

RacerCast Final Report

1st Brianna Lebrun

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

2nd Prince Tufon

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
tufon634@mit.edu

Abstract—We present RacerCast, an FPGA-based pseudo-3D racing engine that renders a real-time driving simulation using a hardware-implemented raycasting and floorcasting pipeline. The system maintains game state through a dedicated logic module that processes user input once per frame and outputs the car’s position and camera angle. A parallel casting engine transforms a 2D grid map into per-pixel wall and ground information, writing compact pixel-type data into an on-chip frame buffer. These pixel types are then mapped to full 24-bit RGB values for HDMI output. The architecture employs BRAM-backed storage for map data, trigonometric lookup tables, and frame buffering, enabling low-latency rendering. By combining fixed-point arithmetic, hardware pipelining, and modular design, RacerCast achieves stable real-time performance while reproducing the visual structure of early raycasted 3D games.

Index Terms—Field-programmable gate arrays, Real-time graphics, Raycasting, Floorcasting, Digital systems, Embedded game engines

I. SYSTEM OVERVIEW

RacerCast is a hardware-accelerated pseudo-3D racing engine that reconstructs a navigable scene from a two-dimensional grid map entirely in FPGA logic. The architecture is organized around a few major subsystems: a *Game Logic* unit responsible for updating the vehicle’s physical state, a *Casting* module that converts that state into a complete video frame using its *Raycasting* and *Floorcasting* submodules, and TMDS/HDMI infrastructure.

The Game Logic module processes new player inputs once per frame, computing the updated vehicle position and direction data used by the other modules downstream. The casting renderer engine then, using a combination of geometry and map data lookups, populates a compact, 3-bit-per-pixel frame buffer. A double-buffered memory interface ensures that rendering and video output can proceed simultaneously without timing interference.

II. MAP REPRESENTATION

The track environment is encoded as a 24×32 grid stored in a 776-entry BRAM structure. Each grid cell occupies two bits and specifies one of four surface types: wall (0), grass (1), track (2), and finish line (3). The final eight entries of the memory image provide initialization metadata: three entries each for the vehicle’s 6-bit row and column coordinates, and two entries encoding the starting angle using a 4-bit representation.

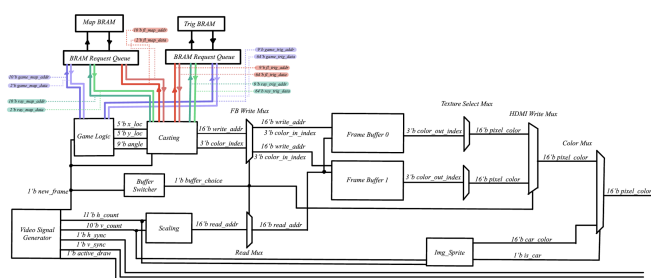


Fig. 1. Block diagram schematic for RacerCast.

This compact representation enables maps to be hand-authored or generated using our Python-based map editor tool. The external tool reduced research and development time significantly, and by exporting directly into the memory format expected by the FPGA, we were able to rapidly test map geometry in various circumstances (see Fig. 2).

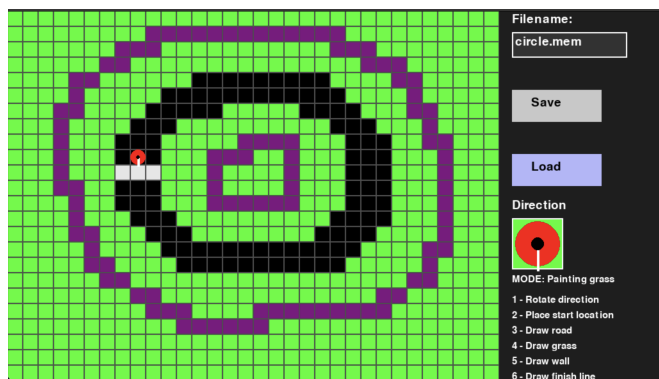


Fig. 2. Pygame-based map editor tool used to simplify map creation.

