

Universidad de Las Palmas de Gran Canaria
Escuela de Ingeniería Informática

VISION WALL

Sistema Interactivo de Evaluación de Posturas Corporales
en Tiempo Real mediante Visión por Computador

Flujo del Sistema:

Cámara → MediaPipe → WebSocket → Godot 4.5

Proyecto Final

Asignatura: Visión por Computador (40982)
Grado en Ingeniería Informática

Autores:

Nicolás Rey Alonso
Wafa Azdad Triki

Tecnologías Principales:

Godot 4.5 Python 3.11 MediaPipe

Curso 2025-2026
9 de enero de 2026

Índice

Resumen	5
1. Introducción	6
1.1. Motivación	6
1.2. Objetivo General	6
1.3. Objetivos Específicos	6
1.4. Alcance del Proyecto	7
2. Tecnologías Utilizadas	7
2.1. Motor Gráfico: Godot 4.5	7
2.2. Detección de Pose: MediaPipe	8
2.3. Comunicación: WebSocket	9
2.4. Modelado 3D: Blender	9
2.5. Audio: LMMS	9
2.6. Lenguajes de Programación	9
2.6.1. Python 3.11	9
2.6.2. GDScript	10
2.7. Entorno de Desarrollo	10
3. Arquitectura del Sistema	10
3.1. Diagrama de Arquitectura General	10
3.2. Estructura del Proyecto	11
3.3. Flujo de Datos	11
4. Componentes Principales	12
4.1. Servidor MediaPipe (mediapipe_util.py)	12
4.1.1. Configuración de MediaPipe	12
4.1.2. Servidor WebSocket Asíncrono	13
4.2. Controlador Principal (mainscript.gd)	13
4.2.1. Variables de Configuración	13
4.2.2. Mapeo de Huesos	14
4.2.3. Algoritmo de Actualización de Pose	14
4.3. Generador de Paredes (WallGenerator.gd)	14
4.3.1. Variables de Exportación	15
4.3.2. Posturas Disponibles	15
4.3.3. Generación de Mesh con SurfaceTool	15
4.4. Evaluador de Posturas (PoseEvaluator.gd)	16
4.4.1. Algoritmo de Evaluación	16
4.5. Selección de Personajes (character_select.gd)	16
4.5.1. Flujo de Selección	16
5. Sistemas de Juego	17
5.1. Sistema de Puntuación y Progresión	17
5.1.1. Progresión de Dificultad (Versión Pygame)	17
5.2. Sistema de Colisiones	17
5.2.1. Configuración de Áreas de Colisión	17
5.2.2. Diagrama de Colisiones	18

5.3. Animación de Personajes en Tiempo Real	18
5.3.1. Proceso de Animación	18
5.4. Generación Dinámica de Paredes	19
6. Detalles Técnicos de Implementación	20
6.1. Protocolo de Comunicación WebSocket	20
6.1.1. Estructura del Mensaje JSON	20
6.1.2. Interpretación de Coordenadas	20
6.2. Transformación de Coordenadas	21
6.3. Material de Paredes	21
6.4. Gestión de Memoria y Rendimiento	21
6.4.1. Limpieza de Paredes	21
6.4.2. Pool de Conexiones WebSocket	21
7. Versión Standalone (Pygame)	22
7.1. Características de la Versión Pygame	22
7.2. Detección de Posición de Brazos	22
7.3. Sistema de Coincidencia de Pose	23
8. Flujo de Ejecución	23
8.1. Inicio del Juego	23
8.2. Loop Principal (_process)	24
8.3. Validación de Paso por Pared	25
9. Configuración y Parámetros Ajustables	25
9.1. Parámetros del Servidor MediaPipe	25
9.2. Parámetros del Juego (Godot)	26
9.3. Posiciones en la Escena 3D	26
10. Personajes del Juego	26
10.1. Proceso de Creación de Personajes	26
10.2. SAW	27
10.3. ET	28
10.4. ELEVEN	29
10.5. HOMER	29
10.6. Sistema de Rigging con Rigify	30
10.7. Estructura del Esqueleto Exportado	30
11. Sistema de Audio	30
11.1. Música de Fondo	31
11.2. Efectos de Sonido	31
11.3. Integración en Godot	31
11.4. Flujo de Creación de Audio	31
12. Resultados y Pruebas	32
12.1. Métricas de Rendimiento	32
12.2. Precisión de Detección de Pose	32
12.3. Usabilidad	32

13. Guía de Instalación y Uso	33
13.1. Requisitos del Sistema	33
13.2. Instalación de Dependencias Python	33
13.3. Ejecución del Sistema	33
13.3.1. Versión Completa (Godot + MediaPipe)	33
13.3.2. Versión Standalone (Pygame)	34
13.4. Controles del Juego	34
14. Galería del Juego	34
14.1. Escenas Principales	34
14.2. Elementos Visuales	34
15. Limitaciones y Trabajo Futuro	34
15.1. Limitaciones Actuales	34
15.2. Mejoras Futuras Propuestas	35
15.3. Posibles Optimizaciones	35
16. Conclusiones	36
16.1. Objetivos Cumplidos	36
16.2. Conocimientos Aplicados	36
16.3. Reflexiones Finales	36
Referencias	37
Anexo A: Índices MediaPipe	38
Anexo B: Estructura de Archivos	38

Resumen

Este documento presenta el desarrollo de **Vision Wall**, un sistema interactivo de evaluación de posturas corporales en tiempo real inspirado en el programa de televisión japonés “Hole in the Wall”. El proyecto integra técnicas avanzadas de visión por computador mediante MediaPipe para la detección de 33 puntos de referencia del cuerpo humano, comunicación en tiempo real a través de WebSocket, y renderizado 3D con el motor de juegos Godot 4.5.

El sistema permite a dos jugadores simultáneos controlar avatares 3D personalizados mediante sus movimientos corporales, debiendo adoptar posturas específicas para atravesar paredes con siluetas humanoides. La arquitectura cliente-servidor facilita una comunicación fluida entre el módulo de procesamiento de imagen (Python/MediaPipe) y el motor gráfico (Godot/GDScript), logrando tasas de actualización de aproximadamente 30 FPS.

Palabras clave: Visión por Computador, MediaPipe, Detección de Pose, Godot Engine, WebSocket, Tiempo Real, Interacción Humano-Computador.

1. Introducción

Este proyecto implementa un sistema interactivo de evaluación de posturas corporales en tiempo real inspirado en el famoso programa de televisión japonés “Hole in the Wall” (conocido en Japón como “Nōkabe”). El sistema utiliza detección de pose mediante MediaPipe y un motor gráfico 3D (Godot) para crear una experiencia de juego inmersiva donde los jugadores deben adoptar posturas específicas para atravesar paredes con agujeros en forma de figuras humanas.

1.1. Motivación

La motivación principal de este proyecto surge de la necesidad de explorar aplicaciones prácticas de la visión por computador en entornos interactivos y lúdicos. La detección de pose humana ha experimentado avances significativos en los últimos años, permitiendo aplicaciones que antes requerían hardware especializado costoso. Con frameworks como MediaPipe, ahora es posible implementar sistemas de seguimiento corporal precisos utilizando únicamente una cámara web convencional.

