

# Augury's Lament: An FPGA Based Wand Tracking Game

1<sup>st</sup> Danny Antonelli  
EECS

Massachusetts Institute of Technology  
danny\_a@mit.edu

2<sup>nd</sup> Lars Barkman  
EECS

Massachusetts Institute of Technology  
larsbarkman@mit.edu

3<sup>rd</sup> Samuel Bennett  
EECS

Massachusetts Institute of Technology  
sambnet5@mit.edu

**Abstract**—Augury's Lament is a fast-paced rock-paper-scissors inspired multiplayer action game where you cast spells by tracing out one of nine symbols with a light-up wand. The two primary challenges of this project are to identify symbols traced with a glowing pointer and to display ASCII text via a spritesheet. To more specifically define the identification challenge, a spell should register as correct if and only if a person attempted to cast it correctly and did a decent job at it. Of particular note, it should be counted correctly if the person cast it too fast or hesitated halfway, but shouldn't be counted if it was cast in reverse or doubled back. This means that time must be taken into account.

**Index Terms**—Digital systems, Field programmable gate arrays

## I. HARDWARE

In addition to the FPGA board, the game requires:

- An HDMI cable connected to an HD monitor. This provides the visual interface for the game for instructions for play, health visuals and turn visuals.
- An ov5640 camera. This allows for tracking of movement necessary for game-play.
- Two light-up wand peripherals. These peripherals provide the light that will be tracked to create the trace paths used in the game.

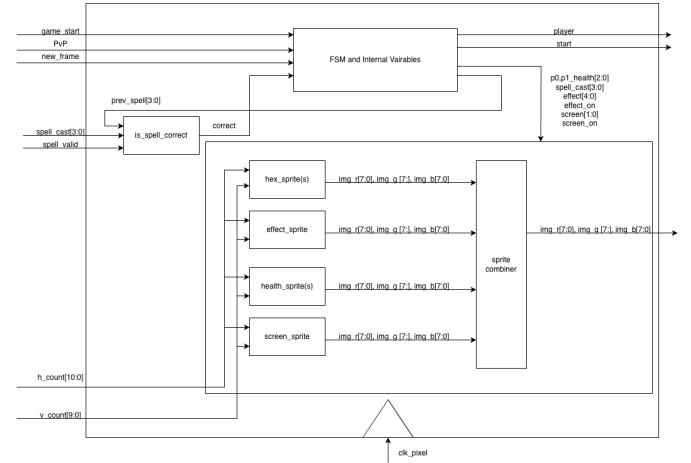
## II. CHALLENGES

The main technical challenge lies in normalizing a user's trace path in a way that is both permissive and discriminative. A wide variety of user behaviors must all map to the same normalized representation: users may draw too fast, too slow, pause mid-stroke, or wobble significantly, yet the system should still correctly recognize an honest attempt at a symbol. At the same time, the normalizer must not accept paths drawn in reverse, with major self-crossings, or with large detours, even though these traces may share a similar bounding box or endpoint structure. Achieving this requires carefully balancing translation, scaling, and timing equalization so that the essential shape of the path is preserved while irrelevant differences are removed. In practice, getting this balance right is subtle: overly strict normalization rejects valid but imperfect attempts, while overly loose normalization causes distinct symbols to become indistinguishable during comparison.

An additional challenge lies in actually displaying the information we need to evaluate the system. While our scaling and normalization can output how much change was required

to bound the drawing, and our comparer can evaluate how close a drawing was to the intended image, we need a solid way for both the players and us as the designers to view these numbers. This requires a visual that updates based on changing calculations rather than strict inputs.

## III. MODULE OUTLINES



### A. Game Logic and Visuals

1) *Game Logic*: The game\_logic module handles most of the game's function. It works as a major/minor FSM. The major FSM handles the start, playing and end states, while the minor FSM handles the repeated actions of each turn such as calling the trace path functions and changing internal variables like health and turn. Specifically, the minor FSM makes a request to the trace\_path\_maker and the modified center\_of\_mass modules, then awaits the response from trace\_path\_comparer. Once it receives a spell, it determines whether or not someone's health should be depleted based on the rules of the game, updates that information, and then awaits new frames while displaying effects before returning to request a new spell.