III. MEMORY AND TABLES

RacerCast relies on BRAM-based memory:

- **Map Memory:** 776 entries, 2 bits each (194 bytes), storing a 24x32 grid plus initialization metadata. Modules Game Logic, Floorcasting, and DDA each had a map BRAM instantiated, for a total of 582 bytes.
- **Frame Buffers:** Two buffers of 57,600 entries at 3 bits per entry (21.6kB each). At any moment, the *active* buffer is written by the Casting module while the *inactive*

buffer is read by the HDMI pipeline. This totals up to 43.2 kB.

- **Trigonometric Lookup Table:** 512 entries, 64 bits each, containing Q8.8 sine, cosine, tangent, and cotangent values (4 kB). Game Logic, Floorcasting, and Raycasting had a trig BRAM instantiated, for a total of 12kB.

The total memory footprint of roughly 55 kB fits comfortably within the on-chip BRAM resources of the target FPGA (~20% utilization), allowing all map lookups, trigonometric evaluations, and frame-buffer operations to be performed without accessing other types of memory, such as external DRAM.

IV. GAME LOGIC

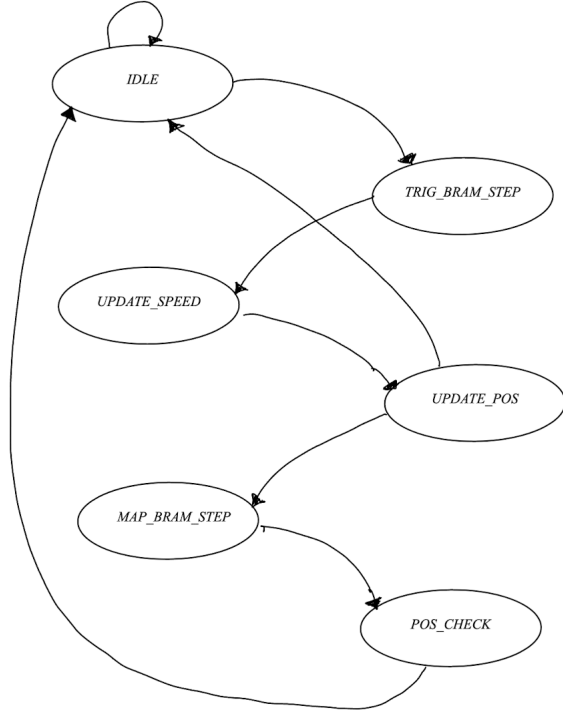


Fig. 3. Game Logic FSM.

The Game Logic module executes once per frame and maintains a small but complete description of the vehicle state: $(x_{loc}, y_{loc}, \theta)$. Inputs from the FPGA buttons determine steering adjustments. Vehicle motion is driven by accelerate and reverse inputs, with the current speed maintained as a signed value in the range $[-MAX_SPEED, MAX_SPEED]$, where negative speeds represent reverse motion. The module uses the map BRAM to enforce collision rules and prevent the vehicle from moving through walls.

Because the Game Logic's output is consumed only once per frame and the updates involve fairly straightforward arithmetic, its latency is largely comprised of BRAM lookup delays and occupies a negligible fraction of the 60 FPS frame window. Because the Game Logic outputs are only consumed at the start of the following frame, the module is also naturally decoupled from the others depending on its outputs.

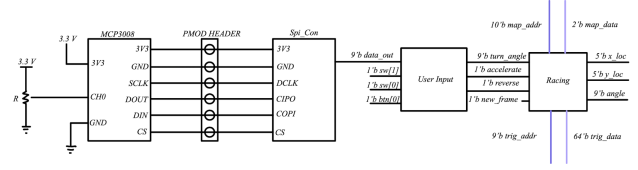


Fig. 4. Block diagram schematic for Game Logic module.

V. CASTING

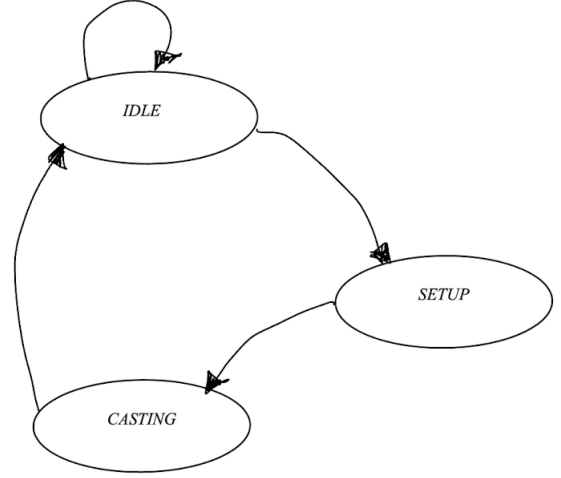


Fig. 5. Casting FSM.

