The Flying Project

February - March 2024

Adrian, Jungwon, Keegan, Lolézio, Yueming

Purpose

Our project's primary objective is to investigate the potential challenges of cloud gaming services thoroughly. By focusing on this exploration, we aimed to create a multiplayer, real-time flight simulator with enhanced accessibility, enabling users to experience global navigation. This allows the players to fly around the entire world without limitations. In a more technical view, this project aims to integrate Field-Programmable Gate Arrays (FPGAs) with high-performance processors to significantly improve the computational efficiency and flexibility required for our real-time flight simulator.

Overview

This project is a 2-player flight simulator built in Unreal Engine, with FPGAs as controllers. The controller data from both players undergo initial processing locally before being relayed to a centralized server for further refinement. Here, the server is responsible for computing new values of the game parameters needed, using data from the controller. Upon completion of processing, the refined data is broadcasted back to both game clients, where it is rendered in real-time, contributing to a synchronized and immersive gaming environment.

Overall Architecture

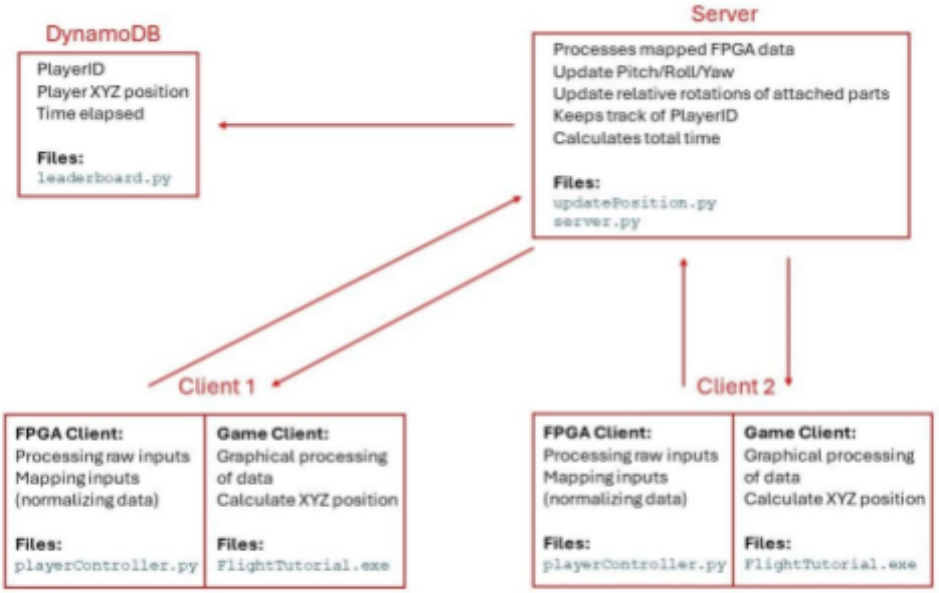

Fig 1: Overall system architecture

The proposed solution to the aforementioned problem is outlined in Figure 1. The fundamentals of the system architecture are that each player is composed of two clients - an FPGA and a game instance that runs in Unreal Engine. Each client is independent and can be run on different machines, with the FPGA dedicated to handling player input and control, while the game engine focuses on graphical processing. This architecture is structured such that the FPGA client does not communicate directly to the game. Instead, information is sent through the server node, thus mimicking real-life cloud-based gaming scenarios and effectively making the FPGA a wireless controller. The server node, therefore,

processes all inputs from both FPGAs concurrently and sends the processed data back to both game clients. The objective of the game is to navigate an aircraft around a map and maneuver through a target. Player timings are tracked via AWS DynamoDB and the winner is decided based on the shortest elapsed time to reach the target destination.

Design Considerations

The flight simulator was initially built to support traditional keyboard and mouse controls. Care had to be taken when porting the controls to the accelerometer to implement the controller for an intuitive and immersive gaming experience. Due to the limitations of the FPGA, the axis mappings of the flight simulator had to be split between the usage of the onboard accelerometer and various buttons and switches. Pitch and roll were controlled through the accelerometer, as it was the most direct and natural means for the player to operate the plane. Acceleration and deceleration were mapped to two separate switches. The reason is that acceleration requires consistent user input, which could become tiresome for the player to push a button continuously. By mapping the acceleration to a switch, the plane speeds up once the switch is flipped, requiring minimal user input. Yawing was mapped to two separate buttons on the FPGA. As the plane's yaw requires more precision than acceleration, it was decided that buttons would be preferable to the switches on the board. This is because buttons only register input when pushed and actively held, whereas switches continue to register input after being flipped and left in position.

