AGE Engine Project - Final Design Document

Brook Khoo

Introduction

This project implements a modular and extensible terminal-based game engine, the AGE engine, supporting multiple game demos built on top of a common framework. The engine abstracts away input parsing, rendering, movement, and collision logic, allowing game-specific behavior, such as for Snake and Pong, to be implemented by composing entities with modular behaviors.

The initial design I proposed on Due Date 1 focused on separating entities, movement, and collision logic, but these ideas have been polished in the final implementation. Particularly, several class hierarchies and relationships were expanded, and responsibilities were more clearly divided between the engine, board, game state, and entity systems. This design document describes the final design, how it was developed, and the reasoning behind the architectural decisions.

Overview

At a high level, the project is structured around a GameEngine class that coordinates four major subsystems:

1. GameBoard - maintains the physical environment, the entities it contains, as well as their interactions with each other.
2. Entity system - represents the in-game objects (players, enemies, obstacles, etc..) as well as their characteristics.
3. Input / View - abstracts away the ncurses library input handling and rendering.
4. Behavior systems – controls the movement and collision logic of the entities and the game environment.

The engine has a main "run" loop that continuously:

1. Reads player input.
2. Advances game state by one "tick".
3. Updates entities and resolves movement and collisions.
4. Draws the game world and entities.
5. Ends the game if a certain condition is fulfilled.

Instead of having behavior and logic hardcoded into entities or the game world, most logic is implemented through composable objects (Movement and CollisionBehavior), which allows new behaviors to be added and removed without modifying the existing entities. Now taking a look at each system in more detail:

Entity System

The Entity class represents any object that can exist inside the game environment.

Examples include the snake head, segments, and apples found in the Snake demo, as well as the pong paddles and ball found in the Pong demo.

Entity objects are also intended to be lightweight. Instead of burying movement or collision logic directly inside it, an entity:

- Has a list of Movement objects that allow it to combine different types of movement.
- Has a list of CollisionBehavior objects that allow it to combine different types of collisions.
- Delegates behavior of it's movement and collision during each update tick.

This design allows entities to be composed of reusable and stackable behaviors, in order to achieve more complex and flexible functionality.

Key responsibilities of Entity include:

- Managing attached movements.
- Responding to collisions.
- Tracking characteristics like damage cooldowns and ticks.
- Coordinating updates with updateSelf() and updateMovements().

Movement System

Movement logic is implemented using the Strategy pattern. Each movement type is derived from the abstract Movement base class and has an apply() method.

Examples include:

- StraightMovement - moving in a straight line.
- CyclingMovement – switching through a number of visual forms, while staying stationary.
- GravityMovement – gravitating towards one of the borders of the game world.
- PlayerMovement – movement controlled by the player.
- SnakeMovement – snake-specific motion and growth behaviour for the Snake game demo.

As mentioned previously, Entities can have multiple Movements stacking at the same time, which allows for more complex behavior (like player-controlled movement plus gravity movement) to be used through composition instead of hard coded logic.

This architecture is different from the original design plan, which assumed a single movement per entity. Supporting multiple movements required refactoring but resulted in a more modular system.

Collision System

Like Movement, collisions are very modular and stackable. They are handled through the CollisionBehavior class hierarchy.

Each CollisionBehavior subclass type decides how an entity reacts when:

- colliding with another entity: handle()
- colliding with a solid game border: onBorder()

They include:

- BounceCollision – entity bounces off a collision.
- DamageCollision – entity is damaged and eventually destroyed. - DestroySelfCollision – the colliding entity is destroyed on collision.
- DestroyBothCollision – both colliding entities are destroyed on collision.
- WinCollision / LossCollision – win or loss condition is triggered on collision.
- EatAppleCollision – Snake game demo collision to facilitate snake growth logic.
- SnakeHeadCollision – Snake game demo collision to ensure game ends if snake hits itself or a game border.

The CompositeCollision class is special because it applies multiple collision behaviors in a row, allowing complex interactions, like damage and bounce collision.

This design replaced the earlier collision model that had the logic inside the game board instead. Moving collision responses into separate classes improved modularity and the potential for more complex behaviors without refactoring.

GameBoard

The GameBoard represents the physical game environment. It:

- Owns and handles all entities.
- Pushes the simulation by one "tick".
- Checks for border collisions.
- Checks for collisions between entities.

The GameBoard doesn't decide what collisions do; it only detects them. Collision response is delegated back to each entity's CollisionBehavior. This separation ensures that spatial logic and gameplay logic remain decoupled, as per the Single Responsibility Principle.

GameEngine

GameEngine acts as the main controller of the ecosystem. It handles:

- Input parsing.
- Simulation steps.
- Rendering.
- Game termination.

Key responsibilities include:

- Running the main loop: run().
- Advancing the simulation: step().
- Drawing the current state: draw().
- Defining status variables and borders.

- Ending the game with win/loss conditions: endGame().

Unlike the initial design, which had none, the final engine exposes more explicit lifecycle methods like step() and endGame(), which made debugging and reasoning about control flow easier.

