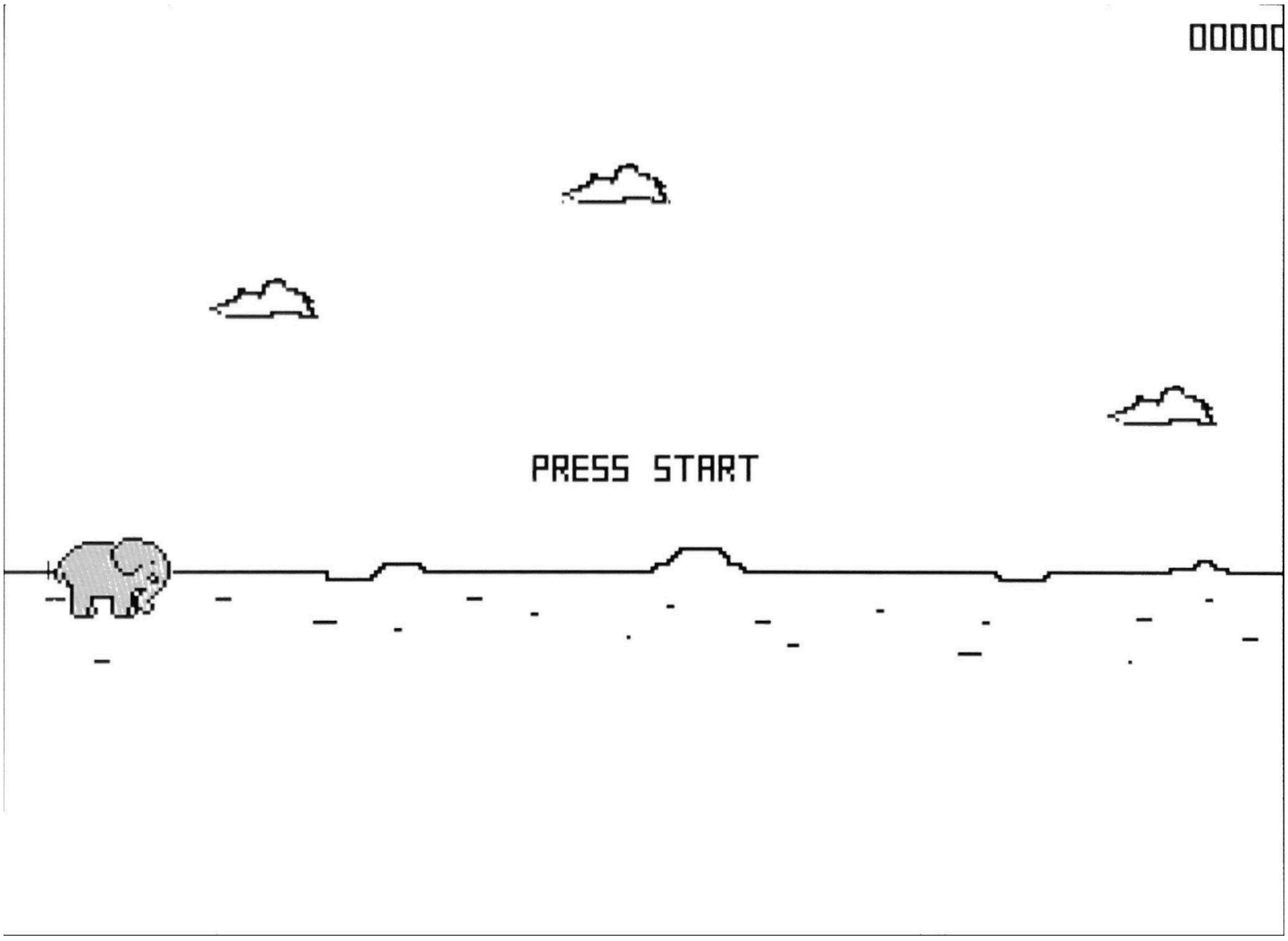
Jumbo Dash

Isaac Medina, Jan Konings, Jacob Gerson, Ryan Chen
Github

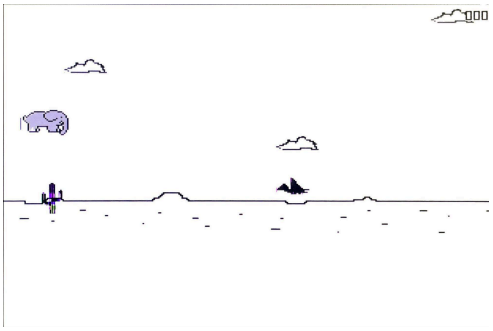


A picture of the Jumbo Dash game start screen

Overview

Jumbo Dash is a side-scrolling endless runner game based on *Google Chrome's Dino Accelerator Game* and *Geometry Dash.* Play as Jumbo, the beloved Tufts mascot, running, jumping, and ducking past the obstacles. The player uses the NES controller connected to the FPGA to interact with the game, which is output over a VGA display. Jumbo Dash has three high-level states: Start game, playing, and game over states–each with its own screen. The game begins in the start game state, displaying the start screen. While in the playing state the user can press A to jump, or B to duck on the NES controller to dodge the oncoming obstacles. Jumbo Dash simulates accurate jumping, running, and ducking animations/physics, accurate hitbox, and collision detection, flying pterodactyl animations, and slow-scrolling clouds in the background for depth. The game speeds up as the score increases, upping the difficulty level as the game progresses. The goal is to get a high score. If Jumbo hits an obstacle, it's game over. In the game-over state, the user can press the start button to play again.

