

Grado en Ingeniería Informática
Asignatura: Visión por Computador

TRABAJO FINAL DE ASIGNATURA

Simon Dice

Autores:

Laura Herrera Negrín
Dunia Suárez Rodríguez
Ayman Asbai Ghoudan

9 de enero de 2026

Índice

1. Introducción	2
1.1. Descripción del proyecto	2
1.2. Objetivos	2
2. Tecnologías y herramientas	3
2.1. Lenguaje y entorno	3
2.2. Librerías principales	3
3. Estructura del repositorio	4
4. Metodología y desarrollo	5
4.1. Fase 1: Recolección del dataset	5
4.2. Fase 2: Entrenamiento (método y cómo)	5
4.2.1. Extracción de características	6
4.2.2. Aumento de datos (data augmentation)	6
4.2.3. Clasificación con MLP	6
4.2.4. Evaluación mediante Validación Cruzada (K-Fold)	6
4.2.5. Análisis de Error: Matriz de Confusión	7
4.3. Fase 3: Desarrollo del juego	8
4.3.1. Arquitectura y Clase Jugador	8
4.3.2. Máquina de estados	9
4.3.3. Niveles de dificultad y trampas	9
4.3.4. Sistema de Audio Personalizado	10
5. Justificación de decisiones de diseño	11
5.1. Estrategia de Validación y Entrenamiento Final	11
5.2. Extracción de características vs. Redes Convolucionales (CNN)	11
5.3. Gestión de Audio: Pre-grabado vs. Síntesis (TTS)	12
6. Desafíos técnicos y soluciones	13
6.1. Variabilidad en la iluminación	13
6.2. Oclusiones parciales	13
6.3. Renderizado de Texto	13
6.4. Latencia en tiempo real	13
7. Manual de usuario	14
7.1. Requisitos previos	14
7.2. Ejecución	14
7.3. Cómo jugar	14
8. Conclusiones	15
9. Bibliografía	16

1. Introducción

1.1. Descripción del proyecto

El presente proyecto, titulado «Simon Dice», consiste en el desarrollo de una versión moderna e interactiva del clásico juego de memoria «Simón Dice». La innovación principal radica en la interfaz de usuario: en lugar de utilizar botones físicos o periféricos convencionales (teclado, ratón o mandos), el juego se controla enteramente mediante **visión artificial (Computer Vision)**.

El sistema es capaz de reconocer gestos faciales y manuales del jugador en tiempo real a través de una webcam, interpretándolos como comandos de juego. Esto permite una experiencia inmersiva donde el jugador debe replicar secuencias de movimientos corporales (gesticular, mover la cabeza, cerrar ojos) para avanzar.

Este enfoque se enmarca dentro de las **interfaces naturales de usuario (NUI)**, que buscan eliminar la barrera física entre la persona y la máquina, permitiendo una interacción más intuitiva y fluida basada en las capacidades motrices humanas naturales.

1.2. Objetivos

Los objetivos principales del proyecto son:

1. **Implementar un sistema de reconocimiento de gestos robusto:** Utilizando técnicas de aprendizaje automático (Machine Learning) para identificar patrones en tiempo real.
2. **Desarrollar una lógica de juego interactiva:** Crear una máquina de estados que gestione la fluidez del juego, desde el menú hasta la detección de movimientos y la respuesta auditiva/visual.
3. **Integración tecnológica eficiente:** Combinar librerías de visión por computador (OpenCV, MediaPipe) con un sistema de audio personalizado para una experiencia multimedia completa.
4. **Entrenamiento de modelos personalizados:** Crear y entrenar clasificadores neuronales (MLP) capaces de distinguir entre diferentes gestos faciales (ojos cerrados, movimientos de cabeza) y manuales (puño, mano abierta, pulgar arriba, etc.).

2. Tecnologías y herramientas

Para el desarrollo de este proyecto se ha utilizado un stack tecnológico basado en Python, aprovechando su extenso ecosistema para ciencia de datos, visión artificial y desarrollo de videojuegos.

2.1. Lenguaje y entorno

- **Python 3.11:** Lenguaje principal del proyecto.
- **Conda:** Gestor de entornos utilizado para aislar las dependencias.
- **Jupyter Notebooks:** Utilizados para el prototipado, generación de datasets y entrenamiento de modelos.