The Casting module transforms the updated vehicle state into a visual display using raycasting for walls and floorcasting for ground and track surfaces. The module follows a deterministic FSM pattern that iterates through all pixels in the frame and issues the appropriate raycasting and floorcasting computations required to populate the frame buffer.

A. Rendering Process Overview

Internally, the module advances independent horizontal and vertical pixel counters, and for each (h_count, v_count) pixel position, it computes the corresponding raycast and floorcast results; the two outputs are then merged such that wall data from the raycaster takes priority over ground data from the floorcaster. The pixel counter is entirely internal to the Casting module and is not tied to HDMI timing, allowing rendering to proceed independently of the display process. The Casting FSM remains in its IDLE state until a new frame starts. Then, it takes a cycle to set the inputs for Floorcasting and Raycasting appropriately in SETUP, then transitions to CASTING, where it waits for valid output signals from the submodule FSMs. Upon completion of the frame, it returns to IDLE.

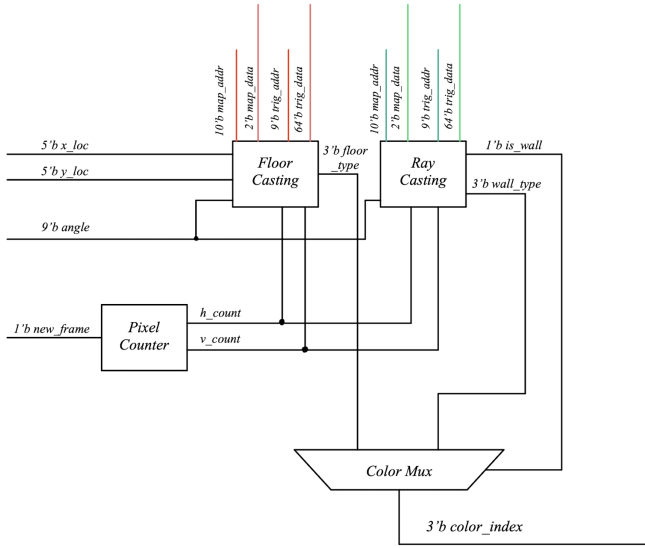


Fig. 6. Block diagram schematic for Casting module.

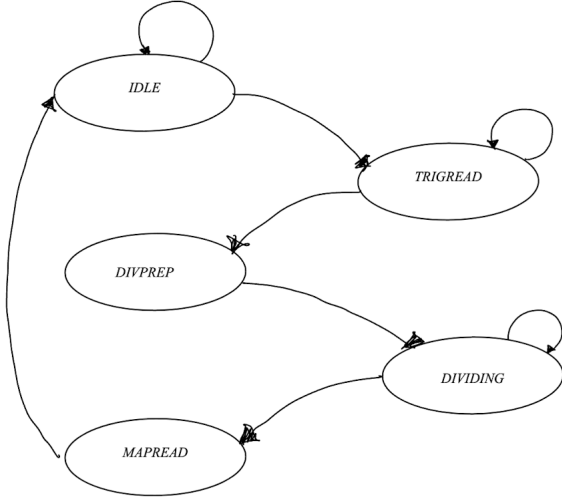


Fig. 7. Floorcasting FSM.

