

ColorTrace Duel Final Report

1st Lara Monagel

Department of Electrical and Computer Engineering
Massachusetts Institute of Technology
Cambridge, MA, USA
laram@mit.edu

2nd Lama Alahdal

Department of Electrical and Computer Engineering
Massachusetts Institute of Technology
Cambridge, MA, USA
lamalahdl@mit.edu

Abstract—This report describes the design and implementation of ColorTrace Duel. This FPGA-based two-player path-following game uses an overhead camera and two colored physical objects as controllers. The FPGA receives a video stream, classifies pixels by color, computes the centers of mass of two players, maps these positions onto a logical grid, and checks whether each player is currently on a procedurally generated path. The system renders a background with path and player markers on an HDMI display and supports two visual themes (Grass Path and Snow Path) that vary the graphics. An FSM manages game flow, including start-zone verification, life-loss logic, and game-over states, while a theme-based renderer draws backgrounds, path tiles, and player markers directly to HDMI.

Our design integrates camera, game, and display clock domains while maintaining deterministic behavior and stable 60 Hz video. We evaluate functional correctness, timing closure, and discuss limitations and opportunities for smoother path generation and richer visual effects.

Index Terms—FPGA, image processing, real-time systems, embedded games, digital design.

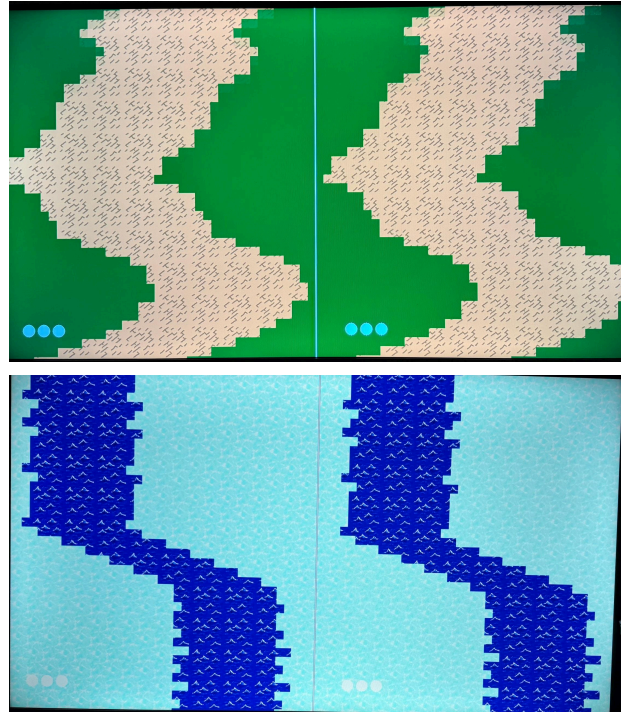


Figure 1. Game screen representation, with two players generating paths that increase in difficulty over time.

I. INTRODUCTION

The goal of ColorTrace Duel is to turn physical motion into a competitive game: two players move colored objects in front of a camera and must stay on an evolving virtual path. Leaving the path costs a life; the last player with remaining lives wins. The project must satisfy three main constraints:

A. Functionality:

- Track two colored objects in real time.
- Maintain a continuous, non-trivial path on a grid that continues downstream over time.
- Implement lives and edge-detection logic when off-path, and a clear winner when the game is over.

B. Display:

- Overlay paths and player objects on a themed background (Grass vs Snow).

- Provide a menu, game-over screens, and a visual life counter on the screen and the seven-segment display.

C. Performance:

- Fit on the FPGA with the target BRAM usage and meet timing constraints.
- Keep latency in the order of one frame so the game feels responsive.

Our final design meets these requirements by not storing full frames from the camera, implementing a grid-based autopath generator, and a pipelined renderer at the HDMI pixel clock.

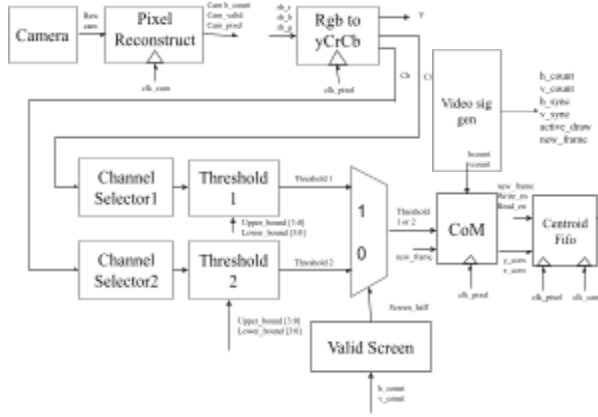


Figure 2. The block diagram of the overall physical setup: camera above the playing area, FPGA board on the side, and the monitor displaying the overlaid result.

