Aisha Avatar Eye Tracking & Body Following Implementation Guide

Overview

This guide will walk you through adding real-time eye tracking and body rotation to your Aisha 3D AI avatar, enabling her to follow users through webcam detection. The implementation uses computer vision for face detection and WebSocket communication to sync movements with your existing lipsync system.

Architecture Overview

```
 ┌─────────────────────────────────────┐
 ┌────────────────────────────────────────┐
 ┌─────────────────────────────────────┐
│ Webcam Input  │────────▶│ Python Backend  │────────▶│ Frontend (3D) │
│  (OpenCV)     │   │ (Face Tracking) │   │  (Three.js)   │
 └─────────────────────────────────────┘
 └────────────────────────────────────────┘
 └─────────────────────────────────────┘

                 │                │
                 ▼                ▼
          WebSocket Server     Avatar Animation
           (Movement Data)     (Eyes, Head, Body)
```

Prerequisites

- Python 3.8+
- Node.js 16+
- Webcam access
- Your existing Aisha avatar setup

Step 1: Set Up the Python Environment

1.1 Create Project Structure

```
aisha-avatar-tracking/
 ├─── backend/
│   ├─── face_tracker.py
│   ├─── avatar_controller.py
│   ├─── websocket_server.py
│   └─── requirements.txt
 ├─── frontend/
│   ├─── avatar_tracking.js
│   └─── integration.js
 └─── config/
     └─── tracking_config.json
```

1.2 Install Dependencies

Create requirements.txt:

```
opencv-python==4.8.1
mediapipe==0.10.8
websockets==12.0
numpy==1.24.3
```

Install:

```
cd backend
pip install -r requirements.txt
```

Step 2: Implement Face Detection

2.1 Create Face Tracker Module

Create backend/face_tracker.py:

```
import cv2
import mediapipe as mp
import numpy as np
from typing import Optional, Dict

class FaceTracker:
    """Handles webcam capture and face detection using MediaPipe."""

    def __init__(self, camera_index: int = 0):
        # Initialize MediaPipe Face Detection
        self.mp_face_detection = mp.solutions.face_detection
        self.face_detection = self.mp_face_detection.FaceDetection(
            model_selection=1,  # 0: short-range, 1: full-range
            min_detection_confidence=0.5
        )

        # Initialize camera
        self.cap = cv2.VideoCapture(camera_index)
        self.cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
        self.cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)

        # Smoothing parameters
        self.smooth_factor = 0.15
        self.current_position = {'x': 0.5, 'y': 0.5, 'z': 0.5}
        self.detection_confidence = 0.0

    def get_face_position(self) -> Optional[Dict]:
        """
        Capture frame and detect face position.
        Returns normalized coordinates (0-1) or None if no face detected.
        """
        ret, frame = self.cap.read()
        if not ret:
            return None

        # Convert BGR to RGB for MediaPipe
        rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        results = self.face_detection.process(rgb_frame)

        if results.detections and len(results.detections) > 0:
            # Get the first (most confident) detection
            detection = results.detections[0]
            bbox = detection.location_data.relative_bounding_box

            # Calculate center of face
            center_x = bbox.xmin + (bbox.width / 2)
            center_y = bbox.ymin + (bbox.height / 2)

            # Estimate depth based on face size
            face_size = bbox.width * bbox.height
            center_z = min(1.0, face_size * 4)  # Normalize to 0-1

            # Apply smoothing for stable tracking
            self.current_position['x'] += (center_x - self.current_position['x']) *
self.smooth_factor
            self.current_position['y'] += (center_y - self.current_position['y']) *
self.smooth_factor
            self.current_position['z'] += (center_z - self.current_position['z']) *
self.smooth_factor

            self.detection_confidence = detection.score[0] if detection.score else
0.5

            return {
                'x': self.current_position['x'],
                'y': self.current_position['y'],
                'z': self.current_position['z'],
                'confidence': self.detection_confidence,
                'detected': True
            }

        return {'detected': False, 'confidence': 0.0}

    def release(self):
        """Clean up resources."""
        self.cap.release()
        cv2.destroyAllWindows()
```

