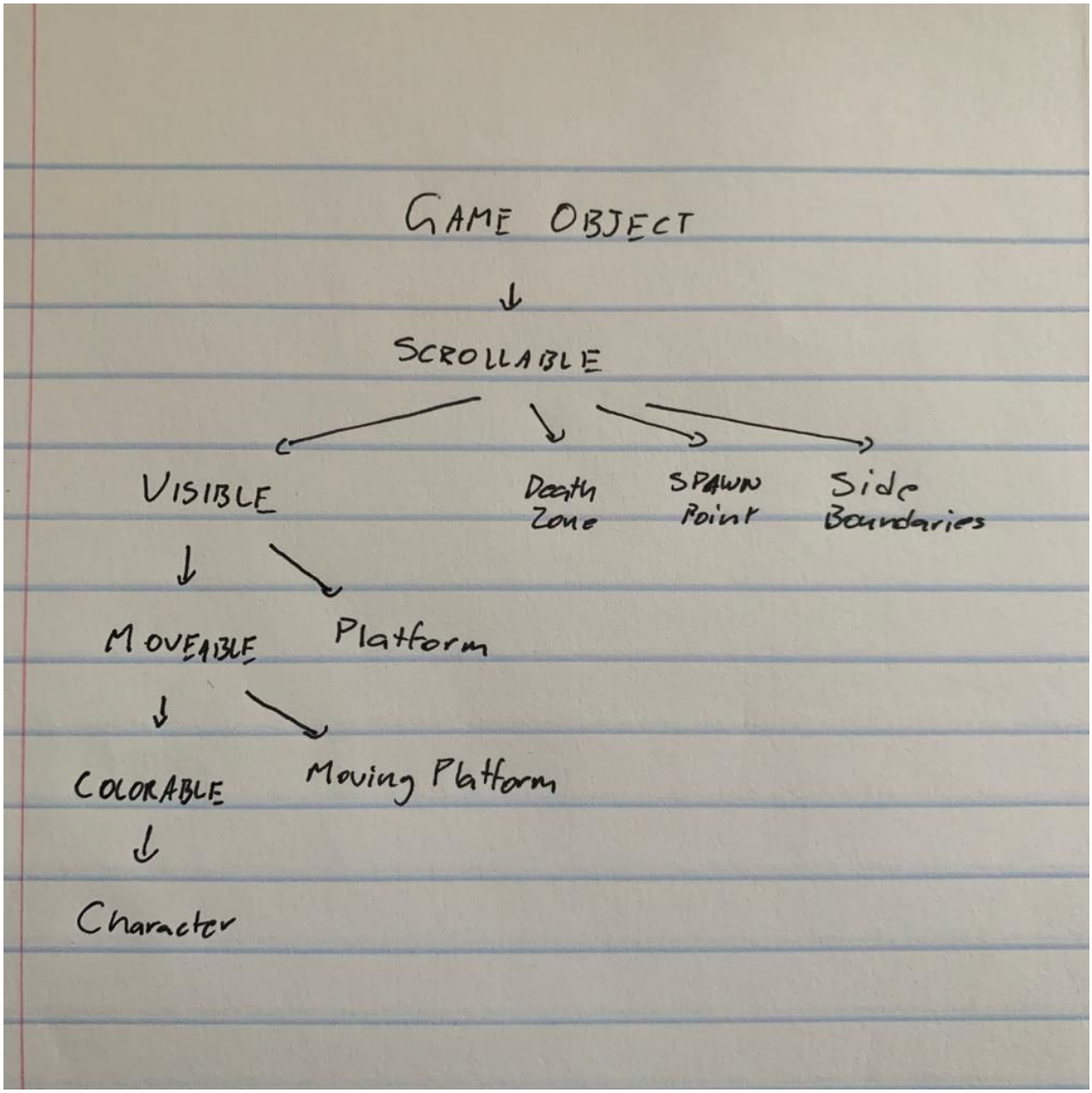
**HW3 - KARAN RAKESH (200316009)**

**Part 1:** For designing a runtime game object model, I was unsure whether to choose a generic model or a monolithic hierarchy but went with the latter since I felt that the ease of implementation since it is a known concept as well as having a small scale model that can be easily represented as a hierarchy and avoiding bubble up since the object model is well constructed to allow for other functionality in the future while encapsulating all the relevant behaviours. So i used a monolithic hierachy with the following structure :

