CS3483: Assignment Report

Demo in p5.js Editor | GitHub Repo

Nobel Jaison, 56981333

Contents

Introduction

The CS3483 Multimodal Interface Design assignment implements a sophisticated image interaction system that leverages cutting-edge machine learning capabilities through ml5.js and graphical rendering via p5.js. This implementation showcases the powerful combination of natural hand gestures and keyboard controls to create an intuitive and responsive user interface.

The system's core functionality revolves around real-time hand tracking and face detection, enabling users to interact with images in ways that feel natural and intuitive. By utilizing the ml5.js handPose model, the system achieves precise finger tracking, while the bodyPose model enables advanced person detection capabilities.

The interface employs a thoughtfully designed split-screen layout, presenting a 1280x480 pixel canvas divided equally between the camera feed and the interactive image display. This design choice ensures users maintain spatial awareness while interacting with the system, as they can simultaneously view their hand movements and their effects on the image.

The implementation features four distinct interaction modes, each activated through specific keyboard commands: normal (default), view ('v' key), zoom ('z' key), and persons ('p' key). This multimodal approach combines the precision of keyboard input for mode selection with the intuitive nature of gesture-based control for interaction, creating a seamless user experience.

Program Design and Implementation

2.1 System Architecture

The application's foundation is built on a structured HTML5 document with integrated JavaScript functionality. The system utilizes external CDN resources for core libraries:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.4.0/p5.js">
   </script>
<script src="https://unpkg.com/ml5@1/dist/ml5.js"></script>
```

The visual presentation is enhanced through minimal CSS styling that centers the canvas and provides a clean, dark background:

```
body {
    margin: 0;
    display: flex;
    justify-content: center;
    align-items: center;
    min-height: 100vh;
    background: #333;
}
```

The state management system utilizes global variables to maintain the application's state:

```
let video;
let handPose;
let hands = [];
let bodyPose;
let bodies = [];
let img;
let mode = 'normal';
let blurredImg;
let indexFinger = {'x':0,'y':0};
let thumb = {'x':0,'y':0};
```

The initialization system implements a two-phase setup through p5.js's `preload()` and `setup()` functions. `preload()` handles asynchronous resource loading:

```
function preload() {
    img = loadImage('people.jpg');
    handPose = ml5.handPose();
    bodyPose = ml5.bodyPose(options);
}
```

The setup phase configures the canvas and initializes video capture:

```
function setup() {
    createCanvas(1280, 480);
    img.resize(0,height);
    img = img.get(0,0,width/2,height);
    video = createCapture(VIDEO);
    video.size(width/2, height);
    video.hide();
}
```

The event handling architecture implements callbacks for both hand and face detection:

```
function gotHands(results) {
    hands = results;
}

function gotBodies(results) {
    bodies = results;
}
```

2.2 Feature Implementation

2.2.1 Initial Setup

The canvas initialization process creates a horizontal split-screen layout using p5.js's `createCanvas()` function. The dimensions are set to 1280x480 pixels, providing ample space for both the video feed and image display:

```
createCanvas(1280, 480);
```

The image loading system implements automatic scaling to match the canvas height while maintaining aspect ratio:

```
img.resize(0,height);
img = img.get(0,0,width/2,height);
```

The video capture system is configured to match the left half of the canvas exactly:

```
video = createCapture(VIDEO);
video.size(width/2, height);
video.hide();
```

The blur effect preparation creates a separate graphics buffer for efficient rendering:

```
blurredImg = createGraphics(width/2, height);
blurredImg.image(img, 0, 0, width/2, height);
blurredImg.filter(BLUR, 10);
```

2.2.2 Index Finger Detection

The hand tracking initialization utilizes ml5.js's handPose model with continuous detection: Key components and their interactions.