1.2. Objetivo General

Desarrollar un sistema completo que evalúe posturas corporales en tiempo real, permitiendo a los jugadores interactuar con un entorno 3D mediante sus movimientos físicos. El sistema debe:

- Detectar la postura del jugador en tiempo real con alta precisión
- Soportar múltiples jugadores simultáneamente (hasta 2)
- Generar paredes con agujeros correspondientes a 7 posturas diferentes
- Validar si la postura del jugador coincide con el agujero de la pared
- Proporcionar retroalimentación visual e interactiva inmediata
- Mantener un sistema de puntuación, vidas y progresión
- Ofrecer una experiencia de usuario fluida con baja latencia

1.3. Objetivos Específicos

1. **Procesamiento de Imagen:** Implementar un pipeline de captura y procesamiento de video que extraiga los puntos de referencia corporales en tiempo real.
2. **Comunicación Cliente-Servidor:** Diseñar un protocolo de comunicación eficiente basado en WebSocket para transmitir datos de pose entre Python y Godot.
3. **Animación de Esqueleto:** Desarrollar un sistema de animación que traduzca los datos de pose capturados a movimientos del esqueleto 3D de los personajes.
4. **Sistema de Colisiones:** Implementar un mecanismo de validación de poses basado en el motor de física de Godot.
5. **Experiencia de Usuario:** Crear una interfaz intuitiva con retroalimentación audiovisual que mejore la experiencia de juego.

1.4. Alcance del Proyecto

El proyecto abarca desde la captura de video mediante cámara web hasta la renderización final en el motor de juegos, incluyendo:

- Módulo de captura y procesamiento de video (Python + OpenCV)
- Detección de pose con MediaPipe Pose Landmarker
- Servidor WebSocket para comunicación en tiempo real
- Motor de juego 3D con Godot 4.5
- Sistema de selección de personajes
- Generación procedural de paredes con diferentes siluetas
- Sistema de puntuación y gestión de partida
- Efectos de audio personalizados

2. Tecnologías Utilizadas

En esta sección se describen las tecnologías empleadas para el desarrollo del proyecto, justificando la elección de cada una de ellas.

2.1. Motor Gráfico: Godot 4.5

Godot es un motor de juegos de código abierto multiplataforma que se ha convertido en una alternativa popular a motores comerciales como Unity o Unreal Engine. Se eligió Godot por las siguientes razones:

- **Código Abierto:** Licencia MIT que permite uso comercial sin restricciones
- **Soporte 3D Nativo:** Sistema completo de Node3D y MeshInstance3D para gráficos 3D
- **Sistema de Materiales:** Shaders y materiales PBR (Physically Based Rendering)
- **Arquitectura de Escenas:** Gestión jerárquica de nodos que facilita la organización del código
- **GDScript:** Lenguaje de scripting nativo inspirado en Python, fácil de aprender
- **Sistema de Colisiones 3D:** Motor de física integrado con detección de colisiones precisa
- **Soporte para Skeleton3D:** Animación de esqueletos compatible con formatos estándar (.glb, .gltf)
- **WebSocket Nativo:** Soporte incorporado para comunicación WebSocket

Cuadro 1: Comparativa de motores de juego considerados

Característica	Godot 4.5	Unity	Unreal
Licencia	MIT (Libre)	Propietaria	Propietaria
Tamaño del ejecutable	40 MB	200 MB	500 MB
Curva de aprendizaje	Baja	Media	Alta
Soporte WebSocket	Nativo	Plugin	Plugin
Exportación Linux	Nativa	Limitada	Limitada

2.2. Detección de Pose: MediaPipe

MediaPipe es un framework de Google para la construcción de pipelines de procesamiento de medios, especialmente diseñado para visión por computador. El módulo **Pose Landmarker** proporciona:

- **33 Puntos de Referencia:** Detección completa del cuerpo incluyendo cara, manos y pies
- **Coordenadas 3D:** Cada punto incluye posición (x, y, z) y visibilidad
- **Tiempo Real:** Procesamiento a más de 30 FPS en hardware convencional
- **Multi-persona:** Capacidad de detectar hasta 2 personas simultáneamente
- **Robustez:** Funciona en condiciones de iluminación variables

Índice	Articulación	Conexiones
0	Nariz	Centro de referencia
11	Hombro izquierdo	12, 13, 23
12	Hombro derecho	11, 14, 24
13	Codo izquierdo	11, 15
14	Codo derecho	12, 16
15	Muñeca izquierda	13
16	Muñeca derecha	14
23	Cadera izquierda	11, 24, 25
24	Cadera derecha	12, 23, 26
25	Rodilla izquierda	23, 27
26	Rodilla derecha	24, 28
27	Tobillo izquierdo	25
28	Tobillo derecho	26

Figura 1: Puntos de referencia principales de MediaPipe Pose (13 de 33 puntos utilizados)

Los índices de los puntos clave utilizados en el proyecto son:

$$\text{KEYPOINTS} = [0, 11, 12, 13, 14, 15, 16, 23, 24, 25, 26, 27, 28] \quad (1)$$

2.3. Comunicación: WebSocket

WebSocket es un protocolo de comunicación bidireccional full-duplex sobre una única conexión TCP. Se eligió sobre otras alternativas por:

- **Baja Latencia:** Conexión persistente sin overhead de HTTP
- **Bidireccionalidad:** Comunicación en ambas direcciones
- **Soporte Universal:** Implementaciones disponibles en Python y Godot
- **Formato JSON:** Facilidad para serializar datos de pose

2.4. Modelado 3D: Blender

Se utilizó Blender 4.x para la creación de todos los assets 3D del proyecto:

- Modelado de 4 personajes inspirados en cultura popular (Saw, ET, Eleven, Homer)
- Sistema de rigging con el addon **Rigify** para esqueletos compatibles
- Skinning y weight painting para deformación natural
- Exportación en formato **.glb** (GL Transmission Format Binary)
- Modelado del escenario (Coliseo) y paredes con diferentes siluetas

2.5. Audio: LMMS

LMMS (Linux MultiMedia Studio) es un DAW (Digital Audio Workstation) de código abierto utilizado para:

- Composición del tema musical principal (turiruriru-01)
- Creación de efectos de sonido para interacción (clicks, hovers)
- Exportación en formatos .ogg y .mp3 compatibles con Godot

2.6. Lenguajes de Programación

2.6.1. Python 3.11

Utilizado para el backend de procesamiento de imagen:

```

1 # Bibliotecas utilizadas
2 import cv2                # OpenCV para captura de video
3 import mediapipe          # Deteccion de pose
4 import websockets         # Servidor WebSocket
5 import asyncio            # Programacion asincrona
6 import json               # Serializacion de datos
7 import numpy as np        # Operaciones numericas

```

Listing 1: Dependencias principales de Python

2.6.2. GDScript

Lenguaje nativo de Godot, similar a Python, utilizado para toda la lógica del juego:

```

1 extends Node3D
2
3 @export var wall_speed: float = 5.0
4 @export var tolerance: float = 0.3
5
6 var socket := WebSocketPeer.new()
7 var players: Array = []
8
9 func _ready():
10     var err = socket.connect_to_url("ws://localhost:8765")
11     if err != OK:
12         push_error("Error de conexion WebSocket")

```

Listing 2: Ejemplo de declaración en GDScript

2.7. Entorno de Desarrollo

- **Sistema Operativo:** Linux (Ubuntu/derivados)
- **IDE:** Visual Studio Code con extensiones para GDScript y Python
- **Control de Versiones:** Git
- **Gestión de Paquetes:** Anaconda/Miniconda para entornos virtuales Python

3. Arquitectura del Sistema

El sistema sigue una arquitectura cliente-servidor desacoplada que separa el procesamiento de visión por computador de la renderización 3D, permitiendo flexibilidad y escalabilidad.

