Rapport Project Space Invaders

Advanced Programming 2019
Mano Marichal

Introduction:

This rapport will consist of two sections.

Glossary:

- object: an Entity, View and Controller class togheter represent an object.

MVC:

The biggest and most important group of objects within the project are the objects, they serve as the actual objects within the game and are divided in subclasses like enemies, projectiles etc. All entities are implemented using the MVC, which means every entity consists of 3 classes:

- **Entities** serve as a small database which keeps the information about the object along with some functions to change the data.
- **View** are responsible for how the object is drawn on the screen, and can read data from RedAlien, but may not make changes.
- **Controllers** is responsible for giving the object its rules, like when can it move, does it have collision, when does it die and so on.

For example lets see how our typical enemy object works, like a green alien. We want green aliens to be simple enemies that periodically fire hostile bullets.

First up, we have the GreenAlien class, we see that it inherits from Enemy, which is an Entity (more on inheritance later).
There is not a lot to talk about. In the update function we see that the alien just call the move function, which is defined in Enemy because all enemies move in the same pattern and that it initiliazes the floats describing the position and size of the object.

Next up is the GreenAlienView class, we can see again that there is almost no code in the files. The View base class has a function draw which draws the sprite and is called every interval.
This means that all that a derived view needs to do is to define what the view should do when it is notified that a change is registered in the Entity it is attached too.
This is realized using the Observer pattern. All views keep a pointer to the Entity to which they are attached and when they get notified they read the change through the pointer. For example a purple enemy shrinks when it takes damage, so in PurpleEnemyView::onNotify() we can see it changes the scale of the sprite based on how damaged the PurpleAlien is.
Most derived views only update the position of their sprite. Note that View::draw() is defined virtual in our base View class, so a derived view can override this. This is for when a view needs to draw multiple sprites, like the lives counter in the top left corner has to do.

Lastly, we have te GreenAlienController class; Controllers are what give meaning to our objects. The handleEvents function is called each interval and is what is where the rules for the object are defined. First we can see in GreenAlienController::handleEvents that when an enemy collides with a standard projectile, it takes 10 damage. Second we can see that the class keeps a stopwatch which tells when to shoot an projectile (and which). Finally, a controller also needs to define when the object should cease to exist, this is why our handleEvents returns a bool. Higher up in the world class when this function returns false, the object (which means in the example our GreenAlien,

GreenAlienView and GreenAlienController) will be deleted. In this case we want our enemy to die if his hitpoints go below 0.
One final thing to note is that controllers keep a reference to the world in which they exist, this is so that when controllers need to create an object, they can tell where to create it. More on object creation/deletion later.

Conclusion:
The goal is that the game logic (Controllers and Entities) and graphical logic (Views) are seperated from each other. When we look at a World object, we can see it keeps 3 seperate vectors with entities, views and controllers, and has 3 functions updateEntities(), drawViews() and handleEvents(). This makes it easier to for example change how an object is represented without having to dip into the game logic and vice versa. Another situation where this is useful for when we want to have multiple views for one object, for example the PlayerHudView class draws the lives the player has, but like PlayerShipView, PlayerShip and PlayerShipController are undependent on the views.

Inheritance:

All entities are derived from a small list of base classes, that we can find in the folder base_classes. These include the classes Entity, View, Controller, Observer and Subject. The goal of inheritance was to make it easy to expand the game, like adding more objects.

Entity:
Entities represent the data of objects, so we can see every entity has 4 floats, 2 for the x and y position and 2 for the size (we work with rectangular collision). It has a constructor to initialize x and y, but the initializing of xSize and ySize is left over to the derived classes. We also see an update function, which is the one that is called in World, this is different for every entity, and every entity needs one, therefore we define it pure virtual.

