CSEE 4840
Embedded Systems -
Project Design Document

Project Title: Air Hockey

Group Members:

Gerald Zhao - zz3427

Derrick Chen - dc3494

Zening Wang - zw3161

Xuepeng Han - xh2718

Spring 2026

Contents

1. Introduction 3
2. System Block Diagram 4
3. Algorithms 5
3.1 Uninterrupted Movement 5
3.2 Colliding with a Wall 5
3.3 Colliding with a Paddle 7
3.3.1 Time of Collision 7
3.3.2 Resulting Velocities 8
3.3.3 Post-Collision Vector 10
3.4 Scoring a Goal 12
4. Resource Budgets 13
4.1 Resource Budgets Table 13
5. The Hardware/Software Interface 14
5.1 Interface Design Overview 14
5.1.1 Interface Philosophy 14
5.1.2 Information Passed Between Software and Hardware 14
5.1.3 Register Organization 15
5.1.4 Coordinate Encoding and Frame Data Format 15
5.1.5 Frame Update Model 15
5.1.6 Synchronization Mechanism 15
5.2 Register Map Overview 16
5.3 Registers Definition 17
5.3.1 STATUS Register (0x00) 17
5.3.2 FRAME_COUNTER Register (0x04) 17
5.3.3 PLAYER_1 Position Register (0x08) 18
5.3.4 PLAYER_2 Position Register (0x0C) 18
5.3.5 PUCK_POSITION Register (0x10) 19
5.3.6 SCORE Register (0x14) 19
5.3.8 SOUND_CONTROL Register (0x18) 20
6. Input Path and Protocol 21
6.1 Software Stack 21
6.2 USB-Level Protocol 21
6.3 Linux Event Conversion 22
6.4 Linux Event Conversion 22
6.5 Input Rate Versus Game Update Rate 23

1 Introduction

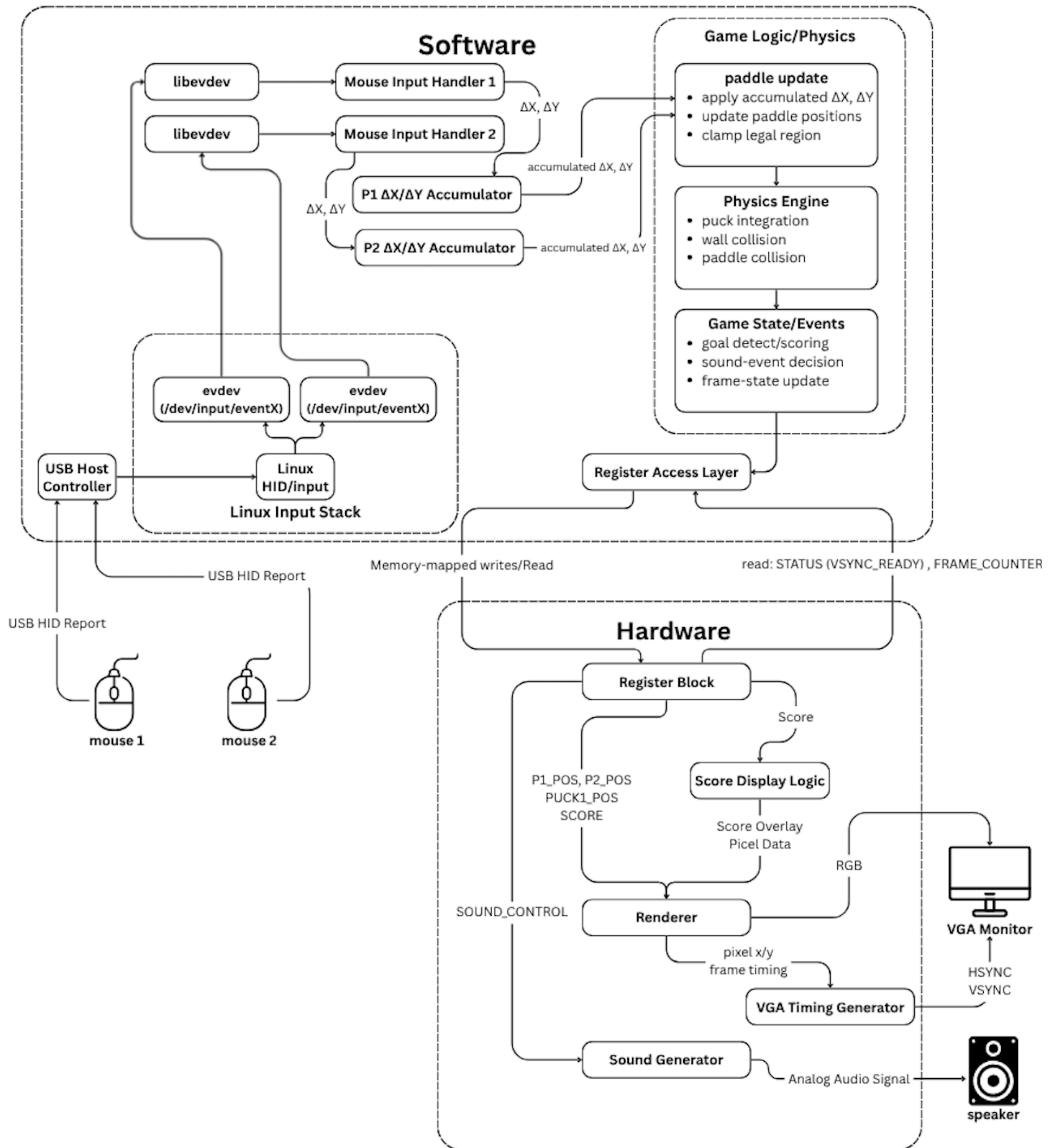
This project involves the implementation of a real-time, two-player digital Air Hockey arcade game on an FPGA SoC platform. To emulate the experience of a traditional arcade, the system is designed to output a 2D top-down perspective to a large flat-panel display laid horizontally. Two players will stand on opposite sides and compete against each other. The gameplay includes continuous paddle control, 2D vector collision physics, synchronized audio feedback for impacts and scoring, and a match conclusion when either player reaches seven points.