B. Floorcasting

Floorcasting fills the visible ground plane beneath each wall segment. Given the car position and angle data $(x_{loc}, y_{loc}, \theta)$, the algorithm computes the point where a ray would hit the floor. It then deduces the associated grid space of the 2D map:

$$h_{off} = (SCREEN_HEIGHT - 1) - v_count$$

$$d = \frac{SCREEN_HEIGHT/2}{(SCREEN_HEIGHT/2 - h_{off}) \cdot \cos(\phi_0)}$$

$$x = \lfloor x_{loc} + \cos(\phi) \cdot d \rfloor$$

$$y = \lfloor y_{loc} + \sin(\phi) \cdot d \rfloor$$

where ϕ_0 is the angle of the ray relative to the car (based on h_{count}), ϕ is the angle of the ray relative to the map, h_{off} is a

horizon offset, and d is the estimated ground distance from the car to the ray's point of contact with the floor. The resulting coordinates (x, y) are mapped into grid indices and classified as track or grass using the map BRAM. For the upper half of the frame buffer the module outputs the pixel type associated with the sky color, and for the lower half of the frame buffer it outputs the aforementioned calculation.

When triggered, the FSM starts a read from its trig bram, preps and fires off its divider FSM, and then makes a map read based on the output of this divider data.

C. Raycasting

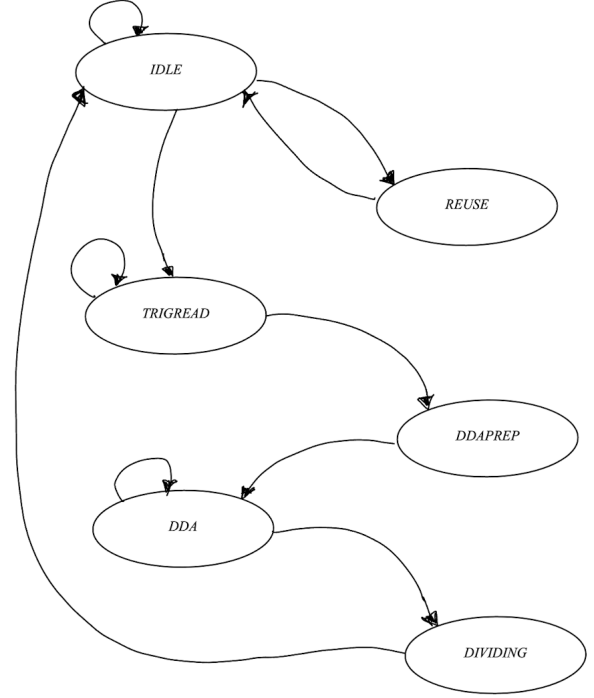


Fig. 8. Raycasting FSM.

For each pixel, the h_count is again mapped to a ray angle relative to the car and the map (ϕ_0 and ϕ respectively). The ray is conceptually traced forward from the vehicle's position until it first intersects a gridline bordering a cell containing a wall. We define the ray length R as the Euclidean distance from the vehicle to the point of contact. The corresponding wall height in pixels is computed as

$$H = \frac{SCREEN_HEIGHT/2}{R \cdot \cos(\phi_0)},$$

where the factor $1/\cos(\phi_0)$ corrects for fisheye distortion. If the vertical pixel counter v_count lies within the range

$$SCREEN_HEIGHT/2 \pm H,$$

the module outputs a signal indicating that a wall is visible at that pixel.

Determining the precise point where a ray first intersects a wall, and whether that intersection occurs along a vertical

However, we were still tightly constrained, as this approach required our DDA module (which has a cycle count linear in the view distance) to cast rays a shorter distance, which ultimately impacted graphics significantly. Our solution was to only cast 1 ray per column of pixels. This did not require adding a significant amount of state, as our implementation simply changed the order in which pixels were written. One column was handled at a time, and the ray was only cast on the first pixel. The data was saved until the module detected

a change in h_count . Our FSM structure allowed for this clean modification, and reusing results decreased the required cycles for Raycasting from 23 per pixel with a maximum view distance of 8 grid cells to less than 3 cycles per pixel on average with a view distance of 24 grid cells (see Fig. 10 for the impact of this improvement).

While this would've given room to scale up resolution to 640x360 while still meeting the timing requirements, Floorcasting became the new bottleneck, as each pixel still required a unique division. As a result, a higher resolution than 320x180 was not achieved by this design. If time permitted, we likely would have taken a higher throughput pipeline approach, which would have increased our resource utilization noticeably. However, we believe the resources were available, as our final design only had just under 8% slice utilization (5.72% slice LUTs, 1.67% slice registers, and 0.49% unique control sets).

