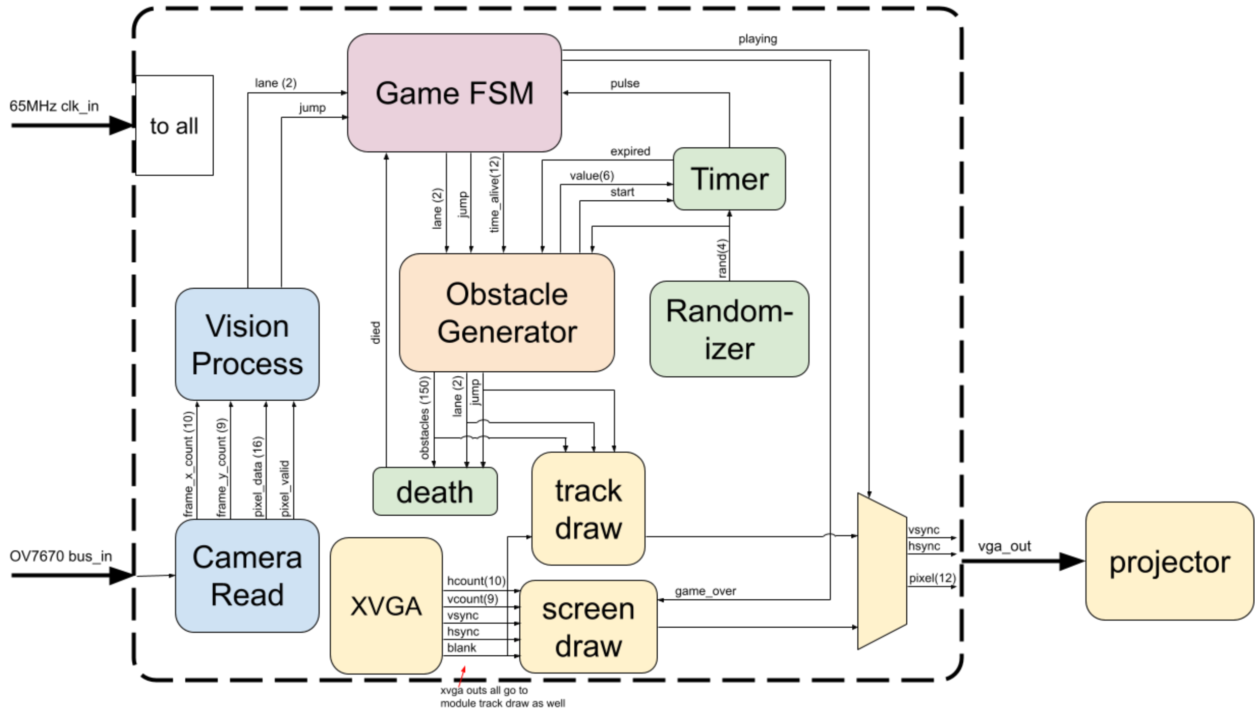
# 6.111 Final Report: Infinite Run

Lili Sun and Alex Studer

## Introduction

Our project was inspired by the popular mobile game *Temple Run*, in which the player must switch between different lanes in order to dodge oncoming obstacles. We wanted to build a real-life version of this game, in which the player would have to physically move themselves and jump in order to avoid obstacles. Inspired by 6.111 projects from previous years, we wanted to use camera input to drive the game, as we felt it would add both a layer of interactivity and a layer of technical complexity to the project.

