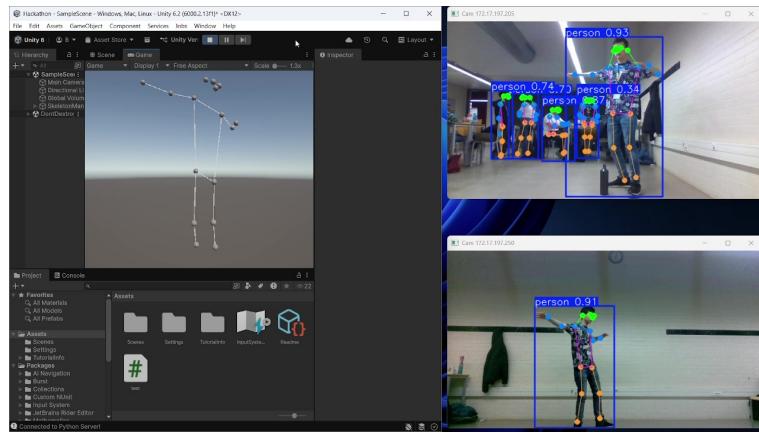


RESOURCES



Documentació Hack EPS 2025: Ingroup

Autors: MARC AIGUADÉ i BRUNO RIOS

22/11/2025-23/11/2025

Índice general

0.1.	Introducción	2
0.2.	Arquitectura General del Sistema	2
0.3.	Módulo <code>sender.py</code>	2
0.3.1.	Captura de vídeo	3
0.3.2.	Control de tasa de frames	3
0.3.3.	Codificación y envío de frames	3
0.3.4.	Gestión de reconexiones	3
0.4.	Módulo <code>receiver.py</code>	3
0.4.1.	Recepción de frames de múltiples cámaras	4
0.4.2.	Configuración geométrica de las cámaras	4
0.4.3.	Inferencia de pose 2D con YOLOv8	5
0.4.4.	Triangulación 3D	5
0.4.5.	Registro en CSV y envío a Unity	5
0.5.	Módulo <code>SkeletonVisualizer.cs</code> en Unity	6
0.5.1.	Conexión de red	6
0.5.2.	Construcción del esqueleto en la escena	6
0.5.3.	Mapeo de datos 3D a objetos Unity	7
0.6.	Flujo de Datos Completo	7
0.7.	Consideraciones y Limitaciones	8
0.7.1.	Calibración aproximada de cámaras	8
0.7.2.	Múltiples personas	8
0.7.3.	Rendimiento	8
0.8.	Conclusión	8

0.1. Introducción

Este documento describe el diseño e implementación de un sistema distribuido de captura y visualización 3D del esqueleto humano a partir de múltiples cámaras. El sistema está compuesto por los siguientes elementos principales:

- Varias cámaras que ejecutan el programa `sender.py` para capturar vídeo y enviarlo a través de la red.
- Un servidor central que ejecuta `receiver.py`, recibe los flujos de vídeo, aplica un modelo de visión por computador (YOLOv8 pose) para estimar los puntos clave (keypoints) 2D, y realiza triangulación para obtener coordenadas 3D.
- Un proyecto de Unity que ejecuta el script `SkeletonVisualizer.cs`, se conecta al servidor y muestra en tiempo real el esqueleto 3D reconstruido.

El objetivo final es visualizar en Unity, en tiempo real, el esqueleto de la persona detectada a partir de las imágenes capturadas por las cámaras, aprovechando la inferencia de una IA de pose humana (YOLOv8) y la geometría de cámaras para recuperar información tridimensional.

0.2. Arquitectura General del Sistema

La arquitectura del sistema sigue un patrón cliente–servidor distribuido (ver Figura 1):

- Cada cámara actúa como cliente TCP, ejecutando `sender.py` y conectándose al servidor de vídeo en el puerto 9999.
- El servidor central ejecuta `receiver.py`, que levanta un servidor TCP para recibir frames de múltiples cámaras y otro servidor TCP (puerto 5000) para enviar al cliente de Unity las coordenadas 3D calculadas.
- Unity actúa como cliente TCP, ejecutando `SkeletonVisualizer.cs` y conectándose al puerto 5000 para recibir las posiciones de los 17 puntos clave del esqueleto.

Figura 1: Arquitectura lógica del sistema: múltiples senders, un receiver y un cliente Unity.