Our initial design involved some BRAM sharing logic, which would have added a nontrivial amount of cycles to both Casting modules, which each would have spent at worst 3 cycles on it's own trig computations, and potentially more to take turns. Considering the fact that the frame buffers represented over 90% of the original memory usage, we chose a design easier to implement and test by providing unique BRAMs in each module that needed to either reference the map or calculate trig results. Despite this increase in memory, we managed to achieve 18% BRAM utilization, providing room to expand. This made sense as a long-run decision as well; if the design had successfully been optimized for 1280x720 resolution, interfacing with DRAM over BRAM would have certainly been required, which would have reduced the BRAM tile utilization further.

Aside from resolution scaling, future additions to this work could include a minimap overlay on a portion of the screen. This could be implemented using one of the two ports of a dual-port Game Logic map BRAM, avoiding introducing extra memory into the system. Another addition, which we hoped to implement if time permitted, was a time-trial aspect, where elapsed time would be displayed on the screen and would stop upon crossing the finish line after certain number of laps (a parameter which could perhaps be specified within the map data itself, along with the other metadata).

VIII. SOURCE CODE

All of our code for the RacerCast is located inside of this repository: <https://github.mit.edu/6205F25/fa25-6205-team47>

IX. AUTHOR CONTRIBUTIONS

All members of the team worked together to design the modular structure of our project and implemented the casting module. Brianna implemented the Floorcasting, and Game Logic modules. Prince wrote the Raycasting and DDA modules.

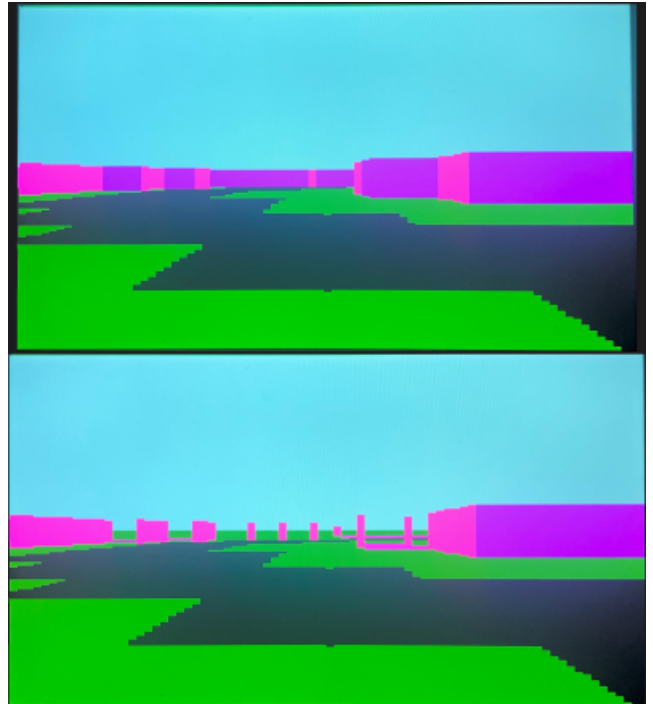


Fig. 10. HDMI display, max view distance of 24 vs 8.

X. ACKNOWLEDGMENTS

We would like to acknowledge the guidance and mentorship of our project advisor and instructor, Joe Steinmeyer. Both he and the 6.205 Fall 2025 course staff were integral to the completion of this project.

We would also like to acknowledge staff member Heba Hussein in particular, as her previous work on raycasting helped guide our research into 3D graphics. In addition, she provided us with some key insights during office hours, one in particular being the double-buffering technique.

REFERENCES

- [1] R. Lode, "Raycasting tutorial," *lodev.org*. Available: <https://lodev.org/cgtutor/raycasting2.html>
- [2] M. Van der Heide, "Pseudo-3D techniques," *Extent of the Jam*. Available: <http://www.extentofthejam.com/pseudo/>
- [3] W. Green, "Fixed point numbers in Verilog," *Project F*. Available: <https://projectf.io/posts/fixed-point-numbers-in-verilog/>
- [4] H. Hussein et. al., "MazeCaster: Pseudo 3D World using Raycasting", *6.205 Final Projects Archive*. Available: <https://fpga.mit.edu/videos/2024/team06/report.pdf>