Nintendo Etherstream

Vassilios Kaxiras
Physics
Harvard College
Cambridge MA, USA
vkaxiras@college.harvard.edu

Christine Imogu
EECS
Massachusetts Institute of Technology
Cambridge MA, USA
ccimogu@mit.edu

Karen Liberman
Department of Computer Science
Massachussetts Institute of Technology
Cambridge, MA, USA
karenlib@mit.edu

***Abstract*—This project allows for the use of a JoyCon to interact with a game that will be displayed on any computer screen. The game will be a simple 2D game, where players can move in a singular direction at a time and shoot at their opponents. The Joycons for each of the players interacts with a computer through Bluetooth, which sends the information through Ethernet to the FPGA, and the game dynamics are processed within the FPGA. The FPGA will then render the game into 3D graphics. The game graphics will be sent over the Ethernet to the computer, utilizing ARP, UDP and RTP processes, and thus can be displayed by any such device that has access to the internet. We evaluate its performance by analyzing how fast the game and graphics react to changes in the JoyCon as well as the quality of the 3D graphics.**

I. Introduction

Field Programmable Gate Arrays (FPGAs) are known for their quick computation speeds. We take advantage of this property with this project Nintendo Etherstream.

II. Physical Construction

The main components of our system are:

1) Joycon controllers.
2) Bluetooth HID controller and UDP forwarder implemented in C# on a regular laptop. This controller will parse commands from a set of Joycons, sent over Bluetooth, and will package them with minimal processing into UDP packets, sent over Ethernet to the FPGA.
3) Nexys 4 FPGA. We utilize the Ethernet PHY built in to this device to communicate with the Bluetooth controller and playback machines.
4) Playback laptop. This listens to the RTP stream generated by the FPGA and plays it back, using an off-the-shelf H.261 decoder.

III. JoyCon Information Transfer

A. Connecting Joycons

For the FPGA to successfully pair and communicate with the Joycons directly using Bluetooth, a Bluetooth-HID stack is needed. As a workaround, using C#, we search for Joycons to pair with on a computer and initialize a JoyConManager object that keeps track of the states of all connected Joycons, for example, button presses.

B. UDP Forwarder

The inputs from all connected Joycons are merged into a byte string, and the UDP header sub-components – the source port, destination port, length, and checksum are identified and calculated. The UDP packet is sent over Ethernet to the FPGA.

C. Decoder

The bit stream received by the FPGA is decoded by an the ReceiveUDPModule. It utilizes an FSM that first reads the IPv4 header and stores it to a buffer, then does the same to the UDP header, and then receives and parses the payload, matching to the associated command of each player. The data is received one byte at a time, and a counter is used incrementing by 8 to insert the bytes at the correct location. This results in the following FSM states:

1) RECEIVE_IP_HEADER
2) RECEIVE_UDP_HEADER
3) RECEIVE_PAYLOAD

The RECEIVE_PAYLOAD part parses the byte utilizing the following byte format: (0, player movement (3 bits), 0, player shooting (3 bits)). The corresponding player information is identified based on the ordering of the packet receipt. So the first byte in the payload corresponds to Player 1, second to Player 2, etc. This is done by another FSM that leads to the next player as each byte is received, then back to the RECEIVE_IP_HEADER when the final player is reached. ReceiveUDPModule then sends out all the player movements and positions at the same time, utilizing a valid_out.

IV. Game Logic

A. Game Functionality and Rules

The game is played on a 2D plane, and each player has the ability to move up, down, right and left, moving only one direction at a time but unable to move outside of the borders of the screen. To facilitate the interaction with multiple parts, the game was hardcoded to have 3 players. Each player also has the ability to shoot in each of the 4 previously mentioned directions even while it moves (up, down, right, left), but the bullets are unaffected by a players direction and only move in a straight line in the direction they were initially launched. A player only has 1 bullet to shoot with, and their bullet resets allowing them to shoot again whenever their bullet comes in

contact with a wall. If a player's location overlaps with the location of a bullet at any point in time, the game ends and that player loses. The game then immediately resets.

