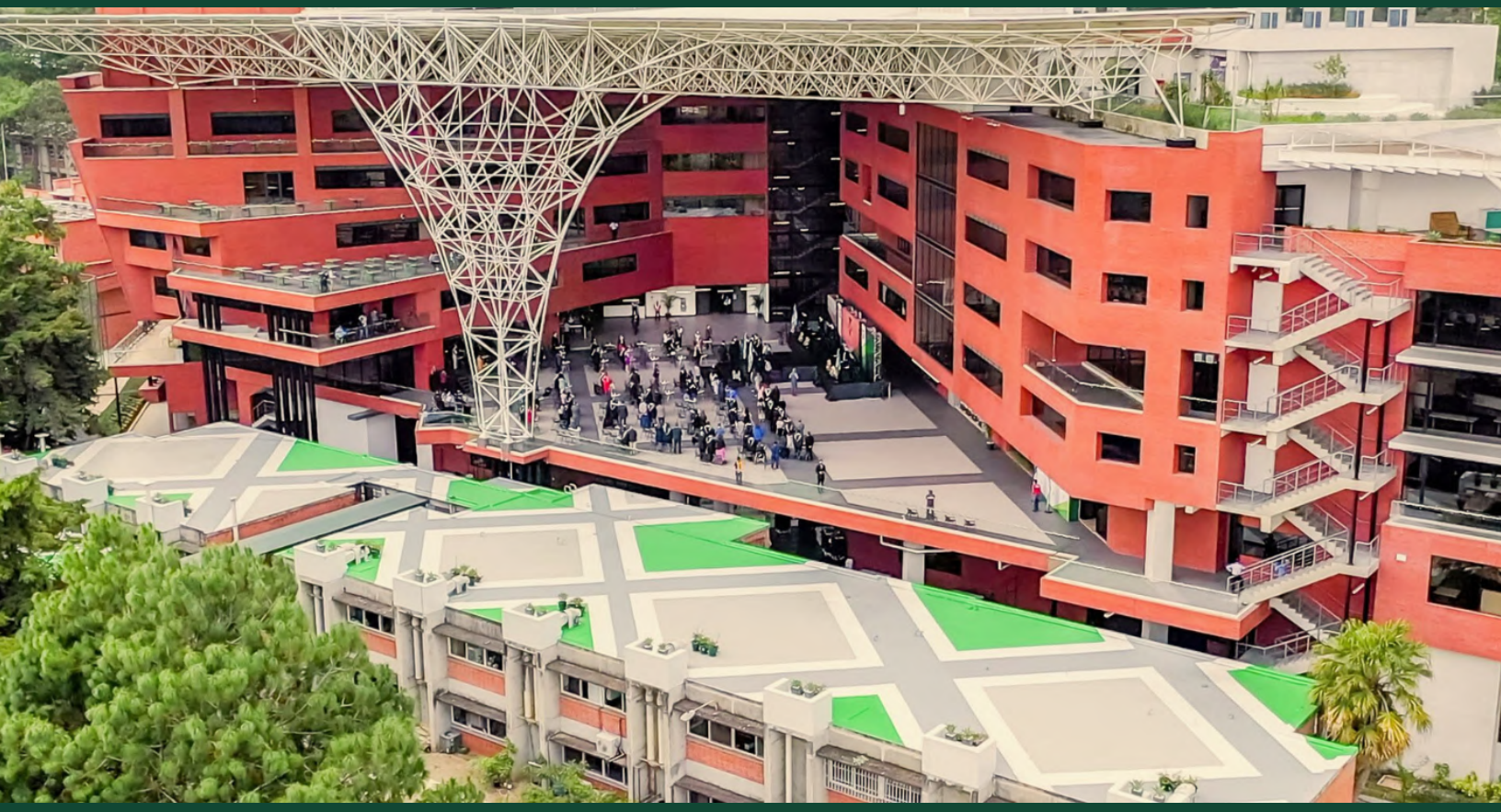
Desarrollo de plataforma de proyección interactiva basada en captura de movimiento para el entorno dinámico Robotat

Juan Daniel Gerardo Cortez Menéndez



UVG | UNIVERSIDAD DEL VALLE DE GUATEMALA

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Desarrollo de plataforma de proyección interactiva basada en captura de movimiento para el entorno dinámico Robotat

Trabajo de graduación presentado por Juan Daniel Gerardo Cortez Menéndez para optar al grado académico de Licenciado en Ingeniería Mecatrónica

Guatemala,

2025

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

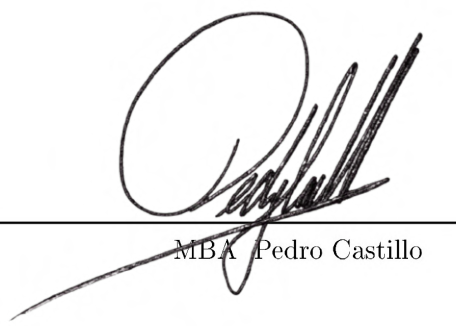
Desarrollo de plataforma de proyección interactiva basada en captura de movimiento para el entorno dinámico Robotat

Trabajo de graduación presentado por Juan Daniel Gerardo Cortez Menéndez para optar al grado académico de Licenciado en Ingeniería Mecatrónica

Guatemala,

2025

Vo.Bo.:



(f) _____

MBA. Pedro Castillo

(f) _____

M.Sc. Carlos Esquit

Fecha de aprobación: Guatemala, 20 de noviembre de 2025.

Índice general

Índice de figuras vii

Índice de cuadros viii

Resumen ix

Abstract x

1. Introducción 1

2. Antecedentes 3
2.1. Realidad aumentada en la Universidad del Valle de Guatemala 3
2.2. Realidad aumentada en procesos de manufactura y servicios 4
2.3. Realidad aumentada en la industria del entretenimiento 5
2.4. Realidad aumentada en la industria tecnológica 5

3. Justificación 7

4. Objetivos 8
4.1. Objetivo general 8
4.2. Objetivos específicos 8

5. Definición del problema y alcance 9
5.1. Definición del Problema 9
5.2. Alcance 10

6. Marco teórico 11
6.1. Protocolos de comunicación y transporte de datos 11
6.2. Direcciones IP y redes privadas 14
6.3. OptiTrack y sistemas de captura de movimiento 15
6.4. GStreamer y el concepto de *pipeline* multimedia 18
6.5. Interfaz gráfica con GTK 19
6.6. Control del mouse mediante ctypes y SendInput 19
6.7. Conceptos fundamentales de programación 20

6.8. Pymunk: simulación física y colisiones . . . 21

7. Servidor de proyección y aplicación estilo mosaico de transmisiones 23
7.1. Instalación de software preliminar y preparación del entorno de trabajo . . . 23
7.2. Desarrollo inicial de la aplicación estilo mosaico de transmisiones . . . 25
7.3. Integración de *watchdog* para manejo de desconexiones . . . 26
7.4. Integración de UDP al sistema . . . 26

8. Lectura de datos y mapeo de coordenadas 32
8.1. Lectura por medio de MQTT . . . 32
8.2. Transformación manual de los planos del sistema . . . 33
8.3. Transformación automática de los planos del sistema e integración de protocolo MQTT . . . 35

9. Primera validación del sistema mediante una aplicación interactiva 36
9.1. Control dinámico del cursor de mouse con coordenadas manuales . . . 37
9.2. Control dinámico del cursor de mouse con coordenadas extraídas del Robotat 38
9.3. Corrección del control dinámico del cursor con eventos reales mediante `ctypes` y `SendInput` . . . 39
9.4. Integración del sistema de proyección para la primera validación completa del sistema . . . 40

10. Segunda validación del sistema mediante aplicación interactiva 42
10.1. Estructura del código . . . 42
10.2. Archivo principal del código . . . 43
10.3. Subclases `DigitMatrix` y `Scoreboard` . . . 43
10.4. Subclases `GameState` y `UIManager` . . . 44
10.5. Subclase `Rink` . . . 45
10.6. Subclase `Player` . . . 46
10.7. Subclase `Puck` . . . 47
10.8. Subclase `Goal` . . . 48
10.9. Clase principal `Game` . . . 48
10.10 Integración del sistema de proyección para la segunda validación completa del sistema . . . 50

11. Conclusiones 51

12. Recomendaciones 52

13. Referencias 53

14. Anexos 56
14.1. Montaje del servidor de proyección e instalación de dependencias . . . 56
14.2. Instalación de dependencias y herramientas del cliente . . . 58
14.3. Configuración del servidor de proyección . . . 60
14.4. Configuración del cliente para iniciar una transmisión . . . 64
14.5. Configuración del cliente para modificar la calibración del segundo plano de detección . . . 66
14.6. Manejo del servidor de proyección y de la aplicación mosaico de transmisiones 68

14.7. Manejo del cliente para transmisiones y control de las aplicaciones de validación 69

Índice de figuras

1. Entorno dinámico de desarrollo Robotat 4
2. Sistema de realidad aumentada para ensamblaje manual 5
3. Realidad aumentada basada en proyección aplicada a las artes escénicas, observada desde tres perspectivas diferentes (lateral izquierda, central y lateral derecha, respectivamente) 6
4. Mesa interactiva de realidad aumentada 6

5. Esquema de comunicación basado en el patrón publicador–broker–suscriptor de MQTT 12
6. Comparación entre los métodos de comunicación de TCP y UDP 13
7. Concepto de port forwarding 15
8. Ejemplo ilustrativo del principio geométrico utilizado para la triangulación 3D de un marcador 16
9. Esquema general de comunicación entre el servidor OptiTrack y un cliente NatNet personalizado 17
10. Ejemplo general de un *pipeline* en GStreamer para lectura, decodificación y reproducción de audio y video 18
11. Comparación entre cuerpos *Kinematic* y *Dynamic* en simulaciones físicas . . . 22

12. Resultado final de la aplicación de proyección estilo mosaico desarrollada con GTK3 27

13. Visualización del segundo plano de detección (área roja) 34
14. Marcador transformado hacia el tercer plano (pantalla 1920×1080) 34

15. Validación del clic izquierdo mediante movimiento del marcador en el entorno Robotat y utilizando la plataforma *chess.com* 40
16. Validación del arrastre mediante clic izquierdo sostenido dentro de *Solitaire* . 41
17. Validación del clic derecho mediante el control por coordenada z en el escritorio de Windows 41

18. Secuencia de interacción física–virtual del juego de *air hockey* proyectado en el entorno Robotat 50

19. Escritorio del servidor Robotat 57
20. Archivos de la aplicación mosaico ubicados en la carpeta del proyecto 60
21. Fragmento del código responsable de la construcción del pipeline SRT 61
22. Dirección IP privada asignada al servidor en una red LAN 64
23. Selección del servicio personalizado en la configuración de emisión de OBS Studio 64
24. Ajustes de salida para transmisión mediante SRT en OBS Studio 65
25. Imagen proyectada utilizada para la calibración del segundo plano 68
26. Configuración de transmisión en OBS Studio 69

Índice de cuadros

1. Resumen de funciones implementadas en `main.cpp` 28
2. Descripción de las funciones implementadas en `StreamSlot.cpp` (Parte 1) . . 29
3. Descripción de las funciones implementadas en `StreamSlot.cpp` (Parte 2) . . 29
4. Descripción de los métodos públicos y atributos declarados en `StreamSlot.h` (Parte 1) 30
5. Descripción de los métodos privados y atributos internos en `StreamSlot.h` (Parte 2) 30
6. Descripción de las funciones implementadas en `Watchdog.cpp` 31
7. Descripción de los métodos y atributos declarados en `Watchdog.h` 31
8. Descripción de las funciones implementadas en el script de lectura MQTT . . 32
9. Descripción de las funciones y métodos utilizados en el script de transformación de planos 33
10. Funciones principales utilizadas en el script de control de mouse 37
11. Funciones principales utilizadas en el script completo de control de mouse mediante coordenadas de captura de movimiento 38
12. Funciones y estructuras principales utilizadas en el script de control de mouse con MQTT y homografía 39
13. Funciones principales empleadas en el módulo `main.py` del juego Air Hockey 2D 43
14. Funciones principales de las subclases `DigitMatrix` y `Scoreboard` 44
15. Funciones principales de las subclases `GameState` y `UIManager` 45
16. Funciones principales de la subclase `Rink` 46
17. Funciones principales de la subclase `Player` 46
18. Funciones principales de la subclase `Puck` 47
19. Funciones principales de la subclase `Goal` 48
20. Funciones principales de la clase `Game` 49

Resumen

La realidad aumentada ha experimentado avances significativos gracias a la integración de sistemas de captura de movimiento, procesamiento en tiempo real y visualización inmersiva. En este proyecto se desarrolló e implementó la infraestructura digital necesaria para crear un entorno interactivo en el entorno Robotat de la Universidad del Valle de Guatemala, capaz de sincronizar la información del movimiento del usuario capturada mediante el sistema *OptiTrack* con la proyección visual distribuida en el ambiente. Para ello se diseñó un sistema de gestión de contenido multimedia que permite controlar de forma remota la proyección a través de la red del Robotat, integrando protocolos como UDP y SRT para la transmisión eficiente de video y soporte para herramientas como OBS y GStreamer.

Se construyó además una aplicación capaz de interpretar datos de posición y orientación transmitidos mediante MQTT, traduciéndolos a coordenadas de pantalla y modificando la visualización en tiempo real. Esta infraestructura permitió implementar interacciones precisas basadas en marcadores pasivos, incluyendo el control de un cursor digital y el desarrollo de un juego de Air Hockey con detección simultánea de múltiples marcadores. El sistema demostró latencias reducidas entre 20 y 50 ms, estabilidad continua ante desconexiones y layouts dinámicos en un mosaico de transmisiones que organiza múltiples fuentes audiovisuales con baja latencia.

La validación final confirmó la sincronización efectiva entre captura y proyección, evidenciando la viabilidad del sistema para experiencias multiusuario. En conjunto, el proyecto establece una plataforma robusta para aplicaciones de realidad aumentada, simulaciones interactivas, experimentación tecnológica y entornos educativos inmersivos.

Palabras clave: realidad aumentada, captura de movimiento, proyección interactiva, Robotat, mosaico de transmisiones.

Abstract

Augmented reality has experienced significant advances through the integration of motion-capture systems, real-time processing, and immersive visualization. This project developed and implemented the digital infrastructure required to create an interactive environment within the Robotat space at Universidad del Valle de Guatemala, capable of synchronizing user motion data captured through the *OptiTrack* system with the visual projection distributed throughout the environment. To achieve this, a multimedia content management system was designed to remotely control the projection through the Robotat network, integrating protocols such as UDP and SRT for efficient video transmission and providing support for tools like OBS and GStreamer.

An application was also developed to interpret position and orientation data transmitted via MQTT, translating them into screen coordinates and modifying the projection in real time. This infrastructure enabled precise interactions based on passive markers, including the control of a digital cursor and the development of an Air Hockey game with simultaneous detection of multiple markers. The system demonstrated reduced latencies between 20 and 50 ms, continuous stability during disconnections, and dynamic layouts in a stream mosaic that organizes multiple audiovisual sources with low latency.

The final validation confirmed the effective synchronization between motion capture and projection, demonstrating the feasibility of the system for multi-user experiences. Overall, the project establishes a robust platform for augmented reality applications, interactive simulations, technological experimentation, and immersive educational environments.

Keywords: augmented reality, motion capture, interactive projection, Robotat, stream mosaic.

CAPÍTULO 1

Introducción

La interacción entre seres humanos y entornos digitales ha evolucionado significativamente en las últimas décadas, impulsada por avances en tecnologías de captura de movimiento, procesamiento en tiempo real y visualización interactiva. Los entornos que integran estos elementos permiten experiencias inmersivas, donde los usuarios no solo observan información, sino que pueden influir activamente en su representación y comportamiento. Estas capacidades tienen aplicaciones en campos diversos, como la educación, la simulación industrial, el arte interactivo y el desarrollo de interfaces humano-máquina [1].

