CS246E Final Project Design Document

Yichen Liu

December 2025

1 Overview

The overarching architecture of the game engine follows the MVC pattern, where the model is the game world (`World`) which owns a collection of game objects that constitute the current game state, and is composed of a collection of systems that operate on game objects and are each responsible for a single type of interaction. Every 0.05 seconds, the world calls the `update()` method of each system, which progresses the game state by one step. The core design of the AGE engine consists of three categories of objects which make up the game state and interactions.

1.1 Game Objects

Game objects (`GameObject`) are used to represent game entitities such as players, obstacles, or even abstract elements of a game such as game state and logic. They are the things that interactions and behaviours of the game are defined on. Fundamentally, they are containers for components which define the internal state of a game object, the types of events that a game object responds to, and the specific behaviour of a game object in response to certain events. This is elaborated on in the next section.

1.2 Components

Components (`Component`) define the data and behaviours of a game object. They can expose data as object fields, such as transform or velocity, and they can hook into certain events by exposing public methods which can be called by systems to perform some operation on their root game object. This is used to simulate things like collision, movement, or input handling. The handling of behaviours using systems is elaborated on in the next section.

It is very common that a component with one responsibility will need to access or modify a component with another responsibility. For example, a movement component may depend on an input handler component in order to implement player-controlled movement. To achieve this, the parent game object exposes a `getComponent()` method that allows components to access other components through their parent game object.

1.3 Systems

Systems (`MovementSystem`, `CollisionSystem`, etc.) are objects inherent to the game world which orchestrate the interactions and behaviours of game objects. Each system is responsible for a single type of interaction, e.g. the movement system is responsible for handling movement, and the collision system is responsible for handling collisions. Since the actual behaviours are defined within components, systems are mainly responsible for checking preconditions, and executing component behaviours.

2 Design

2.1 Game Objects and Components

The main difficulty of designing a game object representation is to accomodate the inclusion or exclusion of an arbitrary collection of states and behaviours. For example, a game object could have any combination of behaviours such as movement, collision, or response to input, and internal states such as position, or velocity, as specified by the game programmer. This is achieved using the **Composite Pattern** where `GameObject`s are implemented as composites, and `Component`s are implemented as components. Then, the game programmer can simply attach components to game objects to define behaviours and interactions.

2.2 Systems

To simplify dependencies, game objects should not directly depend on the overall game state, or on other game objects in the scene. However, some interactions require dependencies between game objects, such as collision. Furthermore, interactions should proceed in a defined order, e.g. movement should be handled before collisions, etc. This creates the need for systems which provide the interactions between game objects, each of which handles a kind of interaction for all game objects. Fundamentally, systems mimic the **Visitor Pattern** in the way they iterate and perform operations over game objects in their `update()` method. However, the logic to discern different kinds of game objects live in systems, since `GameObject` does not inherently know if it supports a type of interaction, but it can be queried for a component which supports the interaction. The pseudocode for an example system is given below for illustration:

```
class System{
    update(gameObjects) {
        for(gameObject : gameObjects){
            Component* component = gameObject->getComponent<Component>();
            if(component){
                component->executeBehaviour();
            }
        }
    }
};
```

2.3 Transform and Movement

`Transform` and `Movement` are the basic components that define the position and movement of a game object. They store this data using the `Vector3D` class, which defines convenient operations on vectors such as addition, scalar multiplication, and dot product. The only behaviour implemented in either of these components is the `move()` method of `Movement`, which just adds velocity to position, and acceleration to velocity. Theoretically, a client could extend movement behaviour by inheriting from `Movement`, and overriding the `move()` method, which would make it an example of the **Strategy Pattern**.

2.4 Collision

`Collider` and `CollisionBehaviour` are abstract components which provide the interface for defining collisions.

The `Collider` component exposes the `intersects()` method, which is a predicate that checks for intersections with other `Colliders`. The game engine provides one concrete subclass of `Collider`, `BoxCollider`, which specifies the dimensions of a box, and detects intersections with other `BoxColliders` using `Transform` and calculating overlap.

The `CollisionBehaviour` component exposes the `collide()` method, which defines how the game object responds to collisions. The game engine provides two concrete subclasses of `CollisionBehaviour`, `StopCollisionBehaviour` and `BounceCollisionBehaviour`.

