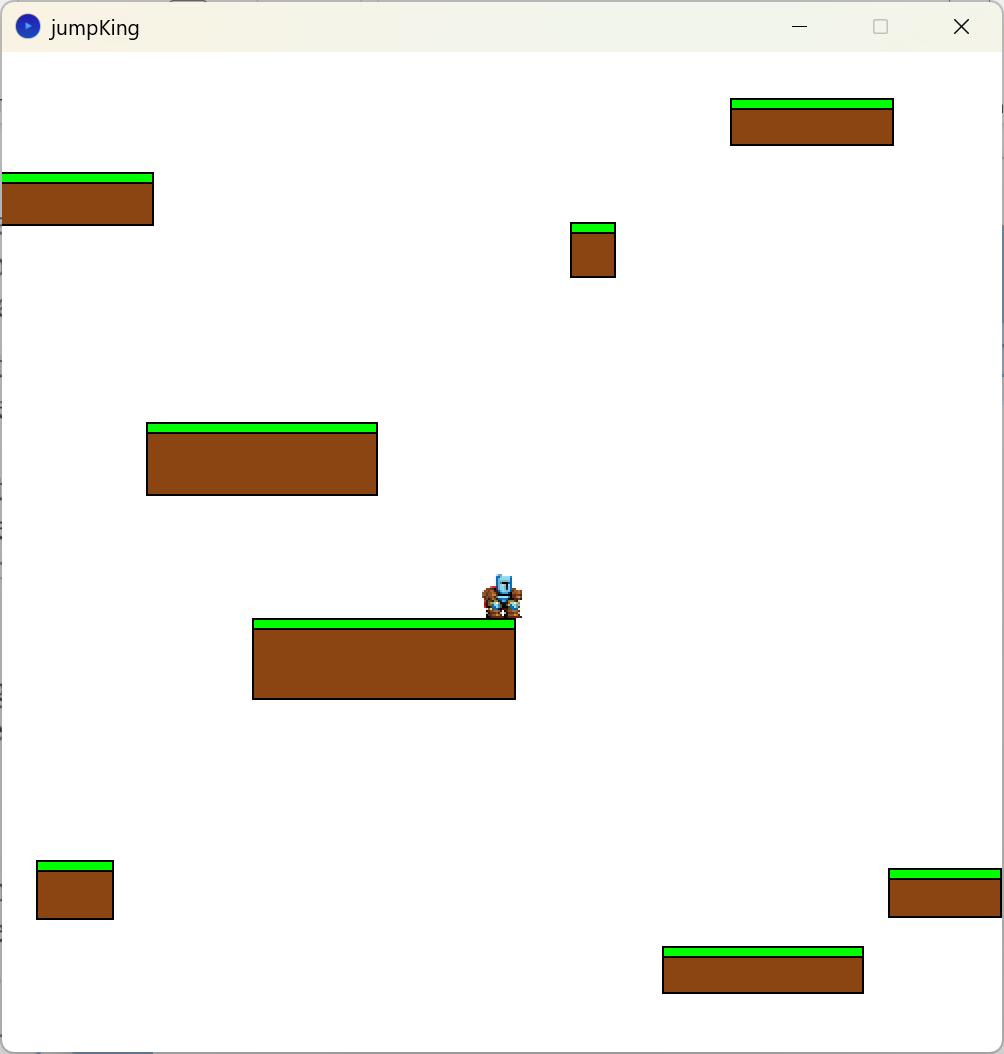
A video game graphics

AI-generated content may be incorrect.

## Graphics and Visual Design

The game features a pixel-art aesthetic inspired by classic platformers. The background, platforms, and player character are designed with a retro-style color palette.

1. Platforms: Designed with support for varying textures to indicate different properties (e.g., solid platforms vs. slippery ones). While I didn’t have time to implement and test these different types, implementing it would be very easy because of the object oriented design.
2. Player Character: A simple yet expressive sprite with animation frames for idle, charging, jumping, and landing.
3. UI Elements: The charge indicator and jump trajectory guide enhance the player's decision-making.



# Code Structure and Classes

The game is structured following an object-oriented programming (OOP) paradigm, with distinct classes encapsulating specific functionalities.

## Game State Class (GameState.pde)

This class manages UI interactions, menu navigation, and global game states.

### Core Functions:

1. displayControls(): Renders the control scheme and gameplay instructions.
2. displayMainMenu(): Manages the startup menu, offering start, controls, and exit options.
3. handleKeyPressed(): Handles input events related to pausing and menu transitions.
4. handleMousePressed(): Detects and processes menu navigation based on user input.

class GameState {

boolean PAUSED;

boolean CONTROLS;

int[] buttonSize;

int buttonGap;

GameState() {}

void displayControls() {}

void displayMainMenu() {}

void handleKeyPressed() {}

void handleMousePressed() {}

}

## ****Environment Class (Environment.pde)****

This class is responsible for collision detection, level transitions, and environmental rendering.

