Imperial College London

Information Processing Project Report

Imperial College London

Department of Electrical and Electronic Engineering

FlappyGA: Real-Time Multiplayer Game with FPGA Control

Authors:
Clemen Kok
Johan Jino
Kevin Gnanaraj
Martin Easterbrook
Shermaine Ang
Sohailul Islam Alvi

Module Leader:
Dr. Christos Bouganis

March 2023

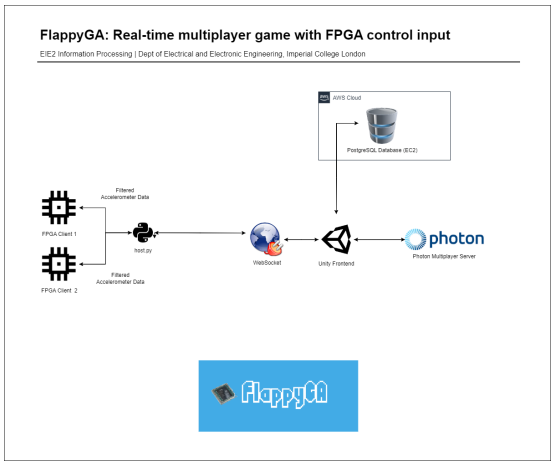
1 FlappyGA Overview

1.1 Purpose of the system

FlappyGA is a real-time multiplayer game where players utilise an FPGA to control a character on the screen, through a series of obstacles without touching them. Players are to tilt the FPGA to make the character on the screen flap its wings and fly higher. The player's objective is to go as far as possible and obtain the maximum possible score, without crashing into obstacles, and they earn points for every obstacle passed. The score is tabulated by counting the number of obstacles passed.

1.2 System Architecture

Figure 1: Architecture



The system architecture of our game (Figure 1) consists of four main components that work together to provide a seamless gaming experience. The first component is the FPGA, which performs the fixed-point FIR filtering on the accelerometer data to smoothen it and remove noise. The filtered data is then sent to a local computer, which determines the board tilt direction. This component allows for local processing of the data and reduces the latency associated with sending data over the network.

The second component of the system is the communication between the local computer and Unity. This communication is done through a websocket, and it allows for real-time updates of the board tilt direction in the game. Unity uses this information to update the game state and provides a responsive gaming experience.

The third component of the system is the Photon Server, which is used to control the game state and allow for multiplayer functionality. The server also manages the game's events and updates the leaderboard data, which is contained in a PostgreSQL Database in an EC2 Instance within the AWS Cloud. This component is crucial for the multiplayer functionality of the game and ensures that the game is fair and competitive for all players.

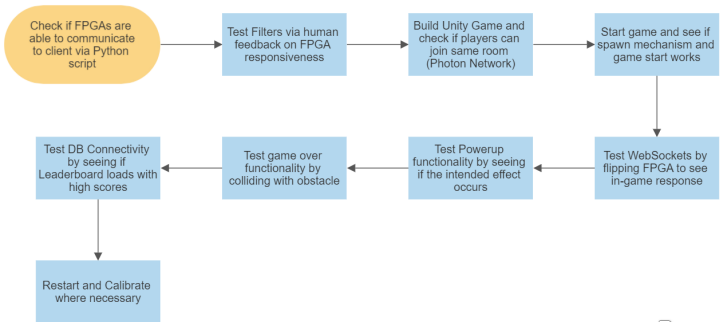
The final component of the system is the communication between the game and the FPGA. The FPGA is updated with the number of obstacles passed, informing the user how long they have lasted in the game. The FPGA also lights up when the player makes a jump. This component is critical for providing the user with feedback and keeping them engaged in the game.

In summary, our game's system architecture is designed to provide a responsive, smooth gaming experience with local processing of accelerometer data, real-time communication between the local computer and Unity, multiplayer functionality with the Photon Server, and user feedback through communication with the FPGA. The system architecture is designed to work seamlessly and ensure that the game is enjoyable for all players.

2 Testing and Performance Analysis

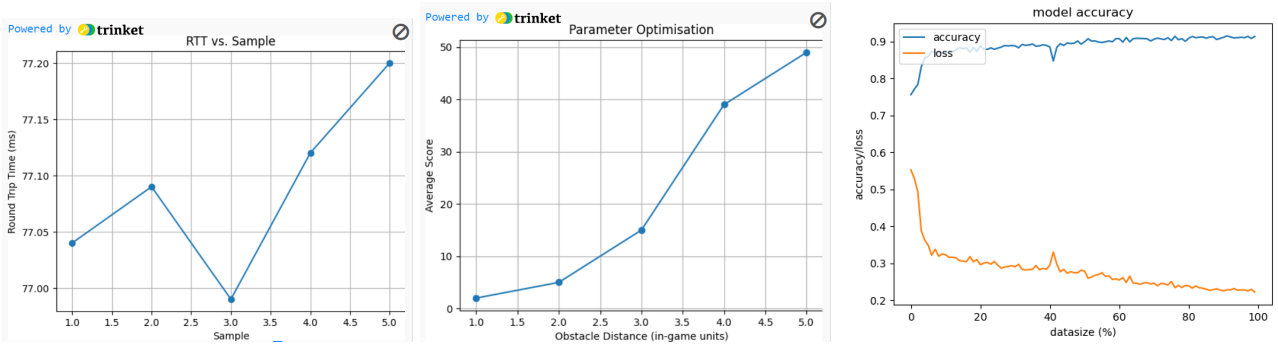
2.1 Testing Flow Chart

Figure 2: Testing Framework



2.2 RTT, User and ML Training Data Analysis

Figure 3: Performance Statistics - RTT, User Feedback and ML Model Result (L-R)



