AGE Project Final design

Fisher Obillos - 21001180

Overview:

The age project, as instructed, allows for the creation of ASCII art video games. At a high level, the program uses the ncurses library to allow the user to draw graphics of any kind to the screen and have them act as game elements. Furthermore, the project follows the conventions of MVC architecture allowing for it to be easily modified to have a different type of display or controller, for instance. In order to compile and run the project, simply navigate to the directory that it is hosted in, type "make" and then run the "age" executable file with either the "-f" or "-b" command line argument to be able to play flappy bird or breakout respectively.

Design:

The Game Class:

The centerpiece of the engine is the Game class which inherits from the abstract Model class. Within this class is the processing of all the logic throughout the game. The game has access to all of the entities, a list of collisionData, a controller to receive player input, and a view to display graphics. The specifications for each are as follows.

In order to store the entities, the Game class houses a map with integers as keys and Entity pointers as values. The keys of the map represent the unique ids of all the entities on the board, and allows for the Game class to query and update aspects of a specific Entity by simply having its id. Furthermore, the values are Entity pointers to allow for the storing of different types of entities, the specifics of which will be discussed further in the documentation.

In terms of collision data, this is all stored in another map, this time with pairs of integers representing the ids of two different objects as keys, and Collision pointers as values. In order for the user to have better control over how specific entities interact with others, I opted to make this part fully customizable, letting them specify the collision interaction between any two distinct id numbers. With this, every clock cycle, a function called checkCollision is run, this function essentially runs through the positions of all the entities on the board (which the Game has access to from the map of entities described above) and checks if any two entities share any number of coordinates. If they do, the game consults the collision data map which will lead it to the specific interaction between the two entities that was specified by the user. Collisions will also be discussed later in the documentation.

For the "View" portion of the MVC design principle, the Game class "has-a" pointer to an "NCurseView" object (which inherits from the abstract View class). Each clock cycle, the Game class calls its method notifyView in order for the NCurseView class to present all of the entities that are on the board for that clock cycle.

Another thing that the Game class handles itself is the displaying of messages to the three status lines below the board. This was done using a variadic template. Each call to displayStatus contains an integer referring to the specific line that the user wants to print to, and then any number of additional arguments that represent the game state. For instance, displayStatus may take in the arguments (0, "Player score: ", playerScore, "Enemy score: ", enemyScore) to show the score in a game of Pong, or perhaps the arguments (0, "It's dangerous to go alone, take this") in order to display dialogue from strange old men in caves. Under the hood, the displayStatus function operates much like an abstract list function from Racket, there is a general function that takes in a templated T object called First as well as another templated Args… args argument called Rest. In this first function, the First argument is inputted into a string stream then the function is called on the Rest. When there are finally no more arguments, a version of the function which takes no arguments is called in which the contents of the string stream are dumped into a string which is finally used in an ncurses mvaddstr function to print the message in the provided bar. The implementation of this function can be found at lines 43 through 53 in the file "age.h".

Lastly, there is the issue of receiving user input. To achieve this, the Game class once again “has-a” pointer to “NCurseController” object which inherits from an abstract Controller Class. In order to get the input from this controller, the Game class calls its method “getControllerInput” which prompts the NCurseController to read in a character using the built in ncurses getch function. This value is returned to the Game class and can be used to control all entities that have player input as part of their movement.

Game Objects:

The age engine supports the full creation of each type of object, those being a single character, a rectangle made up of a single repeated character and a bitmap object. For each type of object, (which I decided to call entities) I created a specialized class which inherited from an abstract Entity class. In these classes exists essentially all the information and methods that the engine needs in order to properly interact with these entities. Specifically, each entity class has fields pertaining to their specialized and unique entity id, their height, integers representing their positions of the board, and a character (or characters in the case of the bitmap entity) which represents the form of the entity. Also, each entity has a vector of Movement pointers that is accessed each clock cycle by the Game class through a method called resolveMovement. In this method, the Game class runs through the vector of Movement pointers for each entity and changes its position based on the fields of each element in the vector. This lets the user assign a combination of movements to each entity. For example, if a specific entity’s moveList looked something like {lineMovement(down), lineMovement(right)}, then the entity would move down and to the right each clock cycle. Finally, in order to implement the movement and display of the three different types of entities, different fields were required for each. These differences will now be outlined.

For charEntity, we simply store two integers representing the x and y coordinates and a character which gets displayed to the screen. To display this type of entity, the view must simply use the three fields as arguments to the ncurses mvaddch function. This lets it print the entity anywhere on the board. Movement is equally simple, each movement object in the movement list can simply edit the x and y fields which will then cause the entity to be printed somewhere else during the next clock cycle.

For rectEntity, we store four integers and a character. The integers are used to store the x and y coordinates of the top left corner of the rectangle as well as the length and width of the entity. The caracter, of course, represents what will fill the rectangle. For printing, a double nested for loop is used to begin printing the rectangle at the top left and proceed by printing one row at a time. For movement, we handle it very similarly to how we did with charEntity, simply letting each movement object in the movement list edit the coordinates of the top left corner.

