Information Processing Report

William Huynh, Sara Chehab, Vishesh Mongia, John Yeo, Fred Dakhnenko

March 2024

1 Overview

The system is a game called *glutton.io*, in which the goal of the player is to dominate the arena as the supreme blob. To achieve this, the player must navigate the environment, growing their blobs by consuming smaller ones. This process is encapsulated in the flow diagram in Figure 1.

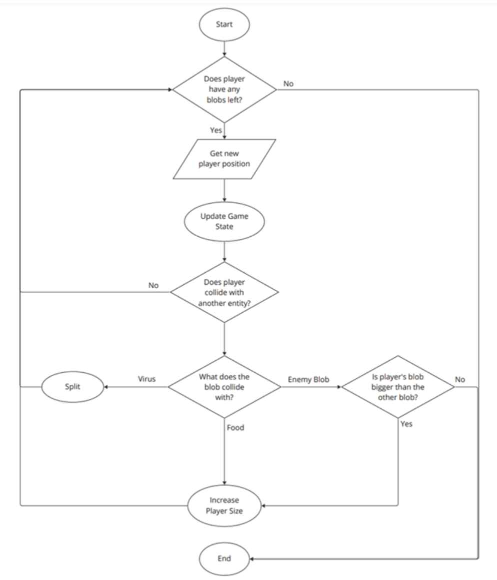


Figure 1: Game flow diagram

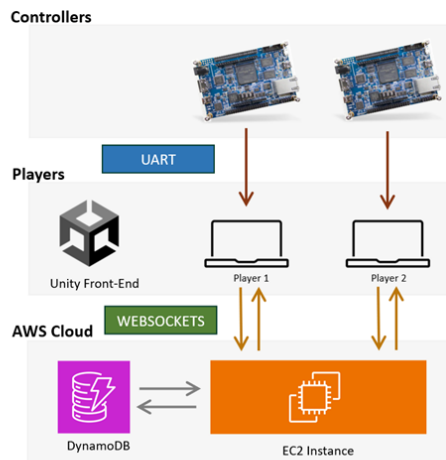


Figure 2: Overall System Architecture

2 FPGA

The main requirements for the FPGA were to create a seamless user experience by designing a responsive controller with low latency. Figure 3 shows the architecture used to achieve such goals.

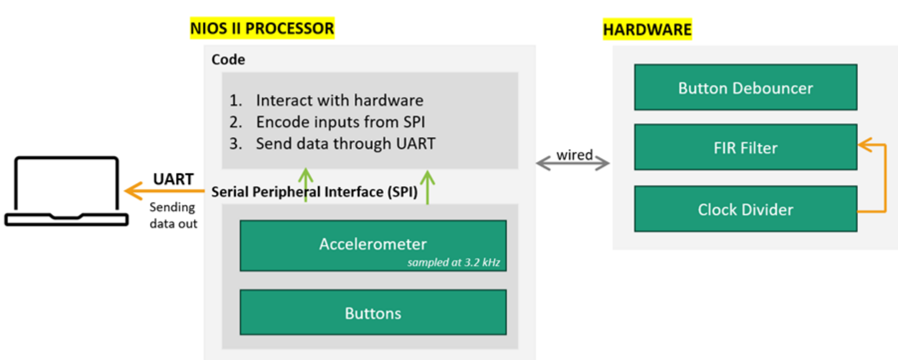


Figure 3: FPGA Architecture

35	25	15	5	4	0
Accelerometer X	Accelerometer Y	Switch	K1	K0	\n

Figure 4: Bit field encoding for UART communication

2.1 Communication between the FPGA and the PC

UART communication was used through the JTAG interface provided. To minimise latency, two major design decisions were made:

1. The bits were to be encoded before sending them over JTAG, as seen in Figure 4.
2. The driver interface was rewritten in C#, to avoid the usage of the NIOS II terminal.

A simple Python game was written to roll around a ball to check the user experience.

By using the NIOS II terminal, our ball was unresponsive and our debounced button signals were missed half of the times. This prompted testing of the latency τ_{UART}, with the results shown below:

Using the NIOS II terminal	Using the rewritten *intel_jtag_uart* library
324 ms	55 ms

We chose C# as the new driver interface language as it is the default on Unity, further simplifying the interfacing of the FPGA and the game. As a result, the translated library significantly reduced controller latency. This also allowed any player with NIOS II drivers installed to connect to the FPGA automatically anytime during the game.

2.2 FPGA hardware

This section revolves around the FPGA virtual machine, an accurate behavioural model that allows code to be tested in software before being flashed to the NIOS II. It also allows data analysis, enabling these design decisions to be objectively measured.