Fundamentally, the system operates as a distributed algorithm divided into a software-driven game engine and a hardware-driven presentation. The software, running in Linux, will act as the brain. It will poll independent raw USB mouse data, calculate floating-point inelastic collision physics, and manage the game state. The hardware will act as the canvas and speaker. It guarantees a 60 Hz timing requirement of VGA video generation and audio, translating software-provided coordinates into on-screen pixels without interrupting the processor's physics calculations.

The remainder of the document will detail the technical implementations of the system:

- **System Block Diagram**: Illustration of the high-level architecture and dataflow.
- **Algorithms**: Details the floating-point vector mathematics and physics used for inelastic collisions (puck-to-paddle, puck-to-wall) and the state-machine logic for scoring.
- **Resource Budgets**: Estimates the logical elements, memory blocks, and processing bandwidth required to sustain the game in a 60 Hz loop.
- **The Hardware/Software Interface**: Defines the strict memory-mapped register contract and VSYNC communication protocol that bridges the software and hardware.

2. System Block Diagram



3. Algorithms

This system operates as a distributed architecture. The complex game logic and calculations are computed in the software. Meanwhile, the pixel rendering and audio processing are executed as hardware algorithms on the FPGA.

The game loop runs at a fixed 60 Hz (16.67 ms). We parameterize the puck's movement using a time variable t, where $t = 0$ is the start of the frame, and $t = 1$ is the end of the frame.

Our game will be displayed on a 640×480 pixel screen. The origin $(0, 0)$ will be located at the top-left corner of the screen. Y will increment downwards, while X increments towards the right. The wall boundaries will have a thickness of 10 pixels, effectively giving us a playable area of 620×460.

3.1 Uninterrupted Movement

During any given frame, the puck travels along a line. We define its position as a function of t where $0 \leq t \leq 1$.

$$X_{puck}(t) = X_{old} + V_x t$$

$$Y_{puck}(t) = Y_{old} + V_y t$$

Assuming that no collisions occur during this time frame, the puck's final position is simply when $t = 1$, thus sending

$$(X_{new}, Y_{new}) = (X_{puck}(1), Y_{puck}(1))$$

as the new coordinate.

3.2 Colliding with a Wall

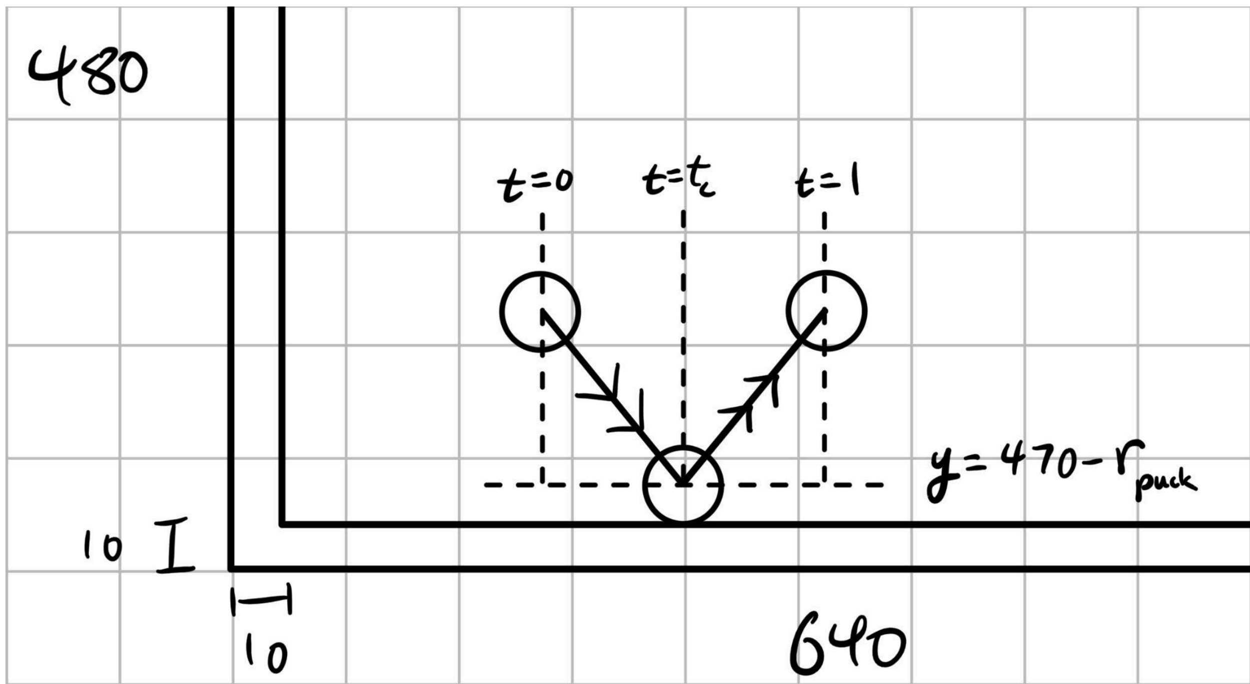
The puck is bounded by the lines $Y = 10$, $Y = 470$, $X = 10$, and $X = 630$. Assuming that we know the puck's current position (X_{old}, Y_{old}), as well as its current velocity (V_x, V_y), we can calculate what happens when it collides with a wall. Note that we are also assuming the puck does not collide with any paddle during this time frame. This means we can effectively represent a collision against the wall as two separate cases of uninterrupted movement.

