Doodle Jump: Design and Implementation

Emir Murat (20231527)

January 2025

Introduction

This report details the design and implementation of the *Doodle Jump* game using C++ and the SFML graphics library. Developed as part of the Advanced Programming course, this project demonstrates the application of object-oriented principles and design patterns. Screenshots are included to clarify specific concepts.

Game Features

The game simulates a dynamic world where the player (the doodler) ascends by jumping on platforms while navigating obstacles and collecting bonuses.

Platform Types

Platforms are randomly generated and categorized as follows:

- **Static platforms** (green): Stationary platforms.
- **Horizontally moving platforms** (blue): Move side-to-side.
- **Vertically moving platforms** (yellow): Move up and down.
- **Temporary platforms** (white): Break after one use.

If a platform is jumped on twice or more, a penalty decreases the score. This penalty is by default -5, but can be specified per platform.

Stages and Platform Distribution

The game progresses through four stages, with platform distribution adapting to player height:

- **Stage 1** ($height < 500$): 100% static platforms.
- **Stage 2** ($500 \leq height < 1500$): 80% static, 20% horizontal platforms.
- **Stage 3** ($1500 \leq height < 4000$): 55% static, 30% horizontal, 15% vertical platforms.
- **Stage 4** ($4000 \leq height < 7000$): All platform types with dynamic weights.

Bonuses and Obstacles

- **Bonuses:**
  - **Spring**: Increases jump height by a factor of 5.
  - **Jetpack**: Propels the player upward for 3 seconds.
- **Obstacles:**
  - **Black Hole(Extra)**: Applies lateral forces, complicating horizontal movement. The black hole has its effect at the moment it is in view, and will apply a force for 2.0 seconds by default. The duration can be customized via the Configs/GameConfig.h file. Black Holes occur when the height is 4000 or higher.

Overall Structure

Smart Pointers

Throughout the project, only smart pointers (shared ptr and unique ptr) are used, ensuring memory safety and preventing memory leaks. A memory check with `Valgrind` confirmed that no memory leaks were detected.

Templates

For the `Subject` and `Observer` classes, I implemented template classes to enhance flexibility and reduce redundancy. This approach allows the creation of an observable `Subject` (e.g., `Score`, `Entity`, etc.) without requiring separate `notify(Sub-Subject*)` functions for each type. Instead of creating distinct classes like `EntityObserver` or `ScoreObserver`, I leveraged templates to declare the observable type directly by inheriting as `class Name : [access specifier] Observer<ObservedSubject>`. This significantly reduces the number of additional classes needed, streamlining the implementation while maintaining type safety.

Namespaces

The game logic is organized into seven namespaces for clarity and modularity:

1. `gameModel`: Core logic objects (e.g., `Entities`, `Score`, `Camera`).
2. `gameControl`: Logic controllers and entity creation (`World`, generators).
3. `gameView`: SFML sprites and views (`EntityViews`, `ScoreView`).
4. `configs`: Global configurations (e.g., FPS, resolution, stage thresholds).
5. `exceptions`: Exception handling for issues like file or texture loading.
6. `assetManager`: Centralized asset storage (`FontManager`, `TextureManager`).
7. `events`: Input conversion for game logic (`EventHandler`).

Design Principles

The project adheres to key object-oriented principles:

- **Single Responsibility Principle:** Each class has a specific role (e.g., `TextureManager` handles and stores textures, `EventHandler` manages inputs, `EntityView` and its derived classes only store the Sprites and get updated, ...).
- **Open/Closed Principle:** Components can be extended without modifying existing code, e.g., new entities derived from `Entity` or new events can be introduced by deriving new classes from the `Event` class. This makes it easier to add new functionality triggered by a user event (for example a shooting event).
- **High Cohesion/Low Coupling:** The elements and functionalities within the classes are highly dependent on each other, which aligns with the Single Responsibility Principle. However, the use of Singleton classes such as Random and Stopwatch slightly violates Low Coupling, as they introduce some shared dependencies. Despite this, the overall functionality of the classes remains largely independent from one another.

Design Patterns

The following design patterns were implemented in the game:

Model-View-Controller (MVC)

Game logic is encapsulated in a static library (`doodle_jump`) without SFML dependencies. Input from the user is processed by the `EventHandler` and passed to the `World` (game control), which updates logical entities (`gameModel`) and notifies corresponding views (`gameView`) to render updates.

Abstract Factory

The abstract class `AEntityFactory` which is shared between the `World` and `Generators`, provides an interface for creating game entities. The concrete factory `CEntityFactory` generates both logic and view objects and attaches them using the observer pattern.

