Humming bird

By Joanna and Margaret Sands

Project Overview/Summary:

For this project, we created a game (Scream-y bird or Humming Bird) in which the player's sprite is controlled by vocal variation. Specifically, we take in a player's vocal cues, identify if the frequency is "high" or "low" and that information to direct the sprite. Game play consists of a side scrolling world in which the player can only move vertically in order to dodge incoming objects. Additional features of the game include adjustable game components as well as a replay mode with audio capabilities.

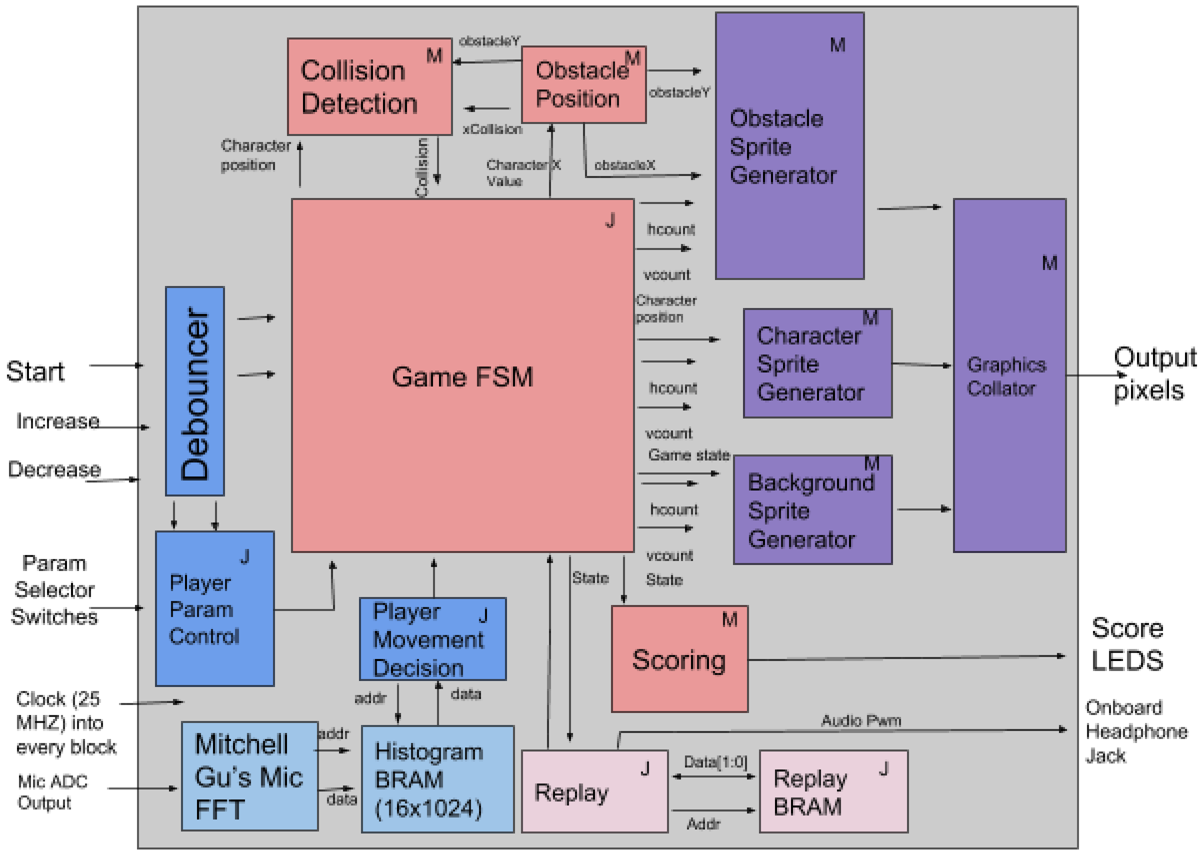

Fig 1. General Block Diagram of the modules that make up the game. Different colors identify functional groups that we used to divide tasks.

Joanna’s Major Components

Game FSM

The Game FSM module acts as a center point for the interconnection of other modules. It also keeps track of the current state of the game.