For example, the bottom wall is the boundary $Y = 470$. A collision here occurs when

$$Y_{old} + V_y t_c = 470 - r_{puck}$$

Solving for the time of collision, we find

$$t_c = \frac{470 - r_{puck} - Y_{old}}{V_y}$$



The point of collision is then

$$X_{collision} = X_{old} + V_x t_c$$

$$Y_{collision} = Y_{old} + V_y t_c$$

We apply a coefficient of restitution, C_R, to simulate energy loss during the collision. The velocity in the Y direction also flips signs. This is simply

$$V_{x_{new}} = V_{x_{old}} C_R$$

$$V_{y_{new}} = - V_{y_{old}} C_R$$

Finally, the puck moves away from the wall after the collision, with

$$t_{remaining} = 1 - t_c$$

$$X_{new} = X_{collision} + V_{x_{new}} t_{remaining}$$

$$Y_{new} = Y_{collision} + V_{y_{new}} t_{remaining}$$

For the other three walls, the calculation is relatively the same.

- Top Wall: Check $Y_{old} + V_y t_c = 10 + r_{puck}$, flipping V_y.
- Left Wall: Check $X_{old} + V_x t_c = 10 + r_{puck}$, flipping V_x.
- Right Wall: Check $X_{old} + V_x t_c = 630 - r_{puck}$, flipping V_x.

3.3 Colliding with a Paddle

3.3.1 Time of Collision

To find the time of the collision between the puck and the paddle, we can approach this by making the paddle “stationary” relative to the puck. In other words, we calculate the relative position and velocity of the puck from the paddle’s perspective. Note that this is only to solve for t_c and that the resulting velocities after the collision will be calculated differently.

$$\Delta P_x = X_{puck_{old}} - X_{paddle_{old}}$$

$$\Delta P_y = Y_{puck_{old}} - Y_{paddle_{old}}$$

$$\Delta V_x = V_{x_{puck_{old}}} - V_{x_{paddle_{old}}}$$

$$\Delta V_y = V_{y_{puck_{old}}} - V_{y_{paddle_{old}}}$$

A collision between the puck and paddle occurs if the squared distance between their center points is equal to the square of their combined radii: $R_{sum} = R_{puck} + R_{paddle}$.

If we track the puck’s relative movement over time, we can find the exact time t_c of the collision.

The relative position between the two centers at any given time is

$$P(t) = \Delta P + \Delta V \cdot t$$

The collision occurs when this magnitude is equal to the combined radii

$$||P(t)|| = ||\Delta P + \Delta V \cdot t|| = R_{sum}$$

By looking at the squared distance, we can see that

$$||\Delta P + \Delta V \cdot t||^2 = (\Delta P + \Delta V \cdot t)^2 = (R_{sum})^2$$

Expanding, we find

$$(\Delta V \cdot \Delta V)t^2 + 2(\Delta P \cdot \Delta V)t + (\Delta P \cdot \Delta P) = (R_{sum})^2$$

$$(\Delta V \cdot \Delta V)t^2 + 2(\Delta P \cdot \Delta V)t + (\Delta P \cdot \Delta P) - (R_{sum})^2 = 0$$

Let

$$A = (\Delta V_x)^2 + (\Delta V_y)^2$$

$$B = 2 \times [(\Delta P_x \cdot \Delta V_x) + (\Delta P_y \cdot \Delta V_y)]$$

$$C = (\Delta P_x)^2 + (\Delta P_y)^2 - (R_{sum})^2$$

Then we can solve this quadratic equation, finding

$$t_c = \frac{-B - \sqrt{B^2 - 4AC}}{2A}$$

Similarly to the case of colliding with the call, if $0 \leq t_c \leq 1$, the collision occurs within the current time frame.

3.3.2 Resulting Velocities

In order to calculate the resulting velocity after the collision, we will treat the paddle as an object with infinite mass. This approach is reasonable since the "push-back" felt on the paddle by the puck will never affect the player.

