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 320×180 resolution. A carefully engineered datapath enables frame rates up to 400 frames per second, far exceeding our baseline performance target. We evaluate the system's BRAM, DSP, and timing characteristics, and discuss optimization opportunities including bounding box and triangle orientation culling.**

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

I. Background Information

Our project aims to create a vastly 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 (both stationary and incoming). 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 latest iteration, we have yet to integrate our game logic with the graphics datapath, but we are reaching speeds of up to 400 FPS at 320x180 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.
- **Triangle Painting (II-D).** The projected triangles are depth-consciously drawn into a single frame.

A. Obstacle Generation

We have found it helpful for our design to subdivide obstacle placement and location information by *half-block*. A half-block is defined as half the length of a train car/ramp. 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
- In the final version, we also aim to include train cars moving in the direction of the sprite.

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
- $\frac{1}{32}$ to middle barriers
- $\frac{1}{64}$ to low barriers
- $\frac{1}{64}$ to high barriers
- $\frac{3}{64}$ to train cars, and
- $\frac{1}{64}$ to ramps.

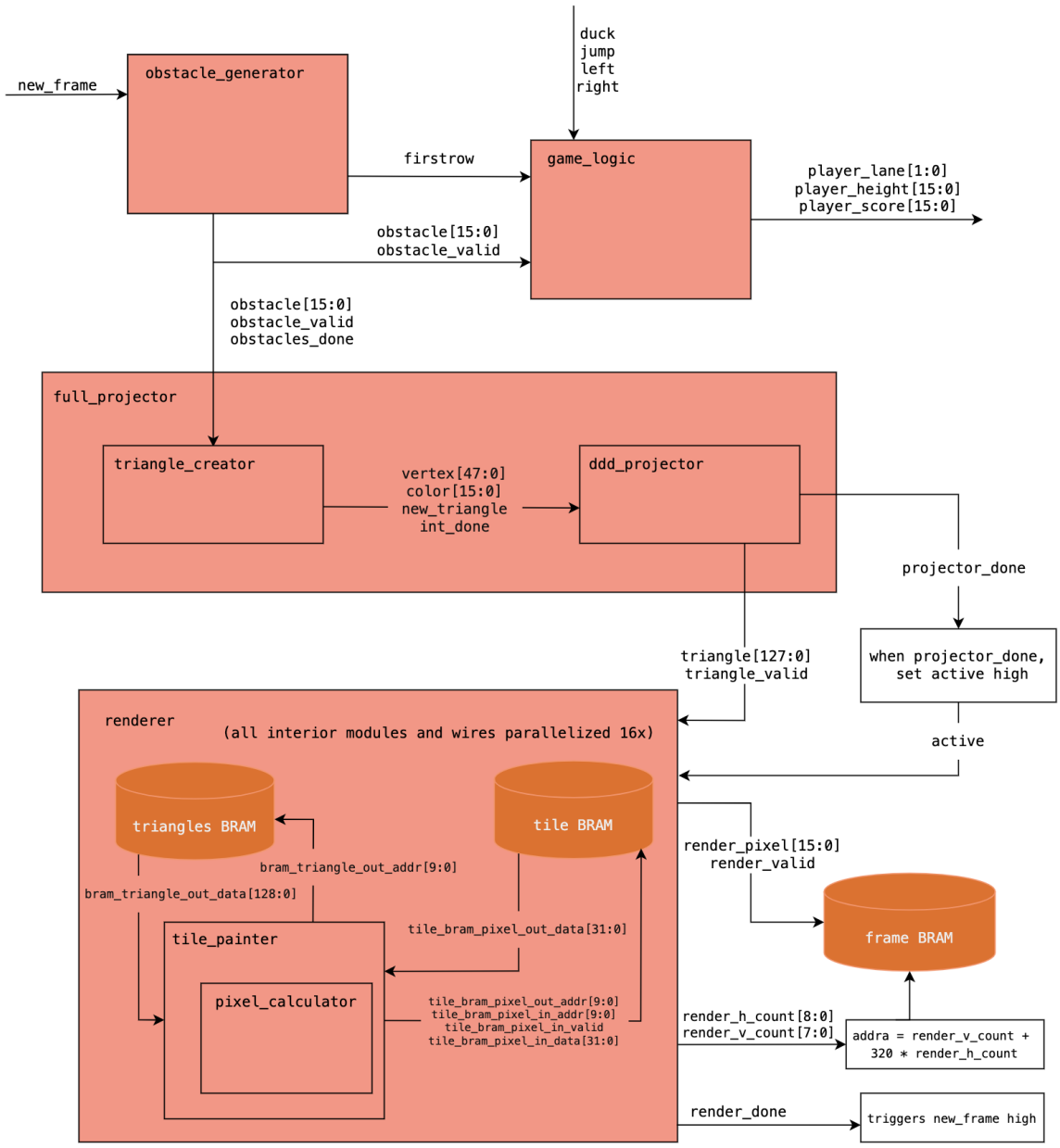

Fig. 1: Block diagram for obstacle generation, game logic, triangle projection, and rendering.

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

Each of the 4 buttons on the FPGA serve as (debounced) inputs (duck, jump, left, right) for player movement in the game. 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, as currently the only criteria for ending the game is crashing into an obstacle.

