FPGA Surfers: Implementing a Popular Video Game with 3D Graphics

Alan Lee, *Department of Electrical Engineering and Computer Science, MIT,*
Roger Fan, *Department of Electrical Engineering and Computer Science, MIT*

***Abstract*—We present *FPGA Surfers*, a hardware-accelerated implementation of a simplified endless-runner video game inspired by Subway Surfers, with a primary focus on real-time 3D graphics generation rather than game logic. Our design pipelines obstacle generation, triangle construction, 3D-to-2D projection, and tile-based rasterization entirely on an FPGA, leveraging extensive parallelism to achieve smooth rendering at full 1280×720 HD resolution. A carefully engineered data path enables frame rates up to 40 frames per second, far exceeding our baseline performance target. We evaluate the system's BRAM, DSP, and timing characteristics, and discuss optimizations made.**

***Index Terms*—digital systems, Field programmable gate arrays, 3D graphics, parallel processing**

I. Background Information

Our project aims to create a simplified version of the Subway Surfers mobile game [1]. In the game, a runner ducks, jumps, and switches lanes to avoid obstacles such as barriers and train cars. We focus especially on the 3D graphics behind the continuous forward-moving aspect of the game, trying to render them as smoothly and efficiently as possible. This presents a challenge due to costly projection calculations and the scale of rendering each pixel at a reasonable frame rate. Our project thus focuses on parallelization and other optimizations to meet proper timing with 3D graphics.

As a baseline level of commitment, we aimed to produce a 320x180 resolution game which renders with at least 5 frames per second (FPS), with at least 8-way parallelization for coloring different parts of the frame. In our final iteration, we have fully functional gameplay and complete pixel-specific depth-based rendering for all 3D graphics at 1280x720 resolution through 16-way parallelization.

II. Design Details

To produce each frame in our game, the following sequence must occur. In the following subsections, we outline in detail the techniques used to make this process as efficient as possible.

- **Obstacle Generation (II-A).** Nearby obstacles are generated so that they may be rendered on frame.
- **Game Logic (II-B).** Obstacles are sent to be processed for both game logic and for graphics, and the former helps determine sprite placement and if the game is over.
- **Projection (II-C).** Each obstacle is converted into a series of triangles of different colors and 3D locations, after which the triangle vertices are projected into 2D space. This stage also receives sprite location information from the game logic module to construct the triangles for the sprite, then projects them as well.
- **Triangle Painting (II-D).** The projected triangles are depth-consciously drawn into a single frame stored in DRAM. Upon completion of all tile painting for a single frame, we proceed to rendering the next frame.

A. Obstacle Generation

We have found it helpful for our design to subdivide obstacle placement and location information by *half-blocks*. A half-block is defined as half the length of a train car/ramp, which we make equal to 64 units in our 3D space. In both Subway Surfers and our project, obstacles are always generated in multiples of half-blocks from other obstacles, which thus serves as the primary discretization unit for storing obstacles.

Obstacles are generated using an FSM that is triggered out of the IDLE state upon each new frame. To progress in sync with the game logic described later, the FSM maintains how deep into the current half-block the sprite is. Obstacles are stored in registers, recording 4 bits for each obstacle, 16 half-block-sized obstacles per lane, and 3 lanes for a total of 192 bits. Using the bottom 3 bits, we assign the following encodings to each type of obstacle:

- 000 - No Obstacle
- 001 - Barrier, Solid Low (must jump)
- 010 - Barrier, Solid High (must duck)
- 011 - Barrier, Middle (can either duck or jump)
- 100 - Train Car
- 101 - Ramp

The fourth bit indicates if we are dealing with the front or back half of train cars and ramps. This allows us to determine the `firstrow` and `valid` signals. The `firstrow` signal is only set high if the obstacle is a full-block obstacle (ramp or train car) that is at most 2 half-blocks away, or any other obstacle that is at most 1 half-block away. The valid signal is only set high if we see the back half of a full-block obstacle, or any other obstacle. This ensures no obstacles are sent twice (for both front and back halves) along the data path.

