Accelither

Information Processing Coursework

Group 1

22 March 2024

Maximilian Adam
CID: 02286647

Lucas Ng
CID: 02014979

Hanif Rais
CID:02234780

Benny Liu
CID: 02015180

Sophie Jayson
CID: 02254802

Mateusz
CID: 02257454

Imperial College London

1 Introduction

Our project is a game based on the online multiplayer game slither.io. This project integrates the accelerometer in the DE10-Lite FPGA board, processing offered by the NIOS-II CPU and AWS to host our game. A general overview of how the whole system works can be seen in the diagram below:

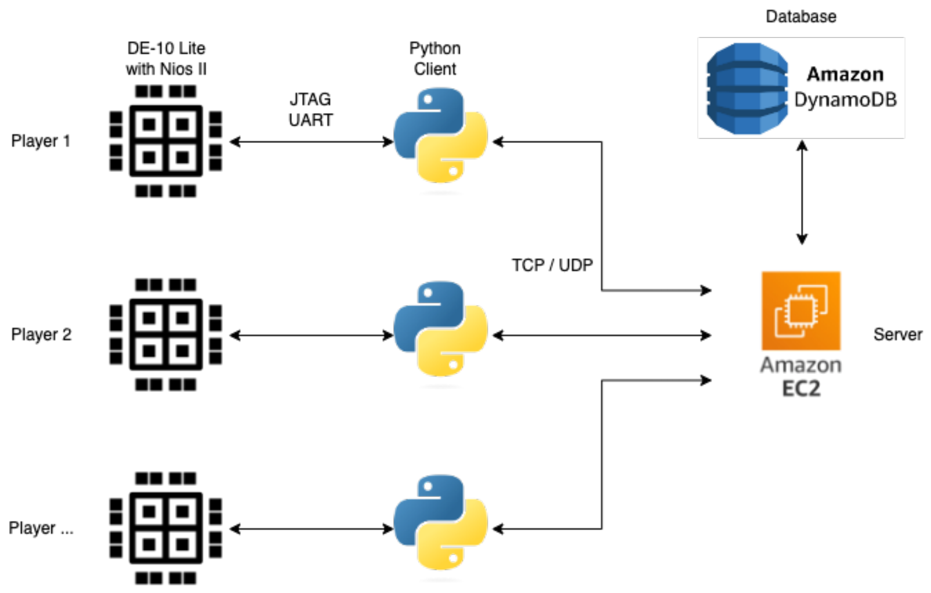

Figure 1: Overall system architecture

The gameplay is rather simple: tilt the FPGA in the direction you would like your snake to go in. Speed up or slow down with your two push buttons. Your goal is to steer your snake to collect as many orbs as possible and get kills by forcing other players to collide with your snake character's body. The highest scoring player gets to be on the top 3 leaderboard.

Our project places great emphasis on the overall smoothness and responsiveness of the game, and extensive optimizations were placed throughout the game to make sure that the delay between the player input and the game updating on the client is kept minimal. In the sections below, placed in the order of data flow, will elaborate in detail the design of the system and its optimizations.

2 Inputs and FPGA

Starting with the players' input to the game, the FPGA provided for us to use is the DE10-lite, which contains an accelerometer, 7-segment displays, LED's, switches and buttons. The gameplay only requires the accelerometer (and we decided to use the X and Y axes as that leads to the most comfortable way to play), however, as we considered additional features and potential expansion of the game, we decided that we should implement full access to I/O on the board.

Our final specifications for the input system are the following:

- Obtains all inputs on the DE-10 lite and sends them to the player's device;
- Allows the player's device to specify all the outputs on the DE-10 lite;
- Perform these operations as quickly as possible: develop for as high of a frequency and low of a latency as possible;
- Includes a Python module which provides simple programmatic access to all the inputs and outputs;

2.1 Obtaining inputs

Before implementing the inputs, we first consider how we communicate these to the device. We used JTAG and NIOS-II as they appear to be the only option, and provide the necessary duplex communication that we need.

2.1.1 Switches and buttons

These were easy to implement: we have direct hardware access via pins on the FPGA, which we pass into a parallel input into NIOS-II.

2.1.2 Accelerometer

The accelerometer requires communication with I2C or SPI to access its readings. Also, these readings are much noisier and require filtering before being used.
We had multiple options here:

1. Use the provided SPI communication in NIOS-II, then stream all readings to the player's device;
2. Same as above, but perform filtering in NIOS-II software before sending to the device;
3. Implement our own hardware solution;

We immediately disregarded 1 as it adds unnecessary processing to the device responsible for rendering, and it doesn't take advantage of the FPGA. However, experimentation showed that this allows the input frequency to be slightly over 1 kHz.