Consider the equations for conservation of momentum and restitution:

$$m_{puck} v_{puck_{old}} + m_{paddle} v_{paddle_{old}} = m_{puck} v_{puck_{new}} + m_{paddle} v_{paddle_{new}}$$

$$e = \frac{v_{paddle_{new}} - v_{puck_{new}}}{v_{puck_{old}} - v_{paddle_{old}}}$$

Note that v is a vector. Additionally, e measures the elasticity of the collision. We will use these two known equations to solve for two variables, $v_{puck_{new}}$ and $v_{paddle_{new}}$.

$$v_{paddle_{new}} - v_{puck_{new}} = e(v_{puck_{old}} - v_{paddle_{old}})$$

$$v_{puck_{new}} = v_{paddle_{new}} - e(v_{puck_{old}} - v_{paddle_{old}})$$

Plugging this into the momentum equation:

$$m_{puck} v_{puck_{old}} + m_{paddle} v_{paddle_{old}} = m_{puck}(v_{paddle_{new}} - e(v_{puck_{old}} - v_{paddle_{old}})) + m_{paddle} v_{paddle_{new}}$$

$$m_{puck}v_{puck_{old}} + m_{paddle}v_{paddle_{old}} = (m_{puck} + m_{paddle})v_{paddle_{new}} - m_{puck}e(v_{puck_{old}} - v_{paddle_{old}})$$

$$(m_{puck} + m_{paddle})v_{paddle_{new}} = m_{puck}v_{puck_{old}} + m_{paddle}v_{paddle_{old}} + m_{puck}e(v_{puck_{old}} - v_{paddle_{old}})$$

$$v_{paddle_{new}} = \frac{m_{puck}v_{puck_{old}} + m_{paddle}v_{paddle_{old}} + m_{puck}e(v_{puck_{old}} - v_{paddle_{old}})}{m_{puck} + m_{paddle}}$$

Dividing the top and bottom by m_{paddle}, we get

$$v_{paddle_{new}} = \frac{\frac{m_{puck}}{m_{paddle}}v_{puck_{old}} + v_{paddle_{old}} + \frac{m_{puck}}{m_{paddle}}e(v_{puck_{old}} - v_{paddle_{old}})}{\frac{m_{puck}}{m_{paddle}} + 1}$$

Since the paddle is being treated as infinite mass, we take $m_{paddle} \to \infty$, meaning $\frac{m_{puck}}{m_{paddle}} \to 0$ .This leaves us with

$$v_{paddle_{new}} = \frac{0 + v_{paddle_{old}} + 0}{0+1} = v_{paddle_{old}}$$

as expected.

To find $v_{puck_{new}}$, we simply go back to our earlier equation

$$v_{puck_{new}} = v_{paddle_{new}} - e(v_{puck_{old}} - v_{paddle_{old}})$$

$$v_{puck_{new}} = v_{paddle_{old}} - e(v_{puck_{old}} - v_{paddle_{old}})$$

$$v_{puck_{new}} = (1 + e)v_{paddle_{old}} - e(v_{puck_{old}})$$

Since we know the previous velocities for both the puck and the paddle, we can calculate the new velocities using these equations.

Note that if our system is completely elastic, we take $e = 1$, giving us

$$v_{puck_{new}} = 2v_{paddle_{old}} - v_{puck_{old}}$$

$$v_{paddle_{new}} = v_{paddle_{old}}$$

3.3.3 Post-Collision Vector

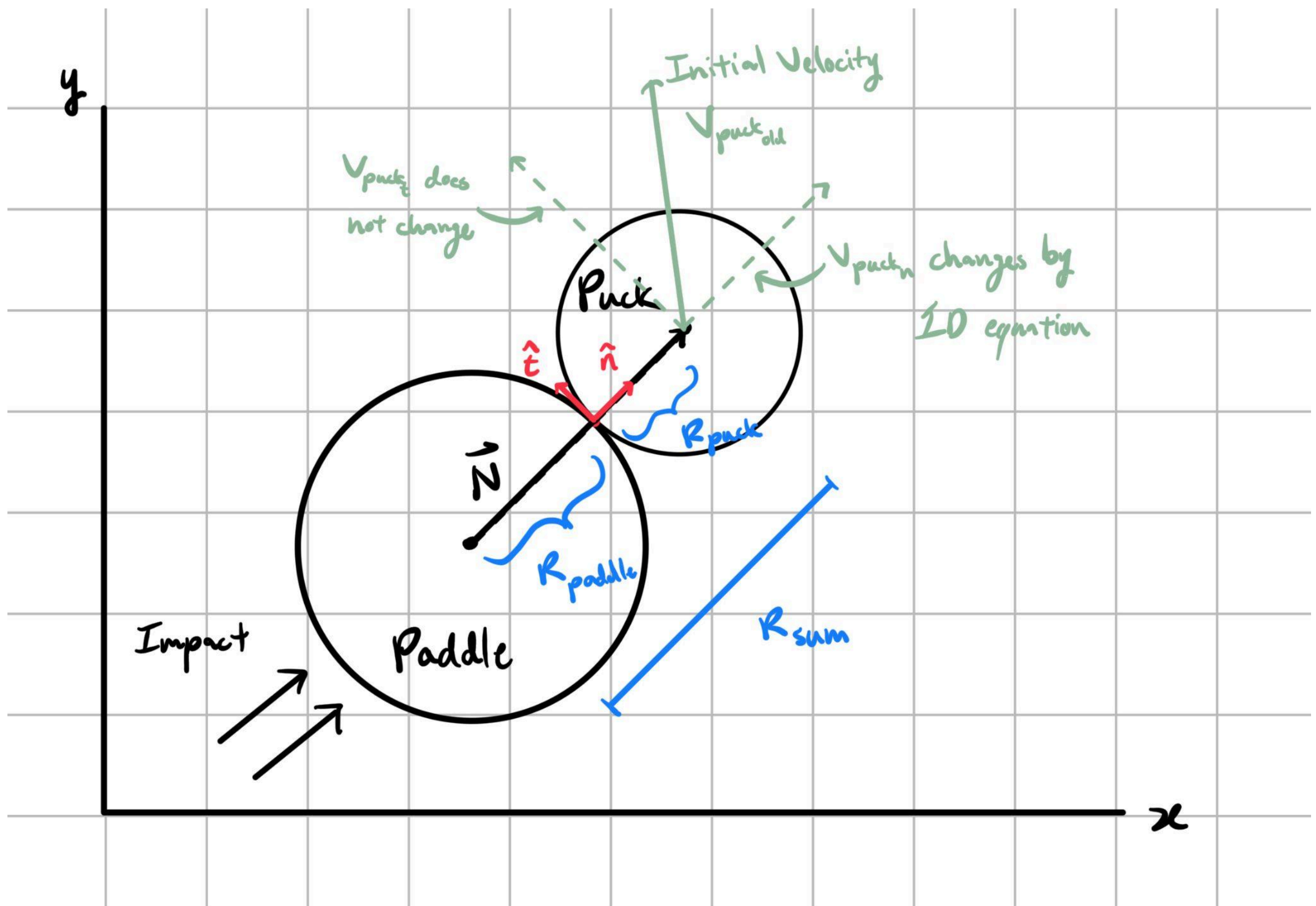
We first calculate the absolute global coordinates of both objects at the exact moment of impact. Note that the velocities here are the velocities before collision:

$$X_{puck_{impact}} = X_{puck_{old}} + V_{x_{puck_{old}}} \cdot t_c$$

$$Y_{puck_{impact}} = Y_{puck_{old}} + V_{y_{puck_{old}}} \cdot t_c$$

$$X_{paddle_{impact}} = X_{paddle_{old}} + V_{x_{paddle_{old}}} \cdot t_c$$

$$Y_{paddle_{impact}} = Y_{paddle_{old}} + V_{y_{paddle_{old}}} \cdot t_c$$



The collision force acts along the position vector pointing from the paddle's center to the puck's center. We define this as the position vector N, where

$$N_x = X_{puck_{impact}} - X_{paddle_{impact}}$$

$$N_y = Y_{puck_{impact}} - Y_{paddle_{impact}}$$

The magnitude of N is equal to the length of the vector, which is just the sum of the two radii, R_{sum}. This means we can find a normal unit vector $\widehat{n}$ where

$$\widehat{n}_x = \frac{N_x}{R_{sum}}, \qquad \widehat{n}_y = \frac{N_y}{R_{sum}}$$

The tangent vector $\widehat{t}$ is the vector orthogonal to $\widehat{n}$. More specifically, it's defined as

$$\widehat{t}_x = -\widehat{n}_y, \qquad \widehat{t}_y = \widehat{n}_x$$

The equation we derived earlier only works in 1D, meaning it's used for head-on collisions. Since our game is in 2D, we can create a temporary coordinate system for the collision:

- Normal axis $\widehat{n}$: our new x-axis, cuts through the centers of both circles and the point of collision.
- Tangent axis $\widehat{t}$: our new y-axis, perpendicular to the normal axis.

Along the normal axis, the system behaves exactly as a 1D head-on collision, meaning we can use the formula that we derived earlier.

Along the tangential axis, because there is no friction, the interaction does not change the velocity in this direction.

Let u be any velocity prior to impact, and v be any velocity after impact. Then,

$$u_{puck_n} = (V_{x_{puck_{old}}} \cdot \widehat{n}_x) + (V_{y_{puck_{old}}} \cdot \widehat{n}_y)$$

$$u_{puck_t} = (V_{x_{puck_{old}}} \cdot \widehat{t}_x) + (V_{y_{puck_{old}}} \cdot \widehat{t}_y)$$

$$u_{paddle_n} = (V_{x_{paddle_{old}}} \cdot \widehat{n}_x) + (V_{y_{paddle_{old}}} \cdot \widehat{n}_y)$$

Note that we don't need the paddle's tangential velocity since it does not transfer force to the puck along that axis. We can now see that

$$v_{puck_n} = 2u_{paddle_n} - u_{puck_n}$$

$$v_{puck_t} = u_{puck_t}$$

Converting this back to global X and Y coordinates, we have

$$v_{puck_x} = (v_{puck_n} \cdot \widehat{n_x}) + (v_{puck_t} \cdot \widehat{t_x})$$

$$v_{puck_y} = (v_{puck_n} \cdot \widehat{n_y}) + (v_{puck_t} \cdot \widehat{t_y})$$

Now that we've established a new velocity vector for the puck, we know that the puck travels along this path for the rest of the time frame $t_{remaining} = 1 - t_c$.