## Overall architecture and block diagram

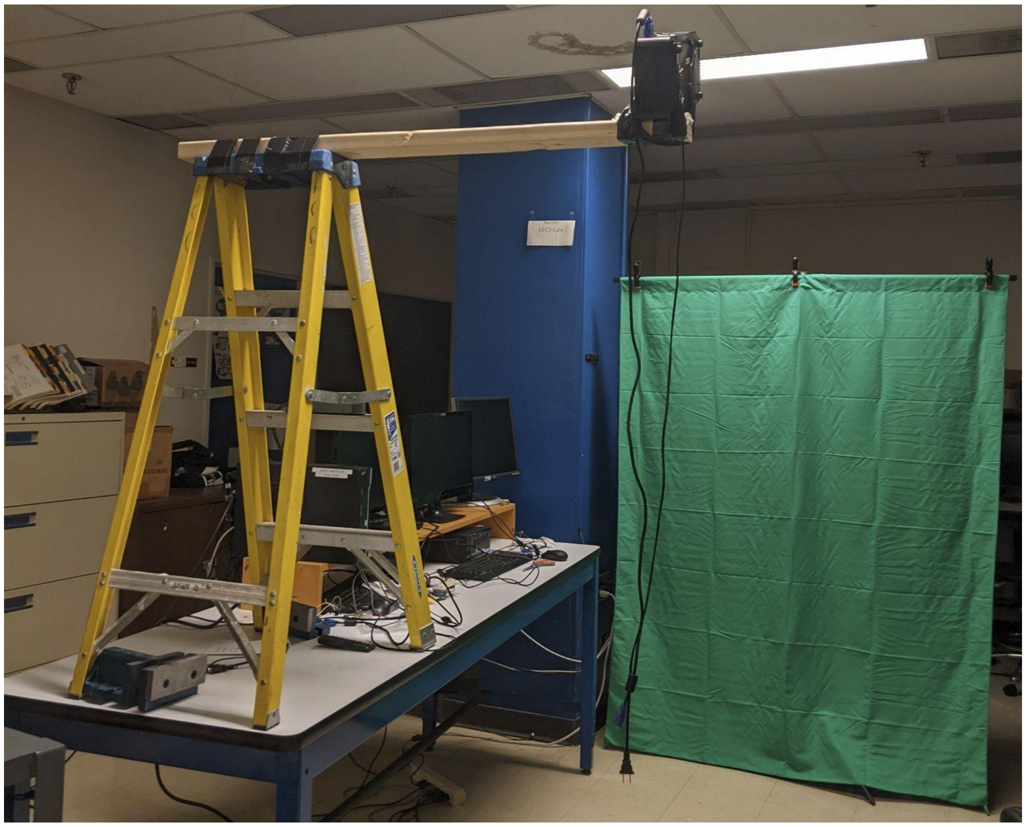
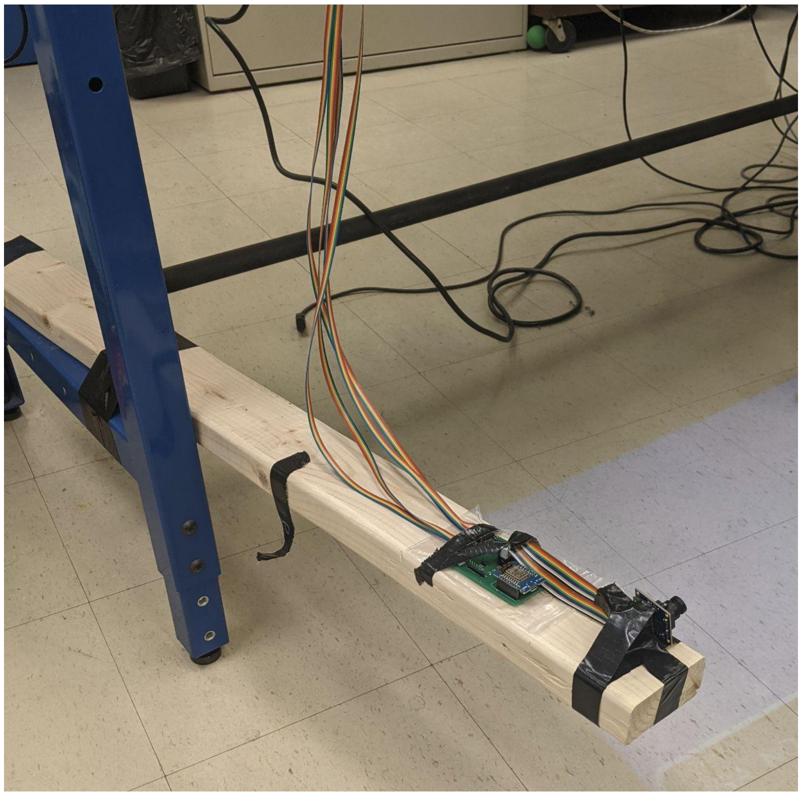


We grouped our system's architecture into three main subsystems:

- Vision pipeline -- processed camera frames into gameplay input
- Gameplay -- managed internal game state
- Display -- handled rendering of the various game screens

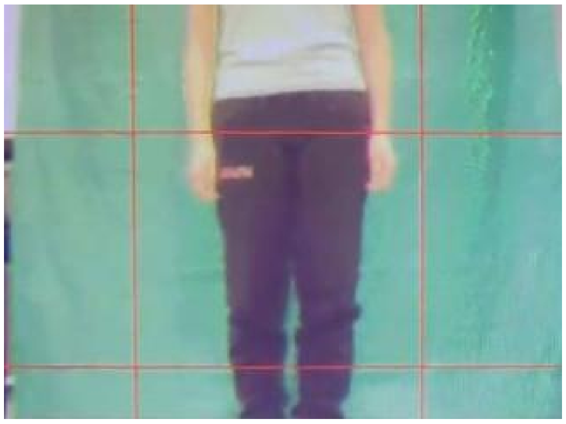
## Hardware setup

For our hardware setup, we had a projector mounted above the floor on a table and ladder setup and had the track projected onto the floor. We also had the camera mounted near floor level to register jumps more reasonably which pointed back at a green screen mounted behind the player. After some experimentation we also placed a white paper on the floor to project onto and make colors more vivid.

## Vision pipeline

As you can imagine, one crucial aspect of the project was in how we processed vision data. We used an OV7670 camera board, which contained an ESP8266 microcontroller to initialize the camera. The basic idea was to break the frames into nine segments, and determine which of the segments were occupied by the player. The segment data could then be processed into a 2-bit lane index and a 1-bit jump state.