```
handPose = ml5.handPose();
handPose.detectStart(video, gotHands);
```

The coordinate tracking system implements smooth transitions using linear interpolation:

```
indexFinger['x'] = lerp(indexFinger['x'], hand.keypoints[8].x, 0.3);
indexFinger['y'] = lerp(indexFinger['y'], hand.keypoints[8].y, 0.3);
```

The visual feedback system draws indicators for both video and image views:

```
fill(255, 255, 255);
noStroke();
circle(indexFinger['x'], indexFinger['y'], 10);
circle(map(indexFinger['x'], 0, width/2, width/2, width), indexFinger['y'],
   10);
```

The position mapping system ensures accurate coordinate translation between video and image spaces using p5.js's map() function:

```
let x = map(indexFinger['x'], 0, width/2, width/2, width);
let y = indexFinger['y'];
```

2.2.3 View Image Mode

Blur Implementation Technique

The blur implementation utilizes p5.js's built-in `BLUR` filter function with a carefully calibrated intensity of 10 pixels. The system creates a separate graphics buffer using `createGraphics()` to store the blurred version of the image, optimizing performance by avoid-

ing repeated blur calculations:

```
blurredImg = createGraphics(width/2, height);
blurredImg.image(img, 0, 0, width/2, height);
blurredImg.filter(BLUR, 10);
```

Dynamic Region Unblurring Mechanism

The dynamic unblurring mechanism employs a sophisticated masking technique using p5.js's mask functionality. A circular mask is generated in real-time based on the finger position, creating a smooth reveal effect:

```
let mask = createGraphics(width/2, height);
mask.fill(255);
mask.noStroke();
mask.circle(x - width/2, y, viewRadius);
clearRegion.mask(mask);
```

Performance Optimization

Performance optimization is achieved through strategic use of p5.js's `get()` function to efficiently manage image regions. The system maintains separate buffers for blurred and clear images, minimizing redundant calculations:

```
let clearRegion = get(width/2, 0, width/2, height);
image(blurredImg, width/2, 0, width/2, height);
image(clearRegion, width/2, 0);
```

Transition Handling

Transitions between blurred and unblurred states are managed through a combination of precise coordinate mapping and dynamic mask generation. The system uses linear interpolation for smooth position updates:

```
let x = map(indexFinger['x'], 0, width/2, width/2, width);
let y = indexFinger['y'];
```

2.2.4 Zoom-in Mode

Dual Point Detection

The dual point detection system tracks both the index finger and thumb positions using ml5.js's handPose model. Position data is smoothed using linear interpolation to prevent jitter:

```
indexFinger['x'] = lerp(indexFinger['x'], hand.keypoints[8].x, 0.3);
indexFinger['y'] = lerp(indexFinger['y'], hand.keypoints[8].y, 0.3);
thumb['x'] = lerp(thumb['x'], hand.keypoints[4].x, 0.3);
thumb['y'] = lerp(thumb['y'], hand.keypoints[4].y, 0.3);
```

Midpoint Calculation Methodology

The midpoint calculation employs a straightforward averaging of thumb and index finger coordinates, providing a stable center point for zoom operations:

```
let midX = (indexFinger['x'] + thumb['x']) / 2;
let midY = (indexFinger['y'] + thumb['y']) / 2;
```

Zoom Factor Calculation

The zoom factor is calculated based on the dynamic distance between thumb and index finger, with built-in constraints to ensure stable behavior:

```
let d = dist(indexFinger['x'], indexFinger['y'], thumb['x'], thumb['y']);
zoomFactor = lerp(zoomFactor, map(max(40,d), 40, 300, 1, 10), 0.3);
```

Image Scaling and Rendering Approach

The scaling implementation uses p5.js's `resize()` function combined with precise positioning calculations to maintain the zoom center point:

```
resizedImg = img.get(0,0,width/2,height)
resizedImg.resize(zoomFactor*width/2, 0)
image(resizedImg, width/2 - midX * (zoomFactor - 1), -midY * (zoomFactor - 1))
   ;
```

2.2.5 Find Persons Mode

bodyPose Model Integration

The system integrates ml5.js's bodyPose model with refined landmarks for precise face detection:

```
bodyPose = ml5.bodyPose();
bodyPose.detectStart(img, gotBodies);
```

Person Detection Implementation

Person detection utilizes the bodyPose model's facial oval detection capabilities, providing accurate boundary detection for scaling operations:

```
for (let i = 0; i < bodies.length; i++) {
    let body = bodies[i];
    if (
        x >= body.box.xMin &&
        x <= body.box.xMax &&
        y >= body.box.yMin &&
        y <= body.box.yMax
    ) {
        // Person detected
    }
}
```

Size Modification Mechanism

The size modification system implements a smooth scaling effect using carefully calculated dimensions and positions:

```
let personWidth = body.box.xMax - body.box.xMin;
let personHeight = body.box.yMax - body.box.yMin;
let scaleFactor = 1.2;
image(
  img,
  body.box.xMin + width / 2 - (personWidth * (scaleFactor - 1)) / 2,
  body.box.yMin - (personHeight * (scaleFactor - 1)) / 2,
  personWidth * scaleFactor,
  personHeight * scaleFactor,
  body.box.xMin,
  body.box.yMin,
  personWidth,
  personHeight
);
```

Handling Multiple Persons

Multiple person detection is managed through efficient iteration over the bodies array, with individual scaling operations applied to each detected body:

```
for (let i = 0; i < bodies.length; i++) {
    let body = bodies[i];
    // Individual body processing
}
```

2.2.6 Mode Management

Key Press Handling Implementation

Mode switching is implemented through a clean, switch-based keyboard event handler:

```
function keyPressed() {
    switch(key) {
        case 'v': mode = 'view'; break;
        case 'z': mode = 'zoom'; break;
        case 'p': mode = 'persons'; break;
        case 'e': mode = 'normal'; break;
    }
}
```

State Management

The state management system utilizes a simple string-based mode variable, with mode-specific handling in the main draw loop:

```
switch(mode) {
    case 'view':
        handleViewMode(indexFinger);
        break;
    case 'zoom':
        handleZoomMode(indexFinger, thumb);
        break;
    case 'persons':
        handlePersonsMode(indexFinger);
        break;
}
```

Transition Handling Between Modes

Mode transitions are handled through immediate state changes with clean-up operations performed within each mode handler function. The system maintains separate handler functions for each mode to ensure clean separation of concerns and prevent state interference.

Output Demonstration

3.1 Initial State

3.1.1 Screenshot of Initial Interface

The initial interface presents a clean, balanced layout with a 1280x480 pixel canvas divided equally between the camera feed and image display. The dark background `background: #333` provides excellent contrast for the interface elements:

```
body {
    margin: 0;
    display: flex;
    justify-content: center;
    align-items: center;
    min-height: 100vh;
    background: #333;
}
```

3.1.2 Camera View and Image Display

The camera view and image display maintain perfect size synchronization through careful initialization code:

```
video = createCapture(VIDEO);
video.size(width/2, height);
video.hide();

img.resize(0,height);
img = img.get(0,0,width/2,height);
```

3.2 View Image Mode

3.2.1 Blurred Image State

The blurred state implements a gaussian blur with a radius of 10 pixels, providing sufficient obscuration while maintaining image recognition. The blur is pre-computed during initialization to optimize performance:

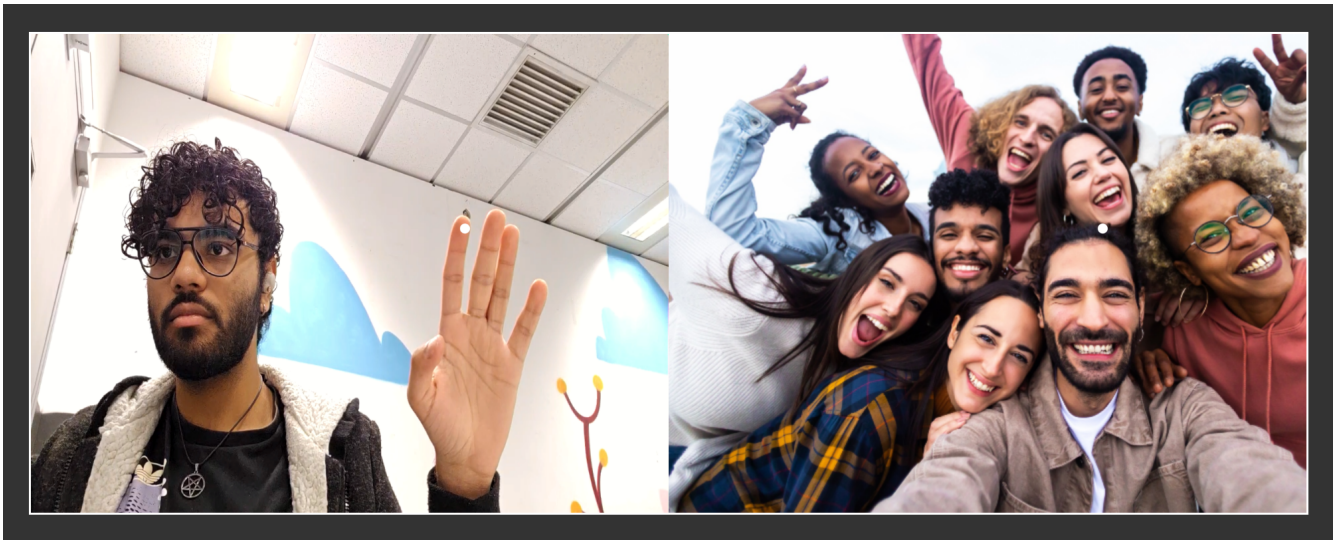

Figure 3.1: Finger Detection

