

Final Design Document

By: Felix Xie

Introduction

My rendition of the adventure game engine (AGE) incorporates many abstractions over traditional game mechanics, allowing the user to dynamically create complex 2D ASCII games. Ncurses is the library of choice used for the view and controller. The games I developed using my engine are the classics **Flappy Bird** and **Space Invaders**.

General Overview

My project follows the MVC architecture, separating the general uses of a game engine to promote organized and modularized code. Let us go through each component individually:

View

The view is a pure abstract class that is inherited by CursesView. It follows a singleton pattern where only 1 instance of it can exist (since we do not want multiple views). Upon instantiation, it sets up all the game window split into two separate subwindows; one for the statuses, and one for the game itself. The windows each have 2 buffers that has characters drawn on one and swapped to from the other previously shown buffer that gets cleared. The drawing is done upon being notified by the model which sends a list of objects. Based off height, it draws the objects in succession (so an object of height 1 overlaps height 0 without consequences).

Controller

Like the view, the controller is a pure abstract class inherited by CursesController. It also follows a singleton pattern for the same reasons. Upon instantiation, it attaches keypad and mouse action functionality to the CursesView. When the model requests an action via `getAction()`, it is notified and returns a `std::variant` of either `NoAction{}` (a struct that represents no action gotten) or a Keyboard/Mouse event. Game developers can then handle the respective events by retrieving the key or mouse position from the event sent to the model.

Model

Majority of the game functionality exists in the model. It is also an abstract class with two fields for the View and Controller with each being attached via `addView()` and `addController()`. The

abstract class extends two functions for its subclasses, `getAction()` and `notifyView()` in order to interact with the view and controller. Now let's go through every subclass in the model:

State

Another abstraction over the model's different states, provides a `start()` and `end()` function for each singleton class that inherits it.

GameState

A singleton class that provides the game loop and instantiates all the other states. The loop keeps track of game time via a clock, sending a frame refresh event per iteration of the game loop. The order of operations is as such:

- Get action from controller
- Update the world
- Notify the view
- Sleep for desired time for a constant frame rate

The engine runs on a default 33 frames per second. It provides a easy API; all game developers need to do is design their custom objects, populate the world as desired via positions, and call the `go()` function of the `GameState`.

WorldState

A singleton class that handles all the world logic, can be treated as the true game state. Here's a brief list of everything it does and how it does it:

- Object movement
 - o All objects have their own `update()` function that returns a position it would move to based on its velocity and direction, the World calls `update()` on all the objects in its own `update()` function
- Collision detection
 - o Following movement, the `update()` function calls a `manageCollisions()` function
 - o Based on the object's solidity (enumerated `SOLID`, `SOFT`, and `FOG`), it will handle it as such:
 - Solid objects impede movement + send a collision event to objects collided
 - Soft objects don't impede movement but sends a collision event to objects collided
 - Fog objects do not collide with anything
 - Any collision with the world boundary sends an event, solid objects are unable to pass though while soft/fog objects are allowed to
- Out of bounds detection + despawning

- Each object has a out of bounds tick count - if it matches their self-defined threshold, the world will despawn and remove the object
- Note that solid objects cannot go through the world border (including player), whereas soft and fog objects can
- Player tracking and movement
 - The world takes ownership of a singular player object
 - The player object is granted an observer role to mouse and keyboard input events and its view is tracked
 - The player is forbidden to go past the border, collision with the border is tracked and respective events are sent
- Event management
 - Acts as the concrete subject of the observer pattern between objects and events
 - New events are added and validated to the world and can be called using notifyObservers(Event)
 - If objects are not registered to an event, it is not notified
- Status tracking
 - Keeps track of the 3 status lines of the view and allows modification as desired with changeStatus(status number, content)

Resources

A singleton class that handles custom shapes and objects. Loads custom “avatars” via a custom file format, as an example (for the file coin.resource):

```

infobegin
length: 5
width: 3
delay: 6
frames: 2
infoend
framesbegin
begin
| o \
|   |
| \ /
end
begin
| / \
| o |
| \ /
end
framesend

```

(This is the main enhancement I implemented for easy object design and creation)

This feature makes it easier for game developers to visualize what their designs look like rather than blindly writing a bitmap of characters and positions. An additional `createRectangle(...)` function is provided for game developers to create `length*width` rectangles of char `c`.

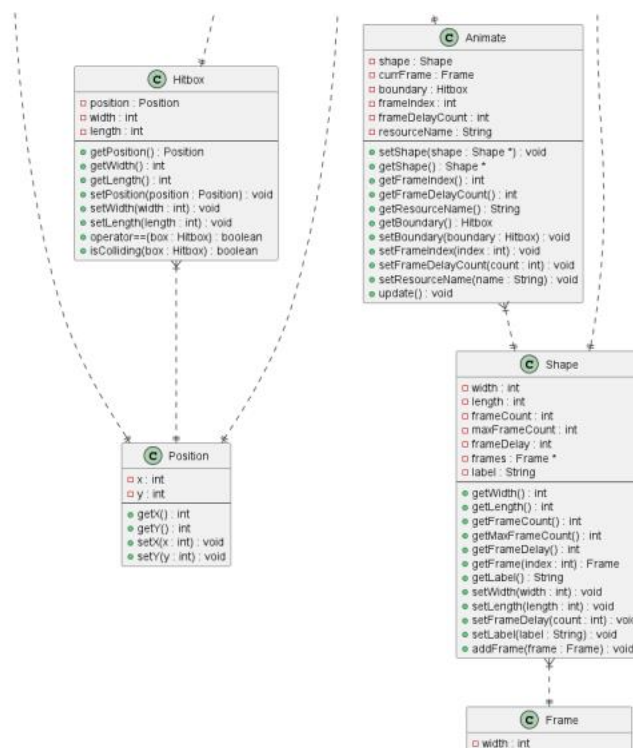
All resources are labelled by the game developer and objects can take ownership of these shapes and animations by simply setting their resource field to the respective resource name.