El presente trabajo se centró en el desarrollo de un sistema que integra la captura de movimiento mediante marcadores pasivos del entorno Robotat en la Universidad del Valle de Guatemala, con la proyección visual interactiva en tiempo real. La infraestructura existente en el Robotat proporciona mediciones de alta precisión de la posición y orientación de cuerpos rígidos, pero carece de un software que permita sincronizar estos datos con la visualización proyectada. Esta limitación reduce la posibilidad de crear aplicaciones avanzadas en las que los usuarios puedan interactuar directamente con elementos digitales y con el espacio proyectado.

Para abordar esta necesidad, el proyecto se organizó en tres etapas principales: en primer lugar, la creación de un mosaico de transmisiones de video que permita recibir y visualizar múltiples transmisiones en tiempo real; en segundo lugar, el desarrollo de una aplicación capaz de mapear los datos de captura de movimiento a acciones digitales, incluyendo el control de un cursor y la simulación de clics; y finalmente, la integración de ambos sistemas mediante una aplicación de validación, implementada como un juego de *Air Hockey* interactivo que refleja los movimientos del usuario y actualiza la proyección en tiempo real.

Los resultados obtenidos demostraron la efectividad del sistema implementado. La construcción del mosaico de streams permitió recibir y organizar múltiples transmisiones con baja latencia y tolerancia a desconexiones, estableciendo una base sólida para la interacción inmersiva sobre la mesa Robotat. Asimismo, la aplicación de control de cursor confirmó la precisión en la traducción de un marker físico a coordenadas digitales, asegurando una sincronización correcta entre captura y proyección.

Finalmente, la validación mediante el juego de *Air Hockey* evidenció la capacidad del sistema para detectar y responder a múltiples markers de manera simultánea, corroborando su viabilidad para aplicaciones multiusuario, experimentación tecnológica y entornos educativos.

Con este enfoque, se pretende no solo habilitar nuevas formas de interacción dentro del Robotat, sino también sentar las bases para futuras investigaciones en entornos de realidad aumentada, interfaces inmersivas y aplicaciones educativas o de entretenimiento, contribuyendo al desarrollo de tecnologías centradas en el usuario y a la optimización de recursos en procesos experimentales.

CAPÍTULO 2

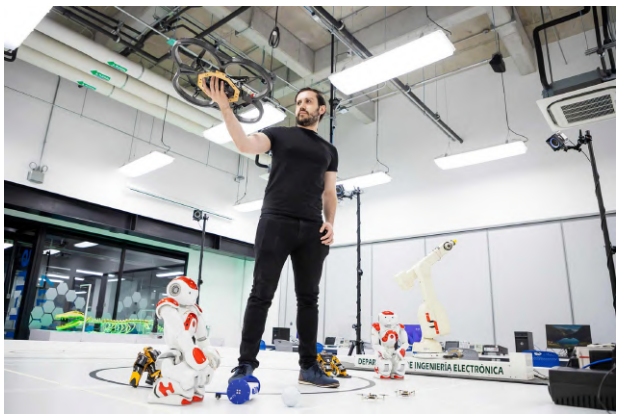
Antecedentes

La automatización de procesos, impulsada por los avances tecnológicos de los últimos años, ha facilitado a diversas industrias el desarrollo de aplicaciones capaces de interpretar el estado actual tanto del mundo físico como del virtual. Esto con el fin de superponer información digital sobre el entorno real y así mejorar la percepción que se tiene de este. De esta forma, el usuario experimenta el mundo virtual como una extensión de su entorno físico; dicha interacción se conoce como realidad aumentada [2].

2.1. Realidad aumentada en la Universidad del Valle de Guatemala

En la actualidad, la Universidad del Valle de Guatemala cuenta con un entorno de desarrollo para aplicaciones de robótica llamado *Robotat* (Figura 1). Este cuenta con una red de comunicación WiFi multiagente que utiliza el protocolo MQTT y opera en paralelo a un sistema de captura de movimiento OptiTrack para obtener la pose efectiva de los agentes presentes [3]. El método de captura de movimiento que se utiliza en el Robotat es el óptico de marcadores pasivos. Dicho método emplea múltiples cámaras infrarrojas fijas de alta velocidad, ubicadas alrededor de una plataforma de captura; al menos dos cámaras registran la posición de cada marcador, y el postprocesamiento genera la pose tridimensional del agente. Este enfoque presenta ciertas complicaciones, tales como la oclusión de marcadores por pérdida de visibilidad y la alta latencia en el procesamiento cuando se recurre a marcadores redundantes. Sin embargo, sus ventajas como la portabilidad del equipo para el usuario, la precisión milimétrica en condiciones óptimas y la baja latencia con una configuración adecuada, superan ampliamente las limitaciones descritas [4].

Figura 1. Entorno dinámico de desarrollo Robotat



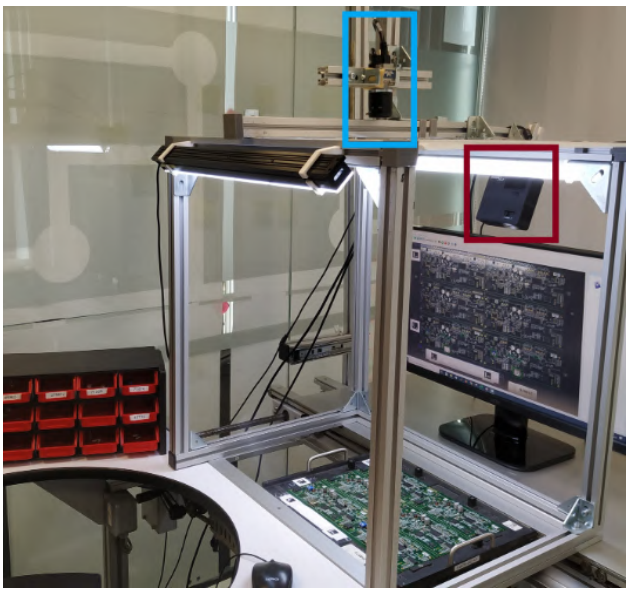
Nota. La figura muestra el entorno dinámico de desarrollo Robotat. [5].

Por otro lado, en trabajos recientes desarrollados dentro del entorno Robotat, se ha observado una tendencia creciente al uso de materiales físicos complementarios con el fin de enriquecer la experiencia de las aplicaciones robóticas y aumentar la sensación de realidad en las pruebas. Estas implementaciones no solo han permitido validar algoritmos de forma más realista, sino que también han demostrado el potencial de integrar componentes físicos o digitales como parte de la experimentación. Por ejemplo, Valdés [6], en su trabajo *Pruebas a escala de algoritmos básicos de visión de computadora y control para vehículos autónomos a escala*, tuvo que fabricar una pista de prueba para validar algoritmos de seguimiento de línea. Por otro lado, Garrido [7], en su trabajo *Simulación y planificación de rutas para vehículos autónomos en entornos urbanos a escala inspirados en la Ciudad de Guatemala*, recurrió a la impresión de una carretera sobre una manta para validar el control de vehículos autónomos usando robots Pololu 3pi+.

2.2. Realidad aumentada en procesos de manufactura y servicios

Actualmente, con el objetivo de reducir el tiempo de producción y mejorar la postura de sus operarios, diversas empresas están optando por utilizar asistencia guiada proyectada en su entorno de trabajo. En el ámbito de la manufactura, se emplea el aprendizaje automático para segmentar el entorno de trabajo y las piezas a ensamblar, generando imágenes de muestra que permiten entrenar al sistema para identificar el tipo y la ubicación de los elementos necesarios. Después, se realiza un mapeo de proyección para visualizar en tiempo real las áreas donde el operario debe realizar su labor, según la fase de ensamblaje correspondiente [8]. En el sistema presentado en la Figura 2, la cámara superior identifica el estado actual de la placa, mientras que el proyector, situado sobre el monitor, proyecta las áreas de interés sobre la placa que está ubicada en la parte inferior del sistema. Un enfoque similar se aplica en la industria de la salud, donde se han desarrollado metodologías de realidad aumentada en neurocirugías para proyectar marcadores de asistencia sobre la cabeza, cráneo o cerebro de los pacientes. Al igual que en el caso anterior, se utiliza aprendizaje automático para analizar imágenes clínicas y generar regiones de interés para los cirujanos [9].

Figura 2. Sistema de realidad aumentada para ensamblaje manual



Nota. La figura muestra un sistema de realidad aumentada para el ensamblaje manual de componentes. La cámara y el proyector se encuentran resaltados en celeste y rojo oscuro, respectivamente. [8].

2.3. Realidad aumentada en la industria del entretenimiento

En el ámbito del entretenimiento, la realidad aumentada se utiliza para proyectar imágenes o videos sobre objetos dinámicos en tiempo real. Esta tecnología resulta muy útil en espectáculos de teatro, musicales o bailes, ya que facilita la modificación del entorno y la adaptación de los trajes de los actores, haciendo más efectiva la expresión de escenas específicas. Para lograrlo, se han implementado técnicas de enmascaramiento basadas en aprendizaje automático que generan siluetas en tiempo real, lo que permite adaptar las imágenes proyectadas a la superficie deseada, como se observa en la la Figura 3. Es importante mencionar que, en aplicaciones dinámicas pueden ocurrir errores de proyección si los movimientos son demasiado rápidos, como al correr, saltar o mover los brazos rápidamente, ya que esto incrementa la latencia del sistema [10].

2.4. Realidad aumentada en la industria tecnológica

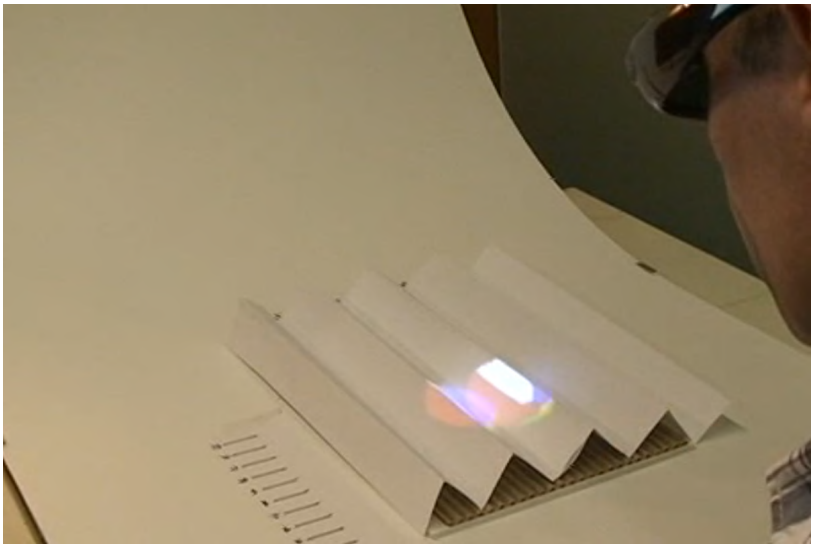
Al igual que en otras industrias, en el ámbito tecnológico se ha integrado la proyección de imágenes y vídeos al rastreo de movimientos mediante sensores infrarrojos y visión por computadora, con el fin de crear aplicaciones inmersivas de realidad mixta. Un ejemplo de ello se muestra en la Figura 4, donde se presenta una mesa interactiva, un dispositivo en forma de mesa que permite la interacción directa con la información proyectada o mostrada sobre su superficie. Para lograr esta interacción, se utilizan cámaras que capturan los movimientos de los ojos y los gestos de las manos del usuario.

Posteriormente, a través de aprendizaje automático, se determina cómo reaccionará la proyección ante los estímulos del usuario [11]. Por otra parte, se han desarrollado métodos de anotación y dibujo en 3D basados en la gesticulación de las manos, nuevamente utilizando cámaras y aprendizaje automático para reconocer patrones y adaptar la reacción al usuario [12]. Estas aplicaciones son especialmente útiles en áreas como el modelado 3D, teleconferencias interactivas, videojuegos inmersivos, entre otras.

Figura 3. Realidad aumentada basada en proyección aplicada a las artes escénicas, observada desde tres perspectivas diferentes (lateral izquierda, central y lateral derecha, respectivamente)

Nota. Realidad aumentada basada en proyección aplicada a las artes escénicas, observada desde tres perspectivas diferentes (lateral izquierda, central y lateral derecha, respectivamente).

Figura 4. Mesa interactiva de realidad aumentada



Nota. La figura muestra una mesa interactiva de realidad aumentada en la que la imagen proyectada puede modificarse en función de la manipulación del entorno físico (cartón plegado). Imagen extraída del video suplementario del artículo de [11].

CAPÍTULO 3

Justificación

Los entornos interactivos que integran captura de movimiento y proyección visual han cobrado relevancia en diversos ámbitos como el arte, la educación, la simulación y el desarrollo de tecnologías centradas en el usuario. Esto se debe a que permiten una interacción más intuitiva, inmersiva y dinámica entre las personas y los espacios digitales. En la práctica, esta interacción suele lograrse mediante cámaras de visión por computadora o sensores infrarrojos que registran las acciones del usuario para luego modificar la proyección en función de estas. Esta metodología presenta algunas limitaciones, especialmente en lo que respecta a la alta latencia y la baja precisión que pueden generar ciertos sistemas.

El entorno Robotat, al ser un espacio de experimentación tecnológica, permite obtener la pose de los cuerpos rígidos presentes mediante el uso de marcadores pasivos con una precisión extremadamente alta. No obstante, su potencial interactivo se ve limitado por la falta de una infraestructura de software que sincronice los datos de movimiento del usuario (o los marcadores) con una interfaz gráfica de visualización. Esta carencia restringe el desarrollo de aplicaciones más avanzadas, en las que es necesario interactuar no solo con los agentes, sino con el entorno mismo. La evidencia de esta limitación se encuentra en los trabajos de graduación previos del Departamento de Ingeniería Mecatrónica de la Universidad del Valle de Guatemala, mencionados en los Antecedentes. En ambos casos, la producción de material físico implicó una inversión significativa de tiempo y recursos que podría haberse evitado con un sistema de proyección interactivo.

En este contexto, el presente trabajo propondrá el desarrollo e implementación de un sistema que integrará la captura de movimiento basada en marcadores pasivos del Robotat con proyección gráfica en tiempo real. A través de algoritmos de mapeo espacial y del análisis de su precisión, se busca habilitar nuevas formas de interacción, que puedan ser aprovechadas en aplicaciones de desarrollo más complejas. Finalmente, se espera que la validación del sistema en escenarios aplicados abra la puerta a futuras investigaciones o a mejoras del sistema de realidad aumentada que se pretende incorporar al entorno Robotat.

CAPÍTULO 4

Objetivos

4.1. Objetivo general

Desarrollar e implementar la infraestructura digital de un entorno interactivo que sincronice la información del movimiento del usuario capturada con el sistema OptiTrack con la vizualización proyectada en el ambiente Robotat.

4.2. Objetivos específicos

- Desarrollar e implementar un sistema de gestión de contenido multimedia que permita controlar de forma remota la proyección visual en el entorno Robotat mediante su red y estructura de proyección.
- Desarrollar una aplicación que utilice los datos del sistema de captura de movimiento para modificar en tiempo real la visualización proyectada en el entorno Robotat.
- Validar la infraestructura desarrollada mediante el desarrollo de una aplicación de realidad aumentada que demuestre la sincronización efectiva entre la captura de movimiento y la proyección visual.

CAPÍTULO 5

Definición del problema y alcance

5.1. Definición del problema

A pesar del creciente desarrollo de tecnologías de realidad aumentada, captura de movimiento y visualización interactiva, muchos entornos de investigación y educativos enfrentan limitaciones importantes para la implementación de experiencias inmersivas en tiempo real. En particular, el laboratorio Robotat de la Universidad del Valle de Guatemala cuenta con un sistema de captura de movimiento de alta precisión (*OptiTrack*), capaz de medir la posición y orientación de marcadores pasivos, pero carece de una infraestructura digital que permita sincronizar estos datos con proyecciones visuales interactivas distribuidas en el espacio. Esta carencia dificulta:

- La creación de aplicaciones interactivas multiusuario que reflejen en tiempo real los movimientos físicos de los participantes.
- La integración de flujos de video provenientes de múltiples fuentes en un mosaico de proyección estable y de baja latencia.
- La implementación de sistemas de control digital basados en la posición de marcadores, que requieran precisión y respuesta inmediata.
- La tolerancia a interrupciones en las transmisiones de datos, que puede afectar la continuidad y confiabilidad de las experiencias inmersivas.