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 and location within the current half-block to determine intersections. Any intersection immediately results in the game ending with game_over being set 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.

On the new frame signal, the sprite is advanced by a depth of SPEED. The player's horizontal movement is dictated by

left and right controls, while the player's vertical movement is dictated by gravity and ducking. 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).

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, which will be included in future iteration graphics of this project.

C. Projection

Once 16-bit obstacles are produced and sent along the data path, the projection stage begins, consisting of two separate modules: triangle 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, it stores the depth of the obstacle, the type of the obstacle, and the lane it is in. Depending on the type of the obstacle, we have hardcoded values for the triangles that should be outputted. The coordinates of these triangles are relative to the lane and depth of the obstacle. 16-bit color will also be outputted alongside of the triangles.

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) Triangle Projection: The triangle projection module currently 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 supplied to this module, along with a single color. The module outputs a 128-bit triangle, with 16 bits for each of the following values in order:

[color][p1x][p1y][p2x][p2y][p3x][p3y][depth]

To project each vertex into 2D space, a plane of projection $z = z_0$ must be determined, after which (x, y, z) is transformed into $(xz_0/z, yz_0/z)$. By setting z_0 to be a power of 2 (64), 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. Finally, depth is naively calculated as the sum of all 9 x, y, z coordinates' squares, with bitshifting as appropriate to avoid overflow.

D. Triangle Painting

As the triangles are projected in sequence by the projection module, they are wired to be stored in 16 triangles BRAMs, each storing a full set of triangles for the frame.

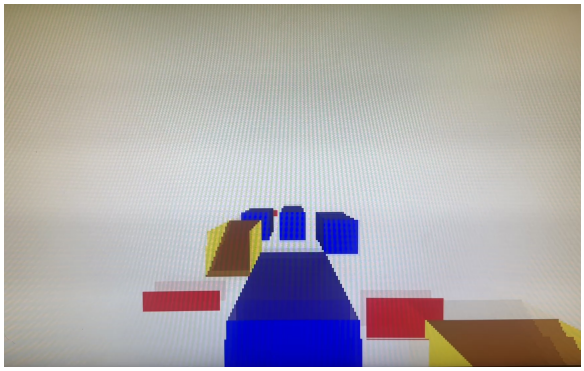

Fig. 2: HDMI output in our current version.

Upon the completion of the projection stage, the rendering stage is triggered. Painting is then handled by a tile painter module, which is wired together with both the corresponding triangles and tile BRAMs in the 16-way parallelization. Pipelining and storage interfacing is done within a broader renderer module to ensure that the following sequence of events happen in order:

1) The module requests a triangle from the triangles BRAM and pixel from the tile BRAM.
2) After 2 cycles, the requested triangle and pixel is provided to the module.
3) A single-cycle pixel calculation is done to determine whether to recolor the pixel. This happens iff the pixel is inside the triangle (computed with a coordinate cross product), and the triangle has a depth smaller than the existing pixel depth.
4) While the same triangle is maintained by the module, all (x, y) coordinates within the tile are repainted by cycling through all pixels, and resulting pixels are outputted to be written to the tile BRAM.
5) The next triangle is requested, restarting the cycling of all pixels in the tile as well.

The tile painter module can also wipe all tiles to prepare for the next set of tiles to be painted in parallel, which is handled by the renderer module. Given that we currently paint 16 20x45 tiles in parallel, we must wipe each tile BRAM 4 times per frame once all stored pixels are sent to the frame buffer. Note that however, we do not wipe any triangles BRAMs, instead overwriting them for each frame and keeping track of how many triangles appear in the frame to be fed into the tile painter module (num_triangles). Upon completion of a full frame of painting, we restart the entire process for the next frame.

In the final version of our product, we anticipate using DRAM for our frame buffer. However, the current iteration at 320x180 resolution is able to use additional BRAM on the FPGA to fully store frame data.

III. Evaluation and Further Directions

We are currently meeting timing with a 83.333 MHz clock, with a WNS of 0.133 ns. We are working to increase slack and to possibly increase our clock speed. Our current design uses

all 75 BRAMs and 99 DSPs, though in our final version, we will be using DRAM for our frame buffer, freeing up BRAMs and possibly allowing more parallelization of our tile painters.

Our current design has around 64 triangles in frame at a time and empirically runs between 256 and 400 FPS at a resolution of 320x180. This already meets our initial goal of 320x180 at 30 FPS. Our stretch goal was to run at 15 FPS at 1280x720 resolution, but this goal is already within reach, as 256 FPS$/16$ times resolution increase $= 16$ FPS. We hope to, however, reach a faster final frame rate on HD resolution by implementing the following optimizations:

- Implement "bounding box" culling in our tile painters, only considering pixels for triangles whose bounding boxes intersect the current tile. We believe this is a quick change, and we are working on this now.
- Implement triangle orientation culling. We would always create our triangles in counterclockwise vertex order, so that if its normal vector points in the positive-z direction, it is the back-face of an obstacle and should not be rendered at all.

Beyond optimized graphics rendering, we still need to integrate our game logic with our project, implement the 3D sprite object, and implement moving train cars. If we have extra time, we hope to implement basic shading or textures.

REFERENCES

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