A pseudorandom number generator based on the clock is used to generate new obstacles in the registers. Currently, we assign chances (independently for each lane) of

- $\frac{7}{8}$ to an empty half-block
- $\frac{1}{64}$ to low barriers
- $\frac{1}{64}$ to high barriers
- $\frac{1}{32}$ to middle barriers

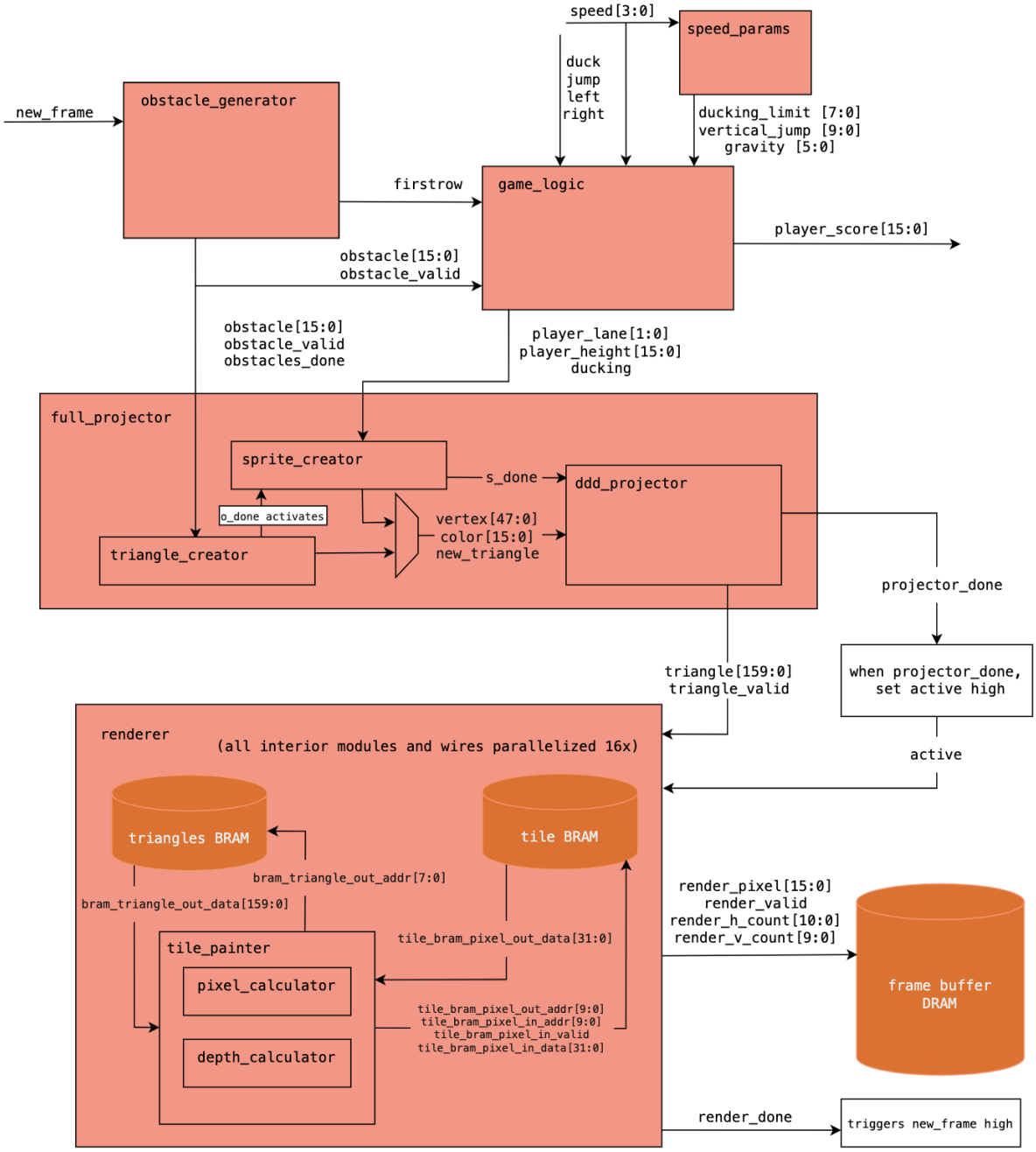

Fig. 1: Block diagram for obstacle generation, game logic, triangle generation and projection, and rendering.