En consecuencia, el problema central se resume en la ausencia de una plataforma capaz de recibir, procesar y proyectar datos de captura de movimiento en tiempo real, coordinando múltiples transmisiones audiovisuales con baja latencia, precisión en la interacción y robustez ante desconexiones o errores en el flujo de información.

5.2. Alcance

El presente proyecto aborda estas limitaciones mediante el desarrollo de un sistema integral que combina transmisión de video, procesamiento de datos de captura de movimiento y visualización interactiva. El alcance incluye:

- Diseño e implementación de un mosaico de transmisiones de video utilizando protocolos de baja latencia (UDP, SRT), compatible con herramientas como OBS y GStreamer, capaz de recibir y organizar múltiples fuentes en tiempo real.
- Desarrollo de un módulo de interpretación de datos de marcadores pasivos transmitidos mediante MQTT, con transformación de coordenadas a pantalla y sincronización con la proyección.
- Integración de interacciones basadas en posición de marcadores, incluyendo control de cursor digital y la simulación de clics, con respuesta en tiempo real y latencias mínimas.
- Implementación de un juego de Air Hockey como caso de prueba, permitiendo la detección simultánea de múltiples jugadores y validando la capacidad del sistema para experiencias multiusuario.
- Aseguramiento de robustez frente a desconexiones temporales de los flujos de video o de la captura de movimiento, manteniendo la continuidad de la experiencia y la estabilidad del mosaico proyectado.

El proyecto no incluye el desarrollo de nuevas técnicas de captura de movimiento ni hardware adicional; se centra exclusivamente en la infraestructura de software y la integración de sistemas existentes para habilitar interacciones inmersivas y experimentación educativa dentro del entorno Robotat.

CAPÍTULO 6

Marco teórico

6.1. Protocolos de comunicación y transporte de datos

6.1.1. MQTT: Mensajería ligera para sistemas distribuidos

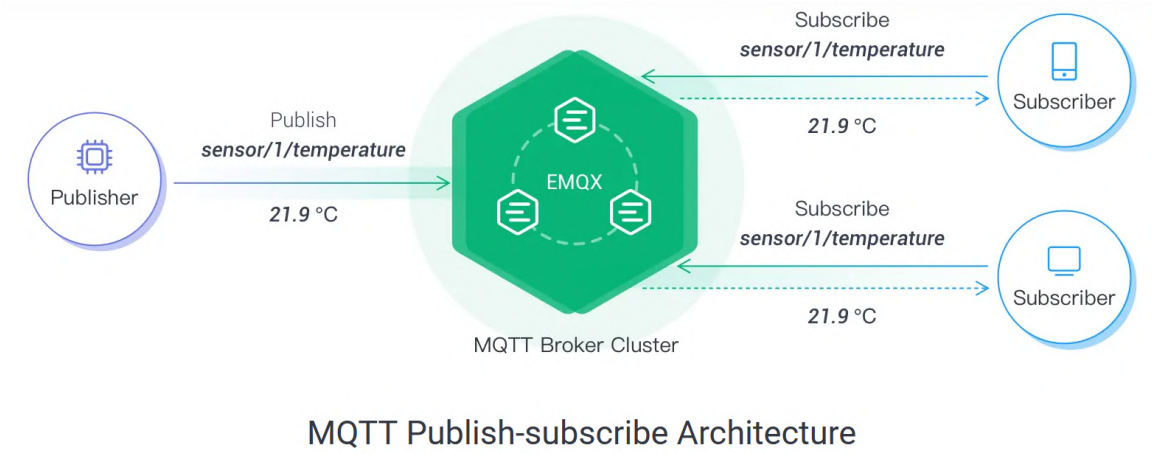
MQTT (Message Queuing Telemetry Transport) es un protocolo de mensajería ligero diseñado para la comunicación eficiente entre dispositivos con recursos limitados o conectados mediante redes inestables. Su funcionamiento se basa en un modelo de publicación y suscripción (*publish/subscribe*), donde los clientes no se comunican directamente entre sí, sino a través de un *broker* central encargado de recibir, almacenar temporalmente y distribuir los mensajes como se muestra en la Figura 5.

A diferencia de los modelos tradicionales de comunicación punto a punto, la arquitectura desacoplada de MQTT permite una baja latencia, alta tolerancia a fallos y una implementación sencilla en sistemas distribuidos. Debido a esto, se ha convertido en un estándar dentro del Internet de las Cosas (IoT), la telemetría industrial y aplicaciones donde se requiere transmitir información en tiempo real con un ancho de banda limitado [13].

Estructura de *topics* y direccionamiento

MQTT enruta los mensajes utilizando *topics*, los cuales funcionan como rutas jerárquicas similares a los caminos de una URL. Un *topic* puede estar compuesto por varios niveles separados por diagonales, por ejemplo: `sensor/1/temperature`. Múltiples publicadores pueden enviar mensajes al mismo *topic*, y el broker los reenviará en el orden en que los recibe. De igual manera, varios suscriptores pueden suscribirse al mismo *topic*, y todos recibirán las publicaciones correspondientes. Esta estructura flexible permite la construcción de sistemas escalables donde nuevas fuentes y receptores pueden añadirse sin reconfigurar los nodos existentes [13].

Figura 5. Esquema de comunicación basado en el patrón publicador–broker–suscriptor de MQTT



Nota. El *broker* central recibe los mensajes enviados por los publicadores y los reenvía únicamente a los suscriptores interesados en el *topic* correspondiente (sensor/1/temperature en el diagrama) [13].

Uso de comodines en *topics*

Una característica fundamental de MQTT es el uso de comodines o *wildcards*, que permiten que un cliente se suscriba a múltiples *topics* usando una sola instrucción. El uso de comodines permite que un solo cliente monitoree grandes grupos de dispositivos sin requerir suscripciones individuales. [13] Existen dos tipos de comodines:

- '+' (comodín de un solo nivel): coincide con exactamente un nivel de la jerarquía. Ejemplo: `sensor/+/temperature` recibirá mensajes de `sensor/1,2.../temperature`.
- '#' (comodín multinivel): coincide con todos los niveles subsiguientes de la jerarquía. Ejemplo: `sensor/#` recibirá cualquier mensaje dentro del árbol `sensor/`.

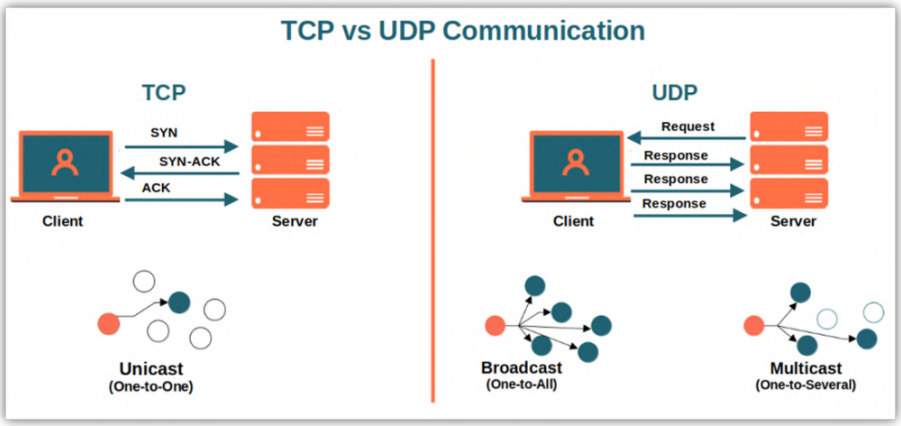
En este proyecto, MQTT se utilizó para enviar datos de posición y orientación generados por el sistema OptiTrack hacia la aplicación responsable de procesarlos y actualizar la visualización en tiempo real sobre la plataforma interactiva. Así pues, esta comunicación permitió mantener un flujo continuo y actualizado sin saturar la red ni comprometer la estabilidad del sistema.

6.1.2. Protocolos UDP y TCP: Fundamentos y diferencias

A nivel de transporte, TCP (Transmission Control Protocol) y UDP (User Datagram Protocol) representan los dos métodos más comunes para enviar datos entre dispositivos. TCP es un protocolo orientado a conexión punto a punto, véase Figura 6. Antes de transmitir datos, establece un enlace entre emisor y receptor, garantizando entrega ordenada y confiable de los paquetes mediante mecanismos de verificación y retransmisión. Por ello es adecuado para aplicaciones donde la integridad es esencial [14].

Por otro lado, UDP es un protocolo no orientado a conexión. No garantiza entrega, orden ni corrección, pero ofrece menor latencia debido a su simplicidad. Se utiliza en aplicaciones donde la velocidad es prioritaria, como streaming, juegos y sistemas de captura de movimiento. En este proyecto se favoreció UDP para las transmisiones de video y datos en tiempo real, debido a la necesidad de reducir la latencia en la interacción.

Figura 6. Comparación entre los métodos de comunicación de TCP y UDP



Nota. Diferencias de comunicación entre TCP y UDP en cuanto a la forma en que se transmiten los datos y el tipo de conexión que cada protocolo establece, si existe alguna conexión [14].

6.1.3. Protocolos de streaming: SRT y RTMP

RTMP: Real-Time Messaging Protocol

RTMP, creado originalmente por Macromedia, opera sobre TCP y requiere una conexión persistente. Ofrece estabilidad, pero con mayor latencia debido al uso de un canal confiable. Al transmitir paquetes de audio y video, cuando los datos no llegan en secuencia o se pierden debido a la naturaleza inherentemente inestable de las redes, TCP se encarga de reordenar los paquetes y solicitar retransmisiones para compensar la pérdida. Este enfoque exige una entrega perfecta y en orden, lo cual puede afectar negativamente la transmisión en tiempo real de flujos de audio y video [15].

SRT: Secure Reliable Transport

SRT es un protocolo moderno que opera sobre UDP y permite transmisión de video con baja latencia en redes inestables. Incorpora corrección de errores, retransmisión selectiva, cifrado y manejo dinámico de retardo entre paquetes. Es ampliamente usado en aplicaciones profesionales de video en tiempo real [15]. Por esta razón, en el proyecto se utilizó SRT para transmitir múltiples streams hacia el mosaico de proyección, manteniendo estabilidad incluso ante fluctuaciones en la red.

6.2. Direcciones IP y redes privadas

6.2.1. Definición de dirección IP

Una dirección IP identifica de manera única a un dispositivo dentro de una red y permite establecer rutas de comunicación entre equipos. Es un componente esencial del Protocolo de Internet (IP), mediante el cual routers y dispositivos determinan el origen y el destino de cada paquete. Aunque para los usuarios finales las direcciones IP suelen estar ocultas gracias a servicios como DNS, su administración y funcionamiento constituyen la base del intercambio de información en redes locales y en Internet [16].

Cada dispositivo conectado a Internet, o a una red privada basada en TCP/IP, debe poseer al menos una dirección IP. Estas direcciones se asignan y administran de forma jerárquica para garantizar escalabilidad en el enrutamiento y evitar duplicaciones. Asimismo, existen direcciones con propósitos especiales (como broadcast, multicast y anycast) que permiten optimizar la distribución del tráfico según el tipo de servicio. Cuando un dispositivo se conecta a Internet global, su dirección debe ser única y estar coordinada con organismos internacionales de asignación.

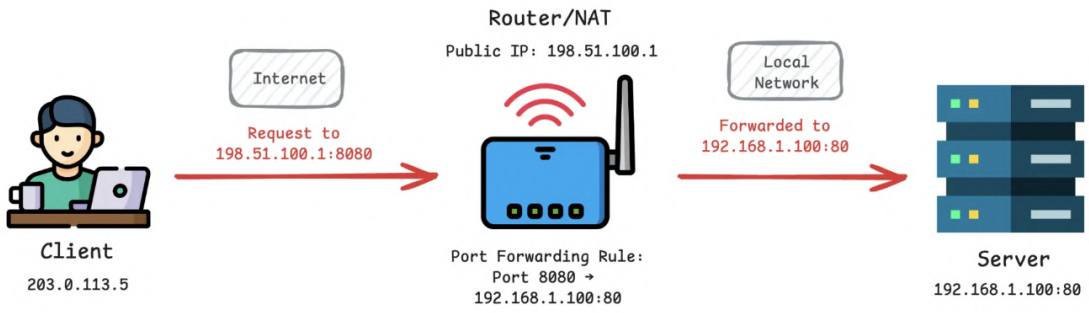
6.2.2. Network Address Translation (NAT)

El mecanismo de *Network Address Translation (NAT)* permite reutilizar los mismos conjuntos de direcciones IP privadas en distintas redes y fue concebido para enfrentar dos problemas principales: el agotamiento del espacio de direcciones IPv4 y la necesidad de mejorar la escalabilidad del enrutamiento. Un caso típico de uso ocurre cuando una red completa se conecta a Internet mediante una sola dirección IP pública. El dispositivo NAT (generalmente un router) recibe todo el tráfico entrante y saliente, y se encarga de traducir entre las direcciones privadas internas y la dirección externa pública. Esto posibilita que múltiples dispositivos internos accedan a Internet aun cuando solo se disponga de una o pocas direcciones globales [16].

6.2.3. Port Forwarding o mapeo de puertos

Una de las principales desventajas del NAT es que los dispositivos internos con direcciones privadas no son accesibles directamente desde Internet. Cuando un servidor ubicado dentro de la red privada necesita ofrecer un servicio hacia afuera, se requiere un mecanismo adicional: el *port forwarding* o *mapeo de puertos*. Este procedimiento permite que el NAT redirija tráfico entrante dirigido a un puerto específico de la dirección pública hacia un dispositivo interno particular como se observa en la Figura 7. De esta forma, un servidor privado puede ser visible desde Internet pese a utilizar una dirección no enrutada. El proceso suele requerir configuración estática, indicando la IP interna del servidor y el puerto correspondiente. Si varios servidores necesitan exponer el mismo puerto, se requieren múltiples direcciones públicas o configuraciones alternativas más avanzadas [16]. En este proyecto, la red Robotat carecía de esta regla en su router y por esta razón las pruebas del servidor de proyección con OBS studio se realizaron con la red de la Universidad del Valle de Guatemala.

Figura 7. Concepto de port forwarding



Nota. `1.` El cliente envía la solicitud a la IP pública (198.51.100.1) en el puerto 8080 `2.` El router recibe la solicitud y comprueba las reglas de reenvío de puertos (port forwarding) `3.` El router reenvía la solicitud al servidor interno (192.168.1.100) en el puerto 80 `4.` El servidor procesa la solicitud y envía la respuesta de vuelta a través de la misma ruta. [17].

6.3. OptiTrack y sistemas de captura de movimiento

6.3.1. Descripción general de OptiTrack

OptiTrack es un sistema de captura de movimiento basado en cámaras infrarrojas de alta velocidad que detectan marcadores pasivos reflectivos. El sistema utiliza múltiples cámaras sincronizadas, dispuestas alrededor del volumen de captura, para obtener diferentes vistas simultáneas del mismo conjunto de marcadores. Esta configuración permite reconstrucciones tridimensionales precisas incluso en aplicaciones que requieren baja latencia, como robótica, biomecánica o entornos interactivos [18].

Un aspecto fundamental del diseño de OptiTrack es la disposición de las cámaras. La calidad de la captura depende directamente de que los marcadores sean observados desde varios ángulos. Colocar las cámaras en posiciones estratégicas alrededor del espacio de captura reduce las oclusiones, amplía la cobertura visual y mejora los ángulos requeridos por el algoritmo de triangulación. Una planificación adecuada (incluyendo la elaboración previa de un plano de distribución) asegura que los marcadores permanezcan visibles para al menos dos cámaras en todo momento, condición necesaria para reconstruir coordenadas 3D sin interrupciones [18].