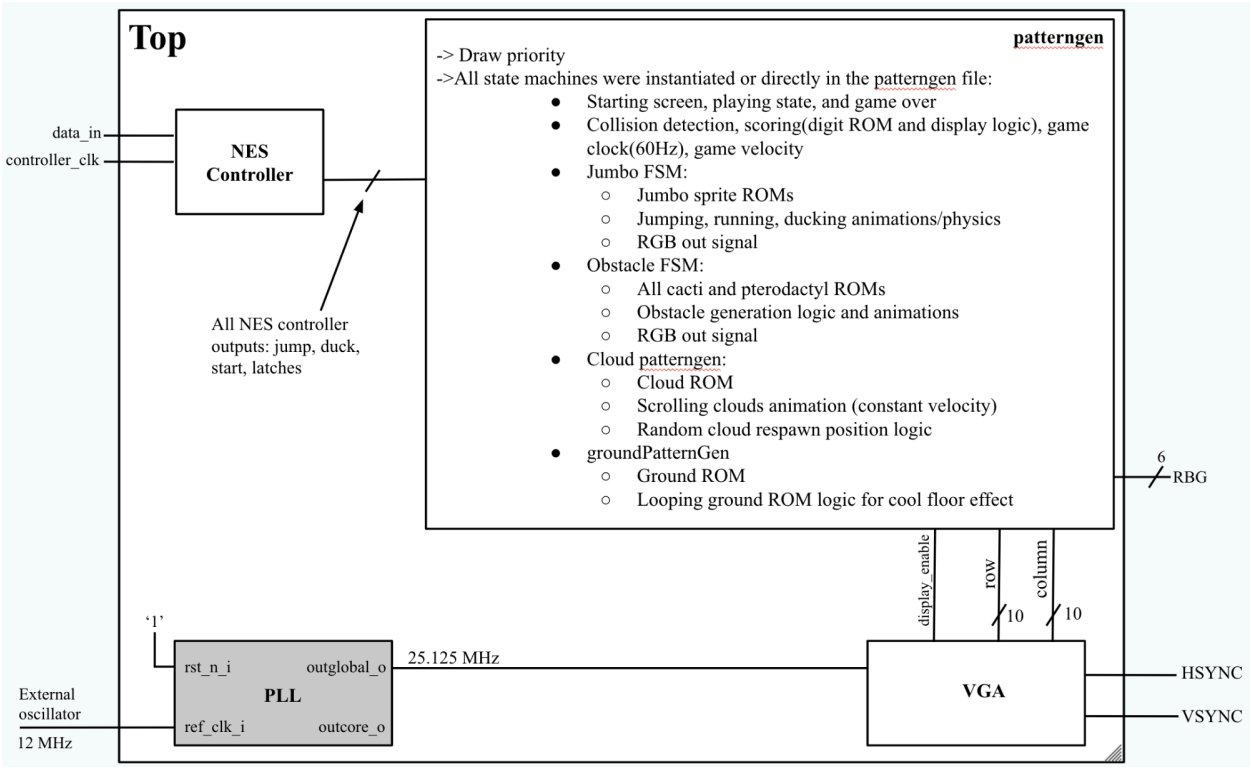
Start Screen



A playing frame

Game Over Screen

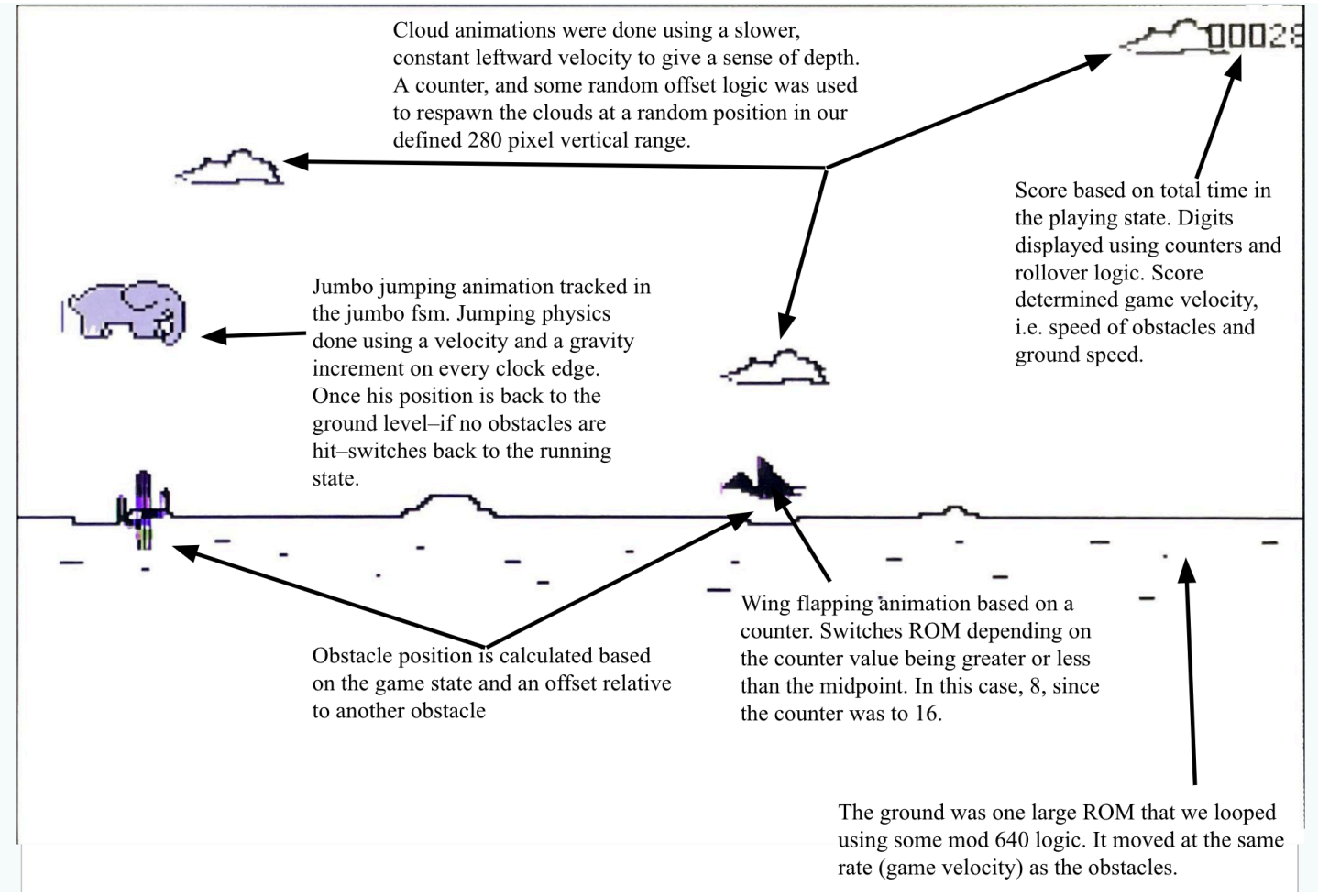
Pictures of each game state



A high-level block diagram of Jumbo Dash

Our game includes PLL, VGA, patterngen, and the NES controller input modules. The patterngen module does the bulk of the work. At a high level, the game state machine inside patterngen keeps track of user input and collision detection to decide the next game state and what to display. Patterngen uses priority logic and separate state machine modules for Jumbo, the obstacles, the floor, and the background to determine what exactly to display to the player. Each submodule inside of patterngen also receives the user inputs and current game state to decide what states the different aspects of the game are in.

Watch some videos of Jumbo Dash in Action



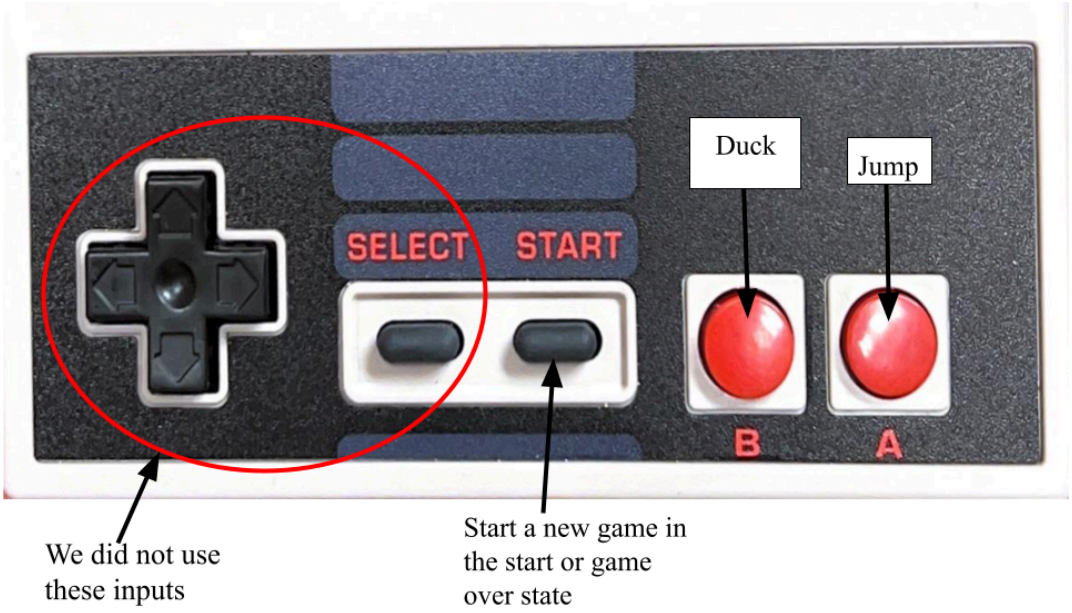
An annotated screenshot of Jumbo Dash outlining the general game logic

---

Technical Description

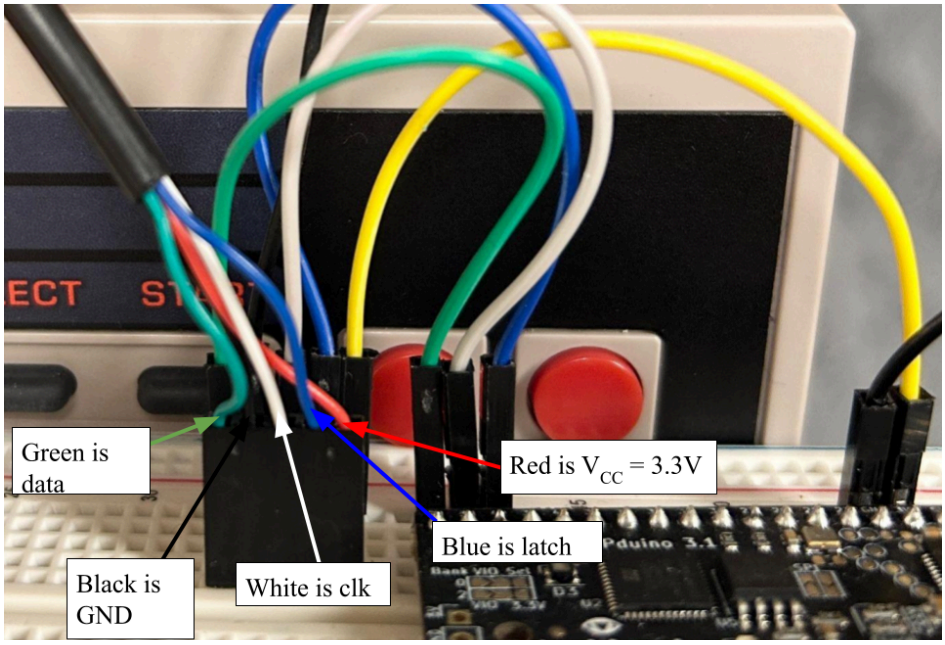
NES Controller:

The NES Controller was Jumbo Dash's central user interface. They have eight possible buttons that can be pressed (seen in the picture below), each of which has a corresponding bit related to them in the controller's 8-bit output. The controller takes only a single input bit from the controller–the input data bit is transmitted over the blue color wire. The controller clock–made using internal HSOSC hardware(around 83 kHz) orchestrates when the button reading process occurs (through the white wire), and the controller then transmits the eight bits of controller data when the latch (connected through the blue wire) is asserted high every 60 Hz or so. These eight bits reacted intuitively to the controller inputs - each of the eight bits was either high or low



Annotated NES controller interface for Jumbo Dash

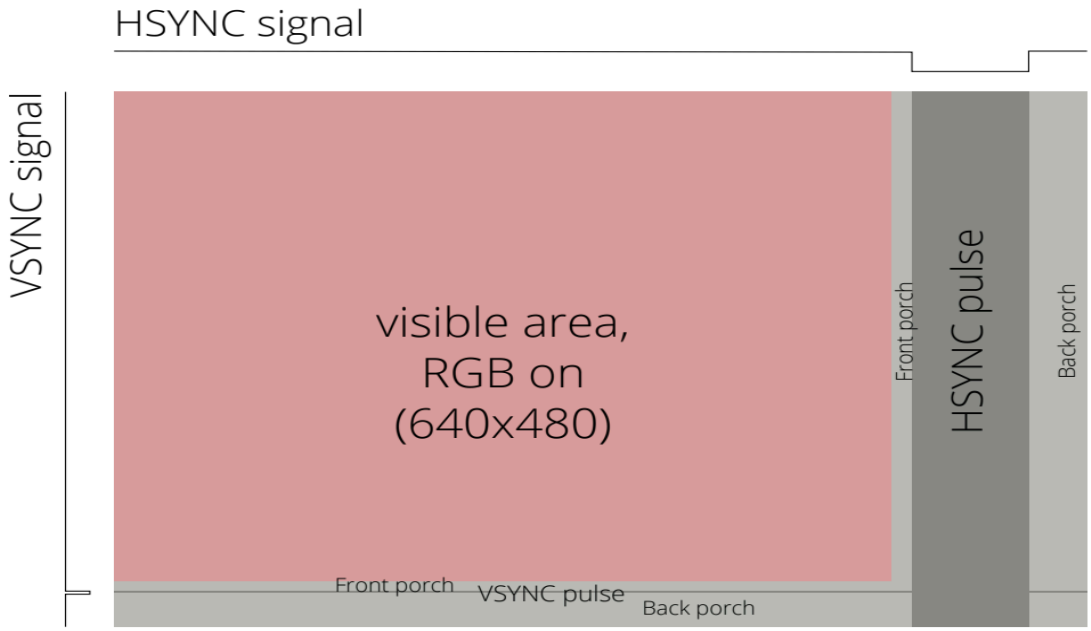
The game was solely focused on 3 of the 8-bits the controller outputted. Jumbo Dash uses the 'A' and 'B' buttons to jump and duck, respectively, and the user moves through the Start and Game Over states with the 'Start' button. The bits that mapped onto the desired buttons were isolated, and those single-standing logic bits drove the in-game action across the project!



