CSC 481 Homework 5 Write-Up

Sarah Willis

11/16/20

For this project, I was tasked with creating programs in C++ using Microsoft Windows 10, Visual Studio 2019, SFML 2.5.1, ZeroMQ release 4.3.2, and Duktape/Dukglue that satisfied the requirements of four different sections (the third of these being optional and the fourth being a reflection). I did not complete the third section. The first section involved adding a script management system to my game in order to script the behavior of game objects as well as event handling and raising. To begin, I took a look at the example posted on Moodle and decided to use the ScriptManager class implementation seen there as a starting point for integrating scripting into my engine. The first change that I decided to make was to have the ScriptManager record the last write time for a loaded script as well as its name so that it would only load a script again if it was a different file than the one loaded or if the write times did not match. By making this change and having the loadScript function return a Boolean indicating whether a script was actually loaded, I allowed for potential performance savings due to only reloading scripts when accessing different files or when there is a modification to the same file as well as due to being able to only run a script when it has actually been modified if it is a script that does not need to be run every iteration of the game loop.

After adding in this new functionality to help with performance, I considered the fact that allowing for multiple contexts could also greatly aid in performance by allowing a regularly run script to remain loaded on the same context for the duration of a program. Ideally, every script would have its own context to minimize the amount of loading and potentially running, or at the very least scripts run more often could have their own contexts while those run less often could share contexts. This led me to consider having the ScriptManager contain a list of contexts and corresponding scripts rather than merely being the wrapper for a single context. Ultimately, however, I opted to maintain the ScriptManager as a wrapper for a single context while keeping it non-static so that multiple instances of the ScriptManager could be used in a program. I made this decision because I realized that allowing for multiple ScriptManager instances would allow for different instances to be accessed simultaneously in different threads, allowing for script loading and execution to be performed more efficiently while also making multithreading with scripts less of a hassle in terms of synchronization. While I did not end up putting this to use in my program due to my limited use of multithreading, if I had utilized multithreading more in my program I imagine that this design would have been helpful in reducing synchronization issues and allowing for better performance due to the use of contexts in parallel across threads rather than restricting access to a single ScriptManager instance.

After making the decision about contexts, I faced the problem of how to input arguments into a script in order to allow inputting event arguments as well as game objects. At first I planned to use a variant implementation similar to the one I had completed for event arguments, but after having trouble with getting unions to be recognized in scripts, I decided to change my approach by instead creating an enum in ScriptManager to specific argument types and having the runScript method take in a list of argument types along with the arguments themselves, allowing it to push each argument based on its corresponding type. As the variant implementation would have necessitated an enum as well and scripts would still have been expecting particular arguments, I believe this approach works fairly well at allowing different argument types to be sent in to a script while avoiding the need to extract an argument and confirm its type within the scripts themselves.

Once I had my ScriptManager implemented, I decided to focus on utilizing scripting on my server rather than my client due to the need for using scripts for event raising and handling (as my event

management system exists solely on my server, with my client mostly handling rendering and sending user input). Thus, I focused my scripts on the game logic which is dictated by the server. For regularly updating a game object, I added a script that changes the velocity of all moving platforms every iteration of the game loop. As this change only has any real effect if the script is actually modified, it would be possible to only run it after loadScript returns true to indicate a script modification, but I did not make use of this functionality in this case in order to make sure that I was demonstrating a regular update of the game object. For event raising and handling, I decided to create a new EventHandler called ScriptActivationHandler that handles a ScriptActivationEvent generated by pressing one of two defined script activation keys (comma and period). The handler sends in the event arguments to the script, allowing it to determine the type of key used and handle it appropriately by raising events through a static function defined in ScriptActivationHandler that parses a passed-in string to get the event type and its arguments. For the event types, I used the existing character death event as well as a new event for the death of all characters. I also added to the script the ability to change the velocity of one or all characters in order to help demonstrate how you can comment and uncomment different lines during program execution to change the event handling.

For the second section of the assignment, I was tasked with creating a second game using all of my engine systems. I decided to implement the classic game Snake because it is a game that I've always enjoyed and I thought it would be interesting to implement the snake movement. I decided to keep my client-server architecture mostly the same, with the server handling the game logic and the client handling rendering, though I restricted connections to a single client due to the game being designed for single-player and some aspects of the implementation, such as how to deal with removing snake blocks upon a character's death, not working properly in a multiplayer environment. Only one client can be connected at a time, but after a client disconnects another can then connect in their place.

As my game object model is property-centric, one of my first tasks was to determine what properties I would need. I found that I was able to reuse my existing Collision, LocationInSpace, Rendering, and Respawning properties, only needing to add one new property, ChainedDirectionalMovement, to handle the unique case of the movement of the various blocks making up a single snake. For this property, I set it up so that objects would be viewed in terms of a chain, each having a head object and a tail object. Each object in a chain also has a connection to the next in the chain through a map defined in the property, allowing for iterating through the whole chain when processing movements. When the processMovement function is called on a chain, the head will move according to a value set by multiplying the change in time by their velocity, with the remaining objects moving based on the previous movement of the object before them. I struggled to get this movement to work properly for a time until realizing that I needed to account for turning, leading me to have objects correct themselves to match the object before them in the chain when they are heading in the same direction. This led to the correct snake movement, though it is not perfect as there is some over-correction at times that leads blocks to overlap each other slightly. This is generally not terribly noticeable visually when playing the game however due to all blocks being the same color.

