ASCII Game Engine Design

By: Larris Xie

Introduction

My AGE implementation provides abstractions over core game mechanics to enable users to build complex 2D ASCII games. To demonstrate the engine's capabilities, I created two games: **Flappy Bird** and **Space Invaders**.

Overview

My implementation follows the MVC architecture to separate general engine responsibilities and promote low coupling and high cohesion.

View

The View is a pure abstract class inherited by CursesView. View is composed of a single virtual method, notify(drawables, statusLines), that the model calls to render a frame. This design allows concrete views, like CursesView, to provide their own rendering logic, which in this case is Ncurses.

Notable parts of CursesView's implementation:

- Initiates 2 windows in fixed layouts: 1 for the game and 1 for the statuses
- The game window uses 2 buffers, scratchBuffer (rebuilt each frame) and prevBuffer (the previous frame), to render only rows that differ from the previous frame, minimizing unnecessary redraws
- For RAII, Ncurses WINDOW* instances are owned by a WinPtr (unique_ptr with a custom deleter) and Ncurses lifetime is managed inside CursesView (initscr() in ctor, endwin() in dtor)

Controller

Like View, the Controller is a pure abstract class inherited by CursesController. Controller is composed of a single virtual method, getInput(), that the model calls to retrieve one input event. This design allows concrete controllers, like CursesController, to provide their own input logic, which in this case is Ncurses keyboard input.

Notable parts of CursesController's implementation:

- Returns an InputEvent variant:
    - NoInput when no key is pressed
    - KeyboardInput{key} for key presses
- This design can be easily extended for other types of input (ex. Ncurses mouse input)

Model
The Model is an abstract class that provides the base functionality needed for a MVC-based engine. Model has two private fields: a list of registered views and a pointer to a controller, and provides the central coordination layer to communicate with both.

Notable parts of Model's implementation:

- Supports view registration via addView(), removeView(), and notifyViews()
    - This supports multiple views observing the same model state, following an Observer pattern where the model acts as the subject and views act as observers
- Stores a single controller pointer attached via setController()
- Defines 3 pure virtual methods for subclasses:
    - collectDrawables() to provide render-specific data to the view
    - collectStatus() to provide the status lines
    - run() to start the main loop
- notifyViews() calls collectDrawables() and collectStatus(), then forwards the results to every registered view via View::notify(...)

Engine
Engine is the concrete implementation of Model that clients run. It owns and coordinates the major subsystems of the engine to run a game.

Notable parts of Engine's implementation:

- Owns the subsystems: Clock, World, EventManager, ResourceManager, and SoundSystem
- Provides setGameUpdate(callback) so game code can inject custom logic that runs every tick

- Implements the main game loop in a consistent order for every tick:
    1. Get input from the controller
    2. Run game-specific callback
    3. Update the world (movement, collisions, borders, etc.)
    4. Process events
    5. Notify views to render the frame
    6. Sleep to maintain a constant refresh rate

First let's explore the clock subsystem of the engine:
Clock
Clock encapsulates timekeeping and frame refresh rate for the engine. It provides information about how long the last tick took and a sleep method that helps the engine maintain a consistent refresh rate.

Notable parts of Clock's implementation:

- Stores a configurable tick duration which represents the refresh rate (default is 20 FPS for a tick every 0.05 seconds)

- sleepUntilNextTick() sleeps for the remaining time in the current tick (if any), preventing the game loop from running as fast as possible and keeping the simulation rate predictable

Now let's explore the main subsystem of the engine:
World
World encapsulates the simulation state of the game. It owns the set of entities, enforces border rules, detects collisions, and provides the data needed by the engine to render the entities and display status information.

Notable parts of World's implementation:

- Updates all entities via World::update(input), where each entity gets
    - Its own Entity::update(input) method called
    - Collisions detected using Hitbox intersection and same height (z-layer) checks
    - Border rules applied (e.g. if entity can leave the borders), including a default offscreen despawn policy (10 ticks fully offscreen)
    - Removed if it is no longer alive

- Collects Drawables for rendering and sorts them by z so higher objects draw on top

- Stores status lines for the view and exposes helpers to add or replace them

Entity
Entity is the object abstraction. Each entity has identity and spatial state, a hitbox for collision detection, a base shape for rendering (optionally animated), and a set of movement components that determine its behavior each tick.