3.1. Diagrama de Arquitectura General

El sistema se organiza en las siguientes capas:

1. **Capa de Entrada:** Cámara Web (640x480)
2. **Capa de Procesamiento:** OpenCV (Captura) → MediaPipe (Pose Landmarker)
3. **Capa de Comunicación:** WebSocket Server (:8765) transmite datos JSON
4. **Capa de Aplicación:** Godot Engine actúa como cliente WebSocket
5. **Submódulos de Godot:**
 - Skeleton3D (Animación)
 - Physics (Colisiones)
 - Renderizado 3D + HUD

3.2. Estructura del Proyecto

```

1 Entrega_FINAL_VC/
2 |-- Entrega.ipynb           # Version standalone (Pygame)
3 |-- mediapipe_util.py       # Servidor WebSocket + MediaPipe
4 |-- pose_landmarker_full.task # Modelo de MediaPipe
5 |-- BrainWallGodot/
6 |   +-- brain-wall/
7 |     |-- project.godot      # Configuración del proyecto
8 |     |-- PoseReceiver.gd    # Receptor WebSocket simple
9 |     |-- main_menu.gd       # Controlador menu principal
10 |     |-- scenes/
11 |       |-- mainScene.tscn   # Escena principal del juego
12 |       |-- main_menu.tscn   # Menu principal
13 |       |-- CharacterSelect.tscn
14 |       |-- WallGenerator.gd # Generador de paredes
15 |       |-- PoseEvaluator.gd # Evaluador de posturas
16 |       +-- game_ui.tscn      # Interfaz de usuario
17 |     |-- Scripts/
18 |       |-- mainscript.gd     # Controlador principal
19 |       |-- character_select.gd
20 |       |-- game_manager.gd   # Gestor de partida
21 |       |-- game_state.gd     # Estado global
22 |       +-- wall.gd           # Comportamiento de paredes
23 |   +-- assets/
24 |     |-- Characters/         # Escenas de personajes
25 |     |-- models/             # Modelos 3D (.glb)
26 |     |-- music/              # Música de fondo
27 |     |-- SoundEffects/       # Efectos de sonido
28 |     +-- Walls/              # Meshes de paredes
29 |-- Modelados/                # Archivos fuente de Blender
30 +-- INFORME/
31 +-- MEMORIA.tex               # Este documento

```

3.3. Flujo de Datos

El flujo de datos del sistema se puede describir en las siguientes etapas:

1. **Captura de Video:** La cámara web captura frames a 640x480 píxeles a 30 FPS.
2. **Preprocesamiento:** OpenCV convierte el frame de BGR a RGB y aplica flip horizontal (efecto espejo).
3. **Detección de Pose:** MediaPipe Pose Landmarker procesa el frame y detecta hasta 2 personas, extrayendo 33 puntos de referencia por persona.
4. **Filtrado de Puntos:** Se seleccionan los 13 puntos clave relevantes para el seguimiento corporal (índices 0, 11-16, 23-28).
5. **Serialización:** Los datos de pose se serializan en formato JSON con estructura:

```

1 {
2   "poses": [
3     [{"x": 0.5, "y": 0.3, "z": -0.1}, ...], // Jugador 1
4     [{"x": 0.6, "y": 0.35, "z": -0.08}, ...] // Jugador 2
5   ]
6 }

```

6. **Transmisión WebSocket:** El servidor envía los datos a todos los clientes conectados.
7. **Recepción en Godot:** El cliente WebSocket recibe y parsea los datos JSON.
8. **Actualización de Esqueleto:** Los datos de pose se traducen a rotaciones de huesos del Skeleton3D.
9. **Detección de Colisiones:** El motor de física verifica colisiones entre jugadores y paredes.
10. **Actualización de UI:** Se actualiza el HUD con puntuación, vidas y postura actual.

Pipeline de procesamiento:

Captura → RGB+Flip → MediaPipe → JSON → WebSocket → Godot → Skeleton → Render

4. Componentes Principales

Esta sección describe en detalle los componentes de software que conforman el sistema.

4.1. Servidor MediaPipe (mediapipe_util.py)

El servidor es el componente encargado de capturar video, detectar poses y transmitir los datos a los clientes.

4.1.1. Configuración de MediaPipe

```

1 BaseOptions = mp.tasks.BaseOptions
2 PoseLandmarker = mp.tasks.vision.PoseLandmarker
3 PoseLandmarkerOptions = mp.tasks.vision.PoseLandmarkerOptions
4 VisionRunningMode = mp.tasks.vision.RunningMode
5
6 options = PoseLandmarkerOptions(
7     base_options=BaseOptions(
8         model_asset_path="pose_landmarker_full.task"
9     ),
10    num_poses=2, # Detectar hasta 2 personas
11    running_mode=VisionRunningMode.VIDEO
12 )
13
14 detector = PoseLandmarker.create_from_options(options)

```

Listing 3: Configuración del detector de pose

4.1.2. Servidor WebSocket Asíncrono

El servidor utiliza programación asíncrona para manejar múltiples clientes:

```

1  async def capture_and_broadcast():
2      cap = cv2.VideoCapture(0)
3      time_ms = 0
4
5      while True:
6          ret, frame = cap.read()
7          rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
8          mp_image = mp.Image(
9              image_format=mp.ImageFormat.SRGB,
10             data=rgb
11         )
12
13         result = detector.detect_for_video(mp_image, time_ms)
14         time_ms += 33  # ~30 FPS
15
16         poses_data = []
17         for landmarks in result.pose_landmarks[:2]:
18             pose_points = [
19                 {"x": landmarks[i].x,
20                  "y": landmarks[i].y,
21                  "z": landmarks[i].z}
22                 for i in KEYPPOINTS
23             ]
24             poses_data.append(pose_points)
25
26         # Broadcast a todos los clientes
27         message = json.dumps({"poses": poses_data})
28         websockets.broadcast(connected_clients, message)

```

Listing 4: Broadcast de datos de pose

4.2. Controlador Principal (mainscript.gd)

El script principal de Godot maneja la lógica del juego, la conexión WebSocket y la actualización de personajes.

4.2.1. Variables de Configuración

```

1  @export var target_height: float = 2.0
2  @export var scale_factor: float = 0.7
3  @export var depth_scale: float = 2.0
4  @export var mirror_mode: bool = true
5  @export var movement_smoothing: float = 0.2
6  @export var rotation_smoothing: float = 0.08
7  @export var lock_z_movement: bool = true
8  @export var player1_offset: Vector3 = Vector3(-1.0, 0.0, 0.0)
9  @export var player2_offset: Vector3 = Vector3(1.0, 0.0, 0.0)

```

Listing 5: Variables exportables del controlador

4.2.2. Mapeo de Huesos

La traducción de datos de MediaPipe a rotaciones de esqueleto requiere un mapeo de índices a nombres de huesos:

```

1  const LIMB_INDICES = {
2      "upper_arm_L": [1, 3],      # hombro izq -> codo izq
3      "forearm_L": [3, 5],       # codo izq -> muñeca izq
4      "upper_arm_R": [2, 4],     # hombro der -> codo der
5      "forearm_R": [4, 6],       # codo der -> muñeca der
6      "thigh_L": [7, 9],         # cadera izq -> rodilla izq
7      "shin_L": [9, 11],         # rodilla izq -> tobillo izq
8      "thigh_R": [8, 10],        # cadera der -> rodilla der
9      "shin_R": [10, 12],        # rodilla der -> tobillo der
10 }

```

Listing 6: Mapeo de extremidades a índices de MediaPipe

4.2.3. Algoritmo de Actualización de Pose

El proceso de actualización de pose sigue estos pasos:

Algorithm 1 Actualización de pose del jugador

```

1: data ← ParseJSON(websocket_packet)
2: for cada player_idx en players_data do
3:     skeleton ← players_skeletons[player_idx]
4:     pose_points ← data.poses[player_idx]
5:     for cada limb en UPDATE_ORDER do
6:         (start, end) ← LIMB_INDICES[limb]
7:         dir ← pose_points[end] - pose_points[start]
8:         dir ← normalize(dir)
9:         rotation ← look_at_rotation(dir)
10:        bone_idx ← skeleton.find_bone(BONE_NAMES[limb])
11:        current ← skeleton.get_bone_pose_rotation(bone_idx)
12:        smoothed ← slerp(current, rotation, rotation_smoothing)
13:        skeleton.set_bone_pose_rotation(bone_idx, smoothed)
14:    end for
15: end for

```

4.3. Generador de Paredes (WallGenerator.gd)

Sistema de generación procedural de paredes con agujeros en forma de siluetas humanas.