- $\frac{3}{64}$ to train cars, and
- $\frac{1}{64}$ to ramps.

To increase the playability of the game, we make a couple of exceptions to random obstacle generation:

- when a ramp is generated, we hardcode the next 3 full blocks to be train cars.
- the latter half-block (only half-block if the obstacle is a barrier) of each obstacle is at least 2 half-blocks apart from the previous obstacle. This prevents barriers from being too close to each other, or train cars and ramps from intersecting with each other.

On each new frame signal, all obstacles stored in registers are outputted sequentially, separated by 30 clock cycles to allow ample time for computation-heavy modules and storage later in the data path. Obstacles leaving the module take the format of 3 bits for the type of obstacle, 2 bits for the lane, and 11 bits for depth. By setting depth to be the distance of the back of the obstacle from the sprite's depth, we ensure depth will always be positive.

B. Game Logic

1) Inputs: Each of the 4 buttons on the FPGA serve as debounced inputs (`duck`, `jump`, `left`, `right`) for player

movement in the game. Inputs are recorded as high whenever a button goes from being unpressed to being pressed. Additional inputs to this module include

- `speed`, how many z units in 3D space the sprite moves in a single frame,
- `ducking_duration`, how many frames a single duck lasts for,
- `vertical_jump`, how much vertical velocity (in $1/128$ y units per frame) the player gains on a jump, or the negative of the vertical velocity that the player gets set to on a duck while airborne,
- `gravity`, how much (in $1/128$ y units per frame^2) vertical velocity the player loses while airborne.

To make sprite movement accurate within the game, these inputs are already finetuned at the top level module to be compatible with each other, with several switches on the FPGA allowing for `speed` to be adjusted. Our project features four levels, with the following parameters for each level based on the choice of `speed`, as in the table illustrated below.

speed	duck_limit	vertical_jump	gravity
1	128	108	1
2	64	220	4
4	32	420	15
8	16	820	60

As the obstacle generation module sends each 16-bit obstacle in sequence, the `firstrow` signal also dictates which obstacles the game sprite could plausibly intersect with.

2) End of Game Detection: Once a new frame is to be computed, each obstacle is provided sequentially to the module. The game logic only considers those with `firstrow` high, and checks for the sprite's lane, height, and depth within the current half-block to determine intersections. Any intersection with a train car or ramp immediately results in the game ending with `game_over` being set high, while for barriers, we only set `game_over` to be high if the player is failing to duck or jump halfway through the half-block, where the barrier presents itself. The sprite's bottom is determined by player height, while its top is combinationally set as either 32 or 16 y units above its bottom, depending on if the sprite is actively ducking. Lane changes outside of the available lanes also causes `game_over` to be high.

While considering the (possibly empty) obstacle currently occupying the sprite's lane and half-block, the game logic module can also calculate the ground level of that half-block, by processing whether it is a ramp, train car, or a floor-based obstacle.

3) Sprite Advancement: On the new frame signal, the sprite is advanced by `speed` z units, and the necessary adjustments to ducking status, height and vertical velocity are made, according to `ducking_duration`, vertical velocity, and `gravity` respectively.

The player's horizontal movement is dictated by the `left` and `right` signals. By keeping track of whether the player is airborne or not, we can apply gravity specifically in the airborne case, and determine in-air collisions with train cars (due to jumping or switching lanes into them).

4) Outputs: The game logic module currently outputs the player's lane, height and score. These are used to calculate the x, y, and z coordinates of the sprite, respectively. The x and y coordinates are passed onto the sprite creator module for triangle creation, while the z coordinate is provided to a seven segment display module on the FPGA board for the user to see their live score. The ducking status of the sprite is also passed on, to allow for different sprite height rendering.

C. Projection

Once 16-bit obstacles are produced and sent along the data path, the projection stage begins, consisting of three separate modules: triangle creation, sprite creation, and triangle projection.