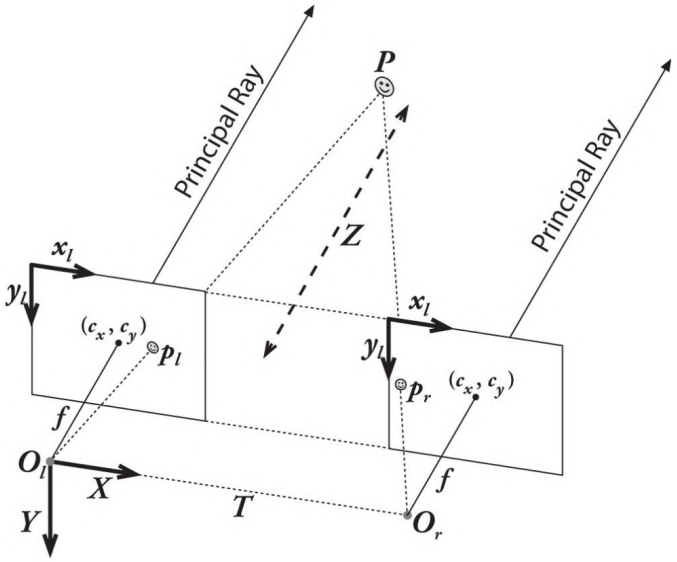
Las cámaras OptiTrack rastrean cualquier superficie cubierta con material retrorreflectante, el cual está diseñado para reflejar la luz entrante de vuelta hacia su fuente. La luz infrarroja emitida por la cámara es reflejada por los marcadores pasivos y detectada por el sensor de la cámara. Luego, las reflexiones capturadas se utilizan para calcular la posición 2D del marcador, la cual es usada por el programa *Motive* para reconstruir y calcular su posición 3D [19].

6.3.2. Funcionamiento de la captura de movimiento

1. Las cámaras emiten luz infrarroja hacia la escena.
2. Los marcadores reflectivos devuelven la luz hacia los sensores de cada cámara.
3. El software identifica los puntos brillantes en cada imagen 2D capturada.
4. Para cada marcador, el sistema correlaciona sus posiciones 2D entre cámaras.
5. Mediante triangulación se determina la posición tridimensional de cada marcador.
6. Los datos reconstruidos se envían a la red Robotat como secuencias de posición, orientación y estructuras superiores (rigid bodies, markersets, skeletons).

El diagrama de la Figura 8 muestra un modelo estereoscópico rectificado similar al utilizado por OpenCV para sistemas estéreo clásicos. Aunque este diagrama sirve para ilustrar el principio geométrico de la triangulación (la intersección de rayos proyectados desde distintas cámaras), es importante aclarar que OptiTrack no opera como un sistema estéreo rectificado tradicional [20]. En los sistemas de captura de movimiento OptiTrack: Cada cámara posee su propia matriz intrínseca calibrada de forma independiente, las cámaras no están rectificadas ni organizadas en pares; por el contrario, se encuentran distribuidas en posiciones y orientaciones arbitrarias dentro del volumen de captura y la reconstrucción tridimensional no se basa en pares de cámaras, sino en una triangulación multivista, utilizando simultáneamente todas las cámaras que observan un marcador [18].

Figura 8. Ejemplo ilustrativo del principio geométrico utilizado para la triangulación 3D de un marcador



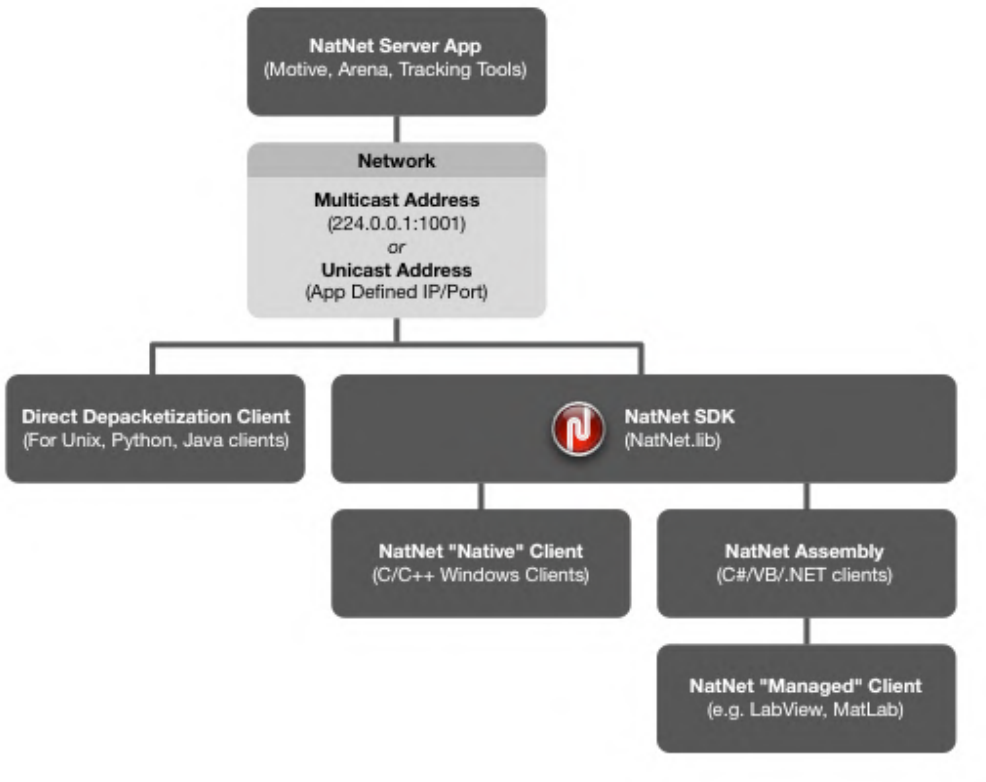
Nota. El diagrama se basa en un modelo estéreo rectificado al estilo OpenCV. Aunque OptiTrack no emplea rectificación ni pares estéreo, el principio geométrico fundamental (intersección de rayos proyectados desde cámaras distintas) es equivalente [20].

6.3.3. Transmisión de datos en tiempo real mediante NatNet

Una vez reconstruidas las posiciones tridimensionales, OptiTrack permite transmitirlas en tiempo real hacia aplicaciones externas mediante el protocolo *NatNet*. El cual es un sistema cliente/servidor optimizado para baja latencia que utiliza comunicación UDP, tanto en modalidades *unicast* como *multicast*, para enviar datos como: coordenadas 3D de marcadores individuales, datos de *rigid bodies*, *markersets* entrenados, esqueletos y articulaciones.

El SDK de NatNet permite crear clientes personalizados capaces de recibir y procesar los datos tridimensionales generados por OptiTrack. En el caso de este proyecto, el servidor OptiTrack envía los paquetes UDP mediante NatNet hacia el broker de la red Robotat, el cual actúa como intermediario para la distribución de la información. Este broker recibe la pose reconstruida de los marcadores y la publica mediante MQTT para que los distintos clientes de la red puedan suscribirse y extraer las posiciones y orientaciones en tiempo real. La Figura 9 muestra el flujo general: el servidor OptiTrack transmite los datos NatNet a la red del entorno Robotat; posteriormente, los clientes MQTT (incluida la aplicación desarrollada en este proyecto) recuperan la información publicada y la transforman al sistema de coordenadas requerido por la pantalla interactiva [21].

Figura 9. Esquema general de comunicación entre el servidor OptiTrack y un cliente NatNet personalizado



Nota. En este proyecto, el cliente en Python implementa *direct depacketization* por medio de MQTT para recibir los paquetes publicados en la red Robotat, los cuales contienen la pose de marcadores enviada previamente por el servidor NatNet [21].

6.4. GStreamer y el concepto de *pipeline* multimedia

6.4.1. GStreamer como framework multimedia

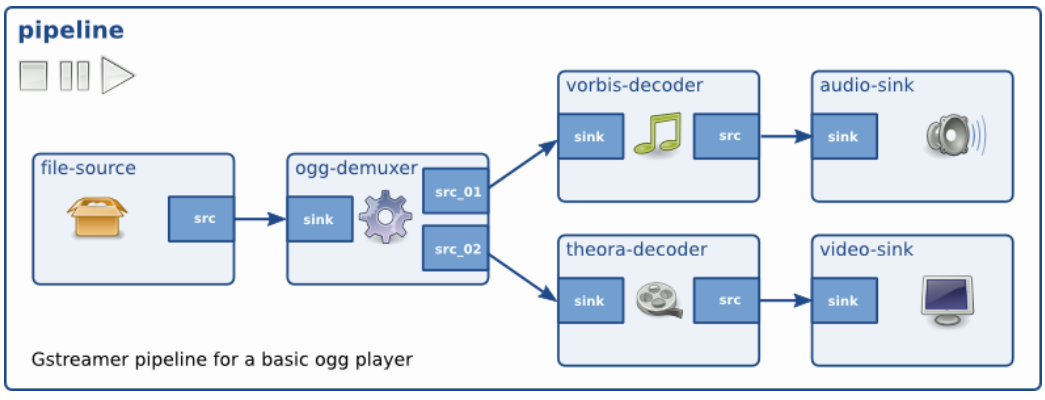
GStreamer es un framework modular diseñado para construir aplicaciones de audio y video mediante componentes interconectados. Debido a su arquitectura flexible, basada en elementos enlazables y compatible con una amplia diversidad de codecs y protocolos, GStreamer se utiliza tanto en reproductores domésticos como en sistemas profesionales de transmisión, procesamiento en tiempo real y visión computacional [22].

6.4.2. *Pipelines*: Flujo estructurado de elementos

Un *pipeline* en GStreamer es la estructura que define el flujo de datos multimedia a través de una cadena organizada de elementos que capturan, procesan, codifican, decodifican o despliegan contenido audiovisual. Internamente, GStreamer organiza estos elementos dentro de contenedores denominados *bins*. Los *bins* funcionan como agrupaciones lógicas de elementos y, al ser también subclases de elementos, permiten controlar múltiples componentes simultáneamente. Esto significa que cambiar el estado de un *bin* (por ejemplo, a *PAUSED* o *PLAYING*) cambia automáticamente el estado de todos sus elementos internos. Además, los *bins* reenvían los mensajes del *bus*, tales como errores, fin de flujo (EOS) o metadatos, simplificando la gestión de eventos [22].

El *pipeline* es un *bin* de nivel superior. Proporciona un *bus* para que la aplicación reciba los mensajes relevantes, gestiona la sincronización entre sus elementos y administra el flujo de datos durante la ejecución. Una vez que un *pipeline* entra al estado *PLAYING*, el procesamiento multimedia comienza en un hilo separado como se observa en la Figura 10. En el mosaico desarrollado para este proyecto, los *pipelines* permitieron recibir múltiples flujos de video, decodificarlos, detectar su disponibilidad, manejar desconexiones y organizarlos visualmente dentro de la interfaz construida con GTK.

Figura 10. Ejemplo general de un *pipeline* en GStreamer para lectura, decodificación y reproducción de audio y video



Nota. El archivo fuente se demultiplexa en audio y video, cada flujo se decodifica por separado y ambos se envían a sus respectivos sumideros para reproducción [22].

6.5. Interfaz gráfica con GTK

GTK organiza su interfaz gráfica a partir de *widgets*, todos derivados de la clase base `GtkWidget`, la cual gestiona su ciclo de vida, estado y estilo. Las interfaces se construyen anidando widgets dentro de otros widgets, formando una estructura jerárquica. Por ejemplo, una ventana `GtkWindow` puede contener un `GtkFrame`, y dentro de este un `GtkLabel` o un `GtkImage` [23].

Los widgets contenedores (*containers*) permiten agrupar y organizar otros elementos. Existen dos tipos principales de contenedores:

- **Contenedores con un solo hijo** (derivados de `GtkBin`): funcionan como decoradores que añaden funcionalidad a su único hijo. Ejemplos incluyen `GtkButton`, que vuelve clicable el widget contenido; `GtkFrame`, que dibuja un marco alrededor del hijo; y `GtkWindow`, que muestra su hijo en una ventana principal.
- **Contenedores con múltiples hijos**: su propósito es administrar el diseño y la disposición. Por ejemplo, `GtkHBox` organiza los hijos en una fila horizontal, mientras que `GtkGrid` define una cuadrícula bidimensional para ubicar widgets.

Para implementar un contenedor personalizado, es necesario definir el método virtual `GtkContainerClass.forall()`, utilizado para el dibujo y operaciones internas. Si el contenedor maneja hijos externos, deben implementarse también los métodos `add()` y `remove()`. En el caso de hijos internos, estos se añaden mediante `gtk_widget_set_parent()` durante la inicialización y se liberan con `gtk_widget_unparent()` en `GtkWidgetClass.destroy()` [23].

Finalmente, en el mosaico de proyección realizado cada stream fue representado como un widget asignado a un contenedor principal. Dado que los espacios del mosaico reciben y reemplazan pipelines dinámicamente, fue necesario gestionar widgets como hijos externos y, en algunos casos, liberarlos o removerlos explícitamente para actualizar el flujo de video al cambiar entre los modos de recepción UDP y SRT.

6.6. Control del mouse mediante ctypes y SendInput

6.6.1. Librería ctypes

Es una biblioteca de funciones foráneas para Python que permite interactuar con código C y C++. Además, permite acceder directamente a las interfaces nativas de los principales sistemas operativos (como `kernel32` en Windows o `libc` en sistemas Unix) y facilita la carga dinámica de bibliotecas compartidas (*DLLs* o *shared objects*) en tiempo de ejecución. Además, proporciona un conjunto completo de tipos compatibles con C y permite definir estructuras complejas, tales como *structs* y *unions*, con control explícito sobre aspectos como alineación y relleno (*padding*) cuando es necesario. Aunque su uso puede resultar algo detallado, en combinación con el módulo `struct`, ofrece un control total sobre la traducción de datos entre Python y código nativo [24].

6.6.2. Funcionamiento de SendInput

`SendInput` es una función de la API de Windows que permite simular eventos de entrada como movimientos de mouse o teclas presionadas. Recibe una lista de estructuras `INPUT`, especificando coordenadas y tipo de evento [25].

En el contexto del control del mouse, estas capacidades resultan esenciales, ya que Python no ofrece de forma nativa funciones de bajo nivel para enviar eventos directos al sistema operativo. Así pues, con `ctypes` es posible invocar la función `SendInput` de la API de Windows, construir las estructuras requeridas por esta interfaz y generar eventos de movimiento, clic y desplazamiento del cursor de manera eficiente y precisa, siguiendo exactamente el formato que espera el sistema a nivel interno.

6.7. Conceptos fundamentales de programación

6.7.1. Clase

Una clase es una estructura fundamental en la programación orientada a objetos utilizada para agrupar datos y comportamientos relacionados. Funciona como una plantilla que define las características que tendrán los objetos creados a partir de ella. Crear una clase implica introducir un nuevo tipo de objeto en el programa, lo que permite instanciar múltiples elementos que comparten estructura y funcionalidad, pero con estados independientes. Este modelo es común en lenguajes como Python, C++, Java, C#, JavaScript (en clases modernas), entre otros [26].

Las clases suelen incluir mecanismos esenciales como:

- Encapsulamiento: control del acceso a atributos y métodos.
- Herencia: capacidad de crear nuevas clases basadas en otras, reutilizando y extendiendo su comportamiento.
- Polimorfismo: posibilidad de redefinir métodos para que funcionen de manera distinta según el tipo de objeto.

6.7.2. Método

Un método es una función asociada a una clase que opera sobre las instancias de esta. A través de los métodos, los objetos pueden modificar su estado interno o ejecutar acciones específicas. En la mayoría de lenguajes orientados a objetos, los métodos reciben implícita o explícitamente una referencia al objeto sobre el cual actúan, lo que permite acceder a sus atributos y a otros métodos. Además, según el lenguaje, pueden existir métodos de instancia, de clase o estáticos, así como métodos sobrescribibles para soportar herencia y polimorfismo [26].

