# FPGA Hole in the Wall

Nathan Mustafa
MIT EECS
Cambridge, US
nmustafa@mit.edu

Cooper Price
MIT EECS
Cambridge, US
cooperp@mit.edu

***Abstract*—We present an FPGA implementation of the television game "Hole in the Wall" in which a player avoids collision with a wall simulated by the FPGA system and displayed via HDMI. The system streams video and game logic at video frame rate allowing for real time gameplay. We implement this system using the Nexys 4 DDR FPGA, evaluate its features, and discuss potential additional features for the system. The code for this system can be found here:** [redacted]

## I. Introduction

The game "Hole in the Wall" is a classic home television game from the late 2000s in which contestants standing on a platform avoid colliding with an approaching foam wall outfitted with shapes the contestants must make with their bodies to fit through the wall.

Fig. 1. Image from the "Hole in the Wall" television show

Our system is an FPGA simulation of this game in which a camera captures the shape of the player's body against a green screen while simulating a wall moving at the player - the player sees the shape of the wall on a screen connected via HDMI as well as an indicator of how far away the wall is. The system progresses the game with new wall shapes until the player fails and the game ends.

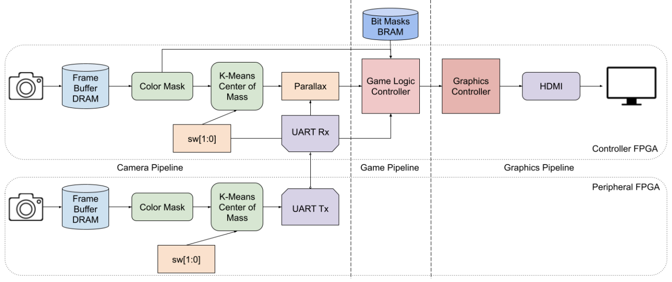

Fig. 2. FPGA "Hole in the Wall" System Diagram

Our implementation consists of three main pipelines: a camera pipeline, a game logic pipeline/component, and a graphics pipeline. The camera pipeline extracts information about the player's location in space - namely their shape and position. The game logic pipeline keeps track of and updates game state based on the input from the camera pipeline and outputs the relevant information for the graphics pipeline to utilize such as collision locations, the shape and location of the wall, and the state of the game. The graphics pipeline takes input from the game logic pipeline and outputs graphics to a screen via HDMI by displaying the camera pixels with an overlay of the wall shape and its current location location.

We take advantage of the FPGA infrastructure to run our game via a streaming pipeline - namely, as much as possible, we look to stream data throughout the pipelines and do necessary calculations "on the fly". For example, we pass pixel data through our camera, game, and graphics pipeline so that calculations are occuring at each cycle.

## II. Camera Pipeline

The system begins by receiving camera data from a camera and stores the full resolution video data into a DRAM frame buffer to cross between the domains of the camera clock and our main system clock. As we receive each pixel we mask out green values (the green pixels from the green screen) to identify what parts of the frame are a player.

Next we identify the pixels corresponding to each player using our variation on the k-means algorithm. In short, our algorithm computes centroids for each player at every frame, performing only a single iteration, before assigning each pixel representing a player to the closest centroid. Then it recomputes the centroids based on the closest pixels at the end of each frame. This allows us to pass the player location information to the game logic controller in order to determine if and which player collided with the wall.

Additionally, we calculate the depth of each player by using the stereo vision equation to parallax the center of mass outputs of multiple cameras and extracting the depth. In order to perform computation on center of masses extracted by two separate cameras, we pass the extracted center of masses from one board to a main "Controller" board that performs the parallax computation and the remaining game logic.

### *A. Parallax*

To estimate the depth of players within the game, we implement parallax using stereo vision. The system configuration consists of two FPGAs positioned 5 inches apart, each facing the players. The parallax module utilizes the baseline distance between the cameras, the cameras' focal lengths, and the pixel densities to compute the distance from the cameras to each player. For this computation, we rely on the centroid outputs of each player obtained from the k-means module, treating these centroids as shared reference points between the cameras for comparison.

