Design Report: Modular C++ Game Engine Architecture

1. Introduction

This report details the architectural design and implementation techniques of a C++20 ASCII-based game engine. The objective of this project was to create a highly modular, reusable, and extensible framework capable of supporting grid-based games such as Space Invaders and Flappy Bird.

The core philosophy of the engine relies on the **separation of concerns**. By strictly decoupling the **Model** (logic/physics), the **View** (Ncurses rendering), and the **Controller** (Input handling), the engine allows developers to define game rules without worrying about low-level terminal commands. Furthermore, the engine utilizes the **Strategy Pattern** heavily to compose complex entity behaviors (such as gravity, user-control, and animation) dynamically at runtime, avoiding the "class explosion" problem often found in inheritance-heavy game architectures.

2. Core Architectural Patterns

2.1 Model-View-Controller (MVC)

The engine is stratified into three distinct layers:

1. **The View Layer (`Screen`, `Window`, `Viewer`):**
   - **Responsibility:** Handles all output to the terminal.
   - **Abstraction:** The code wraps the `ncurses` library within `Window` objects. The `Viewer` classes (`GameViewer`, `StatusViewer`) consume these windows to draw shapes or text. This abstraction ensures that the rest of the engine (`Entity`, `Model`) never calls `ncurses` functions directly, making it possible to swap the rendering backend (e.g., to SDL or OpenGL) in the future without rewriting game logic.
2. **The Controller Layer (`AbstractInput`, `InputKeyboard`):**
   - **Responsibility:** Captures raw hardware events (keystrokes) and buffers them for the game loop.
   - **Abstraction:** `AbstractInput` defines a contract for input polling. This allows for easy implementation of different controllers (e.g., an AI bot controller or a file-based replay controller) by simply subclassing `AbstractInput`.
3. **The Model Layer (`Model`, `Entity`, `Shape`):**
   - **Responsibility:** Maintains the state of the game world.

    - **Physics Engine:** The `Model` class acts as the physics engine. It iterates through entities, calculates next positions based on strategies, enforces boundaries (solid vs. non-solid borders), and detects collisions using an O(N^2) bounding box check.

2.2 The Strategy Pattern (Behavioral Composition)

A critical feature of this engine is how it handles `Entity` movement. In traditional OOP, one might create a `GravityEntity` class or a `PlayerEntity` class. However, what if an entity needs *both* gravity and player control?

This engine uses a **Chained Strategy Pattern**.

- **Base Class:** `MoveStrategy`.
- **Mechanism:** Every `Entity` holds a pointer to a `MoveStrategy`. Crucially, every `MoveStrategy` holds a pointer to the *next* strategy (`nextStrategy`).
- **Recursive Execution:** When `Entity::update()` is called, it triggers `strategy->move()`. That strategy performs its logic (e.g., adds downward velocity) and then calls `nextStrategy->move()`.
- **Application:** In the Flappy Bird implementation, the Bird entity has a `PlayerMovementStrategy` (for flapping) chained to a `GravityMoveStrategy` (for falling). This allows complex physics to be composed of small, reusable logic blocks.

2.3 The Decorator/Component Approach

While not a pure Component-Entity-System (ECS), the engine moves away from deep inheritance hierarchies:

- **Collision:** Handled by `AbstractCollider`. An entity "has a" collider, rather than "is a" collidable.
- **Animation:** Handled by `CyclicTransformation`. This is a `MoveStrategy` that doesn't move coordinates but instead cycles through `Shape` bitmaps, effectively animating the sprite (used for Space Invader aliens).

3. Detailed Class Responsibilities

3.1 The Engine Core

- **`GameEngine`**: The "God Object" that ties the layers together. It owns the `Screen`, `Model`, and `Clock`. It runs the main `while(isRunning)` loop, ensuring that Input, Update, and Render phases happen in the correct order.
- **`Clock`**: Manages frame pacing. It ensures the game runs at a consistent speed regardless of the computer's processing power by sleeping the thread to match a target tick duration.