Step 3: Create Avatar Movement Controller

3.1 Avatar Controller Module

Create backend/avatar_controller.py:

```
import numpy as np
import time
from typing import Dict

class AvatarController:
    """Converts face positions to avatar movement commands."""

    def __init__(self):
        # Current rotations
        self.body_rotation = {'y': 0}
        self.head_rotation = {'x': 0, 'y': 0}
        self.eye_rotation = {'x': 0, 'y': 0}

        # Movement limits (in degrees)
        self.limits = {
            'body': {'y': 45},
            'head': {'x': 30, 'y': 25},
            'eye': {'x': 20, 'y': 15}
        }

        # Smoothing factors (0-1, higher = smoother)
        self.smoothing = {
            'body': 0.08,
            'head': 0.12,
            'eye': 0.25
        }

        # Idle animation parameters
        self.idle_time_start = time.time()
        self.last_detection_time = time.time()

    def calculate_movements(self, face_data: Dict) -> Dict:
        """
        Convert face position to avatar movements.
        Returns rotation values for body, head, and eyes.
        """
        current_time = time.time()

        if not face_data.get('detected'):
            # No face detected - switch to idle animation after 2 seconds
            if current_time - self.last_detection_time > 2.0:
                return self._get_idle_animation(current_time)
            # Return to center gradually
            return self._return_to_center()

        self.last_detection_time = current_time

        # Convert normalized coordinates to centered coordinates (-1 to 1)
        norm_x = (face_data['x'] - 0.5) * 2
        norm_y = (face_data['y'] - 0.5) * 2
        norm_z = face_data.get('z', 0.5)

        # Calculate target rotations based on face position
        movements = self._calculate_target_rotations(norm_x, norm_y, norm_z)

        # Apply smoothing
        self._apply_smoothing(movements)

        # Add micro-movements for realism
        self._add_micro_movements(current_time)

        return self._format_output()

    def _calculate_target_rotations(self, x: float, y: float, z: float) -> Dict:
        """Calculate target rotations based on normalized face position."""
        targets = {}

        # Body rotation (only horizontal, activates when face near edges)
        if abs(x) > 0.3:
            targets['body_y'] = x * self.limits['body']['y']
        else:
            targets['body_y'] = 0

        # Head rotation (follows face more closely)
        targets['head_x'] = x * self.limits['head']['x']
        targets['head_y'] = -y * self.limits['head']['y']  # Negative for natural
movement

        # Eye rotation (most responsive, looks beyond head)
        targets['eye_x'] = x * self.limits['eye']['x'] * 1.5
        targets['eye_y'] = -y * self.limits['eye']['y'] * 1.5

        # Adjust for distance (z-axis)
        distance_factor = 1.0 + (0.5 - z) * 0.3
        for key in targets:
            targets[key] *= distance_factor

        return targets

    def _apply_smoothing(self, targets: Dict):
        """Apply smoothing to prevent jittery movements."""
        # Body smoothing
        if 'body_y' in targets:
            self.body_rotation['y'] += (
                targets['body_y'] - self.body_rotation['y']
            ) * self.smoothing['body']

        # Head smoothing
        if 'head_x' in targets:
            self.head_rotation['x'] += (
                targets['head_x'] - self.head_rotation['x']
            ) * self.smoothing['head']
        if 'head_y' in targets:
            self.head_rotation['y'] += (
                targets['head_y'] - self.head_rotation['y']
            ) * self.smoothing['head']

        # Eye smoothing
        if 'eye_x' in targets:
            self.eye_rotation['x'] += (
                targets['eye_x'] - self.eye_rotation['x']
            ) * self.smoothing['eye']
        if 'eye_y' in targets:
            self.eye_rotation['y'] += (
                targets['eye_y'] - self.eye_rotation['y']
            ) * self.smoothing['eye']

    def _add_micro_movements(self, current_time: float):
        """Add subtle natural movements."""
        # Subtle breathing motion
        breathing = np.sin(current_time * 0.3) * 0.5
        self.head_rotation['y'] += breathing

        # Micro eye movements
        eye_drift_x = np.sin(current_time * 1.7) * 0.3
        eye_drift_y = np.cos(current_time * 2.1) * 0.2
        self.eye_rotation['x'] += eye_drift_x
        self.eye_rotation['y'] += eye_drift_y

    def _get_idle_animation(self, current_time: float) -> Dict:
        """Generate idle animation when no face detected."""
        t = current_time - self.idle_time_start

        return {
            'body': {'y': np.sin(t * 0.1) * 5},
            'head': {
                'x': np.sin(t * 0.15) * 8,
                'y': np.cos(t * 0.2) * 5
            },
            'eyes': {
                'x': np.sin(t * 0.3) * 10,
                'y': np.cos(t * 0.25) * 5
            },
            'blink': np.random.random() < 0.005  # Random blinking
        }

    def _return_to_center(self) -> Dict:
        """Gradually return to center position."""
        decay = 0.05
        self.body_rotation['y'] *= (1 - decay)
        self.head_rotation['x'] *= (1 - decay)
        self.head_rotation['y'] *= (1 - decay)
        self.eye_rotation['x'] *= (1 - decay)
        self.eye_rotation['y'] *= (1 - decay)

        return self._format_output()

    def _format_output(self) -> Dict:
        """Format the output for sending to frontend."""
        return {
            'body': {'y': self.body_rotation['y']},
            'head': {
                'x': self.head_rotation['x'] - self.body_rotation['y'] * 0.3,
                'y': self.head_rotation['y']
            },
            'eyes': {
                'x': self.eye_rotation['x'] - self.head_rotation['x'] * 0.5,
                'y': self.eye_rotation['y'] - self.head_rotation['y'] * 0.5
            },
            'blink': np.random.random() < 0.008
        }
```