Since the game programmer may wish to specify other behaviour on collision, such as triggering a win, or other collider shapes, `intersects()` `collide()` can both be overridden to implement custom collisions, making it an example of the **Strategy Pattern**. However, to extend `Collider`, the client must manually specify intersection logic with other types of colliders.

Collisions are handled by `CollisionSystem`, which takes base `Collider` and `CollisionBehaviour` pointers and relies on polymorphism to support custom collisions.

2.5 Graphics

The way that a game object is displayed in the view is defined by the `Sprite` component. The `Sprite` component contains a texture field, which is either a character, a `BoxTexture` which defines the dimensions and character fill of a box, or `BitmapTexture` which defines a collection of `Pixel`s. Since game objects may reuse textures, sprites do not own their textures.

Since game objects shouldnt directly modify the game state, sprites have no behaviour, they are pure data. Instead, the way that sprites and text are drawn is defined by the `View` class, which exposes the `drawSprite()` and `drawText()`. Views may be responsible for drawing both sprites and text, since they may be drawn to the same area and the view needs to organize them accordingly (such is the case in curses). Classes can inherit from `View` to define their own behaviour for drawing sprites and text. The engine provides one concrete subclass of `View`, `CursesView`, which is the view described in the project guidelines. `CursesView` uses the texture field of a sprite to determine which pixels on the screen should be filled by which ASCII character.

Drawing to views is handled by `GraphicsSystem`, which iterates through all views and game objects, and given a sprite and transform, calls `drawSprite()`.

`CursesView` is also responsible for drawing text to the screen, but the game may need to print text to the view in response to arbitrary interactions between game objects. Since game objects should not interact with systems directly. This creates the need for `TextStatus`, which is part of the world, and represents the text in the view. game objects are given a reference to `TextStatus` when they are added to the world, and may add text to `TextStatus`. `GraphicsSystem` also keeps a reference to `TextStatus`, and forces views to mirror `TextStatus`.

2.6 Input

input handling is defined by the abstract `InputHandler` component, which exposes a `handleInput()` method, which takes a character representing the input as an argument. Since input handling is highly customizable, the game engine provides no concrete subclasses of `InputHandler`. The game programmer is expected to provide input handling logic by overriding `handleInput()`, making it an example of the **Strategy Pattern**.

Input is queried through a `Controller`, which exposes the `getInput()` method. on each pass, the input system queries the controller for input, then forwards the input to all input handlers. The game engine provides one concrete subclass of `Controller`, `CursesController`. Since the game only responds to one keystroke per tick, `CursesController::getInput()` just calls curses `getch()`.

2.7 Events

The client needs a way to specify arbitrary game logic if the game engine wants to support a wide variety of games. However, all behaviours and interactions of

game objects are self-contained, and game objects do not depend on the overall game state. Therefore, the game programmer needs a way to specify game logic directly in the game object itself.

To support this, game objects expose an `emit()` method that emits an arbitrary event specified by the game programmer and represented by a string, which is forwarded to all game objects. The game engine provides the abstract `EventHandler` component, which exposes a `handleEvent()` method to respond to events. Like `InputHandler`, the game programmer is expected to provide event handling logic by overriding `handleEvent()`. This makes it an example of the **Strategy Pattern**.

However, there needs to be a way for events to propagate from one game object to other game objects, yet game objects have no references to each other. This creates the need for `EventQueue`, which is part of the world, and holds a queue for pending events. Game objects are given a reference to `EventQueue` when they are added to the world, and `emit()` pushes new events onto the queue. It is possible and allowed for game objects to emit new events during the event handling stage.

Game objects receive events through `EventSystem`, which, similarly to `InputSystem` reads from the `EventQueue` and forwards each event to all `EventHandler`s. The combination of EventSystem and EventQueue coordinates communication between `EventHandler`s.

2.8 State

The `State` component assists the specification of game logic, acting as a container for arbitrary data attached to a game object. This can be used to model health, score, etc. Data can be retrieved from a state with a string, representing a variable name, and using `getData()` and `setData()`.