6.8. Pymunk: simulación física y colisiones

6.8.1. Funcionamiento general de Pymunk

Pymunk es un *wrapper* de alto nivel para el motor de física 2D Chipmunk2D. Está diseñado para ser fácil de usar, pero conservando la potencia del motor subyacente. Su función principal es permitir la creación y simulación de cuerpos rígidos, fuerzas, colisiones y restricciones [27].

Uno de los conceptos centrales en su arquitectura es el **Space** (`pymunk.Space`), que actúa como la unidad fundamental de simulación. En un *space* se agregan cuerpos, formas y restricciones, y es este espacio el que gestiona toda la interacción física entre ellos. La simulación avanza en pasos discretos mediante el método `Space.step(dt)`, donde el valor `dt` debe mantenerse constante para promover estabilidad numérica durante la simulación [27].

Los elementos principales del motor son:

- **Bodies**: representan objetos físicos y almacenan propiedades como masa, inercia, fuerza, velocidad y posición.
- **Shapes**: definen la geometría utilizada para detectar colisiones (polígonos, círculos, segmentos). Pueden tener fricción, elasticidad y filtros de colisión.
- **Constraints**: permiten unir cuerpos mediante bisagras, resortes, pivotes u otros tipos de articulaciones.
- **Space**: coordina el conjunto de cuerpos, calcula la física y resuelve colisiones en cada paso de simulación.

Durante la ejecución, el *Space* es el encargado de avanzar la física cuadro a cuadro, calcular fuerzas, resolver colisiones y mantener la coherencia del mundo simulado. Esto permite crear dinámicas complejas con comportamientos realistas en tiempo real [27].

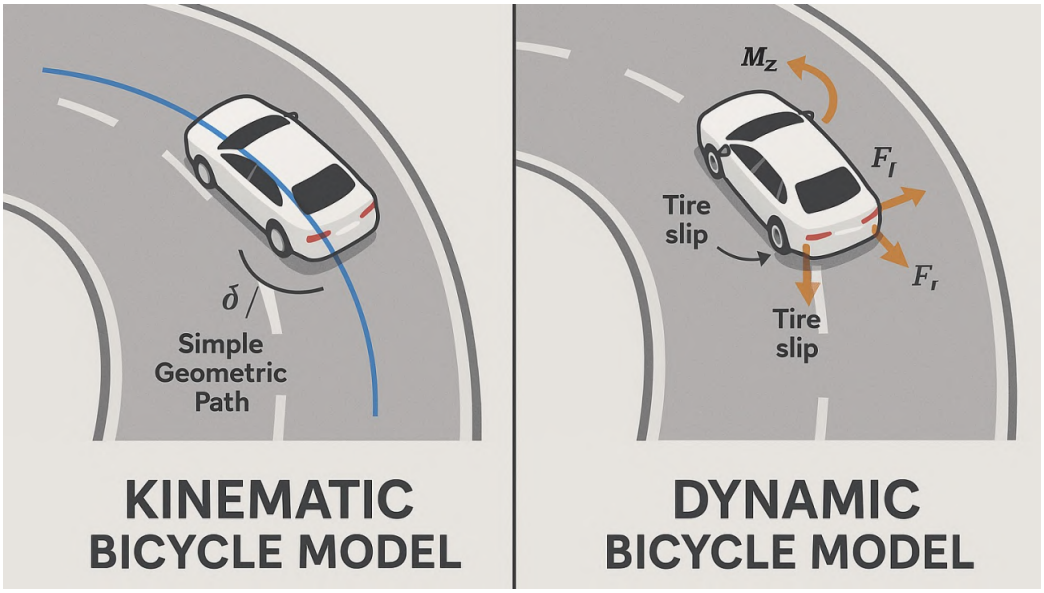
6.8.2. Tipos de cuerpos: Dynamic, Static y Kinematic

Pymunk ofrece tres tipos principales de cuerpos, cada uno con un propósito específico dentro de una simulación física [27]:

- **Dynamic**: son cuerpos completamente simulados por el motor. Responden a fuerzas, gravedad, impulsos y colisiones. Son ideales para objetos móviles como jugadores, discos o proyectiles.
- **Static**: representan cuerpos inmóviles que no reaccionan a fuerzas ni colisiones. Se utilizan como paredes, bordes o suelo. Aunque no se mueven, otros cuerpos dinámicos pueden chocar contra ellos.

- **Kinematic**: tienen movimiento controlado por el usuario mediante la asignación directa de velocidades. No responden a fuerzas externas pero sí pueden generar colisiones con cuerpos dinámicos. Son útiles para plataformas móviles, puertas o elementos guiados.

Figura 11. Comparación entre cuerpos *Kinematic* y *Dynamic* en simulaciones físicas



Nota. Lado izquierdo (Modelo Cinemático): los cuerpos cinemáticos siguen un recorrido predefinido y no reciben fuerzas ni gravedad por parte del motor físico. Lado derecho (Modelo Dinámico): los cuerpos dinámicos responden a fuerzas, colisiones y propiedades físicas como masa y fricción, generando un movimiento realista [28].

6.8.3. Colisiones en Pymunk

En Pymunk, las colisiones se detectan mediante **shapes**, que pueden clasificarse por categorías y máscaras para controlar qué objetos pueden interactuar entre sí. Para ofrecer flexibilidad en la lógica del juego, el motor permite registrar *callbacks* que se ejecutan antes, durante o al finalizar un contacto, posibilitando comportamientos personalizados como rebotes especiales, detección de goles o respuestas sonoras [27].

Un aspecto importante señalado en la documentación es que la forma utilizada para el cálculo físico no necesita coincidir exactamente con la representación visual en pantalla. En la práctica, las formas destinadas a la simulación suelen ser versiones simplificadas del objeto dibujado, lo que optimiza el rendimiento y evita cálculos innecesarios [27].

En el juego de Air Hockey desarrollado, estas colisiones permitieron detectar rebotes, límites del campo, choques controlados por múltiples marcadores y la respuesta dinámica del disco digital dentro del entorno físico.

CAPÍTULO 7

Servidor de proyección y aplicación estilo mosaico de transmisiones

7.1. Instalación de software preliminar y preparación del entorno de trabajo

El desarrollo del sistema inició con la búsqueda de una solución capaz de recibir múltiples transmisiones de video de manera simultánea y posteriormente visualizarlas en un formato de mosaico sobre la mesa interactiva del entorno Robotat. La primera aproximación consistió en implementar un servidor utilizando NGINX con el módulo RTMP, empleando OBS Studio como cliente de transmisión. Esta configuración funcionó correctamente, pero presentó limitaciones importantes en velocidad y latencia, por lo que se evaluaron alternativas con mejor rendimiento [29].

Dado que el servidor de proyección debía ejecutarse sobre Linux, se decidió migrar hacia un servidor basado en el protocolo SRT, caracterizado por su resiliencia ante pérdida de paquetes y su capacidad para trabajar con baja latencia. Entre las opciones disponibles, se seleccionó el proyecto *srt-live-server*, desarrollado por Edward Wu, por su estabilidad, documentación clara y facilidad de compilarse directamente desde su repositorio en GitHub [30].

Un reto adicional surgió debido a que otros estudiantes de la Universidad del Valle de Guatemala necesitaban utilizar simultáneamente el entorno Robotat para sus propios proyectos de graduación. Esto dificultaba contar con acceso permanente al servidor físico para realizar pruebas. Por esta razón, se optó por instalar un subsistema Linux local mediante WSL (Windows Subsystem for Linux), con Ubuntu, para recrear un entorno de trabajo equivalente al del servidor real.

Con el entorno Linux instalado, se compiló el servidor SRT y se configuró OBS Studio con los parámetros requeridos (descritos a detalle en el manual de anexos). Al iniciar

transmisiones desde OBS, el parámetro `size` del servidor SRT incrementaba correctamente, indicando que múltiples transmisiones estaban siendo recibidas de forma simultánea.

El siguiente desafío consistía en visualizar dichas transmisiones. Para ello se evaluaron varias alternativas, concluyendo que GStreamer era la opción ideal debido a:

- Su compatibilidad tanto con Linux como con Windows,
- Su flexibilidad para crear *pipelines* complejos,
- Su integración nativa con múltiples frameworks gráficos,
- Su bajo nivel de abstracción, lo cual permite optimizar la latencia.

Como prueba preliminar se ejecutaron pipelines simples para verificar la recepción de video, por ejemplo:

```
gst-launch-1.0 -v udpsrc port=5000 \
caps="application/x-rtp,media=video,encoding-name=H264,payload=96" \
! rtph264depay ! h264parse ! avdec_h264 ! videoconvert \
! autovideosink sync=false
```

Estas pruebas confirmaron que era posible recibir y decodificar transmisiones desde OBS utilizando GStreamer [22].

Sin embargo, al observar cómo otros estudiantes compartían el espacio físico de trabajo en el Robotat, se identificó la necesidad de permitir que varias transmisiones pudieran visualizarse simultáneamente sobre la mesa. Por tanto, se decidió extender el sistema para soportar hasta seis transmisiones concurrentes, en lugar de limitarse a una sola proyección.

Para lograrlo, se propuso desarrollar una aplicación capaz de recibir, organizar y visualizar múltiples transmisiones en formato mosaico. Dado que el servidor de proyección correría en Linux y GStreamer ofrecía integración completa con GTK+, se seleccionó GTK+ como motor gráfico. GTK ofrecía ventajas adicionales relevantes para este proyecto:

- Amplio soporte en entornos Linux,
- Herramientas estables para manejo de ventanas y contenedores,
- Facilidad para integrar elementos gráficos con *pipelines* multimedia,
- Documentación madura y comunidad activa.

No obstante, GTK requiere entornos de compilación Unix-like para su correcto funcionamiento. Por esta razón fue necesario instalar MSYS2 en Windows para continuar el desarrollo de la aplicación desde la computadora personal.

7.2. Desarrollo inicial de la aplicación estilo mosaico de transmisiones

La aplicación mosaico se desarrolló utilizando C++ como lenguaje base, dado que GTK+ posee mayor soporte para este lenguaje y porque GStreamer también proporciona un API robusto para C/C++. Además, C++ ofrecía ventajas relevantes para un entorno de transmisión en tiempo real, tales como:

- Control de memoria preciso y eficiente,
- Alta velocidad de ejecución,
- Menor latencia al integrar múltiples hilos de procesamiento,
- Capacidad para crear estructuras de datos optimizadas para gestionar varios flujos audiovisuales simultáneos.

Inicialmente se utilizó GTK4 por ser la versión más reciente al momento del desarrollo [31]. La primera versión de la aplicación consistía únicamente en dos archivos: `main.cpp` y `StreamSlot.cpp`. Esta versión inicial generaba los widgets base para cada transmisión, controlaba el diseño general y mostraba divisiones de pantalla predefinidas, además de un fondo negro cuando no había transmisión activa.

El siguiente paso fue integrar GStreamer para crear los espacios de transmisión o *slots* dentro de la aplicación. Sin embargo, durante esta etapa se descubrió que ciertas funciones de GTK4 no interactuaban correctamente con los elementos gráficos utilizados por GStreamer, particularmente en el manejo de *sinks* y contextos de renderizado. Debido a esto se decidió migrar completamente el proyecto a GTK3, versión que proporcionaba mayor estabilidad y compatibilidad con GStreamer.

La segunda etapa consistió en portar todos los widgets construidos originalmente en GTK4 hacia GTK3, replicando su diseño y mejorando la integración con los elementos gráficos de GStreamer. Posteriormente, la tercera etapa se centró en implementar la lógica interna de cada *slot* de transmisión, incluyendo la creación dinámica de pipelines, la administración del estado de cada stream y la detección automática de desconexiones. Los archivos `StreamSlot.cpp` y `StreamSlot.h` agrupan las funciones principales encargadas tanto de la construcción y reconfiguración de los pipelines como de la gestión de la interfaz visual asociada a cada contenedor. Para facilitar la lectura y comprensión del código, en los Cuadros 2 y 3 se presenta un resumen de las funciones implementadas en `StreamSlot.cpp`, mientras que los Cuadros 4 y 5 resumen las funciones declaradas en `StreamSlot.h` y su propósito dentro del sistema.

Finalmente, en cumplimiento con las políticas de uso ético y responsable de inteligencia artificial de la Universidad del Valle de Guatemala, se documenta que durante la etapa de depuración del código se hizo uso de herramientas como ChatGPT (versiones 5 y anteriores) para la identificación y corrección de errores durante la compilación y en el diseño del pipeline multimedia [32].

7.3. Integración de *watchdog* para manejo de desconexiones

Aunque el sistema era funcional en cuanto a recepción y visualización de múltiples transmisiones, surgió un problema importante durante las pruebas: cuando un stream se desconectaba, la última imagen recibida permanecía congelada en pantalla. Esta situación podría generar confusión o interferir con otras aplicaciones desarrolladas por estudiantes de la Universidad del Valle de Guatemala, especialmente aquellas que dependen de retroalimentación visual precisa o indicadores dinámicos.

Para resolver este inconveniente, se decidió implementar un sistema de supervisión o *watchdog* dentro de cada *slot* de transmisión. Este *watchdog* debía cumplir con dos objetivos principales:

1. Detectar periodos prolongados sin recepción de *buffers* de video.
2. Reemplazar temporalmente la imagen congelada por una pantalla negra para indicar la ausencia de transmisión.

El diseño del sistema tuvo que considerar un punto clave: evitar falsos positivos provocados por microcortes de red. Por ello, el *watchdog* no debía activarse ante breves interrupciones sino únicamente cuando la falta de *buffers* excediera un tiempo límite configurable. Una vez superado dicho umbral, el *slot* pasaba a un estado de "transmisión inactiva" y se mostraba un fondo negro. Cuando el flujo de video regresaba, el sistema reiniciaba automáticamente el *pipeline* o reanudaba la reproducción según fuera necesario. Esta funcionalidad permitió que la aplicación mosaico mantuviera un comportamiento estable y robusto incluso en redes con variabilidad o congestión momentánea, garantizando una presentación visual confiable para todos los usuarios del entorno Robotat. Finalmente, en el Cuadro 6 y el Cuadro 7 se presenta un resumen de las funciones implementadas para el mecanismo de *watchdog* dentro del sistema.

7.4. Integración de UDP al sistema

Con la versión inicial de la aplicación ya funcional utilizando transmisiones desde OBS a través del servidor SRT y recuperadas mediante GStreamer, se procedió a evaluar el rendimiento del sistema en escenarios de tiempo real. Durante estas pruebas se observó que, aunque SRT ofrecía estabilidad y corrección de errores, la latencia resultante seguía siendo demasiado elevada para aplicaciones que requerían respuesta inmediata, tales como controles directos o elementos interactivos sobre la mesa.

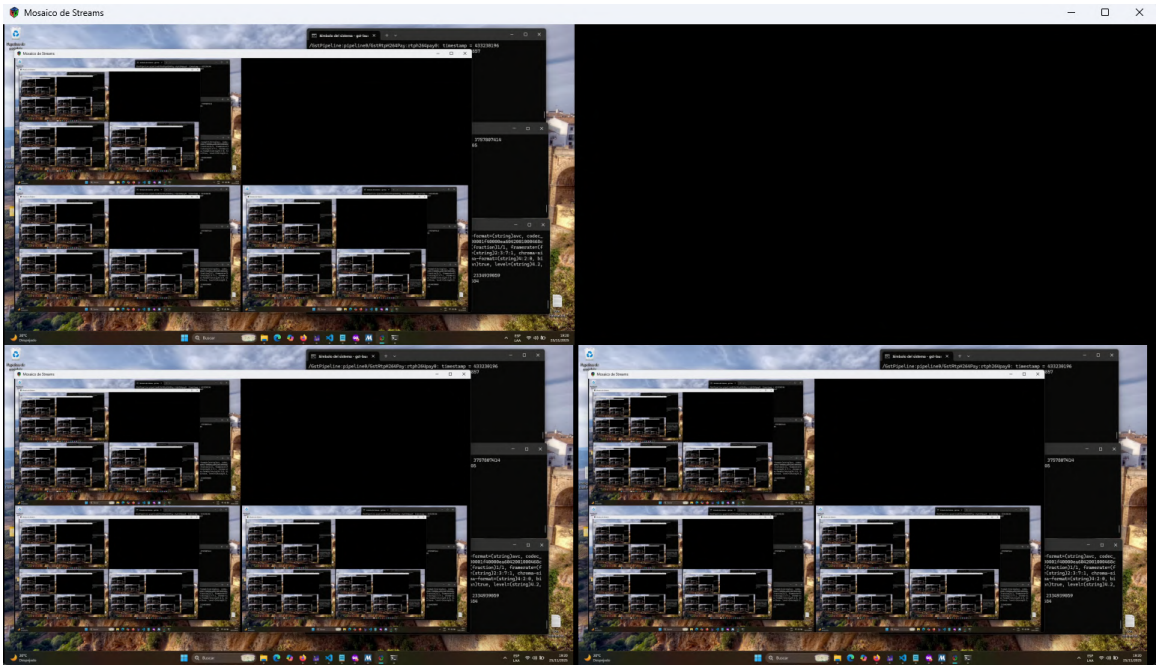
Por esta razón se exploraron otros protocolos, incluyendo TCP directo. Sin embargo, TCP introducía sobrecarga adicional debido a mecanismos de confirmación y retransmisión, lo cual aumentaba la latencia. Tras realizar pruebas empíricas utilizando pipelines simples de envío y recepción en Windows y Linux, y midiendo los tiempos con un cronómetro físico, se concluyó que UDP era la alternativa ideal: ofrecía una latencia significativamente menor y una implementación sencilla tanto para el cliente como para la aplicación mosaico.

