BRR
Technology Solutions

SUPPORT DOCUMENT

Project Name:
Interactive Game Engines and Character Control (Real-Time Skeleton-Based Human Movement Detection and Unity Integration)

- Camera as a Landmark -

Date: November 2024

Industry Details

Company Name: BRR Technology Solutions
ABN: 93475772006
Company address: 21/296 Marrickville Road, Marrickville NSW 2204, Sydney, Australia
Company Profile: Full-service technology Consultancy company delivering high performing and advanced functionality software solutions for growing companies.
Website: https://brrtechnology.com

Industry Assigned Supervisor Detail

Contact Name: Raafat Alsameraai
Email ID: raafat@brrtechnology.com

Intern Name: (Terence) Wanzhong Wu

Internship Start Date: 11 Nov. 2024
Internship End Date: 31 Jan. 2025

Phase: Software Development System

The camera in Unity can act as a landmark or a reference point in your character animation workflow, especially if we are integrating real-time pose data captured from a laptop camera via Python and UDP communication.

1. Landmark

In this setup, the Unity camera can be used as:

- A reference frame for the character's movements:
  - If the pose data from the laptop camera corresponds to a specific view, the Unity camera can be aligned to match this perspective.
- An anchor point:
  - Movements and poses can be relative to the Unity camera's position, ensuring consistency between the real-world feed and the virtual environment.

2. Workflow Integration

- Pose Coordinates Input:
  - The laptop camera captures pose data (e.g., via OpenPose or MediaPipe) and sends it to Unity over UDP.
  - These coordinates can be interpreted in Unity as offsets or transforms for character bones or animations.
- Camera Alignment:
  - To achieve accurate mapping, the Unity camera's field of view (FOV), orientation, and position should mimic the real-world camera capturing the pose data.
  - For example, the Unity camera could be placed in the same position relative to the character model as the real camera is to the person being captured.

3. Real-Time Pose Visualization

- Unity's camera feed can:
  - Act as a viewport to visualize the character's movements that correspond to the real-world input.
  - Help debug the alignment and scaling of pose data with the character model.

4. Practical Use Cases

- Motion Capture:

    - The Unity camera observes how pose data drives the character's animation.
- Augmented Reality:
    - Overlay real-world pose inputs on virtual characters, using the Unity camera as a blending point.
- Interactive Systems:
    - Align the Unity camera with the real-world laptop camera to enhance realism in pose-driven gameplay or simulations.

5. Implementation Tips

- Coordinate System Mapping:
    - Ensure the pose data from Python matches Unity's coordinate system (e.g., flip the Y-axis if needed).
- Scaling:
    - Normalize the pose data to fit the character model's proportions.
- Camera Calibration:
    - Match the Unity camera's intrinsic parameters (like FOV) with the real camera to reduce distortions.

To match the Unity camera with the laptop camera, especially when dealing with variations in user proximity and visibility, you need to align **camera calibration**, **field of view (FOV)**, and **real-time transformations**.

1. Camera Calibration

- Match Intrinsic Parameters: Calibrate the laptop camera using tools like OpenCV to determine its focal length, sensor size, and distortion coefficients.
    - Python's OpenCV can capture intrinsic parameters using a checkerboard or ArUco markers.
    - In Unity, configure the camera's FOV and aspect ratio to match the laptop camera's parameters.
    - Example:
        - Laptop camera FOV: 60 degrees
        - Unity camera: Set FOV to 60 in the camera properties.
- Distortion Correction: Use the distortion coefficients from OpenCV to undistort the pose data before sending it to Unity.

2. Coordinate System Mapping

- Match 2D Pose Coordinates:
    - Convert the 2D pose data captured by the laptop camera to 3D coordinates in Unity using depth assumptions or real-time depth data (if available).
- Account for Axes Differences:
    - OpenCV and Unity use different coordinate systems (e.g., Y-axis might need to be flipped).

3. Depth and Perspective Adjustments

- When the user moves closer to or farther from the laptop camera, their apparent size changes. You must dynamically adjust:
    - Scale in Unity:
        - Calculate a scaling factor based on the distance between key body points (e.g., shoulders or hips) in the 2D frame.
        - Example: Use the ratio of the user's width in pixels (on the laptop camera) to the expected width of the virtual character.
    - Position:
        - Translate the character in Unity along the Z-axis (depth) based on the user's distance from the camera.
        - Use real-world units or inferred depth (e.g., from a stereo camera or pre-calibrated data).

4. Handling Partial Visibility

- Detect Missing Body Parts:
    - If the user's pose is partially visible (e.g., only upper body), infer the missing joints using:
        - Pose Estimation Models: Models like OpenPose or MediaPipe can predict missing keypoints.
        - Default Animations: Use fallback animations for missing data (e.g., stationary legs if feet are out of frame).
- Adjust Field of View Dynamically:
    - Use a script in Unity to zoom in or out (adjust FOV) based on the bounding box of the detected pose.
    - Example: If the user moves forward, reduce FOV or move the Unity camera backward.