0.3. Módulo `sender.py`

El módulo `sender.py` se ejecuta en cada ordenador conectado a una cámara. Su responsabilidad es capturar frames de vídeo, comprimirlos y enviarlos a través de TCP a la máquina que ejecuta `receiver.py`.

0.3.1. Captura de vídeo

Se utiliza OpenCV para acceder a la cámara local:

```
cap = cv2.VideoCapture(0)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 360)
```

De esta forma se fija la resolución a 640x360 píxeles, lo que reduce el ancho de banda necesario manteniendo una calidad suficiente para la detección de pose.

0.3.2. Control de tasa de frames

Para limitar la carga de red y de CPU, se implementa un control simple de FPS mediante un intervalo temporal:

- Se define una tasa objetivo (`target_fps = 10`).
- Se calcula un intervalo `frame_interval = 1.0 / target_fps`.
- Antes de capturar un nuevo frame se comprueba si ha transcurrido suficiente tiempo desde el frame anterior.

0.3.3. Codificación y envío de frames

Cada frame se comprime en JPEG utilizando OpenCV y se envía por un socket TCP:

1. Se codifica el frame en memoria con `cv2.imencode('.jpg', frame, ...)`.
2. Se obtiene el buffer binario y se calcula su longitud.
3. Se envían primero 8 bytes con el tamaño (formato "Q" de la librería `struct`), seguidos de los datos JPEG.

Este protocolo *tamaño + datos* permite al servidor reconstruir los frames de forma robusta, incluso cuando los datagramas TCP llegan fragmentados.

0.3.4. Gestión de reconexiones

Si el servidor no está disponible o la conexión se pierde, el `sender.py` captura excepciones de `socket.error` y `socket.timeout`, cierra el socket y reintenta la conexión después de un breve retardo. Esto permite que el sistema se recupere automáticamente de cortes temporales de red.

0.4. Módulo `receiver.py`

El módulo `receiver.py` centraliza la recepción de vídeo, la inferencia de pose 2D, la triangulación 3D y la difusión de resultados hacia Unity.

0.4.1. Recepción de frames de múltiples cámaras

`exttreceive.py` crea un servidor TCP en el puerto 9999 mediante `socket` estándar. Por cada conexión entrante se lanza un hilo dedicado (función `handle_client`) que:

- Lee los primeros 8 bytes para conocer el tamaño del frame.
- Acumula datos hasta completar el frame JPEG.
- Decodifica el JPEG en un array de imagen con `cv2.imread`.
- Guarda el último frame recibido por cada IP en el diccionario global `latest_frames` protegido por un lock.

Esto permite que el bucle principal del servidor siempre disponga del frame más reciente de cada cámara conectada.

0.4.2. Configuración geométrica de las cámaras

Para poder reconstruir coordenadas 3D a partir de observaciones 2D es necesario conocer la geometría de cada cámara. El código define una lista `CAMERAS` con la posición aproximada de cada cámara en un sistema de coordenadas global (en centímetros):

```
CAMERAS = [
    ('127.0.0.1', -200, 200, -200),
    ('192.168.1.101', 200, 200, -200),
]
```

Cada entrada contiene la IP de la cámara y sus coordenadas (X, Y, Z) . Se asume que el origen $(0, 0, 0)$ está en el centro del espacio de trabajo (por ejemplo, el centro del círculo donde se sitúa la persona) y que las cámaras están elevadas a 200 cm mirando hacia el centro.

La función `get_projection_matrix` construye para cada cámara su matriz de proyección $P = K[R | t]$, donde:

- K es la matriz intrínseca (focal aproximada y centro de imagen).
- R es la matriz de rotación que orienta la cámara hacia el origen.
- t es el vector de traslación $t = -RC$, con C el centro de la cámara en coordenadas del mundo.

Estas matrices se almacenan en el diccionario `CAM_CONFIG` indexado por IP, de modo que durante la triangulación se pueda acceder fácilmente a la matriz de cada cámara que ha detectado un punto concreto.