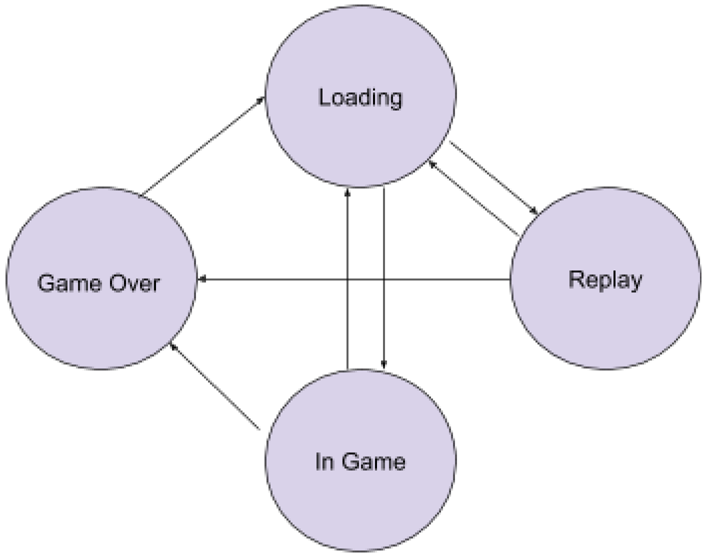

Fig 2. State diagram for overall game state.

In general, the game has four states. When a player starts a new game from any state, the game goes into the loading state, where player and obstacle location are reset and adjustable parameters are loaded in and set for the duration of the game. Once these actions are done, the game transfers into the In Game state.

In the In Game state, if the player hits an incoming object, then they are immediately transferred to the game over state. From the game over state, the player can choose to start a new game, or go into replay mode, which will be further explained below.

Aside from this state machine, the Game FSM module is also tasked with updating player vertical and horizontal location each time the refresh timer (called update_state) is raised high during the In Game state. Because the players would prefer the character x value to move at a much slower rate than the character y, Game FSM also keeps a counter of refresh times so it can only move the character horizontally after some number of update which can be set by the player input. The vertical movement, while more straightforward, also involves bounds checking to make sure the character stays on the screen.

Player Inputs

Tone Recognition

In game, character movement is dependent on whether or not the system recognizes that someone is making noise nearby and classifies said noise as high or low. In order to make a system that could do this, I first tried to visualize what the output of an FFT would look like for a variety of noise inputs. To do this, I used the FFT Demo shown below in figure 3.

To isolate the range of pitches in which humans usually hum, I adjusted the scaling factor and the sampling range in order to amplify inputs within the desired range of sounds. This process was typically trial and error with the scaling factor and sampling ranges mapped to different switches on the Nexys 4. Originally, with 0 scaling, all outputs seemed very large and it was difficult to see the result of any specific sound. Finally, with the correct scaling, I could see peaks for really consistent frequencies, like whistling, but couldn't visually isolate anything for normal speech or for humming due to harmonic resonance. To combat this problem, I then decreased the frequency sampling range to make it such only one or two peaks would appear for each hum or noise.