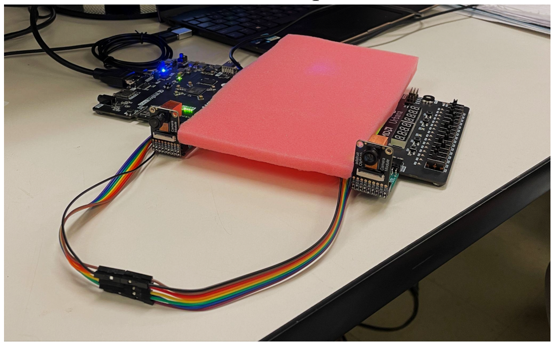
Fig. 3. Two Camera Setup with UART

Specifically, depth calculation involves the focal length of the cameras and the pixel density, which is derived by dividing the resolution width (1280x720) by the camera sensor width, as specified by the manufacturer. The inclusion of pixel density in the numerator accounts for the disparity being measured in pixels.

$$\frac{\left(\frac{\text{Resolution Width}}{\text{Sensor Width}}\right) * \text{Focal Length} * \text{Baseline Distance}}{\text{Disparity}}$$

Fig. 4. Stereo vision equation used

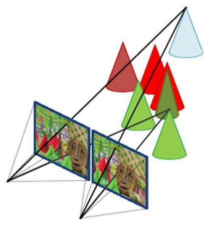
Fig. 5. Illustration of Stereo Vision

The parallax module performs the computation over multiple cycles. The absolute value of the disparity, defined as the difference in centroids between cameras for a given player, is calculated combinationally. Then division is executed over an upper bounded number of cycles to determine the depth. The division is upper bounded by the longest division that could occur - namely the highest numerator possible divided by the smallest denominator - since we use a basic, iterative divider. The highest numerator is based on the resolution width, sensor width, focal length, and baseline distance and is around 1000. The smallest disparity is 1, if the two center of masses are directly next to each other (if the disparity is 0, we don't do division and simply output the highest possible value). Therefore, division is capped at ~1000 cycles which fits well within the vsync period. This multi-step process is designed to ensure adherence to timing constraints and maintain computational efficiency, although it could clearly be improved with a more robust division algorithm. Nonetheless, we find our current process to be sufficient for our needs.

As mentioned, the parallax module is used to determine the players depths in order to determine if the players are the correct distance from the camera. The depth information is passed to the game logic controller, which uses the depth to determine if the player is following the rules about the desired depth.

Our parallax module passes our cocotb tests to verify accuracy, and works on our boards using up to 4 centroid values from each board. Because the centroid values are not inherently ordered, the module must figure out a correspondence between centroids from each board. The module does this by checking which centroid is the closest, which we assume corresponds to the same centroid on the other board.

### *B. Center of Mass Transmission via UART*

In order to extract the depth of players via parallax, the center of mass from the separate cameras must be transmitted to a single board. We do this by sending the center of masses from the "Peripheral" board - which ends its computation/pipeline after sending these center of masses - to the "Controller" board - which utilizes the received center of masses to compute depths and continue with the Game Logic and Graphics pipelines - via UART transmission.

There are up to 4 centers of masses calculated on each board, with *x* and *y* components for each center of mass for a total of 8 values that need to be transmitted. To fully utilize the available peripherals on the board as well as minimize latency of the system, we choose to send all 8 center of mass components in parallel across 8 wires. Although the *y* components are only 10 bits wide compared to the *x* bits which are 11 bits wide, we choose to pad the *y* components to 11 bits for simplicity when transmitting since computation has to wait for the *x* bits to arrive as well.

We send our 8 values once a frame after the k-Means algorithm has finished calculating the center of masses from the previous frame. We send our values at a baud rate of 115200 which means the transmission for the 11 bits of data plus 2 bits of start/stop bits takes 11285 cycles. We note that this is significantly under the available (1280 + 220 + 110 + 40) * 30 = 49500 available cycles of vsync between each frame. Additionally, we buffer our received signals across two registers to avoid metastability which adds 2 cycles of unnoticeable latency.

