FPGA Minigolf

Sebastien Lohier
Department of Physics
Massachusetts Institute of Technology
Cambridge, MA, US
slohier@mit.edu

Abhinav Goel
Department of Mathematics
Massachusetts Institute of Technology
Cambridge, MA, US
amgoel@mit.edu

Lawrence Shi
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, US
lrshi@mit.edu

***Abstract*—We present a design for a FPGA-based Minigolf simulator. Players will use a controller to aim, and can then swing the controller to take a swing in the game. The design can be split into three major parts: User Input, Game Logic, and Visualization. The User Input portion is comprised of an ESP32 which is connected to our main FPGA board by BLE. The Game Logic and Visualization modules is done entirely on the FPGA board. Once a user-created map has been selected, the board calculates the state of the game based on the User Input. Then the game is displayed on to a screen using HDMI. An audio experience is also available if headphones are supplied. The screen visualizations will have two modes. A top-down view of the map and an over-the-shoulder perspective view of the ball. 8**

I. Physical Components

Most of the physical design can be attributed to the user input portion of the design. This consists of:

- Adafruit HUZZAH32 ESP32 Feather: This micro-controller computes with the logic involved with collecting user data.
- MPU6050: This peripheral collects accelerometer data which is refined into player input
- Xilinx Urbana Board: The board calculated the game logic and visualization.

II. User Input

A. Wiring

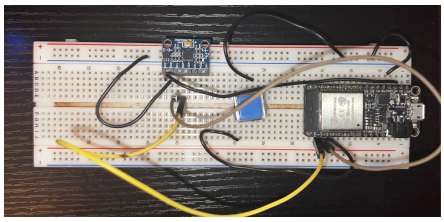

Fig. 1. Image of our controller

The wiring consists of the ESP32, a button, and the MPU6050. The accelerometer is driven using the 3V port of the feather board. It communicates with the board using the SCL and SDA ports. The button is operated with pin 14 acting as a Pull_Down input. The pin constantly reads 1 until the button is pressed and it is connected to ground leading giving a reading of 0. To power the board, we are using a portable charger with a usb port.

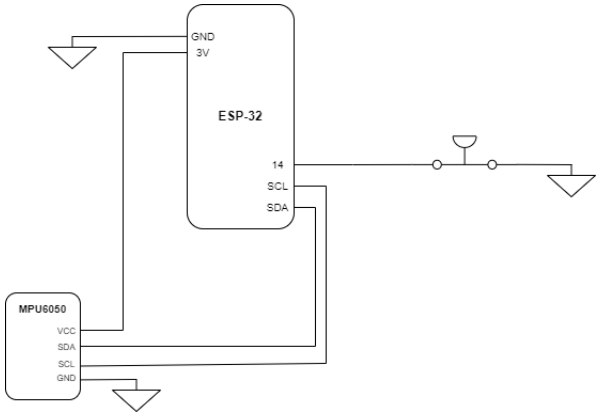

Fig. 2. Wiring diagram of our controller

B. Code

The ESP32 begins by establishing a BLE connection with our main FPGA board. Using the service and characteristic UUID's the board is able to locate the board and begin communicating. During the main loop of the ESP32, it constantly collects accelerometer data from the MPU6050 using an I2C connection. When the button is pressed the micro-controller goes into a "swing mode". While in this mode the board finds the maximum acceleration. When the button is released the ESP32 sends the maximum acceleration, during the duration of the button press, to the FPGA board over the established BLE connection.[1]

C. BLE communication

The Urbana board has a built-in MS50SFB-001 which deals with the intricacies of BLE communication .The module then communicates with the rest of the board using UART communication with a baud rate of 115200. When a UART sequence is completed, our UART communication module triggers a valid input flag which notifies the game of a new user input. This will be ignored unless the Game Logic module is ready for a new user input. In which case the Game Logic will accept the input and move the ball.

[1]Note: The acceleation sent to the board is limited to 8 bits.

D. UART

The UART communicator is implemented as 3 state FSM. This includes an idle state, a collection state, and a processing state. During the idle state the UART module wait for the the UART_TX line to go down. At this point it transitions to the recieving stage where, by polling at the baud rate the module builds up a byte of information. When the transfer is done, the module sets a data_rdy flag to 1 to notify the game logic that a new user input has been registered.