*A camera frame, divided into segments.*

→

| | | |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 1 | 0 |
| 0 | 1 | 0 |

*A display of occupied segments.*

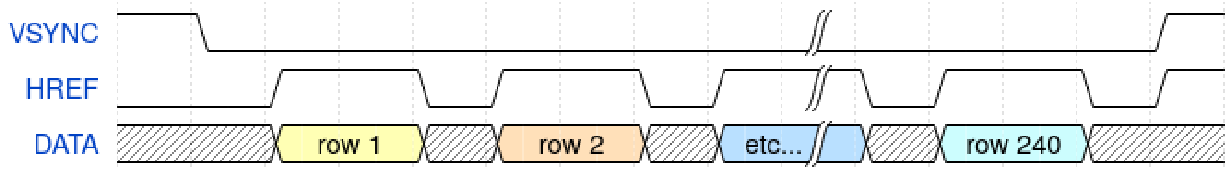
```
lane: 2’b01
jump: 1’b0
```

*The final output of the vision pipeline, as run on this frame.*

### Camera Read module (Alex)

The OV7670’s data was first sent to our Camera Read module. The pixels are in an RGB 5:6:5 format, but the data bus is 8-bits wide, so the OV7670 transmits half of each pixel at a time. The Camera Read module (a modified version of the provided sample camera code) assembles these two halves into a single 16-bit value. It also monitors the camera’s HREF (which indicates the end of a row) and VSYNC (which indicates the end of a frame) signals to track the X and Y coordinate of each pixel.

You can see what these signals look like in relation to each other in the following timing diagram that we created, which shows the transmission of a frame from the OV7670. Note that the timing is not cycle-accurate (it will obviously take more than two clock cycles to transmit a row’s worth of data).



***Note on replicating this***
The timing diagram shown here is specific to the register configuration of the OV7670 that we used, which can be found in our GitHub repository. The behavior of HREF and VSYNC can be controlled via different registers, which can be found on pages 48 through 50 of the OV7670 datasheet.

(this configurability may be part of the reason why there are no clear timing diagrams in the datasheet, or at least none that we could find)

The falling edge of VSYNC indicates to our SystemVerilog code that the frame has started, the falling edge of HREF indicates that the row has ended (and therefore we reset our X counter and increment our Y counter), and the rising edge of VSYNC indicates that the frame has ended.

## Vision Process module (Alex)

The 16-bit pixels and locations are then sent to the Vision Process module. This module compares the green channel of each pixel against a threshold value, and assigns the pixel (based on X and Y coordinates) to one of nine segments. If a segment has at least seven matching pixels, then that cell is considered “active”, meaning the player is occupying that part of the frame.

Once all the pixels in a frame have been processed, the Vision Process module evaluates these cells to determine two outputs: first, what lane the player is in; second, whether the player is jumping or not. In the case where any lane has a majority of segments, that lane is chosen; otherwise, the rightmost lane is selected. A jump is detected when none of the segments in the bottom row are detected.

These modules were intentionally designed in a “streaming” fashion, meaning that, for this aspect of the vision pipeline, no framebuffer is required to store the camera data. The goal of this was to allow using the camera as its native resolution of 640 x 480, as opposed to the provided sample code, which processed frames at a resolution of 320 x 240. Storing full 480p frames in block RAM would require $640 \times 480 \times 16 = 4,800\ Kbits$ of memory. (while the Nexys4 DDR’s FPGA, the XC7A100T, has 4,860 Kbits of block RAM, this would have left no room for any other purposes, such as ILA usage or adding images) However, we ended up needing a framebuffer for other purposes (as you will see in the Camera Debug Draw module), so this benefit was negated somewhat. Performance was also deemed acceptable at the lower 320 x 240 resolution.

## Vision Debouncer module (Alex)

During integration of the vision pipeline with the rest of the gameplay, we noticed that, when jumping, it was difficult to stay in the air long enough to allow the obstacle to fully pass, especially at lower game speeds. The player would correctly jump to avoid an obstacle and yet land too soon, causing the game to end and creating a frustrating experience. Therefore, the Vision Debouncer monitors the jump output, and, if a jump finishes before a certain amount of time (750 ms), the length of the jump is artificially extended, allowing time for the obstacle to continue offscreen.

The Vision Debouncer also handles a clock domain crossing. Most of the system runs at 65 MHz, but the Camera Read and Vision Process modules are necessarily run in the domain of the camera's clock, which is slower and not necessarily at the same phase. This synchronization is achieved by sending the lane and jump outputs through two synchronization registers, which absorb any potential metastability caused by timing violations.

The module outputs synchronized versions of the lane and jump signals, both before and after the jump extension. The extended signals are used in gameplay; however, there are some scenarios where the unextended signals are preferred. For example, when the player initially jumps in order to start the game, the unextended signal is used; otherwise, the player would notice an odd delay between the end of their jump and the start of the game.

