ECE 385

Fall 2025

Final Project

Parallax Renderer

Ben Pegg

Lab Section: JW @ 2:15pm

TA: Jerry Wu

Introduction

For my ECE 385 final project, I designed a graphics renderer that uses a camera which tracks an IR led to generate a parallax effect and create the illusion of 3D depth on a 2D screen. In my project, I used *The Starry Night* by Vincent van Gogh, which I separated into three layers (background, midground, and foreground) that move independently depending on the position of the IR light within the camera frame. In order to detect the position of the IR light, I used an OV7670 camera module with an IR pass filter, which filters out all of the visible light spectrum, allowing the camera to only detect IR light and nothing else. The position of the IR light is updated in real time, which is used to calculate offsets for each layer to create the parallax effect. Everything is done entirely in hardware including the camera drivers, the centroid detection algorithm, and the graphics rendering. The idea for this project was inspired by Johnny Chung Lee's Wiimote projects, one of which uses the IR camera on a Wii Remote to track IR leds, which he uses to create a pseudo-3D image.

Design

The three overarching stages of the project are the video processing, the centroid algorithm, and the graphics rendering.

Video Processing

For the video processing, the base of the project was built on top of the OV7670-camera repo by amsacks. After modifying the SCCB and register loading FSMs to fit my project and adding the VGA to HDMI IP from Realdigital, I was able to confirm my camera was working properly and get a video feed from the camera. Once I was able to confirm the camera was working, I needed to modify the design to omit the features that I wasn't using and configure the camera settings to best detect my IR light. Because I wasn't using the video signal anywhere, this meant that the quality of my video feed didn't matter, and my main priority was making sure that there was a very distinct brightness difference between the IR light and the background colors so that the centroid algorithm could work as reliably as possible. I did this by decreasing the resolution of the camera to QVGA (320x240), configuring the camera to be black and white, and modifying some of the configuration registers to boost the brightness of the image and disable some of the auto-exposure settings that would adjust the image brightness based on available light. Another change I made was removing the BRAM block as we didn't need it anymore because we weren't displaying the video feed. This was essential, because it provided me with more BRAM blocks that would be needed later to store the graphics. With no more video output, I was able to consolidate and simplify a lot of the camera logic.

Centroid Algorithm

The centroid detection algorithm calculates the live position of the IR light based on a brightness threshold. The module receives the frame data, a 4-bit brightness value, and a threshold value. The brightness value is calculated in the top level by averaging the brightness of each R, G, and B channel and boosting the intensity. Because we only care about brightness and not color, we can just feed the 4-bit brightness value into our centroid algorithm module. In the module, each pixel that has a brightness above the specified threshold value is considered a "bright" pixel. For each bright pixel that we detect, we instantiate a count and add the x and y coordinates to sum_x, and sum_y. Once the frame is finished, the algorithm for finding the centroid coordinates is:

$$x_{centroid} = \frac{sum_x}{count},\ y_{centroid} = \frac{sum_y}{count}.$$

The centroid algorithm module just outputs the count and sum signals, and the division is done in the top level using a divider IP from Xilinx. Because we no longer had video output, to test the algorithm, I used debug cores with the Vivado ILA on our sum_x, sum_y, count, centroid_x, and centroid_y signals to confirm that the centroid was updating correctly.

Graphics Rendering

The graphics renderer updates four graphics layers depending on centroid_x and centroid_y. The parallax effect is generated by moving the three painting layers different amounts relative to the position of the centroid while the front wall layer stays static to create a depth and perspective effect. The background layer will move the least, the middle layer moves a moderate amount, and the foreground layer will move the most. To create the correct perspective, the layers move opposite to the position of the centroid. For example, if the viewer moves their head down and to the left, the layers will move up and to the right. We calculate the x and y delta of the centroid relative to the center of a screen, and then store that distance as a signed number. After that, we can calculate x and y offsets for each layer using the x and y deltas. Division is resource intensive in hardware, so we instead use multiplication and bitshifts to get our x and y offsets. For example, the calculation for the x offset for the front layer is:

*off_x_f = (signed_dist_x * 7) >>> 6; // arithmetic shift to preserve sign*

Each layer of the painting is stored as a COE file which is stored in a single port ROM block, which is generated using the [Image to COE repo] by amsheth. To get the layers to fit in the available BRAM on the Urbana board, the images are 350x250 px for the painting layers, the wall/frame layer is 640x480 px, and the color depth is downsampled to 4-bit. The paintings are slightly larger than the hole in the wall/frame layer (300x200 px), which allows the layers to move within the frame without exposing the edges. To retrieve the correct pixel for each layer, the renderer calculates the read coordinate relative to

the top left corner of the hole in the wall layer. Using the current VGA scan position, the offset, and centering constants, the correct pixel to read is calculated. For example, the calculation for the x read for the front layer is:

$$f_read_x = (drawX - HOLE_X) + CENTER_OFF_X + off_x_f;$$

The x and y coordinates are then flattened into a 1-dimensional address that is fed into the ROM blocks. For example, the calculation for the flattened read address for the front layer is:

$$addr_f = (f_read_y * LAYER_W) + f_read_x;$$

We also have bounds checking logic to ensure that the coordinates are valid to prevent reading garbage data from the ROM blocks. The final pixel that gets outputted is priority based (wall/frame > foreground > middle > background). To add transparency, I designated a specific hot pink (4'hF, 4'h0, 4'hF) to correspond to layer transparency. If the renderer checks the pixel on the layer and sees that it is hot pink, it will check the layer below in the priority order.