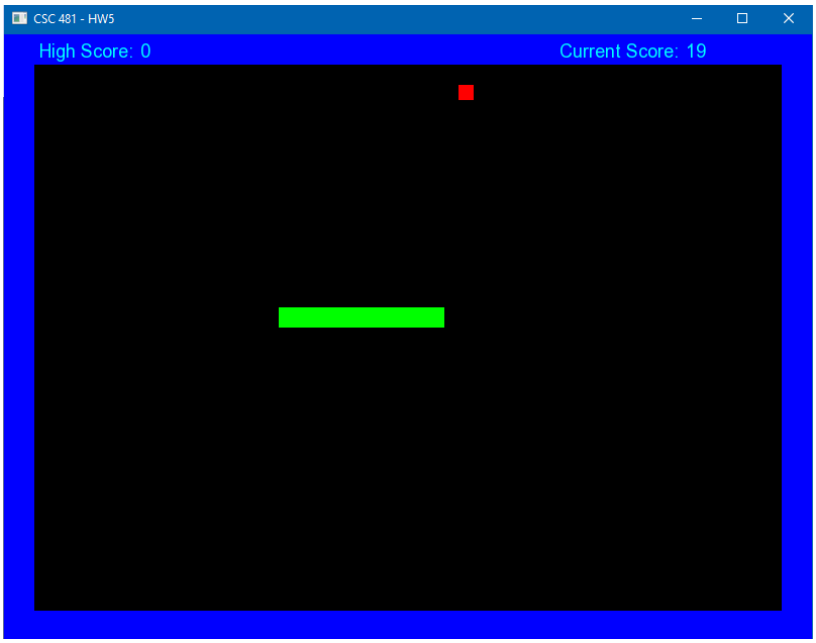
After getting the properties in place, I turned to the utilization of my event management system (which puts the existing timelines to use once again just as in the movement property). I was able to use my existing EventManager class and Event representation, though I did not implement the game so as to reuse the existing replay functionality. I did have to re-implement my specific event handlers to fit the logic of this new game, with the exception of my handler for positional updates which required only minimal changes due to merely sending position updates to the client. In terms of scripting, I decided to have the handling of user input be performed partially through a script, allowing for potential modification of this script later on to change how different user inputs are handled. I also decided to focus on providing appropriate customization through the scripts, this time involving my scripting system on both the client and server. On the client-side, I provided scripted functionality for changing the colors of all objects as well as the keys used for input. Two different color schemes provided in the existing script

are pictured in Appendix A. On the server-side, aside from the scripted user input event handling I also provided scripted functionality for setting the number of blocks for the snake to grow with each piece of food consumed, as the length that the snake grows often varies between games. Unfortunately any number past three would unexpectedly break the game, but values of two or three still seemed to work fairly well despite being a bit buggier at times than the default of one, allowing for the game to be made a bit more difficult. The collision of the snake with itself was also a bit of a tricky issue due to the nature of its movement and can still be overly sensitive at times, occasionally causing an unforeseen death.

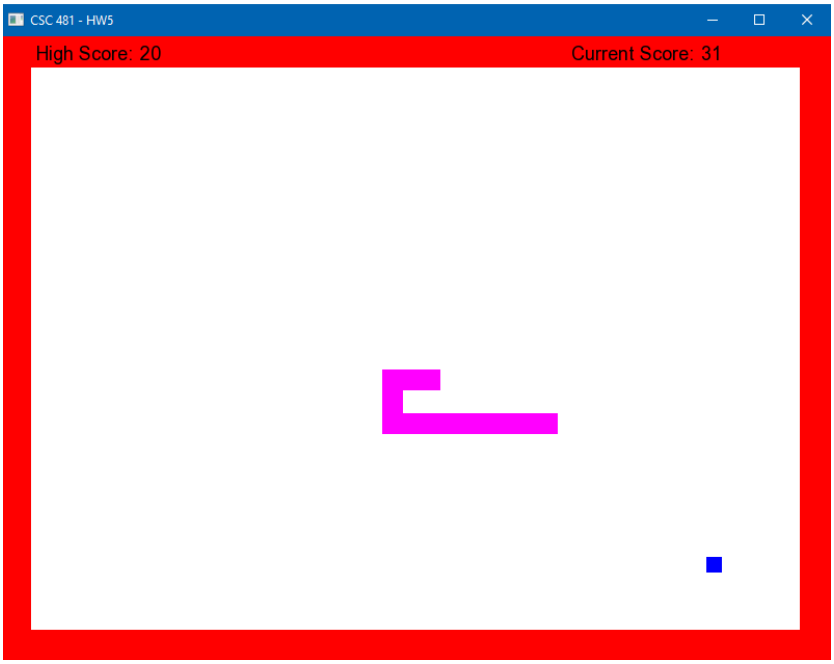
After I finished implementing the second game, I performed a diff between my first and second games as instructed to examine the amount of code that needed to be changed. The results of the diff can be seen in Appendix B. The number of files changed as well as the number of insertions and deletions seem quite high, indicating that I was not able to reuse as much as would have been liked. I think this is mostly due to the fact that I do not always separate out different parts and functions of code as effectively as I should, leading me to need to reimplement these sorts of functions to fit a new project. In the future, I would like to improve at modularizing my code and separating it into appropriate pieces for reuse, but I am glad that I was able to design some parts of my engine such as my properties and event manager appropriately to allow significant reuse in my second game.

Appendices

Appendix A:

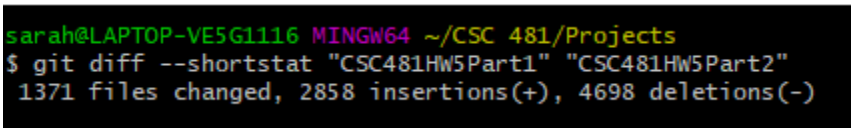


Snake Game: Color Scheme 1



Snake Game: Color Scheme 2

Appendix B:



Diff of First Game and Second Game