2.9 Spawning

Game objects may need to spawn other game objects to mimic behaviour such as shooting bullets or splitting. This requires game objects to add new game objects to the world, but game objects are unaware of the overall game state. Instead, spawning is handled by the `Spawner` component, which maintains a queue of game objects to be spawned during the spawning stage. `Spawner` exposes the `queueSpawn()` method, which takes a game object, and queues it for spawning. To spawn a game object, the parent game object needs to construct a new game object and pass it into `queueSpawn`

In the context of spawning, it is common that a game object may need to spawn a copy of another game object or itself, retaining its current data. However game objects almost always own a variety of components. Components are mostly accessed through the base class `Component` and frequently inherit from each other, meaning we run into the problem of polymorphic cloning. To address this, `Component` uses the **Prototype Pattern** and exposes an explicity and overridable `clone()` method which produces a copy of itself. Since

components are largely simple objects, most of them are already copyable by default. Therefore, the `ClonableComponent<Base, Derived>` template class is used to support cloning using CRTP. `ClonableComponent<Base, Derived>` inherits from `Base` and overrides `clone()` by copy constructing `Derived`. Then, all instantiable component subclasses inherit from `ClonableComponent`. Game object also exposes a `clone()` method, which constructs a new game object and copies over all components.

To support spawning, game objects and systems also need a way to add new game objects to the world, but to simplify dependencies, neither should depend on `World` directly. Instead `GameObjectManager` is part of the world, owns game objects, and provides the interface to add and remove game objects.

Then, spawning is handled by `SpawnSystem` which keeps a reference to `GameObjectManager`. during the spawning stage `SpawnSystem`, reads from all `Spawners` and adds the new game objects to `GameObjectManager`.

2.10 Main Game Loop

All of these elements come together in the main game loop, defined in the update method, which simply calls each of the systems in sequence. The game loop is structured this way to allow for a defined order of operations.

```
void World::update(){
    inputSystem.update(getGameObjects());
    movementSystem.update(getGameObjects());
    collisionSystem.update(getGameObjects());
    // Game programmer may specify custom logic during each tick here
    for(const auto &gameObject : getGameObjects()){
        gameObject->update();
    }
    timerSystem.update(getGameObjects());
    spawnSystem.update(gameObjectManager);
    eventSystem.update(getGameObjects());
    graphicsSystem.update(getGameObjects());
    // Destroyed game objects are removed
    for(auto it = getGameObjects().begin(); it != getGameObjects().end();){
        auto &gameObject = *it;
        if(gameObject->isDestroyed()){
            removeGameObject(gameObject.get());
        }
        else ++it;
    }
}
```

3 Final Question

In terms of my design, I think I would keep the core separation between game objects and systems that I have in my current iteration of the game engine, but there are design decisions that I would reconsider. As mentioned earlier in this document, systems essentially follow the visitor pattern in their core functionality, and there is a lot of unnecessary duplication in the current implementation of systems. Furthermore, the current implementation of systems requires a new system to be created and attached to world for every new component supported by the game engine which violates the open-closed principle. I would want to refactor systems to conform more closely to the visitor pattern, which would allow me to split them into smaller classes and make it easier to extend them. Then, `World` could simply keep a list of visitors, and to add new components, the base visitor would only need to add a new method to support the new component.

There are also parts of the design that suffer from indirect information flow and contribute to global state, which could benefit from being flattened out. The most prominent examples are `TextStatus` and `EventQueue`, which force game objects and systems to depend on a concrete object whose responsibility is not clearly defined. Furthermore, to send events or print to the view, game objects need to write to `TextStatus` and `EventQueue`, which are read by systems, obscuring the coupling between these two objects. An alternative solution could be to handle printing and events within each game object, in a similar way to how spawner handles spawning, although this adds responsibilities to game objects where it is unclear if it should have them. Furthermore, it becomes possible for events and messages to arrive out of order.

In terms of my approach, there are a number of things I would change which I believe would have allowed me to write more organized code in less time. One thing that hindered development was trying to solve the entire architecture before clearly understanding the project requirements, and dependencies between objects. Although it was helpful to plan out the project ahead of time, I spent my time inefficiently, thinking about the optimal design before writing any code. I found that I ended up missing crucial details anyway, and that these details only became clear after I actually tried to implement the design. It was the process of writing code where I learned most about the requirements and dependencies, which then allowed me to think of a good design.

In a similar vein, the project's UML diagram was underutilized during development. As the project grew in complexity, I found that it became difficult to keep an accurate model of the dependencies and responsibilities of each object in my head. Having an updated reference of the relationship between objects would have helped me to find redundant dependencies and potential abstractions.