Annotated NES controller pin-mapping

VGA:

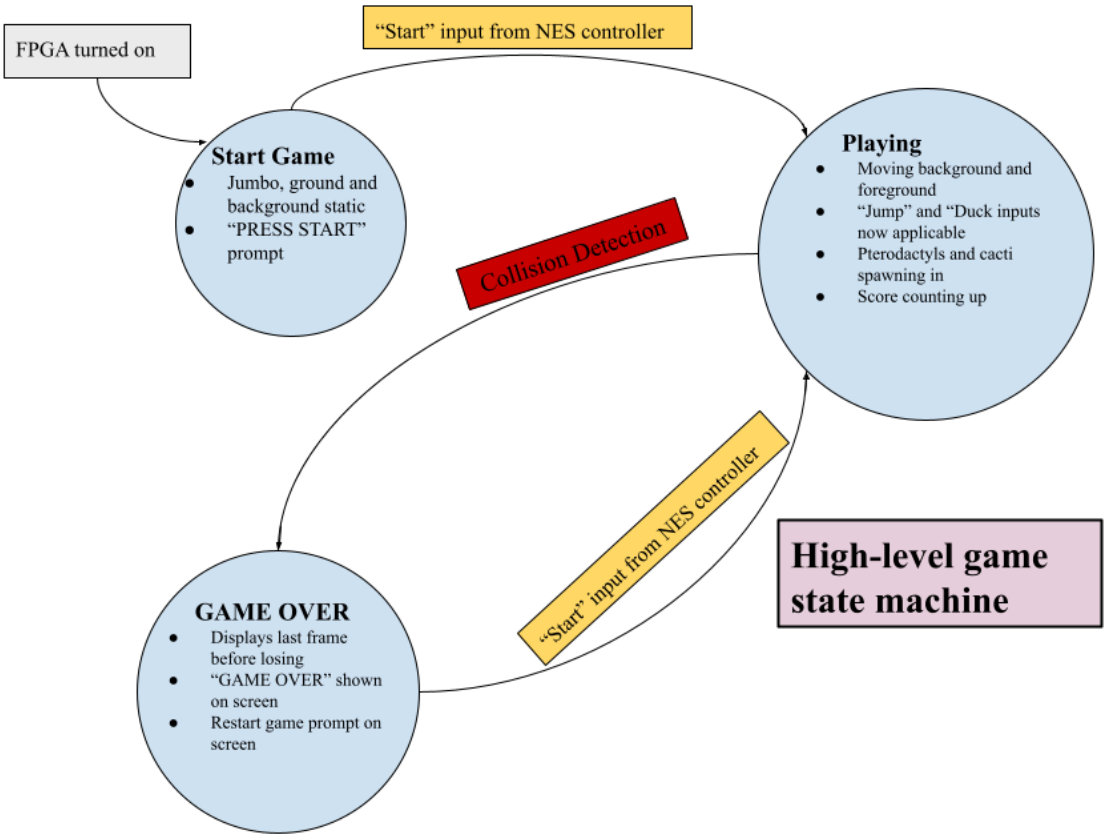
Using the 25.125MHz clock signal from the PLL, The VGA module constantly updates and outputs the 10-bit unsigned row and column coordinate of the pixel being updated by the module. (0, 0) is the top left left corner, and the column (X) axis increases to the right, and the row (Y) axis increases downward. These values represent the game's coordinate system, and they interact with almost every module in the game. It also outputs the HSYNC and VSYNC signals, which tell the VGA monitor when to draw a new line or start back at the top. The VGA requires time between rows and frames to synchronize. The vertical refresh rate is 31.46kHz and the screen refresh rate is 60Hz. Although the VGA monitor can only display 640x480 pixel images, the row and column values go up to 525 and 799 respectively. The display enable signal is high when the row and column values are within the 640x480 drawing grid.



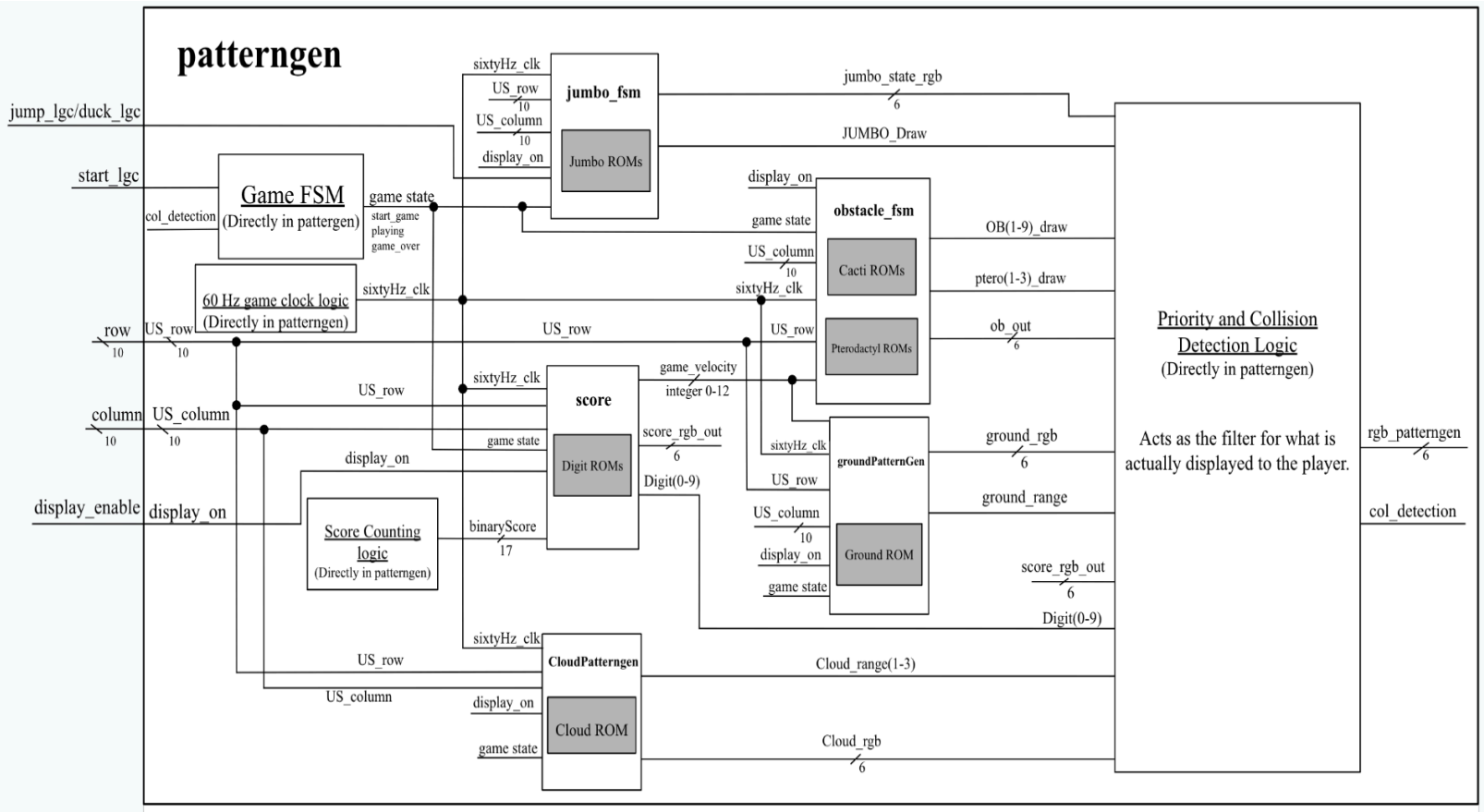
A visual/geometric representation of what the VGA is doing | Credit: *"ES4 Lab 7: driving a VGA monitor"*(Source)

Graphics and patterngen:

➔ patterngen contained the high-level game state machine(the GAME FSM logic in the block diagram):



➔ Block diagram of patterngen:



Using the NES controller inputs, the display priority logic, and collision detection logic, the state machines in patterngen control which graphics are outputted by the submodules. The graphics in Jumbo Dash are essential for providing a visual context for the gameplay. We hand-drew all of the images (Our sprites can be seen in Table 1, page 11) including character sprites, obstacles, and background elements, to represent the game's environment and actions. These images were then converted into ROM files for use on the FPGA, allowing the game's visuals to be rendered on the VGA display. To optimize memory usage, the graphics were scaled down, reducing the size of each image while retaining key visual detail; this is explained more in-depth later under "Accessing ROM files". Our game logic then uses these ROMs to draw our sprites where and when we want to.

➔ Sprites/ROM files:

Jumbo Dash relied heavily on using ROM files. We chose to use ROM's for these key reasons:

1. **Non-Volatile Storage:** The ROM retains its data even if the system is powered off, which makes it perfect for storing static assets such as sprites, background images, game state screens, and other things that do not change throughout gameplay.
2. **Efficiency:** We used scaled-down images, which makes using ROMs very space—and time-efficient. ROMs allow us to store large amounts of data compactly, which is essential given the FPGA's limited storage capacity. ROMs also provide fast data retrieval, ensuring that the graphics required by Jumbo Dash can be accessed quickly without delay, which is crucial for maintaining smooth gameplay.

➔ Creating PNGs for ROM Files:

The website piskelapp.com created the sprites for every object displayed in the game. The scaled sprite was drawn in piskelapp pixel by pixel, finding the smallest box it could fit in. The dimensions of this box are the dimensions of the scaled sprite. These dimensions were used to create "drawing ranges" for each sprite.

Every sprite created is drawn in the top left corner of the screen (0, 0). This means that to draw the sprite in different positions on the VGA display, the starting coordinates where the sprite is supposed to be drawn were scaled to make the starting coordinate (0, 0) because our ROM files draw the sprites beginning at (0, 0).