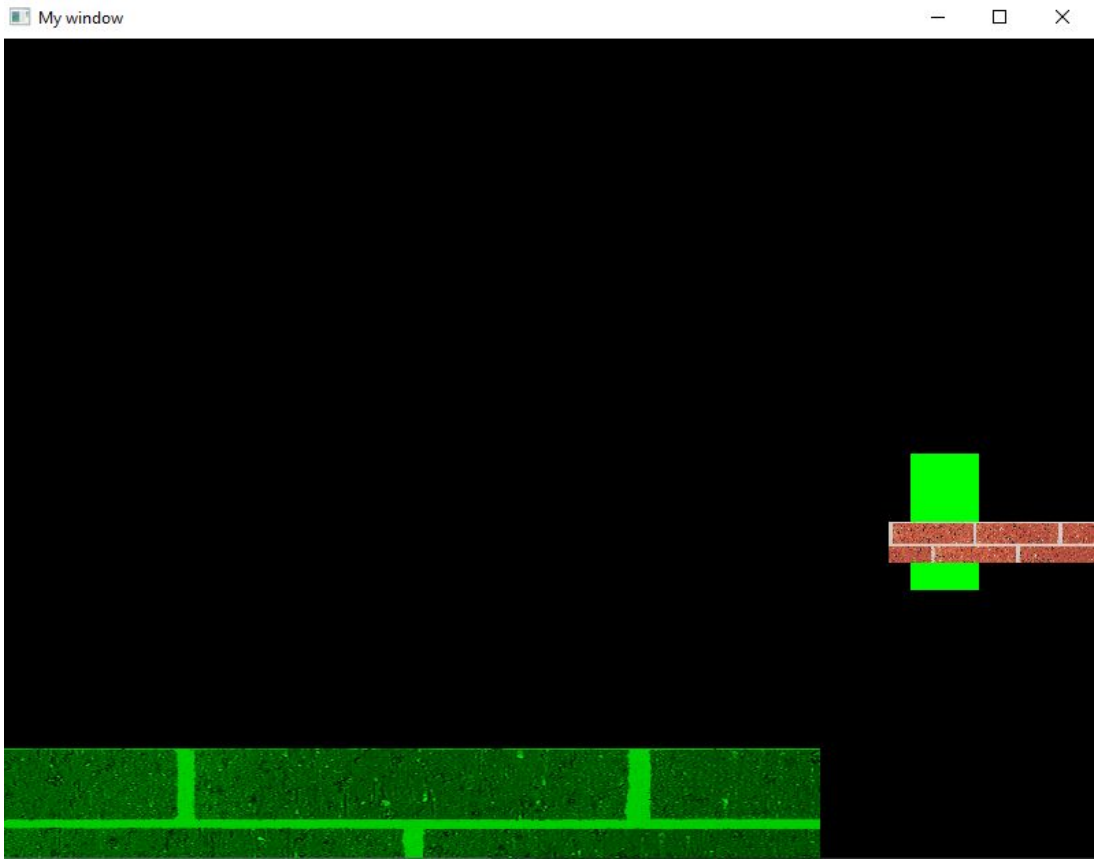



Though in actual implementation I only used Spawn Points as static objects that did not get scrolled, if I wanted separate spawn points I could have used the same. I planned on

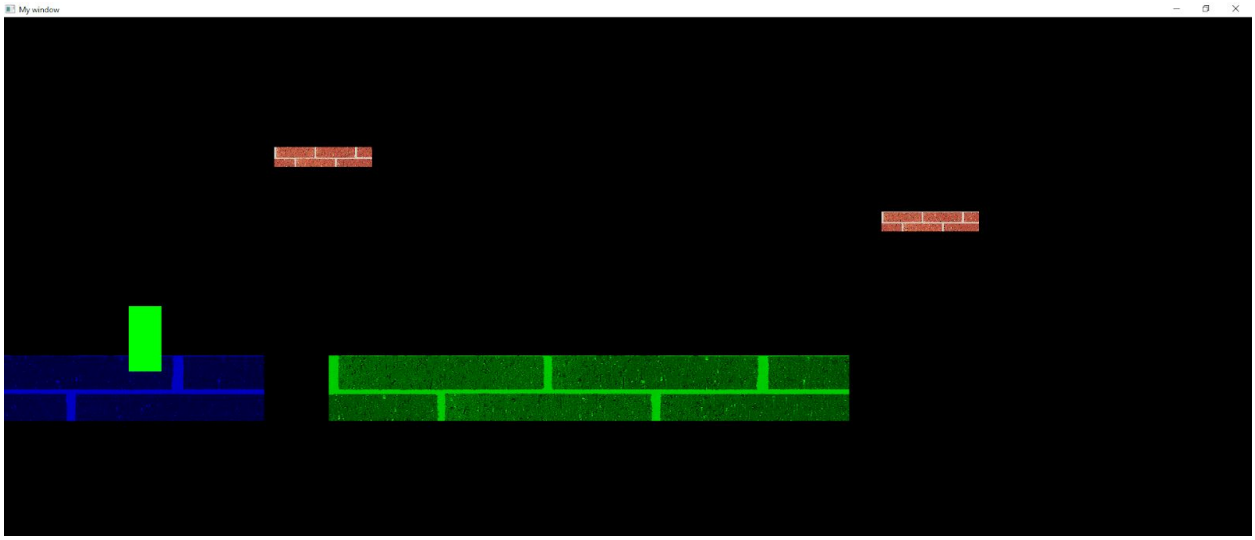
implementing the Score and Timer as Non-Scrollable components but didn't have the time to incorporate those elements into my final solution. Each component's constructor calls the parents constructor to invoke it's constructor and uses API's from the parents wherever possible. In case of hitting the side boundary in character's case, I do not want it to scroll as much as the other elements so that it remains on the screen and hence I over rode the existing function for it. As specified earlier, I used basic inheritance to achieve this model and believe that given the scale of this model and the given implementation I should be able to avoid bubble up and continue integrating elements into this model as required.