### *C. k-Means*

In broad terms, our algorithm differs from the traditional k-means algorithm in that it does not iterate on the centroids until it reaches a best fit. This would require us to iterate through all player pixels in a camera frame multiple times within the time it takes to display a frame. This is not possible, so we had the idea that perhaps there is enough temporal locality between the player pixels in between frames, that recomputing the centroids across frames by initializing centroids using the previous frames' centroids would be enough to find the best fit.

Early experiments, conducted in software using Python prototypes, helped to validate the design choices underpinning the hardware implementation. In these tests, a simplified k-means routine—one that did not perform multiple iterations over the same data—was shown to converge reliably to stable centroid positions. Even when starting from poor initial guesses, the centroids gravitated toward the correct player locations within a handful of frames. This gave us confidence that our single-pass, frame-by-frame approach would translate effectively to a hardware environment. By avoiding the iterative convergence typically associated with k-means, we achieve a more efficient module that still delivers accurate player tracking results.

The k-means is implemented as a finite state machine (FSM) to ensure reliable and predictable behavior in hardware. The key steps include receiving pixel coordinates tagged as player pixels, assigning those pixels to the nearest centroid, accumulating position data for each centroid, and ultimately updating the centroid positions to reflect the new frame's data.

To identify the closest centroid, the module computes Manhattan distances between the incoming pixel and each active centroid. This corresponds to the closest centroid assignment block in figure 6. Manhattan distance is chosen not only because of its simpler arithmetic operations—composed of a few additions and subtractions instead of square roots—but also because it sufficiently captures spatial proximity in this application. This choice reduces hardware complexity and computation required for each pixel, where timing requirements are tightest. Each pixel, once assigned to a centroid, contributes to that centroid's cumulative position data, which is collected in the COMs (center of mass) block shown in Figure 4. This accumulation takes place over the course of the active draw section of a frame, ensuring that at the end of the frame, the average position of all assigned pixels can be easily computed. The average serves as the updated centroid position, capturing the prevailing player location without multiple iterative passes.

Because the number of active centroids can range from one to four, depending on how many players are present in the game, the FSM logic dynamically adjusts its selection and update routines. If there is only one player, the module maintains a single centroid and updates it with the player's pixel positions. For two, three, or four players, the hardware scales by managing multiple centroid computations in parallel. This flexibility is central to ensuring that the k-means module remains effective and efficient across a range of gameplay scenarios—scenarios that might include players entering or leaving the game mid-session (updating the board switches mid-session would also be necessary, however).

To maintain throughput and synchronization, the centroid calculation is split into two pipeline stages. In the first stage, the module reads incoming pixel coordinates, calculates Manhattan distances to determine the closest centroid, and pipes that information forward. In the second stage, the assigned pixels are aggregated into their assigned centroid. Such parallelism is needed to ensure this module meets timing.

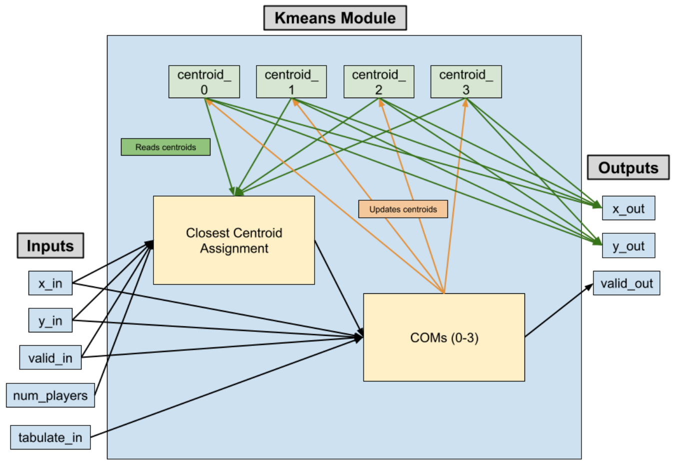

Fig. 6. Kmeans Module Diagram

## III. Game Pipeline

The Camera Pipeline outputs the streamed camera pixels as well as if the given pixel is a player or not. The Game Pipeline accepts this input to update the game state. This component additionally accesses BRAM memory predefined wall shapes are stored for game usage. Using this information, it calculates the following updates to the game state:

- Moving the wall forward as time progresses
- Checking if the player intersect with the wall shape
- When the wall reaches the player, deciding if they collided or not
- Updating the game progress by accessing a new wall shape or ending the game at the end of each round

We expand on how the system implements these tasks below.

### *A. Wall Storage*

Using a Python program that provides a wall generation UI, we generate and store wall shapes in a .mem file used to populate the FPGA BRAM.

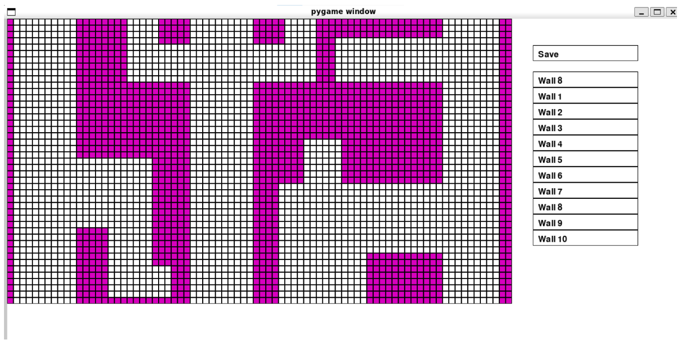

Fig. 7. Screenshot of the wall generation UI

Wall shapes are represented as bitmasks where 1s correspond to walls and 0s to holes in the wall. To avoid excessive memory usage from storing bit masks at the same resolution as the camera input we store bit masks at a 16x lower resolution, i.e. for a 720x1280 video we store 45x80 bitmasks. This resolution provides significant flexibility with wall shapes while allowing for up to 10 wall shapes to be stored per BRAM (with 75 total BRAMs on the board).

With this representation of walls, detecting wall collisions are trivial. The system can simply intersect whether a given pixel was masked to be a player with the upscaled wall bitmask value at the pixel's position - which can be efficiently calculated by dropping the bottom four bits of a pixel's horizontal and vertical position.

Accessing the wall bit masks from memory only happens once a round, so the system waits for the result to return from memory at the end of each round with a small and unnoticeable 2 cycle delay.

### B. *Game Logic*

The game's logic is primarily handled by the Game Logic Controller Module. This component of the system spends the majority of the time advancing the simulated wall at a frequency that ramps as the game progresses. Once the wall reaches the simulated platform in which players are confined to in the real game, the controller monitors for any colliding pixels. If found, the controller outputs a "game over" signal to the graphics pipeline - otherwise, the game continues with a new wall fetched from memory and the speed of the game increases. Once the game has gone through a preset number of rounds, a win screen is displayed.

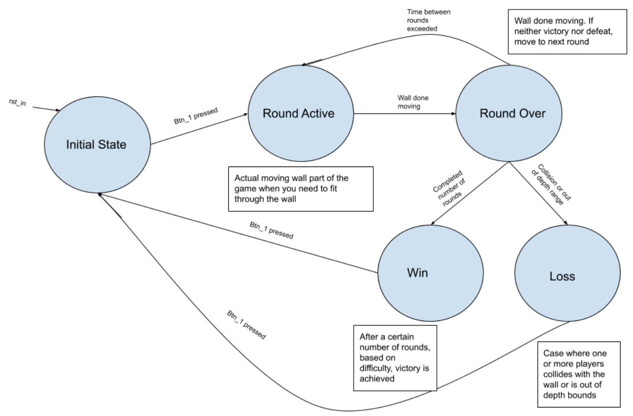
Time between rounds exceeded
Wall done moving. If neither victory nor defeat, move to next round
rst_in
Initial State
Btn_1 pressed
Round Active
Wall done moving
Round Over
Actual moving wall part of the game when you need to fit through the wall
Completed number of rounds
Collision or out of depth range
Btn_1 pressed
Win
Loss
Btn_1 pressed
After a certain number of rounds, based on difficulty, victory is achieved
Case where one or more players collides with the wall or is out of depth bounds