5. Real-Time Synchronization

- UDP Communication:
    - Ensure that pose data is sent with minimal latency. A fixed frame rate (e.g., 30 FPS) helps synchronize real-world motion with Unity animations.
- Camera Offset:
    - If the Unity camera is not aligned with the user's view direction, apply an offset transform to ensure correct alignment.

6. Testing and Calibration

- Use test cases where users move closer, farther, and partially out of frame.
- Record how the laptop camera captures the user and adjust Unity's camera properties (e.g., FOV, near/far clipping planes) until the virtual representation matches the real-world perspective.

 # Sample Unity Script to Adjust Camera Dynamically:

```
using UnityEngine;

public class CameraAdjuster : MonoBehaviour
{
    public Camera unityCamera;
    public Transform character;
    public float baseDistance = 2.0f; // Adjust based on initial calibration
    public float scaleFactor = 1.0f;

    void Update()
    {
        // Simulate depth adjustment based on pose data (replace with actual UDP input)
        float userDistance = GetUserDistanceFromPose(); // Function to calculate distance
        float newScale = scaleFactor * userDistance / baseDistance;

        // Adjust character scale
        character.localScale = new Vector3(newScale, newScale, newScale);

        // Adjust Unity camera position (optional: zoom out)
        unityCamera.transform.position = new Vector3(
            unityCamera.transform.position.x,
            unityCamera.transform.position.y,
            -userDistance
        );
    }

    float GetUserDistanceFromPose()
    {
        // Placeholder: Calculate user distance based on pose landmarks
        return 2.0f; // Example fixed distance
    }
```

```
}
```

To handle different camera types dynamically, we can calibrate and align the Field of View (FOV) automatically.

1. Camera Calibration in Python (OpenCV)

Use OpenCV to calculate the intrinsic parameters of the laptop camera. These include:

- Focal length (fx, fy)
- Principal point (Cx,Cy)
- Distortion coefficients.

 # Python Code for Calibration:

```
import cv2
import numpy as np

# Calibration parameters
CHECKERBOARD = (6, 9)  # Checkerboard size
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# Arrays to store object points and image points
objpoints = []  # 3D points
imgpoints = []  # 2D points

# Generate 3D points for the checkerboard
objp = np.zeros((CHECKERBOARD[0] * CHECKERBOARD[1], 3), np.float32)
objp[:, :2] = np.mgrid[0:CHECKERBOARD[1], 0:CHECKERBOARD[0]].T.reshape(-1, 2)

# Capture images and find corners
cap = cv2.VideoCapture(0)
while True:
    ret, frame = cap.read()
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    ret, corners = cv2.findChessboardCorners(gray, CHECKERBOARD, None)
    if ret:
        objpoints.append(objp)
        corners2 = cv2.cornerSubPix(gray, corners, (11, 11), (-1, -1), criteria)
        imgpoints.append(corners2)
        cv2.drawChessboardCorners(frame, CHECKERBOARD, corners2, ret)
        cv2.imshow("Calibration", frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
cap.release()
```

```
cv2.destroyAllWindows()

# Camera calibration
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1], None, None)

print("Camera Matrix:", mtx)
print("Distortion Coefficients:", dist)
```

- **Camera Matrix (`mtx`):**

$$mtx = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

  - f_x, f_y: Focal lengths.
  - c_x, c_y: Principal point.

2. Send Calibration Data to Unity

Once you have the calibration data, send it to Unity via UDP or a file.

Example, of JSON data:

```
{
  "focal_length_x": 800.0,
  "focal_length_y": 800.0,
  "principal_point_x": 640.0,
  "principal_point_y": 360.0
}
```

3. Adjust FOV in Unity

Convert the focal length to FOV using this formula:

$$\text{FOV (in degrees)} = 2 \cdot \arctan\left(\frac{\text{sensor height}}{2 \cdot f_y}\right) \cdot \frac{180}{\pi}$$

 # Unity script to adjust FOV:

```
using UnityEngine;
using System;
```

```
public class CameraCalibration : MonoBehaviour
{
   public Camera unityCamera;

   // Simulated data from calibration
   public float focalLengthY = 800.0f; // Replace with actual data
   public float sensorHeight = 24.0f; // Sensor height in mm (or normalize as needed)

   void Start()
   {
      AdjustCameraFOV();
   }

   void AdjustCameraFOV()
   {
      // Calculate FOV in degrees
      float fov = 2 * Mathf.Atan(sensorHeight / (2 * focalLengthY)) * Mathf.Rad2Deg;
      unityCamera.fieldOfView = fov;
      Debug.Log("Adjusted FOV: " + fov);
   }
}
```

Dynamic Handling: Replace focalLengthY and sensorHeight with values received from the calibration JSON.