**Part 2 :** My previous submission used nothreading and leveraged the power of sfml to handle multiple clients across a single socket seamlessly to achieve multiple client functionality. So, as we were required to implement a MultiThreaded implementation, I set out to add threads. After much trial and error I figured out how to refactor my code to incorporate threading. Honestly, I was unsure about the benefits of threading and this is what I leveraged in the Part 4 while I did my performance evaluation. My previous implementation already allowed arbitrary number of clients to connect at arbitrary times.I also satisfied the other 3 requirements. Finally to implement graceful disconnects, I implemented a shutdown-like sequence when the C key is pressed where the client sets a flag to shut down and sends the server an exit prompt on which the server deletes the clients key-pair from its registry thus effectively removing it from the server and it reflects on all other clients. Following this the client calls its window.close() and gracefully disconnects from the server. Also all clients were inherently designed to run on different processes so that requirement was already fulfilled by design.

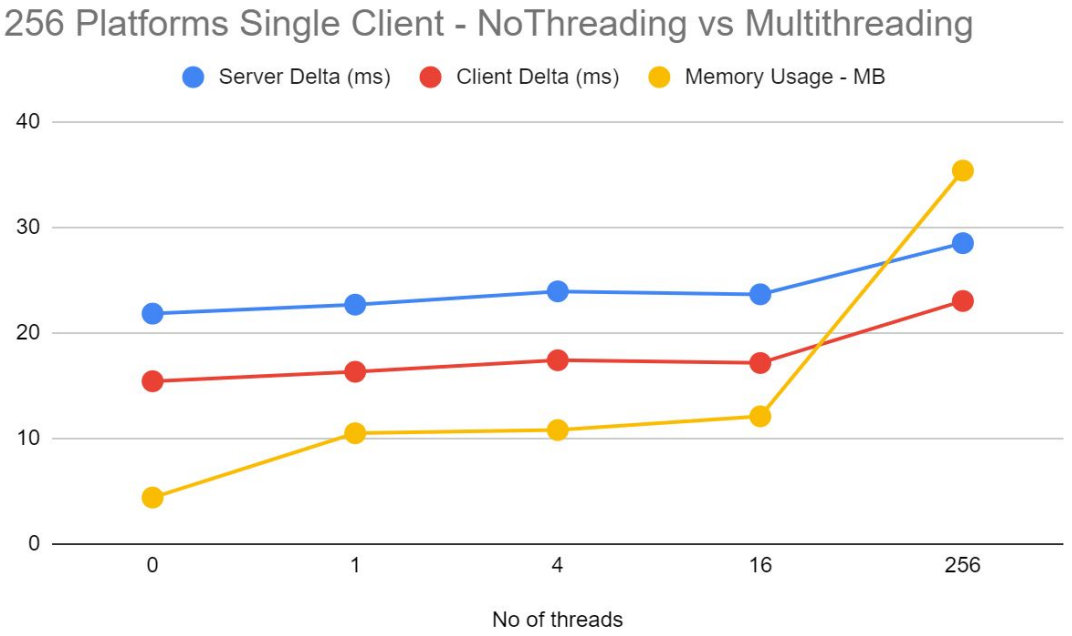
**Part 3 :** This part was relatively straight-forward as compared to implementing the game model, achieving threading and implementing graceful disconnects. I had to redisgn my client side to incorporate multiple platforms and moving platforms (which I initially hardcoded and eventually figured out how to use vectors to represent as it was req for performance evaluation). I also added a deathzone in each screen as well as a big lower deathzone to catch the cases where the character accidentally falls of an edge which wasn't intended to be a deathzone and falls to infinity. I implemented a spwan point which I chose not to move to basically simulate the character always respawning on the screen that he died in. The side boundary took some testing to accurately figure out since it wasn't working properly. Also one bug that I didn't account for was in the second moving platform the platform moves up and down, so I didn't implement a onStand() to make the character's relative position move along with that of a platform hence the platform will translate through the character in such cases.