The game logic is taken care of by four modules, 'Shape Party', 'Move Boxes', 'Move Bullets', and 'Bullet Master'. The all the modules and elements within 'Shape Party' are written combinationally, and the clock it receives is used to send into the modules it calls. The other 3 modules are sequential, utilizing a game clock that will control the rate at which things move. The game was made as simple as possible minimizing the number of elements required, so as to facilitate the job of the 3D Encoder.

B. Managing Player Locations

Player locations are managed by a module called 'Move Boxes'. This module will receive the information sent by each JoyCon whose transfer was explained in the previous part. For each of the JoyCons, the information being conveyed will be two numbers - 'Moving Direction' and 'Shooting Direction' - each of size of 3 bits. Five values are valid for either one of these numbers, and they are be mapped to up, down, right, left and neutral. The 'Move Boxes' module will only be utilizing one of these numbers, 'Moving Direction' which communicates the direction the player should move. This module will update the location of each player at every clock cycle by adding or subtracting a locally defined variable MOVE_AMT to the x or y location variable, and pass that location along to 'Shape Party'. 'Move Boxes' also receives in the 2 bit variable 'dead', and if 'dead' is ever different than 0, it resets the locations of the players to their starting spot.

C. Controlling Projectiles

Projectiles are controlled by 'Move Bullets' and 'Bullet Master', and their location is updated every game clock cycle with the one directional movement of a projectile. When a projectile is launched in a direction, as is conveyed by the 'Shooting Direction' value each JoyCon transferred (as explained in the previous subsection), the 'Bullet Master' will receive that information. It then parses that received value with an FSM to then send it to the 'Move Bullets'. There is an FSM for each of the 3 bullets being controlled. There are 5 states in the FSM: NEUTRAL, RIGHT, LEFT, UP, DOWN, and the state corresponds to the current parsed value for the bullet. In the NEUTRAL state, the parsed value simply becomes whatever state corresponding to the value received by 'Shooting Direction', because that means we currently don't have a bullet being shot. In the other 4 states, parsed is stuck at that state until the bullet reaches a location a the edge of the screen in the direction it is flying in, and then parsed returns back to NEUTRAL.

'Bullet Master' compares the location of each bullet to the location of its enemy players at every clock cycle. It there is an overlap between them, it changes the variable 'dead' to the number of that player for 1 clock cycle (then returns it to 0), indicating the player that lost. The 'Bullet Master' module calls the 'Move Bullets' module, passing in the parsed direction for each bullet, as well as all box (player) locations, and it receives back the bullet locations. 'Bullet Master' then sends the bullet locations to 'Shape Party', the module that called it. 'Move Bullets' also has an FMS for each one of the bullets, with the same 5 states NEUTRAL, RIGHT, LEFT, UP, DOWN. At the neutral state, the bullet is set to the location of the center of its corresponding player box. At all other states, its x or y location is incremented by a locally set parameter MOVE_AMT.

D. Game Logic

The 'Shape Party' module is responsible for uniting the game logic together. It calls 'Move Boxes' and 'Bullet Master', passing between them the information each other needs. It passes along to the 3D graphics the location of all the players and bullets, as well as the dead variable, describing the state of each player.

This module can be adapted to function together with a video signal generator and block sprite creators for each player and bullet elements that return the colors of pixels corresponding to the box locations. This was created and utilized to facilitate testing, though not necessary in the actual final design.

V. NETWORKING

We use the Nexys 4 FPGA because of its integrated Microchip LAN8720A Ethernet PHY. An Ethernet Media Access Controller (MAC) for transmitting is implemented (we did not get to the receiving one). The main components of the networking stack are described below:

1) `BitwiseRTPModule`. We send video over RTP; the RTP sender module keeps a running sequence number (initialized to a random number) that it increments on each RTP packet emitted. This also sets the ssrc to a hard-coded value, while the timestamp is passed from the packetizer.
2) `mac_transmit`. This is the MAC transmitter. It receives data one byte at a time. This is implemented with a state machine, that proceeds through each portion of the frame byte by byte. The bytes are then converted to dibits by a separate module named `byte_transmitter`.
3) `crcbzip2`. This calculates the Ethernet checksum using a large table. It is used by the MAC transmitter.