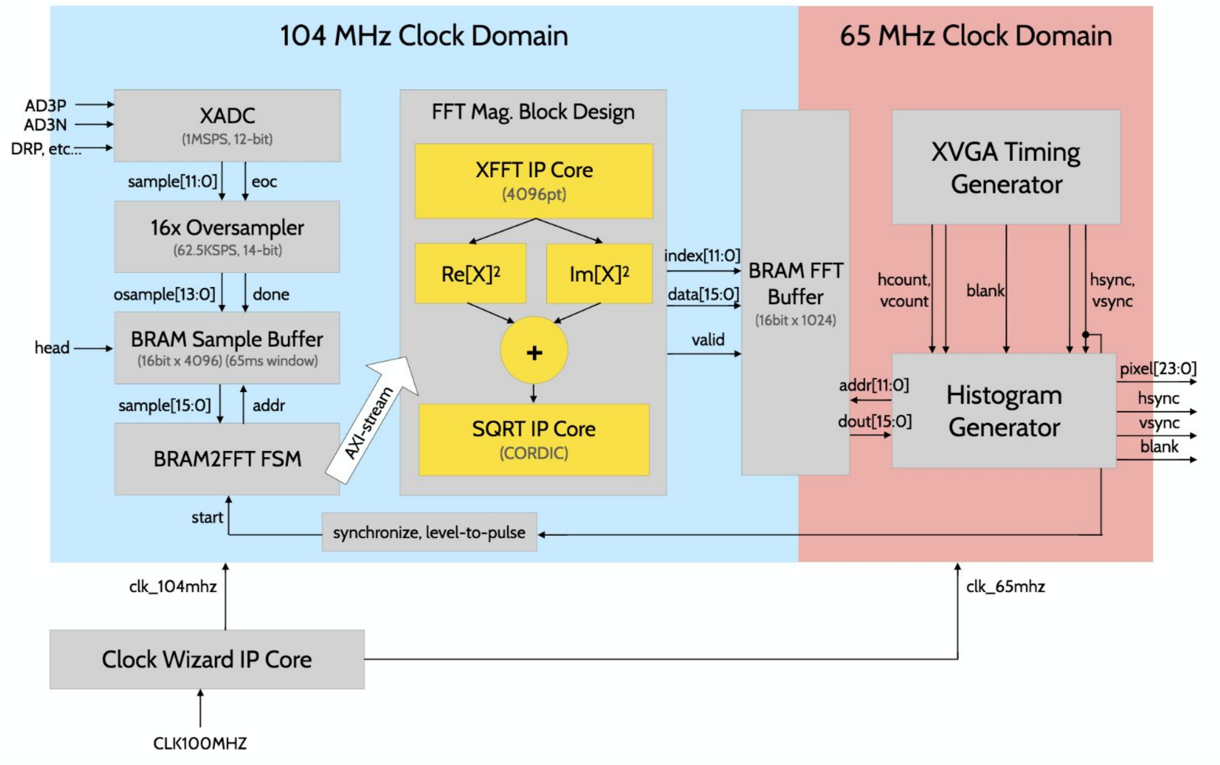

Fig. 3 Block diagram for Nexys 4 FFT Demo taken from github user MitchGu

Once these parameters had been set and I had a relatively useful histogram, I began to implement a way to extract my desired 2-bit output from the histogram. To do this, I first added code to scan the histogram and return high when there was a frequency with an amplitude above a certain level. Once the appropriate volume threshold was determined, given the average noise value within the lab, I then began to keep track of the frequency that had this maximum amplitude. From there, I added a user adjustable frequency threshold. Now, if the stored highest-amplitude frequency is above the frequency threshold, then the system considers it a high value. If it is below the frequency threshold, it is a low value.

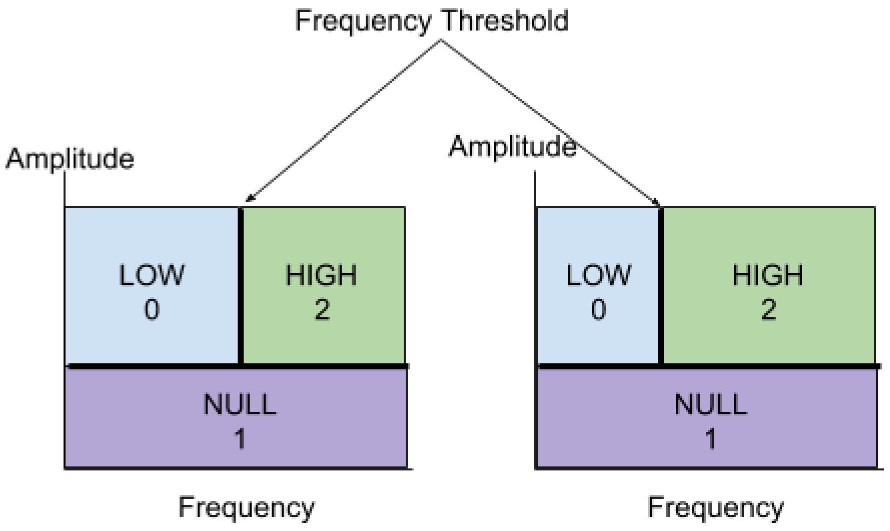

Fig. 4 Each region in the above charts show the system output given the location of the max-amplitude frequency for a given pass of the histogram. Comparison of the two graphs shows that as the frequency threshold decreases, notes that were once considered low are now considered high.

The system as described worked fairly well, however moves were fairly jumpy as they were based on random samples of the outputs of the histogram sweep. To smooth out the movement and make it more consistent, I implemented a rolling average of the incoming values. Instead of actually calculating an average, however, I made sure I didn't need division by keeping a shift register of the 50 most recent samples as well as a rolling total of the values in the register. Whenever a new sample came in, I add its value to the running total and subtract the value of the sample that occured 50 clock cycles ago. I then load the new sample into the 50 sample buffer. I then compare the total to 50*expected average to determine the desired direction (in this case, the expected average is 1 which makes it easier to distinguish between lows and highs) Because the rolling average (rolling total) is an example of a Low Pass FIR filter, it did help a lot in smoothing out possible misreads.