The final coordinates are then

$$X_{puck_{new}} = X_{puck_{impact}} + v_{puck_x} t_{remaining}$$

$$Y_{puck_{new}} = Y_{puck_{impact}} + v_{puck_y} t_{remaining}$$

3.4 Scoring a Goal

Along with the standard bouncing-off-a-wall logic, one consideration that we must make along the left and right walls is when a goal is scored. For example, if we decide that our goals are 120 pixels large, then the goal openings exist between $Y = 180$ and $Y = 300$. To know if we score a goal, we need to check if the puck's Y-coordinate at the time of boundary impact falls within the goal's opening.

We calculate the hypothetical Y-coordinate of the puck at the exact moment it intersects the left boundary plane ($X = 10$). Using the same math from earlier, we have

$$t_c = \frac{10 + r_{puck} - X_{old}}{V_x}$$

$$Y_{collision} = Y_{old} + V_y t_c$$

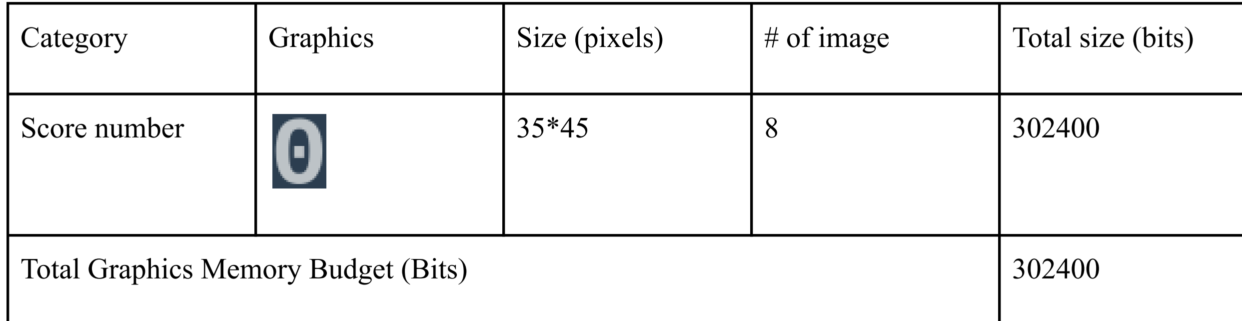
Once we have $Y_{collision}$, we must determine if the puck enters the goal, hits the wall, or hits the corner of the goal opening. To handle the edge case of hitting the corner of the goal, or within the 10 pixel border, we model the top and bottom corners as stationary dummy paddles with $R_{post} = 10$ pixels and infinite mass. The centers are placed at $(0, 170)$ and $(0, 310)$.

Consider the following cases:

1. **Flat Wall:** If $Y_{collision} \leq 170$ or $Y_{collision} \geq 310$, the puck has collided with the wall. We simply apply the same colliding-with-a-wall logic from earlier.
2. **Goal Zone:** If $180 + r_{puck} < Y_{collision} < 300 - r_{puck}$, the puck has completely cleared both rounded corners and has made a goal.
3. **Top Goal Post:** If $170 < Y_{collision} \leq 180 + r_{puck}$, the puck won't strike the wall but will instead intersect the rounded edge of the top post. We can then apply the colliding-with-a-paddle logic from earlier.
4. **Bottom Goal Post:** If $300 - r_{puck} \leq Y_{collision} < 310$, the puck is on a trajectory to hit the bottom goal post. We again apply the colliding-with-a-paddle logic from earlier.

4. Resource Budgets

4.1 Resource Budgets Table



Category	Graphics	Size (pixels)	# of image	Total size (bits)
Score number		35*45	8	302400
Total Graphics Memory Budget (Bits)				302400

Category	Time (s)	Size (bits)
Puck hitting wall	0.05	10024
Puck hitting paddle	0.06	13368
Score	0.5	70216
Total Sound Memory Budget (Bits)		93608

For rendering the paddles and puck, the hardware will use a purely computational approach. For each pixel, the hardware evaluates the condition $x^2 + y^2 \leq r^2$, where x and y are the pixel's offsets from the object's center coordinate. If the condition is satisfied, the pixel is rendered in the object's color, and as a result no on-chip block RAM is needed for graphics storage. All visual elements including the paddles, puck, rink boundaries, and center line are drawn procedurally through comparator and multiplier logic, leaving the on-chip memory budget available exclusively for audio sample storage.

5. The Hardware/Software Interface

5.1 Interface Design Overview

5.1.1 Interface Philosophy

The hardware/software interface for the air hockey system is implemented as a **memory-mapped register interface**. The interface is designed to match the hardware/software partition: software handles controller input, game logic, collision physics, score updates, and sound-event decisions, while hardware handles VGA output, on-screen rendering, frame timing, and sound playback.

The interface is state-based rather than command-based. Software does not send low-level drawing commands to hardware. Instead, once per frame, software writes the current visible game state into a small set of registers, and hardware uses those values directly to render the next frame.

5.1.2 Information Passed Between Software and Hardware

For a holistic view, the information transferred from software to hardware consists of the current game state:

- player 1, 2 paddles center positions
- puck center position
- current score
- sound event request for the current frame

The information transferred from hardware back to software consists of status and synchronization information:

- whether the display is currently in a safe update interval
- whether the display is enabled and active
- a frame counter for debugging and validation

This interface reflects the division of responsibility: software decides what the game state is, while hardware decides how that state is displayed and sounded.