We know from experimentation that 2 takes about 1.8 ms, subject to change based on number of filtering samples. Base on this, we estimated that the complete I/O would be capable of at most 500 Hz. While this is more than sufficient, we were disappointed by the decrease in frequency. We also noticed that, according to the accelerometer's datasheet, it is by default only outputting values with a frequency of 100 Hz, and we were uncertain on how to change this programatically, nor could we take advantage of the highest frequency of 3.2 kHz (not only could we not send values that quickly to the player's device, we also could not poll the accelerometer that quickly to have all available data for filtering)

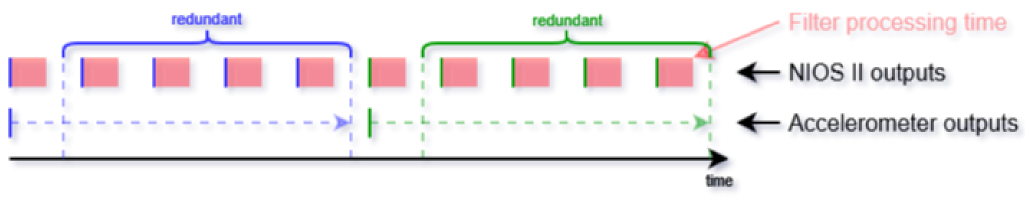

Figure 2: Timings when software filtering: Note the time spent processing and redundant NIOS-II sends.

Therefore, we decided to go with 3. We made hardware which:

- Uses the DE-10's 50 MHz clock to form a 3.2 kHz clock to form the basis for the SPI communication;
- Upon startup, ensures the accelerometer's SPI is in 4-lane mode and is outputting at 3.2 kHz;
- Polls the accelerometer at 3.2 kHz, stores the most recent values, performs filtering in hardware, and passes the most recent filtered value as a parallel input into NIOS-II;

As we want to use the accelerometer to get the current angle at which the board is held, the filter simply averages the last N readings (we do not want a frequency response). We observed less noise as we increased N, but increased latency. Based on our collective opinions, we settled on N = 4, as it provided no noticeable input delay (goes from 313 µs to 1.25 ms), while producing much less noise than the unfiltered values.

2.2 Interacting with the FPGA

The NIOS-II terminal, which the device uses to send data to and receive data from the DE-10, dictates the frequency with which we poll the board. We achieved this by having the board respond with its inputs when we send the outputs. This payed off when we found that the maximum communication frequency varies very noticeably from device to device (some devices can do over 1 kHz, while some can only do about 800 Hz).

We had the challenge of wanting to poll the DE-10 at the highest frequency that we can, and wanting to just use the most recent of these values in the game, removing the delay of communicating with the DE-10 from the game program. Our solution for this was to use shared memory. We made a shell script which used Python and its multiprocessing.shared_memory module to communicate with the DE-10 at a specified (and adjustable) frequency, storing the most recently read values in shared memory reserved for inputs, and sending the most recent values in shared memory reserved for outputs.

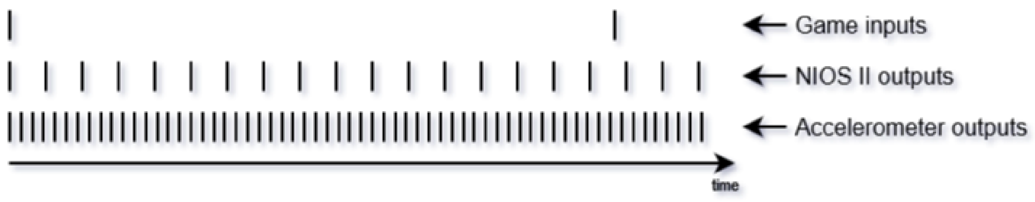

Figure 3: Timings when hardware filtering. Note: the NIOS-II output frequency is double that in 2 due to negligible processing times.

If we sum the 3 accelerometer readings (each of which are 16 bit), 10 switches, and 2 buttons, we get 60 total bits needed for input. If we consider 6 7-segment displays and 10 LED's, we get 52 total bits needed for output. Therefore, the input shared memory is 8 bytes and the output shared memory is 7 bytes. For the actual communication, we used hex (due to simplicity), so we send 13 characters to the board and receive 15. We implemented the encoding/decoding in NIOS-II and Python, then integrated the Python code with a shell script to make a program which acts like a driver, always running in the background to make the device work. Finally, we made a Python module which interacts with the shared memory and processes the accelerometer readings further to convert them to an angle.

