

# ECE 4515: Digital Design 2

## Project 4: Wizard Volleyball Game on VGA Monitor.

Manav Bhavesh Shah  
Bradley Department of Electrical and Computer Engineering  
Virginia Polytechnic Institute and State University  
Blacksburg, USA  
manavbs28@vt.edu

*“I have neither given nor received unauthorized assistance on this assignment.”*

*May 4, 2025*

*~Manav Shah*

## **What does your application do?**

The application I built for this project is inspired by a game named “Bobby Volley” I used to play with my dad on PC. It is a 2-player game which is interactive and involves a lot of graphics.

My game application is a fully playable game with ball physics such as collisions (ball with walls, ball with net, and ball with players 1 and 2), momentum transfer, acceleration, and movement of the ball under the influence of gravity.

## **How does the user operate your application?**

Steps the user must follow to operate my application are extremely straightforward.

Interface for interacting with the game:

There are 4 push buttons on the De1-SOC board. These buttons are the inputs for the two players.



The left 2 push buttons control Player 1 (Left Wizard) and the right 2 push buttons control Player 2 (Right Wizard).

Flipping the reset key will restart the game and reset the score (health bars)

The background used in the game can be used as a screen saver for removing all sprites from the screen and displaying the beach backdrop

as wallpaper. This is done by flipping the screen saver switch of the leftmost switch shown in the diagram above.

### **How did you carry out the design and testing of your application?**

Designing the application was a continual code -> recompile and visually inspect approach. I first started out by generating a background for the game which is the beach setup with clouds, sand and the horizon. For artistic measures, I used a combination of image editing and AI image generators to produce something which looks good on the VGA screen. Once I had this image, I wrote a python script which takes the image along with a few other parameters such as number of pixels for R, G and B and creates a .mif file of proper format for the image. Then I use the IP catalog in Quartus to create ROM and instantiate it with the .mif file of the background generated.

Next, I moved on to the game sprites, i.e. the volleyball itself, the two players (which I made into wizards). The same ROM approach was used to create all these sprites as well.

Testing:

Due to the visual aspect of the project, testing had to be done visually. Any changes I made to the code had to be translated properly onto the display. So, testing involved multiple iterations of compilation and visual inspection to see if the proper functionality was observed. This method was tedious but effective since it helped me to catch bugs in the code and debug the reason based on what the screen displayed wrong.

## **- What difficulties did you face in performing the design and testing of your application?**

### **Major Difficulties in Design and Testing:**

- 1.** Use of integers to implement momentum transfer, acceleration and complex ball physics which looks appealing and realistic on the VGA monitor.
- 2.** Adding masking/transparency bits to the sprites to be able to view the background from the empty spaces withing the sprites. This is what allows me to create a perfect spherical game ball even though the ROM stores a complete image which is square.
- 3.** Logic to detect, differentiate and represent ball collisions with all possible surfaces. (ground, players, net, and walls).
- 4.** Implementing ball position update logic on the display. This was particularly complex since I could not use vectors and trigonometric resolution with integers. As a solution I represent the position and velocity of the ball in 2d cartesian coordinate system with independent registers for position of ball (x,y) and velocity of ball (Vx, Vy). This removed the need to resolve angles.
- 5.** Matching visual refresh rate. The VGA monitor runs at 25Mhz frequency so changing positions at this rate would not be feasible. I wrote a separate tick counter which I calculated to match visual appeal, so the sprite movements feel natural and in real time.

6. Deciding parameters like gravity, collision boosts, initial velocities and acceleration of sprites. This is what makes the game look real and the only way to select proper values was by a hit and trial method to figure out which parameters looks the best on the screen.
7. Optimizing limited resources. To reduce RAM utilization and compilation times, I implemented a 2 port ROM w=for the wizards and switched the color assignments which creates the illusion of different colored wizards. Making this ROM 2 port was important to allow simultaneous indexing into the ram to get pixel information for both players. Another optimization which I did was to go with health bars to keep score instead of number which would add more ROM. These health bars are created on the display and the red bar in them is mapped to represent the remaining life of the player. Every time a player concedes a point, this health bar drops a decided amount to visually represent the conceding of the point and the service changes.
8. Compilation times and multiple iterations to check a simple change made in code and if it correctly captures the functionality I intended.