These images were then run through the Python ROM script (described later) to generate data for the ROM file, which is the "when" lines that tell the VGA what color each pixel should be.

```
process(addy) begin
        case addy is
                when "0000000100" => data <= "000000";
                when "0000100011" => data <= "000000";
                when "0000100100" => data <= "000000";
                when "0000100101" => data <= "000000";
                when "0001000011" => data <= "000000";
                when "0001000100" => data <= "000000";
```

Example ROM file code

➔ Creating ROM Files:

All of the ROMs in the project share a unified structure regardless of the specific content of the image.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;


entity CloudROM is
    port(
        Xin : in unsigned(4 downto 0);
        Yin : in unsigned(4 downto 0);
        data : out std_logic_vector(5 downto 0)
    );
end entity;


architecture synth of CloudROM is

signal address : unsigned(9 downto 0);

begin

address <= Yin & Xin;


process(address) begin
    case address is


        when others => data <= "111111";
    end case;
end process;
end;
```

Xin (X-coordinate): A 5-bit unsigned input that specifies the scaled horizontal position of the pixel to be accessed.

Yin (Y-coordinate): A 5-bit unsigned input that specifies the scaled vertical position of the pixel to be accessed.

Data: A 6-bit output that represents the pixel color value. This value can encode specific colors or grayscale shades, depending on the sprite's requirements.

The ROM uses a concatenation of Yin and Xin to compute a 10-bit address (address) that uniquely identifies each pixel in the sprite. The concatenation ensures a straightforward mapping of 2D sprite coordinates to the linear memory addresses in the ROM.

The process and case statements follow the vhdl structure to instantiate a ROM. We then populate the space between the case statement and the when others case with our when " " statements from our python script, which is image specific and discussed below.

The game state background color is white, so the default "when others" statement in all of the ROMs is "111111" or "white".

➔ Python script to generate ROM content:

This Python script extracts pixel RGB data from the PNG images we created using piskelAPP and formats it for use in ROMs. The FPGA has a 2-bit color depth, so the script also modifies the RGB data to fit into a 2-bit representation, making the RGB values a total of 6 bits. The script encodes the coordinates of each pixel in binary and prints only black pixels. This approach ensures that the game sprites are accurately represented on the display while adhering to the FPGA's ROM storage constraints.

```python
from PIL import Image

# Correct image path using raw string or forward slashes
image_path = 'FILEPATH'

# Load the image
image = Image.open(image_path)

# Get the image size (width, height)
width, height = image.size

# Function to convert RGB value to 2-bit per channel (scaling from 0-255 to 0-3)
def rgb_to_2bit(rgb):
    return tuple(min(3, round(c / 85)) for c in rgb)

# Create and print the formatted output
for y in range(height):
    for x in range(width):
        # Get RGB value at (x, y)
        rgb = image.getpixel((x, y))

        # Check if the pixel is black
        # Convert coordinates to 5 binary strings
        y_binary = f"{y:05b}"  # 10-bit binary for y-coordinate
        x_binary = f"{x:05b}"  # 10-bit binary for x-coordinate

        # Combine them to create a 20-bit binary string
        coords_bin = y_binary + x_binary

        # Convert the RGB value to a 2-bit per channel representation
        rgb_2bit = rgb_to_2bit(rgb)

        # Combine RGB values into a single 6-bit value
        rgb_combined = f"{rgb_2bit[0]:02b}{rgb_2bit[1]:02b}{rgb_2bit[2]:02b}"
        formatted_output = f'when "{coords_bin}" => data <= "{rgb_combined}";'