2.2. Librerías principales

- **OpenCV (cv2):** Herramienta fundamental para la captura de vídeo desde la webcam, preprocesamiento de imágenes (flip, conversión de color) y visualización de la interfaz básica.
- **MediaPipe:** Framework desarrollado por Google. Se utilizan los módulos:
 - **FaceMesh:** Para generar una malla facial de 468 puntos y detectar gestos de la cabeza y ojos.
 - **Hands:** Para detectar 21 puntos clave en las manos y reconocer posturas complejas.
- **Scikit-Learn:** Librería de ML utilizada para implementar el **MLPClassifier** (Perceptrón Multicapa), responsable de clasificar los vectores de características en gestos concretos.
- **NumPy:** Esencial para operaciones matemáticas vectoriales, normalización de coordenadas y manipulación de arrays.
- **Pickle:** Para la serialización (guardado) y carga de los modelos entrenados (.pkl).
- **Pygame:** Utilizada para la gestión de efectos de sonido (feedback de acierto/error) en el juego y narración de instrucciones.
- **Pillow (PIL):** Utilizada para renderizar texto UTF-8. Esto es crítico ya que OpenCV utiliza por defecto fuentes tipo **Hershey**, las cuales carecen de soporte para glifos extendidos (tildes, eñes), lo que comprometería la calidad de la interfaz en castellano.

3. Estructura del repositorio

El proyecto se organiza en la siguiente estructura de directorios, separando el código fuente (`src`), los datos (`dataset`) y la documentación:

```
/
|-- dataset/                                # Imágenes para entrenamiento
|   |-- gestos_cara/                        # Dataset de gestos faciales
|   |-- gestos_manos/                      # Dataset de gestos manuales
|
|-- src/                                    # Código fuente del proyecto
|   |-- game/                              # Lógica del juego (Simon Says)
|       |-- fonts/                        # Fuentes para renderizar texto en Open
|       |-- instructions/                 # Audios para la narración de instrucc
|       |-- sounds/                      # Sonidos para feedback de acierto/erro
|       |-- game_multijugador.ipynb      # Jupyter Notebook principal
|   |-- generate_images/                  # Scripts para captura de datos
|   |-- memory/                          # Documentación LaTeX
|   |-- scripts/                         # Utilidades auxiliares
|   |-- train/                           # Notebooks de entrenamiento (MLP)
|
|-- README.md                             # Documentación general y setup
```

4. Metodología y desarrollo

El desarrollo se ha estructurado en tres fases secuenciales: generación del dataset, entrenamiento de los modelos clasificadores y desarrollo de la lógica del juego.

4.1. Fase 1: Recolección del dataset

Para entrenar un modelo robusto, es esencial contar con datos de calidad. Por ello, para construir el dataset, inicialmente se realizó una búsqueda de imágenes en repositorios públicos como Kaggle. Se identificaron conjuntos de datos relevantes, tales como:

- CEW Dataset (Closed Eyes in the Wild), útil para la detección de ojos cerrados.
- HaGRID (HAnd Gesture Recognition Image Dataset), para el reconocimiento de posturas de la mano.

Sin embargo, para adaptar el sistema a las necesidades específicas del juego y mejorar la robustez ante el entorno real de uso, se desarrollaron dos scripts en Python que permiten capturar imágenes de la webcam sistemáticamente para enriquecer el conjunto de datos.

- El sistema captura ráfagas de imágenes mientras el usuario realiza un gesto específico.
- Las imágenes se guardan automáticamente en carpetas etiquetadas con el nombre del gesto (ej. `dataset/gestos_manos/4_Pulgar_Arriba`).
- Se capturaron múltiples variaciones de cada gesto (diferentes distancias, manos izquierda/derecha, ligeras rotaciones) para mejorar la generalización.

Las clases definidas incluyen:

- **Faciales:** Neutro, Ojos Cerrados, Cabeza Derecha, Cabeza Izquierda.
- **Manuales:** Mano Arriba, Puños Cerrados, Pulgar Arriba, Victoria, Rock, Llamada, OK.

4.2. Fase 2: Entrenamiento (método y cómo)

El entrenamiento se llevó a cabo en los notebooks:

- `EntrenoCara.ipynb`
- `EntrenoManoCuerpo.ipynb`

La metodología empleada se basa en el aprendizaje supervisado sobre características geométricas.