1) Triangle Creation: The triangle creation module takes in 16-bit obstacle data and then serially outputs the 3D triangles that make up the obstacle. In particular, for each obstacle, the obstacle data stores its depth, type, and lane. Depending on the type of the obstacle, we have hardcoded coordinates based on the obstacle data and 16-bit colors for the triangles that should be outputted. Furthermore, we only output triangles of obstacles with the end of their half-blocks at least 40 z units from the perspective of the player to avoid projecting triangles that will not appear in the frame.

All triangles of a given barrier are at the same depth, while train cars and ramps have depth-varying triangle coordinates to make the structures appear 3-dimensional after projection:

- the low barrier has 2 triangles in red, comprising a rectangle from the ground to half the height of a train car.
- the high barrier has 6 triangles in pink: 2 each for three rectangles. One rectangle spans from half the height of a train car to the full height, the other two are 10 x unit-wide poles on either side of the obstacle.
- the middle barrier has 6 triangles in orange: 2 each for three rectangles. One rectangle is a 16 y unit-tall horizontal bar centered at half the height of a train car, while the other two are 10 x unit-wide poles on either side of the obstacle.
- the train car has 8 triangles in different shades of blue: 2 each for four rectangles comprising the left, right, top and front faces of the car. Depending on the lane that the train car appears in, it may not be necessary to render both, or any, sides of the train car. However, we do not make this lane-specific distinction in our design.
- the ramp has 6 triangles in different shades of yellow: 2 each for three rectangles comprising the left and right sides of the ramp as well as the inclined front face.

The triangles are outputted one vertex at a time, so that the triangle projection module only needs to handle projection of a single vertex per cycle. To organize vertices belonging to a single triangle, when a new triangle begins, the `new_triangle` output wire will be held high.

2) Sprite Creation: Similar to triangle creation, the sprite creation module takes in player lane, height and ducking status in order to serially output the 3D triangles comprising the green sprite. We center the sprite at a fixed depth of 192 z

units from the player's perspective, such that player depth is not a necessary input. Two sets of hardcoded values are stored in this module, depending on if the sprite is ducking. Within this module we also produce a series of black triangles to be displayed as lane lines. Using an FSM, the overall projection module first obtains all obstacle triangles to project, followed by sprite triangles, passing them in sequence to the triangle projection module.

3) Triangle Projection: The triangle projection module functions as a 18-stage pipelined module, with the large number of stages necessitated by division. For each triangle, 3 vertices with (x, y, z) coordinates (thus 48 bits total) are serially supplied to this module with 1 vertex per clock cycle, along with a color. The module then outputs a 160-bit projected triangle:

[color][p1x][p1y][p2x][p2y][p3x][p3y][P][nx][ny][nz]

where (p1x, p1y), (p2x, p2y), (p3x, p3y) are the 2D vertices of the projected triangle, (nx, ny, nz) is the 3D normal vector to the triangle, and P is the "plane constant." That is, nx, ny, nz, and P are computed such that the 3D triangle lies on the plane

$$\text{nx} \cdot x + \text{ny} \cdot y + \text{nz} \cdot z = \text{P}.$$

Note p1x, p1y, p2x, p2y, p3x, and p3y are 16 bit values. The value P is 24 bits, and nx, ny, nz are 8 bits each.

Computation of p1, p2, p3: Projection into 2D is done by projecting to the plane $z = z_0$, where z_0 is a constant value that we set to 256. The 3D point (x, y, z) is then projected to $(xz_0/z, yz_0/z)$. By setting z_0 to be a power of 2, the multiplication is a bitshift and the only difficulty becomes division, which we use a 16-bit pipelined divider to complete.

To avoid signed math complications, the module first extracts the sign of each coordinate before dividing only the absolute value of the coordinate by z (which is always positive), then reapplying the sign. The 3 vertices of a triangle are expected to be fed into the module one cycle after the other, which means after $16+2 = 18$ cycles, all x and y coordinates will be available to output. The resulting x and y coordinates are also offset by half the frame width and height, respectively, to center graphics on the screen.