        # Create the formatted output string and only add the black pixels
        if rgb_combined == "000000":
            print(formatted_output) # Print the formatted output
```

➔ Accessing ROM Files:

The ROM modules in this project take in an address and output an RGB color, which corresponds to a pixel on the VGA display. To optimize ROM storage within the FPGA's 120k-bit limit, the ROM images were scaled by a factor of 2 in both the X and Y directions, resulting in a resolution of (640 / 4) x (480 / 4) = 160 x 120. This approach minimized memory usage while still preserving the sprites' detail.

To display these sprites on the 640 x 480 VGA screen, the VGA coordinates were scaled down before being fed into the ROM. Once we reach the VGA coordinates of where we want to start drawing the ROM (top left corner reference point) we start feeding the coordinates into our ROM to get the corresponding RGB value. For each pixel's coordinates, the relative position was calculated by subtracting the top-left corner of the sprite's intended location (Ref_X, Ref_Y) from the VGA pixel coordinates (VGA_X, VGA_Y), essentially making $VGA_X = 0$ and $VGA_Y = 0$:

$$X_{relative} = VGA_X - Ref_X$$
$$Y_{relative} = VGA_Y - Ref_Y$$

The relative coordinates were then scaled down by a factor of 2 to match the ROM's resolution. This scaling was implemented in hardware by dropping the least significant bit (LSB) of each coordinate, meaning the four pixels on the VGA display relative to the single pixel in the ROM will all get the same RGB value.

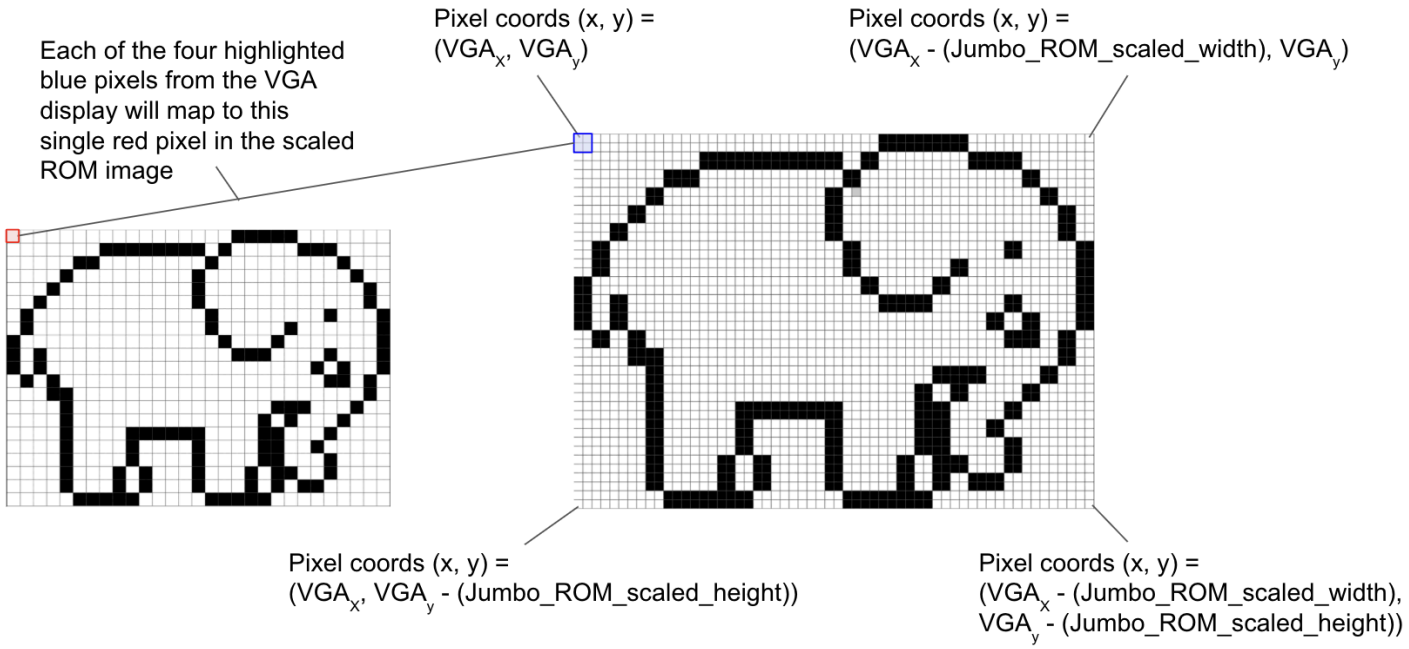


Diagram depicting the pixel correlation between scaled images and VGA display coordinates for ROMs

All of the ROM sprites could be made with dimensions less than 32 x 32 pixels, meaning the smallest number of bits to store in any X or Y coordinate was 5 ($2^5 - 1 = 31$). The VGA display addresses are 10 - 1(scaling) = 9 bits each, so to match the ROM coordinate lengths, the bottom five bits after the scale (LSB) bit are taken off to create the ROM address, as they won't ever be greater than 32 after finding $X_{relative}$ and $Y_{relative}$.

$$X_{in} = X_{relative}[5:1]$$
$$Y_{in} = y_{relative}[5:1]$$

X_{in} and Y_{in} are then the coordinates we send to our ROMs. By actively scaling the VGA coordinates, the system maintained efficient mapping between the high-resolution display and the lower-resolution storage-efficient sprite data stored in ROM. A visual example of the scaling can be seen below along with a table showing our sprites and their scaled dimensions used to calculate reference coordinates.

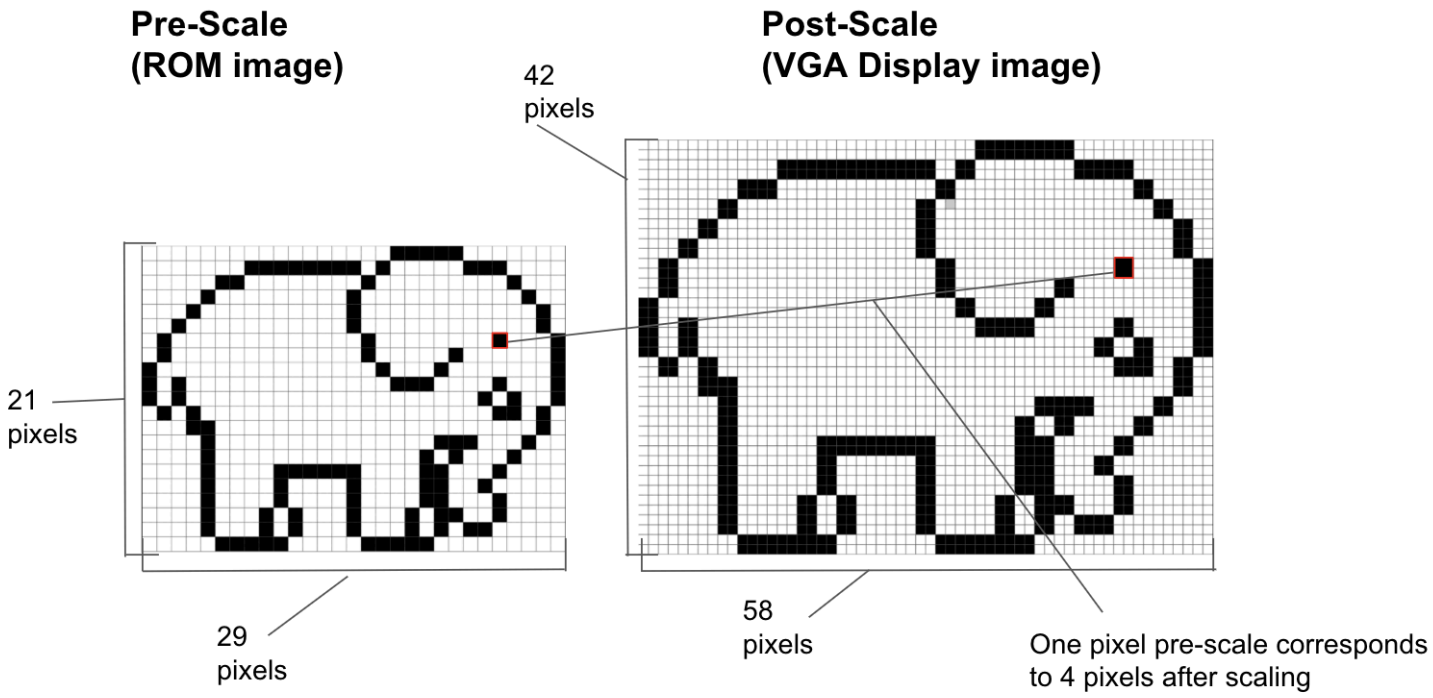


Diagram representing the scaled dimensions of the ROM images

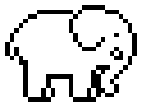
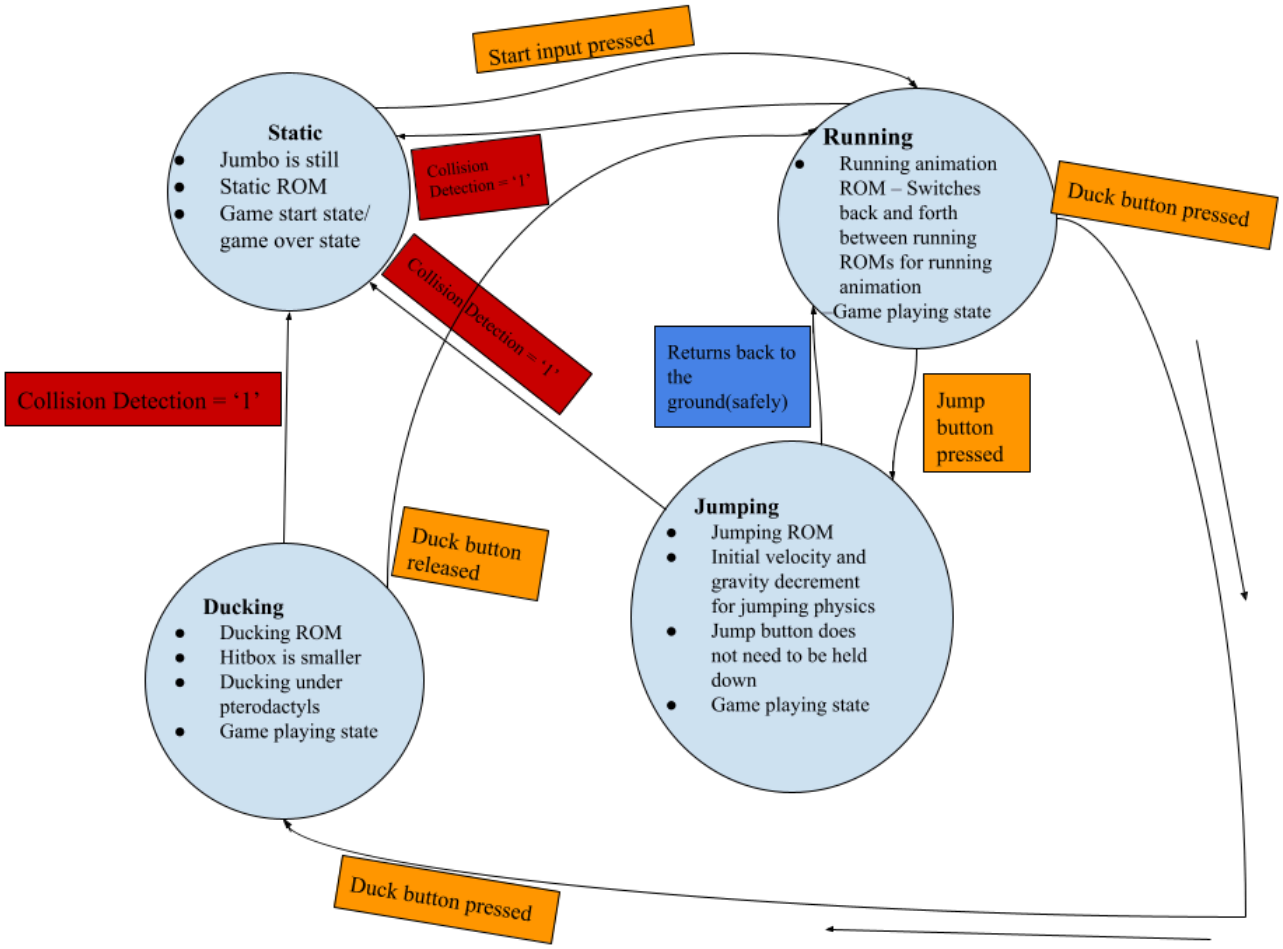
PNG	Sprite	Pre-scale	Post-scale
	Jumbo	29 x 21 pixels	58 x 42 pixels
	Pterodactyl	21 x 13 pixels	42 x 26 pixels
	Cloud	27 x 11 pixels	54 x 22 pixels
	Single Cactus	14 x 23 pixels	28 x 46 pixels
	Small Single Cactus	6 x 13 pixels	12 x 26 pixels
	Double Cactus	24 x 23 pixels	48 x 46 pixels
	Triple Cactus	31 x 23 pixels	62 x 46 pixels

Table 1: Displaying some game sprites and their pre and post-scale dimensions

➔ Animations with ROM:

In many scenarios, we had to simulate animations, for example, the flapping of the pterodactyl's wings, the running motion of the Jumbo, the clouds, and the ground.

