Design of project: Danny Invaders

By Tim Sanders  
SID: 20181853

Game loop

The game loop is a simple while loop that keeps running while the game window is open. The game loop will update all the controllers in our game in the update function. The game keeps track of all these controllers in a simple vector with shared pointers to the controllers themselves, the vector is declared with the superclass Controller so it has no idea what kind of controllers are being updated. This way the game can perform all of its functions without having to worry about any exceptions for certain type of controllers where the rules don’t apply to that controller. After updating all of our controllers the loop then sees if there have been any new entities created by the controllers since last update, one of the reason this can happen is for example is when a controller shoots, a new bullet will be created by that controller and needs to be pushed upwards to the game class and given proper connection to the rest of the game, it will then properly sets up this new controller and moves on the collision check of the game. The collision is just an easy check if the square hitboxes of the entities intersect each other. If there has been a collision it will call on both controllers to handle the collision on their own, this is again because certain controllers have different rules and the game class doesn’t need to worry about this. The collision function also passes along the other colliding controller so that they can interact with each other if needed. Following that we then need to check if any of the controllers are no longer valid and need to be removed from the game, for example if an entity dies or goes of screen is a valid reason to be destroyed. The final part of the update function is checking if a new level needs to be loaded or the game itself is over. The last part of the game loop is telling the graphics manager to update the visual part of the game, we will discuss how this works in the chapter about the SFMLmanager. The way the game loop runs and makes sure that the game speed and the rendering keeps consistent over multiple systems is by keeping track of how much we are lagging behind and keep updating the logic part of the game until we are back on track, this is because rendering takes longer then the logic part.

SFMLmanager

The SFMLmanager is the main part of the graphics side of our game. It’s an observer class and will observe all the controllers in our game that need a visual representation of our game. It will receive a notify with the controller and the event that it needs to execute, these events range from making/removing visuals to updating the text on our screen. It keeps track of all of these visual in views and text where and loops over them every render call in our game loop to update them.

Stopwatch

The stopwatch class keeps track of the time in our game, it uses a singleton design because of the fact that we only have 1 “time” in our game and we want to be able to check time everywhere in our code. The reason why I didn’t make this instance based like some other implementations I have seen online is because these rely on the fact that we would only construct the singleton when its needed in the game, this simply doesn’t happen in my implementation of the game, we always need the stopwatch every game loop so it would be pointless to make it more complex and better to just keep it straightforward to maintain a clear codebase.

MVC

The model-view-controller part of the game is split into the 3 classes: Entity, View and Controller respectively. The function of these are as follows:

* Model:  
  The model is just a simple class that will keep track of all of our data that an Entity needs to be simulated in-game, it has no functions that simulate of itself and completely relies on the controller to do the heavy lifting. This way the entity class can be shared by multiple controllers who just the standard values for an entity and can be inherited from if needed by other entities to easily add new data without having to redesign our code. Its important to mention that the model has no direct ties any other class (except for other data classes like the hitbox for example) so for all it knows its completely alone in the game.
* View:  
  The view class represents our models on the screen, it doesn’t have any new game data of its own except for the data needed to render itself on screen and a pointer to the model it represents. This way it can easily ask for the data it needs to render the entity on screen or change the sprite of entity if needed without disturbing the model. Although it knows the model exists it doesn’t have any links with the controller, so it has no idea of how to behave on screen and only updates the position of the sprite on screen every game loop. Every Entity uses the same view class because there are no special rules for certain entities on how they should be rendered.
* Controller:  
  The controller is responsible for deciding on how an Entity should behave in game. It will always be split into multiple controller classes, which inherent for this class, because not all entities have the same behavior in-game. It consists of a pure virtual update function that makes it so every entity needs its own rules on how to update every loop, this is also true for the collision function. The controller has a link to the model it simulates and is allowed to change it.

I would like to quickly talk about how enemies and the level MCs are designed because they act a little bit differently. First off the enemy controller is responsible for a whole group of enemies instead of a single one. This is because they all move and respond like one “hive mind” then specific individuals, the way I simulate this is by having one controller that controls a linked list of entity models that it loops over. The best way to describe this is with alien the hive mind analogy, the controller is the alien brood mother that has her mindless alien minions which correspond the individual entity classes and tells them to invade earth. The mother will decide what the next action is of the group, it will consider the state of the whole group and then issue orders to each individual one. As for why I chose a linked list, the main reason is that we only really loop over it one way to change all the values or get some maximum values and besides the delete we never have direct access on the chain. Another reason is because all enemies are supposed to die we want the chain to be easily changeable. The model pointer of the controller will be pointing to the model that needs to be changed.

The level is also a MC but instead of actual Entities it has controllers as models. These models are the important Entities currently in the game, these are the current player and enemies. It does have a model of its own because its still a controller and this has remained in the code to be able to add background elements to the game, but this sadly hasn’t been Implemented yet. It is also important to say that even though the in-game controllers act as the models of this controller it is not allowed to update them.

A quick summary

We run through a single game loop as a quick summary. First the game loop calls upon the stopwatch to start a new time interval, after that the stopwatch checks if the game has been lagging behind and starts the updating logic side of the game, updating can happen multiple time to try to catch up. In the update process the game calls every controller to do its thing, the controller will decide on its own what to do next and change the model accordingly. If there are any changes that the rest of the game needs to be aware of the controller will send out a notify to all of its observers. After that the game will then precede to handle the new entities that might have been added to the list of controllers and makes sure they are correctly initialized before moving on to collisions. The last part is to check if the level has been completed and load the next one or end the game if the last level has been completed. After the logic side of the game the rendering will start. We tell the graphics manager to draw everything on screen. The manager will then loop through the views and texts on screen to update them. When a view gets called to update it will get the data it needs to draw and do the right transformations to make sure it all happens correctly.

One last note on the functions In the code return int but don’t actually do anything with it. The idea behind this was to add extra exception handling for when more of the games values would be able to be edited. But this hasn’t been implemented yet so they are just placeholders for now.