Final Project Writeup

Junya Wang

December 2021

1 Overview

AGE employs the Model View Controller design architecture. The Game, which extends Model, handles the game logic and state. The GameView, which extends the abstract View, prints the elements of the game to the screen using ncurses. A wrapper for ncurses windows is provided to make it more C++ friendly. GameView owns two of these wrappers, which display the screen and status bars.

The Keyboard, which extends the abstract Controller, takes in keyboard input from the player and maps it to an Action. The Keyboard provides functions and a constructor that let the client use their own preferred key bindings. Each of these classes are open to extension, and Game can own multiple Views and Controllers, allowing for more options such as multiple or alternate controllers.

The Game is able to get input from Controllers, and print to Views by calling methods from these classes. The Game also owns any number of States, having one in the forefront at a time. A State owns a Physics object and has a map of Entities and their heights. In a client-made AGE game, the client places Entities inside the State with their respective heights, as well as a Physics object. They repeat this for any number of States, adding them to the Game.

An Entity is a set of appearance/animated (Sprite), movement (MovementComponent), and collision (Collider) properties. It can be extended by the client to have specific create and on tick logic, as well as modify the game state by printing status messages, ending the state/game, and spawning other entities.

The client then adds Controllers and Views. When their Game is ready, they call go() on the Game. This calls the client defined create() methods for the first State and its Entities, which is their initialization logic. The Game then updates its views and displays them, allowing us to see our Entities and any status messages we printed on initialization. The Game then enters its core game loop. The Game owns a Clock object for which its tick() function returns true every 0.5 seconds. When tick() is true, the Game gets inputs from all controllers and passes them to the State, which passes them to every Entity it owns. The Game then calls the State's onTick() method before updating its views again. This loop repeats until the endState(bool win) function is called, which clears the views and does the client defined behaviour of ending the State based on win. By default, this initializes the next State and enters the core game loop for that State, or calls stop() and exits the Game if there are no States left.

We look into the State's onTick() method, a key part of the core game loop. It calls the doOnTick() method, which does nothing by default but can be extended by the client to fit the core game loop logic of their game. It then calls its Physics object to move all the Entities based on the Entities' Movement characteristics and the implementation of the Physics object. Both of these can be extended by the client to provide custom movement patterns and custom Physics, such as a wrapping screen.

By the default implementation, Physics gets the velocity of every entity and moves

it one unit at a time, checking for collisions at each step. If a collision occurs, it calls the moving Entity's accept() method, taking the other Entity's Collider as a parameter. Depending on the client's implementation, generic behaviour which is provided by the engine such as the Entities bouncing off each other, being destroyed, or stopping can occur, or the client can easily define custom behaviour by extending a doVisit() method that takes the relevant entity as a parameter. This exhibits Visitor pattern, where Entity is the Acceptor and Collider is the Visitor. Templates are provided to automate the construction of these classes. By default, Physics can either have a solid border or a view border, which depends on a boolean. If the border is solid, Entities can collide with borders, which calls their collision logic. If it is view, Entities can leave the screen and Physics starts the Entity's internal countdown to be despawned.

After the Entities are moved by Physics, State iterates through its Entities and calls their onTick() function. It also calls their functions that return information on how and if they want to update the status, end the state, spawn other entities, or be destroyed. The State executes these requests as they mutate the state, moving the responsibility away from the Entities.

This design is open to extension. The client can extend any of these classes to implement the needed behaviour for their game. Templates make it easier for the client to extend classes. For example, if they want to make a custom Entity called Player, they would extend Acceptor<Entity, Player, Collider>. They can then define custom parameters such as health, and implement overrides for doOnTick(), doCreate(), doEndState(), doBorderCollision(), and doUpdateStatus(). Sprites are easy to make as they can pass in a string, a single character, dimensions for a rectangle, or a vector of cells to the Bitmap constructor. The client can either use a provided movement pattern, or define their own by extending MovementComponent. Since all the movement patterns inherit from MovementComponent, client defined patterns can be combined with provided ones. The client can then define a PlayerCollider extending from Collider, and override the doVisit methods that they want. For example, if you have Circle, Square, and Triangle Entity classes, but you only want custom behaviour for Circle, you can override doVisit(Circle *e) and leave the rest as the default implementation.

