Age Design Document

Edwin Sun (e35sun), for CS246E 2023

Introduction

In this document, I will be explaining my implementation of the AGE game engine. The AGE engine that I will be presenting supports all of the required functionalities outlined on the document, including all the types of object shapes (single character, rectangle, bitmap), all the types of movement (straight line, stationary-cyclic, gravitating towards a border, player-controlled), and any arbitrary combination of the above. The design was heavily influenced by the techniques demonstrated in class and consists of good OOP practice and design patterns. There is sufficient in-line documentation within key points in the code and the design makes full use of separate compilation.

Overview

This engine is designed with the MVC (Model-View-Controller) pattern which connects the main components of the engine together. The model, view, and controller are contained within the main game engine class, and they communicate with each other through the Observer pattern.

Model

The model contained the vast majority of the logic and objects associated with the engine. I had the model be both an observer and a subject so that the model would observe the controller and be a subject for the view. The model observes the controller for any new user inputs and makes updates and changes accordingly. The model is also a subject because when it is done processing the current tick, it notifies the view to update the screen for the user.

View

The view does not make any changes to the game logic itself and is effectively decoupled from the model and the logic of the game engine. The view was made to observe the model because the model is constantly changing every tick and the view is dependent upon the model to decide what is displayed to the user through Ncurses.

Controller

The controller was made as a subject in which the model observes. The controller handles all the user input through the Ncurses library but is not allowed to access any of the logic of the game engine. This effectively achieves decoupling between the model and the view, because now the view only has one job which is to get input and when it receives input, it automatically gives it to its observer, the model.

Updated UML

Please see the attached UML document in Marmoset.

Design

Board

The model was made as an abstract class which had the board inherited from it. This was done because if another class that handles logic were to be added to the engine, then it could easily inherit from the abstract model class. On the contrary, if there was only the board class and no abstract model class, then any additional logic-pertaining classes would not be able to comfortably integrate into the engine. The board, which is inherited from the abstract model class, contained all the objects, a map that maps out the collisions, the status lines for the status bar, and the win/lose conditions. The collisions map, status lines, and the win/lose conditions were added on top of the Dute Date 1 design. I realized that all the collisions had to operate on the (pairs of) objects themselves, and the objects were exactly located in the board and having a separate class to handle the collisions would lead to the coupling of it and the board class since it would need to modify the objects in board. Similarly, the status lines and win/lose conditions were also kept in the board class. I also consolidated the handling of movement and handling of collisions into one function in the board class which processes during each tick. This way, the board is solely responsible for processing the current tick and all game engine needs to do is to call board to process the current tick.

Objects

I made the object class an abstract class because it does not make sense for it to be initialized on its own since all the objects were either a basic object, a rectangular object, or bitmap object. The object class also uses the Interface Segregation Principle because each object needs to be drawable, and it also needs to be collidable. In order for the object to have these functionalities, I

used multiple inheritance to have the object class inherit from both of the above functionalities. In addition, all of object's derived classes fully followed the Liskov Substitution Principle because each object subclass had implemented its own drawing function and was fully able to substitute the object class.

Status Lines

I implemented three different status types which can be used for the status lines. This was different from my Due Date 1 design because the abstraction I was thinking of before was not ideal. Before, I was thinking of having the client define and have access to any of the fields in the board (model) class in order for any one of them to be part of a status line. However, this abstraction severely violates the encapsulation of the board class because the client should not have access to the detailed intricacies and private data structures within the board class. Thus, I provided three options that have a wide range of possibilities. For example, I allowed for the user to have a status based on the value of an attribute in the objects and this is a much better solution in terms of encapsulation because it uses a getter function provided by the board which only allows the status class to get the value of a specific attribute associated to a specific object within the board. This means that the status class can have access to many different attributes (such as health and damage etc.) of the objects to use as statuses, but at the same time, it will not be able to access the other private entities of the board.

