Monkey Madness Design Doc

Jake Torres (jrt2170), William Freedman (wrf2107)
Kyle Edwards (kje2115), Sadie Freisthler (srf2156)
Madeline Skeel (mgs2189)
CSEE 4840 Embedded System Design - Spring 2025

April 19, 2025

Introduction

The motivation of our project is to create a fun arcade game at the intersection of the interests of our group in interesting hardware devices, silliness, and computer graphics. Our fascination with the trackball hardware device, as shown in Figure 1, inspired us to design an arcade console and a video game that uses the trackball as its main input source.

Figure 1: Trackball

In order to find a game that best utilized the trackball input without requiring a full 3D environment, we took inspiration from games like Super Monkey Ball, Marble Madness, and the bonus stages from some of the

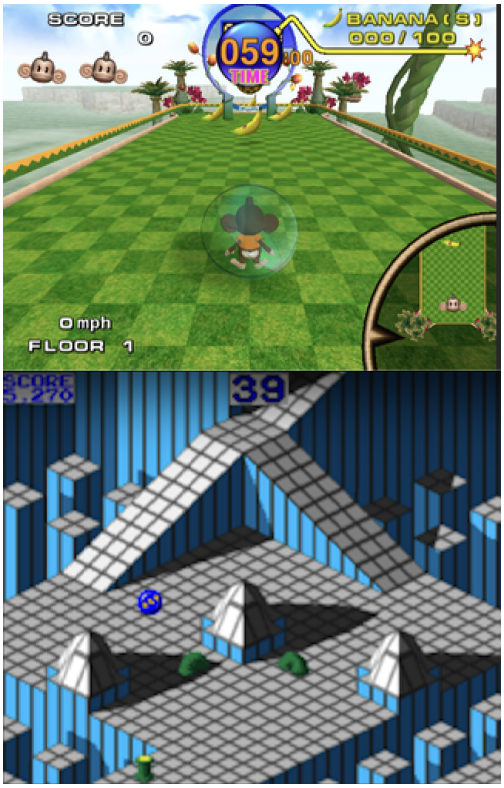


Figure 2: Super Monkey Ball & Marble Madness

Figure 3: Bonus Stage, Sonic Mania

old Sonic games, which all feature spheres moving around perceived three-dimensional planes but are actually rendered in two dimensions. This allows our game to use the full range of motion of the trackball while keeping the scope manageable, and we thought the trackball would pair perfectly with this style of game!

For this project, we would turn the FPGA into an arcade machine, allowing for reset capabilities, video output, input from buttons, and a trackball, and any other features we encounter as necessary for an arcade machine. We would then have to design a device driver for our trackball and any buttons we use. Lastly, we would be designing our game from scratch using the previously mentioned games as inspiration, all of which are shown in Figures 2 and 3.