Parameter Adjustment

To allow the game to be change based on the user's preference, I added a module that would allow users to change the vertical distance between the top and bottom walls, the character x speed, the character y speed and the frequency threshold. By adjusting the switches along the bottom of the Nexys 4 board, user's can select different parameters and display their value on the 7-segment hex displays. To adjust the values of these parameters, users can press the up or down buttons and see the change displayed on the hex screen. These changes won't affect the current game if it is in session. They will be loaded into the system the next time a new game is started.

Replay Mode

In order to allow users to view a replay of the previous game, I implemented a system that stored the player direction on each position update using two bits (saying whether it went up, down or stayed in the same place). In order to replay, I used a ternary expression to pass in the output from the BRAM whenever the state was set to replay. Because the game is deterministic given the same obstacles and inputs, the game state doesn't need any other change to make the replay possible.

While silently watching the replay was interesting, both my partner and I thought it would be more enjoyable if the game imitated noises made during the play through. To do this, I read the replay data from the BRAM and generated a square wave whose frequency was either high or low depending on what direction the bird was supposed to move.

Margaret’s Major Components

Collision Detection

The collision detection model takes in the character’s height, the height of the obstacle that will be colliding with the character, a signal describing if the bird and the obstacle’s x values are colliding, the direction of movement of the bird in the next timestep, and a bus describing the user inputted values. From there, the module determines if there is a collision at each clock cycle by making 3 checks. The first checks that xCollision is 1. The second checks if character’s height is below the height of the obstacle. The third checks if the character’s height is greater than the height of the obstacle + the distance between the top obstacle and the bottom obstacle. If the first and either of the second are true, then the output (collision) is set to 1, meaning there is a collision.

Obstacle Position

Obstacle Position takes in the game state, the user inputted values, the character’s current X value and the noisy height of a bar from the microphone histogram in order to introduce randomness. The obstacle position module has four main parts, it creates the obstacles and keeps track of their heights, it calculates the obstacles on-screen x values, and it determines if the bird has an x collision with any of obstacles,.

All obstacle heights are stored in a 10-bit array which is initialized to value generated by an online random number generator. Each of the obstacles are evenly sized and evenly spaced in a larger “map” with a size that is a power of 2 which loops over time. When an obstacle from the array becomes the one that will next appear on screen, a value is taken from the noisy microphone input in order to create a “random” height for the next obstacle. In order to make sure that a newly generated object does not immediately end the game, the obstacle that covers the starting location is never updated. Because only off screen values are updated, if a player cannot make it past the first couple obstacles then they will restart the game with the same map, however this is better for players learning to play the game so that can practice without having to deal with obstacles changing. These obstacle y values are outputted as a single 60-bit bus with each location taking 10-bits allowing for up to 6 obstacles on screen at anytime.

In order to determine what objects are on screen, the character’s x value must be converted into a coordinate system that describes what of the larger map of obstacles can fit on screen. The bird should always be in the center of the screen which means that its left side has a constant x value of 320 - characterWidth/2 on the screen. On each clock cycle, the left-most obstacle on screen is found by taking the most significant bits of the character’s x to get an index in the array. The left-most obstacle’s in-map x value is then calculated by multiplying its index by the sum of its width and the amount of blank space left between each column. It’s on-screen x value is then calculated by finding the difference between that number and the bird’s current x value offset by the bird’s on-screen x value. From the left-most obstacle, the x value of rest of the obstacles are calculated by adding their width and width of the blank space. These obstacle x values are outputted as a single 60-bit bus with each location taking 10-bits allowing for up to 6 obstacles on screen at anytime.