2.2.1 Creating a responsive FIR filter

The goal was to bring down latency to that of wired controls, which is approximately 10ms. FIR filters are discrete time filters, and have an associated group delay of:

$$\frac{N-1}{2T_s} = -\frac{d\phi}{d\omega}$$

To minimise the delay, we can:

1. Reduce the sampling time (T_s).
2. Reduce N (the number of coefficients)

Our testing environment showed that although the data was coming in at 5300 Hz, the accelerometer only updated every 100 Hz. To reduce the sampling time, the datasheet (pg. 13), allows us to raise the sampling rate from 100 Hz to 3.2 kHz. This allowed the use of a 64-tap filter (for a 10ms group delay), but due to hardware limitations, only use 45 taps could be used.

Figure 5 shows an FFT of real accelerometer data, indicating that noise started at 8Hz, therefore the FIR filter in Figure 6 was designed using MATLAB to attenuate all frequencies above 10 Hz, with a stopband frequency of 100 Hz and stopband attenuation of -20 dB.

Figure 7 verifies the behaviour, clearly showing the group delay under 10 ms.

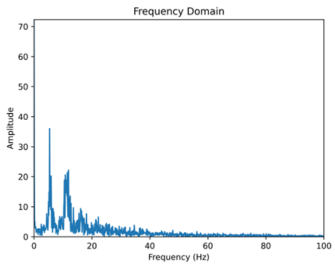


Figure 5: FFT of real accelerometer data

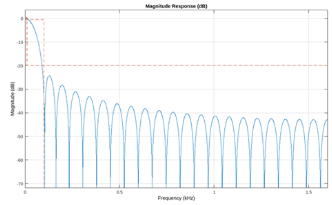


Figure 6: FIR filter

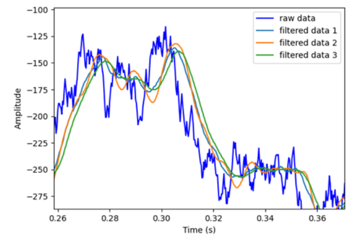


Figure 7: Filter time response

2.2.2 Creating responsive buttons

A button debouncer was implemented in hardware with a time of 10 ms. This was simulated in the Python demo game before being ported to Unity.

2.2.3 Overall latency

The overall latency was measured to be $\tau = \tau_{UART} + \tau_{FIR} + \tau_p = 55 + 10 + 10 = 75$ ms in which τ_p is the processing delay.

3 Game

The game was designed in Unity, as seen in Figure 8, since it provides a wide range of tools for implementing the concept we had envisioned. Classes and functions were all written in C#, the default in Unity.

The gameplay consists of three key parts. A player can:

1. Eat small masses to increase in size
2. Shoot mass from themselves to decrease their size
3. Absorb other players who are smaller.

The speed of the player decreases proportionally to their size, so that smaller players can outmaneuver bigger ones.

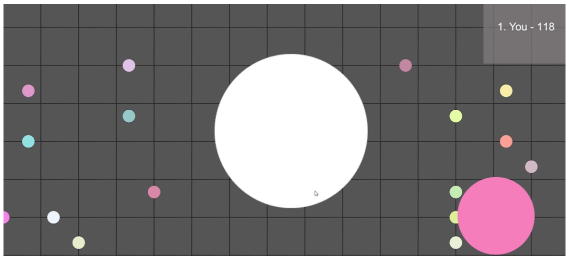


Figure 8: A screenshot of the game

4 Server

The main requirements of the the server were to act as the central authority of the global game state and to provide a smooth experience by minimising latency, keeping the RTT below the human threshold of 200 ms. It must also be scalable, handling 150-250 players before crashing.

4.1 Communication between the clients and the server

4.1.1 Client/Server dichotomy

Figure 9 shows the persistent duplex communication between the server and the different clients.

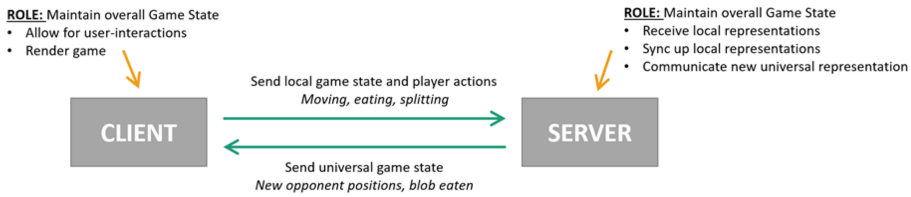


Figure 9: Server/Client roles

4.1.2 Client/Server interaction

The client informs the server of local events. The server then verifies the integrity of the message it received through the universal game state. If no cheating concerns are raised, the server updates the universal game state and broadcasts an update message to all clients, including the client who initiated the trigger.