Fig. 8. Diagram of the finite state machine implemented by the game controller

In terms of specifics, the game controller starts a game when button 2 is pressed on the board. Using the selected number of players, corresponding to switches 2 and 3, the round will start with a wall made for the inputted number of players and progress the game. If the players are out of the depth range when the wall reaches them or if the player collides with the wall, the game will end. Note that collisions are determined based on a threshold of a certain number of pixels corresponding to a player intersecting with the wall. This is to account for noise in the mask.

The rounds continue until the third round, which will display the win screen when completed. From the win or loss state, pressing button 2 will take you back to the initial state.

## IV. Graphics Pipeline

As the final step in the system's pipeline, the Graphics Pipeline outputs the updated game state and game visualization to the player via HDMI.

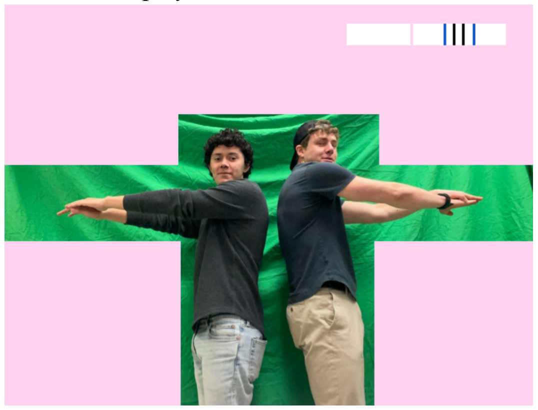

Fig. 9. Artwork demonstrating what the player's view looks like when playing

The Graphics Controller receives game state from the Game Pipeline such as pixel data, collision and wall information, and game state. Given this input, it calculates a pixel value to output by overlaying collision and wall data on top of video data. Note that pixels that are detected to be player-wall collisions are always displayed to the screen even though they only influence the game state when the collision occurs while the wall is in the "play area". Additionally, an indicator of the depth of the wall in relation to the play area is displayed in the top right via a sprite overlaid on top of the screen output.

Figure 8 shows the graphics UI, the pink pixels correspond to the walls and the progress bar is shown in the top right. Note the depth indicator in the top right: the pink vertical line represents the wall's position, which moves to the right as the round progresses. The black lines are the players positions, and the blue lines represent the acceptable range the players need to stand within in order to pass the round.

Figure 8 is a good representation of what the game looks like while being played. Below is an image of the win and loss screens that display when either of those states occur.

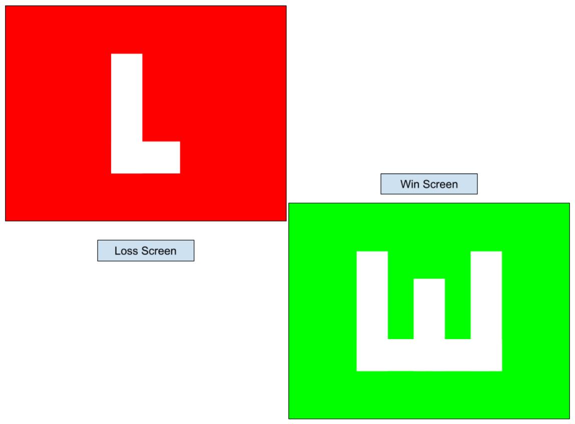

Fig. 10. Loss and Win Screens

Each pixel is then TDMS encoded and sent via HDMI connection to the connected display. From there the players can view the game from the display and follow its prompts to enjoy FPGA hole-in-the-wall!