Block Diagram

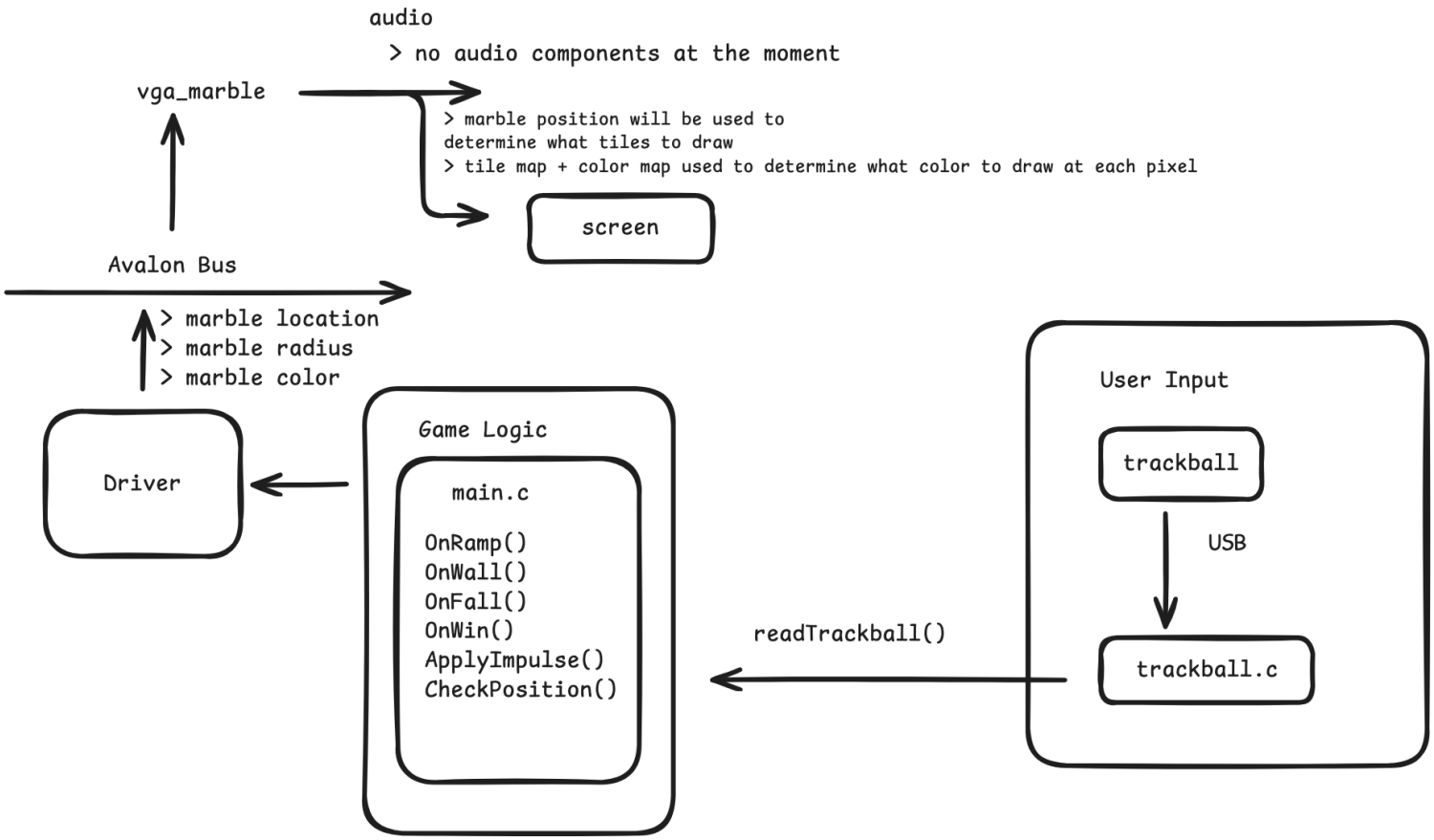


Figure 4: Block Diagram

Algorithms

Game Logic

Starting the Game

When you start the game there will be a main menu screen that shows the title of the game, text indicating the controls used to play the game, and the number of marbles launched (which is another way to say the number of games played) since the game was booted.

Gameplay

The goal of the game is to successfully traverse the marble from the starting location to the final destination marked with a hole for the marble to fall into. The player will move the marble around the course using the trackball.

As the player traverses through the course, they will encounter obstacles such as ramps, cliffs, and bumps in the path. The player must move carefully, as falling off the map returns them to the start screen and they will have to start over.

If the player successfully completes the level by getting the marble to the hole, they will advance to the next level. If they complete the final level, the time it took to finish the game will be displayed.

Game Maps

The software will contain a GameMap that allows the logic to be informed of which tile is in any given x- and y position. The GameMap will be a 2D array of characters, with each character corresponding to a certain type of tile. To conserve space, the dimensions of the matrix can be half of that of the tile map, allowing 4 bytes to be used for each position since there will be fewer than 16 distinct types of tiles.

Game Implementation

The marble object maintains its own velocity and interacts with the environment through a set of member functions. These functions will control

how the marble responds to inputs from the trackball, and the physics of the game map.

applyImpulse(int dx, int dy)
This will add the given change in x and y velocities to the marbles current x and y velocities. This will be called when the player uses the trackball to apply a directional force to the marble.

checkPosition()
Reads the marble's current x and y position to look up the corresponding terrain (ex. ramp, flat, wall) in the GameMap, to determine what terrain the marble is on and based on that what function needs to be called (OnRamp(), OnWall(), OnFall(), or OnWin())

handleRamp(int direction)
Applies a fixed change to the marble's velocity to simulate gravity pulling it down the ramp. The direction of the velocity changes depending on the ramp's orientation which is inferred from the GameMap

handleWall(int nx, int ny)
This function is called when the marble collides with a wall. The input parameters are the wall's normal. Upon collision, the new velocity is computed by reflecting the current velocity vector across the wall normal.

handleFall()
Called when the marble moves out of bounds. This triggers a pause and then returns the player to the start screen. It also updates the counter that tracks how many marbles have been launched since the game booted.

handleWin()
Called when the marble enters a win tile (the hole). This function will trigger the system to advance to the next level or if it is the final level a win screen will be displayed.

Peripheral-Interfacing Functions

setupReader()

This function is called at the start of each level to perform any necessary setup for the trackball-reading thread (discussed below).

readTrackball()
This function returns a `struct` storing the net dx and dy input by the trackball since the last time it was called. These values are then used to calculate the input values to `applyImpulse`.

