CSC 581 Homework 3 Writeup

Jae Jimmy Wong

1 Introduction

In this homework, I designed and implemented an object model and two types of networking architecture for a multiplayer 2D platformer game. The object model supports an unlimited number of platforms and side-scrolling scenes. The performance of the two types of networking architecture is being experimented and analyzed.

2 Game Object Model

In this 2D platformer, the game model contains six types of objects. Starting with *Static platforms* and *Moving platforms*, they have properties that are almost identical to each other. The main difference is that a *Moving platform* has to store its current velocity in order to calculate its next move. For each server loop, there is a phase for executing object movements. During that phase, *Moving platforms* move their positions according to the velocities and adjust the velocities to maintain a repetitive moving pattern. The next ones, *Characters* are the most important objects in this game. In addition to the display-related methods, the model also has to handle player controls and object collisions. To feature an intuitive player control system, left-right movements and jumps need to be handled separately. The design for left-right movements is straightforward. A horizontal velocity is applied to the character when the player is pressing the left key or right key. For jumping, the object model needs to solve the simulation of gravity. The solution is that a constant downward velocity is added to all character objects at the start of movement calculation. It is necessary to apply gravitational acceleration at the start because the acceleration has to be removable while applying object collisions later on. Compared to other systems in the game, the object collision system has an important trait: it involves multiple objects at the same time. Therefore, a common interface has to be established among all collidable objects. The *sf::FloatRect* is selected for the purpose because it can be generated from the SFML function *getGlobalBounds* which exists on all *sf::Shape*. The object collision system has three main differences compared to a naive collision-by-intersecting approach. First, though *sf::FloatRect* has a built-in method that calculates if two rectangles intersect with each other, it is not useful in a 2D platformer since it prevents characters from jumping on a platform from its bottom. Thus, this game implements its own collision detection by determining if a character has a position that is close enough to climb toward a platform. To be close enough, the character needs to have overlap on the x-axis to the platform and touch the top 10 pixels of the target platform. Second, the position of the character is slightly adjusted while colliding to make it perfectly aligned to the platform. The reason why the position needs to be adjusted is to prevent it from dropping off a vertically moving platform. Without such adjustment, the character will be slightly lower than the moving platform each time the platform moves up. While accumulating enough height difference, the character will stop colliding with the moving platform and drop off from it. The third difference of the system is the *supported* flag. This flag helps character objects determine if they can execute jumping in the current position. Compared to re-calculating collisions when the player wants to jump, the usage of the flag not only improves the performance of the game but also helps decouple the player control and object collision system. The last three types of objects are common in one thing: they are all invisible from the perspective of a player. This makes them much simpler than the above three object models. They do not need to have the *get_shape* method nor need to store display-related properties such as color. All they needed to do was containing the positions or sizes. The combination of these six types of objects covers the common feature required to create a 2D platformer.

3 Multithreaded, Networked Scene

To turn the single-player game into a multiplayer version, 0MQ sockets are used on both the server and the clients. The network architecture of this game consists of two kinds of channels: object channel and message channel. The object channel is used to publish the present states of all objects that exist in both the server world and the client world. Every 10 milliseconds, the server updates the server world and sends the states to all clients. This design saves the CPU time used for serializing game objects for each player, since the parsed JSON string is reused on all clients. However, the time difference will only be noticeable if there are enough clients. The message channel is aimed at client-to-server information delivery. The information includes player joins, player leaves, and character movements. For the server to distinguish the type and the source of the information, all JSON message sent through the message channel has two common fields: *type* and *client_id*. When a client program starts running, it randomly chooses a *client_id* from 2147483647 numbers. This *client_id* generation process does not need to communicate with the server, thus improving the startup speed of game clients. On the server side of the message channel, there is a dedicated thread that is created for handling these messages. The server handles the messages one by one but each message only takes a very short time, because there are no world calculations while handling the messages. It only applies changes to individual client objects. By utilizing these two channels properly, the game is able to connect tens of players on the same scene and still works fluently.

4 Performance Comparison

In the treatment group of this performance experiment, the socket type of 0MQ has been changed from pub-sub to request-reply. In the original design, the server and the client each have a publisher socket and a subscriber socket to form a dual-way communication. Since the server does not have a list of client IP addresses and ports, the design has to be different. For both the object channel and the message channel, the client holds the requester sockets and the server holds the replier sockets. Furthermore, the characteristic of a replier socket demands the server to have an additional thread to handle requests. In the experiment setup, three clients connect to a server at the same time. The game loop time of clients is measured as an indication of performance. The experiment results are significant. In the pub-sub control group, it takes 10.6312 seconds on average to run 1000 game loops. In the request-reply treatment group, it takes only 2.1245 seconds to run 1000 game loops. That is a 401% performance increase. However, after conducting more experiments with different setups, the result shows that the performance of request-reply design becomes worse with more clients. It takes 3.97748 seconds on average for a setup of five clients. Compared to pub-sub design, whose game loop time barely changes with 10 clients at the same time, the time increment seems to be too much. To summarize, the request-reply design is better when there are a limited number of clients and the pub-sub design is better when there are a large number of clients.