Con esta decisión se inició el desarrollo de una nueva versión del sistema basada en transmisión directa por UDP hacia la aplicación mosaico, eliminando la necesidad de pasar por un servidor intermedio. Esto permitió obtener mejoras sustanciales en los tiempos de respuesta, volviendo viable el uso de la aplicación para escenarios interactivos en tiempo real.

Sin embargo, también se consideró que algunos usuarios podrían no contar con GStreamer o herramientas compatibles para emitir mediante UDP. Por esta razón, se optó por integrar ambos métodos de transmisión (SRT y UDP) dentro de la misma aplicación mosaico. Esto garantizó compatibilidad con OBS y simultáneamente permitió una vía de comunicación de baja latencia para usuarios avanzados o aplicaciones críticas. Las funciones de la versión final utilizadas en el archivo `main.cpp`, se describen en el Cuadro 1.

La última versión de la aplicación no solo soporta ambos protocolos de entrada, sino que también incorpora funcionalidades adicionales manejadas por teclado para controlar dinámicamente la distribución del mosaico y facilitar la interacción del operador (véase el manual de usuario en los anexos para una descripción completa de estas funciones). Esta integración dual permitió entregar una solución flexible, robusta y optimizada para el entorno Robotat.

Figura 12. Resultado final de la aplicación de proyección estilo mosaico desarrollada con GTK3



Nota. La aplicación muestra múltiples transmisiones distribuidas en un mosaico de contenedores, evidenciando la correcta recepción, organización y manejo de estados de cada stream dentro de la interfaz.

La imagen mostrada en la Figura 12 representa el resultado final de la aplicación de proyección construida con GTK3. En este ejemplo se visualizaron tres transmisiones idénticas, todas enviadas desde una misma computadora pero utilizando puertos UDP distintos, con el fin de validar el funcionamiento de los contenedores del mosaico. Los flujos se asignaron a los contenedores 1, 2 y 4, omitiendo el contenedor 3, lo cual permite observar el diseño del mosaico cuando una transmisión no está disponible para un espacio determinado.

Es importante señalar que el comportamiento observado en la Figura 12 es equivalente al que se obtendría si una transmisión originalmente activa en el tercer contenedor se desconectara por más de cinco segundos. Esto es posible gracias a la lógica implementada en la clase *Watchdog*, la cual detecta la ausencia prolongada de buffers, reemplaza el video por una pantalla negra y restaura la señal automáticamente al reconectarse.

Cuadro 1. Resumen de funciones implementadas en `main.cpp`

Función	**Parámetros**	**Descripción**
`update_layout`	`AppData* app`	Actualiza la visibilidad de los slots según el número de espacios activos establecidos por el usuario.
`apply_black_background`	`GtkWidget* widget`	Aplica un estilo CSS global que establece un fondo negro para la ventana, contenedores y widgets asociados.
`rebuild_pipelines`	`AppData* app`	Reconstruye los pipelines de GStreamer para cada slot dependiendo del modo de transmisión seleccionado (SRT, UDP Safe o UDP Fast).
`on_key_press`	`GtkWidget* widget`, `GdkEventKey* event`, `gpointer user_data`	Maneja los eventos de teclado para cambiar modos de transmisión, bloquear layout, alternar pantalla completa y ajustar la cantidad de slots activos.
`on_activate`	`GtkApplication* gtk_app`, `gpointer user_data`	Inicializa la ventana principal, la grilla, los 6 slots y conecta los eventos necesarios. También aplica estilo visual.
`main`	`int argc`, `char** argv`	Punto de entrada de la aplicación. Inicializa GStreamer, crea la instancia de la aplicación GTK y ejecuta el loop principal.

Nota. Las funciones presentadas corresponden a la lógica principal de la aplicación mosaico, incluyendo inicialización gráfica, manejo de teclado y reconstrucción dinámica de pipelines. Elaboración propia.

Cuadro 2. Descripción de las funciones implementadas en `StreamSlot.cpp` (Parte 1)

Función	Descripción
`bus_call`	Callback del bus de GStreamer. Maneja mensajes de error y eventos EOS.
`buffer_probe_cb`	Probe de buffers que notifica actividad al watchdog.
`StreamSlot: :StreamSlot`	Constructor del slot. Crea contenedor GTK, inicializa watchdog y configura callbacks.
`~StreamSlot`	Destructor seguro. Detiene watchdog, destruye pipeline y limpia widgets.
`on_watchdog_event`	Callback que muestra u oculta la pantalla negra según la actividad del stream.
`remove_existing_- video_widget`	Elimina el widget de video actual evitando widgets huérfanos.
`place_black_- placeholder`	Coloca una pantalla negra cuando no hay stream disponible o se detecta desconexión.

Nota. Parte 1 de la tabla de funciones de `StreamSlot.cpp`. Elaboración propia.

Cuadro 3. Descripción de las funciones implementadas en `StreamSlot.cpp` (Parte 2)

Función	Descripción
`init_with_streamid`	Inicializa un pipeline SRT usando `streamid`. Reconstruye pipeline y reactiva watchdog.
`init_with_udp_- port_safe`	Pipeline UDP seguro con jitterbuffer normal. Calidad estable con latencia moderada.
`init_with_udp_- port_fast`	Pipeline UDP para latencia ultra baja. Jitterbuffer mínimo y watchdog desactivado.
`setup_udp_pipeline`	Construye pipeline UDP, obtiene widget del sink, aplica probes y activa watchdog.
`init_with_black_- screen`	Detiene todo y coloca un placeholder negro fijo.
`get_widget`	Devuelve el contenedor GTK del slot. Si no existe, lo crea e inserta un placeholder negro.

Nota. Parte 2 de la tabla de funciones de `StreamSlot.cpp`. Elaboración propia.

Cuadro 4. Descripción de los métodos públicos y atributos declarados en `StreamSlot.h` (Parte 1)

Elemento	Descripción
`StreamSlot::StreamSlot`	Constructor. Inicializa el contenedor, watchdog y valores predeterminados del slot.
`~StreamSlot`	Destructor. Libera el pipeline, destruye widgets y detiene el watchdog.
`init`	(Reservado) Inicialización básica por índice. No se usa actualmente.
`init_with_streamid`	Inicializa el slot para recibir un stream SRT especificado por un `streamid`.
`init_with_udp_port_safe`	Configura un pipeline UDP estándar, orientado a estabilidad.
`init_with_udp_port_fast`	Configura un pipeline UDP optimizado para latencia mínima.
`get_widget`	Devuelve el contenedor GTK del slot para integrarse en la interfaz gráfica.
`watchdog_enabled`	Indica si el watchdog está activo. Por defecto en `true`.

Nota. Parte 1 de la documentación del header `StreamSlot.h`. Elaboración propia.

Cuadro 5. Descripción de los métodos privados y atributos internos en `StreamSlot.h` (Parte 2)

Elemento	Descripción
`container`	Contenedor general que aloja el video o la pantalla negra.
`video_widget`	Widget del sink de video generado por GStreamer.
`pipeline`	Pipeline GStreamer asociado al stream actual.
`watchdog`	Módulo interno que detecta actividad y desconexión del stream.
`on_watchdog_event`	Callback interna que controla si debe mostrarse pantalla negra o video.
`setup_udp_pipeline`	Construye un pipeline UDP genérico y aplica el watchdog.
`remove_existing_video_widget`	Elimina de forma segura el widget de video actual.
`place_black_placeholder`	Inserta una pantalla negra cuando no hay video disponible.
`init_with_black_screen`	Configura el slot para mostrar únicamente pantalla negra.

Nota. Parte 2 de la documentación del header `StreamSlot.h`. Elaboración propia.

Cuadro 6. Descripción de las funciones implementadas en `Watchdog.cpp`

Función	Descripción
`Watchdog::Watchdog`	Constructor del módulo. Inicializa el callback, el timeout y los contadores internos.
`~Watchdog`	Destructor. Detiene el hilo del watchdog de forma segura.
`notify_buffer`	Notifica que se recibió un buffer e incrementa el contador interno.
`start`	Inicia el hilo interno encargado de detectar actividad o desconexión del stream.
`stop`	Detiene el hilo del watchdog y sincroniza su finalización.
`run`	Bucle interno del watchdog. Compara actividad entre intervalos para decidir si debe mostrarse pantalla negra o restaurar el video.

Nota. Funciones de `Watchdog.cpp`. Elaboración propia.

Cuadro 7. Descripción de los métodos y atributos declarados en `Watchdog.h`

Elemento	Descripción
`Callback`	Tipo de función usada para indicar si debe mostrarse la pantalla negra o restaurarse el video.
`Watchdog::Watchdog`	Constructor público que configura el callback y el timeout del módulo.
`~Watchdog`	Destructor que garantiza la detención del hilo del watchdog.
`notify_buffer`	Método público para indicar que ha llegado un buffer nuevo.
`start`	Inicia el hilo del watchdog.
`stop`	Detiene el hilo interno encargado de la vigilancia.
`run`	Función privada ejecutada por el hilo; compara los buffers recibidos y activa el callback.
`buffer_count`	Contador atómico del número de buffers recibidos.
`timeout_ms`	Tiempo máximo de espera sin recibir buffers antes de disparar el callback.
`watchdog_thread`	Hilo que ejecuta el proceso de vigilancia del watchdog.
`running`	Bandera que controla si el hilo del watchdog debe seguir ejecutándose.
`callback`	Función que se llama cuando se detecta actividad o desconexión del stream.

Nota. Documentación del header `Watchdog.h`. Elaboración propia.

CAPÍTULO 8

Lectura de datos y mapeo de coordenadas

8.1. Lectura por medio de MQTT

Las cámaras OptiTrack envían información sobre la posición y rotación tanto de agentes como de marcadores pasivos hacia la red Robotat dentro de la UVG, utilizando como intermediario un *broker* que opera sobre el enrutador principal del entorno. Por esta razón, fue necesario emplear funciones de retorno (callbacks) y el protocolo de mensajería MQTT para obtener en tiempo real la información del marcador seleccionado para el sistema dinámico de realidad aumentada, el cual fue escogido de forma arbitraria para las pruebas iniciales (véase Cuadro 8).

Cuadro 8. Descripción de las funciones implementadas en el script de lectura MQTT

Función	**Descripción**
`on_connect`	Función de *callback* que se ejecuta al establecer conexión con el broker MQTT; muestra el código de retorno y realiza la suscripción al tópico definido.
`on_message`	*Callback* que procesa cada mensaje recibido; decodifica el paquete JSON, filtra por el identificador objetivo y extrae la posición y rotación del marcador para actualizar las variables globales.
`mqtt.Client()`	Constructor de la clase cliente MQTT; instancia el objeto que gestionará la comunicación con el broker.
`client.connect`	Inicializa la conexión con el broker MQTT utilizando la dirección IP, puerto y tiempo de *keep-alive*.

Nota. Documentación de funciones del script Python para lectura de marcadores vía MQTT. Elaboración propia.

8.2. Transformación manual de los planos del sistema

Una vez extraída la información de posición y orientación del marcador seleccionado, fue posible emplearla para iniciar el desarrollo de las aplicaciones de validación correspondientes a los objetivos del proyecto. Inicialmente, esta información se integró con la aplicación de mosaico de transmisiones, permitiendo generar un sistema interactivo de realidad aumentada para un solo usuario con una única transmisión.

Sin embargo, debido a que el alcance del proyecto se amplió para soportar hasta seis clientes simultáneos, surgió la necesidad de escalar el área de detección de marcadores pasivos de forma que coincidiera con la partición de la pantalla de proyección. Para lograrlo, se definieron tres planos de coordenadas fundamentales dentro del sistema:

1. **Primer plano:** área de proyección entregada por el proyector del entorno Robotat.
2. **Segundo plano:** área de detección de marcadores que se desea transformar.
3. **Tercer plano:** superficie de visualización equivalente a la pantalla del cliente.

Para realizar esta transformación, se desarrolló una aplicación sencilla en Python, sin comunicación MQTT, cuyo propósito era obtener las dimensiones iniciales del primer plano y emplearlas como referencia para definir las dimensiones base del segundo plano. Estas dimensiones podían modificarse directamente en el código para ajustar el área de detección según las necesidades del sistema (véase Cuadro 9).

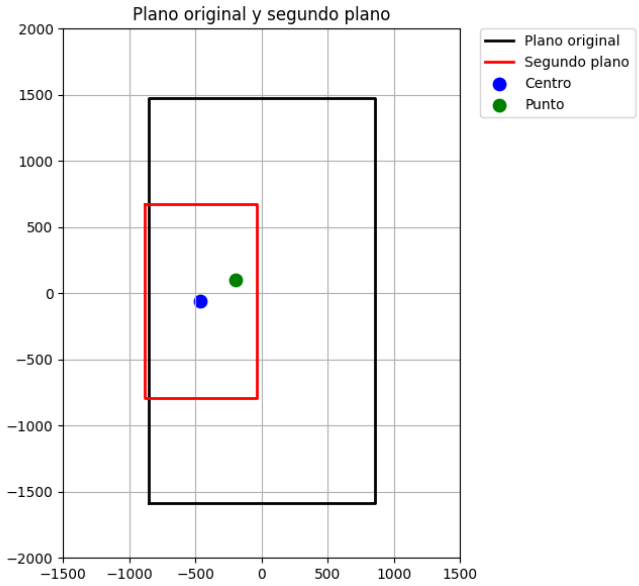
Cuadro 9. Descripción de las funciones y métodos utilizados en el script de transformación de planos

Función	Descripción
`input()`	Función integrada de Python utilizada para solicitar datos al usuario, tales como centro del segundo plano, escala, rotación y punto a transformar.
`np.vstack`	Concatena verticalmente los puntos del segundo plano para cerrar la figura poligonal.
`cv2.find Homography`	Calcula la matriz de homografía entre el segundo plano transformado y el tercer plano (pantalla), permitiendo el mapeo entre ambos sistemas de coordenadas.
`cv2.perspective Transform`	Aplica la homografía calculada para transformar un punto del segundo plano hacia las coordenadas del tercer plano.

Nota. Descripción de las funciones utilizadas en el script para generar la homografía entre planos y visualizar los resultados. Elaboración propia.