III. Gameplay Logic

A. Map Selection

Each map is represented as a 160 (width) x 90 (height) array of logical pixels. Note that these logical pixels are distinct from the display pixels, where the latter is handled by the visualization logic. Each file contains 160x90 lines, where each line is a single number which represent the terrain at that position. We will use a 0 for the hole, 1 for a square wall, 2 for grass, 3 for sand, and 4-11 for different diagonal walls. The pixels are ordered according to left-to-right, then top-to-bottom priority.

We store these *.mem* files in BROM upon initialization of the map.

The user is able to create their own map using any ASCII art website. They can then use our python script to turn it into a .mem file. Finally they can build and flash it onto the board to play.

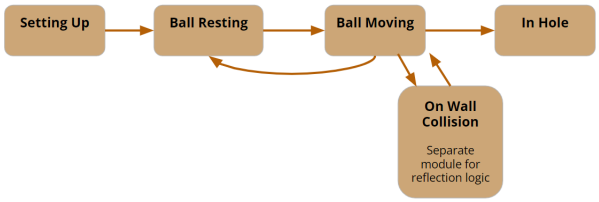

Fig. 3. FSM of gameplay logic.

B. Ball Movement

We represent the game state as an FSM, as shown in Figure 1. Upon launching the game, after doing some visual setup, the ball starts in the Ball Resting state. In this state, the state machine waits for a user input. Once the user input module detects player input via Bluetooth, the state transitions to the Ball Moving state.

Once the ball is moving, the state can transition to either Ball Resting, In Hole, or On Wall Collision. In the Ball Moving state, the ball decelerates at a fixed rate depending on the terrain type (grass or sand). Our grass deceleration value is $-2 * \frac{1}{256} \frac{pixels}{frame^2}$, and our sand deceleration value is $-10 * \frac{1}{256} \frac{pixels}{frame^2}$, where the game is running at 60 frames per second. These values correspond to visually reasonable deceleration.

The x and y positions are updated at every frame according to the ball position at the previous frame, ball speed, and ball direction. To calculate the horizontal and vertical components of the position deltas, we use Python to generate a **.mem** file containing 360 $\cos$ and $\sin$ values, and we wrap that in a cos/sin lookup module that builds and reads from a BROM using those values.

C. Direction, Speed, and Collisions

The direction is encoded as a 16-bit number for an angle (between 0 and 360), which is precise enough for gameplay purposes. However, we need to be more precise for the ball speed and position than numbers of logical pixels. We choose to represent these as 16-bit fixed point numbers, with 8 bits before the decimal point and 8 bits after the decimal point. This ensures that we can obtain a precision of up to $1/256$ of a pixel while still being able to represent all possible x and y values using the remaining 8 bits.

To calculate collisions, we see if any part of the ball is overlapping with any wall terrain. We do this by querying the copies of our map BROM 1 pixel ahead in all 4 directions. If any of these 4 queries are on a wall, we know we have touched a wall on that side. The incidence angle is then passed to a reflection logic module, which calculates and modifies the resulting direction of the ball to correctly simulate a reflection off of that wall. Then, the gameplay transitions back to the Ball Moving state.

We have also implemented 45-degree diagonal walls. The walls are detected in the same fashion as orthogonal walls, but the incidence and reflection angles are calculated differently.

IV. Visualization and Audio

There are two different ways to construct a visualization of the game based on the provided information from the game logic.

A. Static View

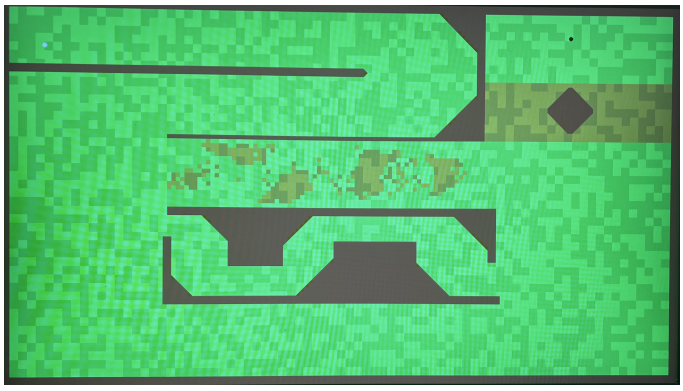

Fig. 4. Top-down view

The first way consists of a static view, where the whole course is shown on the screen, as well as the current position of the ball, and the current trajectory of the ball.