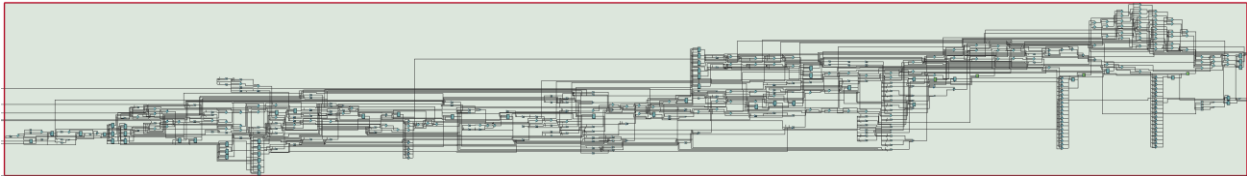
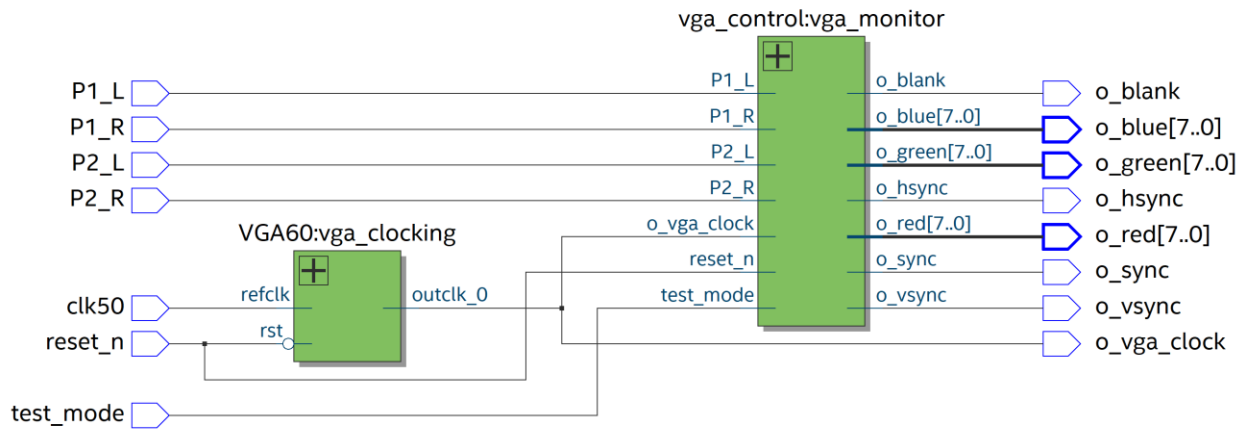
**- What did you learn from the experience?**

This project has significantly improved my grasp on the design process which goes into complex applications. It is a very rewarding experience considering the final product I was able to achieve in the time frame and see a fully playable 2-person interactive game. I also had to make several choices to best utilize the resources available.

**- With more time or resources, what could you do to further improve or enhance your application?**

With more resources (a ps2 keyboard...) I was hoping to add better input controls for the game where you would not need to use the onboard push buttons to control the sprites. I also had a PWM module which I wanted to include in the game to create sound effects but connecting GPIO pins to the audio jack of a speaker but could not manage to find all the extra resources that I would need.

**- Provide a top-level RTL schematic of your model.**



**- Document the methodology you used for doing things like generating the graphics frame or sprite data.**

Methodology for generating graphics:

### 1) Volleyball

I start by taking an image and editing it to the dimensions I like. The next step is to process the image with a python script to generate an MIF file.



This image is converted to the mif file using this python script :

```
from PIL import Image
# Convert RGBA to 4-bit [mask][R][G][B] using alpha for mask.
def rgba_to_masked_rgb4(r, g, b, a, alpha_thresh=10):
    mask = 1 if a > alpha_thresh else 0
    r1 = 1 if r > 127 else 0
    g1 = 1 if g > 127 else 0
    b1 = 1 if b > 127 else 0
    return (mask << 3) | (r1 << 2) | (g1 << 1) | b1

def generate_mif_4bit(image_path, mif_path, width=40, height=40):
    img = Image.open(image_path).convert('RGBA') # Use RGBA to get alpha
    img = img.resize((width, height))
    pixels = list(img.getdata())

    mif_lines = []
    mif_lines.append(f"DEPTH = {width * height};")
    mif_lines.append("WIDTH = 4;")
    mif_lines.append("")
    mif_lines.append("ADDRESS_RADIX = HEX;")
    mif_lines.append("DATA_RADIX = BIN;")
    mif_lines.append("")
    mif_lines.append("CONTENT")
    mif_lines.append("BEGIN")
```



```

for addr, (r, g, b, a) in enumerate(pixels):
    data_bin = format(rgba_to_masked_rgb4(r, g, b, a), '04b')
    addr_hex = format(addr, '04X')
    mif_lines.append(f"{addr_hex} : {data_bin};")

mif_lines.append("END;")

with open(mif_path, 'w') as f:
    f.write('\n'.join(mif_lines))

print(f".mif file written to: {mif_path}")
print(f"Image resized to: {width}x{height} -> DEPTH = {width * height}")

generate_mif_4bit(
    r"C:\VT\SEM2\DigitalDesign2\volley_ROM\source_images\ball.png",
    r"C:\VT\SEM2\DigitalDesign2\volley_ROM\mif_files\ball_4bit_masked.mif",
    width=40,
    height=40
)

```

The format of the .mif file is as follows:

```

DEPTH = 1600;
WIDTH = 4;

ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;

CONTENT
BEGIN
0000 : 0000;
0001 : 0000;
0002 : 0000;
0003 : 0000;
.
.
.
063E : 0000;
063F : 0000;
END;

```

## 2) Beach Background

Following the same approach, this beach image is processed into another python file to generate a .mif file.