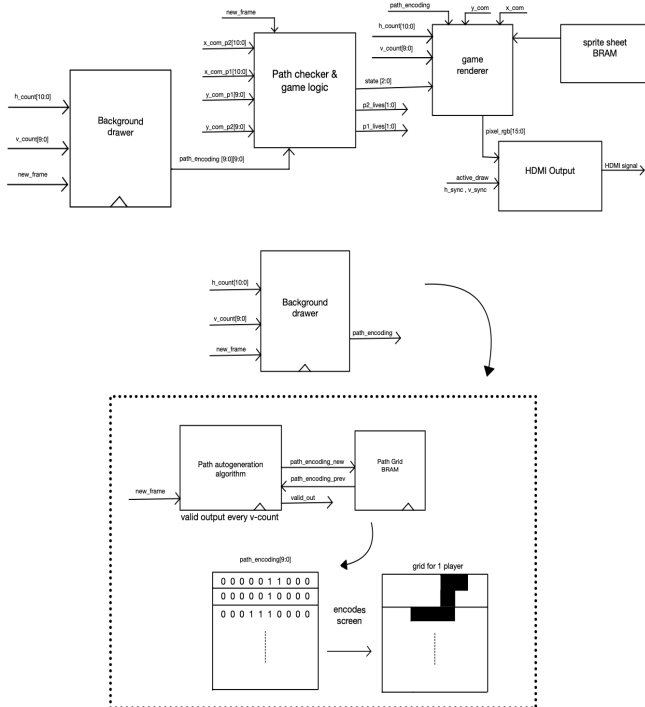


Figure 3. System block diagram showing data flow from path generation and game logic to rendering and HDMI output.

II. DESIGN DETAILS

The project consists of seven primary subsystems (shown in the block diagrams in Figures 2 and 3):

- Camera and centroid extractor
- Autopath Generator
- Path Checker
- Game FSM
- Renderer and HDMI output

Camera pixels pass through an async FIFO, are downsampled to 320×180 , and stored in BRAM. The HDMI domain converts RGB to YCbCr, thresholds Cr/Cb to form two masks, and computes each player's COM. Two autopath_gen modules maintain 40×40 path grids (one per half); a path_checker uses the COM positions and these grids to determine whether each player is on or off their path. The game_fsm uses life-loss events and COM validity to manage the INIT, READY, PLAY, LIFE_LOSS, and GAME_OVER states. The game_renderer uses pixel coordinates, game state, theme, path bits, COMs, and winner signals to generate RGB, which goes into the TMDS encoders and HDMI output. In parallel, a score_ssc module drives the seven-segment display with lives during play and the winner at game over.

A. Camera Interface and Centroid Extractor

The camera interface reconstructs pixels in the clock domain and streams them through an asynchronous FIFO into the HDMI pixel clock domain. There, the pixels are downsampled to 320×180 and written into a dual-port BRAM frame buffer, where they are converted from RGB565 to YCbCr and applied to fixed Cr/Cb thresholds to generate two masks corresponding to the red and yellow player objects. Each mask feeds into a center-of-mass module that uses x, y, and pixel counts across the frame to output a valid COM coordinate at each new frame. A grid mapper divides these COM positions by the cell dimensions to produce the grid indices, clamped to valid bounds and offset as needed, so both players map onto the same grid.

B. Autopath Generator

Two instances of autopath_gen (one per half-screen) maintain 40×40 path grids stored in memory. On reset, they load an initial straight path from a saved initialized file. Once the game starts, the grid scrolls

downward every new frame and synthesizes a new top row by steering an LFSR-driven velocity left or right. The velocity dampens as it approaches the screen edges and favors the middle of the path, with limits on the path's minimum and maximum sizes to keep the game playable. The module gradually reduces the path's half-width after a warm-up period, making the track narrower and more challenging over time.

C. Path Checker

The path_checker receives the COM pixel positions, converts them to coarser grid coordinates, and screens the tiles adjacent to each player. Using the 40×40 bitmasks from the autopath_gen instances, it determines whether each player's circle intersects any off-path tiles via an edge detection mechanism. At the end of the scan, it asserts a one-cycle p1_life_lost or p2_life_lost pulse if that player's object overlaps the background beyond the game path.

D. Game FSM

Events on every new frame call drive the finite state machine.

1. INIT:

The main menu appears, allowing players to toggle switch 0 to choose between grass-mud or snow-ice themes. Button 3 confirms the choice and alerts the FSM to switch to READY.

2. READY:

Waits for the game_start signal, which flags when the game starts after both players are valid and on-path at least once, then transitions to READY.

3. PLAY:

The game is in this state until the path checker outputs a life-lost pulse, indicating one of the players lost a life. The corresponding player's lives are then decremented and moved to LIFE_LOSS state.