3 Server/Client

After receiving the accelerometer data from the FPGA, we now need to use this data to control the snakes and render the game to the user.

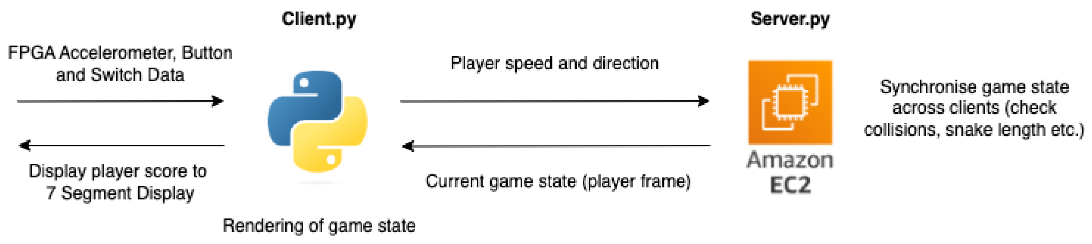
Client.py
Server.py
FPGA Accelerometer, Button and Switch Data
Display player score to 7 Segment Display
Player speed and direction
Current game state (player frame)
Amazon EC2
Synchronise game state across clients (check collisions, snake length etc.)
Rendering of game state

Figure 4: Data flow between client and server

To achieve this, we created two main python programs, one running on the server and one running on the client:

1. The client is responsible for continuously streaming data to the FPGA through TCP and rendering recevied data from the server.
2. The server is responsible for receiving the accelerometer data, changing the direction and speed of the snake controlled by the client, and then relaying the data of objects within the client's frame of reference to be rendered.

We took this centralized approach to ensure that individual client game states do not diverge and are consistent. This could occur due to differences in propagation delay and dropped packets. By having only one true game state hosted on the server, we eliminate this problem. In the subsections below, we will go more in-depth into some of the main functionalities that enable smooth gameplay on the client side.

3.1 Client/Server Communication

3.1.1 Networking Protocol

TCP was chosen for its low error rate and better latency stability. Early in our testing, we found out that UDP had the tendency to drop packets. Since our game worked by constantly sending back positional data, this caused a lot of lag when packets are dropped since the client would have no game state to render.

3.1.2 Data Handling and Compression

The `GameData` class is converted to a dictionary for transmission. `json.dumps()` then serializes this dictionary to a JSON formatted `str`.
We found that a significant proportion of TCP packets are dropped when the packet size exceeds a limit of approximately 1500 bytes. This is likely due to the maximum transmission unit (MTU) size of the network, which is 1500 in most instances. If the packet size exceeds this limit, they are fragmented.

This prompted us to use the Protocol Buffers (protobuf) data format to compress each GameData object. Protocol Buffers are a language-neutral, platform-neutral extensible mechanisms for serializing structured data.

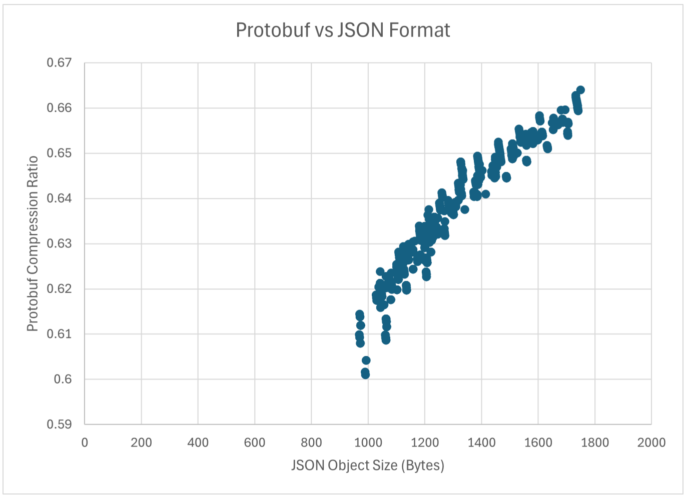

Figure 5: Compression ratio of Protobuf vs JSON

To measure the effectiveness of this data format, each GameData object was serialized via JSON and protobuf in a running game server. In the range of JSON sizes that are typically used in Accelither (900 to 1800 bytes), the compression ratio (protobuf object size ÷ JSON object size) was approximately 0.67, ensuring that each packet size is below the MTU size of 1500.

3.2 Client game state rendering

Pygame was chosen to be the game engine running our game since it offered a straightforward and simple way to draw shapes on the screen, making it ideal for whipping up quick 2D games from scratch given the tight schedule. This choice also gives us an advantage when coding since our group members are all familiar with Python.