In the game logic, these conditions must be met for a spell to be considered correct:

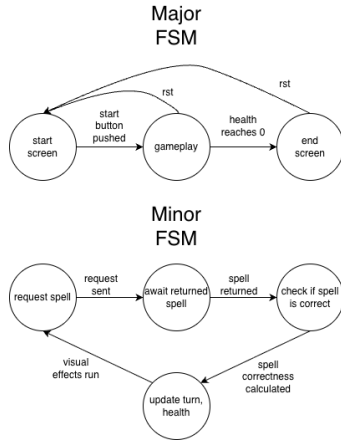
- 1) It is validly cast, i.e. comparer returns a spell
- 2) If the previous spell was correct, it must be between 1 and 4 digits less than the previous spell, modulo the number of spells (9 in this case)

- 3) If the previous two spells were correct, then it also cannot be the same distance from the spell before it as that spell was to the spell before it.

Examples:

| Current Spell | Previous Spell | Previous Difference | Valid |
|---------------|----------------|---------------------|-------|
| 0             | Any            | Any                 | 0     |
| Any           | 0              | 0                   | 1     |
| 1             | 9              | Any                 | 0     |
| 8             | 9              | 2, 3 or 4           | 1     |
| 8             | 9              | 1                   | 0     |
| 9             | 2              | 1                   | 1     |

The following FSM diagram displays how the game\_logic module transitions in its major and minor FSMs:



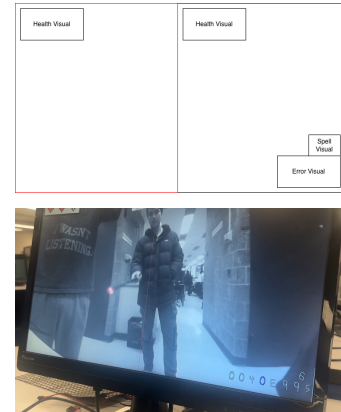
**Point of Complexity:** One function needed for the game\_logic module is a modulus, since we wanted spells that were lower than the previous, modulo the number of spells, to be counted as correct. However, modulus is a somewhat expensive computational block, so it was changed to use less expensive comparison statements. This helped to meet timing requirements.

2) **Visuals:** The visual module uses sprites to display important information for the game to progress. During the start and end screens, there is a full screen sized graphic that displays. This is a scaled up sprite, as a full screen sized graphic is usually too large to store in BRAM. Next, during gameplay there is a turn indicator around half of the screen, hard-coded without a sprite, along with a health bar for each player, and hexadecimal sprites that show which spell was cast and what the error was from that spell's true trace path, as returned by trace\_path\_comparer.

**Point of Complexity:** As the visuals need to output just one set of RGB colors, all of the sprites needed to be merged correctly. However, since they didn't overlap, this could be done computationally cheaply by just using bitwise OR. As all sprites would return 0 when out of that sprite, only the sprite that should be displayed, would be. This allowed us to remove many if statements to prevent timing violations.

Additionally, the hexadecimal sprites used for displaying the error and spell case were used to solve our last challenge, the one of displaying the information. It allowed us to debug by displaying different pieces of modules on screen.

Below is a diagram describing what can be seen on the screen, along with an image displaying a photo of the screen taken during testing:



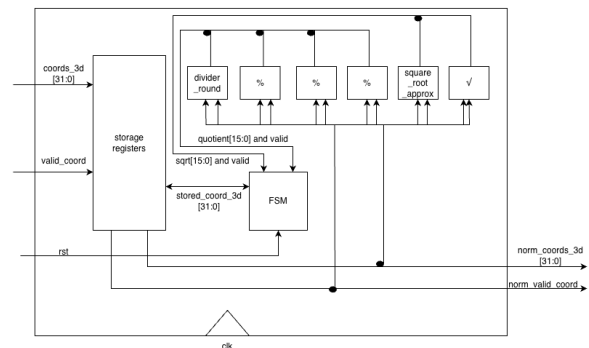
### B. Trace Path Maker

Spell identification follows a three-stage pipeline: conversion, normalization, and comparison. First, the traced shape data must be collected and converted into a standardized format. To preserve time data, we have chosen a data format which we will henceforth refer to as a trace path. A trace path is a set of 3D points that represent the 2D shape the player traced out, with the third axis representing time in frames. For example, (5, 7, 2) means the wand was at pixel (5, 7) on the third frame of casting (time is 0-indexed). A slightly modified center\_of\_mass module reads the camera data and determines where the end of the wand is on any given frame. When game\_logic requests data, trace\_path\_maker converts each frame's data into 32-bit coordinates. The coordinates are in this format:

| x-coord | y-coord | z-coord (frames) |
|---------|---------|------------------|
|---------|---------|------------------|