0.4.3. Inferencia de pose 2D con YOLOv8

En el bucle principal de `receiver.py`, el servidor recorre periódicamente el diccionario de frames más recientes `latest_frames` y, para cada cámara, ejecuta el modelo YOLOv8 de pose:

1. Se carga previamente el modelo con `YOLO('yolov8n-pose.pt')`, que proporciona 17 puntos clave por persona (nariz, ojos, hombros, caderas, rodillas, tobillos, etc.).
2. Sobre cada frame se llama a `model(frame)` para obtener las detecciones.
3. Se selecciona, por simplicidad, la primera persona detectada en cada cámara, y se extrae el array de keypoints de tamaño (17, 3) (coordenadas x , y en píxeles y confianza).
4. Se guardan estos keypoints en un diccionario `keypoints_2d` indexado por IP.

Además, para depuración, se muestra una ventana de OpenCV con el frame anotado que devuelve YOLOv8 (esqueleto dibujado sobre la imagen).

0.4.4. Triangulación 3D

Una vez que se dispone de keypoints 2D desde al menos dos cámaras con matrices de proyección conocidas, se puede obtener la posición 3D de cada punto clave mediante el algoritmo de Triangulación Lineal Directa (DLT).

Para cada índice de keypoint (de 0 a 16):

- Se recorre el conjunto de cámaras válidas y se recopilan todas las observaciones (u, v) del keypoint cuya confianza supere un umbral (por ejemplo, 0,5).
- Para cada observación y su matriz de proyección P se añaden dos filas a una matriz A de la forma:

$$uP_{3,:} - P_{1,:}, \quad vP_{3,:} - P_{2,:}$$

- Se resuelve el sistema homogéneo $AX = 0$ mediante Descomposición en Valores Singulares (SVD). El vector asociado al menor valor singular proporciona la solución $X = (X, Y, Z, W)^T$.
- Finalmente se normaliza X dividiendo por W para obtener las coordenadas 3D en el sistema de referencia global.

Si un keypoint no está visible en suficientes cámaras (menos de dos observaciones válidas), se marca como ausente (`None`) en la lista de puntos 3D.

0.4.5. Registro en CSV y envío a Unity

Tras calcular el esqueleto 3D, el servidor:

1. Escribe en un fichero CSV una fila por frame, con el tiempo, un ID de cuerpo (en este caso 0) y las coordenadas X, Y, Z de cada uno de los 17 puntos clave. Los puntos no disponibles se dejan en blanco.

2. Construye una cadena de texto con el siguiente formato para enviar a Unity:

timestamp, BodyID, $X_0, Y_0, Z_0, X_1, Y_1, Z_1, \dots, X_{16}, Y_{16}, Z_{16}$

separada por comas.

3. Llama a la función `broadcast_to_unity` para enviar esta cadena a todos los clientes TCP conectados en el puerto 5000.

0.5. Módulo SkeletonVisualizer.cs en Unity

El script `SkeletonVisualizer.cs` se adjunta a un objeto en la escena de Unity y se encarga de:

- Establecer una conexión TCP con el servidor en el puerto 5000.
- Recibir continuamente las líneas CSV con datos 3D.
- Interpretar las coordenadas y posicionar esferas (joints) y líneas (bones) en el espacio de Unity.

0.5.1. Conexión de red

En el método `Start()`, el script inicia una corutina `AttemptConnection()` que intenta conectarse repetidamente al servidor:

- Crea un objeto `TcpClient` y trata de conectarse a la IP y puerto configurados.
- Si la conexión tiene éxito, obtiene un `NetworkStream` y lanza un hilo `ReceiveData()` dedicado a la recepción.
- Si falla, espera unos segundos y vuelve a intentarlo.

En `ReceiveData()`, el hilo lee bytes del stream, los convierte a texto UTF-8, separa las líneas por el carácter de nueva línea y las encola en `dataQueue` para que el hilo principal de Unity las procese en `Update()`.

0.5.2. Construcción del esqueleto en la escena

El método `InitializeSkeleton()` crea:

- 17 objetos `GameObject` (esferas u otro prefab) que representan las articulaciones (joints). Estos objetos se crean como hijos del objeto que contiene el script y se desactivan inicialmente.
- Un conjunto de 12 objetos `LineRenderer` que representan los huesos (bones), conectando pares de índices definidos en la matriz `boneConnections` (hombros, caderas, brazos, piernas, etc.).

En cada frame, el método `Update()` extrae la última línea disponible de `dataQueue` y llama a `UpdateFrame()`.

