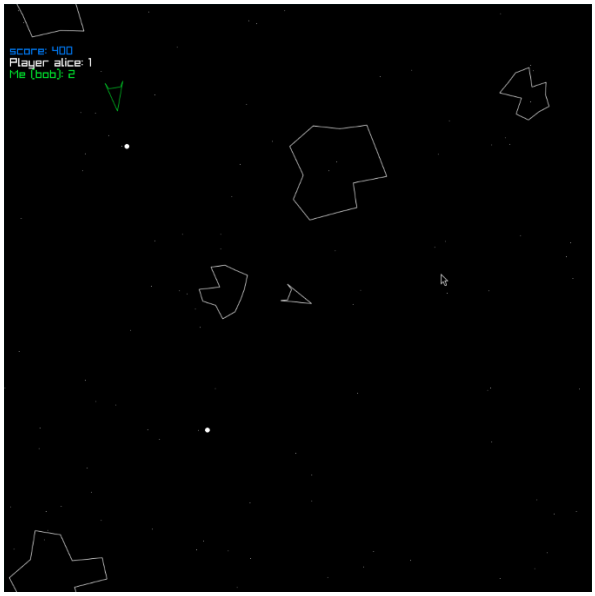
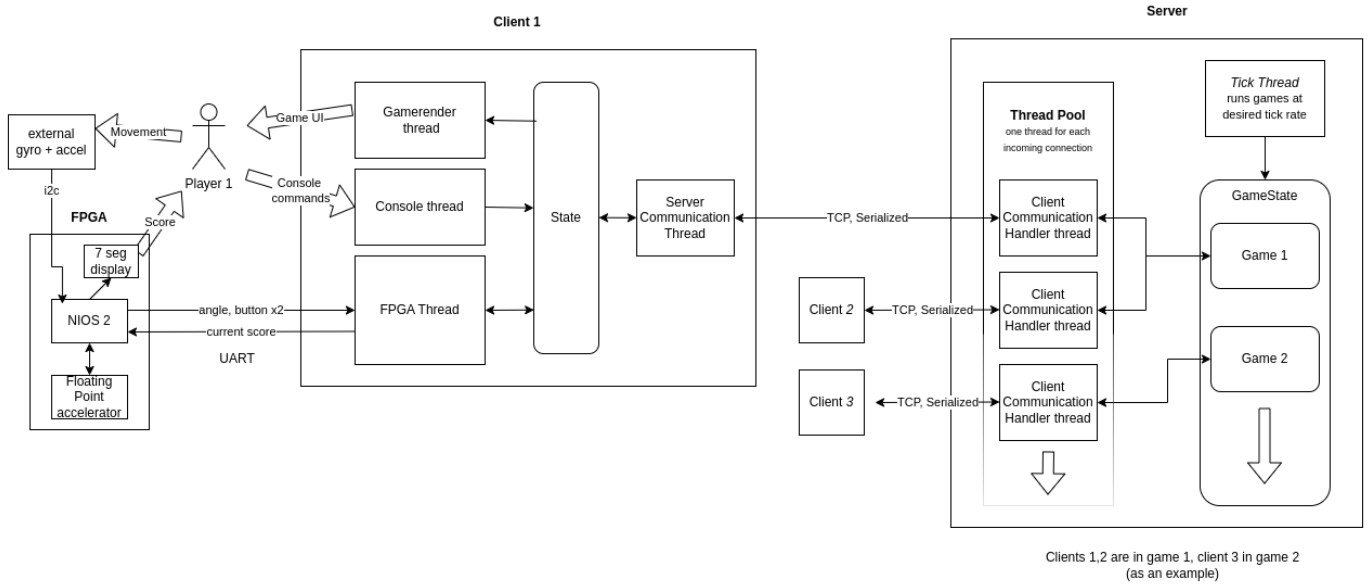
Group 10: Asteroids

Alex Charlton, Shreeharan Chandirassegarane, Quentin Duff, Rahul Ganeshsankar, Kris Pourpongpan

Asteroids is an Atari game from the 1970s, where multiple players fly around a zero-g environment and attempt to shoot down as many asteroids as possible before getting hit too many times and running out of lives. This project is a modern take on take on this, extending the game to a multiplayer environment where players compete to see who can shoot down the most asteroids, adding the twist that player can also shoot each other for a healthy score boost. In the spirit of many popular modern games, the game features a complete lobby system with account and match histories managed by an SQLite database. As a homage to arcade controllers, an FPGA-based controller was developed, featuring accelerometer and button inputs and a display for telling the player their score.