The x-coordinate takes 11 bits, the y-coordinate takes 10, and the other 11 are for z. Trace\_path\_maker will continue to store data until the wand turns off, game\_logic tells it to start over, or the player reaches the maximum frame count. After this point, it will output the coordinates one by one in the order received, one on each clock cycle, not stopping until done. Finally, it will return to wait mode.

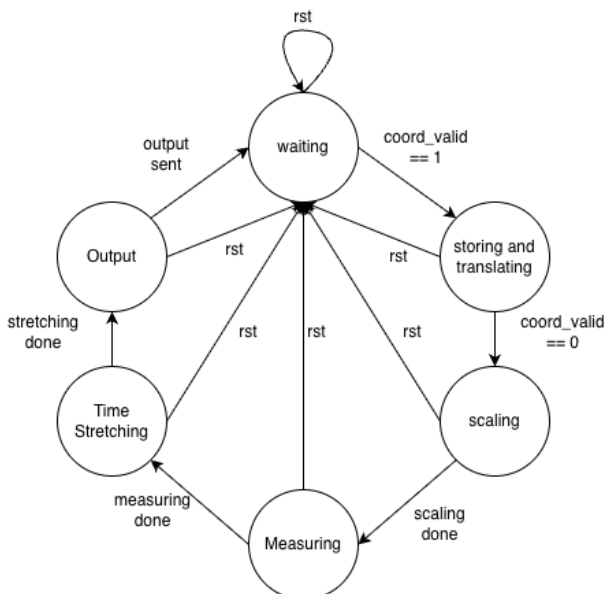
### C. Trace Path Normalizer



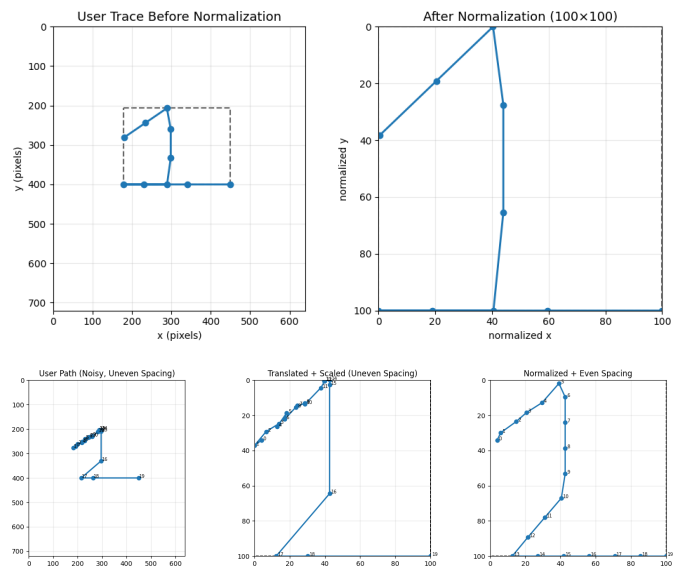
Then, `trace_path_normalizer` normalizes out position, size, and timing variations. That way, the players won't be penalized if they cast slightly off-center, too large, or at a different speed than normal. (Additionally, a large, off-center letter H that was drawn quickly still looks like an H.) First, it determines the bounding box of the trace path, which is the rectangle formed by drawing vertical and horizontal lines from the highest, lowest, leftmost, and rightmost points. It moves all the points along the x and y-axes an equal amount so as to tuck this box snugly into the upper-left corner of the screen, which is (0, 0). Then, it scales the trace path such that the bounding box becomes 100 by 100. (These dimensions were chosen somewhat arbitrarily. All that matters is that they're large enough that they don't erase a meaningful amount of data from the trace path.) It does this by multiplying the x coordinates by 100 and then dividing by the width of the bounding box. The y coordinates follow a similar process, except they're divided by the height. Next, it smooths out the timing differences by placing 32 points at regular intervals along the trace path. It does this by figuring out how long the (now scaled) trace path is, dividing this length by 31, and traveling along the path. Each time it travels the length of an even segment, it places a point. The z-coordinates are useful here, as they preserve the direction of the trace path's casting. Finally, it outputs the coordinates in the same fashion as `trace_path_maker` before re-entering wait mode.