Graphics Generation

Graphics rendering will be the primary function of the FPGA, which will be configured specifically to draw isometric levels. The actual rendering will not affect any game logic.

Graphics generation will be done using three main levels: a background level that will be used for rendering the isometric background, a sprite level that will be used to draw sprites, and a UI level that will be used to draw static UI elements.

Critically, our graphics generation and rendering have as little connection to the software logic as possible, besides updating of position or level. Graphics rendering will all occur on the FPGA using a GameMap, TileMap and ColorMap.

The GameMap will be a top-down 2.5D representation of our levels: for any (x, y) coordinate pair, there can only be one tile, but the tiles can have a height between 0 and 256 (one byte for height). As such, each level will also have an associated height map that specifies the heights of each tile. Hardware will then take this GameMap and the associated height map in order to render an isometric projection of the GameMap.

Tiles will be stored with 4bpp (4 bits per pixel), meaning each tile pixel can be one of 16 colors of its palette. In order to differentiate levels, there may be different palettes associated with each level.

In order to translate from top down 2D coordinates (x, y, z) to isometric coordinates (x', y'), we will use the following simple formulas:

$$x' = \frac{x - y}{2}$$
$$y' = \frac{x + y + 2z}{4}$$

These operations are all powers of two, so they should be incredibly efficient.

We can also easily convert back to top-down coordinates using the following formulas:

$$x = x' + 2y'$$
$$y = -x' + 2y'$$

Sprites will be 16x16 pixels large, with a maximum of 256 (subject to change) sprites on screen at once. This is overkill.

The UI layer will be much simpler, being a static 40x32 grid of 8x8 textures. This will be used for all UI aspects.

Since we are using 4bpp, each color palette will have a maximum of 16 possible colors. It's easiest to store the raw RGB values of each color, meaning each palette will have 45 bytes (one color will always be reserved for transparent).

Resource Budget

Like mentioned in the graphics generation section, we will be using a tile map in order to display our levels to keep our memory usage low. The only sprite that will be in the game will be the sprite for the banana marble itself, while the rest of the visuals are tiles.

In our game, we have 3 different tiles. One tile is for floors and walls, one tile is for ramps facing left and there is one for ramps facing right. We can use these 3 basic tiles to generate the terrain for our level. We plan to split each tile into a hexagon of 6 distinct triangles. We can reuse one of the triangles from the flat floor tile to produce the ramp tiles and the same triangles for the left ramp can be used to generate the right facing ramp. Because of this we can conserve some bytes being used to store the tiles. Here are the resources displayed and layed out in a table.

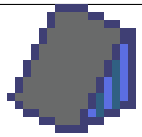
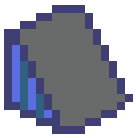
Name	Graphic	Size (bits)	Total Size (bytes)
Banana Ball		16x16	128
Floor Tile Using 6 Triangles		16x16	192
Ramp Left Tile Using 5 Triangles		16x16	160
Ramp Right Tile Using Existing Triangles		16x16	0

Figure 5: Resources - Total Size: 480 bytes

The Hardware

Registers

The NES has the following register layout (though some have been cut out for the sake of brevity):

Name	Type	Description
PPUCTRL	W	Selector Bits
PPUMASK	W	Rendering Selector Bits
OAMADDR	W	OAM Read/Write Address
OAMDATA	R/W	OAM data read/write
PPUSCROLL	W	Y Scroll Bits (there will be no x scrolling)
PPUADDR	Wx2	VRAM address
PPUDATA	R/W	VRAM data read/write

Figure 6: Register Map

Our register map will look very similar to this, though the size of each register will be larger, and we may opt instead to write data before writing address, the thought being that the FPGA would store the most recent data written to it in a register, and any writes to the address register simply copying the contents of that data register to the specified destination.

Peripherals

The user will provide input to the game through a Kensington Orbit Optical Trackball mouse, which provides a 360° trackball, as well as two buttons. The mouse connects to the FPGA via a USB port, and its input can be read through the /dev/input/event0 file.

Inputs are sent and read in the form of the input_event struct, which allows our userspace program to read the mouse's input. Since we are only using the mouse for directional input, we only store packets where the .type field is set to EV_REL, in which case we read the .value field, which will either be an x or y offset, depending on the value of ev.code (REL_X or REL_Y, respectively).

Since the trackball will be sending events far more often than frames will be generated, we will have a thread constantly reading and processing input events. This thread will store and update net dx and net dy values, corresponding to the accumulated change in the ball's position since dx and dy were last read by the main thread. This thread will also use appropriate locking to ensure that the main thread can read dx and dy correctly.