## Camera Debug Draw module (Alex)

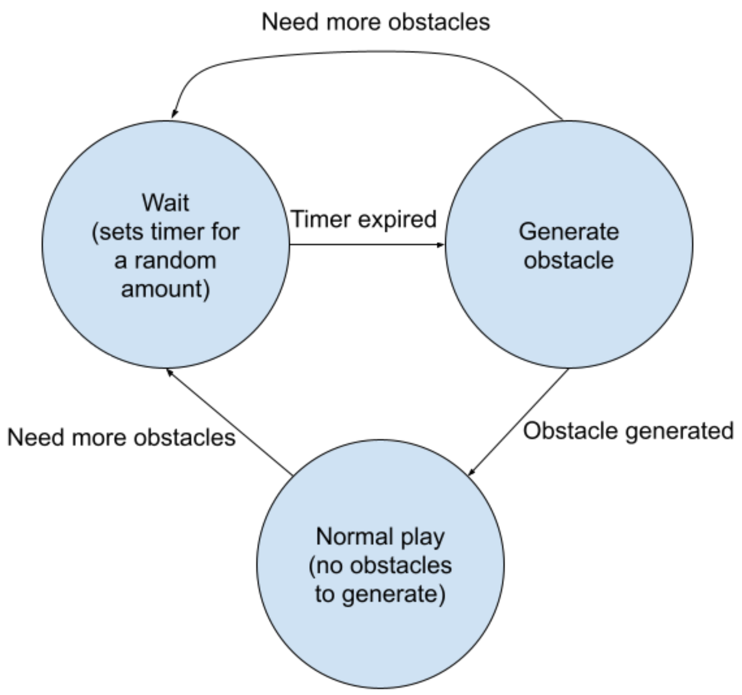
In order to debug this pipeline, the Camera Debug Draw module was developed, which displayed the camera frames, segment results, and final lane/jump data over the VGA output. This way, the results of each stage could be verified.

To display frame data, the pixels from the Camera Read module were stored into block RAM, and then read and displayed on the VGA output. The other information (segment results and lane/jump) could be read from relevant modules and used to display colored rectangles at the appropriate positions. Note that the Camera Debug Draw module contains its own set of synchronization registers, enabling the output of modules (potentially in the camera clock domain) to be connected directly. In practice, the data normally came from the synchronized outputs of the Vision Debouncer module. However, the ability to quickly swap this out for debugging purposes, combined with the minimal latency impact of the additional registers, meant that we left these in.

While it was originally intended that this module would be removed after the pipeline was verified, it ended up being useful for on-the-fly debugging. (for example, if the green screen had been moved, we could use this module to assist in repositioning it) This module's requirement to have block RAM meant that we ultimately needed a framebuffer, even though the rest of the pipeline had been designed to operate without one.

## Gameplay

### Obstacle Generation module (Lili)



*Block diagram displaying how the module decides to and generates more obstacles.*

This module generates new obstacles as the player plays the game. It uses the time alive input from the game FSM to decide how many obstacles should be on-screen at once (with a maximum of 10), as well as how fast the obstacles on screen should be moving. Both of these increase as the player has been alive for longer. Once it decides a new obstacle needs to be added, it waits a random amount of time (using the timer module) and adds it to the array (as in, sets the object's active bit high and other values). This module took three random inputs, four bits for the time to wait as previously described, two bits for the lane for the new obstacle, and another two bits for the sprite type.

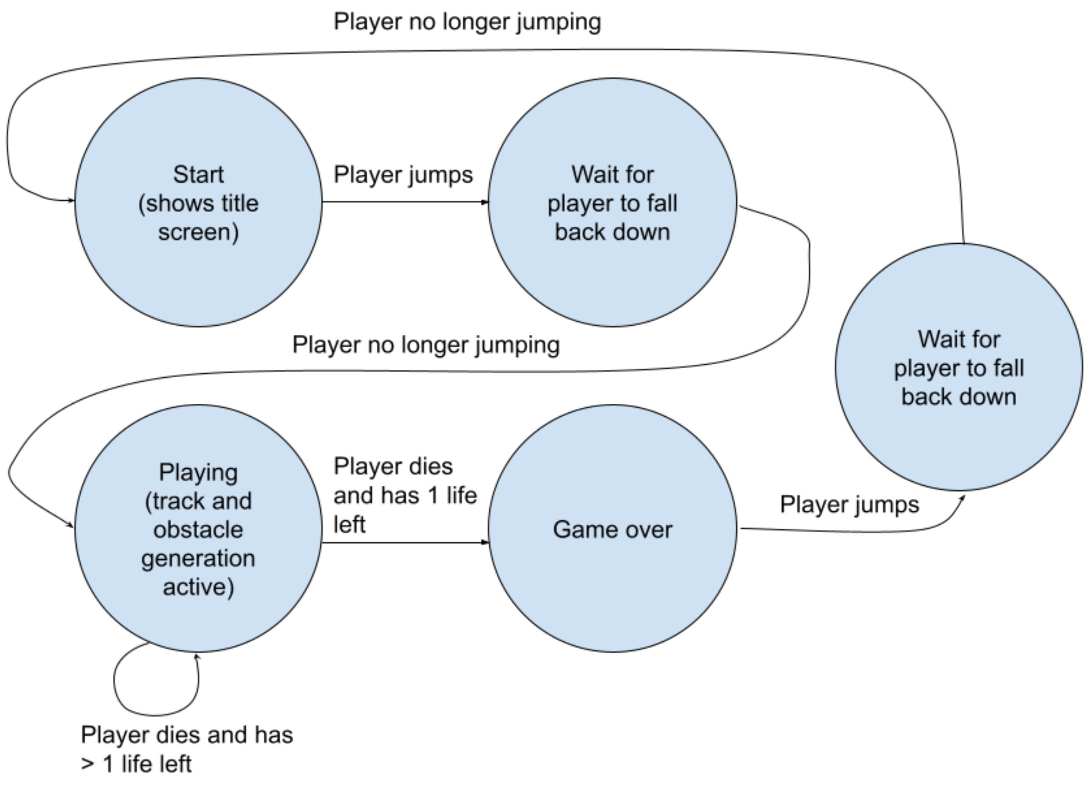
```
// obstacle: type, position, lane, active
// ttppppppppppplla
// [15:14] [13:3] [2:1] [0]
typedef struct packed {
      logic [1:0] sprite_type;
      logic [10:0] position;
      logic [1:0] lane;
      logic active;
} obstacle;
```