We had an interesting challenge with the data flow for packets. Since we use RTP, there are several length fields that have to be filled out in the packet headers before we can write the packet data. This required knowing the size of the packet before starting the writing process, which we achieved using a packet buffer, of size 12000 bits in BRAM. This buffer was filled with an entire packet before it was sent out.

The RTP, UDP and IPv4 transmitters were made in the same module. All the data this module receives from its previous module is bit by bit, but the module sends out data byte by byte, sending the most significant bytes first, but within each byte, ordered by the least significant bit first. The outermost

header is the IPv4, inside it is contained the UDP, and within the UDP is the RTP and then the payload. The module involves an FSM machine with IDLE, SEND_HEADER, and SEND_PAYLOAD states. In the IDLE state, the module waits to receive a prepare_for_data command, which signifies that the header should start being sent. With that command, the payload_size is stored, the state changes to SEND_HEADER and the IPv4, UDP and RTP headers are created. The UDP and IPv4 headers require the size of their package, which is computer combinationally putting together the size of the headers within it, as well as the payload size received. The rest of the information is hardcoded. The RTP adds in a marker and timestamp the module receives, the rtp_sequence that is calculated, and some hardcoded elements including the SSRC. The rtp_sequence value is initiated to a random number upon reset to for security reasons. Then, that number is incremented with every packet sent, so that the receiver can keep track of packet orders.

In the SEND_HEADER state, the full header that was put together combinationally is sent out 8 bits by 8 bits, using a bit_counter incremented by 8 and a valid_out. When the full header is sent, the state is set to SEND_PAYLOAD, and the ready_for_data variable that is outputed by the module is set to 1, allowing the module that calls it to start sending data. The bit counter continues being incremented in the SEND_PAYLOAD state, until it reaches the size of the full header plus payload size, at which the state returns to IDLE. THE SEND_PAYLOAD state increments a new variable, byte_counter, that is reset whenever it is ¿7 and is utilized to put each one of the bits received into a byte as a buffer, and then sets valid_out to 1 to send that byte and starts again.

Additionally, we implement a Serial Management Interface (SMI) module to read and write to the control registers of the Ethernet PHY. This is used to detect when the Ethernet auto-negotiation procedure is complete, which is displayed on one of the board LEDs.

VI. Graphics Engine

This module takes in the 2D pixel coordinates and colors of the game, and projects it onto a 3D plane. The components required for this transformation include:

A. Creating the Polygon Mesh

The game state is tracked using the positions of the players' and their bullets in 2 dimensions. Creating a mesh of triangles with a specific angle of rotation at any axis and size is possible with the created cubemodel.py file. This generates a .mem file for a BRAM that is used to project game objects' 2D locations to a 3D space.

B. Vertex Shader

Given that the game is originally in 2D space, each polygon will have a default value in one axis. Next, for each 3D polygon in the rendering space, the vertex shader transforms the attributes of its vertices using the camera position which is by default at (0, 0, 0), resulting in 2D-polygons that can be displayed on a screen. It does this by scaling the 3D-dimensional vertices in terms of the specified distance to the screen. The vertex of each pixel is converted and then stored in a BRAM.

C. Pixel Shader

The pixel shader is related to the rendering 2D-window i.e. the game screen, and determines the color of each pixel. It does this by identifying the containing 2D-polygons of each pixel using output from the vertex shader and calculates the distance between each polygon and the camera at that pixel location. The resulting pixel color depends on the overlapping polygon with the highest relative z-index.

Finding the 2D-polygons was done by the intriangle module. The original mathematical approach to determine if a point is inside a triangle involves calculating Barycentric coordinates which involves 3 cross products resulting in an additional latency of around 50-ish clock cycles. Using unconventional math with clever sign checks and the constraints of our system which is bounded by the screen, this computation was reduced to 3 to 5 clock cycles. We also reduced the latency of coloring the screen by reducing the search space of the triangles using its vertices instead of searching the entire screen.

Calculating the z-index of each triangle at a location involves pre-calculating the 3-dimensional normal of the plane containing the triangle using the cross-product sub-module. This allows us to do fewer computations per pixel, and halves the latency as a result. The normal is then used along with the dot-product module to calculate the normal using the equation: $\frac{O \cdot n}{r \cdot n}$, where O is the vector from the camera to a vertex on the triangle, n is the normal, and r is the vector from the frame to the camera.