Since the design largely obeys the Single Responsibility Principle and is closed to modification and open to extension, it is intuitive for the client to extend only what they need to create a large variety of games.

2 Updated UML

The main changes in the UML are as listed below:

1. Movement is changed from a Decorator to Composite Design Pattern.
2. Collision handling is removed from Physics.
3. Collision detection is removed from Entity and added to Physics.
4. The Border object is removed.
5. NVI is applied.

We will go over these changes and the reasoning for them.

1. The Decorator design pattern is difficult to use in the case of movement as the user must remember what order they decorated a base object in, due to the linked list-like structure. They must then pop the most recently added decorators off in order to reach the decorator they want to remove, and then add the desired decorators back on. Even with an erase method, having to remember the index of the decorators is difficult. Thus, removing and adding movement patterns, especially at runtime, is difficult. This goes against the specification that combinations of movement patterns should be possible.
   Also, Decorator pattern excels when we have 1 base object that we add extra functionality to. An Inert base object that returns a velocity of 0 is meaningless, and so we lose this advantage of Decorator pattern.

   Composite pattern is very similar to Decorator pattern in that we can add functionality to an object and access it as one object. I will go over its advantages in the Design section.

2. Collision handling is removed from Physics and added to Entity in a Collider object, which inherits from a Visitor to all of the subtypes of Entity, which inherits from a ColliderBase with some generic collision functions such as bounce, destroy, and stop. It makes more sense for entities to define their own collision behaviours, and this change also opens up the ability to use visitor pattern rather than an combinatorial number of dynamic_casts, which are expensive and also violate the Open/Closed Principle, as if we add another subclass we must add a dynamic_cast for every existing subclass, which makes it extremely hard to modify the engine. The challenge of handling collisions will be revisited in the Design section in more detail.

3. Physics already has access to all the Entities, as it is responsible for adding their velocities to their positions to move them. For every unit moved, collision must be checked for once. This is because if Entity A is set to move 5 units right but Entity B is in the way after only 3 units, we must stop Entity A and trigger a collision when

it reaches Entity B, rather than moving it 5 units and thus skipping the collision. Checking collisions is already under Physics' responsibility of moving Entities, and so this design is more in line with the Single Responsibility Principle than passing all the Entities to each Entity and having the Entities move themselves in addition to Entity's existing responsibility of being a wrapper for Entity behaviour.

4. With a Border object, meaningless functions such as checking for border collisions in a ViewBorder or checking for entities being out of bounds in a SolidBorder were added for the sake of maintaining a consistent Border interface. This functionality was moved to the Physics class, which is suitable since border collisions also need to be checked for every unit moved by an Entity, similar to the point above.

5. Non-Virtual-Interface Idiom has now been applied to this UML, as all virtual methods should be private and all public methods should be non-virtual to respect invariants.

3 Design

The design challenges I found most interesting were:

1. Entity collision handling.
2. Making combined movement patterns easy to use.
3. Allowing entities to update the game/state, while reducing coupling.

We will go over these challenges and the techniques I used to solve them.

1. I believe that Entity collision handling is the most difficult part of the engine. We want the behaviour of the collision to depend on the types of both Entities colliding, which brings to mind Visitor pattern. My goal in using Visitor pattern was to avoid using dynamic_cast, and also to avoid having to define collision behaviour for every possible pair of Entities. The Entities were the Acceptors and I provided a template to automate writing the doAccept function for the Entity subclasses. Also, an Entity owns a Collider object, which defines its collision behaviour towards other Entities.

   I used the class ColliderBase to wrap the templated Visitor class, as well as implement generic collision behaviours required by the engine. The Visitor class can take ColliderBase as a template parameter and inherit from it. By having ColliderBase not be a templated class, the generic collision behaviour does not depend on what subclasses the client defines and is closed to modification.

   In the code for the Visitor template, I applied NVI with a public visit() method and a private virtual doVisit() method. While the doVisit() method could be pure virtual in order to encourage and remind the client to implement collision behaviour, this becomes very annoying if the client does not plan for two entities to collide. Thus, the doVisit() method is implemented as {return;} in Visitor but is open to extension, so that the default behaviour for two entities colliding is to pass through. It is easy to change this behaviour by simply overriding the doVisit(T *e) methods for whichever Entities you want to define collision behaviour for, and leaving the rest.

