Caden Greenhalgh

Prof. Siddique

CSCE A201

14 April 2024,

Project Report

**Brief Overview:**

After discarding our original Tetris program (due to complexity and time constraints). We chose to create the snake game as it was the ideal project to represent the “TControl” (Terminal Control) library that we originally developed for the Tetris game.

**Individual Work:**

While my contribution to the TControl library was minimal. I developed and implemented all of the functions that display the game and control the game logic, I also implemented the powerup settings menu, though the menu was just a template of the previous menus developed by my partner.

Specific parts developed:

Functions – playGame(); setBoundary(); powerUpEditorMenu(); createGrid;

Classes/Structures – Snake class + functions, Food structure + functions, Powerup structure + functions

**Development Process:**

setBoundary()

I initially began by implementing a function set the border boundaries in order to set up my next function to develop the border/boundary lines. It was initially meant to be a square, so it would return the smallest value of the screen\_size vector pair . Though, I ended up modifying it to just return the screen\_size pair since I modified the createGrid function to make the grid the entirety of the terminal window (except the top three lines of the terminal since those needed to be reserved for the scoreboard).

createGrid()

My next step was to create a visual grid to display the game field. I developed this function to read in the boundaries of the screen, then loop through the set boundaries and use the setChar function from the terminal class to set the boundaries as well as the field space in between the boundaries as whitespace characters (with background color). Then, I used the draw function from the terminal class to display the grid.

Snake Class

Now that I had the grid set up, it was time to develop the snake object. Since the terminal class used rows and columns to effectively produce an x,y graph display, I decided to use a vector of pairs to store the snake body, since it could store each segment of the body as well as its respective x and y coordinates. That way, when I eventually added the grow function, I could easily add to the body–since you do not have to set an initial size for vectors. The snake class consists of a few main functions, the movement function, the changing direction function, the check self collision function, the check boundary collision function, the draw snake function, and the grow function (implemented later in development). The move functions work by updating the coordinates/position of the head, then updating each body segment back toward the head. It chooses which direction to move them based on a switch that then changes the offsets depending on the value of the character variable direction. Basically, the user can control the movement of the snake using standard movement keys: w, a, s, d. The change direction function is the function that the user interacts with to change the direction in the move function. It first checks to make sure that the users inputted direction is not in the opposite direction that it is currently moving (back towards the body, since that would cause the snake head to collide with the body). Once it makes sure of that, it changes the direction character to the value of the new direction (inputted by the user). The grow function was added later into development (after the development of the boundary and collision detection, as well as the food structure). The grow function works by retrieving the tail segment of the body and uses its position to calculate the new segments position. It also uses a switch to make sure the new segment is directly behind the previous tail. It then uses the vector push\_back function to add the new body segment to the end of the pair of vectors. The checkBodyCollision function was the more difficult of the collision functions to implement because the body moves–unlike the boundaries. I had trouble figuring out how to loop over each segment of the body and compare its coordinates to the head, as well as figuring out the types. But after some research, I figured out about the auto keyword and development continued smoothly. The checkBounadryCollision function was simple to implement since it only required checking if the head coordinates overlapped with any of the coordinates of the static boundaries. The most difficult part of the game logic was developing the drawSnake function. It took many attempts to figure out how to make the snake draw its head–and eventually body–then, as it moves, erase the head–and body– at its previous position. I knew that I wanted the head of the snake to be a different character depending on its direction, so I used another switch to change its head based on the direction of the snake (using getDirection to receive the direction of the snake). Looping through the head and body segment to draw their positions was relatively easy, just using a range-based for loop to loop over each pair in the body, then setting its coordinates in the grid to a character using t.setChar, and using the front() function to receive the head (the first segment of the body in the vector of pairs). Erasing them, on the other hand, was more difficult. I had to update the snake class to store the previous position of the tail, so that way I could set that position to a whitespace character using t.setChar, which took me awhile to figure out and implement.

Food Struct

After developing the rest of the game logic, I developed the food struct to encapsulate the functions required to spawn the food. It begins by initializing the row and column to -1 (this comes into play later inside the playGame function). It contains two functions, one to “spawn” the food (generates random coordinates inside of the boundaries) and check that the food isn’t spawning inside of the snake body. The other to check if the snake ‘ate’ (collided with) the food. It uses a method similar to the collision detection for the boundaries, by looping through each segment of the body (pair in a vector of pairs) to see if any of their coordinates are equal to that of the position of the food. If it is, it returns true (meaning collision is detected).

Powerup Struct

My final contribution to this project was implementing two different powerups, one to disable the body collision and one to slow the speed of the snake. Both of these powerups operate on a ‘timer’ (multiple functions that work to get the time at activation, and subsequently subtract that time from the current time, comparing it to the variable that stores an integer representing the amount of time they are supposed to last). Developing the self-collision disabling powerup was simple, as self-collision already had a flag that made it easy to enable/disable. The slow-mo powerup, on the other hand, was difficult to implement as it kept breaking the scoreboard. I first tried to implement the timer, and powerups directly in the playGame function, but that made the function too bulky, and it was difficult to manage, on top of breaking the game. So instead, I decided to make a struct to encapsulate all the powerup functions. I began by copying the spawn function–and the checkCollision function– from the food struct, tweaking it slightly so that it could randomly spawn one of two different power ups. Then, I developed the functions to activate and deactivate the powerups. After getting the powerups to work, I developed the effect timer and subsequentially the respawn timer. Finally, to make sure that only one powerup could be on the board at a time (and that a powerup cannot spawn if the user currently has a powerup effect), I developed the shouldPowerUpSpawn function. It uses multiple checks to determine if a powerup should spawn (first, checks to make sure user has power ups enabled, then, checks to see if the initial powerup has spawned, finally, checks to see if the user has a powerup active and if the respawn timer has ended and if there is currently a powerup on the board). I then implemented this into the spawn function within the powerup struct.

playGame()

This function changed a lot during development. Initially it just looped to allow the head to move through throughout the grid/field. Then, I added the barrier collision detection, then the food, growth, and game speed, then the body collision, then finally the powerups. I did not implement the pause menu, or the loop to the main menu after game over.

**Conclusion**

In conclusion, working on this project has helped broaden my knowledge of C++ programming and taught me many lessons about how to handle a large, team oriented, project. Not only did I expand my understanding of many aspects of C++ programming (such as classes, structs, vectors, pairs, vector pairs, loops, libraries, headers, etc), but I also learned how to effectively communicate with team members, break down large projects into smaller goals, and also improved my problem-solving skills. If I could do anything differently, I would’ve tried to work at a steady, continual, pace so that way the project could have been even more polished and complete. Though, I am still happy with how everything turned out. A few things I wish we could’ve implemented would be file saving (For settings, and saving high scores), a “view high scores” menu option, different game modes, and a method to change the direction keys within the program.