### Core Functions:

1. update(): Maintains real-time updates on player interactions with the environment.
2. checkCollisions(): Evaluates platform-player collision status.
3. checkLevel(): Determines level transitions when movement exceeds predefined boundaries.
4. checkPlatformCollisions(): Resolves interactions between player and platforms.
5. checkWindowEdges(): Enforces screen constraints on player movement.

class EnvironmentHandler {

Player currentPlayer;

int totalLevels;

int currentLevel;

Platform[][] platforms;

int levelChangeCooldown;

EnvironmentHandler() {}

void update() {}

void checkCollisions() {}

void checkLevel() {}

void checkPlatformCollisions() {}

void checkWindowEdges() {}

}

## Platform Class

This class represents individual platforms in the game.

class Platform {

float x, y, width, height;

Platform(float x, float y, float width, float height) {}

void display() {}

}

## Player Class (Player.pde)

The Player class extends the Person superclass, incorporating movement physics, control input handling, and jump execution logic.

### Parent Class: Person

The Person class serves as a base class for characters in the game, encapsulating fundamental attributes such as position and movement.

### Child Class: Player

The Player class inherits properties and methods from Person while adding functionalities unique to the playable character.

### Core Functions:

1. update(): Processes movement, applying gravity and jump charge accumulation.
2. jump(): Calculates jump trajectory based on user input and environmental context.
3. display(): Renders the player sprite and visual UI elements, including the jump charge indicator.
4. keyEvent(): Detects spacebar input to initiate jump charging.
5. keyReleased(): Executes jumps upon spacebar release.
6. updateMousePosition(): Tracks mouse movement for jump direction calibration.

class Player extends Person {

float xvel;

float yvel;

float baseJumpPower;

float maxJumpPower;

float jumpCharge;

float gravity;

float maxFallSpeed;

float friction;

float bounceMultiplier;

boolean isOnGround;

boolean canJump;

boolean isCharging;

boolean wasOnGround;

boolean facingRight;

float mouseX;

float mouseY;

float arrowLength;

PImage guy;

Player() {}

void update() {}

void jump() {}

void display() {}

void keyEvent() {}

void keyReleased() {}

void updateMousePosition() {}

}

## JumpKing Main File (jumpKing.pde)

This serves as the entry point of the application, orchestrating interactions between various components.

### Core Functions:

1. setup(): Initializes the game environment, platform structures, and input handlers.
2. draw(): Controls the execution flow between menu interactions and gameplay updates.
3. keyPressed(): Routes key input events to the relevant handlers.
4. keyReleased(): Manages post-input state modifications.
5. mousePressed(), mouseDragged(), mouseReleased(): Handles input interactions for both menu selections and level editing.

class JumpKing {

EnvironmentHandler envHandler;

GameState gameState;

PlatformMaker platformMaker;

void setup() {}

void draw() {}

void keyPressed() {}

void keyReleased() {}

void mousePressed() {}

void mouseDragged() {}

void mouseReleased() {}

}

## Platform Maker Class (PlatformMaker.pde)

This is an auxiliary tool facilitating the manual placement and testing of platform objects.

### Core Functions:

1. handleKey(): Activates level editing mode and outputs platform coordinate data.
2. handleMousePressed(): Sets the initial reference point for platform placement.
3. handleMouseDragged(): Adjusts platform dimensions dynamically.
4. handleMouseReleased(): Finalizes and saves platform configurations.
5. display(): Renders platforms in real time for design verification.

class PlatformMaker {

boolean isActive;

boolean isDrawing;

PVector startPoint;

PVector currentPoint;

ArrayList<Platform> createdPlatforms;

PlatformMaker() {}

void handleKey() {}

void handleMousePressed() {}

void handleMouseDragged() {}

void handleMouseReleased() {}

void display() {}

}

# REFERENCES

1. Nexile. (n.d.). Jump King - There is a Smoking Hot Babe at The Top! Retrieved February 8, 2025, from <https://www.jump-king.com/>
2. Nexile. (2019, May 3). *Jump King*. Steam. <https://store.steampowered.com/app/1061090/Jump_King/>
3. Ooga Booga™. (2020, December 22). *Beginner's Guide to Jump King!*. Steam Community. <https://steamcommunity.com/sharedfiles/filedetails/?id=2330386421>