TANK GAME PROJECT REPORT

INTRODUCTION

Our project aimed to make a PVP (Player Versus Player) tank game hosted on a multiplayer server using an FPGA as a controller to dictate movement and fire bullets. Each player initially loads into a lobby, waiting for the other players to ready up. Once all players are ready, their tanks will spawn in a random place within the map. Each player begins with 10 lives and the aim is to be the last tank standing.

THE ARCHITECTURE

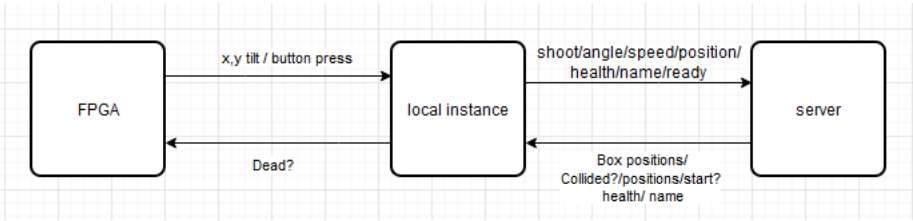

Figure 1: Overall architecture of the project

Above is an overview of the architecture of our project. It is made up of three main components: the FPGA, the local instance on the player's computer and the server ran on AWS.

THE FPGA ARCHITECTURE

The FPGA handles controlling the movement of the tank and allows a player to fire a bullet. The movement and position of the tank are calculated from the tilt angle of the board collected by the accelerometer. This data is then passed through a 10-tap filter which is then streamed to the computer. Similarly, data on whether buttons are pressed is also streamed to the computer to show whether a player has fired a bullet. The streamed data is collected in a file in the directory of the local instant. The FPGA board also utilises "health" data sent from the server back to the local instant. The FPGA board achieves this by processing how many lives each player has and lights up the corresponding amount of LED lights.

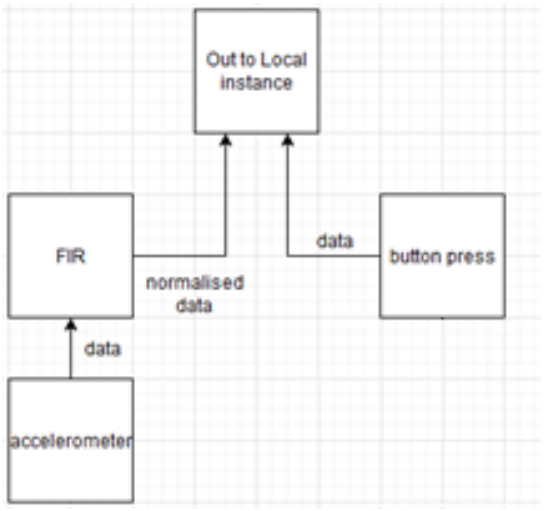

Figure 2: Architecture of the FPGA board

THE LOCAL INSTANCE ARCHITECTURE

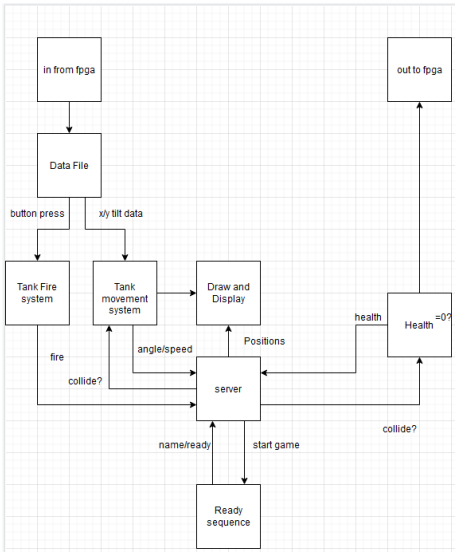

Figure 3: Architecture of the Local Instance

The local instance is formed of various functions which interpret the readings in the FPGA data dump file, send the actions of the player to the server, create and move bullets, and perform collision checks between the game objects. It is also responsible for displaying the game on the players' computers. The instance will receive data from the server telling it:

- When the game should start
- The positions of other players, obstacles and any new bullets that have been created
- The health and name of all other players

If the player has collided with a bullet object, the player will take damage, but if it collides with an obstacle or is out of bounds it will change the velocity of the tank to the negative equivalent, creating a bouncing terrain effect.

When the player runs out of lives it will send a message to the server and the FPGA. The instance will then keep displaying the other alive players on the screen while replacing your tank with a tombstone to mark your last position.