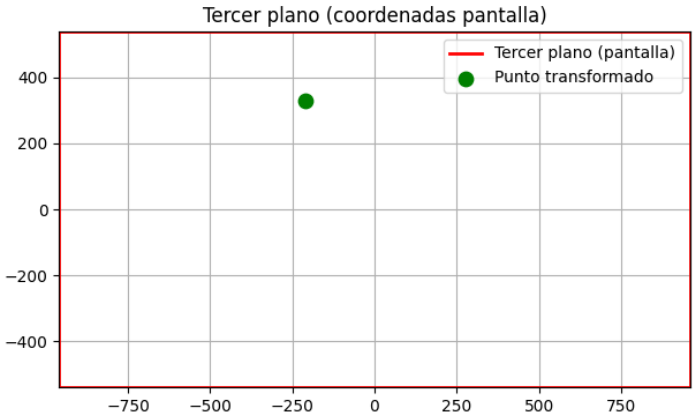
Posteriormente, el usuario ingresaba manualmente la nueva coordenada del centro del segundo plano, el factor de escala y el ángulo de rotación. Con estos parámetros, el sistema calculaba una homografía para transformar puntos del segundo plano hacia el tercer plano, equivalente a una pantalla tradicional de 1920×1080 píxeles. Finalmente, el usuario proporcionaba manualmente la coordenada de un marcador como entrada para simular su posición dentro del área de detección y validar el funcionamiento del sistema de mapeo.

Figura 13. Visualización del segundo plano de detección (área roja)



Nota. La figura muestra el área de detección definida manualmente mediante centro, escala y rotación. El punto verde representa la posición simulada del marcador.

Figura 14. Marcador transformado hacia el tercer plano (pantalla 1920×1080)



Nota. La figura ilustra la homografía aplicada al marcador dentro del plano de pantalla. El mapeo corresponde al tamaño estándar de un monitor HD.

Los resultados muestran que, si un marcador se encuentra dentro de la zona de detección definida (el segundo plano, en rojo en la Figura 13), su posición es correctamente mapeada al tercer plano con las dimensiones de una pantalla tradicional 1920×1080, como se observa en la Figura 14.

8.3. Transformación automática de los planos del sistema e integración de protocolo MQTT

Tras validar que el método de homografía funcionaba de manera adecuada, el siguiente paso consistió en automatizar el proceso. Para ello, los parámetros de centro, escala y rotación del segundo plano se incorporaron directamente en el código fuente, eliminando la necesidad de ingresarlos manualmente mediante la terminal.

Los valores de ancho y alto del segundo plano fueron obtenidos mediante mediciones físicas con una cinta métrica, tomando como referencia la superficie real proyectada sobre la mesa del Robotat. De manera similar, el centro del segundo plano se determinó ubicando un marcador físico en el centro de la proyección y registrando su coordenada mediante la primera versión de la aplicación.

Posteriormente, se integró la lógica de comunicación MQTT para obtener en tiempo real las coordenadas del marcador, reemplazando la entrada manual utilizada anteriormente. Este cambio permitió observar inmediatamente la posición del marcador dentro del sistema, visualizándose de forma idéntica a las figuras mostradas anteriormente (Figura 13 y Figura 14), con la diferencia de que el punto verde en el gráfico (representando el marcador real) se mueve dinámicamente al desplazarse físicamente el marcador sobre la mesa Robotat.

CAPÍTULO 9

Primera validación del sistema mediante una aplicación interactiva

Dado que ya se contaba con una etapa funcional capaz de obtener coordenadas mediante MQTT desde la red Robotat, y que además era independiente del tamaño de la pantalla proyectada gracias a la homografía de planos, se procedió a desarrollar una aplicación que integrara tanto el sistema de proyección implementado como la lectura calibrada de coordenadas. Es importante mencionar que, sin dicha etapa de calibración, esta validación únicamente habría permitido alcanzar el segundo objetivo del proyecto: utilizar datos provenientes del sistema de captura de movimiento para modificar la visualización proyectada. Sin embargo, al integrar la calibración de coordenadas, esta validación permitió también cumplir el tercer objetivo relacionado con lograr una sincronización efectiva entre la captura de movimiento y la proyección visual.

Por practicidad, se determinó que esta aplicación debía mover el cursor del mouse del cliente. Esto permitió evaluar no solo el mapeo espacial calibrado sobre la pantalla del cliente, sino también la utilización de coordenadas en el eje z para generar eventos interpretables como acciones reales en la interfaz. De acuerdo con las políticas de uso ético y responsable de la inteligencia artificial de la Universidad del Valle de Guatemala, se indica que para esta etapa de validación se empleó asistencia con ChatGPT 5 y versiones anteriores para depurar errores durante la compilación del código fuente [32].

9.1. Control dinámico del cursor de mouse con coordenadas manuales

El primer paso consistió en desarrollar una aplicación en Python que permitiera mover el cursor del mouse no en tiempo real, sino utilizando coordenadas introducidas manualmente por el usuario en la terminal. Para ello se utilizó la biblioteca `pyautogui` y las funciones descritas en el Cuadro 10, aunque fue necesario desactivar la opción *failsafe* de la librería con el fin de evitar bloqueos al acceder a los bordes de la pantalla.

Cuadro 10. Funciones principales utilizadas en el script de control de mouse

Función	Descripción
`pyautogui.size()`	Obtiene el tamaño total de la pantalla en píxeles, utilizado para escalar las coordenadas transformadas hacia el sistema de pantalla.
`map_to_screen()`	Función definida por el usuario que convierte coordenadas medidas en milímetros al espacio de coordenadas de la pantalla (en píxeles). Incluye normalización, inversión de ejes y recorte seguro.
`pyautogui.moveTo()`	Mueve el cursor hacia las coordenadas proyectadas en pantalla, permitiendo el control en tiempo real desde los datos del marcador.
`pyautogui.mouseDown()`	Simula la acción de presionar un botón del mouse (izquierdo o derecho) según el valor del eje Z del marcador.
`pyautogui.mouseUp()`	Simula la liberación del botón del mouse cuando el marcador sale del rango definido para un clic activo.
`input()`	Recibe desde consola las coordenadas del marcador en milímetros (x, y, z), permitiendo probar el sistema sin conexión a un capturador de movimiento.

Nota. Funciones esenciales utilizadas para mapear coordenadas reales a la pantalla y generar acciones de mouse dentro del sistema de control interactivo. Elaboración propia.

El resultado fue satisfactorio, ya que el cursor era capaz de desplazarse a las coordenadas especificadas e incluso ejecutar clics derechos e izquierdos según los valores ingresados. No obstante, esta etapa no permitió observar interacciones complejas como arrastre de objetos debido a la lentitud inherente a la introducción manual de coordenadas.

9.2. Control dinámico del cursor de mouse con coordenadas extraídas del Robotat

La siguiente etapa consistió en integrar el sistema previamente desarrollado de mapeo de coordenadas con el control del cursor, de manera que el programa ya no dependiera de valores manuales, sino de las coordenadas proporcionadas en tiempo real por el sistema Robotat. Las nuevas funciones integradas al programa se describen en el Cuadro 11:

Cuadro 11. Funciones principales utilizadas en el script completo de control de mouse mediante coordenadas de captura de movimiento

Función	Descripción
`threading.Thread()`	Permite ejecutar en paralelo tanto el cliente MQTT como el monitor de la tecla ESC, evitando bloquear el loop principal.
`map_to_screen_-from_marker()`	Función definida por el usuario que aplica la homografía al punto (x, y) del marcador y lo convierte a coordenadas absolutas de pantalla.
`keyboard.is_-pressed()`	Detecta si la tecla ESC se mantiene presionada; permite terminar el programa de manera segura mediante un hilo secundario.
`json.loads()`	Convierte la cadena recibida vía MQTT en un diccionario Python, permitiendo el acceso estructurado a los datos del movimiento.

Nota. Descripción de las funciones no repetidas esenciales utilizadas en la adquisición de datos, cálculo de homografía y control del puntero mediante un marcador físico. Elaboración propia.

Los resultados fueron nuevamente satisfactorios en cuanto al movimiento del cursor y la generación de eventos de clic sobre la pantalla. Estas pruebas se realizaron inicialmente sobre el escritorio del sistema operativo, ya que facilitaba la visualización inmediata de cambios. Sin embargo, el comportamiento no fue adecuado al interactuar con navegadores u otras aplicaciones: algunos clics no eran detectados o no generaban la acción esperada. Tras investigar las limitaciones de `pyautogui`, se concluyó que dicha biblioteca simula eventos de mouse en un nivel superficial, y diversos navegadores o aplicaciones de Windows bloquean estos eventos por razones de seguridad.

9.3. Corrección del control dinámico del cursor con eventos reales mediante `ctypes` y `SendInput`

Debido a las limitaciones anteriores, fue necesario implementar un método diferente para generar eventos de mouse de forma que el sistema operativo los interpretara como eventos físicos reales. La solución fue utilizar `ctypes` junto con la función `SendInput` de la API de Windows, lo cual permitió enviar directamente al kernel eventos equivalentes a los producidos por un dispositivo físico. Las funciones y estructuras principales utilizadas en esta implementación se resumen en el Cuadro 12.

Cuadro 12. Funciones y estructuras principales utilizadas en el script de control de mouse con MQTT y homografía

Función / Componente	**Descripción**
`MOUSEINPUT / INPUT` (ctypes)	Estructuras definidas para interactuar directamente con la API de Windows mediante `SendInput`, permitiendo movimiento y clics del mouse a bajo nivel.
`move_mouse()`	Convierte coordenadas absolutas a la escala de Windows (0–65535) y mueve el cursor utilizando eventos `MOUSEEVENTF_MOVE | MOUSEEVENTF_ABSOLUTE`.
`left_down()`, `left_up()`, `right_down()`, `right_up()`	Simulación directa de clics presionados y liberados usando `SendInput`, sin librerías externas.
`start_mqtt()`	Inicializa el cliente MQTT y ejecuta el loop de escucha en un hilo independiente para no bloquear el programa principal.
`cv2.findHomography()`	Calcula la matriz de homografía entre el plano local del Robotat y la pantalla del proyector, permitiendo transformar todas las coordenadas del marcador.
`cv2.perspective Transform()`	Aplica la homografía calculada para convertir una coordenada (x, y) del marcador al sistema de referencia de la pantalla.
`map_to_screen_from_marker()`	Aplica la transformación homogénea completa y ajusta las coordenadas resultantes a píxeles reales de pantalla, corrigiendo orientación e inversión de ejes.
`esc_monitor()`	Hilo que detecta si la tecla ESC se mantiene presionada durante 5 segundos para finalizar el programa de manera segura.

Nota. Resumen de los componentes esenciales utilizados en el sistema final de control de mouse mediante datos de captura de movimiento, ctypes y Sendinput. Elaboración propia.

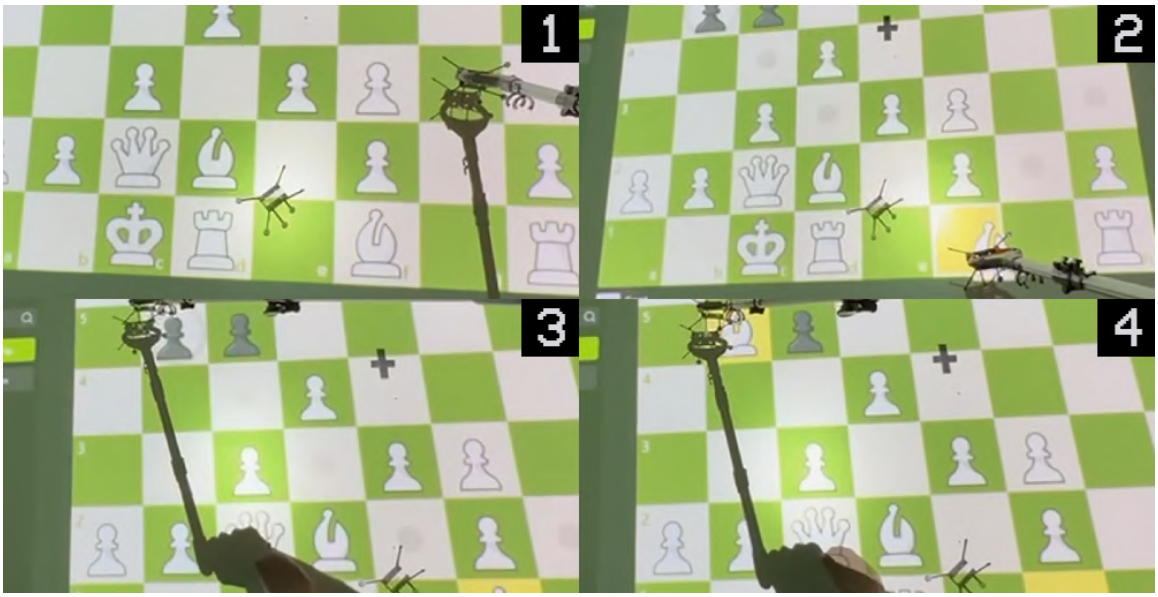
El funcionamiento integrado del sistema dependió directamente de la coordenada z del marcador, utilizada como eje de control para distinguir distintos eventos de mouse. Para ello se definieron tres umbrales: entre 0 y 100 mm se registró un clic izquierdo; entre 100 y 300 mm se liberó cualquier clic activo; y entre 300 y 1500 mm se generó un clic derecho. Este esquema permitió una interacción robusta, intuitiva y completamente compatible con el sistema de proyección utilizado en el entorno Robotat.

9.4. Integración del sistema de proyección para la primera validación completa del sistema

Para esta etapa se integró el sistema de proyección mediante uno de los puentes disponibles hacia el servidor de proyección *OBS/GStreamer*. Con esta integración fue posible validar el desempeño del sistema de control del mouse dentro del entorno de realidad aumentada proyectado. Los resultados obtenidos fueron exitosos en tres escenarios distintos, cada uno evaluando una parte fundamental del sistema de control del mouse.

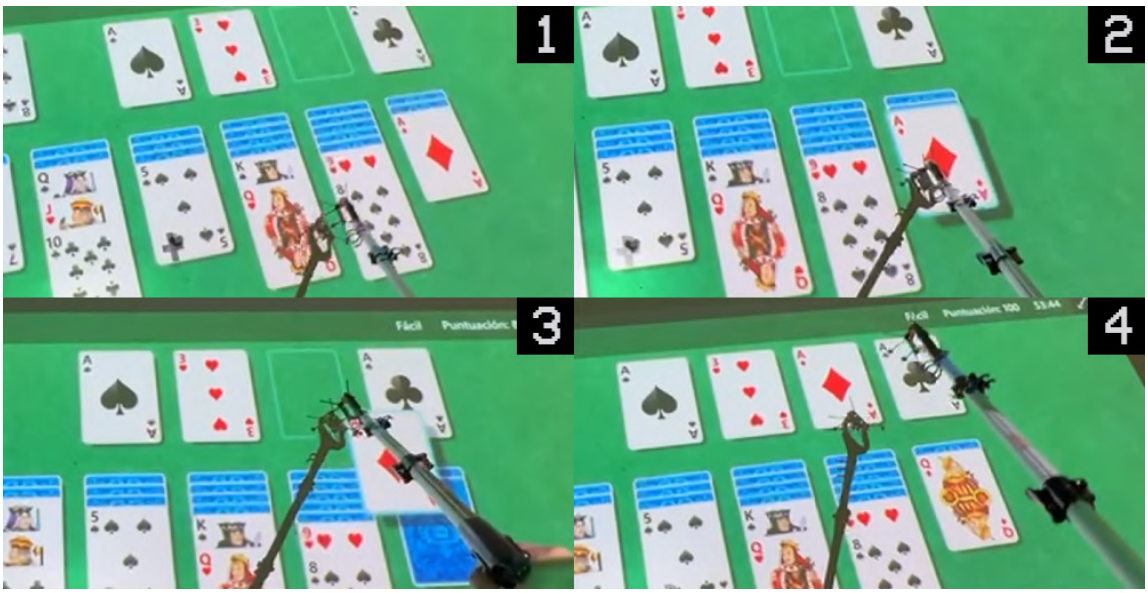
El primer escenario utilizó el navegador Firefox junto con la plataforma *chess.com* para validar el clic izquierdo, tal como se observa en la Figura 15, verificando que las piezas respondieran correctamente al ser seleccionadas, movidas y utilizadas para capturar, y que el movimiento del cursor permaneciera sincronizado durante toda la interacción. El segundo escenario empleó la aplicación *Solitaire* de Windows para evaluar el mecanismo de arrastre, como se evidencia en la Figura 16, comprobando que mantener presionado el clic izquierdo permitiera desplazar elementos (en este caso, cartas) de forma precisa y fluida. El tercer escenario consistió en utilizar el escritorio de Windows para validar el clic derecho, según se aprecia en la Figura 17, comprobando que el menú contextual apareciera correctamente tras superar el umbral correspondiente y que la transición entre estados fuera interpretada sin interferencias.