The beach .mif file looks like this: ii

```
DEPTH = 307200;  
WIDTH = 9;  
  
ADDRESS_RADIX = HEX;  
DATA_RADIX = BIN;  
  
CONTENT  
BEGIN  
0000 : 010101111;  
0001 : 010101111;  
.  
.  
.  
4AFFE : 111110100;  
4AFFF : 111110100;  
END;
```

### 3) Wizard Sprite:

The wizard image is shown below. I wrote a python script which uses masking bits to make the white spaces in between the body to be transparent so the background can be seen through the wizard's body on the screen.



#### Python script:

```
from PIL import Image

def rgba_to_masked_rgb7(r, g, b, a, alpha_thresh=10):
    mask = 1 if a > alpha_thresh else 0
    r2 = r >> 6 # Top 2 bits
    g2 = g >> 6
    b2 = b >> 6
    return (mask << 6) | (r2 << 4) | (g2 << 2) | b2
```

```

def generate_mif_7bit_centered(image_path, mif_path, width=120, height=120):
    img = Image.open(image_path).convert('RGBA')

    # Resize image
    img.thumbnail((width, height), Image.Resampling.LANCZOS)
    centered_img = Image.new('RGBA', (width, height), (0, 0, 0, 0))
    upper_left_x = (width - img.width) // 2
    upper_left_y = (height - img.height) // 2
    centered_img.paste(img, (upper_left_x, upper_left_y))

    pixels = list(centered_img.getdata())

    mif_lines = []
    mif_lines.append(f"DEPTH = {width * height};")
    mif_lines.append("WIDTH = 7;")
    mif_lines.append("")
    mif_lines.append("ADDRESS_RADIX = HEX;")
    mif_lines.append("DATA_RADIX = BIN;")
    mif_lines.append("")
    mif_lines.append("CONTENT")
    mif_lines.append("BEGIN")

    for addr, (r, g, b, a) in enumerate(pixels):
        data_bin = format(rgba_to_masked_rgb7(r, g, b, a), '07b')
        addr_hex = format(addr, '04X')
        mif_lines.append(f"{addr_hex} : {data_bin};")

    mif_lines.append("END;")

    with open(mif_path, 'w') as f:
        f.write('\n'.join(mif_lines))

    print(f".mif file written to: {mif_path}")
    print(f"Image centered and resized to: {width}x{height} -> DEPTH = {width * height}")

generate_mif_7bit_centered(
    r"C:\VT\SEM2\DigitalDesign2\volley_ROM\source_images\wizard.png",
    r"C:\VT\SEM2\DigitalDesign2\volley_ROM\mif_files\wizard.mif",
    width=120,
    height=120
)

```

#### 4) Game over screen



Python code to convert this image to mif file:

```
from PIL import Image

def is_approx_red(r, g, b, a, alpha_thresh=10):
    """Return 1 for pixels that are approximately red, else 0 (black)."""
    if a <= alpha_thresh:
        return 0 # Transparent → treat as black
```

```

    if r > 150 and g < 100 and b < 100:
        return 1
    return 0

def generate_gameover_mif_fuzzy(image_path, mif_path, width=640, height=480):
    img = Image.open(image_path).convert('RGBA')

    # Ensure image is exactly 640x480
    img = img.resize((width, height), Image.Resampling.LANCZOS)
    pixels = list(img.getdata())

    mif_lines = [
        f"DEPTH = {width * height};",
        "WIDTH = 1;",
        "",
        "ADDRESS_RADIX = HEX;",
        "DATA_RADIX = BIN;",
        "",
        "CONTENT",
        "BEGIN"
    ]

    for addr, (r, g, b, a) in enumerate(pixels):
        bit = is_approx_red(r, g, b, a)
        addr_hex = format(addr, '05X')
        mif_lines.append(f"{addr_hex} : {bit};")

    mif_lines.append("END;")

    with open(mif_path, 'w') as f:
        f.write('\n'.join(mif_lines))

    print(f".mif file written to: {mif_path}")
    print(f"Image resized to {width}x{height}, total entries: {width * height}")

generate_gameover_mif_fuzzy(
    r"C:\VT\SEM2\DigitalDesign2\volley_ROM\game_over.jpg",
    r"C:\VT\SEM2\DigitalDesign2\volley_ROM\mif_files\game_over_fuzzy.mif",
    width=640,
    height=480
)

```

**- Are the modules well-documented and easy to follow? (As the module will likely consist of many instances of lower level modules, it's important for your documentation to provide a high-level view of how your system is structured.)**

My application uses only 3 modules.