5.1.3 Register Organization

All interface registers are 32 bits wide and are memory-mapped into the processor address space. The register set is divided into three functional groups:

- **control/status registers**, used for reset, enable, and synchronization
- **game-state registers**, used to communicate object positions and score
- **sound-control registers**, used to request playback of collision or goal sounds

5.1.4 Coordinate Encoding and Frame Data Format

The paddle and puck positions are communicated as center coordinates. Each object uses one 32-bit register, with: bits [15:0] storing the x-coordinate, and bits [31:16] storing the y-coordinate

Although the visible screen resolution only requires 10 bits for x and 9 bits for y at 640 × 480 resolution, 16 bits are allocated to each coordinate for consistency.

The hardware renderer interprets these coordinates as the centers of circular objects. Paddle radius and puck radius are fixed constants inside the hardware design rather than configurable through the interface.

5.1.5 Frame Update Model

The game operates on a **per-frame update model**. Software computes the next frame's game state and writes it into the hardware registers once per display frame. A software update cycle is:

1. read both mice and compute updated paddle positions
2. update puck motion and collision physics
3. update score and determine whether a sound event occurred
4. wait for the next safe hardware update interval
5. write the new positions, score, and sound event to the memory-mapped registers
6. repeat for the next frame

Under this model, hardware always renders from a stable set of register values corresponding to one complete game-state update.

5.1.6 Synchronization Mechanism

Synchronization between software and hardware is implemented through **memory-mapped polling** rather than hardware interrupts. The hardware exposes a VSYNC_READY bit in the STATUS register. This bit is asserted during the vertical blanking interval and deasserted while visible pixels are being drawn.

Software polls VSYNC_READY and only writes updated game-state registers when this bit is asserted. After completing the register writes, software waits for VSYNC_READY to return to 0 before beginning the next update cycle.

A FRAME_COUNTER register is also provided for debugging and validation. This register increments once per completed frame and allows software to confirm that the display pipeline is advancing correctly.

5.2 Register Map Overview

This table summarizes the memory-mapped register interface between the software and the FPGA air hockey peripheral. All registers are 32 bits wide, and offsets are given relative to the base address.

Offset	Register Name	Access	Purpose
0x00	STATUS	R	Hardware status bits, including VSYNC_READY
0x04	FRAME_COUNTER	R	Counts completed display frames
0x08	P1_POS	W	Player 1 paddle center position
0x0C	P2_POS	W	Player 2 paddle center position
0x10	PUCK1_POS	W	Puck 1 center position
0x14	SCORE	W	Player 1 and Player 2 scores
0x18	SOUND_CONTROL	W	Sound event command sent from software to hardware

5.3 Registers Definition

5.3.1 STATUS Register (0x00)

This register is written by hardware and read by software.

Bit(s)	Name	Meaning
[0]	VSYNC_READY	1 during vertical blanking interval, 0 otherwise
[1]	DISPLAY_ACTIVE	1 if rendering is enabled
[2]	SOUND_BUSY	1 while a sound effect is currently playing
[31:3]	Reserved	Reads as 0

VSYNC_READY is the most important status bit for synchronization. Software polls this bit to determine when to write the next frame's game state. When `VSYNC_READY == 1`, software may safely write the next frame's game-state registers without risking mid-frame tearing. Hardware updates this register continuously as part of the display timing logic.

5.3.2 FRAME_COUNTER Register (0x04)

This register increments once per displayed frame.

Bit(s)	Name	Meaning
[31:0]	FRAME_COUNT	Total number of completed display frames since reset

This register is mainly used for debugging and verifying that the display pipeline is advancing correctly.

5.3.3 PLAYER_1 Position Register (0x08)

This register stores the center coordinates of Player 1's paddle.

Bit(s)	Name	Meaning
[15:0]	P1_X	x-coordinate of Player 1 paddle center
[31:16]	P1_Y	y-coordinate of Player 1 paddle center

When software writes P1_POS, it updates the center coordinate of Player 1's paddle for the next rendered frame. The lower 16 bits are interpreted as the x-coordinate and the upper 16 bits as the y-coordinate. These coordinates are measured in screen pixels relative to the top-left corner of the visible display region, with x increasing to the right and y increasing downward.

The renderer interprets this value as the center of a solid circular paddle with a fixed radius of 20 pixels. For each pixel being drawn, the renderer compares the current pixel location against the stored center coordinate and paddle radius. If the pixel lies within the circle, the renderer outputs the paddle color for that pixel; otherwise, it continues evaluating other scene elements.

5.3.4 PLAYER_2 Position Register (0x0C)

This register stores the center coordinates of Player 2's paddle.

Bit(s)	Name	Meaning
[15:0]	P2_X	x-coordinate of Player 2 paddle center
[31:16]	P2_Y	y-coordinate of Player 2 paddle center

Same as 5.3.3 Player 1 Position Register.

5.3.5 PUCK_POSITION Register (0x10)

This register stores the center coordinates of the puck.

Bit(s)	Name	Meaning
[15:0]	PUCK1_X	x-coordinate of Puck 1 center
[31:16]	PUCK1_Y	y-coordinate of Puck 1 center

Same as 5.3.3 Player 1 Position Register, except radius is 10 pixels.