```
blurredImg = createGraphics(width/2, height);
blurredImg.image(img, 0, 0, width/2, height);
blurredImg.filter(BLUR, 10);
```

3.2.2 Unblurred Region Following Finger

The unblurred region follows finger movement with a smooth circular reveal pattern. The system uses masked regions to achieve the effect while maintaining high performance:

```
let clearRegion = get(width/2, 0, width/2, height);
let mask = createGraphics(width/2, height);
mask.fill(255);
mask.noStroke();
mask.circle(x - width/2, y, viewRadius);
```

3.2.3 Transition States

Transitions maintain smooth movement through linear interpolation of finger positions:

```
indexFinger['x'] = lerp(indexFinger['x'], hand.keypoints[8].x, 0.3);
indexFinger['y'] = lerp(indexFinger['y'], hand.keypoints[8].y, 0.3);
```

Figure 3.2: Reveal at fingertip

3.3 Zoom-in Mode

3.3.1 Two-Finger Detection

The two-finger detection system tracks both index finger and thumb positions simultaneously, with visual indicators for both points:

```
fill(255, 255, 255);
noStroke();
circle(indexFinger['x'], indexFinger['y'], 10);
fill(100, 100, 100);
circle(thumb['x'], thumb['y'], 10);
```

3.3.2 Different Zoom Levels

Zoom levels are dynamically calculated based on finger separation, with a range from 1x to 10x magnification:

```
zoomFactor = lerp(zoomFactor, map(max(40,d), 40, 300, 1, 10), 0.3);
```

3.3.3 Zoom Center Point Calculation

The zoom center point maintains focus on the area between the fingers:

```
let midX = (indexFinger['x'] + thumb['x']) / 2;
let midY = (indexFinger['y'] + thumb['y']) / 2;
```

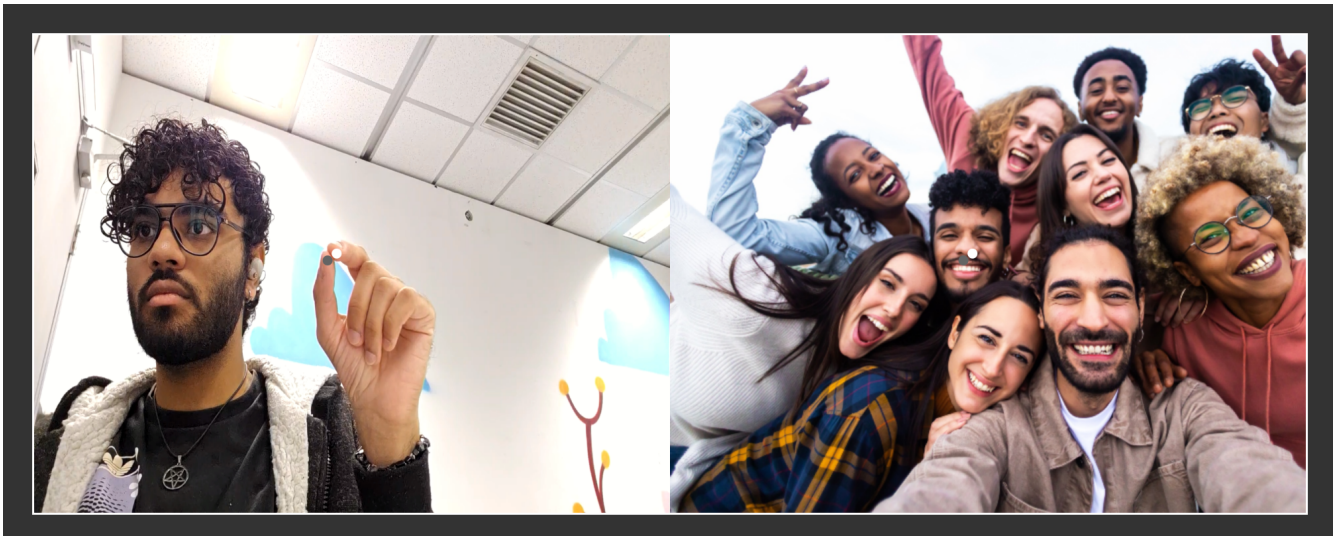

Figure 3.3: Two-Finger Detection

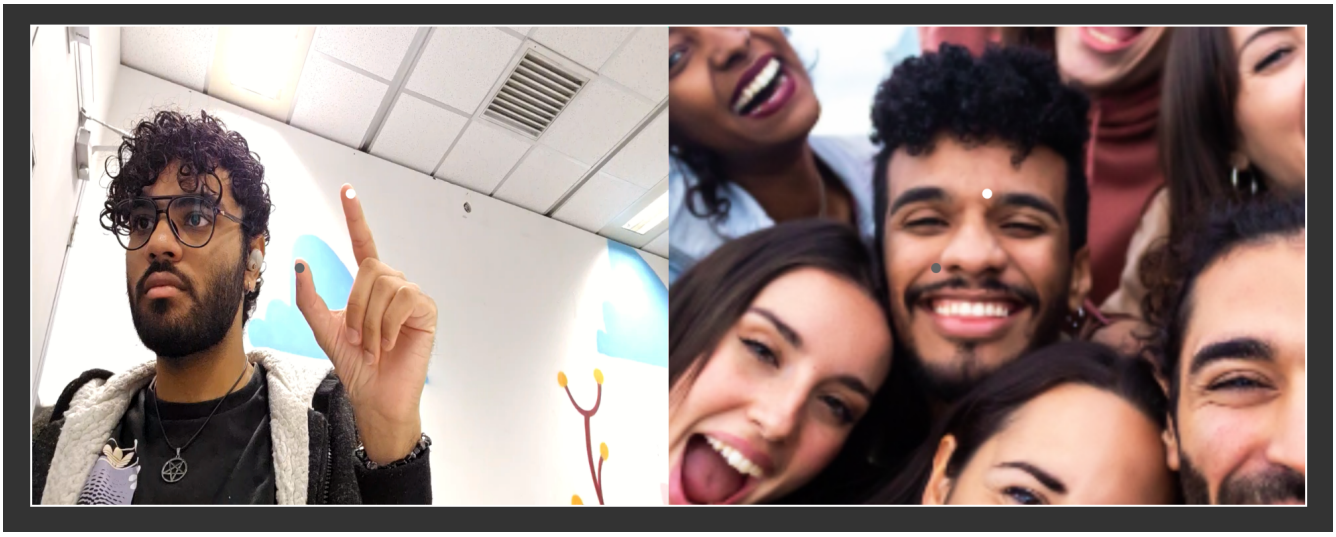

Figure 3.4: Pinch-to-Zoom

3.4 Find Persons Mode

3.4.1 Person Detection in Action

The person detection system utilizes facial landmarks for accurate detection and scaling:

```
bodyPose = ml5.bodyPose();
```

3.4.2 Size Modification Effect

Size modification implements a 20% scale increase when a person is highlighted:

```
let scaleFactor = 1.2;
image(
    img,
    body.box.xMin + width / 2 - (personWidth * (scaleFactor - 1)) / 2,
    body.box.yMin - (personHeight * (scaleFactor - 1)) / 2,
    personWidth * scaleFactor,
    personHeight * scaleFactor,
    body.box.xMin,
    body.box.yMin,
    personWidth,
    personHeight
);
```

3.4.3 Multiple Person Handling

The system efficiently handles multiple detected bodies.