View:
Views consist of 3 things: an Entity to which it is attached, a sprite which it draws and a texture for the sprite. The sprite and texture are unique pointers because the object is the sole owner of these, and a shared pointer for the entity. The view constructor is a little bit more interesting, it has a string and a pointer to an entity as parameters. Since all Views standard have 1 sprite, we can create the sprite in the base class, which avoids a lot of code duplication. The texture and sprite are created based on how big the entity is so if we want for example to make the playership twice as big, we do not need to edit the view file. The view also attaches to the entity. The sprite has the same size and position as the entity it is looking to, right before drawing the view uses the transform() function form the Transformation class to obtain a copy of the sprite with scale and position in pixel values.

In the derived classes is not much to see, since most of the stuff happens in the base class. The init function exists because we can’t use shared_from_this() from the constructor, and there exist no ethically correct solutions for this. This means the draw() function can’t be called if the view was not initialized, if it is called, it will throw an exception.

Controller:
Unfortunately, there is not much we can do in the controller class, because we can never predict what rules controllers will give to their objects. Each controller does have to define a handleEvents() function for when it is called in the World.

Enemy:
This class is used so we can refer to all enemies through the same type of pointer. We want for example that the playership takes damage when it touches an enemy, this way we can avoid an if statement for every type of enemy. We can see every enemy has a move function (all enemies move in the same pattern). This function is virtual because maybe in the future we want an enemy which has a different move pattern.

Projectile:
Same purpose as Enemy, but then for projectiles. This is used for example in shields, because they get damaged by all projectiles.

Subject & Observer: More on this later when we talk about the Observer Pattern.

Conclusion:
When we take a look at the code in our actual derived objects, we can see that except for the controller part, most files only have a few lines of codes in them.
Because we use inheritance we also can be sure that we dont need to alter other objects when adding new ones, unless it is specifically for this object. For example a friendly projectile is destroyed when it collides with an Enemy, so when we make new enemies we do not need to say in the code of the projectile that it should cease to exist when it collides with our new enemy because we check for base class Enemy.

Combining the different parts

The part of my code that brings togheter all these things are 2 classes: Game and World. World represents all the entities that are on screen, and manages them. Game is responsible for controlling the World. It tells it what level to load, when to update the views, entities etc.

The World:
This class is responsible for managing all objects currently active, therefore it has 3 vectors which contain shared pointers to the objects. 1 for the active entities, 1 for the active views and 1 for the active controllers. These three are the same size. Objects are saved by placing their entity part in the active entities, view part in the active views, and controller part in the active controller part. This makes it easier to add and delete objects.
When a controller returns false from their handleEvents() function, we know that the object should be deleted, we know where the entity and view are because they are at the same index in their vectors as the controller, so we save the index of the controller in the buffer objectsToDelete. This buffer exists because we want to delete objects after all objects got a chanche to update. The good part about this is that we do not need to look in the other vectors when deleting, since we have acces to the index. The bad part is that we can't have a singular view, entity or controller, for example a view that draws the lives the player has on the screen. To work aroud this, this view is owned by another view, when PlayerView needs to draw, it also calls the draw function of PlayerLivesView.

The functions updateEntities() , drawViews() and handleEvents() do what their names imply. These are in different functions to make it clear that game logic and graphical stuff are seperated.
World also has a function to load a level. The final thing is that world is an observer of our playership, this is because when the playership runs out of lives, we want the world to stop running the level.

The Game:
This class is what manages the game loop. This means playing levels, showing the game over screen and so on. It has 2 public functions, one being the constructor and a play() function. The constructor needs a location to a settings file. In this settings file we define 2 things, one being the size of the window. The second being an array containing the file locations of the levels we want to play. If we then call the play() function, the game object will start playing the levels. The game class is a singleton. I will talk more about the game loop when I will talk about the Clock class.