0.5.3. Mapeo de datos 3D a objetos Unity

En `UpdateFrame(string csvLine)` se realiza lo siguiente:

1. Se divide la línea CSV usando comas. Las dos primeras columnas son el tiempo y el ID del cuerpo; a partir del índice 2 están las coordenadas de los keypoints.
2. Para cada uno de los 17 puntos, se obtienen las cadenas `sX`, `sY`, `sZ` y se convierten a números en `float` usando `CultureInfo.InvariantCulture`.
3. Si algún punto no tiene datos (cadenas vacías), el joint correspondiente se desactiva.
4. Si los tres valores son válidos, se construye un `Vector3` con las coordenadas $(x, -y, z)$, multiplicado por un factor de escala configurable (`scale`) y desplazado por un `offset`. El signo negativo en el eje y permite ajustar la diferencia de convenciones entre el sistema de coordenadas del mundo 3D definido en Python y el sistema de Unity.
5. Se posiciona cada joint en `transform.localPosition` y se activa el objeto.
6. Finalmente se actualizan los `LineRenderer` de los huesos para que sus extremos coincidan con las posiciones de los joints activos.

De esta forma, el esqueleto se representa visualmente en la escena de Unity y se actualiza en tiempo real a medida que llegan nuevos datos del servidor.

0.6. Flujo de Datos Completo

Resumiendo, el flujo de datos del sistema es el siguiente:

1. **Captura local:** Cada cámara ejecuta `sender.py`, captura frames mediante OpenCV y los envía comprimidos en JPEG al servidor en el puerto 9999.
2. **Recepción y bufferización:** `receiver.py` recibe los frames de cada cámara, mantiene sólo el más reciente por IP y los almacena en `latest_frames`.
3. **Inferencia 2D:** El servidor recorre periódicamente los frames disponibles, ejecuta YOLOv8 pose sobre cada uno y obtiene un conjunto de 17 puntos clave 2D con sus confianzas.
4. **Triangulación 3D:** Para cada punto clave se combinan las observaciones 2D desde al menos dos cámaras, utilizando sus matrices de proyección P , para resolver su posición 3D mediante DLT.
5. **Registro y broadcast:** Las coordenadas 3D se registran en un fichero CSV y se envían en formato CSV por TCP a todos los clientes Unity conectados al puerto 5000.
6. **Visualización en Unity:** El script `SkeletonVisualizer.cs` recibe las líneas CSV, actualiza las posiciones de 17 joints y 12 bones, y muestra en la escena el esqueleto 3D de la persona.

0.7. Consideraciones y Limitaciones

0.7.1. Calibración aproximada de cámaras

En la implementación actual, la configuración de cámaras (posición y orientación) es aproximada y está codificada a mano en **CAMERAS**. Esto permite una reconstrucción 3D razonable para demostraciones, pero no sustituye a una calibración fotogramétrica precisa. Para aplicaciones que requieran alta exactitud métrica, sería recomendable:

- Realizar una calibración intrínseca y extrínseca de cada cámara con patrones tipo tablero de ajedrez.
- Obtener matrices K , R y t reales y sustituir las aproximaciones actuales.

0.7.2. Múltiples personas

El sistema procesa actualmente sólo la primera persona detectada en cada frame (primer esqueleto). La extensión a múltiples cuerpos implicaría identificar correspondencias de IDs entre cámaras y ampliar el formato de datos para incluir varios esqueletos por frame.

0.7.3. Rendimiento

El uso de YOLOv8 para detección de pose es computacionalmente intensivo. El rendimiento dependerá de:

- La potencia de la GPU/CPU del servidor.
- El número de cámaras conectadas simultáneamente.
- La resolución de las imágenes y la frecuencia de captura.

Se ha limitado la tasa de frames en los senders a unos 10 FPS para lograr un compromiso entre fluidez y uso de recursos.

0.8. Conclusión

Se ha desarrollado un sistema completo de captura y visualización 3D de pose humana basado en múltiples cámaras, un servidor central de visión por computador y un cliente de visualización en Unity. A partir de dos o más vistas de una misma persona, el sistema estima su esqueleto 3D y lo muestra en tiempo real, lo que abre la puerta a aplicaciones en realidad aumentada, análisis de movimiento, interacción hombre-máquina y videojuegos.

Aunque la calibración actual es aproximada y existen múltiples posibles mejoras (precisión geométrica, soporte multi-persona, optimización de rendimiento, soporte WebGL, etc.), la arquitectura presentada proporciona una base sólida y extensible para futuros desarrollos.