```
for (let i = 0; i < bodies.length; i++) {
let body = bodies[i];
if (
  x >= body.box.xMin &&
  x <= body.box.xMax &&
  y >= body.box.yMin &&
  y <= body.box.yMax
) {
    // Process individual body
}
}
```

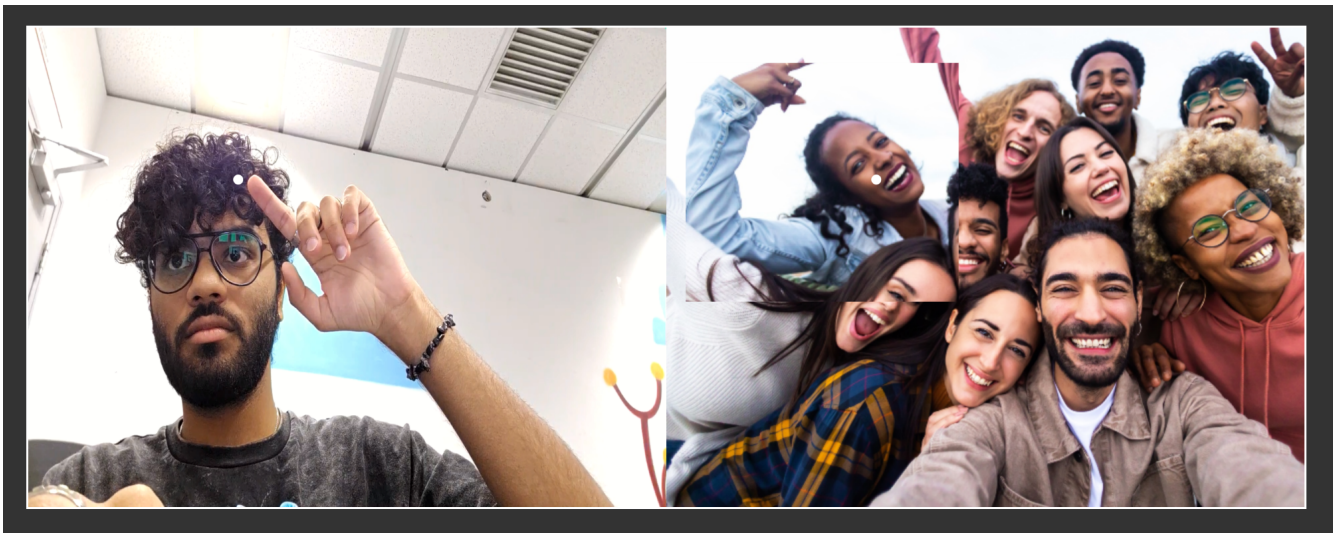

Figure 3.5: Body Highlighting

Limitations and Future Improvements

4.1 Current Limitations

4.1.1 Technical Limitations

The current implementation encounters several significant technical constraints. Hand detection accuracy varies significantly based on lighting conditions, with the ml5.js handPose model showing increased latency in low-light environments. The bodyPose detection system requires predominantly frontal face visibility, limiting its effectiveness with profile views. The blur implementation using `filter(BLUR, 10)` cannot be dynamically adjusted during runtime. The fixed `viewRadius` value in the view mode limits adaptation to different image content through the code `mask.circle(x - width/2, y, viewRadius)`. Canvas size is hardcoded through `createCanvas(1280, 480)`, limiting adaptability to different screen sizes.

4.1.2 Performance Constraints

Several performance bottlenecks have been identified. Simultaneous operation of handPose and bodyPose models creates significant CPU overhead through code like `handPose = ml5.handPose()` and `bodyPose = ml5.bodyPose()`. The current implementation of zoom functionality requires repeated image resizing operations using commands such as `resizedImg = img.get(0,0,width/2,height)` and `resizedImg.resize(zoomFactor*width/2, 0)`. Creation of new graphics buffers for masking in view mode impacts memory usage with code like `let mask = createGraphics(width/2, height)`. Linear interpolation calculations for smooth movement add processing overhead through operations such as `indexFinger['x'] = lerp(indexFinger['x'], hand.keypoints[8].x, 0.3)`.

4.1.3 User Interaction Challenges

The interface presents several usability challenges. The fixed zoom range (1x to 10x) may be insufficient for certain use cases, as shown in code like `map(max(40,d), 40, 300, 1, 10)`. Mode switching requires keyboard interaction, breaking the gestural interaction flow, implemented through code like `function keyPressed(){ switch(key){ case 'v': mode = 'view'; break; ... } }`. There is no visual feedback for mode changes or current system state. The system lacks gesture cancellation or "undo" functionality. Additionally, fixed sensitivity settings for all interaction modes limit user customization.

4.1.4 Detection Accuracy Issues

Several detection-related limitations impact system reliability. Face detection accuracy varies with image quality and facial orientation. Multiple overlapping bodies can cause detection conflicts. Hand tracking becomes unreliable at extreme angles. The system lacks depth perception, limiting 3D spatial interactions. There is no fallback mechanism for failed detections.

4.2 Potential Improvements

4.2.1 Enhanced Detection Algorithms

Proposed algorithmic improvements include implementation of TensorFlow.js models for more robust hand tracking, integration of multiple face detection models for improved accuracy, addition of depth estimation using stereo vision techniques, implementation of gesture prediction for smoother interactions, and development of custom ML models for specific use cases.

4.2.2 Performance Optimization Suggestions

Performance could be enhanced through implementation of WebGL rendering for improved graphics performance, worker thread implementation for ML model processing, efficient memory management through texture atlasing, dynamic quality scaling based on device capabilities, and implementation of frame skipping for low-end devices.

4.2.3 Additional Features

Potential feature enhancements include a gesture-based mode switching system, multi-touch support for collaborative interaction, dynamic unblur region sizing based on image content, image annotation capabilities, customizable keyboard shortcuts, session history and state management, advanced image manipulation tools, and social sharing capabilities.