4.3.1. Variables de Exportación

```

1 @export var wall_spawn_interval: float = 3.0
2 @export var wall_speed: float = 5.0

```

Listing 7: Configuración del generador

4.3.2. Posturas Disponibles

El sistema define 7 posturas diferentes que los jugadores deben adoptar:

Cuadro 2: Posturas disponibles en el juego

Postura	Descripción	Dificultad
T_POSE	Brazos extendidos horizontalmente a los lados	Fácil
ARMS_UP	Ambos brazos levantados verticalmente	Fácil
ARMS_DOWN	Brazos pegados al cuerpo	Fácil
LEFT_ARM_UP	Solo brazo izquierdo arriba, derecho abajo	Media
RIGHT_ARM_UP	Solo brazo derecho arriba, izquierdo abajo	Media
SQUAT	Posición agachada con rodillas flexionadas	Difícil
JUMP	Posición de salto con cuerpo elevado	Difícil

4.3.3. Generación de Mesh con SurfaceTool

Las paredes se generan proceduralmente usando el SurfaceTool de Godot:

```

1 func create_t_pose_wall(st: SurfaceTool):
2     var left = -2.5
3     var right = 2.5
4     var top = 1.25
5     var bottom = -1.25
6
7     # Agujero T (brazos y cuerpo)
8     var arm_left = -1.0
9     var arm_right = 1.0
10    var arm_top = 0.75
11    var arm_bottom = 0.0
12
13    var body_left = -0.375
14    var body_right = 0.375
15    var body_top = 0.0
16    var body_bottom = -1.0
17
18    # Marco superior (encima de los brazos)
19    add_rect(st, left, top, right, arm_top)
20
21    # Marcos laterales de los brazos
22    add_rect(st, left, arm_top, arm_left, arm_bottom)
23    add_rect(st, arm_right, arm_top, right, arm_bottom)
24

```

```

25  # Marcos laterales del cuerpo
26  add_rect(st, left, arm_bottom, body_left, body_bottom)
27  add_rect(st, body_right, arm_bottom, right, body_bottom)
28
29  # Marco inferior
30  add_rect(st, left, body_bottom, right, bottom)

```

Listing 8: Generación de pared T-Pose

4.4. Evaluador de Posturas (PoseEvaluator.gd)

Componente que evalúa la coincidencia entre la postura del jugador y la postura objetivo.

4.4.1. Algoritmo de Evaluación

La evaluación se basa en el análisis de ángulos de las articulaciones:

```

1  func evaluate_t_pose(skeleton: Skeleton3D) -> float:
2      # T-Pose: brazos horizontales (0 grados)
3      var left_arm = get_arm_angle(skeleton, "Brazo.L")
4      var right_arm = get_arm_angle(skeleton, "Brazo.R")
5
6      # Score inversamente proporcional a la desviación
7      var left_score = 1.0 - (abs(left_arm) / 90.0)
8      var right_score = 1.0 - (abs(right_arm) / 90.0)
9
10     return clamp((left_score + right_score) / 2.0, 0.0, 1.0)

```

Listing 9: Evaluación de T-Pose

El score final se calcula como:

$$\text{score} = \frac{1}{2} \left(1 - \frac{|\theta_L|}{90} + 1 - \frac{|\theta_R|}{90} \right) \quad (2)$$

donde θ_L y θ_R son los ángulos de los brazos izquierdo y derecho respectivamente.

4.5. Selección de Personajes (character_select.gd)

Sistema de selección de personajes para modo de 2 jugadores.

4.5.1. Flujo de Selección

1. Se muestran 4 opciones de personajes
2. Jugador 1 selecciona su personaje (marcado en verde)
3. Jugador 2 selecciona su personaje (marcado en magenta)
4. Las selecciones se almacenan en metadatos del árbol de escenas
5. Se carga la escena principal con los personajes seleccionados


```

1 func _on_character_selected(char_name: String):
2     if current_player == 1:
3         get_tree().root.set_meta("personaje1", char_name)
4         current_player = 2
5     else:
6         get_tree().root.set_meta("personaje2", char_name)
7         get_tree().change_scene_to_file("res://scenes/mainScene.
            tscn")

```

Listing 10: Almacenamiento de selección

5. Sistemas de Juego

5.1. Sistema de Puntuación y Progresión

El sistema de puntuación está diseñado para recompensar la precisión y la consistencia:

Cuadro 3: Sistema de puntuación

Acción	Puntos/Penalización
Pasar pared correctamente	+10 puntos
Colisión con pared	-1 vida
Vidas iniciales	3 vidas
Umbral de acierto (versión Pygame)	40 % de coincidencia
Tolerancia (versión Godot)	0.3 (30 %)

5.1.1. Progresión de Dificultad (Versión Pygame)

La dificultad aumenta progresivamente:

```

1 # Subir de nivel cada 500 puntos
2 if self.score > self.level * 500:
3     self.level += 1
4     # Aumentar velocidad de paredes
5     self.wall_speed = min(4, 1.5 + self.level * 0.3)
6     # Reducir tiempo disponible
7     self.time_per_wall = max(7.0, 10.0 - self.level * 0.3)

```

Listing 11: Sistema de progresión

5.2. Sistema de Colisiones

El sistema de colisiones en Godot utiliza Area3D para detectar cuando un jugador entra en contacto con una pared.

5.2.1. Configuración de Áreas de Colisión

```

1 func create_fallback_collision_area(player: Node3D,
2                                     player_idx: int) -> Area3D:
3     var area = Area3D.new()
4     area.name = "PlayerCollisionArea_" + str(player_idx)
5     area.set_meta("player_idx", player_idx)
6
7     var collision_shape = CollisionShape3D.new()
8     var capsule = CapsuleShape3D.new()
9     capsule.radius = 0.3
10    capsule.height = 1.5
11    collision_shape.shape = capsule
12    collision_shape.position = Vector3(0, 0.75, 0)
13
14    area.add_child(collision_shape)
15    player.add_child(area)
16
17    # Conectar señal de colision
18    area.area_entered.connect(
19        _on_player_area_entered.bind(player_idx)
20    )
21
22    return area

```

Listing 12: Creación de área de colisión para jugador

5.2.2. Diagrama de Colisiones

Sistema de Validación de Colisiones:

- **Pared con agujero:** Se genera una forma irregular en cada pared
- **Jugador:** Representado por sus puntos clave de pose
- **Movimiento:** Las paredes avanzan hacia el jugador en el eje Z
- **Validación:**
 - Si el jugador encaja en el agujero: **+10 puntos**
 - Si el jugador choca con la pared: **-1 vida**

Figura 2: Diagrama de validación de colisión

5.3. Animación de Personajes en Tiempo Real

La animación de los personajes se realiza mediante la manipulación directa del Skeleton3D:

5.3.1. Proceso de Animación

1. **Recepción de datos:** Se reciben coordenadas 3D de las articulaciones.

