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

6.205 Final Project Block Diagram

Alan Lee (alanlee), Roger Fan (rogerfan)

October 28, 2025

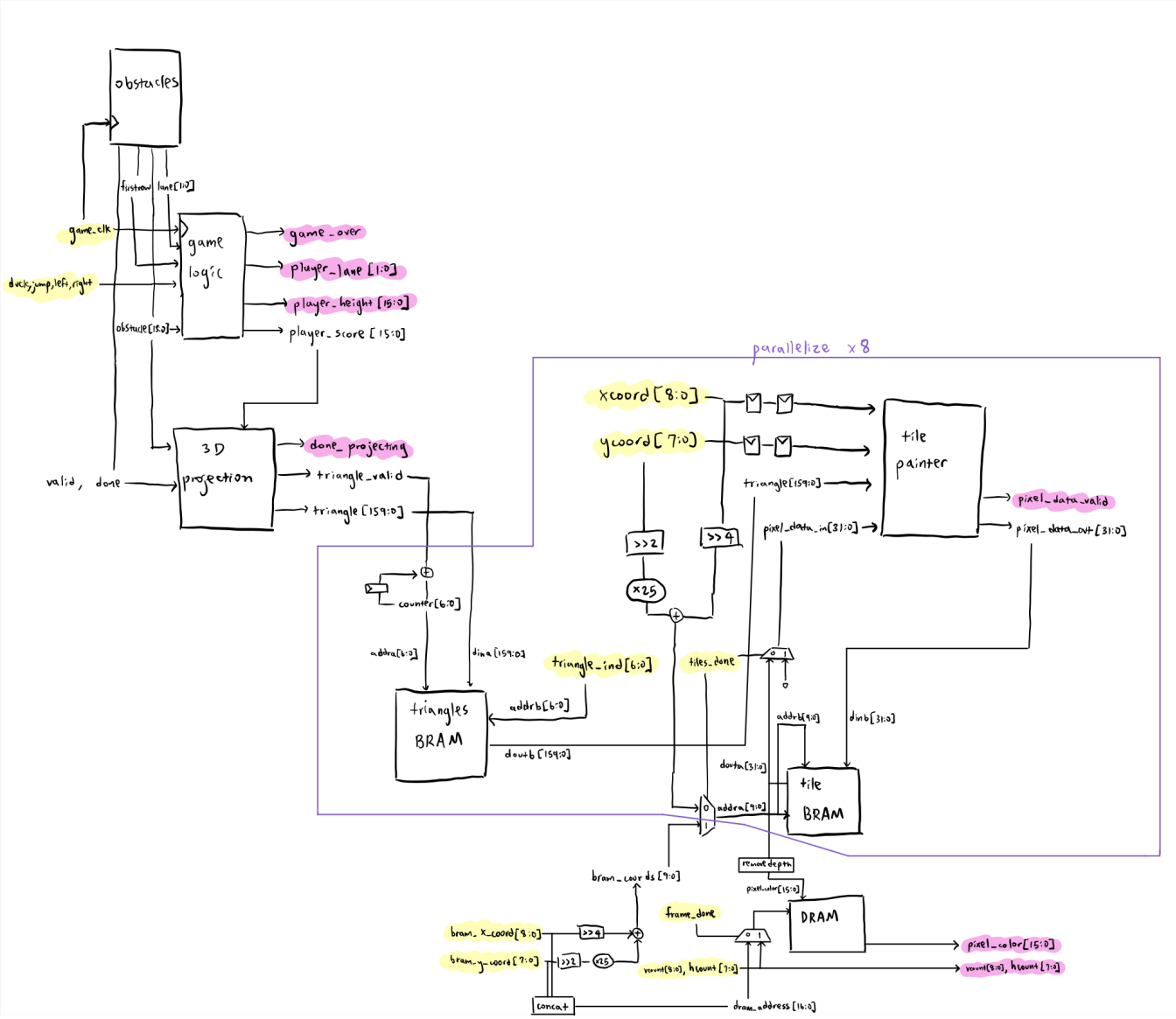
1 System Overview

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 will 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. In this report, we outline the key modules that will come together to support efficient 3D graphics (focusing mostly on obstacle rendering) and variegated game logic.