4.2.1. Extracción de características

En lugar de utilizar redes neuronales convolucionales (CNN) que procesan la imagen completa (pixel a pixel), se optó por un enfoque basado en **landmarks** (puntos clave).

1. Se procesa cada imagen del dataset con MediaPipe.
2. Se extraen las coordenadas (x, y, z) de los puntos de interés (468 para cara, 21 para mano).
3. **Normalización:** Crucial para que el modelo funcione independientemente de la posición del usuario en la pantalla. Se centra el gesto restando el centroide y se escala dividiendo por la máxima distancia absoluta desde el centro.

4.2.2. Aumento de datos (data augmentation)

Dado que el dataset propio es limitado en tamaño, se aplicaron técnicas de aumento de datos sintético sobre los vectores de características (no sobre las imágenes), lo cual es muy eficiente:

- **Ruido:** Se añade ruido gaussiano aleatorio a las coordenadas para simular el "jitter" de la cámara.
- **Escalado y rotación:** Se aplican transformaciones matriciales 2D para simular que el usuario está más cerca/lejos o inclina la mano.

4.2.3. Clasificación con MLP

Se entrenaron dos modelos `MLPClassifier` (Multi-Layer Perceptron) independientes:

- **Modelo facial:** Arquitectura (128, 64). Capaz de detectar sutilezas como el parpadeo.
- **Modelo manual:** Arquitectura (64, 32). Suficiente para distinguir posturas de dedos.

Los modelos alcanzaron una precisión superior al 95 % en el conjunto de validación y se exportaron como archivos `.pkl` para su uso en producción.

4.2.4. Evaluación mediante Validación Cruzada (K-Fold)

Para garantizar que el modelo no solo memorice los datos (overfitting) sino que sea capaz de generalizar ante nuevos usuarios, no se ha utilizado un único reparto de entrenamiento y prueba. En su lugar, se ha implementado una estrategia de **Validación Cruzada Estratificada (Stratified K-Fold)** con $k = 5$.

Este proceso funciona de la siguiente manera:

1. El dataset total se divide en 5 bloques o "pliegues" (folds) que mantienen la proporción de cada gesto.
2. El sistema realiza 5 iteraciones de entrenamiento. En cada una, utiliza 4 bloques para entrenar y el bloque restante para evaluar la precisión.

3. Se calcula la media de los resultados para obtener una **precisión estimada real**.

Una vez obtenida una precisión satisfactoria en esta fase (generalmente superior al 95 %), se procede al entrenamiento final utilizando el 100 % de los vectores de características. Esto asegura que el archivo `.pkl` resultante contenga el máximo conocimiento posible derivado del dataset.

4.2.5. Análisis de Error: Matriz de Confusión

Para profundizar en el rendimiento del modelo más allá del porcentaje de precisión, se generó una **matriz de confusión**. Esta herramienta es fundamental para identificar qué gestos específicos pueden inducir a error al clasificador.

Al utilizar la técnica de `cross_val_predict`, cada predicción reflejada en la matriz proviene de un subconjunto de datos que el modelo no utilizó para su entrenamiento en esa iteración particular. Esto permite observar:

- **Falsos Positivos:** Casos donde un gesto neutro se confunde con una acción.
- **Confusiones Geométricas:** Por ejemplo, si el modelo confunde Cabeza Derecha con "Neutro" debido a una inclinación insuficiente.
- **Sensibilidad del Parpadeo:** Verificar si el cierre de ojos se distingue correctamente de un gesto facial neutro.

Este análisis permitió ajustar los hiperparámetros del MLP y mejorar el proceso de normalización de los *landmarks* en las clases con mayor tasa de error.