Notable parts of Entity's implementation:

- Entity::update(input)
    - Applies each MovementComponent instance to update its position
    - Applies the next Animation frame (optional)
    - Increases the ageTicks by 1, which is used for despawn logic

- Contains id and tag fields so game code can label objects and react based on ids (for specific instances) or tags (e.g. "player", "pipe", "ship")

- Stores a Hitbox and a height value, which is used for draw ordering (pass-through behavior between different z-layers)

- Provides setOnCollision(callback) so game code can inject custom entity-based logic responding to collisions

- Stores a Solidity (Solid, Trigger, Ghost) that lets World::handleCollisions() detect intended collisions

MovementComponent
MovementComponent is a pure abstract class that is composed of a single virtual method, apply(entity, input). By moving movement logic into separate components, how an object moves is decoupled from the Entity itself (which improves reusability), and allows an Entity to hold multiple movement behaviours at once without being hard-coded.

The concrete implementations support the AGE requirements:

- StraightMovement: constant velocity in a straight-line path

- GravityMovement: constant downward velocity

- CycleMovement: periodic sequence of *k* forms applied over time
- PlayerControlledMovement: keyboard-based movement based on configurable key bindings

Now let’s explore the resources subsystem of the engine (also relevant to Entity):
Resources (Shape, Drawable, ResourceManager)
The rendering system is intentionally decoupled from the view, so the model only needs to provide lightweight Drawables to the view to render the entities.

Notable parts of the resource system:

- Shape stores a sprite id and a 2D vector of characters, and exposes at(row, col) to read a specific pixel
- Drawable is a lightweight class containing (shape*, x, y, z) and is used by the view to render without knowing about entities
- ResourceManager owns shapes (unique_ptr) and returns non-owning pointers so many entities can share the same Shape

Now let’s explore the events subsystem of the engine:
Events (EventManager, Event)
The event system follows an Observer pattern: EventManager acts as a central subject that components can subscribe to, and subscribers are notified via callbacks when relevant events are dispatched. This keeps subsystems decoupled while still allowing game logic to react to built-in events (e.g. CollisionEvent, BorderEvent, SoundEvent, GameOverEvent), as well as game-specific events by defining new Event subclasses and emitting them through the same EventManager interface.

Notable parts of the event system:

- Events inherit from a common pure abstract base class, where each event has a string type() discriminator used for subscription matching
- EventManager supports subscribe(type, callback), returning an id that can later be used for unsubscribe(...)
- Events are queued via emit(...) and dispatched in batches (callback is run) every tick via processEvents()

Now let’s explore the sound subsystem of the engine, which is my enhancement:
Audio (SoundSystem)
Sound is implemented as a subsystem behind an abstract interface so games do not depend directly on a specific audio backend. Game code can request sounds via SoundEvents, and the engine abstracts away the logic for playing them through the configured sound backend.

Notable parts of the sound system:

- Pure abstract class SoundSystem is composed of virtual methods for playing, stopping, and mute control, and Engine holds it as a unique_ptr<SoundSystem> so users can swap concrete backends at runtime via polymorphism
- Multiple backends are supported (Ncurses terminal beep(), SDL playback, and a null backend for silence/testing)
- The engine subscribes to SoundEvent and plays sounds by id, keeping “play sound” as a decoupled side effect
- In the SDL backend, SoundClip maintains RAII for Mix_Chunk* by freeing the audio resource in its destructor, similar to how CursesView manages WINDOW* with WinPtr