Step 4: Set Up WebSocket Server

4.1 WebSocket Server

Create backend/websocket_server.py:

```
import asyncio
import json
import websockets
import logging
from face_tracker import FaceTracker
from avatar_controller import AvatarController

# Set up logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class TrackingServer:
    def __init__(self, host='localhost', port=8765):
        self.host = host
        self.port = port
        self.clients = set()

    async def handle_client(self, websocket, path):
        """Handle individual client connection."""
        self.clients.add(websocket)
        logger.info(f"New client connected. Total clients: {len(self.clients)}")
        tracker = FaceTracker()
        controller = AvatarController()

        try:
            while True:
                # Get face position
                face_data = tracker.get_face_position()

                # Calculate avatar movements
                movements = controller.calculate_movements(face_data)

                # Send to client
                await websocket.send(json.dumps(movements))

                # Control frame rate (30 FPS)
                await asyncio.sleep(1/30)

        except websockets.exceptions.ConnectionClosed:
            logger.info("Client disconnected")
        except Exception as e:
            logger.error(f"Error: {e}")
        finally:
            self.clients.discard(websocket)
            tracker.release()

    async def start(self):
        """Start the WebSocket server."""
        logger.info(f"Starting tracking server on {self.host}:{self.port}")
        async with websockets.serve(self.handle_client, self.host, self.port):
            await asyncio.Future()  # Run forever

if __name__ == "__main__":
    server = TrackingServer()
    asyncio.run(server.start())
```