Architecture summary:



Our system is comprised of 3 components. The client is a minimal program running on the user's computer to interface with the cloud hosted server and the FPGA based controller. The server handles most of the system's logic, being responsible for executing the game code, spinning up new game instances and providing a user / lobby system for tracking .

FPGA

The FPGA provides three inputs to the client – attitude angle to control the player's rate of rotation in the game, and two buttons for controlling the thrusters and guns of the spaceship. While handling button presses is simple, we only want to measure the attitude of the FPGA whereas the inbuilt accelerometer will also measure linear acceleration -we therefore need to use a combination of an accelerometer and gyroscope to isolate the attitude. We therefore decided to use an external module that combines both an accelerometer and gyroscope, utilizing custom RTL designs to take advantage of the FPGA's fast processing potential to implement a Kalman filter.

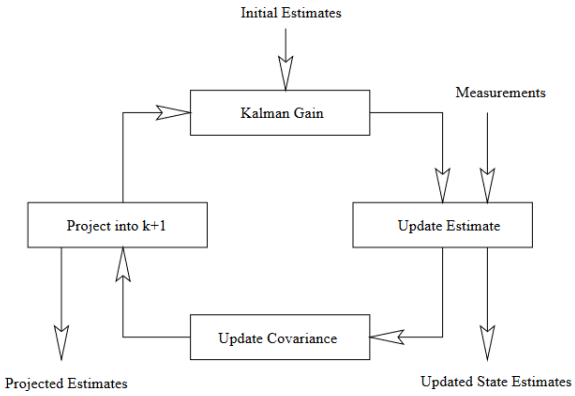


Figure 1

Error! Reference source not found. shows the stages of our filtering design, in which we include an "Update Covariance" stage - this makes the filter better than a standard complementary blend of Acceleration and Gyro readings as the potential error of each sensor is dynamically calculated by refreshing the Covariance error every iteration based on information from the other.

The Quartus project configures the FPGA with multiple modules to speed up processing from what would otherwise be relatively slow running on the NIOS II CPU. To measure the magnitude of performance gained, both hardware coprocessing and pure CPU execution have been compared.

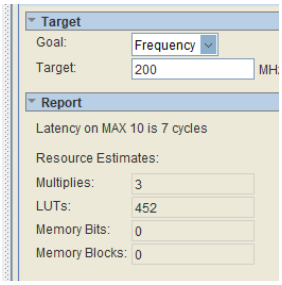


Figure 2

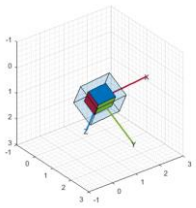
Although the entire filter process would ideally be implemented entirely in hardware, each FP multiplier uses around 500 LUTs (as shown in Figure 2) whereas the FPGA on the DE-10 Lite only contains 5.1k LUTs. Fixed point systems are also not an option as the inevitable impossible balance between low error and clipping cannot be addressed.

The primary parts that required acceleration were the trigonometry calculation and the many floating point multiply accumulate operations. Using an I2C master in HW instead of a software interrupt based one allows me to use the 400KHz mode of the MPU6050 module. The table below shows the speedeup in time.

Calculation	NIOS II (ns)	HW (ns)	Speedup (ms)
Atan2	3259	0-1	3.3
I2C Data fetch	1920	1435	0.5
Covariance Mat Mul	2600	13	2.6
Iteration Preamble	1359	N/A	N/A

The sampling rate has increased from 100Hz to 330Hz.

The Kalman filter can be tuned with the goal of rejecting physical acceleration from the attitude measurement. To do this, the attitude was visualized using MATLAB and the covariance parameters were adjusted by hand until it was accurate enough.



The FPGA interacts with the physical components of the board using the PIO (Parallel IO) module in system designer, and provides the 3 comma-separated values to the client (current angle, and two buttons) over the UART interface.