**Point of Complexity:** In order to calculate distances, a `square_root_approx` module was necessary. We implemented this with one round of Newton's Method and hard-coded thresholds. In a possibly unnecessary attempt to keep timing down, the thresholds were all powers of two. That way, a comparison check could simple OR the most significant bits. This produces an error of less than half a pixel for any value less than 141, which is the maximum distance a path segment can be in a 100 by 100 box.

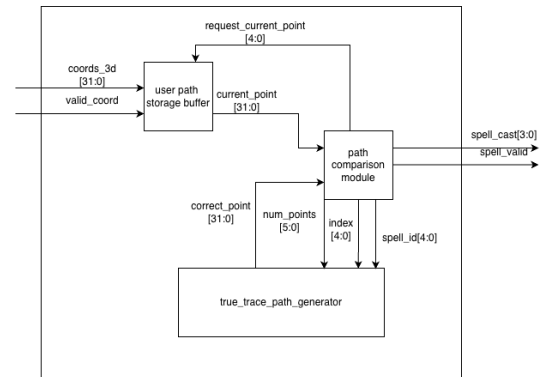
A diagram of the normalizer's FSM can be seen below:



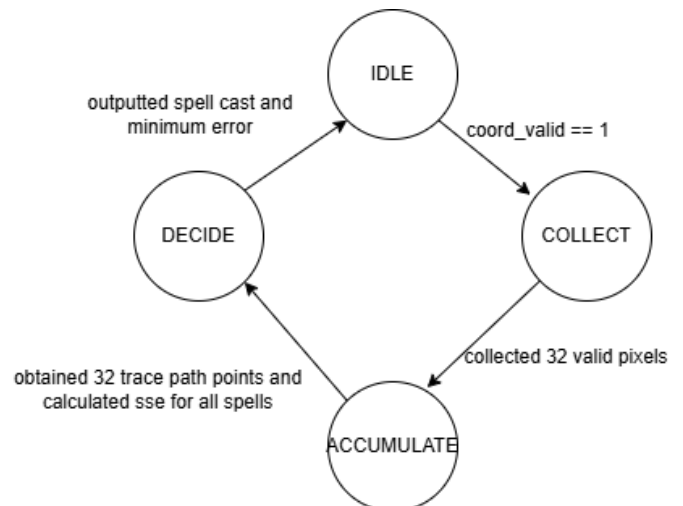
The graphs below demonstrate how the `trace_path_normalizer` would scale and translate an example of a trace path given by the `trace_path_maker` module. Note that only the individual points are what is collected by the system, the overall trace is shown as an example.



#### D. Trace Path Comparer and True Trace Path Generator



A diagram of the comparer's FSM can be seen below:



Upon receiving the first valid point, the `trace_path_comparer` moves from IDLE to COLLECT, where it obtains the sequence of 32 points from the `trace_path_normalizer` and buffers them in memory. Upon receiving all of the normalized points, the comparer then moves to the ACCUMULATE stage. Here, the comparer sends requests to the `true_trace_path_generator` where it receives all true trace path points for each spell. Upon receiving a true trace point, the module combinationally computes the sum of squared errors (SSE) between the 32 normalized points and the sequence of true trace path points for each of the 9 spells and moves to the OUTPUT stage. If any of the spell's SSEs are below a chosen threshold, the module outputs a single cycle valid and the spell which resulted in the minimized SSE. Through our testing, we found an error threshold of 10000 (decimal) to result in generous but fair gameplay for 32 frames per spell.

One area of complexity for this module is the generation of the 'true value' paths for each of the 9 spells. Since each true value path is comprised of equally spaced points along the numerical curve, the sequence of 25 true value points is for example entirely different than the sequence of 32 true value points. Since the number '32' for frames per spell cast may change, we wanted to make sure not to 'hard code' the true value spell paths. Thus, the true trace path generation is handled by a separate module: `true_trace_path_generator`. In fact, we did test 64 and 128 frames per spell end-to-end, and all of the modules and gameplay functioned as expected, although the game was 2-4 times as slow.