2. **Cálculo de dirección:** Para cada extremidad, se calcula el vector dirección:

$$\vec{d} = \frac{\vec{p}_{end} - \vec{p}_{start}}{|\vec{p}_{end} - \vec{p}_{start}|} \quad (3)$$

3. **Conversión a rotación:** Se utiliza `Basis.looking_at()` para convertir el vector en una rotación quaternion.

4. **Interpolación:** Se aplica interpolación esférica (SLERP) para suavizar el movimiento:

$$q_{new} = \text{slerp}(q_{current}, q_{target}, t) \quad (4)$$

donde t es el factor de suavizado (típicamente 0.08-0.2).

5. **Aplicación:** La rotación suavizada se aplica al hueso correspondiente.

```

1 const UPDATE_ORDER = [
2     "upper_arm_L", "forearm_L",    # Brazo izquierdo
3     "upper_arm_R", "forearm_R",    # Brazo derecho
4     "thigh_L",    "shin_L",        # Pierna izquierda
5     "thigh_R",    "shin_R"         # Pierna derecha
6 ]

```

Listing 13: Orden de actualización de huesos

5.4. Generación Dinámica de Paredes

Las paredes se generan cada intervalo de tiempo usando SurfaceTool de Godot:

```

1 func add_rect(st: SurfaceTool, x1, y1, x2, y2):
2     """Crea dos triangulos formando un rectangulo"""
3     var z = 0.0
4
5     # Vertices
6     var v1 = Vector3(x1, y1, z)
7     var v2 = Vector3(x2, y1, z)
8     var v3 = Vector3(x2, y2, z)
9     var v4 = Vector3(x1, y2, z)
10
11     # Triangulo 1
12     st.add_vertex(v1)
13     st.add_vertex(v2)
14     st.add_vertex(v3)
15
16     # Triangulo 2
17     st.add_vertex(v1)
18     st.add_vertex(v3)
19     st.add_vertex(v4)

```

Listing 14: Función auxiliar para crear rectángulos

6. Detalles Técnicos de Implementación

6.1. Protocolo de Comunicación WebSocket

La comunicación entre el servidor Python y el cliente Godot se realiza mediante WebSocket sobre TCP.

Cuadro 4: Especificaciones del protocolo WebSocket

Parámetro	Valor
Dirección	localhost (127.0.0.1)
Puerto	8765
Protocolo	WebSocket (RFC 6455)
Formato de datos	JSON
Frecuencia de envío	30 mensajes/segundo
Latencia típica	<50ms

6.1.1. Estructura del Mensaje JSON

```

1 {
2   "poses": [
3     # Jugador 1: Array de 13 puntos
4     [
5       {"x": 0.52, "y": 0.18, "z": -0.12}, # Nariz (0)
6       {"x": 0.38, "y": 0.42, "z": -0.08}, # Hombro L (11)
7       {"x": 0.62, "y": 0.41, "z": -0.09}, # Hombro R (12)
8       {"x": 0.25, "y": 0.55, "z": -0.05}, # Codo L (13)
9       {"x": 0.75, "y": 0.54, "z": -0.06}, # Codo R (14)
10      {"x": 0.15, "y": 0.68, "z": -0.03}, # Muneca L (15)
11      {"x": 0.85, "y": 0.67, "z": -0.04}, # Muneca R (16)
12      {"x": 0.42, "y": 0.72, "z": -0.10}, # Cadera L (23)
13      {"x": 0.58, "y": 0.71, "z": -0.11}, # Cadera R (24)
14      {"x": 0.40, "y": 0.88, "z": -0.08}, # Rodilla L (25)
15      {"x": 0.60, "y": 0.87, "z": -0.09}, # Rodilla R (26)
16      {"x": 0.38, "y": 0.98, "z": -0.05}, # Tobillo L (27)
17      {"x": 0.62, "y": 0.97, "z": -0.06} # Tobillo R (28)
18     ],
19     # Jugador 2 (mismo formato)
20     [...]
21   ]
22 }
```

Listing 15: Formato de mensaje de pose

6.1.2. Interpretación de Coordenadas

- **x**: Posición horizontal normalizada [0, 1], donde 0 = izquierda, 1 = derecha
- **y**: Posición vertical normalizada [0, 1], donde 0 = arriba, 1 = abajo
- **z**: Profundidad relativa, valores negativos = más cerca de la cámara

6.2. Transformación de Coordenadas

Las coordenadas de MediaPipe se transforman al espacio 3D de Godot:

```

1 func transform_pose_to_3d(point: Dictionary) -> Vector3:
2     var x = (point.x - 0.5) * 2.0 * scale_factor
3     var y = (0.5 - point.y) * 2.0 * scale_factor
4     var z = point.z * depth_scale
5
6     if mirror_mode:
7         x = -x
8
9     if lock_z_movement:
10        z = 0.0
11
12    return Vector3(x, y, z) + offset

```

Listing 16: Transformación de coordenadas

6.3. Material de Paredes

Las paredes utilizan un material semitransparente para permitir visibilidad:

```

1 var material = StandardMaterial3D.new()
2 material.albedo_color = Color(0.2, 0.2, 0.8, 0.8) # Azul 80%
3 material.transparency = BaseMaterial3D.TRANSPARENCY_ALPHA
4 material.cull_mode = BaseMaterial3D.CULL_DISABLED # Doble cara

```

Listing 17: Configuración de material

6.4. Gestión de Memoria y Rendimiento

6.4.1. Limpieza de Paredes

Las paredes se eliminan cuando salen del área de juego:

```

1 func _process(delta):
2     for wall in get_children():
3         wall.position.z -= wall_speed * delta
4
5         # Eliminar paredes fuera de rango
6         if wall.position.z < -30:
7             wall.queue_free()

```

Listing 18: Limpieza automática de paredes

6.4.2. Pool de Conexiones WebSocket

El servidor mantiene un conjunto de clientes conectados:

```

1 connected_clients = set()

```

```

3 async def stream_pose(websocket):
4     connected_clients.add(websocket)
5     print(f"Cliente conectado. Total: {len(connected_clients)}")
6
7     try:
8         await websocket.wait_closed()
9     finally:
10        connected_clients.remove(websocket)
11        print(f"Cliente desconectado. Total: {len(connected_clients)}")

```

Listing 19: Gestión de clientes conectados

7. Versión Standalone (Pygame)

Además de la versión Godot, el proyecto incluye una implementación standalone en Python utilizando Pygame, contenida en el archivo `Entrega.ipynb`.

