Clash Wars Design Document

Nelson Ong
Parth Pathak
Blake Jones

Idea

Our idea is a two-player tapping Android game, where players can use DragonBall Z characters to face off against their opponents in a reverse tug-of-war style match determined by tapping. The winner is the player with the faster tapping speed or the player who is ahead when the countdown timer reaches zero. Each client character sprite will have a beam emanating from their hands and will meet, or clash, with the opponent's beam in the center. Every time a tap is registered from a client the beam will be pushed slightly toward the client's opponent. This will continue until one player's beam reaches the the opponent's character sprite. The client will be implemented in Java, making use of LibGDX, and run on Android devices and the server will be written in C++ and run in a UNIX environment hosted by Amazon Web Services.

Implementation

Threading

The final revision of our code implemented seven different types of threads: Main, Connection, TimeoutCheck, TriggerShutdown, Game, Client, TriggerClientShutdown. The flow of how each thread is created is outlined in Figure 4 below. The exact implementation details are outlined in the "Implementation" section below.

Shared Resources

Each game thread on the server accesses a tracker variable, shared amongst the two communication threads. This tracker variable keeps track of the percentage from one client's perspective and is updated after each tap. Then, the percentage is sent back to the tracked client while (100% - tracker percentage) is sent back to the other client. The clients update their views based on the percentages they receive from the server.

Since only one thread can access the tracker variable at once, we used a mutex for synchronization. The 2 client threads wait for the mutex to be unlocked before proceeding to update the tracker variable and sending the tracker percentage back to each client through the 2 stored sockets.

Procedure

During the implementation, we realized more threads were needed than previously thought. The diagram below attempts to illustrate the overall behaviour of the system. The initial thread, Main Thread, spawns a new thread called Connection Thread and then just waits on user input to signify that it should shutdown. If, at anytime, the user enters "kill" into the Main Thread the thread will handle the graceful termination of itself and the rest of the threads that are created during the lifetime of the program.

```
int main(){
    bool kill = false; // Used to terminate all active threads
    std::string killResponse; // Used to initate termination process

    // The main thread only spawns one thread that manages all connections to
    // the server by dynamically spawning "response threads"
    std::thread *connection = new std::thread(connectionThread, &kill);

    do{
        std::cout<<"Type 'kill' to close server: ";
        std::cin >> killResponse;
        if(killResponse == "kill"){
            kill = true;
            break;
        }
    } while(true);

    // Wait for the connection thread
    connection->join();
    delete connection;
    return 0;
}
```

Figure 1: Main Thread

Next, we have the Connection Thread which is responsible for handling new connections from clients as they attempt to establish communication with the server. Each new client gets pushed onto a deque. Once at least two clients have been added to this deque, the Connection thread will begin the process of creating a game thread. Before it actually spawns the thread, however, it does a heartbeat check of the two clients that will be passed to the game thread in order to ensure that an active connection is still present. The heartbeat check is implemented by spawning two timer threads (one for each client) that function in a similar manner to the Linux "watchdog". That is, they consist of a timer that must be stopped or else it will close the socket to the client that failed to provide a response to the heartbeat check.

```
void timeoutCheck(Communication::Socket* client1,int timeout, bool *killVar){
    // Wait for a specified timeout period before closing communication with client
    // Used to verify client heartbeat incase client has disconnected while in queue
    std::this_thread::sleep_for(std::chrono::milliseconds(timeout));
    if(*killVar == false){
        // Close socket
        client1->Close();
    }
}
```

Figure 2: Heartbeat check timer

After the server has verified both client's heartbeats, it spawns a Game Thread to handle communication with the two clients. To facilitate communication with both clients, the Game Thread spawns two Client Threads used to read from each of the client sockets and each Client Threads spawn a Client Shutdown Thread to respond to the kill trigger from the Game Thread.

The Game Thread keeps track of the game by using a tracker variable that both the Client Threads modify during the course of the game. This tracker variable is protected using a mutex and is used to track the position of the beam (Shared Resources). Once the tracker variable reaches a specified minimum or maximum value, the game is considered finished and a win or loss code is sent back to each client and the Game Thread terminates after joining all threads it spawned.