As shown above, the map of the current mini-golf hole is represented by a 160 by 90 map, with 90 horizontal rows, and 160 columns. Thus, each position in the map is represented by an 8 by 8 pixel square. Additionally, the visualization is given the ball position (16 bits each for x and y position, with 8 bits before and after the decimal point), as well as the angle of ball, which would be an integer from 1 to 360, in degrees.

A *sprite module*, given pixel coordinates, calculates the RGB values for each pixel. This process consists of a few steps. Dividing both coordinates by 8 yields the position of the map the pixel would be a part of, and then reading from the map,from BROM, yields the RGB value of the pixel. Essentially, this version is just a shrunken view of the whole map.

However, we also need to show the ball. We have the position of the center of the ball, but with much more precision than just integer coordinates. The original idea was to just use integer coordinates in the 160 by 90 map, and to turn the corresponding 8 by 8 square of pixels into the ball. However, then the ball movement was not fluid, so we set a pixel as the center of the square. Given ball position ballx[15:0] and bally[15:0], the pixel that is the center of the ball is approximated as ballx[15:5] and bally[14:5]. Once we have this, we construct the ball as shown in Figure 4 to appear round. We construct the hole to be round in a similar manner.

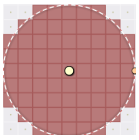

Fig. 5. Ball and Hole Construction

Additionally, we also have a trajectory for the ball. This is done by finding the pixels a distance of 30, 60, and 90 away from the center of the ball in current direction given. We find the sin and $cosine$ of our angle using the lookup table, and can then calculate the component in each direction. We had to pipeline 4 cycles for calculating RGB values above, so this lookup table does not use any additional cycles.

Thus, this completes the static version of the map. It takes the map file, and the current state of the game, and displays that position. By updating the map files and the palette files, additional features can be added, which could seamlessly use without updates to this module. This visualization for a very simple map is shown in Figure 5.

B. Dynamic View

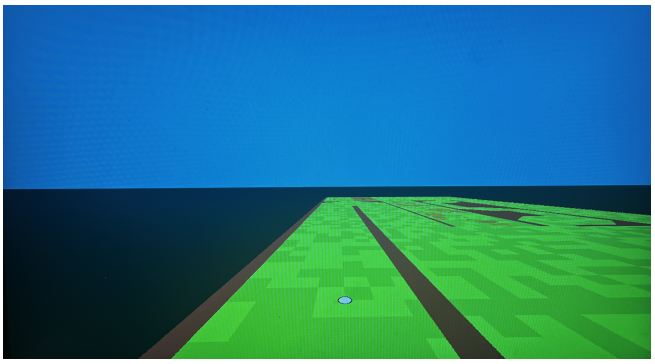

Fig. 6. Over the shoulder perspective

The basis of our the Dynamic View is a transformation of form traditional Field-of-Vision to the 2D screen of the monitor. We begin by mapping the top half of the screen to the sky, to give the user a sense of perspective. The bottom half of the screen will be the golf course. We want to map each pixel to some position on our map, which will then correspond to some RGB value.

The calculation begins by first defining the position of our camera. We do this by placing the camera directly behind the position of the ball. The actually position is determined by the cosine and sine of our view angle. The cosine/sine values are stored within a BROM which is accessed 2 cycles after the each query. We use these same cosine/sine values to calculate our field of vision.

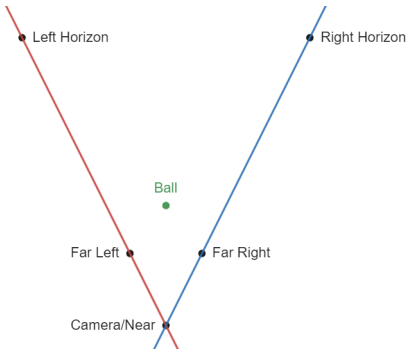

Fig. 7. Mathmatical Model for visualization

Consider the graphic above. Our perspective will be at the position of the Camera, which is determined by placing the camera directly behind the position of the ball. Directly behind means in the opposite direction of the view angle. Additionally, we have the points Far Left and Far Right in our diagram, which correspond to the the furthest distance of points that we cannot see.

Essentially, the triangle formed by the Camera (also called Near), Far Left, and Far Right are points that are too close to be

seen properly. (The diagram is not to scale, as this triangle is actually very small). Our screen will consist of all the points inside the trapezoid with vertices Far Left Far Right, Right Horizon, and Left Horizon. To do this, we will take each row of our screen, and match it with some horizontal row going from the red left edge to the blue right edge. We will scale it appropriately to match with the 1280 pixels per row of the screen. We make some assumption about how close Far Left and Far Right will be from the camera, and set it to be our bottom row of the screen. We will proceed to halfway up the screen, which will be the row from Left Horizon to Right Horizon (which will be mathematically defined in terms of the camera position, Far left, and Far right).