A continuación se presentan las matrices de confusión resultantes del entrenamiento final:

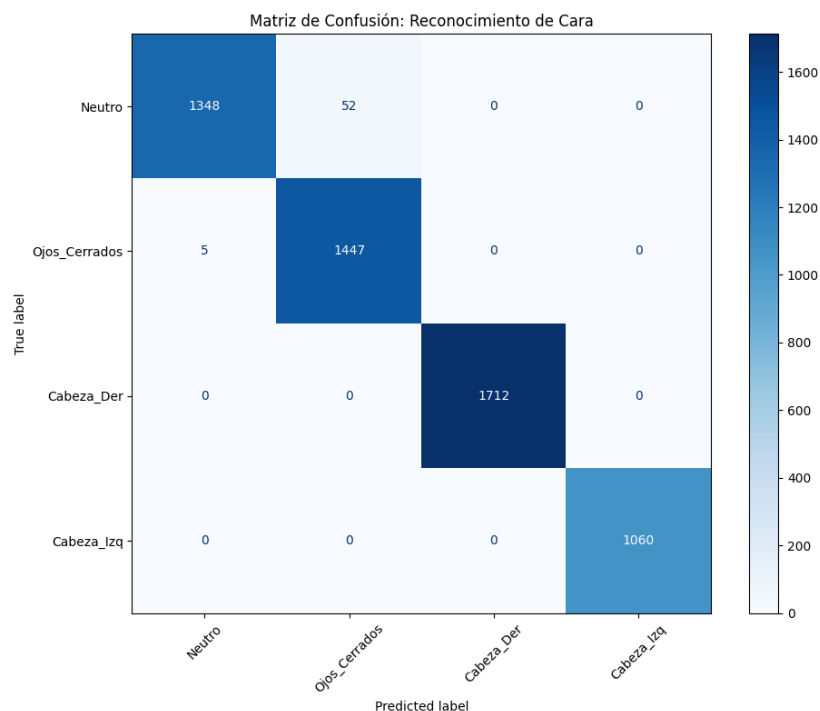


Figura 1: Matriz de confusión del modelo de gestos faciales.

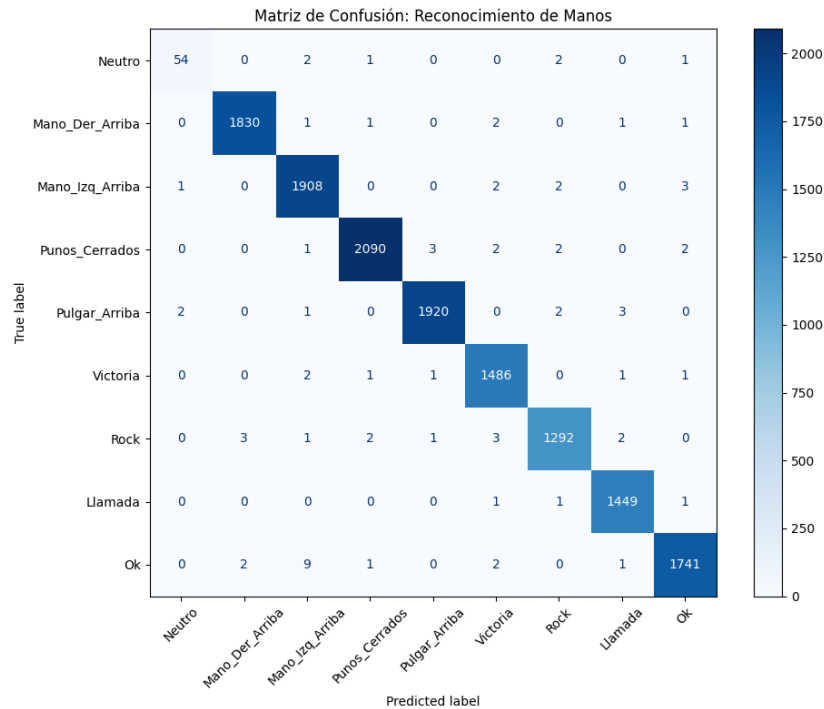


Figura 2: Matriz de confusión del modelo de gestos manuales.

Como se observa en las Figuras 1 y 2, la mayor concentración de predicciones se encuentra en la diagonal principal, lo cual indica un alto nivel de acierto en la clasificación.

Para el **modelo facial**, se obtuvo una precisión estimada mediante validación cruzada del **98.99 %**. Los errores son mínimos, lo que confirma que el modelo distingue correctamente entre ojos abiertos/cerrados y los giros de cabeza, a pesar de la sutileza de estos movimientos.