We anticipate the core modules for this project to be

- a 3D projection module responsible for reading obstacle data, converting the data into triangles with 3D coordinates, then projecting the 3D coordinates into 2D HDMI space while maintaining a depth of each triangle vertex.
- multiple tile painter modules, which will interact with BRAMs as they are used to iterate through pixels to draw triangles in a depth-preserving manner. We will plan for 8 initially to evaluate the feasibility of painting 8 tiles in parallel, and scale up or down accordingly. We also anticipate having logic to avoid unncessary math if a triangle's bounding box doesn't intersect with the module's associated tile at all.
- a game logic module responsible for determining various game components such as if the game is over, jumping or ducking behavior, and resetting a game.

2 Block Diagram



2.1 High Level Details

We will be using the HDMI pixel_clock from previous labs to mandate statefulness within our system. Furthermore, we will also define a game_clk to rise once every time a full frame has been produced and written to DRAM.

The game_clk operates the obstacle generation and game logic modules. The latter takes in player controls (buttons and switches to be determined from the FPGA board) to compute whether the game is over, as well as player location information encoded in the form of lane (x), height (y) and score (z).

During the **triangle generation** stage, obstacles will be fed one at a time to the 3D projection module, which will convert the obstacle into a set of triangles to be displayed. Each triangle will be written to 8 different triangles BRAMs for parallelization, and the triangle_valid signal will be used to inform whenever the 3D projection module has completed the projection of one triangle. The done_projecting signal will be used to notify when to move onto the next stage of tile painting.

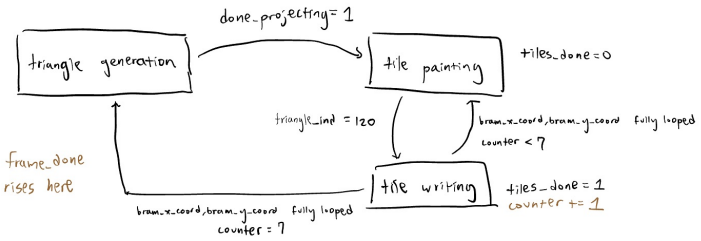
Tile painting happens on 8 tile BRAMs at once, each containing the data for a 20 x 45 tile of pixels. For each of the at most 120 triangles stored in the corresponding of the 8 triangles BRAMs, the triangle will be read out using triangle_ind. Then looping through xcoord and ycoord, we will use tile painters to continually recolor pixels in the tile whenever a triangle contains the pixel and is closer to the player than any previous triangle. We use pixel_data_valid as a signal to determine when we are ready

to rewrite the modified pixel for a given triangle to the corresponding tile BRAM.

After all 8 tile BRAMs have been populated with the pixel data, we move onto the **tile writing** stage, where using bram_x_coord and bram_y_coord to iterate through the appropriate tile BRAMs, we can write the tile data into DRAM. At this point, the tiles_done signal will be held high to allow for specific data flow from the tile BRAM to the DRAM.

The tile painting and tile writing stage will be alternated between for 8 times for each frame to obtain all 64 tiles necessary for a given frame. Then frame_done can be held high and we will use HDMI hcount and vcount signals to read from the DRAM.

2.2 FSM for 3D Graphics



As mentioned, we will have three main states for our graphics finite-state machine. To transition from the triangle generation state, we need to be done with projecting and storing all data in the triangles BRAMs. We then alternate 8 times between the tile painting and tile writing states, setting tiles_done as low and high respectively to dictate when we are writing to tile BRAMs or reading from them to write to the DRAM.

To go from the tile painting to tile writing state, we must have iterated through all 120 triangles stored in the triangles BRAM (for each of the 8 parallel data paths). Note that we only move on to the next triangle in this initial design if all 900 (xcoord, ycoord) pairs for the tile have been passed through the tile painter.

To return from the tile writing state to the tile painting state, we must have iterated through all 900 (bram_x_coord, bram_y_coord) pairs for the relevant tile, and all 8 tiles on their parallel data paths must have finished iteration to all for a new set of 8 tiles to start being painted.