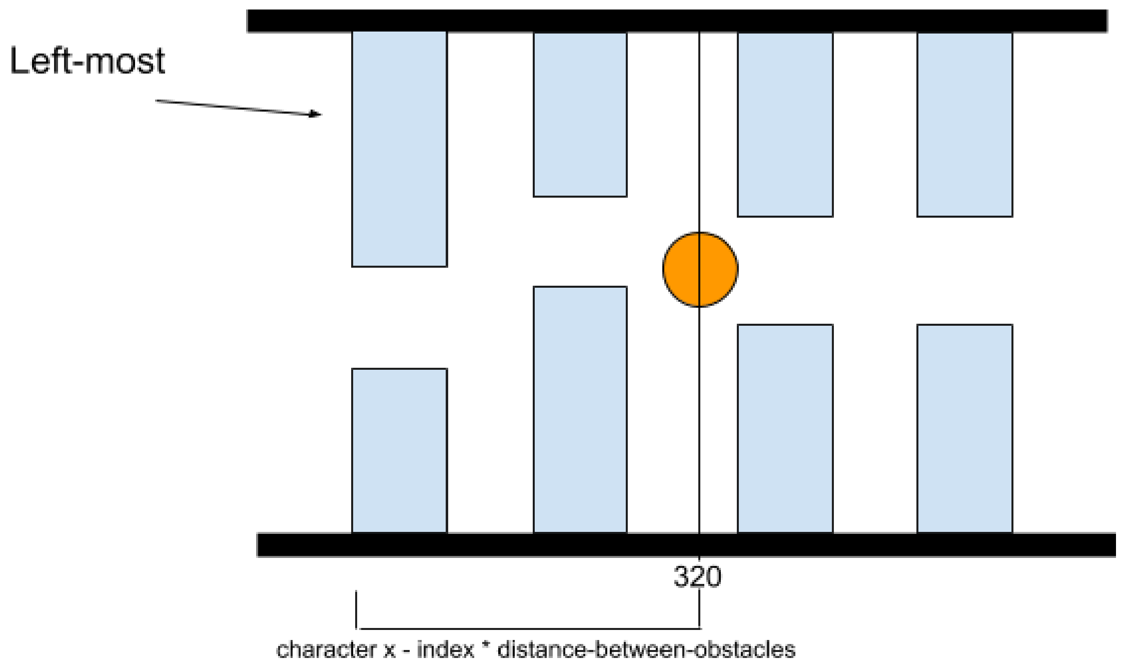

Fig. 5 Position of obstacles on the screen are determined based on their distance from the bird in the map

Because each obstacle and the screen have a set width, and the on-screen x-values are calculated starting from the leftmost obstacle, the obstacle that will collide with the bird has a specific index in the list of obstacles that appear on screen (in our case this was the 4th obstacle) which we call the colliding obstacle. Using the colliding obstacle, we check the sides of the character against the the sides of the obstacle. More specifically, there is no collision if the x-value of the right side of the character is less than the left side of the obstacle or the x-value of the left side of the character is greater than the right side of the obstacle.

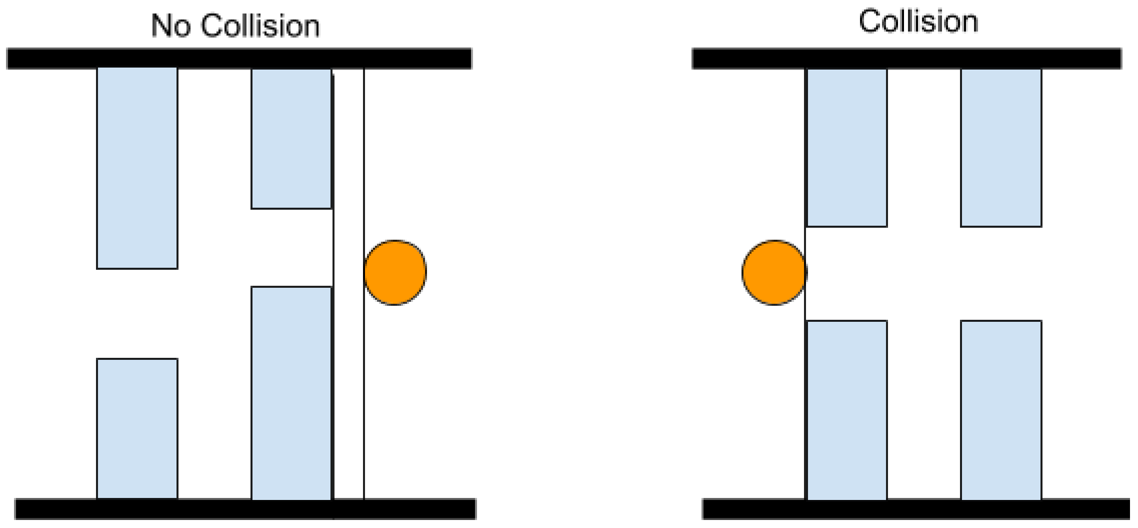

Fig. 6 In order to an x-collision to not occur, the left side of either the character or the obstacle must be greater than the right side of the other

Scoring

The scoring module is a simple counter that takes in the state of the game. It starts the counter when the game switches to the In Game state and continues to increment until the state changes. The counter is reset when the game switches to the Loading state meaning that it is about to start a new game.