- 1) VGA clocking: This module is a PLL which generates a 25MHz clock for the VGA monitor.
- 2) VGA monitor: This module controls everything that is displayed on the VGA monitor. It houses the control logic of the VGA, sprite pixel generation and everything else associated with the game logic.
- 3) Top module to connect these two modules together and make connections for clock, push buttons and switches.

My modules are heavily commented to show different parts of the code logic and are easy to follow with everything happening in a single place.

**- Does the VGA controller generate graphics from data stored in a frame buffer, in sprite ROMs, or both?**

Yes, the VGA controller generates a ton of graphics from data stored in Sprite ROMs. Namely, the background itself, the wizards' sprites, the ball. Graphics such as the health bars and game NET are created by simple code snippets. I use flags for all sprites and when a particular flag is on since that pixel is being drawn due to values of hsync and vsync, a multiplexer type of logic to relay pixel information for that sprite component from the ROM.



**- Do the graphics generated satisfy the minimum requirements for display resolution? (200 x 200 pixels)**

My application uses the entirety of the VGA monitor. (640 by 480 resolution).



**- If the design contains a user input interface, does the interface operate according to the design description?**

To interact with the application, there are 4 push buttons on the De1-SOC which control the left and right movement of the Wizard Sprites.



There is a single reset switch and a screen saver switch to only display the background.

**- What combination of frame buffer and sprite graphics has the designer implemented?**

The position of sprites is continuously updated according to the visual refresh signal. The ball sprite for example is drawn at different places every frame based upon the position registers which change according to the ball physics logic implemented.

**- What is the extent of the user interaction with the design?**

This is a non-repeating game, and every round of play will depend upon the players' inputs. The service also changes once a point is conceded by any player and the game can be played on until one of the players' health bars runs out which will be indicated by a game over screen. Once this happens, you can restart the game and reset the health bars by toggling the reset switch.

**- What other features has the designer incorporated into the design result?**

1. Visually appealing graphics. Generating images which look good on the screen while keeping that retro game vibe was a challenge.
2. Implementing ball physics such as momentum transfer, acceleration, gravity, collisions etc.
3. Perfectly Spherical Ball and being able to see the background setup even through the empty spaces of the

sprite. I had to add masking bits to the mif file and code the logic into the display control to properly handle what pixels change.

4. Making it look correct. The most difficult thing is to have motion which looks like what we see in real life but generating that with integers instead of using floating point calculations. Every parameter in my code (gravity, collision boost, velocity, etc) has been tweaked to be what it is after hundreds of iterations to find what looks best on the VGA monitor.
5. Making creative choices like using health bars to keep scores to save memory and reduce compilation times.
6. Screen saver mode which can be used as a standby to view the background beach image.


**(These features may overlap with those considered for additional credit.)**

**Features such as these will be considered for additional credit, up to the number of points shown. Features not listed here will be considered if they are commensurate in difficulty/challenge with those listed. Bring these features to the grading staff's attention during evaluation and in your documentation.**

**- Do your graphics change? Are the graphics generated from frame buffer data OR a sprite generated from ROM data change or move in a way that is controllable by the user or determined by an additional module such as a state machine? (5 points)**

The movement of the ball is controlled by the momentum transfer on the ball imparted during the collisions with the sprites. The ball also collides with the net, walls and ground so there is a lot of movement and it is majorly controlled by the user inputs since the velocity of the Wizard at the time of collision will determine the direction and velocity of the ball after impact.

## Resource Utilization and Flow Summary.

Flow Summary	
 <<Filter>>	
Flow Status	Successful - Mon May 5 08:22:44 2025
Quartus Prime Version	23.1std.1 Build 993 05...4/2024 SC Lite Edition
Revision Name	volley_top
Top-level Entity Name	volley_top
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	928 / 32,070 ( 3 % )
Total registers	328
Total pins	36 / 457 ( 8 % )
Total virtual pins	0
Total block memory bits	3,179,200 / 4,065,280 ( 78 % )
Total DSP Blocks	2 / 87 ( 2 % )
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	1 / 6 ( 17 % )
Total DLLs	0 / 4 ( 0 % )

### Conclusion:

In this project, I have created a Wizard Volleyball Game which can be played by 2 players. The game looks and plays like a traditional retro game and is built without any references from the ground up. There is no use of a readily available IP except the ROM IPs used to store Sprite Pixel information. The game has a simple physics engine which should make it more enjoyable and playable. All codes were written in System Verilog.