Ideally, this information would be passed into another buffer, so that the z-index of at each location at the screen could be read before overwriting, but this results in an extra 2 cycle latency for each pixel in the screen. A solution to this problem is yet to be found.

D. Future Work

Due to the limited time to work on this module, it was difficult to efficiently implement matrix operations which are needed for light and shading. After some research into efficient algorithms such as Strassen's and using Gaussian elimination, it appears that for 3×3 matrices, it is best to follow the classical algorithm in mathematics. However, this still results in an overall large latency.

VII. Video Streaming

The video generated by the 3D graphics engine and deposited in the framebuffer is encoded via the H.261 protocol. [1] In order to ensure the frame fits in BRAM, we use the QCIF format (176 x 144 pixels). The encoding protocol is implemented with a state machine, of the following high-level states:

1) PICTURE_HEADER: Output the 4 byte picture header.

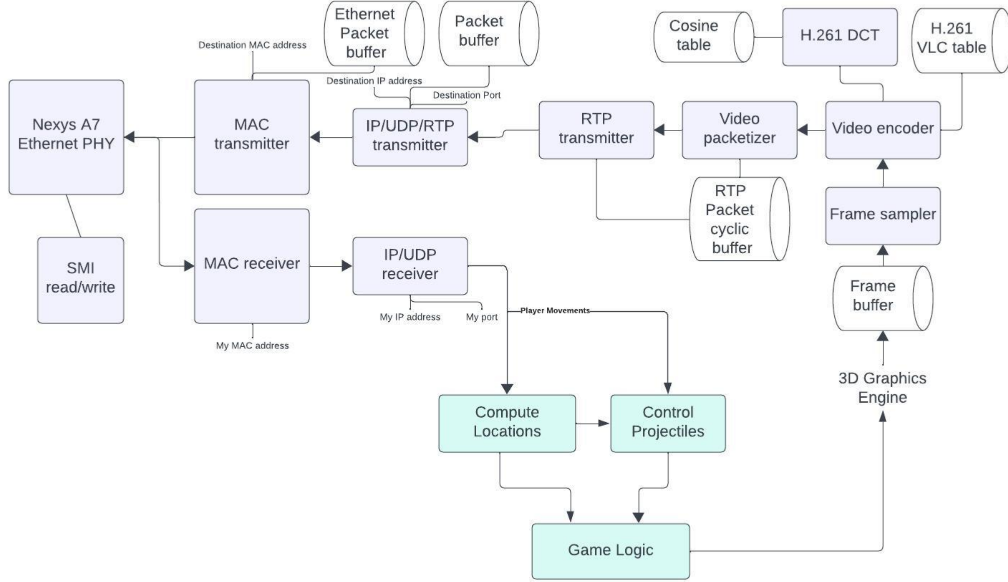

Fig. 1. Block diagram of networking, game logic and video encoding subsystems. Boxes are modules, while the hard drive icons are BRAM.

2) GOB_HEADER: Output a 26-bit Group of Blocks (GOB) header.
3) MACROBLOCK_HEADER: Output the variable-length Macroblock header.
4) BLOCK_DATA: Generate the variable-length coded (VLC) pattern representing all 6 blocks in the current macroblock and append it to the bitstream.

The DCT is implemented with a separate secondary module. The DCT is described by the following equation:

$$F(u,v) = \frac{C(u)C(v)}{4} \times \sum_{x=0}^{7} \sum_{y=0}^{7} f(x,y) \cos[\pi(2x+1)u/16] \cos[\pi(2y+1)v/16] \quad (1)$$

where u, v are the transformed coordinates, x, y are the real-space coordinates, and $C(a) = 1/\sqrt{(}2)$ if $a = 0$ or else 1. The number of possible arguments to the $\cos$ above is 64, so we store the entire set of $64 * 2$ possible values in the summand (the 3 comes from the possible values of $C(u)$) in distributed memory. Then we index into this memory $64*64*2$ times and sum all the terms together, to obtain the final DCT coefficients. All values are stored as double-precision floats (a total of approximately 1 Mb BRAM), and the summation is done using the Xilinx floating point IP for double-precision. [2]. In total, we use an IP block for converting the fixed-point integer inputs to floating point, another block to multiple the two cosine values together, a third to multiply-and-add, and a fourth to round the result to fixed point. Unfortunately, we did not get to integrate this into the main encoder, but tested it separately.