*Obstacle struct type including sprite type, position, lane, and active bit.*

One issue we ran into was keeping track of where in the array we could place the next obstacle. Obviously as the obstacles move off screen the module has to keep track of it and set the active bit to 0. Initially we implemented this using a two pointers approach (keeping track of the start and end indices of current active obstacles in the array, and although this was test-benched, not all cases were covered in those tests). Once we tested it in integration with track draw and other modules we found there were some bugs with the wraparound so we ended up using some combinational logic on the obstacles array to get the next free index.

```
wire [3:0] next_free_slot = (!obstacles_out[0].active) ? 4'd0 :
                            (!obstacles_out[1].active) ? 4'd1 :
                            (!obstacles_out[2].active) ? 4'd2 :
                            (!obstacles_out[3].active) ? 4'd3 :
                            (!obstacles_out[4].active) ? 4'd4 :
                            (!obstacles_out[5].active) ? 4'd5 :
                            (!obstacles_out[6].active) ? 4'd6 :
                            (!obstacles_out[7].active) ? 4'd7 :
                            (!obstacles_out[8].active) ? 4'd8 :
                            (!obstacles_out[9].active) ? 4'd9 : 4'hF;
```

## Game FSM module (Lili)


Player no longer jumping
Start (shows title screen)
Player jumps
Wait for player to fall back down
Wait for player to fall back down
Player no longer jumping
Playing (track and obstacle generation active)
Player dies and has 1 life left
Game over
Player jumps
Player dies and has > 1 life left

*Game State Machine Block Diagram*

This game state module received various inputs from different modules and maintained the main state of the game (start, playing, game over). It also maintained the time alive counter, the reset game pulse, and number of lives. The time alive counter was used by the obstacle generator to increase the difficulty of the game as the player progressed. The time alive counter was updated using a 100ms pulse output from the timer module. In our initial system design we didn't have a reset game pulse, just the whole system reset controlled by a button on the FPGA. We added this to signal to reset obstacles and other timers. The number of lives was used in the FSM to figure out when the game was over as well as to display at the top of the track.

When testing this integrated, we found there were some "debounce" issues when it came to the death and jumping signals. For example, at some point when a player jumped on game over, the jump signal would continue to register and it would go back to the playing state after very briefly displaying the start screen, thus requiring the need for the wait for fall states.

## Randomization module (Lili)

For this module we used a Linear Feedback Shift Register (LFSR) to generate pseudo-random bits to determine obstacle lane, obstacle sprite, and how long to wait until the next obstacle. An LFSR is fed back its last state and shifts as per a certain function. These functions are determined by a characteristic polynomial. For each desired bit length, there is a particular characteristic polynomial(s) which will result in the maximum FSR length, which we found from a Xilinx application note. Initially we implemented this for 4 bits. After testing this we noticed it was less "random" in terms of lane generation than we would have liked, as it would sometimes show patterns. So, we increased it to 8 bits and took an interval of it for each value we needed. The randomization module only reset upon system-wide reset, which made the obstacle characteristics still fairly random from game to game.