A player eats a food blob as soon as it collides with it. This interaction was made possible through a close server-client dynamic, with the full details shown in Figure 10.

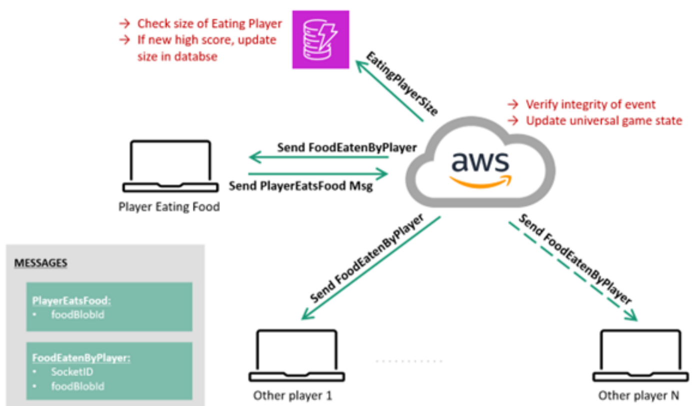


Figure 10: Full interaction of a player eating a mass object

4.1.3 Factoring in latency

Multiple strategies were adopted to keep the latency minimal, as per the specifications .

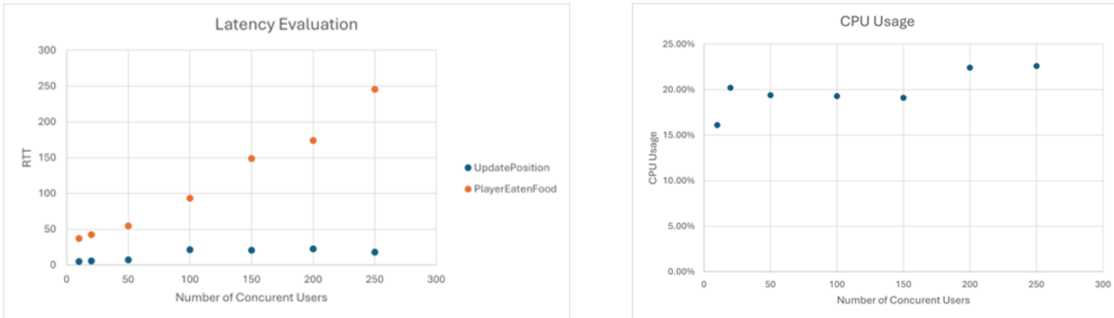
Event detection would take place on the client by sending a trigger to the server, to improve the time complexity of detections from $O(mn)$ for naïve server checks to $O(m)$ for client-side detections. Here, m and n respectively represent the number of food blobs and the number of players.

A second concern was raised when considering the UpdatePosition event. It was chosen to only send UpdatePosition messages to the clients when all blobs had reportedly moved.

4.1.4 Load testing

Server load testing was conducted through the Artillery framework. Virtual users were created to emulate real user interacting with the game. These virtual users would:

1. Send a Join message to the server.
2. Send an UpdatePosition message to the server every 10 ms.
3. Send PlayerEatsFood message every 500 ms.



(a) Latency (b) CPU usage

Figure 11: Server load testing

4. Record the time it takes to receive a response to each message sent.

Testing was conducted on arenas of different size. The results are shown in Figure 11.

4.2 Server infrastructure

4.2.1 Specifications

An AWS EC2 instance was chosen to meet the requirements (see the start of section 4) all the while considering cost. Its specifications are shown in the table below:

Component	Specification	Purpose
Instance Type	t2.micro	Determines available computational resources
vCPU	1	Determines number of simultaneous threads running
Memory	1GB	Used to store temporary game data and player information

4.2.2 Communications and Database

As seen in Figure 2, WebSockets were used for its concurrency handling and duplex communications.

A MongoDB database was maintained to save individual players high scores.

5 Testing flow

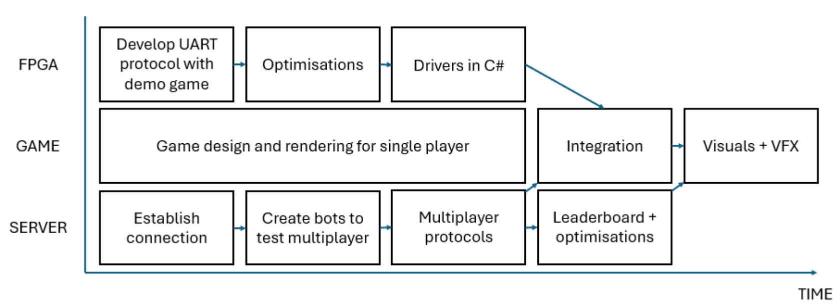


Figure 12: Testing flow diagram