Fig 1: Starry Night Background Layer

Fig 2: Starry Night Midground Layer

Fig 3: Starry Night Foreground Layer

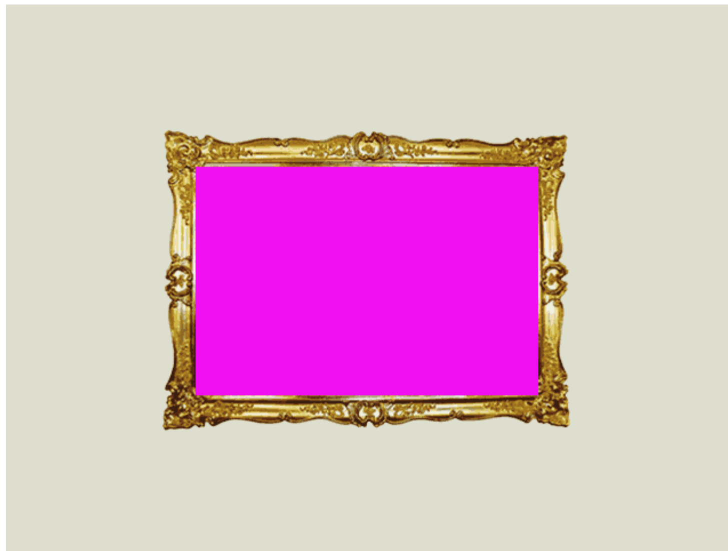

Fig 4: Wall Layer

Top Level Block Design

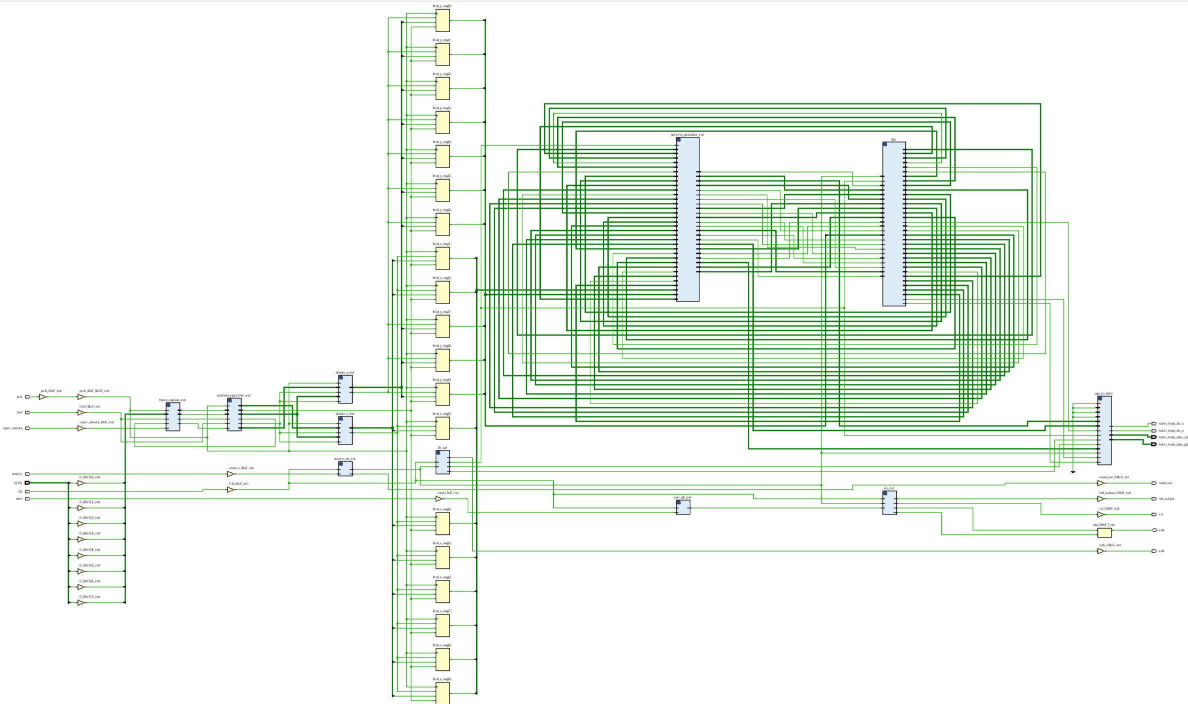

Fig 5: Top Level Block Diagram

External Hardware/Materials

For the camera, I used an OV7670. I chose this camera because it was relatively inexpensive (~$10 for 2 cameras) and well documented. Additionally, the "cheapness" of the camera proved to be beneficial, as more expensive camera modules tend to have an IR cut filter in front of the sensor that removes IR light. To connect the module to the FPGA, we need to add a pull up resistor to the SIO_D (data line, SDA equivalent for SCCB) and SIO_C (clock line, SCL equivalent for SCCB) lines, as the data and clock lines are both open drain. After connecting the camera pins to the PMOD ports of the FPGA, I attached an IR pass filter to the front of the camera. This filters out all visible light, leaving only IR light, allowing the centroid detection algorithm to be significantly more accurate.