Computation of n and P: As the 3D vertices (v_1, v_2, v_3) are fed in, the module will store v_1 in registers, along with $a := v_1 - v_2$ and $b := v_1 - v_3$. The vector normal n is then calculated as $b \times a$, which uses 6 multiplications.

Out of necessity, the values nx, ny, and nz are deliberately culled to 8 bits to reduce the size of the LUT multiplier implemented in the depth calculator module. To do this, we simply find the position of the most significant bit and bitshift by an appropriate value.

The value of P is then computed as the dot product between n and the stored value of v_1, which requires 3 multiplications. The resulting values of n and P are then pipelined to wait for the calculation of p1, p2, and p3.

D. Triangle Painting

As the triangles are projected in sequence by the projection module, they are wired to be stored in 16 triangle BRAMs, each storing a full set of triangles for the frame. The final step in the data path is the rendering, which paints the triangles onto a frame buffer.

1) Rendering Approach: We implement rendering by maintaining a z-buffer for each pixel along with its color, which stores the z-position of the closest object to the camera seen by the pixel. Whenever a new triangle is considered, it will only override the color of a pixel if its z-position is closer than the value stored in the z-buffer.

We initially attempted to approximate the z-position of a triangle by precomputing the average distance of its vertices from the camera. This approximation, however, led to various clipping issues. To remedy this, we now perform a full computation of the depth of the intersection between the triangle and the ray projected out from the camera to each pixel. This computation allows for flawless rendering, but is extremely resource intensive.

2) Implementation Approach: Rendering 70 triangles to a full 1280x720 frame buffer requires an enormous number of operations. If, naïvely, one pixel-triangle pair was considered every clock cycle, we would require around 64 million clock cycles, which almost takes a full second for a single frame at 83.333 MHz.

The first, most obvious speed-up is parallelization. Parallelization is limited by our FPGA's resources. As discussed in Section III, the number of DSP multipliers limit our parallelization to 16-way.

There are two ways to parallelize: across pixels or across triangles. If we had parallelized across triangles, each pixel would compute its intersection with 16 triangles in parallel. However, this approach misses out on the second speed-up: triangle locality.

That is, the pixels in a triangle are all grouped together within the triangle's bounding box. The majority of pixels are outside of the bounding box, and it would be a waste of computing power to perform the multiplication-based check for whether the pixel is inside of the triangle. It would therefore be much more efficient to consider the triangles one by one, and then only iterate over the pixels within their bounding boxes. This iteration would then be parallelized 16 ways.

However, this approach still runs into issues with the DRAM frame buffer. If 16 pixels were considered every clock cycle, we would have to write 16 pixels per cycle to the DRAM, which can only handle 1 pixel write per *two* cycles (see Section III). The solution comes by using *BRAMs*. We use 16 parallel BRAMs to represent 80x10 tiles of the board, so that an entire 1280x10 row is considered at once. Because there are 16 BRAMs, we are able to perform 16 pixel writes each clock cycle. Each tile BRAM independently considers each triangle, only iterating through the pixels in the triangle's bounding box. Once all tile BRAMs have been populated with their final values, their values will be written to the DRAM.

3) Renderer: The renderer module takes in the triangles to be rendered and outputs the correct pixel values to the DRAM. It maintains 16 independent tile painter modules, which not only require a tile BRAM each, but also a triangle BRAM each to read from. The tile BRAMs each store 800 pixels,

which each comprise 32 bits, with 16 bits for color and 16 bits for their depth. The triangle BRAMs each store up to 256 triangles, which each are 160 bits from before.

The renderer module is responsible for directing the tile painters to tiles of the screen and for piping the tile BRAM values to the DRAM after all the tile painters are done. The order of operations is roughly as follows:

- The renderer module begins in the IDLE state, writing any inputted triangles to all 16 triangle BRAMs.
- Once the active signal is held high, the module directs the 16 tile painters to the tiles in the current row and activates them.
- Once all 16 tile painters are done, the values of the tile BRAMs are then outputted row by row, adding a fading-in effect to pixels with large depth. (The depth minus a threshold, 796, is appropriately bitshifted and added to the red, green, and blue values individually.) Note pixel values are only outputted to the DRAM every *other* cycle, as the DRAM cannot handle a faster rate of input.
- Then, the tile painters are set to the WIPE state, so that each tile BRAM is iteratively wiped blank to the background white color.
- This process repeats until all 72 of the height-10 rows have been painted.