THE SERVER ARCHITECTURE

The server is primarily responsible for sending the required data back to the local instances. Data that is received includes:

- Player information for each client connection
- Boxes that need deleting
- Bullets that need creating
- A command from the client

The server keeps a record of players' data (names, current health, position, etc.) which it forwards to all local instances as well as processing data that decided when the game started which is based upon whether it receives the ready command from all players.

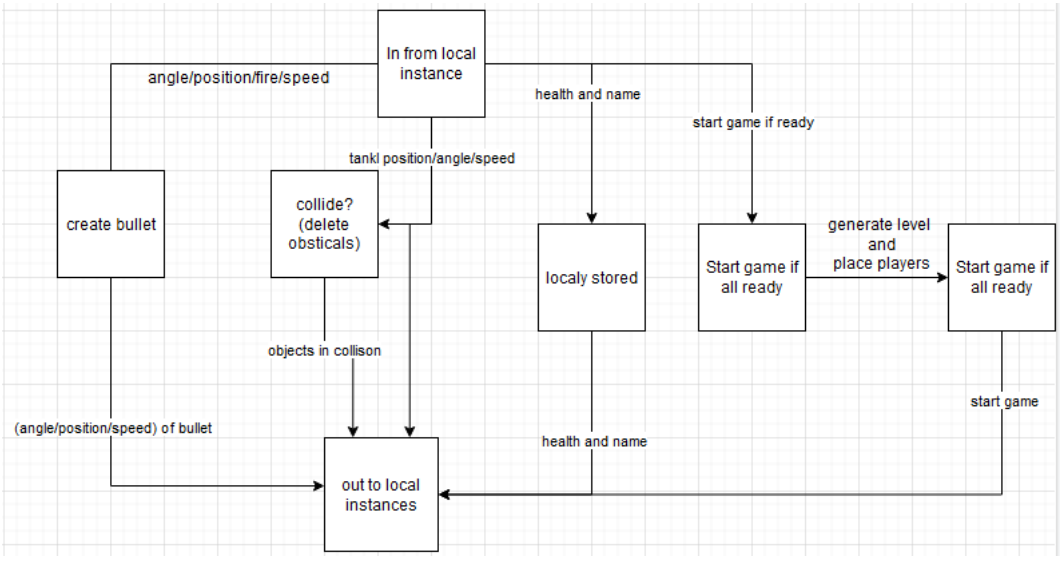

Figure 4: Architecture of the server

PERFORMANCE

When comparing the performance of the game being played on the EC2 server, to playing the game on the local host we realised that the performance on the EC2 server was noticeably worse. The game is designed to display 30 frames per second, and whilst this was achieved when playing locally, when playing off the EC2 server this dropped to around 5 or 6 frames being displayed per second. This made playing off the EC2 server feel slow and unresponsive as it would take up to 6 or 7 seconds for controller inputs to be fully carried out in-game. This performance issue was due to the AWS server we used, not having enough processing power to run the game at the same rate our computers could, and so could be fixed by upgrading to a stronger server instance.

We opted instead, to reduce the workload of the server and increase the workload of the host computers, by moving certain tasks like bullet creation and movement to the host computers. Whilst this did not fully solve the issue, it reduced the controller input lag to around 3 or 4 seconds, which was much more playable. Another approach we investigated was swapping to a different protocol like UDP, instead of TCP to increase the speed of transmission. We decided against this in the end, because of how we changed the game to reduce server workload. Individual clients handled the spawning of bullets and sent this action to the server without storing this action anywhere else. If UDP was used and a packet of data was lost, the server would not know about a bullet generation incident, and so when transmitting the list of active bullets to all the other clients, the newly generated bullet would be missing. This could have potentially been fixed by storing and sending a list of all the bullets generated by a client to the server, rather than a single bullet generation instance, so that even if a packet of data was lost, in the following packet all the data would still show up. However, we decided against this, because of the following issue we ran into.

When transmitting data between the client and server we ran into an error where the receiving end would not receive the full amount of data expected. As the receiving end received less data than it expected, it was not possible to decode the received data, and so we ended up with a “truncation error.” To fix this we reduced the amount of data we sent across, mainly by decreasing the number of terrain items we wanted to spawn into the map and opting for a locally stored list of active bullet items instead of sending each object across the connection.