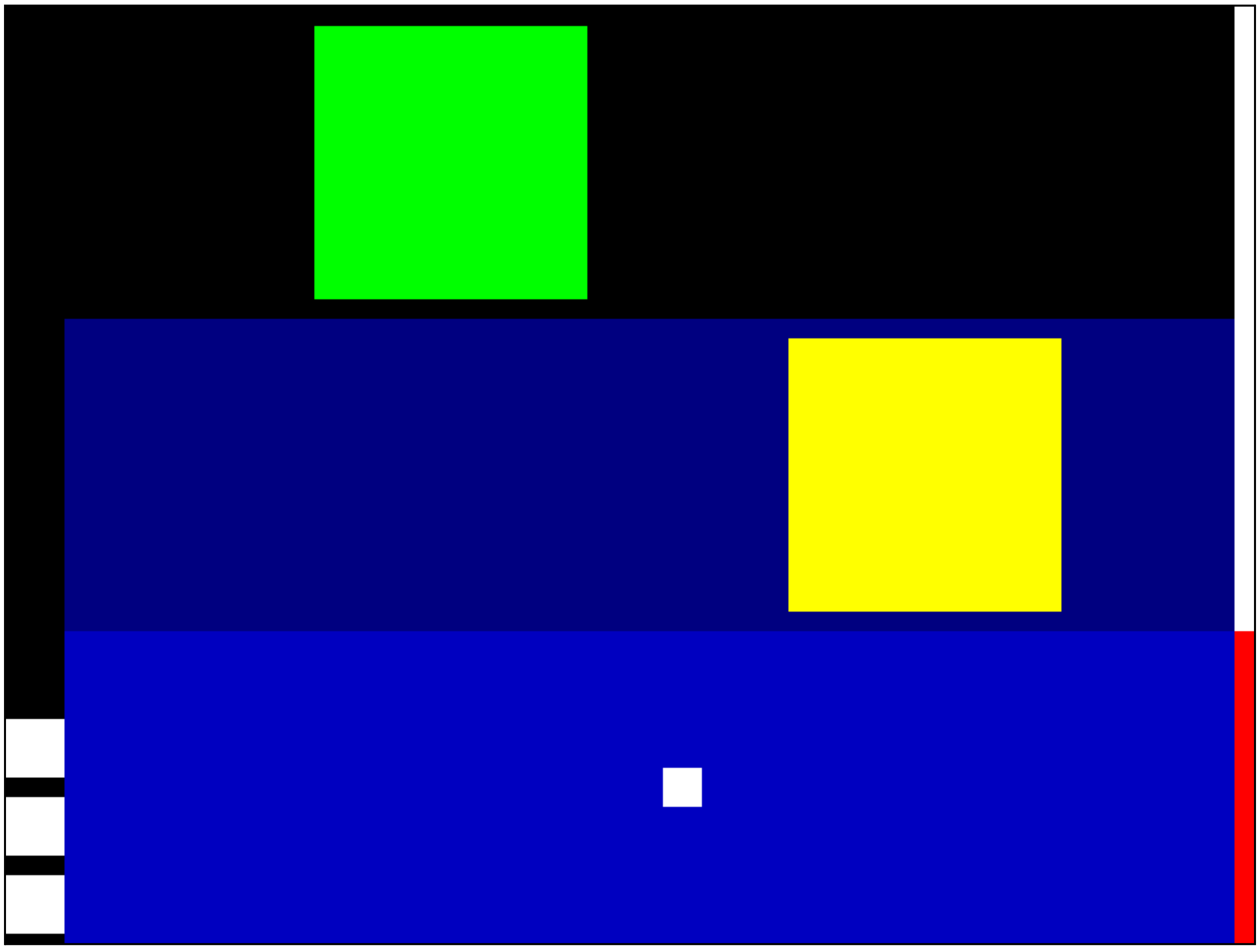
## Death module (Alex)

The death module compared the current state of the obstacles (from the Obstacle Generation module) with the current state of the player (from the vision pipeline, specifically the output of the Vision Debouncer). Based on this information, it signaled to the Game FSM when the player intersected with an obstacle.

Additionally, if the player intersected with a powerup (which was determined by the sprite_type field of the obstacle struct), then it instead signaled to the Game FSM that the player received a powerup and should therefore gain a life.

## Display

### Track Draw module (Alex)



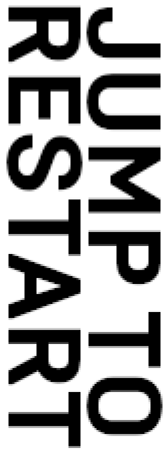
*A sample output frame from the Track Draw module, showing two obstacles and a powerup.*

Once the obstacles had been generated, they still needed to be displayed. The Track Draw module receives the obstacles array and generates the track imagery. It also reads the current lane and jump data from the vision pipeline, in order to display an indicator at the bottom, telling the player what lane they've been detected in and whether they are jumping or not. The number of lives, provided by the Game FSM module, is used to display an indicator at the top of the screen.

The module computes the value of the current pixel "just-in-time", based on the current VGA horizontal and vertical counts. This meant that there was a lot of combinational logic used to calculate what to display (what obstacle is active, which life should be shown in the current position, etc.)

### Start and Game Over Draw module (Lili)

The screen draw module displayed the title and game over screens. It used COE files in ROM to display the desired screen. An issue we ran into initially (which is described more below) is that we tried to display a whole 1024x768 full color image for the start screen, which got an error that it exceeded the memory limit. After this we scaled down the image significantly (potentially a bit conservatively) and displayed a simple gray scale image with a red background for the game over screen.