Finally, for bitmap Entities, we store a vector of tuples of two integers and a character. The two integers in each tuple represent the x and y coordinates of that "part" of the entity, and the character represents what will be printed at those coordinates. For printing, we simply loop through the vector, moving to the specified x, y coordinate and printing the specified character. For movement, we perform another loop, this time changing the coordinates at each iteration.

Movement:

Moving on to movement, all of the specified movement options have been fully implemented, those being line movement, gravitating movement, and in place cyclical movement. A class has been created for each type of movement, all inheriting from the same abstract Movement class. Furthermore, each movement class implements three different versions of the "updatePosition" function, one for each type of entity. The specifics of these implementations are as follows.

For line movement, the class is initialized with a user specified dy and dx value. This makes it easy for this type of movement to change the position of an entity as it simply adds the dy and dx values to the corresponding y and x values of the entity.

For gravitate movement, this class is initialized with an integer (called border) ranging from 0 to 3 inclusive representing the border that the entity is supposed to gravitate to. In order to move the entity, the movement object consults its border field to determine which value (x or y) to increase or decrease, thus making the entity take one “step” towards the specified border after each clock cycle.

For in place movement, the class is initialized with a vector of characters. The class also holds two integers called size (representing the size of the character vector) and index (allowing us to save our place in the character vector). Each time update Position is called by this Movement object, the fill character of the entity is changed to the character contained in the vector at index index. This lets the entities with this movement specification cycle through various different character representations, thus giving the illusion that they are, blinking, shining, shrinking, etc.

Collision:

Lastly, there are the various collision classes which handle the behavior of the program when two intersect with one another. All of the specified interaction types have been fully implemented, those being stop collision, bounce collision, pass collision, game end collision, and destroy collision. The previously mentioned collision classes are implemented as follows.

For stop collision, when two entities with this collision specification interact, the method resolveCollision is called. For this class, the method simply clears the movement lists of each of the two entities through a method called clearMovement which is a public method residing in the entity class. Thus, each entity has no more movement to resolve each clock cycle and they have effectively stopped moving.

For the bounce collision, this class is initialized with two boolean fields. These two fields, called y and x respectively, determine which components of the movement will be inverted upon collision. For instance, when the y field is set, and the collision occurs, the resolve collision method will invert only the y components of all the movement objects in each entity’s move list via another public method in the entity class called flipMovement. If, on the other hand, the x field is set to true, it is instead the x component that is flipped.

For pass collison, the implementation is very simple. The class is initialized with no arguments and when its version of resolveCollision is called, the method simply returns, allowing the two entities to continue moving as they were.

For gane end collision, this class is again initialized with no parameters. When two objects with this collision specification interact, the resolve collision method sets the game over field in the game class to be true. Thus, this terminates the game loop and effectively ends the game.

For the destroy collision, once again the class is initialized with no parameters. When two objects with this collision specification interact, its version of the resolve collision will call the remove entity function on each of the two entities involved in the collision. Before it does this however, the method checks the value of the private boolean field called “invincible” which is present in the entity class. If either of the two entities involved in the collision have this field set to true, they are not destroyed and instead continue with the movement they had prior to the collision.

<u>**Deviations From Plan of Attack:**</u>

In terms of deviations, most of the specifications for the implementations of the various classes stayed the same. If anything at all, most things just required more private methods than I had initially intended.

When it comes to the self administered due dates, this is where I really deviated from my plan of attack. I had initially intended to complete the implementation of all the movement classes by December 3th. This turned out to be a massive miscalculation seeing as how I ended up fully finishing the movement classes on the due date of December 14th. For the collision classes, I originally intended to have them fully implemented by December 5th, once again I only ended up completing this task by the due date of December 14th. Next, I thought I was going to complete the first game by December 8th but instead finished it again on December 14th. Finally, not only the submission date but also the actual second game was different than I thought it would be. I thought I was going to create the game pacman, and have it finished by December 8th, but it ended up being the game breakout and it was finished on December 14th as well.

<u>Question:</u>

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

There are a few things that I believe I would have done differently if I had the chance to start over. Most likely the main thing I would change however would be to stick to using smart pointers and vectors for the entire project. Since ncurses leaks a little bit of memory, it makes it difficult to effectively track memory leaks that were caused by improper memory allocation on my part. Had I used smart pointers, I could have avoided the headache that was solving all my memory errors.

Another thing I would probably change would be the way I approached the UML and plan of attack on due date 1. Admittedly, at the beginning of the project, before having actually implemented any code, I was completely unsure of how certain functions would actually work in combination with others. As a result, I created my UML as a sort of “best guess” of what the full implementation of the project would look like. The same goes with the plan of attack, I was totally unaware of how long certain tasks would take in comparison to others and thus took major deviations from the dates outlined on my plan of attack. This led to an enormous amount of stress when the due date inevitably began to close in. If I had given more thought to the original UML and the plan of attack, then I may have saved myself a lot of stress and headaches when I had to change the implementation of certain classes.