Win/Lose Conditions

In the Due Date 1 design, I only had the win-lose conditions originating from collisions. However, I was able to add an additional pair of win-lose conditions, inheriting from an abstract win-lose condition class, and located within the board class. I did this because the original win-lose conditions from the collisions were not a very good abstraction, and it did not provide the client with much functionality in terms of end game possibilities. With the new win-lose conditions, however, the client is able to define their end game conditions based on object attributes and the count of a certain type of object remaining, both of which make use of constant getter functions provided by the board class, in order to maintain encapsulation.

Object Spawning from Collisions and Movements

A challenge I was faced with was how to enable the addition/spawning of objects which originate from collisions or movements. Before this, each of the collision and movement classes had their own overridden functions to move an object and process the collision of two objects, respectively. However, this was not sufficient for adding a new object because within the movement and collision classes, I had no access to the board, which was essential for adding objects. To solve this problem, I extended two new subclasses from the collision and movement classes which were responsible for the event of adding objects. These new subclasses also inherited from the subject class, utilizing the Observer pattern. These classes would then add the board as an observer. This way, whenever any new object(s) needs to be added as a result of a collision or movement, the board can be notified and add the objects accordingly.

Collision

The majority of the Due Date 1 design for collision had remained unchanged, with the classes of each type of collision having to implement their own collision-handling function for two arbitrary objects. However, there was an issue with the bounce-off collision which features the reversing of the movements of both objects that are part of the collision. The issue came in reversing the movements because there were several types of movements each with their own distinct ways of moving. Thus, I had the abstract movement class inherit the reversible interface so that each of movement's subclasses are required to implement the action of reversing the direction of its own type of movement.

Although I had considered all the possible types of collisions in the Due Date 1 Design, I did not factor in how the actual collision between two arbitrary objects was going to be checked. This is because the types of collisions I had (which also were the ones outlined in the pdf) were technically all end-results of collisions. Before that, it was required to check whether two objects intersected or not. Since there are many different types of objects capable of occupying different magnitudes of space, and the object class already had an interface for being drawable, I used the Interface Segregation Principle to add the additional interface "collidable", which required each object to implement a function that returns a vector of all the points it occupied. This way, I can easily find whether two objects collide by using the vectors they return from collidable.

Movement

The movement was handled with the implementation of an abstract movement class, with all the possible types of movements inheriting from it and overriding the pure virtual method "updatePosition" which is used to update the position of a given object. Each movement is stored inside the object that it is associated with, and each object calls its own movements, in order, to update its own position during every tick. The challenging part was coming up with an abstraction for the movement subclass that handled stationary, cyclic movements. In the design of Due Date 1, I had specified the data structure used to store the cyclic movements as a vector of integer pairs. However, this was insufficient because the abstraction needed to be applicable with all types of objects, especially the bitmap object. This is because the symbols that compose a bitmap object can be arbitrary and so a vector of integer pairs to store the state of the bitmap object during each cycle was insufficient. Therefore, I changed the data structure used to store cyclic movements to a vector of a vector of tuples. This way, each cycle of the object could be stored in the inner vector, and the tuples allowed for objects such as the bitmap object to be properly integrated because now the tuple contains the details to the exact row, column, and also the symbol of the cell.

Final Question

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

If I had the chance to start over, I would have provided a better abstraction for the client when it came to defining the attributes of an object. For example, I would try to use either templates or unions to allow the client to define their own types of attributes instead of only being able to define integer attributes. I would also try to make additional classes that are solely responsible for the handling of the collision and movement, so that the board class will not need to be fully responsible for the details of those operations. This would enhance decoupling and make the board class a little bit less congested. I would also make use of more interfaces and abstract classes which require subclasses to implement their pure virtual methods because this would be a solution to get rid of the parts where I need to use casting such as during the collision handling.

Conclusion

I found that the design of this project required a lot of critical thinking and back-and-forth changes. I also fell off the Due Date 1 schedule a bit because I was not able to properly estimate the time it took to complete the abstractions for movement and collision before implementing the two games. As a result, I was able to complete the implementation of the first game a day later than the date mentioned on the deadline. Overall, I found this project to be very interesting as it touched on many different aspects of OOP design and a handful of the topics discussed in class.