Graphics

The graphics for Humming Bird are separated into 4 modules. The Character Sprite Generator takes in the character's y positions as well as the hcount and vcount. It then uses the picture blob code from lab 3 to draw the flappy bird image with the top left corner at ((320-character_width/2), character y). The loading of the image from the ROM takes two clock cycles so there is a delay unless the address is offset by 4. The Background Sprite Generator returns the background color of the game.

The Obstacle Sprite Generator takes in the 2 60-bit data buses that describe the on-screen x and y coordinates of the obstacles that could appear on screen a well as the state and the user input databus. Depending on the state, it will color the blobs a different color in order to help the player understand when there is a change of state. It then draws an upper and lower portion for each obstacle using the “obstacle blob” module (a modified version of the blob module from lab that takes in the height of the blob as an input). Each upper portion has a y value of 0 (starts at the top of the screen) and uses the height from the on-screen y coordinate. Each lower portion has a y value of the on-screen y coordinate added to the wall distance given by the user input and a height equal to the screen height minus its y value. In order to have a single output, the outputs of each of the blobs are connected by ors.

The final module is the Graphics Coallator which takes in the output of the other module and creates a z-order where walls are displayed then the character is displayed and then the background.

Challenges

We struggled a lot when integrating our code with one another. Not because the modules were incompatible, but because we had a fatal error in our GitHub repo that resulted in Vivado being unable to start after every attempted merge. To solve this issue, we began to only have one of us working at a time, using one Athena account. This prevented us from having to use any kind of merging protocol but made simple tasks take longer because we couldn’t work in parallel.

Another challenge that we faced was determining what coordinate system a calculation should be done in and what system the outputs should be in. There were many times where an input x value was in terms of the screen but was used as if it were in terms of the map and vice versa. Similarly, there were time when we struggled with the signs of each of the terms when translating between systems which caused confusion when the bugs appeared much later.

Along with design challenges, there were times when the verilog implementation proved to be a challenge for us. One example is that we originally planned to send obstacle y values as a shift register, however there were bugs in the implementation that caused it to shift too often or not often enough because the design relied on waiting extra clock cycles to complete a shift. This was fixed by implementing obstacle values as an array instead.

One final challenge we encountered was syncing values between what parameters had been updated by the player during the Game Over state, and what the rest of the system should use for the next game. Originally, I had them as different inputs and outputs and was updating them on the start signal and raising another flag to say when the values had been changed, however this had two problems of adding a lot of inputs and outputs to all of the modules and creating a system where the estimated delay was incorrect so the game FSM would latch the old value before the new value was written. In the end, I created an additional state called loading, which made the Game FSM wait an extra clock cycle before latching the data that was coming in from the parameter cycle.

Lessons Learned/Advice for Future Projects:

In order to simplify logic, we made a large number of constraints that the sizes and number of the obstacles must follow. If this project is replicated, it might be better to make a more robust system that is ready for change when we want to change the length of the array which is the limiting factor on replay mode. Similarly, in order to keep a player from dying immediately, the obstacle height of the obstacle closest to the starting value cannot be changed. There may be a way to redraw or transform values in order to keep this edge case from occuring while also making all of the obstacles random as the player goes through the game.

For this project, the FFT we were interfacing with and the histogram it was creating both ran at different clock speeds than the rest of our project. For a while, we really wondered how it would be possible to align the clocks such that the histogram didn’t change while we were reading the values. Eventually, however, we realized that instead of doing these calculations, it might be better to create a system that samples the output such that changes to the histogram during the read through would not actually hurt the results or make them unreliable. For instance, a change of values in the histogram that have already been checked does not actually alter the histogram seen by the sweep, and a change in the values that have not yet been checked means the new values are no different to the operation than the values that had previously been there. This is especially true given that one reading does not directly go into the output of the movement decision module.