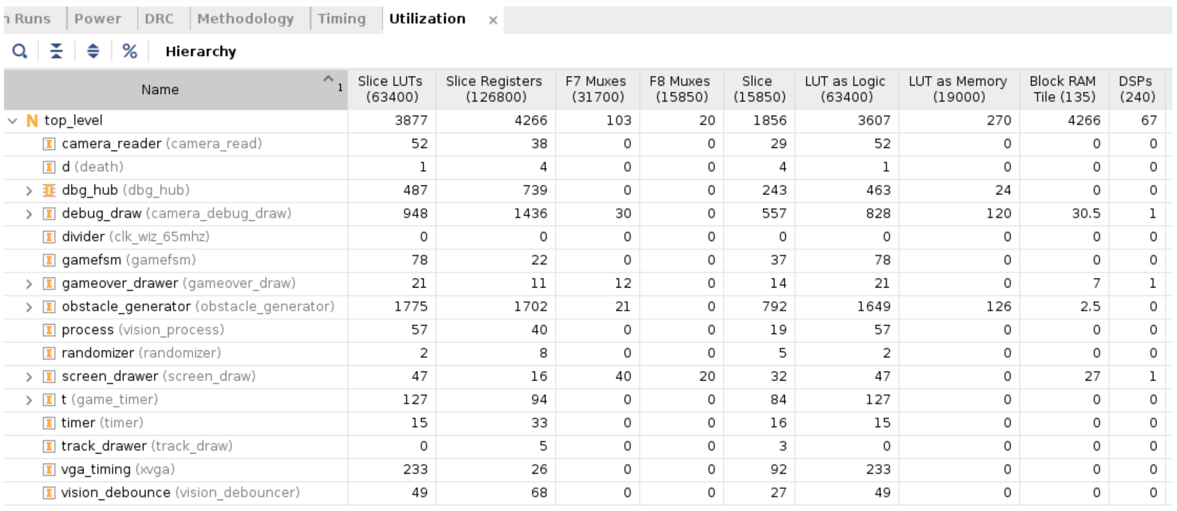
## Challenges

### BRAM utilization

One initial idea we had for the project was to have the obstacles be various different MIT and Infinite Corridor-themed images. This would require some way of storing the images, for which we wanted to use block RAM. However, we ran into issues where Vivado reported that we had run out of available block RAM resources on the FPGA. This was somewhat surprising, since this was not a feature we used extensively; however, due to time constraints, we were not able to investigate this fully.

In retrospect, it is likely that this error was a combination of two factors: first, the fact that we had two separate ILAs (used for debugging the camera and obstacle generation modules) enabled when we tried to add this feature. These ILAs had several probes (7 and 8 each) and a higher-than-normal sample depth, as we had just been finishing some rather debugging of both the vision pipeline and the obstacle generation. Second, we initially tried to create a very large image (for the background of the entire frame, 1024 x 768), which when we first were thrown off by the error.

In the following utilization report, you can see that with the two ILAs configured in the same way that we had them when trying to add the obstacle sprites, the BRAM utilization is rather high:



| Name | Slice LUTs (63400) | Slice Registers (126800) | F7 Muxes (31700) | F8 Muxes (15850) | Slice (15850) | LUT as Logic (63400) | LUT as Memory (19000) | Block RAM Tile (135) | DSPs (240) |
|---|---|---|---|---|---|---|---|---|---|
| top_level | 3877 | 4266 | 103 | 20 | 1856 | 3607 | 270 | 4266 | 67 |
| camera_reader (camera_read) | 52 | 38 | 0 | 0 | 29 | 52 | 0 | 0 | 0 |
| d (death) | 1 | 4 | 0 | 0 | 4 | 1 | 0 | 0 | 0 |
| dbg_hub (dbg_hub) | 487 | 739 | 0 | 0 | 243 | 463 | 24 | 0 | 0 |
| debug_draw (camera_debug_draw) | 948 | 1436 | 30 | 0 | 557 | 828 | 120 | 30.5 | 1 |
| divider (clk_wiz_65mhz) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| gamefsm (gamefsm) | 78 | 22 | 0 | 0 | 37 | 78 | 0 | 0 | 0 |
| gameover_drawer (gameover_draw) | 21 | 11 | 12 | 0 | 14 | 21 | 0 | 7 | 1 |
| obstacle_generator (obstacle_generator) | 1775 | 1702 | 21 | 0 | 792 | 1649 | 126 | 2.5 | 0 |
| process (vision_process) | 57 | 40 | 0 | 0 | 19 | 57 | 0 | 0 | 0 |
| randomizer (randomizer) | 2 | 8 | 0 | 0 | 5 | 2 | 0 | 0 | 0 |
| screen_drawer (screen_draw) | 47 | 16 | 40 | 20 | 32 | 47 | 0 | 27 | 1 |
| t (game_timer) | 127 | 94 | 0 | 0 | 84 | 127 | 0 | 0 | 0 |
| timer (timer) | 15 | 33 | 0 | 0 | 16 | 15 | 0 | 0 | 0 |
| track_drawer (track_draw) | 0 | 5 | 0 | 0 | 3 | 0 | 0 | 0 | 0 |
| vga_timing (xvga) | 233 | 26 | 0 | 0 | 92 | 233 | 0 | 0 | 0 |
| vision_debounce (vision_debouncer) | 49 | 68 | 0 | 0 | 27 | 49 | 0 | 0 | 0 |

It is unclear why Vivado claims that the total Block RAM Tile count is 4266; however, we can add the individual numbers (30.5 + 7 + 2.5 + 27) to get 67. (given that 67 is displayed one column over, it is possible that this is just a display bug in Vivado) This is around 50% of the FPGA's block RAM resources.