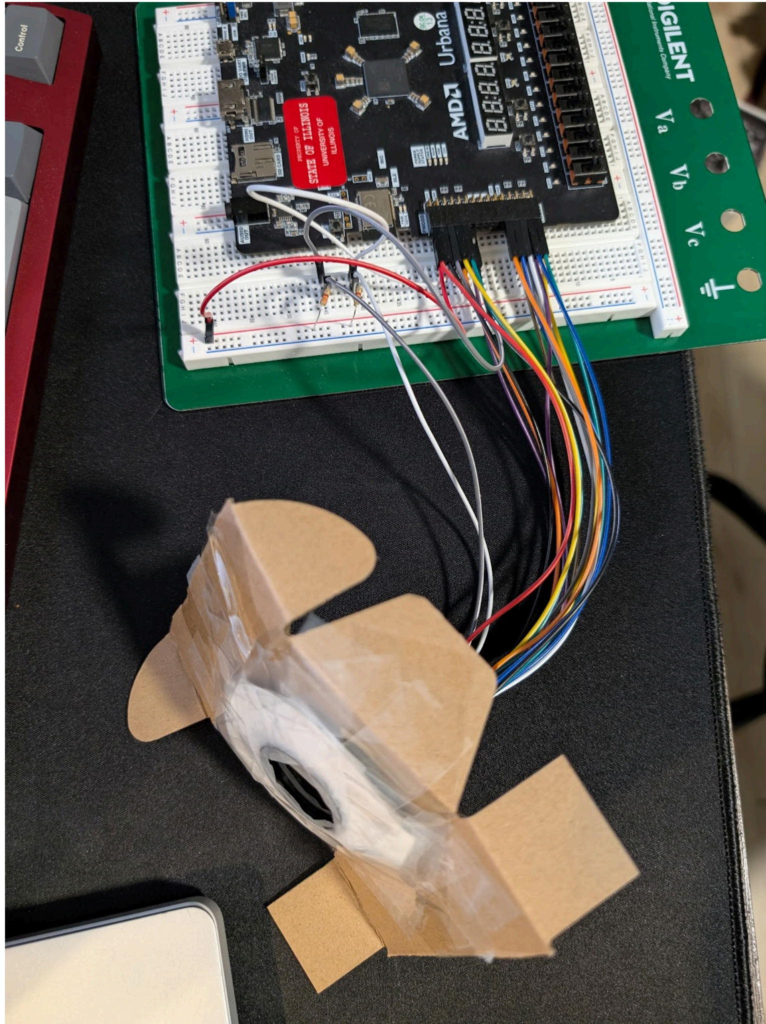

Fig 6: Hardware Setup

Simulation

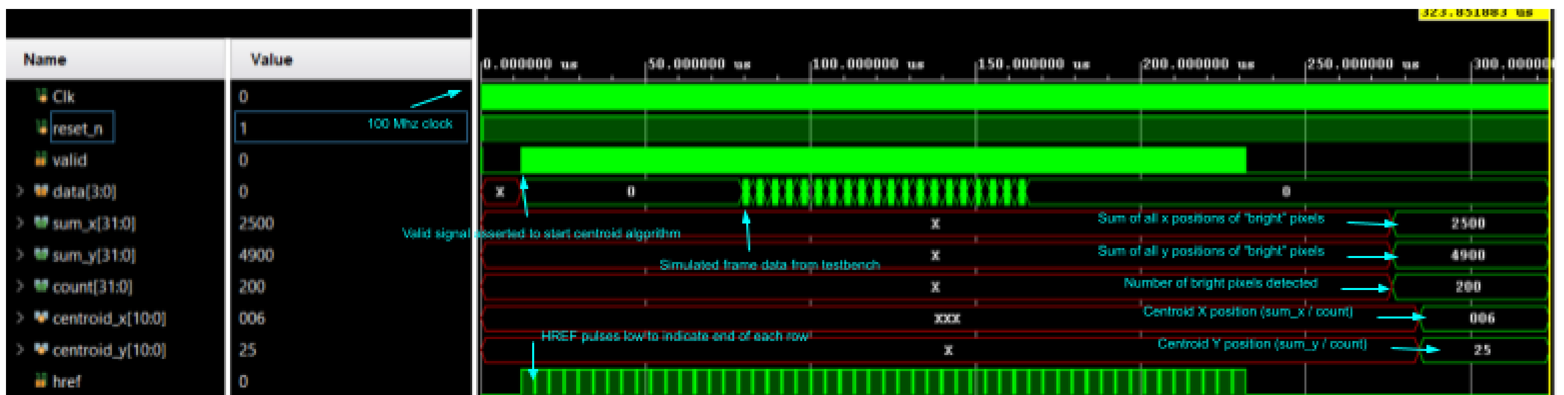

Fig 7: Waveform of simulated frame and centroid detection

SCCB FSM

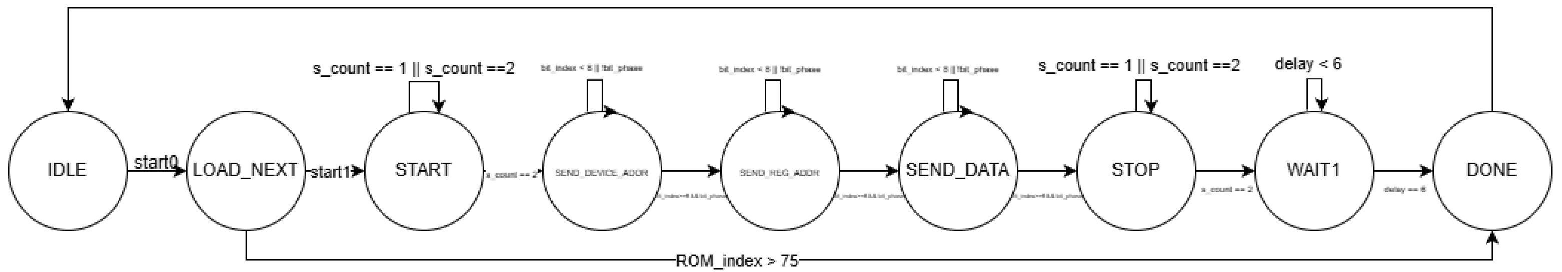

Fig 8: FSM of the SCCB protocol

Design Resources

LUT	2840
DSP	4
Memory (BRAM)	70.50
Flip-Flop	5831
Latches	0
Frequency	127.910 Mhz
Static Power	0.076 W
Dynamic Power	0.344 W
Total Power	0.420 W

My main limitation for board resources ended up being BRAM, as I expected. This is from the image layers, which are stored in ROM blocks. In my proposal, I was originally planning on using a MicroBlaze SoC to do the floating point division for the centroid algorithm. I instead decided to use the Xilinx Divider IP, which ended up being a lot more efficient given the available board resources and simplified the project significantly. Omitting MicroBlaze also allowed me to retain as much color depth and resolution as possible in the final layers, allowing for a better visual experience.

SystemVerilog Modules

Module: camera_top.sv
Inputs: Clk, start, reset_n, [7:0] D, pclk, vsync_camera, href
Inout: sda
Outputs: xclk, scl, reset_out, led_output, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n, [2:0] hdmi_tmds_data_p
Description: Top-level system integrating camera capture, SCCB configuration, live centroid tracking, and an HDMI graphics renderer.
Purpose: Processes real-time video to calculate user head coordinates and render a multi-layered parallax display.