Jumbo FSM:



FSM Model for Jumbo sprite

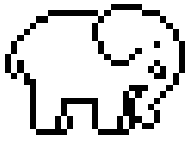
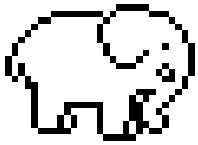
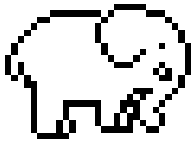
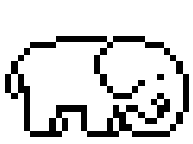
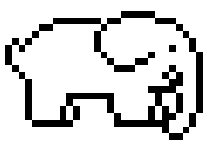
				
Static Jumbo	Back feet up Jumbo	Front feet up Jumbo	Crouching Jumbo	Jumping Jumbo

Table 2: Jumbo animation sprites

➔ Jumbo Running Animations:

Two different ROMs implement Jumbo's running animation. One ROM displays the Jumbo with its front leg up, while the other displays the Jumbo with its back leg up (seen in Table 2). When pressing the start button on the NES controller, the VGA display continuously switches from pulling from the first Jumbo running ROM to the second one, creating the Jumbo's running animation. The swapping is implemented by incrementing a counter on every rising edge

of the 60Hz clock. Once the counter reaches 16, it is reset to 0. If this counter is greater than 8, then the VGA displays the first Jumbo running ROM. If it is less than 8, it displays the second Jumbo running ROM.

For clarity: Jumbo is horizontally static in every state of Jumbo Dash. His running animation works as the rest of the game around him (the background and obstacles) moves quickly by. This gives the user an illusion of racing through an expansive desert, something that the running animation lends itself nicely to.

➔ Jumbo Crouching and Jumping Animations:

For the jumping and crouching animations, we created two different ROMs: a crouching Jumbo and a jumping Jumbo. When the crouch button on the NES controller (B) is pressed, the VGA display switches from pulling from the Jumbo running ROMs to pulling from the crouching ROM. Once the crouch button is released, the VGA display pulls from the Jumbo running ROMs again. Similarly, when the jump button on the NES controller (A) is pressed, the VGA display switches from pulling from the running animation ROMs to pulling from the jumping ROM. Once the Jumbo reaches the ground, the VGA display begins pulling from the Jumbo running ROMs again.

➔ Jumping Animation Physics:

To simulate jumping physics, we used an initial velocity and a gravity decrement. In the Chrome Game, the dino sprite is in the air for exactly 0.5 seconds. Since we are using a 60Hz game clock, this timing corresponds to 30 clock edges. If Jumbo is in the air for a total of 30 clock edges, this means that he will be going up and down for 15 clock edges each. Therefore, the ratio between Jumbo's initial velocity and the gravity decrement on every clock edge must be 15:1. We picked the lowest ratio and Jumbo's initial jumping velocity was 15 pixels per clock edge, and the gravity value was -1 pixels per clock$edge^2$. Using this logic, Jumbo will rise and fall 120 pixels in 0.5 seconds:

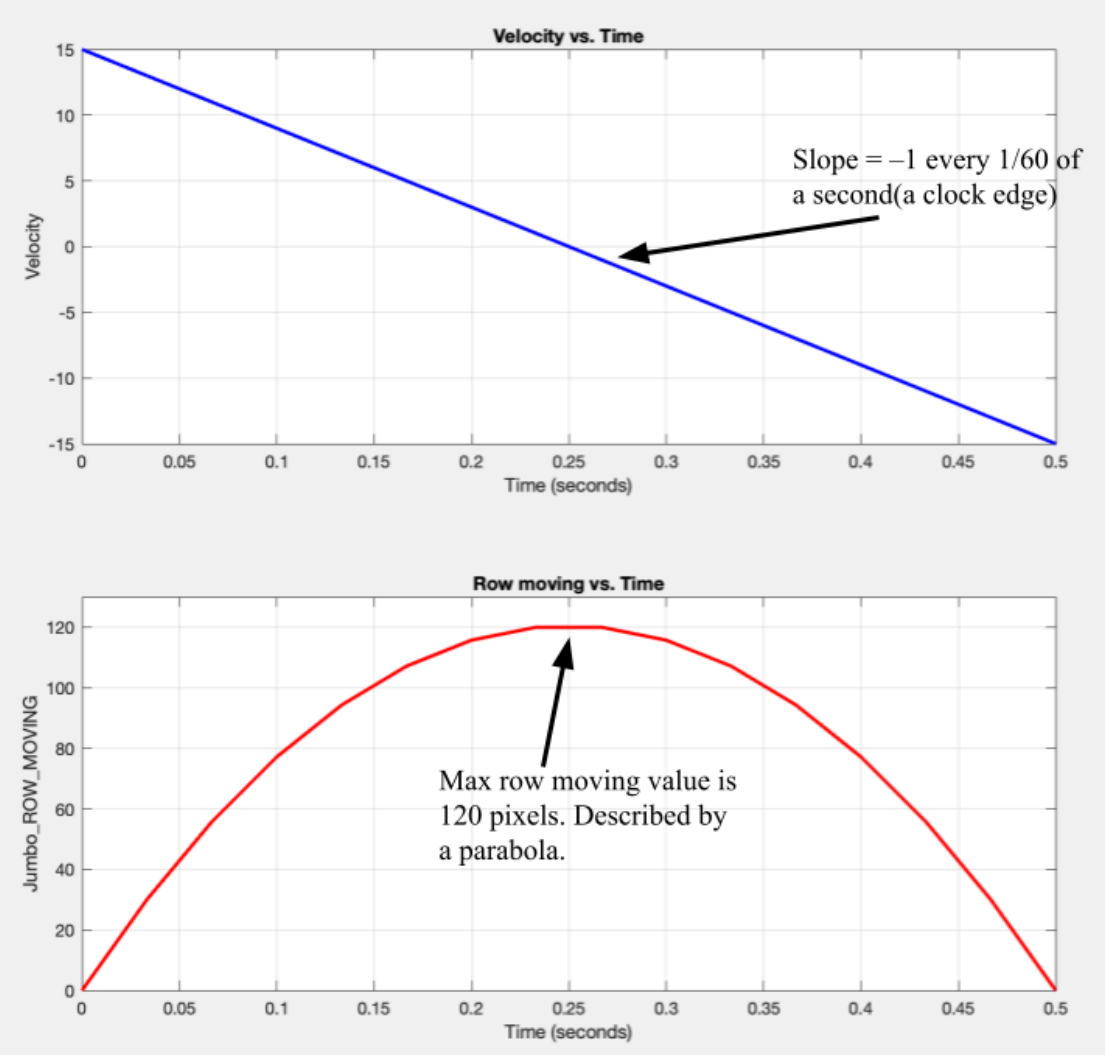
$$15 + 14 + 13 + 12 + . . . + 1 = 120$$

When Jumbo gets to the top of the jump animation, the direction changes and he accelerates back to the ground. We defined what the top and bottom of Jumbo's draw box were while running, and connected those to a changing row signal when in the jumping state.

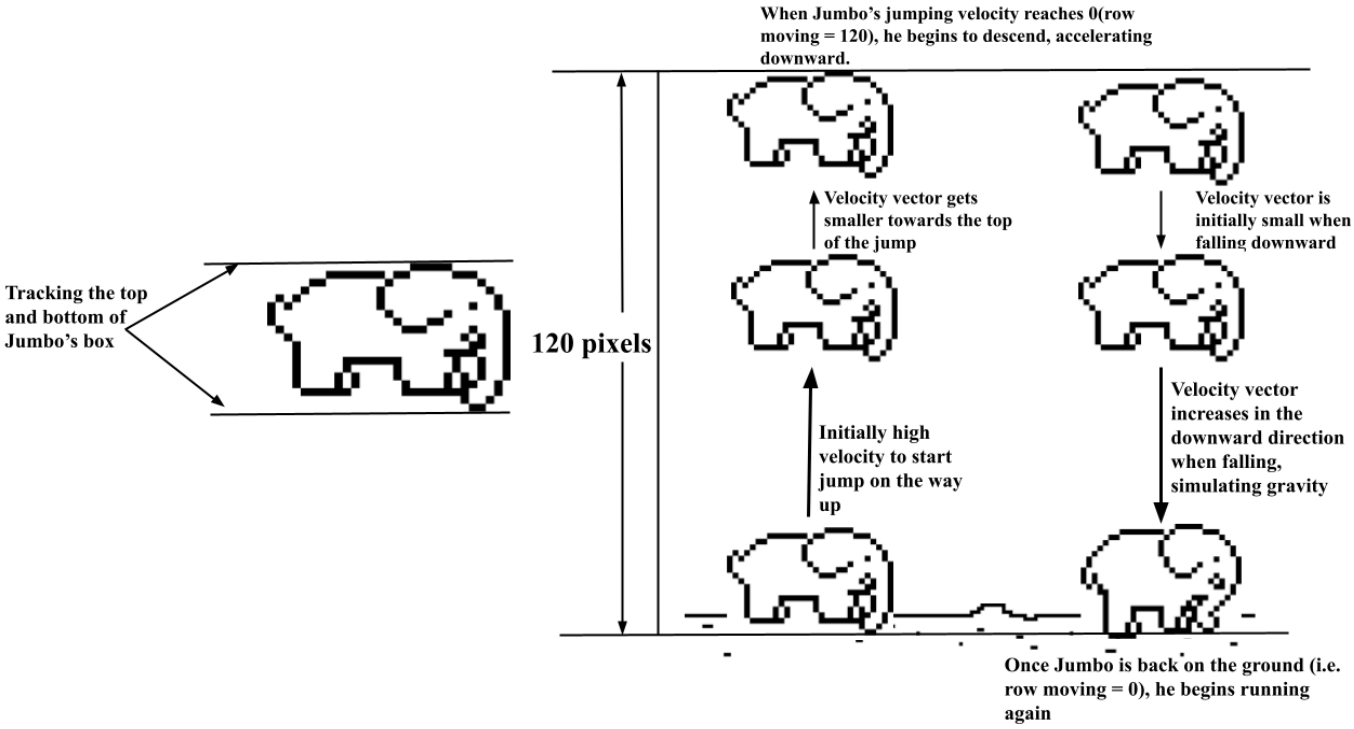
```
signal jumbo_ROW_MOVING : integer range 0 to 120 := 1;
constant JUMBO_ROW_START_TOP : integer := 275;
constant JUMBO_ROW_START_BOTTOM : integer := 322;