On the FPGA board, we utilised a 10-tap filter to smoothen the outputted tilt values. Without the filter, we found that the tilt values be too sensitive to any change in the controller orientation. As it was somewhat challenging to hold the FPGA perfectly straight, we opted to create a dead zone, where the controller output would be 0 regardless of the controller orientation, around the neutral x and y tilt positions. The 10-tap filter implantation we used meant that the FPGA board had to work with floating-point values, despite the fact the DE-10 board does not support floating-point arithmetic directly. Whilst we could have implemented the filter differently so that we wouldn’t have to do floating-point arithmetic on the board, we decided against it, as the FPGA board was still outputting controller values at a fast enough rate.

KEY DESIGN DECISIONS AND OPTIMISATIONS

Initially, our idea was to build a plane game using the accelerometer data for pitch and yaw control, however, we realised that it would require a dynamic camera position to properly function, so we moved to a 2D game instead. We knew we wanted a multiplayer game as this would allow more interactions with multiple clients and would utilise the server the most.

The first design choice to be made was which framework was going to be used for the game design. We decided to use Pygame for several reasons. Pygame is a module for python, the language in which the server-client connection was going to be made, and a language we were all comfortable with. It is a lightweight package that allows for computer graphics within game design, and a package that one of the team already had experience with. This meant that the focus could be on the development of the product instead of learning unfamiliar technologies.

Our inspiration for our project was the game Tank Trouble. In our version of the game, we had to decide whether we wanted bullets to rebound off terrain or destroy it. While initially neither was implemented and terrain was indestructible, we realised that while bullet bouncing was workable, it needed clear visuals, while destructible terrain allowed more randomness for generation, and the terrain pieces to look better.

Another consideration was what data should be sent back to the FPGA. We were considering displaying lives on the LEDs, but from a gameplay perspective, keeping them all on the screen made more sense, rather than encouraging players to look down. Whilst we had the implementation for this, we decided not to return any data to the FPGA board, choosing to streamline and focus on the gameplay loop instead.

A major consideration was what information to process on the FPGA, host machines, and the server. The most powerful processors were the host machines, so we tried to put more strain on the hosts. We achieved this by moving collision detection and bullet movement, originally performed on the server, across to the local host machines. Moving these processes off the server and performing collision on each client reduced the amount of processing the server had to do, streamlining the connections between client and server.

One clever optimisation that was made to lighten the collision processing load, was to first perform rectangular collision before performing pixel-perfect collision. Under the earlier system of only performing pixel-perfect collision, a lot of processing power was wasted checking every pixel of each game object with every other object. By first performing rectangular collision and only then doing pixel-perfect collision only if the rectangular collision returns true, we drastically reduce the amount of processing power required for the collision detection.

TESTING APPROACH

The implementation of our project was done in a modular fashion in which each module was tested before moving on to the next. The diagram below shows the order in which we did our testing. There are three main branches: FPGA, Game (Local Instance) and Server. However, we implemented the server after we developed the first version of our game. We decided to develop the game and the FPGA controller separately. This meant that the game could be worked on using keyboard inputs straight away, rather than having to wait for an FPGA controller to be finished.

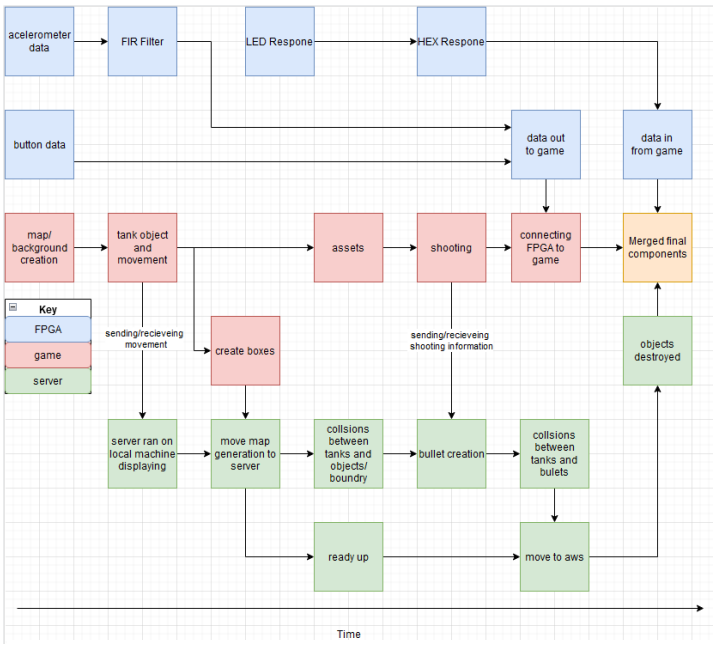

Figure 5: Diagram outlining our testing approach for the entire project

In terms of testing the game design, the best approach was to simply run it and ensure that it functioned as expected. Testing often and after each added feature, ensured that the game remained functional throughout the development and testing period. Once the FPGA controller had been designed, we had the choice of testing the game with both the keyboard controls and the FPGA controller. Testing with the keyboard controls was quicker, however, we made sure to test with the FPGA controller whenever a change was made that would affect the relationship between the board and the host as well as at given intervals to ensure functionality.