4. LIFE_LOSS:

After PAUSE_FRAMES to allow the player to go back on track, the state returns to PLAY if both players still have lives, or moves to GAME_OVER otherwise.

5. GAME_OVER:

Here, the winner is determined by who has the remaining lives. The screen freezes, and the winner is output on the FPGA display. Upon reset, it returns to INIT.

E. Renderer and Output (HDMI and Scoreboard)

The game_renderer operates at the HDMI pixel clock and consumes the current pixel, game state, confirmed theme, path grids, COM positions, and winner code. In the INIT state, it draws two menu sprites ("Grass

Theme" and "Snow Theme") from dedicated BRAMs. During READY, PLAY, and LIFE_LOSS, it selects the appropriate tile set (grass/mud or ice/ocean), renders the background, overlays path pixels using the 40×40 grid for each half-screen, and draws player circles via a radius check around the COMs. The resulting 8-bit RGB stream drives the HDMI output.

A separate score module drives the board's eight-digit seven-segment display using time-multiplexed scanning. A rotating anode selector activates each digit in turn, while a "bto7s" block encodes the 4-bit value to display. During gameplay, the numbers show the remaining lives for both players; in GAME_OVER, they display "P1" or "P2" using a fixed segment pattern for "P" and a numeric digit for the winner.

III. EVALUATION

We evaluate ColorTrace Duel along four axes: latency, pipeline structure, resource usage, and goal completion. All metrics refer to the final implementation described in the previous sections.

A. Latency and Pipeline Depth

1. Clock domains

- Camera: 1280×720 input, sensor clock.
 - HDMI/game: 1280×720 , 60 Hz (pixel clock).
- ##### 2. Pixel-level pipeline (HDMI domain)
- Frame buffer read and rescale: 2 cycles.
 - RGB to YCbCr conversion: 3 pipeline stages.
 - Threshold + mask registration: 2 cycles.
 - COM update, path check, FSM reaction, and renderer selection: 2–3 cycles.

3. Latency

- The camera delivers a new 320×180 frame around every 33 ms.
- HDMI consumes each camera frame (60 Hz).
- COM and life-loss decisions update once per new_frame.

B. Resource Usage (BRAM, Logic, DSP)

We highlight only the dominant memory and compute costs; small control/state machines are negligible in comparison.

1. Frame buffer

320×180 , 16-bit RGB565 = 921,600 bits.

2. Path grids

40×40 bits * 2 paths = 3,200 bits.

3. In-game tile textures

One tile set (e.g., mud or ocean) has an image ($64 \times 64 \times 8$ -bit indices = 32,768 bits) and a palette (256×16 -bit RGB565 = 4,096 bits), therefore a total of 36,864 bits.

4. Menu sprites (theme selection)

Each sprite: $128 \times 128 \times 2$ frames $\times$ 8-bit indices = 262,144 bits.

Palette: 256×24 -bit = 6,144 bits.

This gives one theme 268,288 bits; two themes 536,576 bits.

C. Checklist and Goal Completion

We met all the required items on the project checklist and our initial design goals. For camera and tracking, we implemented two independent color masks driven by camera inputs, per-player COM extraction with validity, and a 40×40 logical grid shared across both halves. For gameplay, we implemented an autopath generator that scrolls and shrinks over time, a path checker that detects off-path events, and an FSM with explicit INIT, READY, PLAY, LIFE_LOSS, and GAME_OVER states, including a winner code. For the user interface, we added two themes (Grass vs Snow), a menu, and a seven-segment scoreboard that displays lives during play and “P1”/“P2” at game over.

IV. LIMITATIONS

The main limitations are in sensing and path representation. Color classification uses fixed YCbCr thresholds, which are efficient but sensitive to lighting and background colors; strong reflections or backgrounds close to the token color can cause missed or noisy detections.

V. INDIVIDUAL CONTRIBUTIONS

We both collaborated on most of the project, including the report, the video, and the program. Specifically, Lama worked on the game logic, the renderer, and the path checker, while Lara focused on autopath generation, player rendering, COM extraction, threshold settings, and test benches. We both debugged cross-module integration and tuned game parameters, iterated on theme visuals, and gameplay.

VI. CODE REPOSITORY

<https://github.mit.edu/6205F25/fa25-6205-team17.git>

VII. ACKNOWLEDGMENT

We thank Professor Joe Steinmeyer and the 6.205 teaching staff for their guidance and support throughout the design and implementation of this project.

VIII. REFERENCES

- [1] C. T. Johnston, K. T. Gribbon, and D. G. Bailey, “FPGA-based remote object tracking for real-time control,” in *Proc. Int. Conf. on Sensing Technology*, Nov. 2005, pp. 66–72.”
- [2] D. Kendri, “FPGA Camera System,” *Hackster.io*, Feb. 2019. [Online]. Available: <https://www.hackster.io/dhq/fpga-camera-system-14d6ea>