Round Trip Time (RTT): We collated RTT data across multiple games ($N = 5$) to assess the impact of latency on gameplay performance. We did this by pinging the IP address of the Photon Master Server (that our nodes would connect to in order to play in the same room). We found out was that the RTT had an average of $\sim 77.09\ ms$. As this was a real-time multiplayer game, there was a minor delay between an FPGA flip and seeing the character jump on the screen, which could be attributed to latency or internal gameplay mechanics. The RTT between the AWS EC2 Instance hosting our database and our local client was also similar, indicating that the data centers could be located relatively near each other.

User Feedback: We optimised two major parameters according to user feedback: (1) FPGA Filter Coefficients and (2) Obstacle Size. We measured the player's score against the distance between the top and bottom obstacles as well as the FPGA High Pass Filter Coefficient to optimise game parameters so that users do not find the game too difficult to play.

ML Accuracy: We plotted accuracy and loss of the ML model as it was trained based on the dataset size, which corresponds to the accuracy of the model with the amount of user play recorded. We observe that at one point, it shows asymptotic behaviours since the learning hits peak and no new information can be learnt from the given set or game. Further, the sudden dips are due to some incorrect data in the dataset which do not match the logic learnt. The effect of this dies off as the model is being trained more.

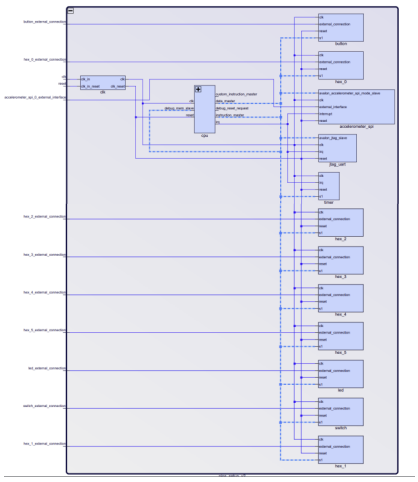
3 Hardware and Communications

3.1 Hardware

The Accelerometer Serial Peripheral Interface (SPI) captures movement in our FPGA. It detects when the user suddenly flips the FPGA in the positive z-axis, to a 13-bit resolution of $\pm 16g$. An SPI allows for short distance communication

between the micro-controller and the rest of the "nodes". From the hardware schematic in Figure 4(a), we can see the connections that we have implemented with the various hex, the JTAG-UART, the button and the accelerometer.

Figure 4: Hardware Schematic and FPGA Flick Capture



(a) Hardware Schematic

```
while (1) {
    alt_up_accelerometer_spi_read_z_axis(acc_dev, & z_read);

    if (z_read > 0xffff0000 && read_data) {
        alt_printf("1");
        led_blink(1);
        read_data = 0;
      }

    if(z_read < 0xffff0000){
        led_blink(0);
        read_data = 1;
        }
```

(b) Code for FPGA Flick Capture

As seen in Figure 4(b), we use the boolean `read_data` alongside the z_axis read (`z_read`) to ensure that out of the continuous stream of position values received, we only output 1 "jump" to the game once the threshold value is reached. The code used can be found here.

3.2 Commmunication Software

We use JTAG-UART for 2-way communication between the node and server. UART is a universal asynchronous receiver and transmitter and JTAG is a standardised unit that is useful for testing devices such as Integrated Circuits. We then use Python and C# in combination, to read off data, sending it to the server and also receiving data from the server. Python is initially used to write files that contain accelerometer readings from the FPGA, and then sends them to the C# file to be read as a communication system to the game server. The game score is then read by the C# server, and pushed back to the Python script. When the flag (`RECV_FLAG`) is set to high, which means that the game is over, the C code allows the game score to be pushed back to the FPGA to be displayed on the hex display, via JTAG-UART (Figure 5).