On the completion of the 8th and final tile writing state, frame_done will be held high to control game_clk and we will return to the triangle generation state to begin painting the next frame.

2.3 IPs Needed

- BRAM interface - we plan to store intermediate tiles (those without all triangles processed/drawn) in BRAMs, as well as lists of projected triangles for tile painters to access.
- DRAM interface - we will need DRAM as described below for storing fully rendered frames before reading from the DRAM to display the data through HDMI. We will likely use an implementation similar to that in lab 6 which allow for alternating reads and writes from DRAM.

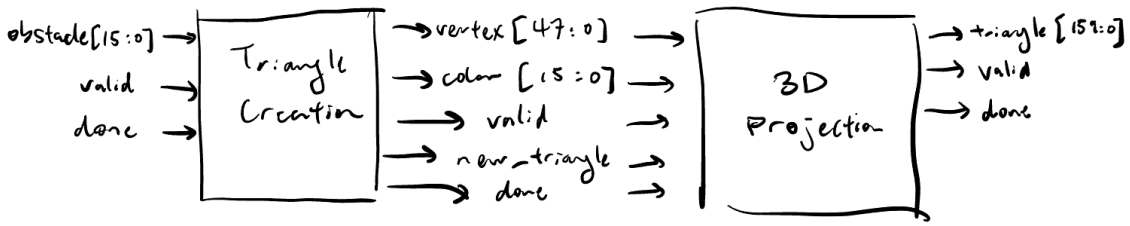
2.4 Memory

The most memory-intensive part of this project will be the pipeline along which obstacles are transformed into triangles, which are then painted one by one onto rectangular tiles, before finally being consolidated into a single frame to be displayed via HDMI. We outline the estimated memory requirements of each stage below.

- **Obstacles.** Obstacles can take the form of jump barriers, duck barriers, jump/duck barriers, ramps, stationary trains, and moving trains. An obstacle needs information about which lane it is in (2 bits), what type of obstacle it is (3 bits), and how far it is away from the player (11 bits). The obstacles will be stored in a grid which already encodes spacial information (see 5 Obstacle Generation), requiring a total of only 192 bits to store all the obstacles, which can be done with registers.

- **Triangles.** We anticipate each obstacle to be comprised of at most 6 triangles, each of which will have three vertices with three coordinates ($3 \cdot 3 \cdot 16 = 144$ bits). To start off, each triangle will have just one color (16 bits). We anticipate at most 20 obstacles visible needing to be displayed across the three lanes of gameplay at any given time, hence at most 120 triangles. This leaves us with $120 \cdot 160 = 19.2$ kilobits, a task suitable for BRAMs. We plan to start with 8 BRAMs, all of which contain an *identical* list of triangles. Although this is redundant, this can achieve the goal of parallelization between the tile painters, which will not need to share BRAM reads to complete their job.
- **Tiles.** Painting the entire screen will be done in small rectangular chunks called *tiles*. We will likely start off with a small 20 x 45 tile size (leading to 64 such tiles in a 180 x 320 screen). For each triangle drawn on a tile, we will need to review all pixels on the tile, decide if the triangle contains it, and if so, decide if the triangle should be drawn over the current pixel using depth comparisons. This requires depth and pixel color to be stored in the BRAM, both of which should not exceed 16 bits. This leads to $16 \cdot 2 \cdot 20 \cdot 45 = 28.8$ kilobits, which we think is appropriate for individual BRAMs, for a total of 8 additional BRAMs (one for each tile painter module).
- **Fully Rendered Frames.** To ensure seamless stitching of tiles into a single total frame, we will need to eventually store all data in a much larger memory space. For 180 by 320 graphics, this will require at least $16 \cdot 2 \cdot 180 \cdot 320 = 1.84$ Mbits, and at most up to double that size if we build one frame while displaying another.

3 3D Projection Module



This module will comprise the triangle creator module and the projection module. The triangle creator module will take in an obstacle and serially output a series of triangles situated in 3D space. The projection module will then take the triangles and project them onto the camera, returning the final triangle coordinates.

Note we also make the simplifying assumption throughout that the camera is situated at $(0, 0, 0)$ in space and points in the positive z direction. We will then situate the obstacles accordingly.