4. Dynamic Adjustment for Different Cameras

- **Detect Camera Properties in Python**: Use OpenCV to dynamically detect the laptop camera's resolution, sensor size, and focal length.
- **Normalize Data for Unity**:
    - Use a standard sensor height (e.g., 24 mm for full-frame cameras) or detect it if the camera provides metadata.
    - Adjust Unity's FOV based on the laptop camera's resolution and the viewport dimensions.

5. Handling User Movement

To handle user movement dynamically:

- **Track Body Size**: Calculate the bounding box of the detected pose (e.g., width of shoulders) and use it to estimate the user's distance from the camera.
- **Update Camera Position**: Move the Unity camera backward/forward based on the inferred distance.

 # Example in Unity:

```
void UpdateCameraDepth(float userDistance)
{
   float defaultDistance = 2.0f; // Default distance in meters
   unityCamera.transform.position = new Vector3(
      unityCamera.transform.position.x,
      unityCamera.transform.position.y,
      -userDistance + defaultDistance
   );
}
```

For most real-time pose estimation and Unity animation projects, calibrating the camera for **forwards and backwards movements (depth)** is typically sufficient to ensure that the character matches the user's relative position in the virtual environment. However, if the user **turns their body to the right or left**, additional calibration or tracking is required to maintain realistic alignment and prevent distortions.

How to Handle Body Turning in Unity

To handle body turning (rotation along the Y-axis) alongside forwards and backwards movement, consider the following approaches:

1. Track Body Orientation

- Use the detected **pose landmarks** to estimate the user's orientation relative to the camera.
    - Example: Compare the horizontal distance between the shoulders (left_shoulder and right_shoulder).
    - A significant difference in X-coordinates indicates body rotation.
- **Implementation**:
    - In Python (e.g., using MediaPipe or OpenPose), calculate the angle of rotation:

```
import math

def calculate_body_orientation(left_shoulder, right_shoulder):
   # Assuming left_shoulder and right_shoulder are (x, y) tuples
   dx = right_shoulder[0] - left_shoulder[0]
   dy = right_shoulder[1] - left_shoulder[1]
   angle = math.atan2(dy, dx) * (180 / math.pi)  # Angle in degrees
   return angle
```

- Send this angle to Unity over UDP.

- In Unity, rotate the character model based on this angle:

```
void UpdateCharacterRotation(float bodyAngle)
{
   // Smoothly rotate the character
   Quaternion targetRotation = Quaternion.Euler(0, bodyAngle, 0);
   character.transform.rotation = Quaternion.Slerp(
      character.transform.rotation,
      targetRotation,
      Time.deltaTime * rotationSpeed
   );
}
```

2. Use Depth for Perspective Adjustments

- For body turning, depth data can help identify whether the user is leaning or twisting, not just moving forwards or backwards.
- Combine depth with pose landmarks:
  - If depth data shows asymmetry (e.g., one shoulder is farther than the other), adjust the Unity character's rotation to match.
- **Tools**:
  - Stereo cameras (e.g., Intel RealSense).
  - AI models like OpenPose 3D or MediaPipe Holistic (if 3D pose estimation is available).

3. Dynamically Adjust FOV for Rotations

- If body turning introduces extreme perspective shifts, dynamically adjust the FOV in Unity to better match the user's view.
- Use a larger FOV for side profiles and a smaller FOV for front-facing poses.
- Unity Script Example:

```
void AdjustCameraFOV(float bodyAngle)
{
   // Increase FOV for side profiles
   float newFOV = Mathf.Lerp(minFOV, maxFOV, Mathf.Abs(bodyAngle) / 90.0f);
   unityCamera.fieldOfView = Mathf.Lerp(unityCamera.fieldOfView, newFOV,
Time.deltaTime * fovAdjustmentSpeed);
}
```

4. Optimize for Partial Visibility

- When the user turns, parts of their body (e.g., arms or legs) may go out of frame.
- Handle missing joints by:
    - **Extrapolation**: Predict the missing joints based on existing pose data.
    - **Fallback Animations**: Use default animations or static poses for missing parts.

Conclusion

- **Forwards and backwards movements** (depth adjustments) are sufficient for simple scenarios.
- **Body turning (rotation)** becomes critical in interactive systems like games, VR, or applications requiring high realism.
- To account for body turning:
    - Track pose landmarks and estimate body orientation.
    - Use depth and symmetry to refine character rotation.
    - Dynamically adjust the Unity camera's FOV and alignment for better perspective.

By combining these techniques, your system will handle both **linear movements** (forward/backward) and **rotational movements** (left/right turning) effectively, creating a smooth and immersive experience.

Full Workflow Summary

1. **Calibrate** the camera in Python and send intrinsic data (focal length, principal point) to Unity.
2. **Adjust Unity Camera FOV** based on the calibration data.
3. Use pose detection and bounding box scaling to **handle user proximity** dynamically.
4. **Synchronize movements** between the real and virtual environments using UDP.

This approach ensures the Unity camera aligns accurately with any laptop camera while automatically adapting to different camera types and user behaviors.