4) Tile Painter: The individual tile painters operate independently of each other, interfacing with their own triangle and tile BRAMs. Each tile painter operates as follows:

- When activated, the module requests a triangle from the triangle BRAM.
- After 2 cycles, the requested triangle is stored to registers. The module then calculates the intersection of the triangle's bounding box with the tile, and begins iteratively requesting the tile BRAM's pixel values for each pixel within this intersection.
- For each pixel value returned by the tile BRAM, a single-cycle pixel calculation is done to determine if the pixel lies within the triangle. This happens iff the pixel is inside the triangle (computed with a coordinate cross product). At the same time, a 27-cycle depth calculation is performed to determine the depth at which the pixel intersects the triangle. After this computation, if the calculated depth is less than the previous stored depth and the pixel lies within the triangle, the triangle's color and depth are outputted to overwrite the previous pixel value in the tile BRAM.
- After all pixels have been considered, the next triangle is requested, restarting the process.
- After all triangles have been painted, the module outputs a high done signal. If, then, the `wipe` input is held high, the module is then set to the WIPE state, and iteratively overwrites the values in its tile BRAM to the default white.

5) Depth Calculation: The intersection of a plane $n_x \cdot x + n_y \cdot y + n_z \cdot z = \mathrm{P}$ and the vector v has z-coordinate

$$v_z \cdot \frac{\mathrm{P}}{n \cdot v}.$$

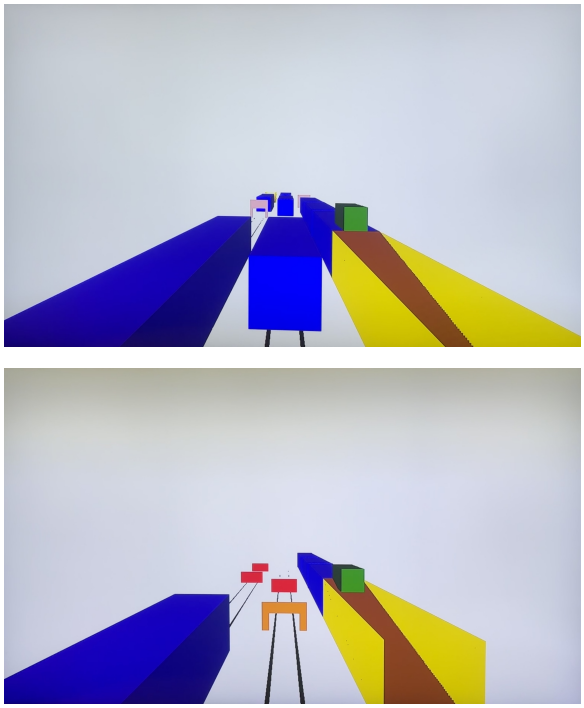

Fig. 2: Example frames in FPGA Surfers.

To see this, note the planes

$$\mathrm{nx} \cdot x + \mathrm{ny} \cdot y + \mathrm{nz} \cdot z = 0,$$

$$\mathrm{nx} \cdot x + \mathrm{ny} \cdot y + \mathrm{nz} \cdot z = n \cdot v,$$

$$\mathrm{nx} \cdot x + \mathrm{ny} \cdot y + \mathrm{nz} \cdot z = \mathrm{P}$$

are parallel and contain the origin, point (v_x, v_y, v_z), and point (x, y, z), respectively. Thus, the ratio of the z-coordinates between (v_x, v_y, v_z) and (x, y, z) must be

$$\frac{\mathrm{P} - 0}{n \cdot v - 0} = \frac{\mathrm{P}}{n \cdot v}.$$

Multiplying by v_z results in the desired z-coordinate.