Step 5: Frontend Integration

5.1 Avatar Tracking Client

Create frontend/avatar_tracking.js:

```
// avatar_tracking.js
export class AvatarTracking {
    constructor(avatarModel, config = {}) {
        this.avatar = avatarModel;
        this.config = {
            wsUrl: config.wsUrl || 'ws://localhost:8765',
            enableBlinking: config.enableBlinking !== false,
            enableMicroMovements: config.enableMicroMovements !== false,
            ...config
        };
        this.ws = null;
        this.isConnected = false;
        this.bones = {};
        this.morphTargets = {};

        this.initialize();
    }

    initialize() {
        // Find and store bone references
        this.findBones();

        // Find morph targets for blinking
        this.findMorphTargets();

        // Connect to WebSocket
        this.connectWebSocket();
    }

    findBones() {
        // Standard Ready Player Me bone names
        const boneNames = {
            body: ['Hips', 'mixamorigHips'],
            spine: ['Spine', 'mixamorigSpine'],
            head: ['Head', 'mixamorigHead'],
            neck: ['Neck', 'mixamorigNeck'],
            leftEye: ['LeftEye', 'mixamorigLeftEye'],
            rightEye: ['RightEye', 'mixamorigRightEye']
        };

        for (const [key, names] of Object.entries(boneNames)) {
            for (const name of names) {
                const bone = this.avatar.getObjectByName(name);
                if (bone) {
                    this.bones[key] = bone;
                    console.log(`Found bone: ${key} -> ${name}`);
                    break;
                }
            }
        }
    }

    findMorphTargets() {
        // Find mesh with morph targets
        this.avatar.traverse((child) => {
            if (child.isMesh && child.morphTargetDictionary) {
                this.morphTargets = {
                    mesh: child,
                    dictionary: child.morphTargetDictionary,
                    influences: child.morphTargetInfluences
                };
                console.log('Found morph targets:',
Object.keys(child.morphTargetDictionary));
            }
        });
    }

    connectWebSocket() {
        console.log('Connecting to tracking server...');

        this.ws = new WebSocket(this.config.wsUrl);

        this.ws.onopen = () => {
            console.log('Connected to tracking server');
            this.isConnected = true;
        };

        this.ws.onmessage = (event) => {
            try {
                const data = JSON.parse(event.data);
                this.updateAvatar(data);
            } catch (error) {
                console.error('Error parsing tracking data:', error);
            }
        };

        this.ws.onerror = (error) => {
            console.error('WebSocket error:', error);
        };

        this.ws.onclose = () => {
            console.log('Disconnected from tracking server');
            this.isConnected = false;

            // Attempt to reconnect after 3 seconds
            setTimeout(() => {
                if (!this.isConnected) {
                    this.connectWebSocket();
                }
            }, 3000);
        };
    }
```

```
updateAvatar(trackingData) {
    // Update body rotation
    if (this.bones.body && trackingData.body) {
        this.bones.body.rotation.y = this.degToRad(trackingData.body.y);
    }

    // Update spine for more natural movement
    if (this.bones.spine && trackingData.body) {
        this.bones.spine.rotation.y = this.degToRad(trackingData.body.y * 0.3);
    }

    // Update head rotation
    if (this.bones.head && trackingData.head) {
        this.bones.head.rotation.x = this.degToRad(trackingData.head.y);
        this.bones.head.rotation.y = this.degToRad(trackingData.head.x);
    }

    // Update neck for smoother transition
    if (this.bones.neck && trackingData.head) {
        this.bones.neck.rotation.x = this.degToRad(trackingData.head.y * 0.3);
        this.bones.neck.rotation.y = this.degToRad(trackingData.head.x * 0.3);
    }

    // Update eye rotation
    this.updateEyes(trackingData.eyes);

    // Handle blinking
    if (trackingData.blink && this.config.enableBlinking) {
        this.blink();
    }
}

updateEyes(eyeData) {
    if (!eyeData) return;

    const eyeX = this.degToRad(eyeData.x);
    const eyeY = this.degToRad(eyeData.y);

    // Update eye bones if available
    if (this.bones.leftEye) {
        this.bones.leftEye.rotation.x = eyeY;
        this.bones.leftEye.rotation.y = eyeX;
    }

    if (this.bones.rightEye) {
        this.bones.rightEye.rotation.x = eyeY;
        this.bones.rightEye.rotation.y = eyeX + this.degToRad(2); // Slight offset
```

```
for realism
        }
    }

    blink() {
        if (!this.morphTargets.dictionary) return;

        // Common morph target names for blinking
        const blinkTargets = [
            'eyesClosed',
            'eyesClosedLeft',
            'eyesClosedRight',
            'blink',
            'Blink_Left',
            'Blink_Right'
        ];

        const targetIndices = [];

        for (const target of blinkTargets) {
            if (this.morphTargets.dictionary[target] !== undefined) {
                targetIndices.push(this.morphTargets.dictionary[target]);
            }
        }

        if (targetIndices.length === 0) return;

        // Close eyes
        targetIndices.forEach(index => {
            this.morphTargets.influences[index] = 1;
        });

        // Open eyes after 150ms
        setTimeout(() => {
            targetIndices.forEach(index => {
                this.morphTargets.influences[index] = 0;
            });
        }, 150);
    }

    degToRad(degrees) {
        return (degrees * Math.PI) / 180;
    }

    disconnect() {
        if (this.ws) {
            this.ws.close();
            this.ws = null;
        }
    }
}
```

5.2 Integration with Existing System

Create frontend/integration.js:

```
// integration.js
import { AvatarTracking } from './avatar_tracking.js';

export class AishaAvatarController {
    constructor(scene, avatarModel) {
        this.scene = scene;
        this.avatarModel = avatarModel;
        this.tracking = null;
        this.lipsyncActive = false;

        this.initializeTracking();
    }

    initializeTracking() {
        // Initialize tracking with custom config
        this.tracking = new AvatarTracking(this.avatarModel, {
            wsUrl: 'ws://localhost:8765',
            enableBlinking: true,
            enableMicroMovements: true
        });
    }

    // Call this when lipsync starts
    onLipsyncStart() {
        this.lipsyncActive = true;
        // Optionally reduce head movement during speech
        if (this.tracking) {
            this.tracking.config.reducedMovement = true;
        }
    }

    // Call this when lipsync ends
    onLipsyncEnd() {
        this.lipsyncActive = false;
        if (this.tracking) {
            this.tracking.config.reducedMovement = false;
        }
    }

    // Update loop (call in your render loop)
    update(deltaTime) {
        // Tracking updates automatically via WebSocket
        // Add any additional logic here
    }

    dispose() {
        if (this.tracking) {
            this.tracking.disconnect();
        }
    }
}

// Example usage in your main app
export function initializeAisha(scene) {
    // Load your avatar model (example with GLTFLoader)
    const loader = new THREE.GLTFLoader();

    loader.load('path/to/aisha-avatar.glb', (gltf) => {
        const avatarModel = gltf.scene;
        scene.add(avatarModel);

        // Initialize controller with tracking
        const controller = new AishaAvatarController(scene, avatarModel);

        // Return controller for external access
        return controller;
    });
}
```