3.1 Triangle Creation Module

Each obstacle consists of at most 6 triangles, and there are around 6 types of obstacles. First, the "center" (x_c, y_c, z_c) of the obstacle will be computed from its depth and lane. y_c will be a constant negative value, x_c depends on the lane, and z_c is simply the depth of the obstacle, possibly bitshifted for scale. (More complicated transformations can be performed later on, but this is a good place to start.)

Given the center coordinates, the triangles composing the obstacle will be encoded in relative terms to its center. The triangle creator module will output one vertex of the triangle at a time, consisting of three 16-bit signed values. There will also be a valid wire and a one-cycle-high new-triangle wire, indicating whether or not the data is valid, and whether or not the current vertex is part of a new triangle.

When new-triangle is high, the current and next 2 clock cycles will be the coordinates of the same triangle. There will also be a 16-bit color output wire that will hold a constant value during these 3 cycles. Finally, if the input done is high, then the module will also output done as high.

3.2 Projection Module

To project these 3D-triangles, note a point at (x_0, y_0, z_0) will be projected onto the plane situated at $z = D$, where D is some constant, at $(x_0 \cdot D/z_0, y_0 \cdot D/z_0)$. We will choose D to be some power of 2 to make the multiplication a simple bitshift.

We will also start with the simplifying assumption that, for the purposes of deciding which triangle is in front of which, we can simply consider the average depth of their vertices. Thus, in addition to a triangle's projected (x, y) coordinates, we will also calculate (with a pipelined divider) the sum of its z coordinates divided by 3.

After everything is computed, we will hold the output valid high, and output the full 160 bit triangle for the BRAM. (As before, if the input done is high, then we will output done high as well.)

(In a full system, for every pixel on our screen, we would ideally compute the 3D depth at which the ray from the observer to the pixel intersects each triangle. This allows us to be precise in determining which triangle lies in front of which. In this case, we will have to calculate the vector normal to the triangle, which involves 6 multiplications that can be done in parallel, and the vector norm's dot product with the triangle's vertices, which involves 3 more multiplications.)

4 Tile Painter Modules

We will be using 8 tile painter modules in our initial design to parallelize the process of computing pixel colors within tiles. Each tile painter will take as input a set of coordinates in 320 x 180 space (9 bits for x coordinate and 8 bits for y coordinate), along with a 160 bit triangle and 32 bit pixel_data_in. The latter object stores 16 bits of color information and another 16 bits of depth information.

The tile painter module will use the pixel coordinates given to it to first determine if it lies within the triangle. This can be done with a set of cross product calculations; suppose that we want to decide if a point $P = (p_x, p_y)$ is inside the triangle ABC with similarly defined x and y coordinates. Defining the vectors

$$AB = (b_x - a_x, b_y - a_y), BC = (c_x - b_x, c_y - b_y), CA = (a_x - c_x, a_y - c_y)$$

$$AP = (p_x - a_x, p_y - a_y), BP = (p_x - b_x, p_y - b_y), CP = (p_x - c_x, p_y - c_y)$$

and cross products

$$C_1 = AB \times AP, C_2 = BC \times BP, C_3 = CA \times CP,$$

it is a mathematical fact that the cross products are all the same sign if and only if P lies within ABC. We plan to do all three multiplications on hardware in one clock cycle to compute the three different values, and since we will be using up to 8 tile painter modules at a time, our overall design requires at most 24 hardware multipliers. We do not anticipate this number to significantly increase due to the general avoidability of multiplication in other modules.

Then, if a pixel does lie within the triangle, the module compares the depth of the triangle with the depth of the existing pixel from pixel_data_in. If the triangle is closer to the player than the existing pixel, the color information is overwritten using the triangle's color, and otherwise unchanged. When computation is finished, the output will be placed on pixel_data_out and pixel_data_valid will be held high to indicate completion.

As a result, for any given pixel, after passing all 120 triangles into a tile painter and repeatedly feeding pixel_data_out back into the module as pixel_data_in (while storing intermediate pixel data in BRAMs), the final color should be correctly chosen as that of the closest triangle containing that pixel to the player.