Another critical aspect of our design strategy is delivering a smoother flying experience through local processing of the aircraft's position. The system can predict the aircraft's location by directly interpolating between frame ticks based on current parameters without requiring constant inputs. This approach effectively mitigates latency and enhances the smoothness of the plane's motion, ensuring a sense of responsiveness and seamlessness for the player. Through these design considerations, we aim to create a controller that meets the demands of a sophisticated flight simulator and elevates the overall user experience.

In order to give the impression of interacting with the second player, it was crucial that each game client could receive and process state information pertaining to both aircraft and render their movements accordingly. The game renders two distinct planes, giving each player the perspective of one of these planes, hence creating the impression that both planes coexist within the same world. This functionality was achieved through using the asyncio Python library, which supports asynchronous execution of functions. This library was chosen due to its direct integration with

websockets, the protocol for facilitating Unreal Engine web connections. websockets is built on a TCP connection but is optimized for high-speed, real-time connections, thus ideal for minimizing delays.

System Verification and Optimisation

The testing phase comprises three principal stages, illustrated in Figure 3. Stage One entails validating the functionality of individual components. This encompasses verifying the accuracy of the FPGA's data transmission to the local host computer, ensuring the stability of connections between clients and the server, and testing server robustness. Stage Two marks the integration of all components and verification of the functionality of each pathway within the system. Finally, Stage Three focuses on optimisation strategies to minimise delays across all system paths.

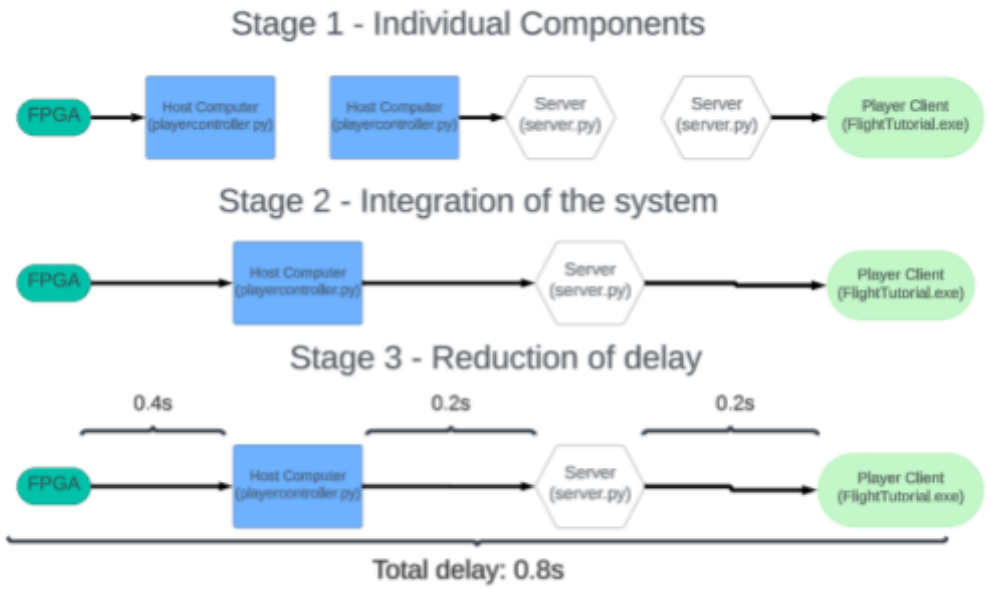

Fig. 3: Testing flowchart

The total delay shown in Fig. 3 was measured with a slow-motion camera at 240fps. Human reaction time is around 0.25s[1], meaning that any delay smaller than this would not be recognisable by the player, and the project's goal of a smooth gaming experience would be achieved. Hence, we aimed to reduce the total delay from the FPGA to the server and back to the game client to 0.2s.

Initially, however, the latency observed was recorded at 40 seconds, a substantial deviation from the goal. This significant delay triggered an automatic disconnection of the client by the server. As illustrated in Fig 4, when configuring the processing speed of each node in the system to a

Fig. 4: Processing speed mismatch causing delay

[1] A comparative study of visual and auditory reaction times on the basis of gender and physical activity levels of medical first year students - PMC (nih.gov)

uniform 30Hz, the FPGA's notably higher operational frequency of 500Hz caused the JTAG-UART buffer to be flooded. This was because the FPGA's rapid data input outpaced the host computer's capacity for extraction. Furthermore, the effective data extraction rate, as depicted in Fig. 4, was found to be lower than the intended 30Hz due to inherent processing delays in the system.