Now, the triangle in our diagram has vertex angle 110 degrees. This means that from our viewing angle (call it θ, we can see 55 degrees to the left and to the right. We take two cycles to learn the sin and cosine values of the three angles θ, $\theta + 55$, and $\theta - 55$. This allows us to calculate the coordinates of any point.

To calculate the coordinate of any point, we first find the map coordinate of the first and last pixels of the same row. Then, by taking a weighted average of those coordinates we determine the map coordinates of a pixel. We refer to the magnitude of a row to be the distance from the camera to the point on the edge of the row. The near point has magnitude 0, which means that it is just the position of our camera.

Now, we can use fixed point arithmetic to calculate the x and y coordinates of each of the far positions.

$$\text{f}ar\_left_x = \text{c}amera\_pos_x - (\cos(\theta + 55) \cdot \text{f}ar\_mag)$$

$$\text{f}ar\_left_y = \text{c}amera\_pos_y - (\sin(\theta + 55) * \cdot \text{f}ar\_mag)$$

$$\text{f}ar\_right_x = \text{c}amera\_pos_x - (\cos(\theta - 55) * \text{f}ar\_mag)$$

$$\text{f}ar\_right_y = \text{c}amera\_pos_y - (\sin(\theta + 55) * \text{f}ar\_mag)$$

Where θ is the direction the player is looking. Now, we calculate the map position of the corresponding coordinate when given some vcount and hcount representing a pixel on the screen. Having the camera positions as well as far coordinates helps because they define a set ratio that all points must then follow. We follow the corresponding formula

$$\text{l}_x = \frac{360}{360 - \text{v}count} \cdot (\text{f}ar\_left_x) + (1 - \frac{360}{360 - \text{v}count}) \cdot \text{n}ear_x$$

$$\text{r}_x = \frac{360}{360 - \text{v}count} \cdot (\text{f}ar\_right_x) + (1 - \frac{360}{360 - \text{v}count}) \cdot \text{n}ear_x$$

$$x = \frac{\text{h}count}{1280} \cdot \text{l}_x + \frac{1280 - \text{v}count}{1280} \cdot \text{r}_x$$

The same is done for the y coordinate. This works because we see that the density of points further away has an inverse relationship with how close we are to the camera. Thus, the golf course itself, which would not be too far from the camera, would be heavily represented with vcount closer to the bottom of the screen. We then just take a weighted average to find the exact point.For reference note the left and right horizon points are generated when $\text{v}count = 361$. And the Far left and Far right values are generated when vcount = 720

We also allowed the adjustment of the far_mag value to give the user control of their view height. This way players could move up and down in their current position.

Now, once the map location attributed to a pixel has been calculated, we can just proceed in a similar fashion to the static module by using the Map BROM. However, there is one more layer of complexity as the calculation of the ball is non-trivial and a $\text{map}_c oordinate might not be positive. Thus, we generate a border of 223 un$

Due to the magnitude of the calculations required to generate our perspective view, it cannot be done in one cycle. To circumvent this, it has been pipelined so that it takes around 50 cycles for a coordinate to be generated given some hcount and vcount.This is partially due to the size of our position variables. To ensure no overflow our positions are 40 bit fixed point numbers. More importantly divisions are the most costly, so only one was done, which was $\frac{360}{360 - \text{v}count}$, where a 40 cycle divider was used. Moreover, large multiplications were pipelined into 6 cycles.

Once this part was done, a ball sprite was inserted into the code to appear on the pixels corresponding to the ball position, as well as touching up the hole to make it appear circular.

Additionally, to the score is also displayed on the FPGA using the seven_segment_controller implemented in an earlier lab.

C. Additional Visual Features

We created grass and sand texture to make the 2D and 3D representations look more realistic and polished. This was accomplished by pseudo-randomly mapping each 2x2 cluster of pixels to one of two colors using a 16-bit linear feedback shift register (LFSR). This LFSR runs through all pixels once when the map is initialized. This mapping is stored in a BROM that lives in the top level module whose inputs and outputs are passed down to child modules.