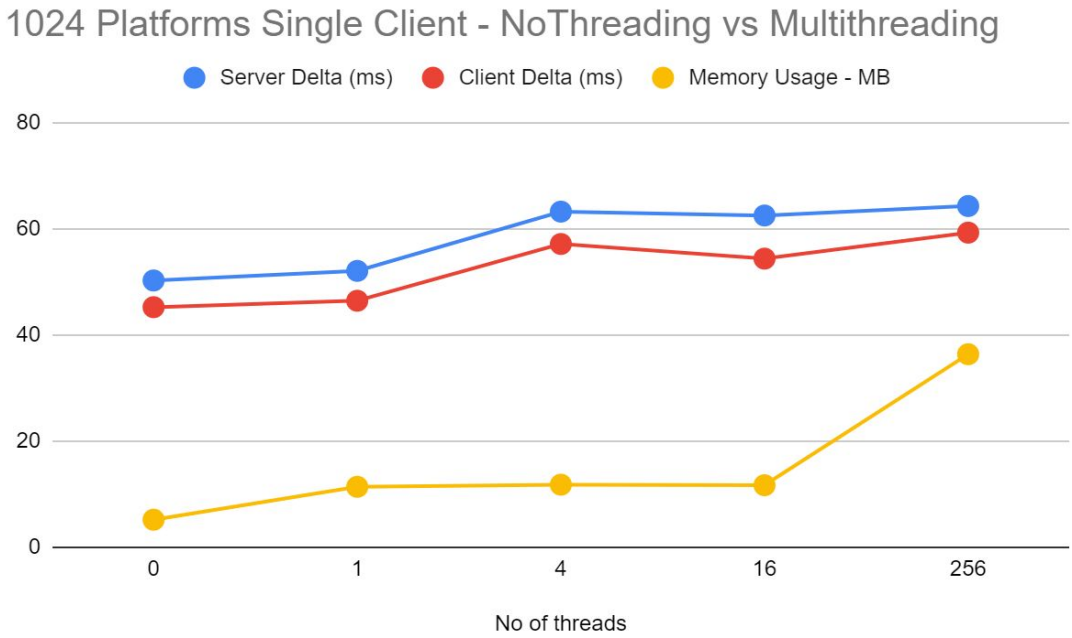



I implemented lateral translation on colliding with the side boundary where I moved all the objects a fixed amount to the left or right depending on which screen the  object was on. Here is a view of the entire game at once :



**Part 4 :** Since I was not convinced with the concept of threading, I found it most useful and interesting to compare the programs with and without threading. I had to modify significant portions of my code to incorporate both threading vs nothreading as well as an arbitrary number of platforms as well as threads. Once I did the modifications, I got the following observations :




The testing environment was as follows :
Used 256 / 1024 moving platforms with the other elements remaining the same as in the figure with the game layout. Apart from this implemetnted a frame_delta counter and also computed a running average. Computed the frame delta for 100 frames to calculate the above values since once we normalize over a very large set of iterations there are very insignificant differences in the models since the scale of the model is small. It was tested on my personal gaming PC. so the results will vary based on the system as well. I used the same runtime model since I modified my existing code as a stand alone application to run this part.

The following observations were noted :

- The overhead of thread handling for a single client far outweighed the benefits. No Threading consistently outperformed threading in terms of frame deltas on client and similar loop time counters on the server as well as memory utilized.
- Though at around 16 threads the slight benefit of threading is noticed as all the values slightly reduce as compared to botht the neighbouring measurements.
- On introducing multiple clients (i.e. 4 in this case) the memory usage of no threading equals that of the 256 threaded counterpart and the client delta time shoots up to twice the amount of the threaded variant which suggests that on multiple clients connecting, the time taken to compute the values of the positions of each moving platform per client(total 4 * 1024 calculations vs 1024 sent to 4) request signinificantly slows down the performance.
- Despite performance metirc wise the threading variant performing better, due to the parrelization of the tasks there is a slight difference in the position of the

moving platforms when computed on different threads vs a fixed thread but the overall game play is significantly smoother. So if we use a set of threads per scene we should have the best performance.

- At 1024 moving platforms, the physics started to break down at times. The client would move beyonf the platform since the time taken to compute all calculations resulted in the client jumping further than the bounding box of the static platforms and hit the universal deathzone below. Sometimes it even skipped that and fell to inifinity.
- I have attached the excel sheet with my calculations as well. I have also included the code i used for the scaleable server as well as the scaleable client. I didnt not have time to incorporate 2 directions of motion in the scaleable variant and hence I have only included it as .cpp files.