Objects and Events (+ objectmanager and eventmanager)

Objects have a hitbox, position, solidity, despawn time, animation frame time, etc. They also keep a list of event names they are subscribed to. The object class somewhat resembles a factory design pattern:

- Any objects created by the game developer are automatically added to the world with a new id with an `ObjectManager`, a vector based abstraction over objects
- The object class provides a virtual `handleEvent` function for any custom objects, all custom game logic essentially goes into `handleEvent` for a player object
 - o This makes the process of building a game very simple and organized

The design of the Object class's ownerships is summarized by this part of the UML:



Events just provide a type field to its inheritors so any objects that receive it can call `getType()` and handle it accordingly. `EventManager` provides a vector-based abstraction over many events.

Design

A lot of the design has been covered by the general overview, so here are just the major design choices I made:

- MVC
 - o Self-explanatory
- Singleton Pattern
 - o One instance for each aspect of the game makes sure that the resources are not corrupted when shared
 - o All states implement this, along with Resources
- Factory Pattern
 - o Custom objects defined by the game developer are automatically added to the game world with default fields and settings
 - o Lessens the load on the developer for managing how objects interact with the world, also allows cool things such as objects spawning other objects (seen in Flappy Bird)
 - o Object and ObjectManager implement these themselves
 - Eg. In Flappy Bird, the BirdObject can spawn new RectangleObjects
- Observer Pattern
 - o For world events to be handled by any object observing them
 - o Handled between Object and Event classes
- Command Pattern
 - o Requesting an action from the controller returns a variant that encapsulates the request, allowing the parameterization of clients
 - o Implemented by Controller and Model classes
- Prototyping Pattern
 - o Objects can construct and clone themselves as necessary during runtime in their functions (seen in Space Invaders)
 - Not explicitly in the class design but developers can easily allow prototyping themselves
 - Eg. In Space Invaders, every alien that dies spawns a new clone of itself

SOLID principles were used whenever necessary as seen by the low coupling and high cohesion of my code base. Good OOP practices can be seen by my abstract classes and subclasses.


Changes To Original UML

- Controller.h
 - o Instead of struct, changed Action to a variant
- Model.h
 - o Added hasView() and hasController()
- View.h
 - o Changed fields to pointers
- CursesView.h
- CursesController.h
- Position.h
- Hitbox.h
 - o Added isColliding() for QOL collision detection
- Frame.h, Shape.h
 - o Removed draw() function since it creates too much coupling by directly interacting with the view
- Animate.h
 - o Added a hitbox for easier debugging, changed draw() to update() and added a current frame field to be able to pass the state of the animation to the view
- Event.h
 - o Removed observers field due to incorrect Observer pattern usage (lack of concrete)
- EventManager.h
- KeyboardEvent.h
 - o Changed string to int (originally a typo)
- MouseEvent.h
- CollisionEvent.h
- RefreshEvent.h
- DespawnEvent.h
 - o Originally "OutOfBoundsEvent", now removed object field due to directly handling on an object to object basis
- Object.h
 - o Added velocityCount to have multi frame duration movement, prevPosition to move back to original when colliding with solids
 - o Added events handling to objects themselves and despawn semantics
- ObjectManager.h
- State.h
 - o Removed event handling and moved it to WorldState
- GameState.h
- Clock.h
- WorldState.h
 - o Added statuses as a field, moved event handling over from State.h
 - o Removed draw() to reduce coupling, modified collision handling
- Resources.h
 - o Added a createRectangle function to easily create rectangular objects

Extra Credit Features

As mentioned previously, instead of making the game developer write bitmaps of chars and positions for a shape, I implemented a file resource system that allows easy shape design and object creation:

Eg. A star I used multiple times in Space Invaders is easily loaded by this file:

```
> game2resources > 
1  infobegin
2  length: 5
3  width: 3
4  delay: 10
5  frames: 2
6  infoend
7  framesbegin
8  begin
9  |  ^
10 |  <  >
11 |  /  \
12 |  end
13 |  begin
14 |  |  /\
15 |  |  <  >
16 |  |  /\
17 |  end
18 framesend
```

This required a custom parsing and loading algorithm that I won't explain in depth but can be viewed in Resources.cc for the extra credit if possible.

Final Question

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

I would've started my coding following RAI rather than my original plan of refactoring my code to do so. It saves so much time from debugging segfaults and dealing with memory leaks. I ended not having enough time to refactor my codebase and resultantly missed out on the bonus marks.

Additionally, I would start earlier as I did not expect the amount of consecutive hours that I had to spend during exam season on this project, which sacrificed my time spent studying for other classes. I also missed the deadline on my original plan to add colours and audio to my game engine.

Overall, I enjoyed the process of creating such a complex project, thank you CS246E!