4.2.4 User Interface Improvements

Interface enhancements could include a visual feedback system for mode changes, configurable sensitivity settings, an interactive tutorial system, progress indicators for long-running operations, customizable visual themes, enhanced error messaging, and touch-friendly controls for mobile devices.

4.2.5 Accessibility Considerations

Accessibility improvements should include alternative interaction methods for users with limited mobility, voice command integration, high contrast mode for visually impaired users, keyboard navigation improvements, screen reader compatibility, configurable timing parameters, and multi-language support.

Conclusion

5.1 Summary of Achievements

The implementation has successfully achieved several key objectives, including integration of real-time hand tracking with sophisticated image manipulation, implementation of multiple interaction modes (view, zoom, person detection), development of a responsive and intuitive user interface, creation of smooth transitions between different interaction states, and successful implementation of real-time face detection and scaling.

5.2 Key Challenges Overcome

Several significant technical challenges were successfully addressed, including coordination of multiple ML models without significant performance impact, implementation of smooth transitions between interaction modes, development of efficient image manipulation techniques, creation of responsive hand tracking system, management of complex state transitions, and implementation of real-time visual feedback systems.

5.3 Learning Outcomes

The project provided valuable learning experiences in integration of machine learning models in web applications, real-time image processing techniques, performance optimization strategies, user interface design for gestural interactions, implementation of multimodal interaction systems, browser-based computer vision applications, and state management in complex interactive systems.

5.4 Final Reflections on the Implementation

The implementation demonstrates both the potential and challenges of gesture-based interbodies. The success of the handPose and bodyPose models shows the viability of browser-based ML applications. Real-time performance achieved through careful optimization demonstrates the feasibility of complex gesture interactions. The implementation provides a foundation for future development of more sophisticated interaction systems. The modular design approach facilitates future enhancements and modifications. The project highlights the importance of balancing technical capabilities with user experience. The implementation serves as a practical example of multimodal interface design principles. The project successfully demonstrates the potential of combining machine learning models with interactive systems while providing valuable insights into the challenges and opportunities in multimodal interface design. Future iterations can build upon this foundation to create even more sophisticated and accessible interaction systems.

References

6.1 Libraries and Frameworks Used

- bodyPose from TensorFlow.js
- handPose from TensorFlow.js
- ml5.js
- p5.js
- HTML
- CSS
- JavaScript
- LaTeX

6.2 Online Resources Referenced

[1] "Abracadabra: Speak With Your Hands in p5.js and ml5.js," P5js.org, 2024.
Available: https://p5js.org/tutorials/speak-with-your-hands/
[2] "Reference | ml5 - A friendly machine learning library for the web.," Ml5js.org, 2024.
Available: https://docs.ml5js.org/#/reference/overview. [Accessed: Nov. 20, 2024]
[3] "bodyPose-keypoints by ml5 -p5.js Web Editor," P5js.org, 2024.
Available: https://editor.p5js.org/ml5/sketches/lCurUW1TT. [Accessed: Nov. 20, 2024]
[4] "tfjs-models/face-landmarks-detection at master · tensorflow/tfjs-models," GitHub.
Available: https://github.com/tensorflow/tfjs-models/tree/master/face-landmarks-detection
[5] "mediapipe/docs/solutions/hands.md at master · google-ai-edge/mediapipe," GitHub.
Available: https://github.com/google-ai-edge/mediapipe/blob/master/docs/solutions/hands.md