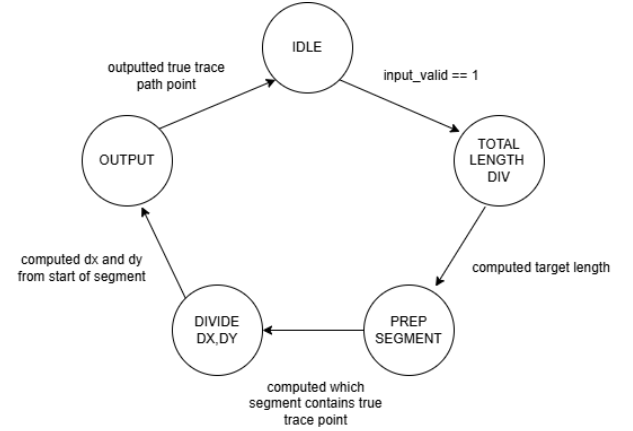
The `true_trace_path_generator` module (TTPG) takes in `spell_id`, `index`, and `num_points` and returns the (x,y) coordinate (in the 100 by 100 grid) of the point in the true trace path for spell id `spell_id` at index `index` where this trace path has `num_points` points. To do this, we hardcode a set of 6 'anchor points' for each spell, and given `index` and `num_points`, interpolate between the anchor points to determine the point at the fraction (`index/num_points`) of the total path. As a result, we procedurally determine the true trace paths 'on the fly', rather than hard-coding them in storage.

To do this, TTPG works in 5 main stages. Upon receiving valid inputs, TTPG moves to the total length division stage. Here, TTPG computes the quantity `total_len * index / num_points`, or the distance that the target point lies into the true trace path. Upon receiving the division result, TTPG moves to the segment prep stage. Here, the module combinationally computes which of the 5 segments the target point lies in, as well as the total  $\Delta_x$  and  $\Delta_y$  of that segment. Then, the module moves to the Divide DX and Divide DY stages, where it computes the  $dx$  and  $dy$  of the target point from the start of the target segment. Once these values have been retrieved from the divider module, TTPG moves to the OUTPUT stage, where it combinationally adds  $dx$  and  $dy$  to the target segment start point, clamps the result between 0 and 99, and returns that point.

Because of the TTPG module, the system is ready for change in that if a new spell is to be added, only 6 points

on the 100 by 100 grid would need to be manually calculated for that spell's trace path.

A diagram of the `true_trace_path_generator`'s FSM can be seen below:



## IV. EVALUATIONS

### A. Timing

Timing was an issue that we ran into. Much of the game logic and game visual modules were designed with minimizing consecutive ifs to limit combinational logic each cycle. Similarly, the normalizer and comparer modules use FSMs to split their math over multiple cycles. This allows us to do computationally complex calculations like division and square roots needed for normalization and comparison. Since we only need to do the calculations in around a frame, this gives us many clock cycles, so latency was never an issue as long as WNS stayed positive.

However, we did have some cases where WNS would go negative. Namely, when testing the the game with the number of points collected increased to 128, the WNS would sometimes drop to around -0.050ns depending on the build (Vivado could occasionally find a route that still had a positive WNS, due to non-deterministic building). With our most recent build with 64 collected points, our WNS was 0.167ns.

### B. Memory Usage

Memory use was a large concern for the image displays in game visuals. There are only 75 RAM36s and 150 RAM18s, so even just a start and two end screens of full 720p is enough to overflow them, let alone a screen full of effects. However, if we scale down the images to only 80x45, and then scale the sprites back up when displaying, we can save a lot of the RAM usage.

Our current RAM usage is 55 RAM36s and 2 RAM18s. Additionally, 36 of the RAM36s are from frame buffer and hd frame buffer. Only 18 RAM36 and 1 RAM18 were used by additions made for the visuals. Due to the compression, the screens only take 1 36RAM to hold, and the effects only took 4 of the RAM36s and 1 of the RAM18s.

We used palette compression as in lab 5, but due to the limited number of colors, it may have been more efficient to forgo this step, as this was really only for images that used a large number of colors.