D. Audio

Audio clips for the ball hitting a wall and the ball going into the hole are played whenever these events happen. The raw audio clips (in .wav form) were downloaded online, and we used Python to convert these files to an 8-bit, 12kbps format. This data is stored in .mem files for the FPGA to use. We have a module that handles Pulse-density modulation (PDM) is used to generate the speaker audio.

V. Discussion and Evaluation

1) Latency Analysis: In terms of the latency of our game, the largest delay paths came from the calculations needed for our over-the-shoulder rendering calculations. The other delay paths were trivial in magintude. The rendering required 40-bit multiplications and division. Using separate division and multiplication modules, we were able to do each in 40 and 6 cycles, respectively. Additionally, each of our BROM accesses took 2 cycles. Putting this all together, our 3D rendering module took a total of 50 cycles. To align this module's output with the required HDMI signals, we pipelined our top level module accordingly.

We found this overhead unavoidable; without the heavy cost of our multiplications and divisions, we would not be able to make a realistic 3D rendering of our map. However, the latency is negligible because the gameplay runs at 60 fps, which is several orders of magnitude slower than our clock.

2) RAM Usage: We have 4 types of BROMs within the design. The first of which was used to store Cosine/Sine values to be used for angle calculations. These BROMS had a width 16 bits and a depth of 361. This BROM was used a total of 8 times. This works out to a total bit cost of 5.776 KB. Our second BROM was used to store the map information. This BROM had a width of 4 bits and a depth of 14400 (160 width by 90 height). This BROM was used a total of 6 times for a total BROM usage of 43.2 KB. Third, we used a BROM to store the randomized texture data. This BROM had a width of 1 bit and a depth of 7200. This BROM was used twice for a total usage of 1.8 KB. Fourth, 2 BROM's were used to store the audio data. Each with a width of 8 bits. One had a depth of 65536 and other has a depth of 9600. For a total usage of 75.135KB. Our total BROM usage totals to 125.911 KB.

Looking at the post_place_util file, our design has a utilization percentage of 90.67% and our DSP utilization 30.85%.

3) Timing Constraints: Due to our design choice of not using a frame buffer, our logic was constrained to the clock speed of the HDMI signal generator, which is 74.25 MHz. We got around this by using pipelined modules for our divisions and multiplications. We met this timing constraint, with a positive slack of.

Additionally due to the interface of our game logic and our audio logic we cross from a faster to slower domain to mediate this we put two registers in between them to prevent metastability of the registers.

4) Use Cases and Goals: Our system was built for the purpose of simulating mini-golf. While the 2-dimensional view and playing with buttons on the FPGA allowed people to play the game, the main aspect that should have be captured is to resemble golfing in real life. This was accomplished this by adding the bluetooth controller on the golf club, as well as over-the-shoulder perspective.

The commitment goal was to complete a simple prototype of a golf simulator, which could be player solely on an FPGA with a 2-dimensional view, which was completed. The goal was to have a controller that would use bluetooth, as well as a realistic over-the-shoulder perspective, which both took some time, but were completed. Essentially, these goals were to create the game, and then make it realistic.

The stretch goals were to add additional features to the game. Some of these were completed like adding texture to the map, as well as adding diagonal walls with appropriate collisions. There were additional features, such as portals, that could have been added to the map itself that we did not implement. However, these changes would not be very difficult, as maps could easily be changed to add more features, and this could be reflected in the game logic by adding another case statement about how the game logic should change if such an element is encountered. Another feature that could be added is allowing players to customize their experience by being able to choose different types clubs. The only code that would be needed is game state changing the maximum power parameter per club.

One other use case we could implement with our system would be to make the game multiplayer (two people play against one another on the same system). This would not take too many changes, as mostly the game logic would have to store game state values for both players and switch between them appropriately. The controller would work as usual and the visualization would show whatever the game state would tell it.

Our most in-depth implementation would be that of the over-the-shoulder visualization. From the beginning, it would have to use ray-casting, but having a closed rectangle meant having issues in hitting walls and corners. Using an infinite boundary allowed going in all directions for an unlimited distance, and setting "undefined" points identically to the boundary. This allowed for the same algorithm to be used for all points.

One feature we realized could be done in a better way is being able to read multiple maps from our BROM. Currently, the system is set to only play one map, and so multiple maps cannot be played in a single build, even though they are written. We could change this by merging all maps into a single file, and having all BROMs read from that single file. The game state could store some map variable which would be used in the BROM address to determine position should be read from for a specific map. .