Por su parte, el **modelo manual** alcanzó una precisión del **99.47 %**. Esto demuestra que la normalización de los puntos de la mano es efectiva para distinguir entre gestos complejos como «Victoria», «Rock» o «Ok», incluso con variaciones en la ejecución por parte del usuario.

La baja tasa de falsos positivos y negativos valida la estrategia de utilizar *Data Augmentation* sintético, permitiendo que el modelo generalice correctamente sin necesidad de un dataset masivo de imágenes reales.

4.3. Fase 3: Desarrollo del juego

La lógica principal del juego reside en el archivo principal (`src/game/game_multijugador.ipynb`). El juego está implementado utilizando programación orientada a objetos para permitir la escalabilidad hacia múltiples jugadores.

4.3.1. Arquitectura y Clase Jugador

Se ha introducido la clase **Player** para gestionar el estado individual de cada participante. Esta clase encapsula:

- **Identificación:** ID, nombre y límites de pantalla asignados (bounds) para dividir el área de detección en el modo dos jugadores.

- **Estado del juego:** Vidas restantes, estado de eliminación y progreso en la ronda actual.
- **Detección:** Gesto actual detectado, contadores de estabilidad (frames consecutivos) y tiempos de reacción.

La clase **Player** encapsula el estado del participante. Un aspecto técnico crítico es el **contador de estabilidad**. Para evitar falsos positivos debidos al ruido de la cámara o detecciones efímeras, el sistema solo valida un gesto si este se mantiene idéntico durante un umbral de fotogramas consecutivos (habitualmente $N = 5$).

Esta estructura permite que el juego gestione dos versiones:

1. **Un Jugador:** El usuario ocupa toda la pantalla. La lógica se centra en superar la secuencia creciente.
2. **Dos Jugadores (VS):** La pantalla se divide en dos secciones. Ambos jugadores compiten simultáneamente para completar la misma instrucción. El sistema penaliza individualmente los fallos o la falta de rapidez.

4.3.2. Máquina de estados

Para gestionar el flujo del juego, se implementó una máquina de estados finitos mejorada:

1. **MENU:** Pantalla de selección de modo (1 o 2 jugadores).
2. **CHECK_PLAYERS:** (Solo multijugador) Verifica que ambos jugadores estén en posición y listos antes de comenzar.
3. **SHOW_NEW_STEP:** El turno de la CPU. El sistema:
 - Añade un nuevo paso a la secuencia.
 - Reproduce el archivo de audio correspondiente a la instrucción.
 - Muestra texto en pantalla usando **Pillow**.
4. **WAIT_NEUTRAL:** Estado de sincronización que obliga a todos los jugadores activos a volver a una posición neutra antes de la siguiente acción, evitando falsos positivos.
5. **PLAYER_TURN:** Fase de ejecución. El sistema valida en tiempo real si el gesto detectado coincide con el esperado para cada jugador activo.
6. **GAME_OVER / SUCCESS_SEQUENCE:** Estados finales de ronda o partida.

4.3.3. Niveles de dificultad y trampas

Una característica clave es la lógica de "trampas", inspirada en el juego real:

- **Nivel básico:** El juego siempre dice "Simón dice...".
- **Nivel intermedio:** Aparecen comandos como "Modesto dice..." El jugador **no** debe realizar el gesto. Si se mueve, pierde.

- **Nivel avanzado:** Se omiten palabras clave ("Simón...", "Dice...").

Esta lógica se gestiona mediante la función `add_sequence_step`, que genera aleatoriamente instrucciones válidas o trampas según la puntuación actual.

4.3.4. Sistema de Audio Personalizado

Inicialmente se contempló el uso de librerías de síntesis de voz (TTS) para narrar las instrucciones. Sin embargo, se detectaron problemas de bloqueo (freezing) al ejecutar el motor TTS en un hilo paralelo al procesamiento de vídeo, lo que afectaba a la fluidez del juego.

Como solución, se implementó un sistema de audio basado en clips pre-grabados:

- Se generaron archivos de audio (.mp3) individuales para cada instrucción posible (ej. "simon_1_Mano_Der_Arriba.mp3").
- Se utiliza `pygame.mixer` para la reproducción asíncrona de estos sonidos, garantizando que el bucle de vídeo no se detenga.
- Este enfoque permite además personalizar la voz del narrador, añadiendo variedad con voces para "Simón", "Modesto." instrucciones genéricas.