3.2 The Entity System

- **`Entity`**: A container for state (`x`, `y`, `z`), visual data (`Shape`), and behavior (`MoveStrategy`). It provides hooks like `hasCollided()` for game-specific logic.
- **`Shape`**: Encapsulates the visual representation (ASCII characters) and their relative coordinates. It contains the logic for checking overlaps (bounding box collision).

3.3 The Game Implementation (`AbstractGame`)

- **`AbstractGame`**: This interface defines the specific rules of a game. It receives a `onTick()` callback from the engine.
- **`Space Invaders` Example**: The `SI_Game` class manages the win/loss conditions and spawning logic, while specific classes like `lazer` and `enemy` configure the generic `Entity` with specific strategies (Linear movement for lasers, Cyclic animation for enemies).

4. Advanced Techniques & C++20 Features

4.1 C++20 Modules

The project utilizes C++20 Modules (`export module entity`, `import shape`) instead of traditional `#include` headers.

- **Benefit:** Faster compilation times and better encapsulation. Modules prevent macro pollution and symbol clashes that are common in large C++ projects.

4.2 Resource Acquisition Is Initialization (RAII)

Memory management is handled automatically via Smart Pointers:

- **`std::unique_ptr`**: Used for `Window` management in `Screen` and `Entity` management in `Model`. This ensures that when the game shuts down or an entity is destroyed (e.g., a laser hits an enemy), the memory is strictly freed without manual `delete` calls.
- **Ownership Semantics**: The `Model` "owns" the entities. The `GameEngine` "owns" the views. This clear ownership hierarchy prevents memory leaks.

4.3 Recursive Strategy Resolution

The `MoveStrategy` class implements recursive velocity calculation:

C++
```
inline int getxvelocity() const {
    if (nextStrategy) {
        return velocity.first + nextStrategy->getxvelocity();
    }
    return velocity.first;
}
```

This allows multiple strategies to contribute to the final movement vector. For example, a "Wind" strategy could add +1 X velocity, while a "Player" strategy adds +1 X velocity, resulting in a net +2 X velocity.

5. Design Evolution: Comparison to Previous Iteration

5.1 From Hardcoded Logic to Strategy Pattern

Previous Design: In early iterations, movement logic was likely hardcoded into the `Entity::update()` method. To create a "ZigZag Enemy," we would have had to inherit `class ZigZagEnemy : public Entity` and override the update method. **Current Design:** We simply create a `ZigZagStrategy` and attach it to a standard `Entity`.

- **Impact:** Drastically reduces code duplication. We can reuse the `GravityStrategy` for a Bird, a falling rock, or a particle effect without creating new Entity subclasses.

5.2 From Direct Ncurses to View Abstraction

Previous Design: Calls to `mvprintw()` were likely scattered inside `Entity::draw()`. **Current Design:** Entities render to an abstract `Window` reference.

- **Impact:** Testing is easier because we can mock the `Window` class. Additionally, we can now support multiple windows (e.g., a Status bar separate from the Game board) easily, as seen in the `StatusViewer` implementation.

5.3 From Raw Pointers to Smart Pointers

Previous Design: `std::vector<Entity*> entities;` requiring manual `delete` in the destructor. **Current Design:** `std::vector<std::unique_ptr<Entity*>> entities;`.

- **Impact:** Exceptions during entity creation or updates no longer cause memory leaks. The lifecycle of an entity is strictly tied to its presence in the Model vector.

5.4 Physics decoupling

Previous Design: Collision checks might have been done inside the specific Game class (e.g., Space Invaders checking bullet positions). **Current Design:** The `Model::updateEntities` generic loop handles O(N^2) collision detection for *any* entity with a collider.

- **Impact:** The game logic (`AbstractGame`) is much cleaner. It focuses on game rules (score, spawning), while the `Model` handles the physics "grunt work."

6. Conclusion

The resulting engine is a robust framework for ASCII games. By leveraging C++20 modules, the Strategy Pattern, and RAII, the codebase is modern, safe, and highly extensible. The strict MVC separation ensures that the engine can grow—perhaps adding sound controllers, graphical views, or network inputs—without destabilizing the core game logic