A. Contributions

The user input, which includes the construction of the controller, the code of the controller, and the communication with the FPGA, was done by Sebastien. The gameplay, which consists of ball position and speed calculation and tracking, as well as interactions with various map elements, was done by Lawrence. For the visualization, the static view was done by Abhinav, while the over-the-shoulder view was done by both Abhinav and Sebastien. Additional features like texture and audio were added by Lawrence, while diagonal walls were added by Sebastien. All members contributed in writing maps,

pipelining the code to fix timing errors, and writing the report.

B. Code Repository

https://github.com/Gflex39/6.111_fin_proj

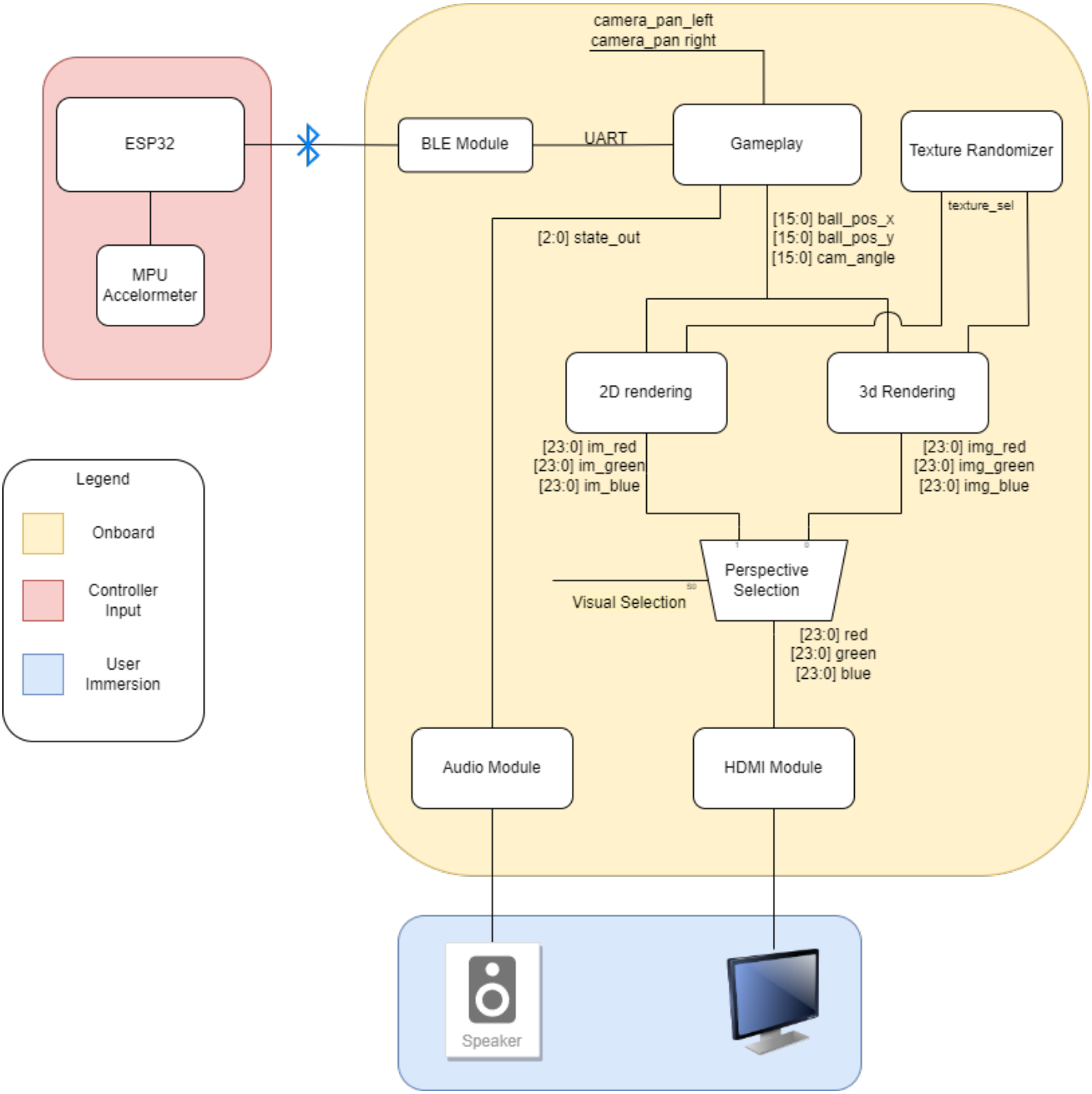

Fig. 8. Total system design