5. Justificación de decisiones de diseño

En el desarrollo de sistemas de visión por computador e interacción en tiempo real, la elección de las herramientas y arquitecturas determina la viabilidad y usabilidad del proyecto. A continuación se detallan y justifican las decisiones técnicas más relevantes adoptadas durante el desarrollo.

5.1. Estrategia de Validación y Entrenamiento Final

A diferencia de los flujos de trabajo convencionales de Machine Learning que dividen el dataset en tres conjuntos estáticos (entrenamiento, validación y prueba), en este proyecto se ha optado por una estrategia basada en **Validación Cruzada Estratificada (Stratified K-Fold Cross-Validation)** seguida de un re-entrenamiento total.

Esta decisión se fundamenta en los siguientes puntos:

- **Optimización del Dataset:** Al trabajar con un dataset generado específicamente para este juego, la validación cruzada permite utilizar cada muestra tanto para entrenar como para evaluar en diferentes iteraciones. Esto proporciona una estimación de la precisión mucho más robusta y menos sesgada que un simple *split* de 80/20, especialmente valioso cuando el volumen de datos es moderado.
- **Garantía de Generalización:** El uso de `StratifiedKFold` asegura que cada "pliegue" (fold) mantenga la proporción de clases original, evitando que el modelo se sesgue hacia los gestos con más muestras durante la evaluación de su rendimiento.
- **Modelo de Producción de Máximo Aprendizaje:** Una vez validada la arquitectura del MLP y confirmada su alta precisión (superior al 95 %), se procedió a entrenar el modelo final utilizando el 100 % de los datos disponibles. En entornos de producción, esta práctica garantiza que el clasificador haya "visto" todas las variaciones posibles de los gestos, maximizando su capacidad de respuesta ante el usuario final.

Nota sobre el Data Augmentation: Se es consciente de que la aumentación de datos se realiza de forma previa a la validación. Aunque esto puede optimizar los resultados de la métrica, la robustez final se comprueba mediante las pruebas unitarias en tiempo real descritas en la fase de desarrollo, donde el modelo demuestra su capacidad de generalización ante nuevos usuarios.

5.2. Extracción de características vs. Redes Convolucionales (CNN)

Una decisión fundamental fue optar por una arquitectura de dos etapas: extracción de *landmarks* (puntos clave) con MediaPipe seguida de clasificación con un Perceptrón Multicapa (MLP), en lugar de utilizar una Red Neuronal Convolutiva (CNN) *end-to-end* que procese la imagen completa.

- **Eficiencia Computacional (Latencia):** Una imagen estándar de webcam (1280×720) contiene casi un millón de píxeles. Procesar esta matriz con una CNN profunda en cada frame requiere una potencia de cálculo considerable (GPU), lo que limitaría el juego a ordenadores potentes. Al utilizar MediaPipe, delegamos la detección espacial a un modelo altamente optimizado para CPU, y nuestro clasificador MLP solo necesita procesar un vector de entrada muy pequeño (42 valores para manos, 936 para cara). Esto garantiza una tasa de fotogramas estable (> 30 FPS) incluso en equipos portátiles estándar.
- **Robustez y Generalización:** Las CNNs entrenadas con pocas imágenes tienden a aprender patrones irrelevantes, como el color de fondo o la ropa del usuario (overfitting). Al trabajar exclusivamente con coordenadas geométricas normalizadas, nuestro modelo se vuelve “ciego” al entorno. Esto significa que el sistema funciona igual de bien si el usuario está en una habitación oscura, luminosa o con un fondo complejo, ya que solo “ve” la posición relativa de los puntos de la mano y la cara.

5.3. Gestión de Audio: Pre-grabado vs. Síntesis (TTS)

La experiencia de usuario requiere instrucciones auditivas claras. Inicialmente se implementó síntesis de voz en tiempo real (*Text-to-Speech*), pero fue descartada en favor de clips de audio pre-grabados.