5.3.6 SCORE Register (0x14)

This register stores the current score.

Bit(s)	Name	Meaning
[2:0]	SCORE_P1	Player 1 score
[5:3]	SCORE_P2	Player 2 score
[31:6]	Reserved	Must be written as 0

When software writes SCORE, it updates the numeric scores shown on the screen for Player 1 and Player 2. The lower score field is interpreted as Player 1's score and the upper score field as Player 2's score. These values are not used for game logic inside hardware; they are only used for display.

The score display logic reads this register during rendering and converts the stored values into on-screen digits in the score region of the display. For pixels that fall within the score-display area, the score logic determines whether the current pixel belongs to one of the active digit segments or digit bitmap elements, and the renderer overlays the score graphics onto the scene.

5.3.8 SOUND_CONTROL Register (0x18)

This register is used by software to request sound playback. Software decides **what happened** in the game, while hardware decides **how it sounds**.

Bit(s)	Name	Meaning
[2:0]	SOUND_EVENT	Encoded sound event ID
[3]	SOUND_TRIGGER	Pulse or edge used to start playback
[31:4]	Reserved	Must be written as 0

The SOUND_EVENT field is defined as follows:

0 - no sound; 1 - puck hit wall; 2 - puck hit paddle; 3- goal scored.

When software detects one of these events, it writes the appropriate event code into SOUND_EVENT. The hardware sound generator immediately produces the corresponding sound effect. In the case of a new code being written while a sound is being played, the new sound event will take place immediately, cutting off the old sound.

6. Input Path and Protocol

6.1 Software Stack

The two player controllers are standard USB HID mice connected to the board's USB host ports. The project will use the normal Linux USB stack, Linux HID driver, and Linux input subsystem rather than parsing USB packets directly in custom FPGA logic. In userspace, the game program will read the mice through the evdev interface exposed as /dev/input/eventX device files. The userspace library used for this is libevdev, which is a wrapper library for evdev devices.

The software path is:

USB mouse → USB host controller → Linux HID/input drivers → `/dev/input/eventX` → `libevdev` → game software

Linux device drivers communicate to the hardware, including via USB, and generate events, while the `evdev` handler passes those hardware-independent events to userspace. It also states that `evdev` is the preferred interface for userspace to consume user input.

6.2 USB-Level Protocol

The endpoint is polled by the host controller at the interval specified by the endpoint descriptor's polling interval field. USB HID therefore uses host-driven polling for mouse input rather than device-initiated transmission.

For a HID boot mouse, the raw mouse input report format is fixed for the first three bytes. The HID 1.11 specification states that the first three bytes of the report for a boot mouse are:

- **Byte 0:** button bits
- **Byte 1:** X displacement (relative X motion)
- **Byte 2:** Y displacement (relative Y motion)

6.3 Linux Event Conversion

The project software will not read raw USB HID bytes directly. Linux's HID and input layers will translate the USB HID reports into evdev events. Those events are delivered to userspace as struct input_event records read from /dev/input/eventX. The exact structure used by Linux is:

```C/C++
struct input_event {
        struct timeval time;
        unsigned short type; // event class
        unsigned short code; // specific axis / button
        int value; // event payload
};
```

For a simple mouse movement with no button change, the game software will receive event records of this form:

1. type = EV_REL, code = REL_X, value = dx
2. type = EV_REL, code = REL_Y, value = dy
3. type = EV_SYN, code = SYN_REPORT, value = 0

6.4 Linux Event Conversion

The design uses **two independent mice**, one for each player. Each mouse will be opened as a separate evdev device file, for example /dev/input/eventA and /dev/input/eventB. The software will keep these device streams separate so that each player's motion updates only one paddle.

This is one reason evdev is chosen here over the aggregated /dev/input/mice interface: evdev preserves per-device event streams, while /dev/input/mice is a shared aggregate mouse interface.

6.5 Input Rate Versus Game Update Rate

The mouse input rate and the paddle update rate are not the same quantity. At the USB layer, input reports arrive according to the mouse endpoint polling interval. At the Linux userspace layer, the program receives evdev event records whenever the kernel posts new translated mouse events. The exact report rate is therefore determined by the mouse device and USB polling interval, not by the game frame rate.

The game itself runs at **60 Hz**, synchronized to the display frame rate. Therefore, the software does **not** update paddle positions every time an individual mouse event arrives. Instead, it accumulates motion deltas between frames, then applies the accumulated movement once per frame during the synchronized game update.

For each mouse, the software maintains motion accumulators:

- accumulated x delta
- accumulated y delta

Each time an evdev event is received:

- if type == EV_REL and code == REL_X, the value is added to that player's x accumulator
- if type == EV_REL and code == REL_Y, the value is added to that player's y accumulator
- if type == EV_SYN and code == SYN_REPORT, that event packet is complete

Once per display frame, the game loop performs the following sequence:

1. wait until VSYNC_READY == 1
2. apply the accumulated deltas to Player 1 and Player 2 paddle positions
3. clamp both paddles to their legal movement regions
4. clear the mouse-delta accumulators
5. write to registers
6. wait until VSYNC_READY == 0
7. continue collecting input for the next frame