Updated UML

- **Model/View Observer relationship**: the original UML modeled a single View* with notifyView(). The updated UML stores vector<View*> and provides addView/removeView/notifyViews(), making the model a subject that can broadcast updates to multiple observers (Observer pattern)
- **Model/Engine responsibilities**: the updated UML includes collectDrawables() and collectStatus() to clearly show how the model produces the rendering info for notifyViews(), which I previously excluded as private methods
- **InputEvent simplification**: the updated UML excludes MouseInput in InputEvent, which the AGE doesn't need
- **Engine simplification**: the updated UML consolidates execution (previously through start(), go(), end()) into run() and adds a GameUpdateCallback so games can inject per-tick logic
- **World simplification**: the updated UML simplifies World::update(input) and stores a non-owning EventManager* inside World, reducing excessive parameter passing
- **BorderMode, SoundBackend, Solidity enums**: instead of strings, the updated UML introduces multiple enums for stronger typing and clearer intent
- **World/Entity decoupling**: the original UML tightly coupled movement/collision methods to World. The updated UML makes Entity::update(input) self-contained and reacts to collisions at the Entity level via CollisionCallback
- **Entity RAII**: MovementComponent and animation ownership becomes explicit through unique_ptrs
- **Frame decoupling**: the original UML had Frame store a spriteId thus was coupled with ResourceManager (missing in the UML). The updated UML has Frame reference a non-owning Shape* directly
- **ResourceManager cohesion**: the original UML placed both shapes and sounds into ResourceManager, but the updated UML focuses

ResourceManager on shapes, with sound assets moved to the sound subsystem instead of being a generic “resource”

- **EventManager redesign**: the original UML had GameEvent and EventListener, with EventManager holding listeners. The updated UML introduces Event as an abstract class and merges EventListener into EventManager, reducing unnecessary complexity
- **SoundSystem redesign**: instead of making SoundSystem an EventListener coupled to ResourceManager, the updated UML removes that inheritance and instead has the engine subscribe to SoundEvent and be its own subsystem of Engine
- **CursesView RAII**: instead of storing raw WINDOW*, the updated UML stores windows in WinPtr to enforce RAII

Design

Many design choices were discussed during the overview so here are the most significant patterns used in my implementation:

- MVC
    - Separation between Model/Engine (game state + orchestration), View/CursesView (rendering), and Controller/CursesController (user input)
- Observer Pattern
    - Model maintains vector of views and notifies all registered views via notifyViews()
    - EventManager maintains callback subscriptions: engine systems observe events by subscribing, and the world produces notifications by emitting events that are later dispatched
- Strategy Pattern
    - Movement behaviors are encapsulated as strategies through MovementComponent::apply(...) that can be swapped/combined per entity
- Command Pattern
    - InputEvent is an extensible variant of input commands which allows the controller to return a single typed result that the engine/world can interpret without depending on Ncurses details
- Factory Pattern
    - createSoundSystem(SoundBackend) is a factory function that takes an enum value and returns the appropriate concrete SoundSystem implementation (TerminalSoundSystem, SDLSoundSystem, or NullSoundSystem) behind the abstract interface. This centralizes object creation and hides the concrete types from client code

Overall, my implementation applies SOLID principles where appropriate, achieving low coupling and high cohesion. Strong OOP practices are demonstrated by the use of abstract base classes and their concrete implementations.

Extra Credit Features

RAII

The program runs without leaks (checked through Valgrind, leaks ONLY through Ncurses), and all memory management was handled via vectors and smart pointers. For external libraries, as mentioned in the overview, Ncurses WINDOW* is managed by an unique_ptr + custom deleter (WinPtr) and SDL_mixer Mix_Chunk* is wrapped by SoundClip with a dtor that frees the chunk. Thus, there are no delete statements in my program.

SoundSystem

The sound system is abstracted in the engine so games do not depend directly on a specific audio backend. Game code can request sounds via SoundEvents (abstracting sound as events through EventManager), so the engine simply subscribes to SoundEvents and calls sound->play(soundId), which works for ANY configured sound backend.

The pure abstract class SoundSystem is composed of virtual methods for playing, stopping, and mute control, and Engine holds it as a unique_ptr<SoundSystem> so users can swap concrete backends at runtime via polymorphism

Multiple backends are supported:

- TerminalSoundSystem: minimal audio supported by Ncurses using beep(), default for AGE as it is always compatible with CursesView\
- SDLSoundSystem: real audio playback via SDL_mixer, storing loaded sounds in a map by id
- NullSoundSystem: silent backend for silence/testing

The two main challenges were decoupling the sound system from game logic and implementing RAII for audio clips for the SDL backend. I originally had the game call methods directly from SoundSystem, but I solved the coupling by routing audio requests to the event system, removing the need for engine specific implementations. The second I solved through SoundClip as mentioned above in the RAII section.