7.1. Características de la Versión Pygame

- **Autónoma:** No requiere servidor separado
- **Segmentación de fondo:** Utiliza MediaPipe Selfie Segmentation
- **Interfaz 2D:** Visualización simplificada del esqueleto
- **Sistema de niveles:** Dificultad progresiva

7.2. Detección de Posición de Brazos

```

1 def check_arm_position(self, shoulder_idx, elbow_idx, wrist_idx):
2     """
3     Determina si un brazo esta arriba, al lado, o abajo
4     """
5     shoulder = self.detected_landmarks[shoulder_idx]
6     elbow = self.detected_landmarks[elbow_idx]
7     wrist = self.detected_landmarks[wrist_idx]
8
9     # Mueneca mas alta que hombro = arriba
10    if wrist[1] < shoulder[1] - 0.15:
11        return 'up'
12
13    # Muneca a la misma altura y alejada = al lado
14    if abs(wrist[1] - shoulder[1]) < 0.2:
15        if abs(wrist[0] - shoulder[0]) > 0.15:
16            return 'side'
17
18    return 'down'

```

Listing 20: Algoritmo de detección de posición de brazo

7.3. Sistema de Coincidencia de Pose

El cálculo de coincidencia compara la pose detectada con la pose objetivo:

```

1 def calculate_match_score(self):
2     target = self.current_pose["check_points"]
3
4     left_arm = self.check_arm_position(11, 13, 15)
5     right_arm = self.check_arm_position(12, 14, 16)
6     left_leg = self.check_leg_position(23, 25, 27)
7     right_leg = self.check_leg_position(24, 26, 28)
8
9     points = 0
10    max_points = 8 # 4 brazos + 4 piernas
11
12    # Verificar brazos
13    if target.get("left_arm_up") == (left_arm == "up"):
14        points += 1
15    if target.get("right_arm_up") == (right_arm == "up"):
16        points += 1
17    if target.get("left_arm_side") == (left_arm == "side"):
18        points += 1
19    if target.get("right_arm_side") == (right_arm == "side"):
20        points += 1
21
22    # Verificar piernas
23    if target.get("left_leg_up") == (left_leg == "up"):
24        points += 1
25    # ... (similar para otras extremidades)
26
27    return int((points / max_points) * 100)

```

Listing 21: Cálculo de puntuación de coincidencia

8. Flujo de Ejecución

8.1. Inicio del Juego

1. Usuario inicia el servidor MediaPipe (python mediapipe_util.py)
2. Usuario abre el proyecto Godot y ejecuta la escena principal
3. Aparece el menú principal
4. Selecciona modo de juego (1 o 2 jugadores)
5. Selecciona personajes para cada jugador
6. Se carga mainScene.tscn
7. mainscript.gd inicializa:
 - Conexión WebSocket a localhost:8765

- Carga de personajes seleccionados
- Configuración de cámara y luces
- Inicialización de WallGenerator
- Creación del HUD
- Inicio de música de fondo

Estado	Transición	Siguiente Estado
Menú Principal	Pulsar "Iniciar"	Selección Personajes
Selección Personajes	Confirmar selección	Gameplay
Gameplay	Vidas = 0	Game Over
Game Over	Pulsar "Reiniciar"	Menú Principal

Flujo del juego:

Menú Principal → Selección Personajes → Gameplay → Game Over → Menú Principal

Figura 3: Diagrama de estados del juego

8.2. Loop Principal (_process)

1. **Polling WebSocket:** Procesar paquetes entrantes
2. **Parseo JSON:** Convertir datos recibidos a diccionario
3. **Actualización de jugadores:** Aplicar poses a esqueletos
4. **Movimiento de paredes:** Desplazar paredes hacia jugadores
5. **Detección de colisiones:** Verificar choques con paredes
6. **Actualización de puntuación:** Sumar puntos o restar vidas
7. **Generación de paredes:** Crear nuevas cada intervalo
8. **Limpieza:** Eliminar paredes fuera de rango
9. **Actualización HUD:** Refrescar interfaz de usuario

8.3. Validación de Paso por Pared

Algorithm 2 Validación de paso correcto

```

1: for cada wall en walls do
2:   if wall.z < 1,0 and wall.z > -1,0 then
3:     if wall no ha sido validada then
4:       pose_type ← wall.get_meta("pose_type")
5:       for cada player en players do
6:         player_pose ← evaluate_pose(player.skeleton)
7:         if player_pose = pose_type then
8:           score ← score + 10
9:         else
10:          collision_detected ← CheckCollision(player, wall)
11:          if collision_detected then
12:            lives ← lives - 1
13:          end if
14:        end if
15:      end for
16:      wall.set_meta("validated", true)
17:    end if
18:  end if
19: end for

```

9. Configuración y Parámetros Ajustables

9.1. Parámetros del Servidor MediaPipe

Cuadro 5: Parámetros configurables del servidor

Parámetro	Valor por defecto	Descripción
num_poses	2	Número máximo de personas a detectar
model_asset_path	pose_landmarker_full.task	Modelo de MediaPipe a utilizar
running_mode	VIDEO	Modo de procesamiento para video
CAP_PROP_FRAME_WIDTH	640	Ancho de captura de cámara
CAP_PROP_FRAME_HEIGHT	480	Alto de captura de cámara

9.2. Parámetros del Juego (Godot)

Cuadro 6: Variables exportables en GDScript

Variable	Valor	Descripción
wall_spawn_interval	3.0 s	Intervalo entre paredes
wall_speed	5.0 u/s	Velocidad de movimiento de paredes
target_height	2.0 m	Altura objetivo del personaje
scale_factor	0.7	Factor de escala de movimiento
depth_scale	2.0	Escala para coordenada Z
mirror_mode	true	Efecto espejo en movimientos
movement_smoothing	0.2	Suavizado de posición
rotation_smoothing	0.08	Suavizado de rotación
tolerance	0.3	Tolerancia para evaluación de pose
points_per_success	100	Puntos por pose correcta

9.3. Posiciones en la Escena 3D

Cuadro 7: Posiciones de elementos en la escena

Elemento	Posición (x, y, z)	Color/Notas
Jugador 1	(-1.0, 0.0, 0.0)	Verde
Jugador 2	(1.0, 0.0, 0.0)	Magenta
Cámara	(0, 2, 8)	Vista frontal
Spawn de paredes	(0, 2, 20)	Fondo de escena
Límite de limpieza	$z < -30$	Eliminación automática

10. Personajes del Juego

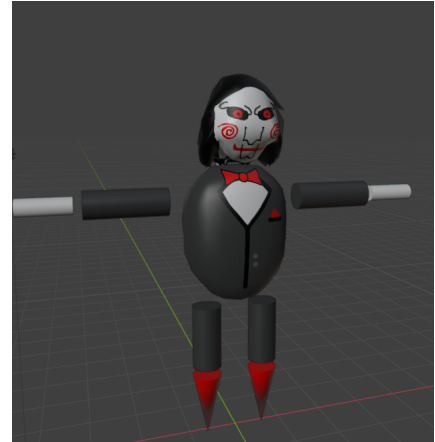
El juego incluye 4 personajes basados en iconos de la cultura popular, cada uno modelado y rigado en Blender utilizando el complemento Rigify para asegurar compatibilidad con el sistema de animación de Godot.

10.1. Proceso de Creación de Personajes

1. **Modelado:** Creación de la malla 3D en Blender
2. **Rigging:** Aplicación de esqueleto con Rigify
3. **Skinning:** Asignación de pesos de deformación
4. **Exportación:** Formato .glb con armadura incluida

5. **Importación:** Carga en Godot como escena .tscn

10.2. SAW



Descripción: Personaje inspirado en el antagonista de la saga de películas de terror “Saw” (2004). Representa un personaje oscuro y misterioso.

Origen: Cine de Horror - “Saw”

- **Archivo fuente:** SAW.blend1
- **Archivo exportado:** Saw.glb
- **Tipo:** Humanoide
- **Características:** Expresión seria, vestimenta oscura

10.3. ET



Descripción: Personaje alienígena basado en el icónico extraterrestre de “E.T. the Extra-Terrestrial” (1982) de Steven Spielberg.

Origen: Ciencia Ficción - “E.T.”