Each transformed value in the output of the DCT is variable-lenth coded per the H.261 spec using a large number of nested switch statements. These variable length codes are then read out separately.

Additionally, the encoder module generates a 90,009 Hz clock by dividing the 100 MHz master clock by 1111. This clock is sampled at the SAMPLE state of the encoder state machine, which generates the timestamp associated with that frame. This timestamp is passed up through the packetizer to the RTP transmitter.

The stream is then packetized using a separate module. This packetizer splits the bitstream along macroblock boundaries. It receives a single bit from the encoder, and buffers the stream, waiting for the end of a macroblock (detected with a singe cycle assert from the encoder). When it reaches the beginning of a new macroblock, it checks if the current packet size is larger than 2048 bits. If so, it then single-cycle asserts the RTP transmitter that the first packet buffer is ready for transmission. While the buffer is being sent, we pause the encoder.

VIII. Design Evaluation

A. *Latency, Throughput and Overheads*

Graphics Engine: The vertex shader takes 3.5 microseconds at a 100MHz clock. This results in a significant delay between when a player moves and a change is made on the screen.

Video encoding

1) The DCT video pipeline needs at little more than 4096 cycles to perform the 64 calculations per 64 entries in a block. We use 6 DCT modules in parallel so we can compute one macroblock in 4096 cycles or so. Since there are 99 macroblocks in a QCIF frame, this gives a total computation time of just over 400,000 cycles,

or about 4 milliseconds. This is well within the 33 millisecond limit for 30 frames per second.

2) When encoding a block, we may have to write up to 20 bits per level, which corresponds to $33 * 6 * 64 * 20 \approx 250,000$ bits, which can be written in about 2.5 ms.

B. Timing Requirements

We avoided large combinational blocks, so we had not issues meeting timing requirements.

C. Use Cases and Deliverables

Our minimal goals were partially reached, as well as some of our ideal goals The game has been fully implemented, and it handles the use cases of actions by 3 players. The system is able to fully receive UDP packets, as well as send RTP packets utilizing MAC. For future versions, some improvements could be made. Our system is not well created to handle the use cases of packet errors, corruption or loss. The checksum available in the IPv4 and UDP packet headers were not utilized (they are optional), and an improved version would utilize those to better ensure data transmission quality. Another use case that could be implemented without too many changes would be to make the number of players variable. To do that, one of the unused bits in the Joycon to UDP to FPGA transmission payload could be used to indicate whether a player is in the game or not, and if statements will be implemented throughout the game logic utilizing those bits corresponding to each player to decide whether to consider the player in the game logic or not.

IX. Contributions

Karen designed the game logic, and the modules that manage player and bullet locations. She created an interaction utilizing elements from a 6.205 class lab to test the performance of the game. Karen created the UDP decoder, as well as the IPv4, UDP, and RTP transmitters, doing research into what their formats and headers are, and how they work together. Karen also helped making the code for the BLOCK_VCL, creating the code matching values to VLC table for TCOEFF, as well as the variable-length encoding (block_transform and TcoeffTable modules).

Christine implemented the Graphics Engine, making the Vertex Shader and Pixel Shader as well as the math formulation and research required. She also researched existing C code that connects to JoyCons to a computer, as well as C code to send that information through a UDP package for the UDP Forwarder.

The code used communicate with the Joycons through SPI and Bluetooth was largely based on JoyconLib, the Joycon Library for Unity, and the UDP server/client was originally written by Louis Erbkamm.

Vassilios created the Ethernet Media Access Controller (MAC) to receive and send packages. He also coded most of the video streaming modules, creating the SAMPLE, P_HEADER, GOB_HEADER, MACROBLOCK and BLOCK_DCT, as well as helping debug the BLOCK_VCL.

References

[1] GCCITT, SGXV. ”Video Codec for Audiovisual Services at p 64 kbit/s Recommendation H. 261.” (1993).

[2] AMD, Xilinx. ”Floating-Point Operator.” https://www.xilinx.com/products/intellectual-property/floating_pt.html.