Since multiplications are nonetheless costly compared to additions or comparisons, we are also considering expediting the process by computing the coordinates of the bounding box of the triangle first. If the pixel lies outside of this bounding box, it will pass the module unmodified.

5 Obstacle Generation

Obstacles aren't arbitrarily generated, but instead are only generated at half-train blocks. We will keep track of 16 half-blocks for every lane, storing all the data in *registers*. Note that the position information of the obstacle is represented by which register the obstacle is in, so each half-block need only store 3 bits for what type of obstacle it holds, for a total of $3 \cdot 3 \cdot 16 = 192$ bits.

Every game_clk rising edge, we will increment a counter representing how far into the current half-block the player is. If this counter is the length of a half-block, we reset it and must now perform additional operations:

- First, shift all obstacles in the registers a half-block forward.
- Then, generate obstacles in the new half-blocks at the end that are available. As a minimum viable product, we will randomly generate any obstacles that fit, but as we iterate, we hope to be able to use more complicated logic that ensures that no impossible patterns are generated.

Once this process is over, the module will then output the obstacles serially, one per clock cycle (or slower, depending on how slow the following computations are). The obstacle's depth will be calculated based off of its depth in the grid of half-blocks and the player's current depth into the first half-block.

Additional information of the obstacle will be outputted with it, like the obstacle's lane, and whether or not the obstacle is in the first row. After all obstacles are finished outputting, the done output will be held high for the rest of the game_clk cycle.

6 Game Logic Module

The game logic module will hold internally the state of the player. It will hold:

- 16 bits for the player's height
- 16 bits for the player's score
- 2 bits for the player's current lane
- 1 bit for whether or not the player is jumping/airborne
- 1 bit for whether or not the player is ducking
- 8 bits for how long the player has been ducking
- 8 bits for the player's vertical velocity
- a counter for how far the player is into the current half-block (see 5 Obstacle Generation)
- 1 bit for whether or not the game is over

At the rising edge of game_clk, the player's new position in the absence of obstacles will be computed:

- If jump input is high, and the player is not already jumping, the jump state will be set to 1 and upwards velocity will be set to some predetermined value.
- If duck input is high, set the duck state to 1. If the player is jumping, set downward velocity to some fixed negative number.
- If left input is high, decrease the player's lane (if not already 0).
- If right input is high, increase the player's lane (if not already 2).
- Regardless of inputs, if the player is jumping, decrease vertical velocity by a fixed gravity value, and add vertical velocity to the player's height. If the player's height is nonpositive, set jumping to 0 and vertical velocity to 0. If the player is ducking, increase the counter for how long the player has been ducking, and if it is at some fixed value, set the ducking state to 0.

Then, when the module receives obstacles from the obstacle generator, we wait until we find an obstacle that is in the player's lane, and in the first row. We then split based on the type of obstacle:

- If the obstacle is a barrier, and the player is halfway into the current half-block, we check if the player's height is over the barrier, or if the player is ducking. If not, the game is over.
- If the obstacle is a full train car, check if the player is standing above it (with some margin of error), or if the player's previous height was above it, but the new height is below. Then, set the player's height to the top of the train car, and set jumping and vertical velocity to 0. If this check fails, and the player is inside the train car, then the game is over.

- If the obstacle is a ramp, we will calculate the height that the ramp is given how far into the current half-block the player is, and then do the same collision detection as before.

The module will then output the player's height, score, current lane, and status (whether or not the game is over).

7 Conclusion

This design satisfies the major requirements for interacting with and visualizing a simplified version of Subway Surfers. The game logic should relatively easily accomplish the first goal. Furthermore, as we later experiment with time and resource allocation across the 3D projection and tile painter modules, we will be able to take advantage of the FPGA's 2 clock-cycle BRAM memory and parallelization to improve our design and allow for higher-resolution or higher-frequency graphics.

By starting off with a very simplified and constrained set of design objectives, we hope to achieve a rudimentary but working end-to-end game that can both be tested at the module level and the overall game level. Most of the unsolved problems in this project remain unknown, and we anticipate this project to be highly iterative; the best way to discover and address these problems is to get a working version as quickly as possible, then make small changes one at a time to each part of the block diagram, targeting efficiency, scale and graphical quality.

References

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