- **Archivo fuente:** ET.blend1
- **Archivo exportado:** ET.glb
- **Tipo:** Humanoide fantástico
- **Características:** Proporciones exageradas, piel texturizada

10.4. ELEVEN



Descripción: Personaje inspirado en Eleven de la serie de Netflix “Stranger Things” (2016-presente), la joven con poderes telequinéticos.

Origen: Serie de TV - “Stranger Things”

- **Archivo fuente:** Eleven2.blend1
- **Archivo exportado:** Eleven.glb
- **Tipo:** Humanoide
- **Características:** Proporciones juveniles, diseño moderno

10.5. HOMER



Descripción: Personaje basado en Homer Simpson de la serie animada “The Simpsons” (1989-presente), añadiendo un toque de humor al juego.

Origen: Animación - “The Simpsons”

- **Archivo fuente:** HOMER_SIMPSON.blend1
- **Archivo exportado:** Homer.glb
- **Tipo:** Humanoide caricaturesco
- **Características:** Cuerpo rechoncho, color amarillo

10.6. Sistema de Rigging con Rigify

El proceso de rigging sigue un flujo estandarizado:

```

1 # 1. Seleccionar modelo
2 bpy.ops.object.select_all(action='DESELECT')
3 model.select_set(True)
4
5 # 2. Agregar metarig de Rigify
6 bpy.ops.object.armature_human_metarig_add()
7
8 # 3. Ajustar huesos al modelo
9 for bone in metarig.bones:
10     bone.head = corresponding_vertex_position
11     bone.tail = calculate_tail_position(bone)
12
13 # 4. Generar rig final
14 bpy.ops.pose.rigify_generate()
15
16 # 5. Aplicar automatic weights
17 bpy.ops.object.parent_set(type='ARMATURE_AUTO')
18
19 # 6. Refinar pesos manualmente si es necesario

```

Listing 22: Proceso de rigging en Blender (pseudocódigo)

10.7. Estructura del Esqueleto Exportado

Los personajes exportados contienen los siguientes huesos relevantes para la animación:

```

1 Skeleton3D
2 |-- spine
3 |-- upper_arm.L / upper_arm.R
4 |-- forearm.L / forearm.R
5 |-- hand.L / hand.R
6 |-- thigh.L / thigh.R
7 |-- shin.L / shin.R
8 +-- foot.L / foot.R

```

11. Sistema de Audio

El juego incluye un sistema de audio completo diseñado para mejorar la experiencia de juego mediante retroalimentación sonora inmersiva.

11.1. Música de Fondo

Cuadro 8: Especificaciones de la música de fondo

Propiedad	Valor
Archivo	turiruriru-01.mp3
Fuente original	turiruriru-01.mmpz (LMMS)
Género	Electrónica/J-Pop
Modo	Loop continuo
Volumen	Submix para no interferir con SFX

11.2. Efectos de Sonido

Todos los efectos fueron creados manualmente en LMMS:

Cuadro 9: Efectos de sonido del juego

Efecto	Archivo	Duración	Uso
Click	soundEffect_buttonClick.ogg	0.2s	Al presionar botones
Hover	soundEffect_buttonHover.ogg	0.15s	Al pasar cursor sobre botón

11.3. Integración en Godot

```

1 @export var audioStreamPlayer: AudioStreamPlayer3D
2
3 func _ready():
4     if audioStreamPlayer:
5         audioStreamPlayer.play()
6
7 func play_button_sound():
8     var audio = AudioStreamPlayer.new()
9     audio.stream = preload("res://assets/SoundEffects/buttonClick.
10         ogg")
11     add_child(audio)
12     audio.play()
13     # Auto-eliminar al terminar
14     audio.finished.connect(audio.queue_free)

```

Listing 23: Reproducción de audio en Godot

11.4. Flujo de Creación de Audio

1. Composición en LMMS (.mmpz)
2. Selección de instrumentos y sintetizadores
3. Ajuste de duración y envolventes

4. Exportación a .ogg (Vorbis) para efectos
5. Exportación a .mp3 para música (mayor compresión)
6. Importación en Godot (automática)
7. Configuración de AudioStreamPlayer en escenas

12. Resultados y Pruebas

12.1. Métricas de Rendimiento

Se realizaron pruebas de rendimiento en el siguiente hardware:

Cuadro 10: Especificaciones del sistema de pruebas

Componente	Especificación
Sistema Operativo	Linux (Ubuntu-based)
Cámara	Webcam integrada 720p
Resolución de juego	1280x720

Cuadro 11: Métricas de rendimiento observadas

Métrica	Versión Godot	Versión Pygame
FPS de detección (MediaPipe)	30	30
FPS de renderizado	60	30
Latencia total	<100ms	<80ms
Uso de CPU	15-25 %	20-30 %
Uso de RAM	400MB	300MB

12.2. Precisión de Detección de Pose

La precisión de detección se evaluó cualitativamente:

- **Iluminación óptima:** Detección consistente al 95 %+
- **Iluminación baja:** Precisión reducida, falsos negativos
- **Fondo complejo:** Detección estable gracias a segmentación
- **Movimientos rápidos:** Ligero retraso en tracking

12.3. Usabilidad

El sistema fue probado informalmente con usuarios:

- Tiempo de aprendizaje: <2 minutos para entender el concepto

- Calibración: No requerida, funciona “plug and play”
- Tolerancia: Umbral del 40 % permite juego accesible
- Retroalimentación: Visual clara con porcentaje de coincidencia

13. Guía de Instalación y Uso

13.1. Requisitos del Sistema

- Python 3.11 con pip
- Godot Engine 4.5 (para versión completa)
- Cámara web funcional
- Conexión a Internet (solo para descarga inicial)

13.2. Instalación de Dependencias Python

```

1 # Crear entorno virtual (recomendado)
2 conda create -n brainwall python=3.10
3 conda activate brainwall
4
5 # Instalar dependencias
6 pip install opencv-python mediapipe websockets asyncio numpy
7
8 # Para version Pygame standalone
9 pip install pygame

```

Listing 24: Instalación de dependencias

13.3. Ejecución del Sistema

13.3.1. Versión Completa (Godot + MediaPipe)

```

1 # Terminal 1: Iniciar servidor MediaPipe
2 cd Entrega_FINAL_VC
3 python mediapipe_util.py
4
5 # Terminal 2 / Godot Editor
6 # Abrir proyecto: BrainWallGodot/brain-wall/project.godot
7 # Ejecutar escena principal (F5)

```

Listing 25: Ejecución del sistema completo

13.3.2. Versión Standalone (Pygame)

```

1 # Abrir Jupyter Notebook
2 jupyter notebook Entrega.ipynb
3
4 # O ejecutar directamente el código Python
5 python -c "exec(open('Entrega.ipynb').read())"

```

Listing 26: Ejecución de versión Pygame

13.4. Controles del Juego

Cuadro 12: Controles del juego

Tecla	Acción
ESPACIO	Iniciar juego / Confirmar selección
ESC	Pausar / Volver al menú
Movimiento corporal	Control del personaje

14. Galería del Juego

14.1. Escenas Principales

- **Menú Principal:** Pantalla de inicio con opciones de juego
- **Selección de Personajes:** Interfaz para elegir entre 4 personajes
- **Gameplay:** Vista 3D con personajes animados y paredes
- **HUD:** Puntuación, vidas y postura actual

14.2. Elementos Visuales

Cuadro 13: Elementos visuales de la escena

Elemento	Descripción	Propiedades
Plataforma	Base del escenario	Gris oscuro, 10x0.5x5u
Paredes	Obstáculos con siluetas	Azul semitransparente
Personajes	Avatares 3D rigeados	Colores distintivos
Skybox	Fondo HDR	Nebulosa espacial
Iluminación	Luz direccional + ambiente	Tono cálido