In retrospect, it makes sense that the ILAs consume block RAM resources (they are not magical and must have somewhere to store their samples); however, we did not realize this at the time. While this error was somewhat confusing, this issue was equal parts due to time limitations. Had we known about the utilization report, we could have checked it first, which would have made the issue immediately apparent. Alternatively, had we genuinely been out of BRAM, we could have investigated alternative approaches for storing/loading the images. We simply did not have enough time to pursue this path any further once we received the initial error.

## Vivado version compatibility

To collaborate on the project, we set up a shared GitHub repository, where we could store and track changes to the various files involved in the design. While this worked pretty well, we ran into issues in sharing IP blocks with each other. We tried to commit the XCI files that contained the configuration of each IP block, and while this did work locally, it caused errors when trying to use the IP on each others' computer. (specifically, we were told the IP was "locked" and that synthesis failed)

It was eventually determined that we were using slightly different versions of Vivado (2019.1 vs 2019.2), which had different versions of the IP blocks, causing this error. At the time, we did not know this, and so were able to work around the issue by manually sending screenshots of every IP block's configuration, a slow and tedious process that led to some confusion due to some settings mistakenly not being replicated exactly between our two computers. It would have been best to validate this aspect of the project setup earlier on, as we only discovered this issue later on in the project's timeline, when integrating our modules.

## Lessons learned

One aspect of the project that went particularly well was how our initial design was pretty well defined, down to specific formats and number of bits of data like the obstacles array. This meant that we were able to work on our modules in parallel and did not face many challenges when integrating them. However, we did encounter an issue where some modules could not be effectively synthesized and tested in hardware due to some other dependencies. For example, the Obstacle Generation module could not be verified until the Track Draw module worked. While we did test the Obstacle Generation module in simulation, there were still some issues that needed to be resolved once it was tested on a real FPGA. We could have resolved this by prioritizing the development of a very simplistic version of the Track Draw module. This could then be used for validation of obstacle generation while the details of track draw were worked out.

Another lesson learned was that we could have prioritized the hardware setup more, as it affected the playability of the game. The playability overall was functional enough but definitely could be better. We didn't get the projector set up until after Thanksgiving, and it was initially a bit tricky to figure out how to mount it. We did not realize how high it actually needed to be mounted to work well with a human player. That is, we wanted the width of the projected track to be at least three human widths wide so it would be more intuitive for the player to move lanes.

Additionally, we could have messed around more with how lighting would affect both the camera thresholding as well as the visibility of the projected image. For example, sometimes the thresholding worked less well when the player was wearing lighter colored pants. We were projecting at the floor, which had a pattern and was reflective which made some colors harder to see (we ultimately put a piece of white poster paper on the floor).

Finally, our alignment of the green screen, projector, and camera were not completely optimal. Not only did the green screen need to fill the field of view of the camera, but also both it and the camera needed to be centered and in line with the projector. In our setup, for one of the edge lanes, we needed to be partially off the projected track in order to be in the correct part of the green screen. Another surprise was just how much we needed to jump with the camera placement. When we first integrated, we noticed the player essentially needed to: a) jump quite high and b) cannonball their body for their outline to be above the bottom third. We fixed this by both lowering the threshold for the bottom third of the screen as well as mounting the camera lower. This made the jump far more reasonable but still not a natural, stationary hop. We can however fix this issue by marketing the game as a home fitness apparatus.

## Future improvements

A lot of future improvements stem from challenges and lessons learned as described above.

### Hardware Mounting

As described above, we had some issues with the mounting, and it was definitely a bit hacky. Intuitive play wasn't our highest priority during the project (and we were the only ones testing it) so fixing all the alignment issues would make it easier for others to play. Additionally we could move our set up to where the emergency lights aren't, and add lighting just for the player and screen. This could help with both projection and camera thresholding.

### Sprites and Other Imagery

Adding in more sprites that are images could make game play more fun. The life power ups that were displayed were rectangles, but those could be changed to hearts again to make it a bit more intuitive to someone who doesn't know how the game works. Additionally, having the obstacles be certain things known to MIT students could make it more engaging. In order to implement this we would probably also need to use an SD card.

### Power Ups

Our only power up currently is adding extra lives. In the future we could be more thoughtful about how often to let this power up come onto the screen, taking into account how many lives the player currently has. Additionally we could have some other cool power ups (or downs) including distortion, increasing speed, timed invincibility, and more to make the game more challenging and interesting to play.

### Sound

One of our stretch goals was adding sound to the game, including a background track as well as sound effects during parts of the game like when colliding with an obstacle, jumping, and power ups.

## Link to code

https://github.com/thatoddmailbox/infinite-run

We have separated the code into the following folders:

- coe -- contains various coe files used for images in the game
- hdl -- contains the main SystemVerilog for the system
- ip -- contains xci files for Vivado IP
- ov7670_control -- contains code for camera reset program, on ESP8266
- sim -- contains testbenches
- xdc -- contains constraints file