Testing on the FPGA was slower due to the need to flash the board and download the .elf file each time we made a change. We built up and tested the controller step by step so that any new features that introduced errors would be easier to spot and deal with. We began by getting both the x and y accelerometer readings to display, and then the buttons. After this we investigated how to send data from the FPGA to the NIOS II terminal, starting with the given program in lab 4 to send some characters across. This program turned out to be flawed, and so after working with a GTA we came up with a new program to send data to the NIOS II terminal. Once this was working, we integrated the two, so that the message being sent to the terminal was now a stream of data that gave the x and y accelerometer values, as well as the status of buttons 1 and 2. Using the python subprocess routine, we extracted values from the terminal onto a text file and then sent these to the server. At this point, the FPGA board could now be used to control the tank in the game.

RESOURCES UTILISED BY THE DE-10 FPGA

The DE-10 board utilises Altera's highly customisable softcore NIOS II processor. This implementation parts from a base design, and apart from a couple of necessary functional units, such as the program counter and ALU (Arithmetic Logic Unit), any number of other functional units and custom peripherals can be added and removed, to suit our application's requirements. On the Quartus *Platform Designer*, a large set of peripherals can be configured and connected to our processor. The first peripheral needed for our game is the accelerometer, connected to the board via the SPI. The CPU samples the data from the accelerometer through the SPI bus, when requested through the clock line. These values are then locally processed on the board and then communicated to a program on the host computer. The JTAG-UART peripheral is added to the platform design for this purpose. It allows for serial asynchronous communication between the 2 devices. LED peripheral (10-bit output) and HEX display (six 7-bit outputs) peripherals are also implemented on the chip architecture, as the LEDs on the board display the life count of the player live, and the Hex display shows the position the player finished in at the end of the game. However, these two features have not been fully implemented on the game side of things. Lastly, a 2-bit button peripheral is added, to allow shooting and other button functions. All the peripherals are connected in a master-slave fashion, where the processor controls all the processes and serves as the communication hub.

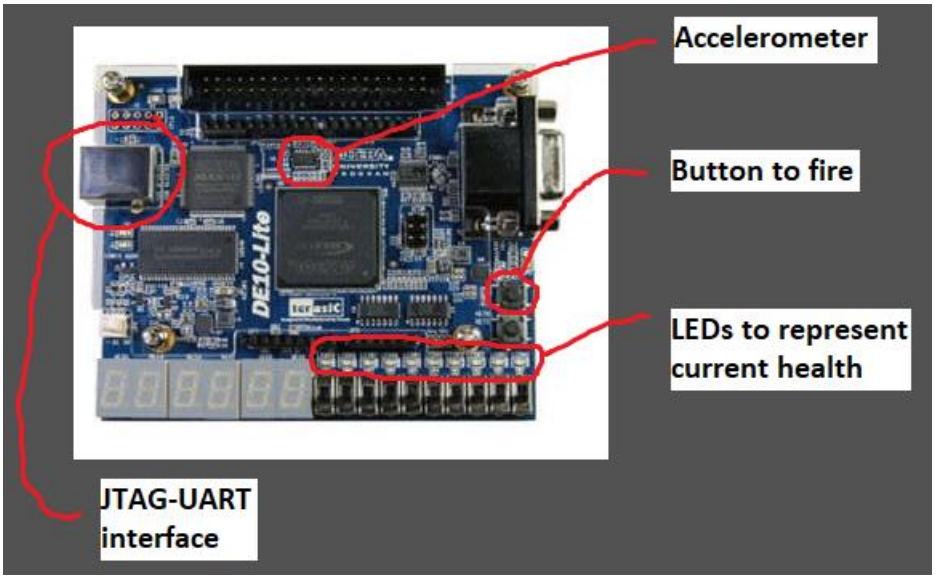

Figure 6: Labelled diagram of the FPGA board