Figure 5: Transmitting Scoreboard Data

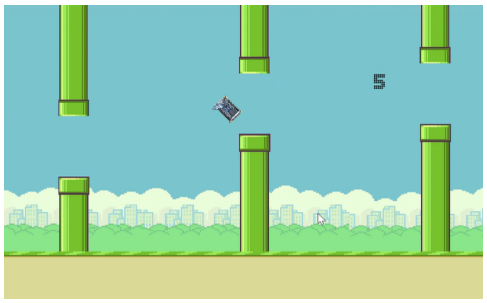
```
if(RECV_FLAG){
    if(RECV_CHAR != old_score){
        if(RECV_CHAR == 'e'){
            exit(1);
        }
        if(RECV_CHAR == '0'){
            tenth_value++;
        }
        update_score( concat_integers(tenth_value, atoi(&RECV_CHAR)) );
        old_score = RECV_CHAR;
    }
```

4 The Game

4.1 Base Mechanics

In the game, the player tilts the FPGA up to trigger a jump to avoid obstacles; collision with an obstacle reduced the number of lives of that player. Each obstacle that the player passes accords to 1 point; the player with a higher score will win the round. Photon is used to meet the two node requirement; it is a cloud-based framework for creating real-time multiplayer games. It uses the Photon Unity Networking application to connect users to a Master Server, where they can join a lobby and select a virtual room to play the game. Once in the room, Photon handles all network communication between players for a smooth gaming experience. The Room creator can start the game and spawn players at random spawnpoints.

Figure 6: Gameplay



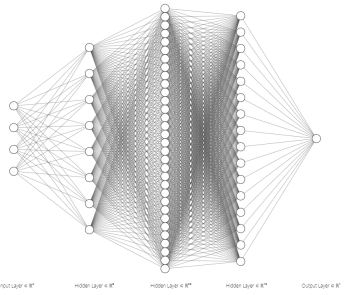
4.2 Machine Learning Bot

As an X-Factor for the project, we built a Machine Learning bot built from scratch in TensorFlow in Python. This was done in a few stages: (1) **Data Collection from User Gameplay**: Data is collected as users play the game and the data is saved into a binary file. This can later be used to process and train the ML bot. (2) **Training the Model**: The model is created on the specification shown in Figure 7(a). The mathematical functions used are based on tested results of performance and ideology. For instance, we use Sigmoid and Binary cross-entropy as the activation function for the output layer, since it returns a value between 0 and 1, which is the probability of the jump. Through multiple epochs, we train the model using an Adam optimiser, which used a Stochastic Gradient Descent with moment, which allows for a good learning rate and prevents the model from overtraining, since it is based on the gradient calculated by the loss function. (3) **ONNX Model Conversion**: The Model in `.h5` is then converted to ONNX and then loaded directly into C#, which drastically improved performance. (4) **Dataset**: Our final design data has an array of 4 values: Y coordinate of player, X and Y coordinates of nearest obstacle in front of the player and the time since the last jump.

Figure 7: Neural Network Architecture in TensorFlow

```python
import tensorflow as tf
import pickle

with open("gamedata.bin", 'rb') as f:
    data = pickle.load(f)
with open("gamedata_ref.bin", 'rb') as f:
    data_ref = pickle.load(f)

print(len(data))            #for debugging and verification
print(data_ref.count(0))    #for debugging and verification
print(data_ref.count(1))    #for debugging and verification

# Define your training data
# Binary output (0 or 1) for each input row

def create_model(data, data_ref):
    # Define the neural network model
    model = tf.keras.Sequential([
        tf.keras.layers.InputLayer(input_shape=(4,)),  # Input layer with 3 neurons
        tf.keras.layers.Dense(units=8, activation='relu'),  # Hidden layer with 8 neurons and ReLU activation
        tf.keras.layers.Dense(units=32, activation='tanh'),  # Hidden layer with 32 neurons and tanh activation
        tf.keras.layers.Dense(units=16, activation='relu'),  # Hidden layer with 16 neurons and ReLU activation
        tf.keras.layers.Dense(units=1, activation='sigmoid')  # Output layer with 1 neuron and sigmoid activation
    ])

    # Compile the model with binary crossentropy loss and Adam optimizer
    model.compile(loss='binary_crossentropy', optimizer='adam')

    # Train the model on some example data

    model.fit(data, data_ref, epochs=100)
    model.save("FlappyGA.h5")
```

(a) TensorFlow structure



(b) Neural Network Architecture

4.3 Powerups

The game currently has two power-ups: one adds a life to the player, allowing them to collide with objects without losing the game, and the other power-up decreases the player's size and inverts gravity. The plan was to let players activate the power-ups with buttons on the FPGA, but this was not implemented due to time constraints. If more time were available, additional power-ups like speed boosts and invincibility would have been added.

4.4 Leaderboard and Cloud Server

At the beginning of the game, players are prompted to enter their names. Their name and score are stored in a PostgreSQL database on the server. The server sends the top 5 scores to the C# client, which displays them on the game screen as a leaderboard. The server was programmed in Python and can accept multiple connections and process data from different clients simultaneously, allowing users to play the game at the same time and receive real-time leaderboard updates. An Amazon Elastic Compute Cloud (EC2) instance was used to run the server, as it provides scalable compute capacity in the cloud and is designed to be highly available and fault-tolerant. TCP and PuTTY were used to communicate with the EC2 instance, and Python was used to run the TCP server as a service using `systemd`. The client was programmed in C# to integrate seamlessly with Unity and provide better performance and faster response times.