Module: i2c.sv
Inputs: clk, rst_n, start
Inout: sda
Outputs: scl, done, busy
Description: Implements an I2C/SCCB master controller using a FSM
Purpose: Does initialization sequence for the OV7670 camera by handling timing, start/stop conditions, and data transmission.

Module: camera_config_ROM.sv
Inputs: i_clk, i_rstn, [7:0] i_addr
Outputs: [15:0] o_dout
Description: Implements a lookup table containing a predefined sequence of initialization registers for the OV7670 camera sensor.
Purpose: Stores 16-bit configuration words (address and data) used to set the camera to RGB444 mode and calibrate image parameters such as scaling, gamma, and gain.

Module: pixel_capture.sv
Inputs: D, pclk, vsync, href
Outputs: RGB, wr_addr, wr_en
Description: Implements an FSM to capture 8-bit camera data over two clock cycles and reconstruct it into 12-bit RGB pixels.
Purpose: Decodes the sequential byte stream from the OV7670 camera into parallel color data and generates synchronized memory write addresses for centroid detection.

Module: centroid_algorithm.sv
Inputs: pclk, vsync, href, [3:0] data, [3:0] threshold, valid
Outputs: [31:0] sum_x, [31:0] sum_y, [31:0] count, frame_done
Description: Real-time coordinate accumulator for bright pixel detection.
Purpose: Tracks the horizontal and vertical positions of pixels exceeding a brightness threshold and maintains running sums to enable centroid calculation at the end of each video frame.

Module: VGA_controller.sv
Inputs: pixel_clk, reset
Outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY
Description: Implements a standard 640x480 VGA timing controller using horizontal and vertical counters.
Purpose: Generates the necessary sync pulses and scan coordinates to drive a video display and synchronize external graphics logic.

Module: painting_processor.sv
Inputs: clk, [9:0] drawX, [9:0] drawY, [9:0] headX, [9:0] headY, blank
Outputs: [3:0] red, [3:0] green, [3:0] blue
Description: Graphics renderer that calculates parallax offsets and manages address generation for four image layers stored in ROM.
Purpose: Creates a 3D depth effect by shifting background, middle, and foreground layers at varying amounts relative to user head position and performing layer compositing.

Module: wall_palette.sv
Inputs: [3:0] index
Outputs: [3:0] red, [3:0] green, [3:0] blue
Description: Implements a 16-color lookup table .
Purpose: Converts 4-bit indexed color data from memory into 12-bit RGB values for the VGA/HDMI pipeline, including the magenta key used for transparency.

Module: starry_b_palette.sv
Inputs: [3:0] index
Outputs: [3:0] red, [3:0] green, [3:0] blue
Description: Implements a 16-color lookup table .
Purpose: Converts 4-bit indexed color data from memory into 12-bit RGB values for the VGA/HDMI pipeline, including the magenta key used for transparency.

Module: starry_m_palette.sv
Inputs: [3:0] index
Outputs: [3:0] red, [3:0] green, [3:0] blue
Description: Implements a 16-color lookup table .
Purpose: Converts 4-bit indexed color data from memory into 12-bit RGB values for the VGA/HDMI pipeline, including the magenta key used for transparency.

Module: starry_f_palette.sv
Inputs: [3:0] index
Outputs: [3:0] red, [3:0] green, [3:0] blue
Description: Implements a 16-color lookup table .
Purpose: Converts 4-bit indexed color data from memory into 12-bit RGB values for the VGA/HDMI pipeline, including the magenta key used for transparency.

Module: sync_debounce.sv
Inputs: clk, d
Outputs: q
Description: Implements a dual flip-flop synchronizer combined with a counter-based debouncing circuit.
Purpose: Filters mechanical contact bounce from physical pushbuttons and synchronizes asynchronous inputs to the system clock domain to prevent metastability.

Sources Referenced

- Johnny Chung Lee – Wiimote Projects
- Amsacks – OV7670 Camera
- Amsheth – Image to COE
- Cornell University – SCCB Datasheet
- MIT – OV7670 Datasheet
- Vincent van Gogh – The Starry Night
- ECE 385 course materials

Conclusion

I really enjoyed building my final project despite how time consuming it ended up being. Being able to bring the project from an idea to something tangible that I could interact with was extremely gratifying, and I was really proud of my final product after all of the work that I put into it. Although challenging at times, I found ECE 385 to be my favorite ECE class I've taken so far and I'm excited to do more digital design in ECE 411 next semester.