For our purposes, v is a pixel on the projection plane with coordinates $(x, y, 256)$. Note P and n are precomputed for each triangle, and that the dot product $n \cdot v$ can be done with 2 multiplications, as $n_z \cdot v_z$ is a bitshift. Note $v_z \cdot \mathrm{P}$ is also a bitshift, so that we only require a 16-cycle pipelined divider for computing the final depth $\frac{v_z \cdot \mathrm{P}}{n \cdot v}$.

One of the 2 multiplications here ($n_y \cdot v_y$) is done by a 9-cycle pipelined LUT-based multiplier due to limited DSP resources. The total depth calculation is then pipelined with 27 cycles.

III. Evaluation and Insights

A. Timing and Resource Utilization

We comfortably meet timing under a 83.333 MHz clock with a WNS of 1.105 ns achieved via heavy pipelining across computation-heavy modules such as triangle projection and depth calculation. Due to our heavy pipelining throughout multiplication and division stages, we use 23354, or roughly $\frac{1}{3}$ of the available flip-flops on the FPGA. We also use almost

all (30586) of the lookup tables on the FPGA due to frequent pipelined divisions throughout the data path.

Our current design uses

- 48 18-kilobit BRAMs, three for each triangle BRAM,
- 16 36-kilobit BRAMs, one for each tile BRAM, and
- 4 additional 36-kilobit BRAMs for the DRAM interface.

We also use *all* 120 DSPs:

- 9 for the computation of nx, ny, nz and P during triangle projection,
- 96 across the 16 parallel paths which use 6 DSPs each for the cross product in pixel calculation, and
- 15 more for a portion of the multiplication in the parallel depth calculation modules.

To finish calculating $n \cdot v$ in the depth calculation module, we use a pipelined multiplier. The majority of the DSPs are used for pixel calculation, but 6 multiplications are required for calculating the cross product, effectively requiring all 96 DSPs if parallelization is not sacrificed.

B. Checklist

Our current design has anywhere between 30 and 150 triangles in frame at a time and empirically runs at approximately 40 FPS at a resolution of 1280x720. Furthermore, our rendering pipeline enjoys 16-way parallelization, with optimizations such as bounding box-only iteration. All game logic is properly integrated with the 3D graphics, and the game sprite is fully responsive to player controls. This meets all commitment and goal targets, as well as the stretch goal of 15 FPS at 1280x720 resolution. Furthermore, the fading-in effect of far obstacles is our first step toward the stretch goal of shading.

Despite choosing not to implement moving train cars, we believe this to be a relatively easy addition to game logic and triangle generation, using player score to calculate the car movement from its original depth.

C. Implementation Insights

Once we had a working pipeline, we focused on optimizations (see Section II-D2) such as parallelization, which alone led to 400 FPS at 320x180 resolution. With the bounding box-only iteration by tile painters, we were able to achieve 1200 FPS at 320x180 resolution, which would translate to $\frac{1200}{4 \cdot 4} = 75$ FPS at 1280x720 resolution.

However, upon discovering that a pixel can only be written to DRAM every two cycles, we were effectively limited to $\frac{83333333}{1280 \cdot 720 \cdot 2} \approx 45$ FPS, which led us to focus on other optimizations such as finer depth calculation.

Finally, in our implementation, we actually attempt to synthesize 121 multiplications ($7 \cdot 16 = 112$ in renderer, 9 in projection). Somehow, Vivado manages to use 120 DSPs for this. We believe it would be possible to reduce the number of multiplications used in projection, but we ended up not having to deal with this because Vivado is magic.

IV. Contribution Statement

Both Roger and Alan worked together to formulate the initial approach for the data path and optimizations. Roger implemented obstacle generation and most of the obstacle/sprite triangle conversion, while Alan implemented game logic. Roger and Alan worked together on triangle projection. Roger implemented most of the renderer and wiring the BRAMs and DRAM with the rest of the module, except for the pixel calculation, which was authored by Alan.

V. Acknowledgements

We would like to thank Joe Steinmeyer for his instructional skills, constructive feedback, and continued support as the mentor for this project.

VI. Code Repository

The code for this project can be found at https://github.mit.edu/6205F25/fa25-6205-team50.

References

[1] MA Darlene Antonelli. How to play subway surfers. *WikiHow*, 2025.