In the case of a client disconnecting from the server, the Client Thread contains a check for a dead socket right before it attempts to block on the read. This allows the game to continue as if the disconnected player was not playing.

```
try{
    // Block in case client leaves mid game (Stops thread from proceeding
    // to Read(message))
    while(!(client->open) && !*killClient){ // Checks for termination
        std::this_thread::sleep_for(std::chrono::milliseconds(250));
    }

    // Attempt to retrieve message from client
    client->Read(message);

} catch(std::exception e){
    // Break loop in case of error
    break;
}
```

Figure 3: Handling client disconnecting mid-game

On the next page, a complete diagram of all the threads composing the server can be seen. Red fields designate shutdown threads, yellow correspond to threads responsible for checking client heartbeats and blue are logic threads that do various functions.

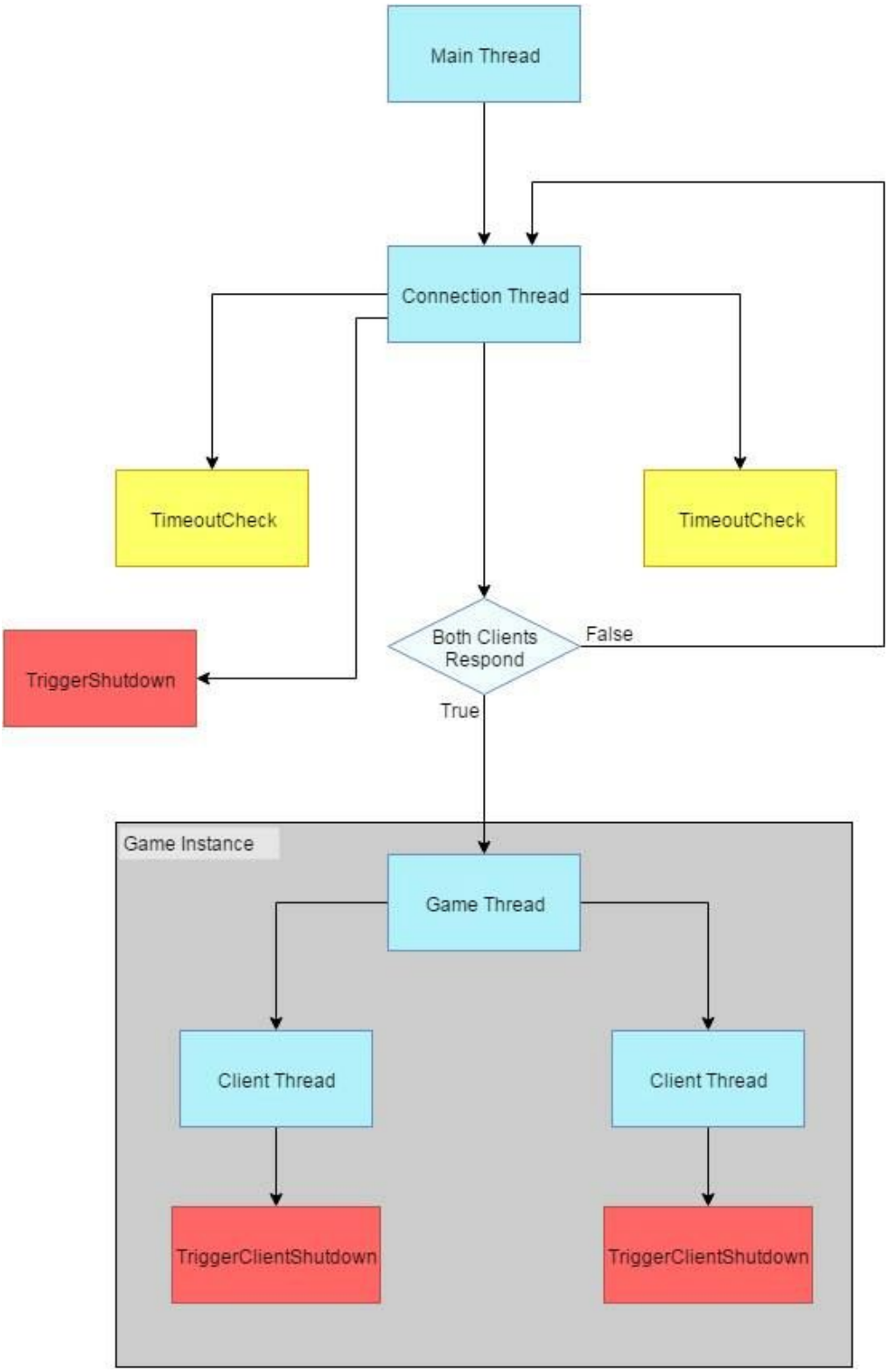

Figure 4: Server Concurrency Visualization

Client Side

The client is written in java and uses the wrapper framework libGDX to create an Android application. The following is an image of the client while in game:

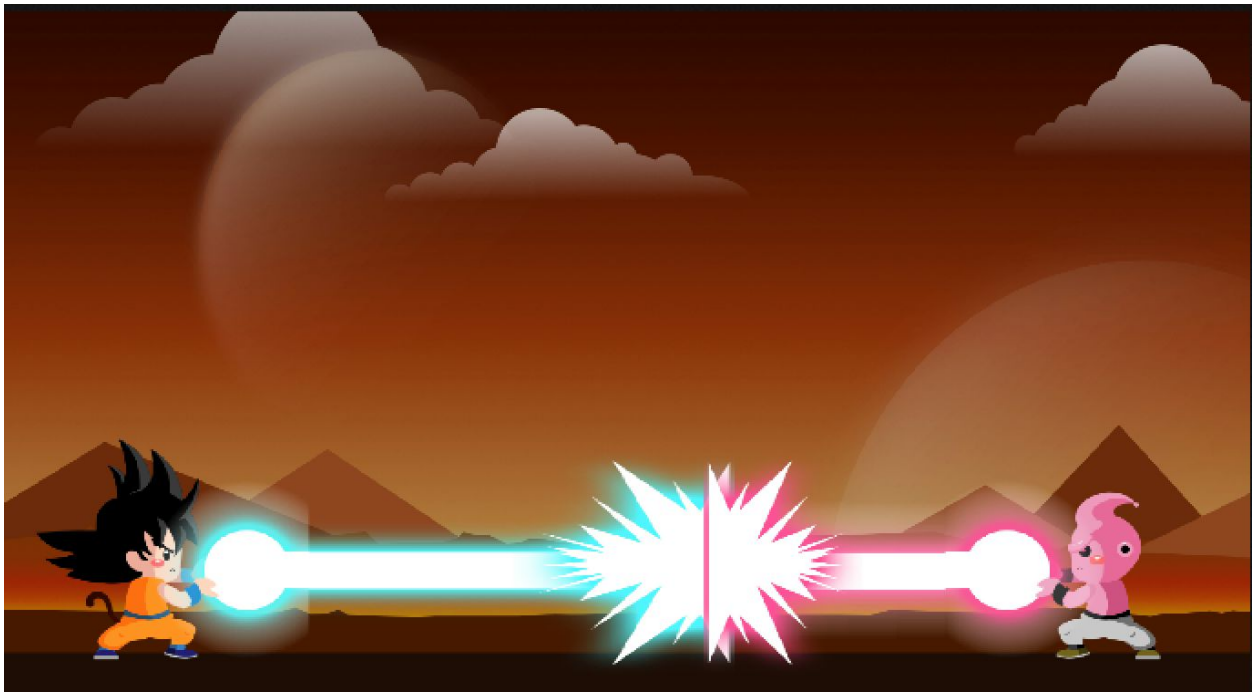

Figure 5: Gameplay exhibit

The client can be modified to communicate with a server by changing the Connection.java file:

```
public class Connection {
    // Server variables
    public static final String SERVER = "54.201.252.9";
    public static final int PORT = 2200;
```

Figure 6: Connection.java server constants

Note: If character sprites are not loading properly, a black box will appear around them; this is a visual bug that happens due to textures not being loaded into memory since some phones cannot process images with large widths.