   Since I as the engine designer don't know what subclasses the client intends to create, the client add their classes to the Visitor template themselves. Moreover, as Entity has a virtual doAccept method for accepting Colliders, Colliders must be able to visit Entities. The Collider can either be able to visit every subtype of Entity defined, or have a visit method for the Entity superclass. For the latter option, since Physics takes Entity pointers and calls their accept functions whenever a collision occurs, the Entity visit method is always called over the subclass methods. Then we would need to dynamic_cast to the subclass pointers, which makes this entire pattern redundant. Thus, the better option is for the client to modify the Collider class to take all of their Entity subclasses. This process is very simple, as it just needs a forward declaration and adding the class to the template. Though I am not delighted with the client having to edit the Collider code, I believe I found a

relatively painless way for the client to implement Entity collisions easily and that the tradeoff is worth it.

2. As mentioned before, I used the Composite pattern for movement. The MovementComponent defines the interface for all classes related to movement patterns. The leaves, Linear, Gravitate, Control, and Follow, are single movement patterns. Movement is a composite, which stores the leaves inside a map and lets them be treated as one object, as well as implementing add and remove methods.

   Thus, the total velocity can be obtained from one Movement object, similar to Decorator pattern, but we can identify patterns with strings and thus easily remove them. Since a composite can store a composite, this pattern becomes even more powerful as we can identify a group of movement patterns with the string "base", and let this become the base behaviour for an entity. We can then add other movement patterns, such as speeding up or bouncing back from a collision, with identifiers such as "collide" and "accelerate". When we want to remove them, we only need the string and will never affect the base behaviour.

   I observed that this deviation in the design made a significant difference in how easy it was to write client code, both because we do not have to construct the useless Inert class every time we define movement, and because we can modify and combine movement patterns easily. MovementComponent is also easily extendable, allowing us to define custom movement patterns.

3. Entities need to be able to update the game. For example, they need to print strings to the status lines such as their health or dialogue, and they need to be able to end the game if the player satisfies some win or condition involving Entities, such as killing a boss. Passing a game pointer or reference into every Entity couples these classes, and also exposes Game's interface to be used at inappropriate times by Entities. For example, an Entity could call go() on the Game, nesting the game loop. Entities also need to update the State by spawning other entities.

   My solution to this was to provide updateStatus() and endState() methods, and a vector of Entities waiting to be spawned to allow Entities to make "requests" to the State. Every tick, the State iterates over its Entities and calls updateStatus() and endState() for each one, and the State will update the status or end the state accordingly to the results. It will also move the Entities from the Entity's spawn vector to its own map of Entities, adding them to play. Like this, Entities are able to make changes to the game state promptly and easily, but unnecessary coupling and danger of tampering is avoided. This maintains the responsibilities of States and Entities without combining them, thus obeying SRP.

Some design patterns I used often that I have not mentioned yet are:

1. Template Method Pattern - Many of the methods that handle logic, such as State's onTick(), use Template Method Pattern. This allows them to run the required logic such as moving and processing entities, while still providing a hook for user customization through overloading doOnTick().

2. Dependency Inversion Principle - Modules depend on abstractions. For example, Game and Model depend on the abstract View and Controller, rather than the concrete GameView and Keyboard. This allows Game and Model to use other types of Views and Controllers. Entity depends on Sprite and MovementComponent, allowing for polymorphic treatment of subclasses that both I and the client define.

4 Extra Features

The memory management is done using RAII.

5 Final Question

If I could start over, I would start with the collision handling system and try to find a way in which the client does not have to modify code that Entity depends on. As it stands right now, this code is open to modification as well as to extension. Ideally, Entity would depend on some generic Collider which would visit Entities, but Collider would be able to be extended for the subtypes of Entity defined by the client. Also, the visit() functions that take subtype parameters would have to be called instead of the visit(Entity *) function, which may be difficult to find a way to do. I spent a long time on this problem, but I needed to make a tradeoff due to the deadline. As it is right now, there is too much code to change, so this would have to be figured out in a smaller project or a code rewrite.

I would also rewrite the positional and movement systems to use doubles instead of ints for more precise movement. An angle and speed system, similar to Java's Turtle, would be interesting and open up a lot more collision and movement opportunities without the client having to make calculations themselves.

Altogether I am pleased with the solutions I found for this challenge and would like to revisit it some time.