Observer Pattern

View objects act as observers of logical entities. If the internal state of the logical entities is changed, the observers in the `Subject` which represents the observable logic elements, notifies the appropriate "Views". This provides a structure to easily add new visual elements or audios.

Singleton

Classes like `Random`, `Stopwatch`, `FontManager`, and `TextureManager` are implemented as singletons to ensure a single instance throughout the game. This is useful for providing a central access point for all fonts, sprites and ensuring one (synchronized) chronometer is present in the game.

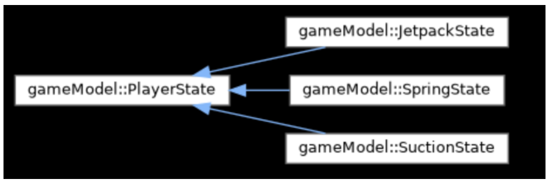
State Pattern (Extra)

The `PlayerState` interface defines a flexible structure for managing different "player states." The concrete implementations of this interface are `JetpackState`, `SpringState`, and `SuctionState`. This design allows the player's behavior to change dynamically without modifying the `Player` class itself. Bonuses and obstacles that trigger a state change (e.g., upon collision) create a new state as a unique pointer and transfer it to the player using `std::move` semantics.

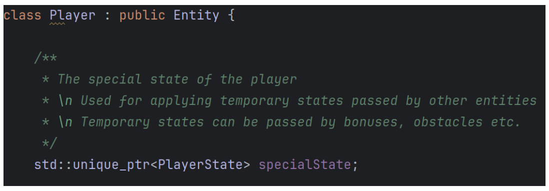
Additional Features

- **Start Screen:** A playful entry point where the game starts upon pressing **Space**.
- **Game Over Screen:** Displays the current score and high score, saved in `HighScore.txt`.
- **Black Hole Obstacle:** Adds lateral forces, increasing game difficulty without killing the player.

The State Pattern makes it easy to add new entities with unique behaviors, such as bonuses or obstacles, without altering existing code.



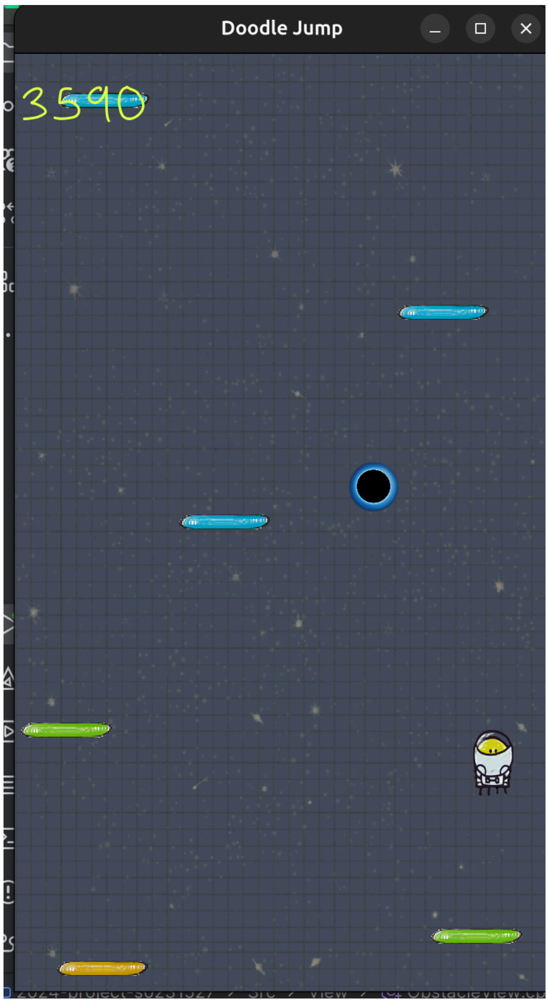
(a) PlayerState hierarchy

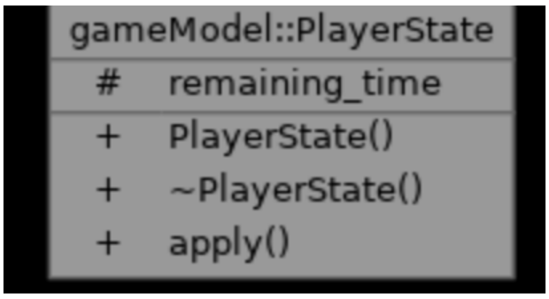


```
class Player : public Entity {

    /**
     * The special state of the player
     * \n Used for applying temporary states passed by other entities
     * \n Temporary states can be passed by bonuses, obstacles etc.
     */
    std::unique_ptr<PlayerState> specialState;
```

(b) PlayerState pointer in Player class





(c) PlayerState interface

(d) Black Hole example

Figure 1: Extra clarifying images