Rendering the game, after decoding the server data and converting it back into a dictionary, is as simple as passing in all the cartesian coordinates of the shapes to be drawn into pygame. However, ensuring smooth gameplay, is a trickier problem. When testing, we found out that TCP does not always send the entire package in one go, and it could fragment the data. This causes our client to receive partial chunks of critical game data for rendering, leaving the client hanging to listen out for another timeout cycle. Therefore, we tuned timeout values and sleeps so that the game will not be caught waiting excessively for a late packet while also maintaining acceptable data integrity to ensure smooth gameplay.

4 Database

DynamoDB was selected as our database owing to its seamless integration within the AWS ecosystem, allowing efficient communication with the server operating on an AWS EC2 instance. Furthermore, DynamoDB provides a Software Development Kit for Python, Boto3, enabling easy interfacing with the database, which aligns well with our game server which is coded in Python. In our latest iteration, our database handles players' kills, highscore and deaths. As for the structure it only contains one primary key which is the unique player ID initiated during log-in. The database could update item, delete item and fetch item based on requests from the game server.

5 Testing and version control

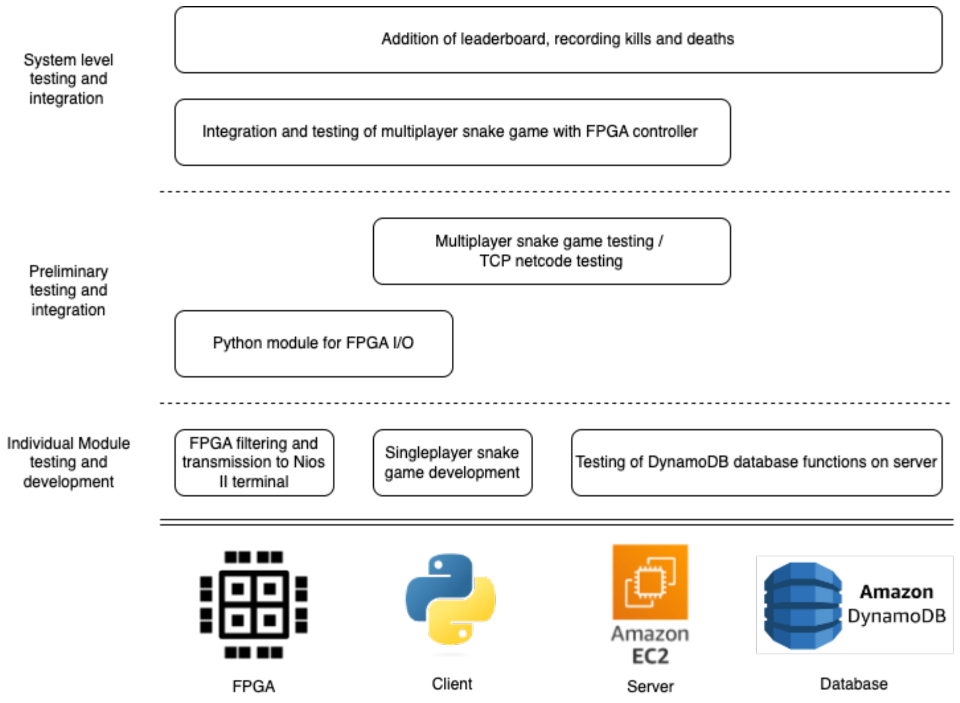
System level testing and integration
Addition of leaderboard, recording kills and deaths
Integration and testing of multiplayer snake game with FPGA controller
Preliminary testing and integration
Multiplayer snake game testing / TCP netcode testing
Python module for FPGA I/O
Individual Module testing and development
FPGA filtering and transmission to Nios II terminal
Singleplayer snake game development
Testing of DynamoDB database functions on server
FPGA
Client
Amazon EC2
Server
Amazon DynamoDB
Database

Figure 6: Testing and development flow

Shown in figure 6 above shows the development timeline flowing from left to right. The project can be broken down into 4 different stages:

The first stage would be to get the data out of the hardware, from the FPGA into the client machine. No progress was made on other parts of the project until smooth, consistent data could be extracted from the FPGA into the client.

After which, reliability of the hardware layer can be assumed, and development of client and server can proceed. This is done in tandem with further improvements to the hardware as well, as the game demanded more hardware interaction further along the development. However, since robust standards and minimum performance were established before, adding features to the hardware was easy.

Lastly, once the client/server interaction was robust enough, the database can be integrated into the project to act as a database for high scores and other player data like kills and deaths.