15. Limitaciones y Trabajo Futuro

15.1. Limitaciones Actuales

1. Dependencia de hardware:

- La calidad de detección depende de la cámara web
- Requiere iluminación adecuada para tracking preciso
- Rendimiento variable según GPU/CPU

2. Limitaciones de software:

- Máximo 2 jugadores simultáneos
- Sin persistencia de puntuaciones (no hay base de datos)
- Paredes con tamaño fijo
- Validación de pose basada en ángulos, no en volumen 3D

3. Limitaciones de red:

- Solo funciona en localhost (mismo equipo)
- Sin autenticación ni cifrado en WebSocket

15.2. Mejoras Futuras Propuestas

1. Corto plazo:

- Añadir más efectos de sonido (colisión, éxito, nivel)
- Implementar sistema de partículas para feedback visual
- Crear más variaciones de posturas
- Añadir animaciones de idle para personajes

2. Medio plazo:

- Modo multijugador en red (LAN/Internet)
- Sistema de puntuaciones persistente (SQLite/Firebase)
- Dificultad adaptativa mediante IA
- Soporte para más de 2 jugadores
- Personalización de personajes

3. Largo plazo:

- Versión móvil (Android/iOS)
- Integración con VR/AR
- Editor de niveles personalizado
- Modo historia con progresión
- Sistema de logros y desbloques

15.3. Posibles Optimizaciones

- Implementar predicción de pose para reducir latencia percibida
- Utilizar modelo de MediaPipe más ligero para hardware limitado
- Cachear meshes de paredes en lugar de generarlos cada vez
- Implementar LOD (Level of Detail) para personajes lejanos

16. Conclusiones

Este proyecto ha demostrado la viabilidad de crear un sistema interactivo de entretenimiento utilizando técnicas modernas de visión por computador. Los principales logros alcanzados son:

16.1. Objetivos Cumplidos

1. **Detección de pose en tiempo real:** Se logró implementar un sistema de detección de 33 puntos corporales funcionando a 30 FPS utilizando MediaPipe, sin necesidad de hardware especializado.
2. **Comunicación cliente-servidor eficiente:** El protocolo WebSocket demostró ser una solución robusta para la transmisión de datos de pose con latencia inferior a 100ms.
3. **Animación de esqueleto:** Se implementó exitosamente la traducción de datos de pose a rotaciones de huesos del Skeleton3D de Godot, con interpolación suave.
4. **Sistema de juego completo:** El juego incluye selección de personajes, sistema de puntuación, vidas, y progresión de dificultad.
5. **Dos versiones funcionales:** Se desarrollaron tanto una versión standalone en Pygame como una versión completa 3D en Godot.
6. **Contenido audiovisual original:** Todos los modelos 3D, música y efectos de sonido fueron creados específicamente para este proyecto.

16.2. Conocimientos Aplicados

El proyecto integra conocimientos de múltiples áreas de la informática:

- **Visión por Computador:** Detección de pose, segmentación de imagen
- **Gráficos 3D:** Modelado, rigging, animación de esqueletos
- **Desarrollo de Videojuegos:** Sistemas de juego, física, colisiones
- **Redes:** Comunicación WebSocket, serialización JSON
- **Programación Asíncrona:** Servidor WebSocket con asyncio
- **Diseño de Audio:** Composición musical, efectos de sonido

16.3. Reflexiones Finales

El desarrollo de Vision Wall ha permitido explorar el potencial de las tecnologías de detección de pose para aplicaciones interactivas. MediaPipe ha demostrado ser una herramienta accesible y potente que democratiza el acceso a capacidades de visión por computador avanzadas.

La integración con Godot Engine mostró la flexibilidad del motor para proyectos que combinan entrada no convencional con gráficos 3D. La arquitectura cliente-servidor

adoptada facilita la separación de responsabilidades y permitiría escalar el sistema a escenarios más complejos.

El proyecto cumple satisfactoriamente con los objetivos planteados para la asignatura de Visión por Computador, demostrando una aplicación práctica y entretenida de los conceptos estudiados.

Referencias

1. Google. (2024). *MediaPipe Pose Landmarker*. https://developers.google.com/mediapipe/solutions/vision/pose_landmarker
2. Godot Engine. (2024). *Godot Engine Documentation 4.x*. <https://docs.godotengine.org/en/stable/>
3. OpenCV. (2024). *OpenCV-Python Tutorials*. https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html
4. Blender Foundation. (2024). *Blender Reference Manual*. <https://docs.blender.org/manual/en/latest/>
5. WebSocket Protocol. (2011). *RFC 6455 - The WebSocket Protocol*. IETF.
6. LMMS. (2024). *LMMS Documentation*. <https://docs.lmms.io/>
7. Lugaresi, C. et al. (2019). *MediaPipe: A Framework for Building Perception Pipelines*. Google Research. arXiv:1906.08172

Anexo A: Índices de MediaPipe Pose Landmarks

Cuadro 14: Índices completos de landmarks de MediaPipe Pose

Índice	Landmark	Índice	Landmark
0	Nariz	17	Meñique izq
1	Ojo interno izq	18	Meñique der
2	Ojo izq	19	Índice izq
3	Ojo externo izq	20	Índice der
4	Ojo interno der	21	Pulgar izq
5	Ojo der	22	Pulgar der
6	Ojo externo der	23	Cadera izq
7	Oreja izq	24	Cadera der
8	Oreja der	25	Rodilla izq
9	Boca izq	26	Rodilla der
10	Boca der	27	Tobillo izq
11	Hombro izq	28	Tobillo der
12	Hombro der	29	Talón izq
13	Codo izq	30	Talón der
14	Codo der	31	Pie índice izq
15	Muñeca izq	32	Pie índice der
16	Muñeca der		

Anexo B: Estructura de Archivos del Proyecto

```

1 Entrega_FINAL_VC/
2 |-- Entrega.ipynb                                # Version Pygame standalone
3 |-- mediapipe_util.py                            # Servidor WebSocket
4 |-- pose_landmarker_full.task                    # Modelo MediaPipe
5 |-- README.md                                    # Instrucciones del proyecto
6 |
7 |-- BrainWallGodot/brain-wall/                  # Proyecto Godot
8 |   |-- project.godot
9 |   |-- PoseReceiver.gd
10 |   |-- main_menu.gd
11 |   |-- scenes/
12 |     |-- mainScene.tscn
13 |     |-- main_menu.tscn
14 |     |-- CharacterSelect.tscn
15 |     |-- WallGenerator.gd
16 |     |-- PoseEvaluator.gd
17 |     +-- game_ui.tscn
18 |   |-- Scripts/
19 |     |-- mainscript.gd
20 |     |-- character_select.gd
21 |     |-- game_manager.gd
22 |     +-- wall.gd
23 |   +-- assets/
24 |     |-- Characters/
25 |     |-- models/

```

```
26 |      |-- music/
27 |      +-- SoundEffects/
28 |
29 |-- Modelados/                                # Archivos Blender
30 | |-- SAW.blend1
31 | |-- ET.blend1
32 | |-- Eleven2.blend1
33 | |-- HOMER_SIMPSON.blend1
34 | +-- Wall[1-5].blend1
35 |
36 |-- Sonido/                                    # Archivos LMMS
37 | |-- turiruriru-01.mmpz
38 | |-- soundEffect_buttonClick.mmpz
39 | +-- soundEffect_buttonHover.mmpz
40 |
41 +-- INFORME/
42 +-- MEMORIA.tex                                # Este documento
```