To circumvent this issue, the data output rate of the FPGA was deliberately slowed down using a wait function. Knowing the Nios-II processor's clock speed of 50MHz, a function that wastes cycles was implemented to introduce a calibrated delay between consecutive accelerometer measurements, aligning with the operational frequency of the host computer. Each subsequent node was then sped up to account for the processing delay in a pull-up fashion so that any data added to a buffer is immediately processed and passed on to the next node.

Further optimisation involved measuring the time taken for each data transfer, as shown in Fig. 3, Stage One, using the Python `time` library and large averages of values. Despite increasing the FPGA speed, it was found that the frequency at which the host computer can extract values from the FPGA plateaued at 20Hz, owing to the inherent limitations of the JTAG-UART data transfer protocol which cannot be modified. The frequency of the host computer could then be matched to this constraint at 28Hz, factoring in processing overheads.

The server is an Amazon Web Services t2.micro EC2 instance that can accommodate 4 simultaneous client connections. Thus, without overloading the server, a maximum frequency of 50Hz was determined as the optimal throughput threshold. This significant increase in operating frequency ensures that all processing can be completed and sent to the client immediately. Moreover, the game client operates at an even higher frequency of 60fps (60Hz), with additional logic to render the trajectory of each rendered element between received data packets from the server. This ensures a perceptibly seamless gameplay experience and immediate responsiveness to controller inputs.

These optimizations are crucial in mitigating timeouts due to unresponsive clients or server overload, thereby ensuring gameplay sessions that can be sustained over prolonged durations. The final delay measured using a slow-motion camera was around 0.29s, substantial progress toward achieving the project's gameplay objectives. Nevertheless, the exact performance depends on various factors, including the computational capabilities of the host computers and clients, the traffic intensity within the TCP connection, and the speed of the local network, meaning the observed delay can vary.

The final system is shown in Fig. 5, with the operating frequencies used at each node.

Fig. 5: Final system node delays and frequencies

FPGA Design and Verification

The primary software utilized for designing the system operating on the FPGA comprises Quartus, responsible for configuring the FPGA with the appropriate setup, and Eclipse, which facilitates the generation of the `.elf` file to execute our C program on the FPGA and extract accelerometer data.

The architecture of the Nios-II CPU was specifically designed to integrate essential components such as the accelerometer, buttons, switches, and HEX display units on the FPGA. The objective was to utilise the accelerometer's x- and y-axis readings to emulate pitch and roll motions, with buttons controlling the yaw and two switches regulating thrust. `KEY0` and `KEY1` control left and right yaw, respectively, while the `SW0` switch increments thrust, and the `SW9` switch reduces it. All other switches must remain in their 'OFF' position for the thrust control to function correctly. Furthermore, to enhance the visual aspect of the FPGA as a controller, the HEX displays were configured to display P1 or P2 alternately, depending on the FPGA configuration variant used.

The C program was designed to retrieve values from the aforementioned components, incorporating filtering of accelerometer values to mitigate noise interference and smoothen variations in accelerometer readings. However, filtering is applied conservatively to ensure sufficient data refinement without introducing excessive processing delays on the FPGA. Filtering on the FPGA involves storing accelerometer values in a buffer and computing the average of all stored values. Subsequently, the filtered accelerometer data and button and switch data were packaged into a string before transmitting to the host computer via JTAG-UART communication. Moreover, a wait function was introduced to induce a CPU stall on the FPGA to regulate the data transmission rate from the FPGA to the host computer. This wait function operates by executing nop instructions, effectively creating a stall. Leveraging the known clock frequency of the FPGA at 50MHz, the stall time can be controlled by calculating the appropriate number of cycles.

Upon receiving data from the FPGA, the host computer extracts and maps the values of each component to a range of -1 to 1, aligning with the input specifications expected by the game. To validate the accuracy of data acquired, a prototype Python script was used to capture FPGA values transmitted via JTAG UART and display them on the host computer terminal, ensuring that values adjust accordingly with tilting of the FPGA and activating the buttons and switches. Demonstrating correct data extraction, x- and y-axis readings simulate mouse movements, while buttons and switches mimic keyboard inputs on the host computer using the pynput.keyboard and pynput.mouse Python libraries. Following this validation step, the FPGA is deemed ready for integration into the system.