The Transformation class

This class has as function to provide views the necessary functions to transform game logic coordinates to pixel values on the screen. Everything in my project works dynamically. When reading in a level, we can set the coordinate system we want the game logic to use.
To see this in action, run the game with different_coordinates.json settings file. This contains 3 levels, both using 3 different coordinate systems: [-4, 4][-3, 3], [-8, 8][-6, 6] and [-8, 8][-12, 12]. The levels are the same. For example in the [-4, 4][-3, 3] level there is a Red Alien at (-1.5, 2.5), in the [-8, 8][-6, 6] level at (-3, 5) and in the [-8, 8][-12, 12] level at (-3, 10). The window size is the same for all 3. As we can see when we play the game, all 3 levels look and play the same. To accomplish this, we have the SpaceSettings class, this is a singleton containing all the info about the coordinates we want to use. The World sets these variables when reading in a level. We also need to make sure we never hardcode values. In for example PlayerShip.cpp we can see that everything, ranging from the size of the object to the move functions is set relative to the SpaceSetting val

Now for the Transformation class itself, which is also a singleton. The class provides 2 functions getXPixelValue and getYPixelValue to transform game logic coordinates to pixel values, and getXScale and getYScale. We provide a template functon transform to avoid code duplication. The template argument can either be an sf::Text or an sf::Sprite. It takes the argument and returns a copy, but with transformed coordinates and scale so they represent how the object looks in pixel values.

The Game loop

The main game loop, arguably the most important part in the project, is what manages everything. The loop is located in Game::play(). There are a few important aspects:

Displaying stuff on screen:
The first part of our game loop is just a simple while loop and a for loop that says which screens we need to display when, and when the world should load & play a new level.

Game::runWorld
This is where the we tell the world what to do, and when. It comes down to a while loop where we first tell the world to handle the events, then update the entities and then draw the views. We do need to make sure the ticks between intervals are the same on every computer, and deal with lag.

Ticks between intervals and lag.
We want our objects to move at the same pace on every computer and deal with lag, this is what we use the Clock class for. I used this website to implement a solution for those 2 problems:
https://gameprogrammingpatterns.com/game-loop.html
It comes down to that we update the entities & controllers with a fixed time step, but have variable rendering. We look at how much time has passed real time, and then calculate how much game time this is. Then we let the game logic catch up, by calling updateEntities and handleEvents multiple

times, and when the game logic has catched up, we draw the frame. This both deals with the game running to fast, and to slow. The Clock class is also a singleton.

Singleton pattern:

Singletons are objects where only one instance of is available at all times, to accomplish this, I made the constructors of my singleton class private, private, so no instances can be created, and all the members static. I also delete the copy & assigment constructors of my singletons

Exception handling:

There were not many cases where I needed complex exception handling in the project, most of the exeptions were things the code can't handle by itself, like for example a filename that is not found. Therefore I found it appropriate to use std::runtime_errors to resolve these.

Namespaces:

I use namespaces that reflect my directory structure, for example the RedAlien class has namespace objects::enemies::red_alien, and the SpaceSetting class would have namespace util

Travis:

My project compiles & works with Travis

Extras:

I implemented a few extra features that I will talk about here:

- Shields:
Shields are objects that block all projectiles, and can be defined in the level.json file

- Transformation:
You can choose the coordinate system that the game logic will use in the level.json file, every level can use a different coordinate system as well

- Different types of enemies:
I implemented 2 extra types of enemies;
    - GreenAlien: an enemy that shoots back at you in a set interval
    - PurpleAlien: an enemy that takes multiple hits to kill, and shrinks whenever it gets hit

- Endless mode:
After completing all levels, the game will enter endless mode, which keeps spawning enemies faster and faster at random locations until you die. The randomness is limited here, it will be the same every time you play endless so that you can practice and do not have to deal with bad rng. You can quickly see the endless mode by running endless_test.json

- Extra levels
You can specify what levels you want to play and the screen size through the settings file. Run the executable space_invaders from ./build with as argument the relative location of the settings file.
I have 10 standard levels which are played from the standard_settings file, and 3 to show the different coordinate system.