### C. Spell Recognition Accuracy

Testing the project end-to-end was the most time consuming and laborious task of the project. By the nature of the manual tests, evaluation and debugging of the system was difficult because it was difficult to know the true reason behind an incorrect spell cast. Is the normalizer getting stuck? Did the path maker cut short (e.g. output only 10 frames)? Did COM modified fail to recognize the wand for one frame? Despite these persistent questions, after thorough simulation and hours of end-to-end testing, we believe the system to be in a state of commendable accuracy (up to human error).

Firstly, one issue we ran into was that a spell would be cast too early: that is, even though 1 second had not passed, the system would ‘immediately’ return a spell. We found this issue to mainly be due to the minimum threshold in COM modified. When moving the wand too quickly, we found that the camera fails to recognize the chroma red of the wand for  $\geq 1$  frame, resulting in trace\_path\_maker to cut short and the resulting casted spell to be ‘incorrect’. To fix this, we tuned the color mask of COM specifically to the red wand color (chroma red with 0xB minimum and 0xF maximum as the board mask settings) and set the minimum pixel count threshold to 10.

With this tuning, we found that the accuracy of the system improved as we did (as path makers). We do not have a complete statistical analysis, however over our most recent 30 attempts, we can confidently say roughly 60-70% return the correct spell (this is true for all spell ids).

### D. Testbenching

Testbenches were created for the trace path pipeline modules. This includes the true trace path generator and trace path comparer helper functions. Additionally, the game logic’s spell comparison logic was checked using testbench.

## V. CHECKLIST AND FUTURE IMPROVEMENTS

### A. Checklist

We met all of the commitments we made. We record, store, translate and scale points in a 3d space, compare them for accuracy against generated paths, and display UI visuals for turns and health. We also met almost all of our goals. We successfully scale the trace paths in time, and can even modify parameters to store more than our originally planned 32 points. We also implemented visual effects after spells are cast depending on the success or failure of the cast (based on game logic, not comparer). We also created a hexadecimal sprite sheet that can display the error for testing and debugging purposes. We didn’t reach any of our listed stretch goals, but many would be attainable from our current position with only a bit more time.

### B. Improvements

There are many additional things that we wanted to add to the project that did not make the final cut, either due to time or to slack constraints. For example, we did not get the chance to add in the sound effects or music to the game, and would implement that if given the chance. Additionally, we didn’t

add an AI to give the 1-player capability originally planned. Another improvement could be modifying the comparer to use paths other than drawn numbers. Many pairs of numbers, like (1 and 2) and (4 and 9) are drawn very similarly, which makes it hard for the comparer to identify which number was intended. Combined with the reversing of the camera meaning that currently the numbers need to be drawn backwards and the game becomes very difficult. Finally, one easy future improvement is simply fine tuning the error thresholds of the comparer and the spell\_is\_casting thresholds of the modified COM module to make the game more difficult. While we got the game to a playable state, these specific thresholds could be modified over time to give a better experience.

## VI. SOURCE CODE AND IMAGE CREDIT

The source code for the project can be found at <https://github.mit.edu/6205F25/fa25-6205-team57>

Although not used in our current build, the image is on our GitHub, so here is the credit to the creator of the royalty-free wizard sprite, obtained using a free account: ID 412403805 — Christos Georghiou — <https://www.dreamstime.com/wizard-fantasy-pixel-art-game-cartoon-character-blue-video-sprite-style-squares-wide-high-image412403805>

## VII. CONTRIBUTIONS

### A. Code

Sam wrote center\_of\_mass\_modified, trace\_path\_maker, trace\_path\_normalizer, and wrote the tests for the latter two. Lars wrote and tested trace\_path\_comparer and true\_trace\_path\_generator. Danny wrote game\_logic and game\_vis, along with the sprite models for visuals and debugging. We all worked on top\_level and helped debug each other’s modules.

### B. Final Report

Each person wrote the part pertaining to the modules they wrote. Non-module sections were contributed to by all members. Additionally, Lars and Danny did the Overleaf formatting and created the block diagrams and diagrams for the FSMs. Sam edited all of the final video and created the music which, while not in the final game, can be heard in said video.

## VIII. ACKNOWLEDGMENT

We want to thank Professor Joe Steinmeyer for teaching us this semester, as well as being our mentor for this project. Additionally, we want to thank all of the teaching staff of the 6.2050 course, as they provided invaluable advice during the planning and coding stages of the project. We understood that our semester was a particularly crowded one, so we appreciated every bit of one-on-one help.