## V. Evaluation

Our primary form of evaluation for our system are the features and capabilities of our game. We have reached our goals of creating a multiplayer game with UART communication, parallax depth calculations, k-means multi-player tracking, and also game logic for multiple players. We have even reached our stretch goals, making a wall creation script that lets us design walls to play in game. With more time, we would have liked to add more features to make the game feel more like a fully packaged game such as the ability to set difficulty, adding audio, adding a team and versus mode, and general improvements to graphics.That's to say, we have developed a strong foundation for the game with all the main components that could be desired - and all that would be left would be to polish the game with minor gameplay improvements.

Notably, we faced the issue that the player depths extracted from parallax - although generally within the correct range - tend to shift rapidly. We discovered this is because one of our cameras is blue-tinted which makes it noisily and randomly mask the background as if they were players which causes its calculated centroid to shift rapidly. This rapid shift in the transmitted center of mass causes the depth to shift rapidly - although it is generally centered on the correct depth. We would recommend solving this by building different mask thresholds for both cameras.

Additional criteria of evaluation include latency and memory usage. In a game played on a human timescale, latency is negligible as any noticeable latency would have to be very large. Throughout our system, we take advantage of the human timescale of gameplay such as in our k-Means algorithm which assumes that there is little player movement between iterations of the algorithm (i.e. between each frame). At the same time, however, this algorithm allows for center of mass calculations at each frame which is a non-trivial feature. Another instance of negligible latency is the 2 cycle delay on wall bit mask access from BRAM.

Our main timing requirement is completing all end-of-frame calculations, namely calculating and sending centroids as well as calculating depth, between each frame. As mentioned above, there are 49500 cycles of vsync in which we can perform this calculation. Our UART transmission takes 11285 cycles, parallax takes at most ~1000 cycles, and k-Means - which is bounded by division - takes ~10000 cycles. Therefore, we are guaranteed to complete in the available v-sync time.

In terms of clock cycle timing, we look to meet our 10ns clock period timing constraint. Our k-Means module and parallax modules both posed challenges in terms of timing requirements because they both do lots of computation, and also division. This is why both of those modules are pipelined, as without the pipelining they did both cause negative slack. Other sections of our system are primarily passing data with minor computation and, therefore, pose little challenge for meeting timing. In total, we meet timing with .156ns of slack.

In terms of memory usage, the system uses significant memory to buffer video input - however, this is unavoidable for "high" resolution video and several bits of information are already dropped down to a 16 bit 565 pixel format in order to reduce memory usage. The other source of memory usage is the storage of wall bit masks in BRAM. The usage of BRAM is dependent on the number of generated wall shapes - for our 20 walls we use 2 BRAMs. It is worth noting, however, that the memory usage is fairly efficient as up to 7,500 wall shapes can be stored on the 750 BRAMs on the board - and even more if the resolution of the bit masks were to be decreased.

Finally, we would like to point out some implementation insights we would pass along. Simulating our k-Means algorithm in software before implementing it on hardware gave us confidence that our novel algorithm would work as desired. Extensive testing of all our modules additionally significantly helped the development process. Finally, since we were working with two FPGAs with generally similar pipelines, we decided to make a custom build script that utilizes macros to build different parts of the code depending on which FPGA (controller or peripheral) is being built. We strongly recommend this, along with strong use of a version control software, to make development fast and efficient.

## VI. Contributions

Cooper has primarily worked on the k-Means algorithm implementation, parallax computation, green-screen masking, and graphics display. Nathan has primarily worked on the bit mask storage, UART communication, game logic, the custom build script, the wall generation UI, and graphics display. Both have collaborated on the skeleton for the system code, design and planning for the system, write-ups, and assisted each other on each of the modules.

## Acknowledgment

We acknowledge and thank the 6.2050 course staff for instruction and assistance throughout the creation of this project. Specifically, we acknowledge Kiran Vuksanaj and Joe Steinmeyer for mentoring and providing significant input to the system ideation process. Additionally, we acknowledge the use of 6.2050 course material from lab assignments throughout our system.