- **Bloqueo del Hilo Principal (Thread Blocking):** Las librerías de TTS en Python, como `pyttsx3`, suelen operar de forma síncrona o tienen una gestión de hilos compleja que entra en conflicto con el bucle principal de OpenCV (`while True`). Esto provocaba micro-congelamientos (“stuttering”) en el vídeo cada vez que el juego hablaba.
- **Consistencia Estética:** Los motores TTS dependen de las voces instaladas en el sistema operativo del usuario, lo que hace que el juego suene diferente en cada ordenador (a veces con voces robóticas de baja calidad). El uso de archivos `.mp3` garantiza que todos los jugadores escuchen la misma voz, con la entonación y emoción correctas para el contexto del juego.

6. Desafíos técnicos y soluciones

Durante el desarrollo del proyecto surgieron varios retos inherentes a la aplicación de visión por computador en entornos no controlados:

6.1. Variabilidad en la iluminación

Los algoritmos de detección visual son sensibles a las condiciones de luz extremas (contraluz o oscuridad).

Solución: La normalización de las coordenadas de los landmarks hace que el modelo dependa de la geometría relativa de los puntos y no de los valores de intensidad de los píxeles, mitigando parcialmente este problema.

6.2. Oclusiones parciales

En ocasiones, al realizar gestos cerca de la cara, las manos pueden ocultar partes del rostro, confundiendo al detector facial.

Solución: Se implementó una lógica de priorización en el bucle principal del juego y se ajustaron los umbrales de confianza (`min_detection_confidence`) para reducir falsos positivos.

6.3. Renderizado de Texto

OpenCV no soporta caracteres UTF-8 nativamente, lo que impedía escribir “Simón” o “Puño” correctamente.

Solución: Se creó la función auxiliar `poner_texto_utf8` que convierte el frame a formato PIL Image, dibuja el texto con una fuente TrueType (.otf) personalizada y lo reconvierte a OpenCV.

6.4. Latencia en tiempo real

El procesamiento de vídeo requiere un alto rendimiento para no afectar a la jugabilidad.

Solución: El uso de clasificadores MLP (redes neuronales simples) sobre vectores de características ligeros, en lugar de redes convolucionales profundas (CNN) sobre imágenes completas, garantiza una tasa de fotogramas (FPS) alta incluso en equipos con hardware modesto.

7. Manual de usuario

7.1. Requisitos previos

Es necesario disponer de una webcam funcional y un entorno con las librerías instaladas:

```
1 pip install numpy==2.2.6 matplotlib==3.10.7 opencv-contrib-python  
   ==4.12.0.88 mediapipe==0.10.14 scikit-learn==1.7.2 pygame==2.6.1  
   sounddevice==0.5.3 pillow jax jaxlib ipykernel
```

7.2. Ejecución

1. Abrir la terminal en la carpeta `src/game/`.
2. Ejecutar el archivo principal del juego en Jupyter o Python.
3. Seleccionar el modo de juego: [1] Un Jugador o [2] Dos Jugadores.
4. Seguir las instrucciones en pantalla y audio.

7.3. Cómo jugar

1. **Atención:** El juego te hablará y mostrará texto.
2. **Ordenes Válidas:** Si dice “Simón dice: [Acción]”, debes imitar el gesto.
3. **Trampas:**
 - Si dice “Modesto dice...” o no menciona a “Simón”, ¡No te muevas!
 - Mantén la posición neutra hasta que pase la ronda.
4. **Secuencia:** Debes memorizar y repetir todos los pasos acumulados.

8. Conclusiones

Este proyecto demuestra la viabilidad de crear interfaces hombre-máquina naturales y accesibles utilizando hardware de consumo estándar (webcam). La combinación de la extracción de características geométricas (mediante MediaPipe) con clasificadores ligeros (MLP) permite una inferencia en tiempo real extremadamente rápida, adecuada para videojuegos.

La integración de feedback multimedia (voz y sonido) enriquece significativamente la experiencia de usuario, haciendo el juego más dinámico y accesible. El uso de técnicas como la normalización de coordenadas y el aumento de datos ha sido crucial para obtener un sistema robusto.

9. Bibliografía

- Lugares, C., & Zhang, F. (2019). *MediaPipe: A Framework for Building Perception Pipelines*. Google Research.
- Kartynnik, Y., et al. (2019). *Real-time Facial Surface Geometry from Monocular Video on Mobile Devices*. arXiv preprint arXiv:1907.06744.
- Scikit-Learn Documentation: <https://scikit-learn.org/stable/>
- OpenCV Documentation: <https://docs.opencv.org/>