Step 6: Configuration

6.1 Create Configuration File

Create config/tracking_config.json:

```
{
    "tracking": {
        "camera_index": 0,
        "fps": 30,
        "detection_confidence": 0.5,
        "smoothing": {
            "position": 0.15,
            "body": 0.08,
            "head": 0.12,
            "eyes": 0.25
        }
    },
    "movements": {
        "limits": {
            "body_rotation": 45,
            "head_rotation_x": 30,
            "head_rotation_y": 25,
            "eye_rotation_x": 20,
            "eye_rotation_y": 15
        },
        "idle_animation": {
            "enabled": true,
            "activation_delay": 2.0,
            "intensity": 0.5
        }
    },
    "websocket": {
        "host": "localhost",
        "port": 8765,
        "reconnect_delay": 3000
    },
    "features": {
        "blinking": {
            "enabled": true,
            "frequency": 0.008,
            "duration": 150
        },
        "micro_movements": {
            "enabled": true,
            "breathing_rate": 0.3,
            "eye_drift": 0.3
        }
    }
}
```

Step 7: Running the System

7.1 Start Script

Create start_tracking.py:

```
#!/usr/bin/env python3
import sys
import asyncio
from backend.websocket_server import TrackingServer

def main():
    print("Starting Aisha Avatar Tracking System...")
    print("Make sure your webcam is connected.")
    print("Press Ctrl+C to stop.")

    try:
        server = TrackingServer()
        asyncio.run(server.start())
    except KeyboardInterrupt:
        print("\nShutting down tracking system...")
        sys.exit(0)
```

```
if __name__ == "__main__":
    main()
```

7.2 Launch Commands

Terminal 1 - Start tracking server:

```
python start_tracking.py
```

Terminal 2 - Start your existing Aisha app:

```
# Your existing launch command
npm start  # or however you start your frontend
```

Step 8: Testing and Debugging

8.1 Test Script

Create test_tracking.py:

```
import cv2
from backend.face_tracker import FaceTracker
from backend.avatar_controller import AvatarController
import json

def test_tracking():
    tracker = FaceTracker()
    controller = AvatarController()

    print("Starting tracking test. Press 'q' to quit.")

    while True:
        face_data = tracker.get_face_position()
        movements = controller.calculate_movements(face_data)

        # Print formatted output
        print(f"\rTracking: {json.dumps(movements, indent=2)}", end="")

        # Show camera feed (optional)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    tracker.release()

if __name__ == "__main__":
    test_tracking()
```

Step 9: Performance Optimization

9.1 Optimization Tips

1. **Reduce Detection Frequency**:
   # Only detect every 3rd frame
2. if frame_count % 3 == 0:
3.   face_data = tracker.get_face_position()
4.
5. **Use Threading**:

```
    import threading
 6.
 7. class ThreadedTracker:
 8.     def __init__(self):
 9.         self.latest_position = None
10.         self.thread = threading.Thread(target=self._track_loop)
11.         self.thread.daemon = True
12.         self.thread.start()
13.
```

14. **Implement Level of Detail (LOD)**:

```
    // Reduce update frequency when avatar is far from camera
15. const distance = camera.position.distanceTo(avatar.position);
16. const updateFrequency = distance > 10 ? 0.5 : 1.0;
17.
```

Step 10: Troubleshooting

Common Issues and Solutions

Issue	Solution
No face detected	Check lighting, ensure face is centered in frame
Jittery movements	Increase smoothing factors in config
WebSocket connection fails	Check firewall, ensure ports are open
High CPU usage	Reduce FPS, use frame skipping
Avatar bones not found	Check bone naming, use traverse to debug

Deployment Considerations

1. **Production WebSocket**: Use WSS (WebSocket Secure) with proper SSL certificates
2. **Error Handling**: Implement comprehensive error recovery
3. **Privacy**: Add user consent for camera access
4. **Performance Monitoring**: Add metrics logging
5. **Cross-browser Support**: Test on Chrome, Firefox, Safari

Next Steps

- Add emotion detection using facial landmarks
- Implement gesture recognition
- Add multi-user tracking support
- Create a calibration interface
- Add voice activity detection integration

This implementation provides a solid foundation for adding natural, responsive eye tracking and body movement to your Aisha avatar while maintaining compatibility with your existing lipsync system.