velocity <= velocity + gravity;
jumbo_ROW_MOVING <= jumbo_ROW_MOVING - velocity;

jumbo_Row_draw <= Jumbo_Row_in - 20 + jumbo_ROW_MOVING;
```



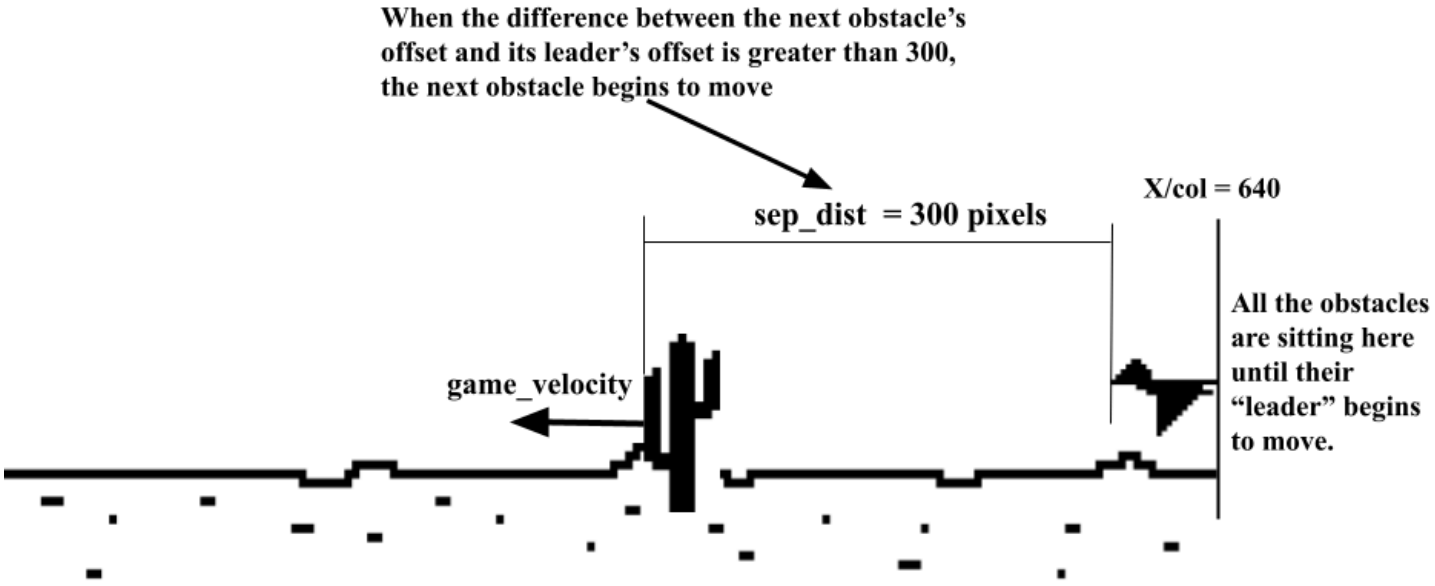
Graphs showing velocity and row moving vs. time in seconds



Visual description of the jumping animation

Obstacle Generation:

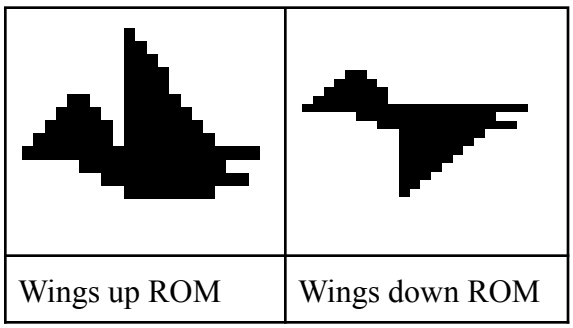
Obstacle generation was done using conveyor belt logic and an offset. To make the game hard, but not too hard, there is a constant 300-pixel separation distance between each obstacle. We defined ten main obstacles and three pterodactyls. All these obstacles moved relative to another object and formed a loop by connecting the last object to the first object.



Obstacle generation logic visually

➔ Pterodactyl Flying Animation:

To simulate a flying pterodactyl animation, we used the same logic as Jumbo's running animation. Two different pterodactyl ROMs were created, one with its wings up and one with its wings down. The VGA display continuously swapped from pulling from the wings up ROM to the wings down ROM, which created the flapping wings animation. If the counter is greater than 8, then the VGA displays the wings down ROM. If it is less than 8, it displays the wings up ROM.



Pterodactyl ROMs

Ground and Background:

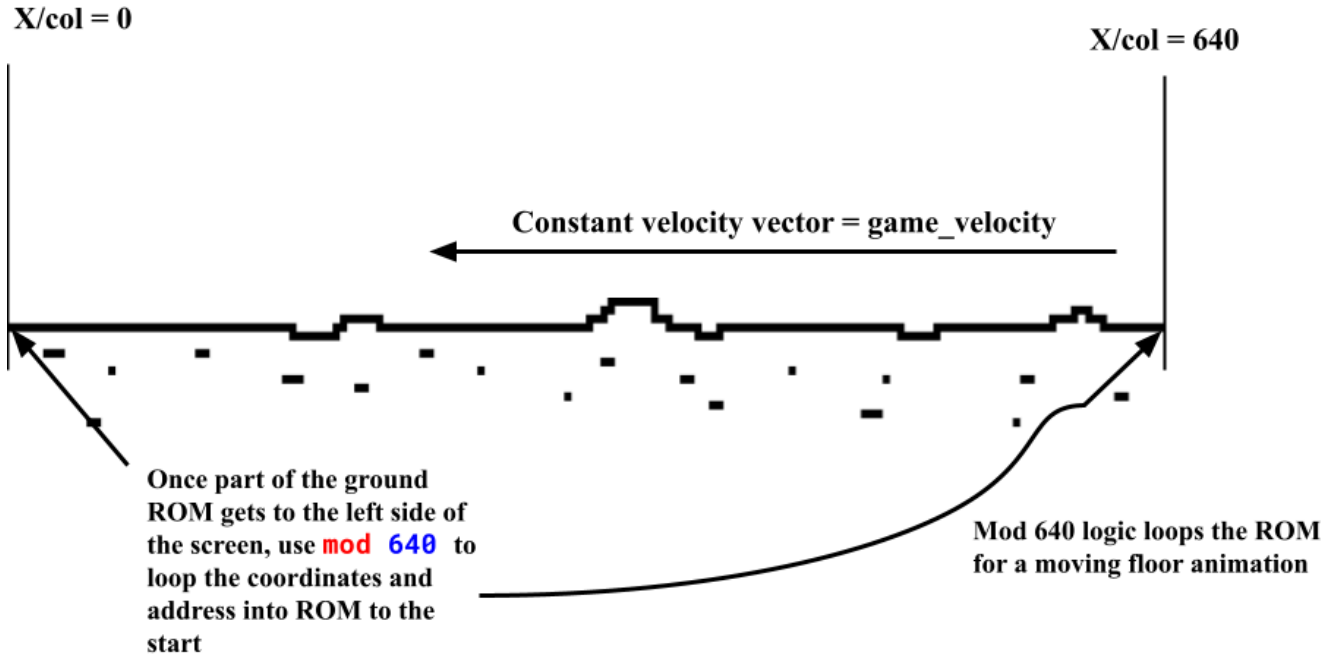
➔ Cloud Animations

The cloud animation uses a constant velocity of 1 pixel per clock edge and an offset for each cloud. It subtracts the cloud ROMs column by this constant velocity to display the clouds moving from right to left on the VGA display.

When the cloud ROMs reach the end of the display, they restart and begin displaying on the right side of the VGA display. To display the clouds at different vertical position on the right side of the screen, a counter increments by one on every rising edge of the 60-hertz clock. Once the counter reaches 210, it is reset to 0. Then, the y-pos of the cloud ROMs are set to this counter plus the starting height of each cloud ROM whenever the cloud ROMs are reset back to the right side of the screen.

➔ Ground Animations:

The ground animation was done using the game velocity and a single ground ROM that spanned the entire 640-pixel screen width. Using modding, the ground ROM loops easily and looks great.



Ground animation logic visually

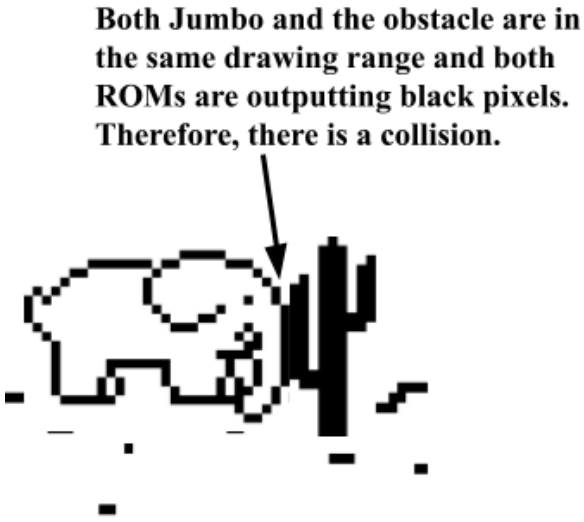
Collision Detection:

➔ Collisions with ROM

To deal with collisions, we used color and sprite priority to create a collision detection signal. Collisions occur when two different sprites "drawing ranges" overlap each other. When this happens, sprite and color priorities determine which ROM to pull from for each pixel. For example, to have the Jumbo stay in front of the moving background when the "Jumbo draw range" and "ground draw range" overlapped, the VGA display would pull from the Jumbo ROM whenever the output was black and would pull from the ground ROM whenever the output was white. Color priority was required for collisions because each sprite is drawn in the smallest box it can fit in–where all of the pixels not making up the sprite are made white. This means that the sprites contain white pixels around them that we do not want to draw if other sprites' ROMS are outputting pixels of non-white colors at those coordinates.

The game is intended to end whenever the Jumbo collides with a cactus or a pterodactyl. To implement this, first, it was determined whether the "drawing range" of the Jumbo overlapped with the "drawing ranges" of any of the cacti or the pterodactyl. If any of these ranges overlapped, then the color of Jumbo's ROM and the color of the obstacle's ROM that Jumbo collided with were checked. If both of the ROMs are outputting black, then a collision has

occurred because Jumbo's ROM is attempting to draw a black pixel at the same coordinate as the ROM the Jumbo collided with. If the Jumbo ROM or the ROM the Jumbo collided with is white, this means that one of the extra white pixels in the box of each sprite has collided with another sprite but not the actual sprite itself, so it is not considered a collision. Whenever a "game-ending" collision occurs, the game state is swapped to the "game over" state.



Collision detections logic

Score:

The score is one of the fundamental parts of our game, without it, it would be incomplete. The score is what creates the adrenaline of high scores, the incentive for competition, and helps the user visually predict when the game is going to speed up. To implement the score we created a new module dedicated to all of the score logic.

➔ Score display implementation:

This module instantiates digit roms for the numbers 0 - 9 and port maps their RGB outputs each to a respective signal correlated to the digit number. This way, whenever we want to display a certain digit, we can just set the RGB output of the scoring module to be the signal holding the specific digit ROMs output RGB color. To determine where our score digits are gonna be drawn we created 5 boolean area signals for digits next to each other in the top right corner of our screen. We calculated these areas using the dimensions of our digit ROMs, which all have the same dimensions and added necessary padding pixels to separate our digit boxes from each other and the borders of our display. These signals just hard-coded the bucket coordinates to put our digits in which adds a lot of modularity to our design.

Then within if/elsif statements using the boolean area values, we can tell patterngen to pull from a specific digits ROM while in a certain bucket. For example, if the score was 59 this diagram represents how we use the area signals and ROM RGB output signals to map digits to score boxes.

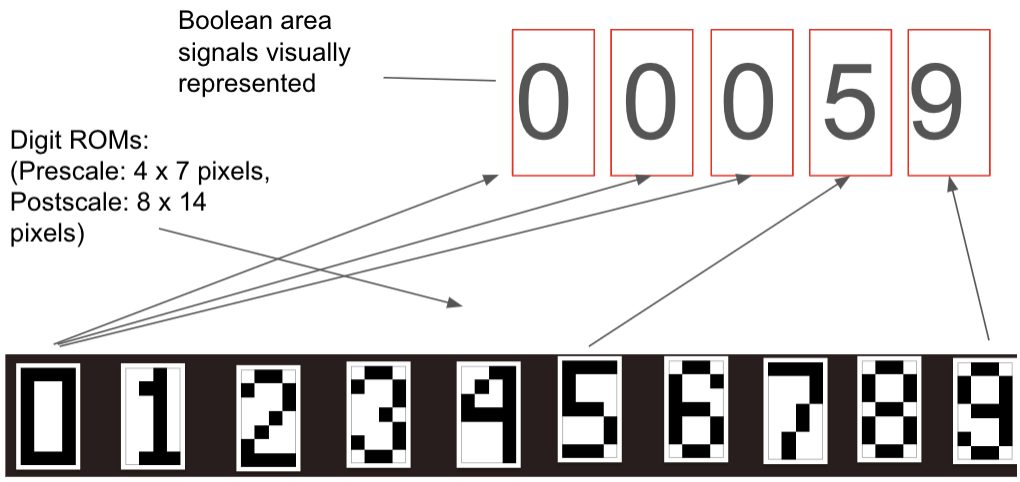


Diagram displaying the area signal buckets and corresponding ROM RGB output signals to map digits to score boxes.

➔ Score Counter Implementation:

To implement our score tracker we created a new submodule that ran with our 60 hz game clock. This module calculates and tracks the game score, dynamically adjusting the update rate based on the velocity, which is also dependent on the score. Using the 60 Hz game clock, the module increments a timer signal that ranges from 0 - 6 and gets + 1 on every rising edge of the game clock. Every time this timer reaches 6, we increment our score by game velocity. Effectively dividing the clock signal so that the score increases by ten times every second, which we thought suited the speed of the score. The score is held by 5 discrete signals that each range from 0 - 10 and correspond to the value that should be sent to each area bucket described above. Every time the timer gets to six, the first digit (most right digit on the screen) gets incremented by game velocity. If this digit (which is held by an integer range 0 to 10) reaches ten it gets set back to zero and increments the next digit. Then if the next digit gets to 10 it goes back to zero and the third digit gets + 1. This cascading chain of constant overflow checking allows us to keep a decimal score, and not use any divisions or mods, which would be necessary if we had to convert a binary score to decimal. This module then simply just returns the digit values corresponding to each bucket back to the score module.

➔ Choosing the right digit ROM:

The signals that come from our score counter are then used within every correlating if/elsif boolean area statement in the display logic by getting fed into a function that receives the digit held by the signal and returns the correct ROM digit RGB signal to pull from. This digit RGB output is then what gets sent back to patterngen from the score module. In this way, we create a very harmonious and modular connection between our score counter, our display logic, and our score rgb output back to patterngen.

Game Velocity:

Within our patterngen module, we instantiated a binary score counter to increment in parallel with our decimal score counter in the score module. The binary score increments using the same 60 hz clock division used within the score logic to initially only increment 10 times every clock cycle. The value of the increment for all of our movements is always determined by a signal called game_velocity.

➔ Calculating game velocity:

We used a separate module called speedCalc that took in the binary score, and depending on its value, returned a specific game velocity which was port-mapped to patterngen's game velocity signal. Within speedCalc we made it so that if the binary score was less than 250 (0 - 250) then the game velocity would be 6. We tried multiple starting game velocities and starting at 6 was a perfect speed for our initial ground and obstacle speed. Then with more elsif statements, we created different checkpoint scores that would increase the game velocity by one. Whenever the binary score reached a certain amount, the speedCalc module would automatically update the game velocity being used and sent out through patterngen, effectively manipulating the speed of the whole game.

Results

We are incredibly proud of Jumbo Dash. It is fully functional, consistent and, most importantly, a super fun game. One particularly special quality of Jumbo Dash is its standing as an endless runner game. There is no end to the game - the highest score recorded yet is in the low 1000s, but it's possible to score as high as 99, 999. The game's difficulty is part of its draw - reaching new personal high scores is exhilarating and pulls you back in.

Links to videos(Again)

Picture of Jumbo Dash at the showcase

Jumbo Dash isn't perfect. There are a few small improvements to Jumbo Dash that would fully realize it as the game we originally envisioned.

➔ Fully Random Obstacles Generation:

To new users, the frequency and pattern of the obstacle generation is not obvious. To these newcomers, Jumbo Dash produces obstacles that pop up randomly. This is not entirely the case. The game actually sends obstacles through a predictable loop of ten cacti and three pterodactyls, which were hardcoded into the game.

Fixing this would mean implementing a random VHDL element, like an LFSR, in the game's obstacle generation module. This LFSR would present a random number of any desired length whenever called, which could end up driving which obstacle ends up being displayed in the game.

➔ Working High Score Tracker:

A high score tracker would be another neat feature that Jumbo Dash doesn't offer yet. The score display resets every time the game enters the "Game Over" state, so there is no active recording of a high score in Jumbo Dash. Adding a high score tracker would always print the highest game score recorded to the screen, regardless of whether that score is being actively played at the moment. This would be an exciting addition to Jumbo Dash.

➔ Tweaking Game Difficulty:

As mentioned above, Jumbo Dash is a difficult game. No user has passed the 1050-point benchmark–the game past that point is still unseen. The Jumbo Dash user experience would be improved significantly if it was slightly more rewarding. Getting stuck, run after run, is demoralizing for new users. Slowing down the game's velocity increment through playtime would be an easy and effective way to do so.

➔ Jumbo ducking Mid-Jump:

Implementing logic so the user can press the duck button while jumping and shoot down to the ground really fast would add an enhancing mechanic to the game. This mechanic would enable players to go further into the game, bringing it closer to Google Chrome's depiction than Geometry Dash.

Reflection

➔ How did the project go?

- Isaac: Jumbo Dash went fantastically. I have never spent more time on one thing before, and I am proud of the final product. We pushed what we had learned during the semester to the next level in this project, surpassing my expectations. I feel a lot more confident and comfortable with VHDL, and I feel like I have a much deeper understanding of the language. This is the biggest programming project I have ever done, and I am glad that all the hard work paid off for a rewarding final product.
- Jacob: Looking back on the project, I'd say it went extremely well. Personally, I've never spent so much focused time and energy on a singular effort before making Jumbo Dash. I can proudly say I am much more comfortable with VHDL and large scale programming projects now as a result of contributing to this game. It was a lot of fun, and absolutely worth the effort!
- Jan: I think overall the project was extremely successful. We were able to achieve most of the things we wanted, with some minor shortcomings such as the sound being left out. The game came out exactly how I had imagined it from the beginning. I feel as if I

understand VHDL and digital logic much better than I did before. Being able to go through struggle, and then persevering as a group really helped us all strengthen our skills and comprehension of the ES4 material. I have never been as proud of a final product as I am for this one. We all dedicated so much time and effort into this project, and I think we should all be more than proud of what we have been able to accomplish.

- Ryan: In my opinion I would say that the project went very well. As a group we focused so much energy and time into making this project and I can say that I am proud of us and what we have accomplished. This project immensely improved my VHDL programming skills and also helped me learn more about how to create large projects. In addition my debugging skills in VHDL have also improved as we encountered many issues that we had to fix during the project. This is the most programming that I have done for a project so it was great to see it completed. This project was super fun to complete and definitely worth the late nights in Halligan.

➔ <u>What went well?</u>

- Isaac: Our collective interest and enthusiasm for this project and the tangible results on screen motivated us to work hard and enjoy the process. We were all eager to get the next aspect going and we all had awesome ideas. We had a collective vision, and we pursued it with curiosity, bringing things together nicely in the finished product.
- Jacob: I was really happy with how involved and enthusiastic the group was about the project. We were transparent with each other, and were all eager to make a game that we could all be proud of. I think having a collaborative and excited group culture is a huge motivator in a group project - we definitely had that going for us and it made the project much more fun than it would have been if we didn't have that harmony.
- Jan: I think that one of the main things that allowed our group to thrive was our perseverance. We had many moments where we ran into fundamental errors, or other very demoralizing issues while implementing our code. Regardless of these setbacks, we always continued working until we had something we were happy with. This way, our team morale and motivation was able to stay high throughout the whole project.
- Ryan: Overall, this project went very well. Every group member was very motivated and involved with the project, spending lots of time in Halligan. This project was definitely a challenge but our group stayed persistent and worked hard in order to complete our project. We all were able to work together well, helping explain topics to each other and also helping find bugs in the code.

➔ <u>What didn't go well?</u>

- Isaac: On occasion, we struggled to express exactly what our ideas were and how we could realize them in VHDL, leading to trouble trying to get other group members on the same page. We sometimes would hop into something without understanding it on paper, and we would get errors and bugs. I felt that when we sat down and understood exactly what we wanted to do, we made the most progress. We could have saved a lot of time if we had been less impulsive with how we implemented new VHDL code.
- Jacob: The way we went about implementing our ideas sometimes didn't go well. We struggled at certain points breaking down our ideas in dialogue before implementing them. This led to more tedious time struggling through bugs and errors than if we just slowed down earlier and worked through ideas as a team. There were memorable moments where we'd just rush in, almost thoughtlessly, to an idea that had flawed

reasoning. We were all a victim of this at some point in the project, and we likely wasted many hours needlessly jumping into Radiant before breaking down every idea as a group.

- Jan: I think we could have done a better job with splitting up our work, and modularizing the code more. We were usually all working together on one screen, which often made communication overwhelming with too many voices. I think if we had found a better way to work on the project separately and then divided it into explicit sections, we could have been more modular and successful.
- Ryan: One thing that did not go well is how we went about implementing new features into our project. I think that we should have spent more time making sure everyone was on board with the way the new feature would be implemented as there was times when we would have to stop and explain why were were implementing a feature in a certain way. If we planned out the implementation of every feature before I think it would of saved us a lot of time.

➔ <u>What would we do differently if we could start over?</u>

- Isaac: I would try to be more streamlined and thoughtful about what we wanted to do. I would suggest implementing smaller code blocks at a time–Code a little, test a little. I think we should also have started with more modularity in our code, implementing each piece of the game in a separate file as we went. This would have improved the organization and efficiency of the project while saving time along the way.
- Jacob: If we could start over, I would suggest we make a set of rules/guidelines for the process of implementing ideas into code. If we established a loose set of rules where you HAD to talk over what you planned to write before writing it, and then we took small bites at writing that code (synthesizing every 5-10 minutes instead of every 20-30), we'd probably have spent less late nights in the Halligan Lab.
- Jan: I think that we set ourselves back a little bit by not organizing our program from the beginning. We started ripping code into our patterngen instead of making organized modules for each section of our code. Then once we had already made substantial progress within our code, we had to then split it all up into more modularized sections, which was much harder than if we would've started with them. However after we started modularizing everything, the process went much smoother and we were much more productive. I think if we could start over it again, it would be very important to have a layout planned and to stick to it, instead of trying to figure everything out one by one.
- Ryan: If we could start over, I think we should have spent more time testing small aspects of features slowly before implementing the full feature. This would have allowed us to catch bugs in our features much earlier rather than trying to debug after the feature was fully implemented. I also think we could have improved the modularity in our code as for example, our pattergen.vhd contains a lot of if-statements that could have been modularized into a function.

Work Division

Jumbo Dash is a game we are incredibly proud of. Over countless hours in the lab, we worked together brainstorming, diagramming, and debugging to bring this project to life. Our different backgrounds and shared willingness to collaborate allowed each of us to contribute meaningfully to different aspects of the game. While the final product reflects a blend of our unique skills, we believe it also embodies the collective dedication, effort, and camaraderie of the entire team.

No one member did less than was expected; we supported and challenged each other every step of the way. Our work was well-organized and seamlessly integrated—for example, when Jacob and Isaac focused on obstacle generation, Jan and Ryan were creating new ROMs. Each of us played a key role in shaping the project, and we leaned on one another to overcome challenges and achieve our shared vision.

- ➔ Isaac: Jumbo Animations, Ground and Cloud Animations, Obstacle Generation, FSMs, Collision Detection, VGA, Block Diagrams
- ➔ Jacob: NES Controller Implementation, Obstacle Generation/ FSM, Priority Animation, Game Speed Implementation
- ➔ Jan: Obstacle Generation, Score, Drawing and Making ROMs, Collision, Game Speed Implementation, Python Script for ROMs, Ground and Cloud Animations,
- ➔ Ryan: Obstacle Generation, Drawing and Making ROMs, Crouching Logic, Collision, Ground and Cloud Animations,