Input and View Abstraction

Input and rendering are abstracted through abstract Input and View interfaces. This allows ncurses-specific classes (CursesInput and CursesView) to be isolated from the rest of the engine. These classes take care of a lot of boilerplate related to input and rendering, like parsing individual keystrokes, waiting and timing user input, setting up input streams to receive user input, printing individual characters, and printing specific elements in the game environment.

The engine depends only on the abstract interfaces, which improves modularity and the potential for new features or systems.

Update UML

The updated UML included with this submission reflects the class hierarchy found in the final implementation of the AGE engine. It includes all concrete and abstract classes as well as their public methods (excluding the Big 5, constructors/accessors/mutators, and private methods/fields).

Key points visible in the UML:

- Entity composes Movement and CollisionBehavior objects.
- GameBoard owns all entities and oversees their physical interactions.
- GameEngine oversees the system and low-level boilerplate logic.
- Input and View are abstracted behind interfaces, with ncurses-specific classes.
- Some game-specific logic like SnakeHead are subclasses of Entity.

Please see the attached UML diagram for more details.

Design

There were two main design challenges present in this project: implementing robust collision detection and composing multiple movement types.

One of the initial challenges was to accommodate the multiple movement specification in the project guidelines. An early idea involved hardcoding movement combinations, such as straight movement and gravity, and calling those directly. However, the final solution was to use the Strategy pattern, with an abstract Movement superclass, and multiple concrete implementations (movement.h, movement.cc). Entities could own multiple subclass movement strategies simultaneously, which were applied sequentially during each update step. This design followed the Open/Closed Principle, as new movement types could be added without changing existing code, and the Single Responsibility Principle, as each movement implemented exactly one form of motion.

Another challenge was implementing robust collision detection. This was difficult because there are many stages in resolving a collision. Firstly, the GameBoard must check if a collision has happened, by iterating over each entity in the world, and then iterating over the Pixels of each entity to determine if they are overlapping with another entity. If a collision is detected, then it must be resolved according to the type of collision the entity has. Bounce collisions combined with straight movement require determining the axis on which the collision is happening, and reversing the corresponding velocity component. Bounce collisions combined with gravity movement require temporarily reversing gravity, and then reverting it to simulate bouncing on a surface. Other collision types, such as solid collisions, require keeping an entity's position paused, while win/loss collisions must update the GameState accordingly.

There was a lot of fine coordinate-based logic to deal with, and many bugs to trawl through, but the end result was a robust collision system that could handle most situations, and also most uncommon ones.

Extra Credit Features

One feature implemented beyond the project requirements is collision stacking. During testing it became clear that there were a lot of situations in which having multiple collisions would be useful, such as an entity bouncing off another one while damaging it.

In a naïve design, collision handling is often implemented as a single conditional: when two entities collide, the engine decides what happens and applies an outcome. This approach breaks down in several situations requiring multiple responses to the same collision, such as a snake head eating an apple and growing, while the apple is destroyed.

Attempting to encode collision combinations inside the game board or entity logic would have led to large conditionals, duplicated logic, and inefficient refactoring when trying to extend the system later.

Another challenge was ordering. When multiple collisions occur, it is difficult to determine in which order they should occur, for example damage before destruction, or bounce before movement resolution.

To fix these issues, collision handling was redesigned using both the Strategy and Composite design patterns. Firstly, the system had the abstract Collision-Behavior interface, which defined two functionalities:

- handle(self, other, state): for entity-on-entity collisions.
- onBorder(self, state): for entity-on-border collisions.

Each CollisionBehavior subclass implements exactly one of these. To support stacking, another subclass, CompositeCollision, was written, which implements

CollisionBehavior but also stores a collection of other collision behaviors. When a collision occurs, CompositeCollision simply forwards the collision event to each contained behavior in sequence.

Because CompositeCollision is compatible with the same interface as all other collision behaviors, entities can treat it exactly like a single collision strategy. This preserves substitutability and avoids special-case logic in the engine or board.

Final Question

What would you have done differently if you had the chance to start over?

If I were to do a single thing differently, if building this project all over again, I would definitely start working on collisions a lot sooner. Initially, I made my plan of attack with 6 phases of development, and I had hoped to begin working on collisions in Phase 4, which was about a week off the due date. However, while I stuck to the plan for the most part, I severely underestimated how much work it would be to fine tune all of the different edge cases that come from a variety of collision types. I had to fix a ton of bugs related to directional bouncing, clipping, what happens if an entity is moving too fast, etc.. It felt like each bug made 3 new ones, and by the time I was deep into the work, the due date for the project was a couple days away, which added some stress. Also, struggling to get collisions working meant that while the code worked, it was not as readable as I would have liked it to be, and so I was forced to move on to finishing up the other functionality, rather than make it more readable. Hence, I would start with collisions earlier if I were to do this again.

Conclusion

The final AGE Engine architecture prioritizes modularity, composition, and clear separation of responsibilities. The game engine handles control flow, the game board manages physical logic, and entities achieve complex behaviors through stackable movement and collision strategies. This design avoids complex conditionals and monolithic architecture in favor of behavior-driven components.

Although the design evolved slightly from the initial plan, the final structure is more complex, versatile, and modular. The updated UML also accurately reflects the implemented system and, along with this final design document, provides a complete and self-contained description of the project.