Client

The client runs on the device of the player and serves as the interface between the FPGA and the server, and also displays the game for the player. The client is comprised of the following python threads:

1. The main game thread renders the game to the personal computer's display. The renderer uses the OpenGL wrapper Raylib for high performance rendering. It takes the positions, rotations and other key values (EG lives) of the game's entities provided by the server and renders them on the screen. The Procedurally generated asteroids and stars, as well as a "flame" appearing behind accelerating players is a unique take on the original Atari game's aesthetic. We also added a debugging mode which displays additional diagnostic information such as the FPS, poll rate, network RTT, and collision hitboxes.
2. FPGA communication thread, to read values from values from the FPGA.

   The FPGA sends a comma-separated string of three values – the current angle (float in range –100 to 100), and two button values (both boolean, sent as either 0 or 1). These values are then stored to the current client state, which will need to be used by the server thread to send to the server.

3. The server interface thread communicates with the server by sending a request to the server 20 times per second. both request and response data are stored in state shared by all client threads. The net code thread simply serialized this data with Capnproto (https://capnproto.org/), and sends it to the server over a TCP socket to the server.
4. A console thread provides a basic CLI for the player to interface with the server's matchmaking system (EG: login, list games, join games). This thread simply reads commands from stdin to the client state and prints out the server's response (from the client state).

Server

The server is responsible for managing players and hosting games. This includes managing user data, spinning up game sessions, and running the underlying game logic. The server is a threaded system written in Rust 🦀 for performance and thread safety.

At the core, is the global game state, which contains data about all game lobbies. This is accessible by all threads behind synchronization primitives for thread safety.

- TCP Server thread

The server listens continuously for new connections, spinning off a new thread for each client, which solely services a single client.

The thread handles communication with the client, whereby the client will send a request which the server thread will deserialize, update the relevant game state, and then respond with the serialized game state.

The request may also contain a list of commands for the lobby interface system. The user account and match history part of this system are built on top of a highly normalized SQLite database providing data permanence between server restarts. Because all clients are disconnected and games cancelled when the server restarts, the lobby system (which handles creating and joining games) is built on purely volatile states spanning the user's TCP thread and the server's global state (a vector of games).

- Game tick thread

The game tick thread periodically iterates through each game, running their tick function. This handles the underlying logic of the game, as well as saving the match results and stopping the game once all players have run out of lives.

Notable features of the game code include the Newtonian spaceship physics designed to closely resembles the original game, the tick-rate-independent based physics (allowing identical behaviors at varying tick rates), and the low complexity distance-checking based approach to collision detection.

Performance metrics and optimizations:

- Latency

  Low latency is important for a pleasant gaming experience

  For our game all processing is done on the server, as we decided this was necessary to correctly determine game state, and this meant we didn't have to perform client-side prediction and server reconciliation. This does, however, add a small delay.

  There is also a delay from reading the data from the FPGA. Since we have a thread constantly reading from the stream and instantly updating the values this means the delay is negligible. However, the client only sends data to the server every 50ms, so there is an expected delay of 25ms, plus the delay to the server over the network for simulation, as well as delay back from the server to the client. If the server is running on AWS, we tested this to have a ping of around 15ms, so there is therefore an average delay of 55ms delay from a player pressing the shoot button to seeing their player shoot. From our testing, this has not affected the playing experience of the game.

- Client Rendering performance

  The client renderer is very performant, as Raylib is only a thin wrapper around OpenGL, and the coordinate mappings are calculated by us. With VSync disabled, we have been able to exceed 3700fps on a desktop with a mid-range GPU, so this is clearly not an issue. Increasing the player count will mean the client must render more players, but this has minimal overhead.

- Server Player handling count

The server must be able to handle many players and simultaneous games efficiently and with low latency.

We also decided to benchmark how many players the server could handle. We wrote a benchmarking program *(benchmarking/benchmark.py)*, that opened 16384 connections to the server, and registered an account for each of them. CPU usage hovers around 9% and uses 168.4MiB memory (with each client continuing to poll the server)

The benchmark script then created 2048 lobbies, where 8 players joined each, and then started the game. The server program then uses around 25% CPU usage, and 232MiB memory. This shows that the game simulation logic (which is set to run at a tick rate of 60Hz has minimal computational requirements).

Increasing the player count further runs into Linux kernel restrictions, as each connection requires an open file (for the socket) and thread, both of which are limited.