Figura 15. Validación del clic izquierdo mediante movimiento del marcador en el entorno Robotat y utilizando la plataforma *chess.com*



Nota. La figura muestra el estado del tablero tras hacer clic sobre un alfil y moverlo para capturar un peón, validando la correcta interpretación del clic izquierdo y la sincronización del cursor.

Figura 16. Validación del arrastre mediante clic izquierdo sostenido dentro de *Solitaire*



Nota. La figura muestra la carta As de diamantes siendo arrastrada hacia la pila de base (*foundation*), evidenciando la detección correcta del clic sostenido y el movimiento continuo.

Figura 17. Validación del clic derecho mediante el control por coordenada z en el escritorio de Windows

Nota. La figura muestra la elevación del marcador para alcanzar el umbral del clic derecho y luego el retorno al estado intermedio, lo que permite visualizar correctamente el menú contextual de Windows.

CAPÍTULO 10

Segunda validación del sistema mediante aplicación interactiva

El objetivo de esta segunda validación fue evaluar nuevamente la sincronización efectiva entre la captura de movimiento y la proyección visual, tal como en la validación previa. Sin embargo, esta etapa amplió el alcance al incorporar múltiples marcadores dentro del entorno dinámico Robotat, permitiendo comprobar el funcionamiento del sistema en un escenario multiusuario. Para ello se desarrolló un juego de *air hockey* que utilizó las coordenadas de dos marcadores para representar las posiciones de los jugadores, quienes interactuaban con un entorno digital proyectado sobre la superficie del Robotat.

De acuerdo con las políticas de uso ético y responsable de la inteligencia artificial de la Universidad del Valle de Guatemala, se menciona que en esta etapa se utilizó ChatGPT 5 y versiones anteriores para generar las imágenes empleadas como fondo y *assets* del juego, así como para asistir en la depuración de errores durante la compilación del código fuente [32].

10.1. Estructura del código

La aplicación se desarrolló bajo una estructura modular sencilla. Los archivos correspondientes a las clases y lógica del juego fueron almacenados en la carpeta `src`, mientras que los recursos gráficos, fuentes, audio y demás elementos necesarios se ubicaron en la carpeta `assets`. Esta organización facilitó la depuración, la escalabilidad del proyecto y la independencia de los componentes del juego.

10.2. Archivo principal del código

Aunque el código principal mantuvo un diseño simple, fue responsable del manejo de ventanas, de la configuración de la resolución de pantalla y de permitir la futura inclusión de nuevas interfaces, como un menú de opciones o incluso juegos adicionales dentro de la misma aplicación. El Cuadro 13 describe con mayor detalle las funciones utilizadas en el archivo principal. Por motivos de claridad, primero se presentan las subclases empleadas por la clase principal `Game`, para posteriormente explicar cómo dicha clase integra y coordina a cada una de ellas.

Cuadro 13. Funciones principales empleadas en el módulo `main.py` del juego Air Hockey 2D

Función	Descripción
`pygame.init()`	Inicializa todos los módulos principales de Pygame necesarios para gráficos, eventos y control del juego.
`pygame.display.set_mode()`	Crea la ventana del juego con la resolución especificada, sirviendo como superficie principal para renderizar.
`Game.run()`	Función central del juego que contiene el ciclo principal: manejo de eventos, actualización de estados y dibujado de todos los elementos.
`pygame.quit()`	Cierra correctamente todos los módulos de Pygame al finalizar la ejecución del programa.
`main()`	Encapsula toda la configuración inicial y lanza la ejecución del juego, actuando como punto de entrada del programa.

Nota. Tabla reducida que resume las funciones esenciales utilizadas en el módulo principal del juego Air Hockey 2D.

10.3. Subclases `DigitMatrix` y `Scoreboard`

Tras definir el espacio de juego en la clase `Game`, las primeras subclases desarrolladas fueron las relacionadas con el marcador. Estas dependían exclusivamente de la biblioteca `pygame`, lo que permitió familiarizarse con las herramientas gráficas disponibles.

Primero se definieron dimensiones estándar para los dígitos y se estableció un mapa de bits `DIGIT_MAP` que contenía las configuraciones de segmentos necesarios para formar cada número. La clase `DigitMatrix` se encargó de renderizar estos dígitos y posteriormente escalarlos mediante operaciones de *blit*. Por su parte, la clase `Scoreboard` administró la lógica de actualización del marcador, determinando qué dígito debía mostrarse según las acciones de juego, como la anotación de goles o el avance del temporizador. Las funciones principales asociadas a estas subclases se describen en el Cuadro 14.

Cuadro 14. Funciones principales de las subclases `DigitMatrix` y `Scoreboard`

Método	Descripción
`DigitMatrix.render_digit()`	Genera y devuelve una superficie (*Surface*) con el dígito renderizado a partir de la matriz LED de 7×4.
`DigitMatrix.draw()`	Dibuja el dígito renderizado sobre la superficie objetivo, con soporte opcional de escalado.
`Scoreboard.set_score()`	Define directamente las puntuaciones de ambos equipos, asegurando que nunca sean negativas.
`Scoreboard.add_point()`	Incrementa la puntuación del equipo indicado (1 o 2).
`Scoreboard.set_time()`	Establece el tiempo restante del marcador, limitado a valores no negativos.
`Scoreboard.tick()`	Reduce el tiempo del cronómetro según el intervalo *dt* en segundos; se llama cada frame.
`Scoreboard.draw_number()`	Renderiza y dibuja un número de dos dígitos utilizando internamente la clase `DigitMatrix`.
`Scoreboard.draw_time()`	Dibuja el tiempo en formato `MM SS`, compuesto por dígitos LED y con espacio reservado para el separador ":".
`Scoreboard.draw()`	Genera el marcador completo: puntaje del equipo 1, tiempo y puntaje del equipo 2 dentro de una región fija.

Nota. Resumen de las funciones esenciales que permiten generar dígitos LED estilo matriz y un marcador completo para el juego Air Hockey 2D.

10.4. Subclases `GameState` y `UIManager`

Estas subclases se crearon con el fin de mantener a `Game` lo más modular posible, separando la lógica de estados y la interfaz gráfica. Las funciones de ambas subclases se detallan a profundidad en el Cuadro 15.

`GameState` permitió definir estados internos del juego, agrupando métodos relacionados con cada uno de ellos y mejorando así el control del flujo general.

`UIManager`, inicialmente encargado de cargar *assets* y dibujar elementos auxiliares como un temporizador, evolucionó para gestionar fuentes tipográficas, mensajes de pausa, continuación y fin del juego. Conforme avanzó el desarrollo, también administró los mensajes de error, mostrando imágenes predefinidas de la carpeta `assets` cuando ocurrían condiciones no válidas dentro de la lógica del juego.

Cuadro 15. Funciones principales de las subclases `GameState` y `UIManager`

Método	Descripción
`UIManager.set_warning(warning_type)`	Activa una advertencia específica ('center', 'player1', 'player2'), mostrando la imagen correspondiente y cambiando el estado a `RESET_WARNING`.
`UIManager.clear_warning()`	Desactiva cualquier advertencia activa y retorna el estado a `RUNNING` si corresponde.
`UIManager.draw_ready_go(text)`	Dibuja un overlay semitransparente con el texto centrado para la secuencia READY/GO.
`UIManager.update_timer(dt)`	Reduce el temporizador si el estado es `RUNNING`, y cambia a `FINISHED` cuando el tiempo llega a cero.
`UIManager.toggle_pause()`	Alterna el estado entre `RUNNING` y `PAUSED`.
`UIManager.draw_overlay(alpha)`	Dibuja un overlay semitransparente sobre toda la pantalla con el nivel de opacidad dado.
`UIManager.draw_center_text(text, font, color)`	Dibuja texto centrado en la pantalla usando la fuente y color especificados.
`UIManager.draw_timer()`	Muestra el tiempo restante en formato MM:SS en la parte superior central de la pantalla.
`UIManager.draw(continue_timer)`	Dibuja mensajes, overlays y advertencias según el estado actual del juego (`PAUSED`, `FINISHED`, `RESET_WARNING`).

Nota. Resumen de funciones clave para el manejo de estados del juego y la interfaz visual en la clase `UIManager`.

10.5. Subclase `Rink`

Una vez establecida la interfaz general del espacio de juego, se comenzaron a añadir sus elementos físicos. El primero fue el *rink* o pista de juego, diseñado con paredes rectas y esquinas redondeadas. Cada segmento fue almacenado en un arreglo y visualizado mediante un método de depuración.

Cuando se integró la biblioteca `pymunk` para simular físicas en 2D, a las paredes también se les añadieron propiedades físicas como elasticidad, fricción y tipos de colisión, definiéndolas como cuerpos estáticos. Sin embargo, durante las pruebas se descubrió que las esquinas redondeadas podían dejar pequeños espacios que permitían al disco escapar de la pista. La solución fue generar segmentos auxiliares con ayuda de la biblioteca `math`, rellenando dichos espacios y corrigiendo el comportamiento físico. Las funciones principales asociadas a esta subclase se describen en el Cuadro 16.

Cuadro 16. Funciones principales de la subclase `Rink`

Método	Descripción
`Rink.__init__-(space, x, y, width, height, corner_radius, wall_thickness)`	Crea la pista de juego, definiendo el rectángulo base y generando las paredes rectas y esquinas redondeadas usando segmentos de `pymunk`. Configura propiedades físicas como elasticidad, fricción y tipo de colisión, y agrega todas las paredes al espacio de simulación.
`Rink.draw_-debug(screen)`	Dibuja las paredes de la pista sobre la pantalla para fines de depuración, mostrando las líneas de los segmentos con color verde.

Nota. Resumen de las funciones clave para la creación y visualización de la pista de juego en la subclase `Rink`.

10.6. Subclase `Player`

La subclase `Player` se diseñó para representar a cada jugador. Inicialmente consistió en un círculo cuya posición dependía de las coordenadas del mouse, visualizado mediante un método de depuración. Luego se añadieron propiedades físicas usando `pymunk`, definiendo a los jugadores como cuerpos cinemáticos que podían interactuar con otros objetos sin verse desplazados por ellos. Las funciones clave asociadas a esta subclase se describen a continuación en el Cuadro 17.

Cuadro 17. Funciones principales de la subclase `Player`

Método	Descripción
`Player.__init_-_(space, x, y, radius, mass, asset_path)`	Crea un jugador con cuerpo kinemático en `pymunk`, define su colisión como círculo, asigna propiedades físicas (elasticidad, fricción) y carga opcionalmente una imagen para renderizado.
`Player.update(dt, rink, target_x, target_y)`	Actualiza la posición del jugador hacia las coordenadas objetivo, limitando el movimiento dentro del rectángulo de la pista y ajustando colisiones contra las paredes. Calcula la velocidad necesaria para mover el cuerpo cinemáticamente.
`Player.draw(screen)`	Dibuja la representación visual del jugador en la pantalla. Si tiene imagen, la renderiza centrada en su posición; si no, dibuja un círculo azul.
`Player.draw_-debug(screen, rink)`	Dibuja un contorno del jugador (círculo rojo) para depuración de colisiones y posición dentro de la pista.

Nota. Resumen de las funciones clave para la subclase `Player`, incluyendo inicialización, actualización de posición y renderizado.

Se identificó un problema importante: los jugadores podían salirse de la pista si el mouse abandonaba sus límites. Se intentaron soluciones como convertirlos en cuerpos dinámicos o redefinir la pista dentro de esta clase, pero solo generaron comportamientos erráticos. La solución definitiva fue agregar una verificación dentro del método `update`, limitando la posición del jugador al área de la pista. Finalmente, el método `draw` se actualizó para usar imágenes de `assets` en lugar de las figuras de depuración.

10.7. Subclase `Puck`

La subclase `Puck` se desarrolló de manera similar a `Player` y sus funciones principales se detallan en el Cuadro 18. El disco se creó desde un inicio como un objeto dinámico en `pymunk`. Sin embargo, se detectó que podía salir de la pista al recibir impactos de alta velocidad o acumulación de energía. Para corregir esto se implementaron limitaciones a su velocidad máxima y mecanismos de amortiguación al acercarse a las paredes. Además, su método `draw` se modificó para usar un *asset* gráfico en lugar de su figura de depuración.

Cuadro 18. Funciones principales de la subclase `Puck`

Método	Descripción
`Puck.__init__(space, x, y, radius, mass, max_speed, asset_path)`	Crea el puck con cuerpo dinámico en `pymunk`, define colisión circular, asigna propiedades físicas (elasticidad, fricción), limita velocidad máxima y carga opcionalmente una imagen para renderizado.
`Puck.reset(x, y)`	Reinicia la posición del puck y establece su velocidad a cero, normalmente usado para iniciar un punto o tras un gol.
`Puck.limit_speed()`	Controla que la velocidad del puck no exceda el límite máximo y aplica ligera amortiguación para simular fricción.
`Puck.keep_inside_rink(rink)`	Asegura que el puck permanezca dentro de la pista, corrige colisiones con paredes y esquinas, y ajusta velocidad con rebotes y amortiguación para estabilidad.
`Puck.draw(screen)`	Dibuja el puck en la pantalla; si tiene imagen se renderiza centrado, si no, dibuja un círculo gris oscuro.
`Puck.draw_debug(screen)`	Dibuja un contorno rojo del puck para depuración de colisiones y posiciones.

Nota. Resumen de funciones clave de la subclase `Puck`, incluyendo inicialización, reinicio, control de velocidad y renderizado.

10.8. Subclase `Goal`

Esta subclase fue la más sencilla, ya que consistió únicamente en crear dos rectángulos definidos como sensores capaces de detectar cuando el disco ingresaba a las porterías. Su diseño inicial fue suficiente y no requirió modificaciones posteriores; además, sus funciones se detallan a continuación en el Cuadro 19.

Cuadro 19. Funciones principales de la subclase `Goal`

Método	Descripción
`Goal.__init__-(space, x, y, width, height, team)`	Crea la portería como un cuerpo estático en `pymunk` con forma rectangular, asigna un tipo de colisión distinto según el equipo y la define como sensor para detectar goles sin bloquear el puck.
`Goal.draw_-debug(screen)`	Dibuja el área de la portería en azul para depuración visual; no se usa en la partida real.

Nota. Resumen de funciones clave de la subclase `Goal`, incluyendo inicialización y visualización para debug.

10.9. Clase principal `Game`

La clase principal se desarrolló en paralelo con las subclases anteriores y concentró gran parte de la lógica del juego. Inicialmente incluía manejo de colisiones, detección de goles, dibujo general del espacio y actualización del estado de cada objeto en cada *frame*.

Uno de los retos importantes fue la definición de condiciones de error para evitar situaciones injustas o inconsistentes derivadas del uso de marcadores físicos. Por ejemplo, antes de iniciar una partida o al reiniciar el disco tras un gol, la posición de los marcadores podía generar conflictos si no estaban situados correctamente. Para resolverlo, se implementaron mensajes de error que detenían temporalmente el juego hasta que los jugadores colocaran los marcadores en posiciones válidas. Para ello se desarrollaron controladores específicos y se integró una etapa de comprobación previa al inicio del ciclo principal. La inclusión de estados mediante `GameState` permitió administrar estas condiciones sin interferencias. Para mayor información acerca de las funciones principales de esta clase véase el Cuadro 20.

Por último, al igual que en la validación anterior, se integraron métodos para recibir coordenadas de marcadores mediante MQTT, reutilizando y adaptando lógica de aplicaciones previas. Algunas subclases, como `Player`, se modificaron para reemplazar la dependencia del mouse con el uso de datos capturados desde Robotat, asegurando un control físico real del juego.

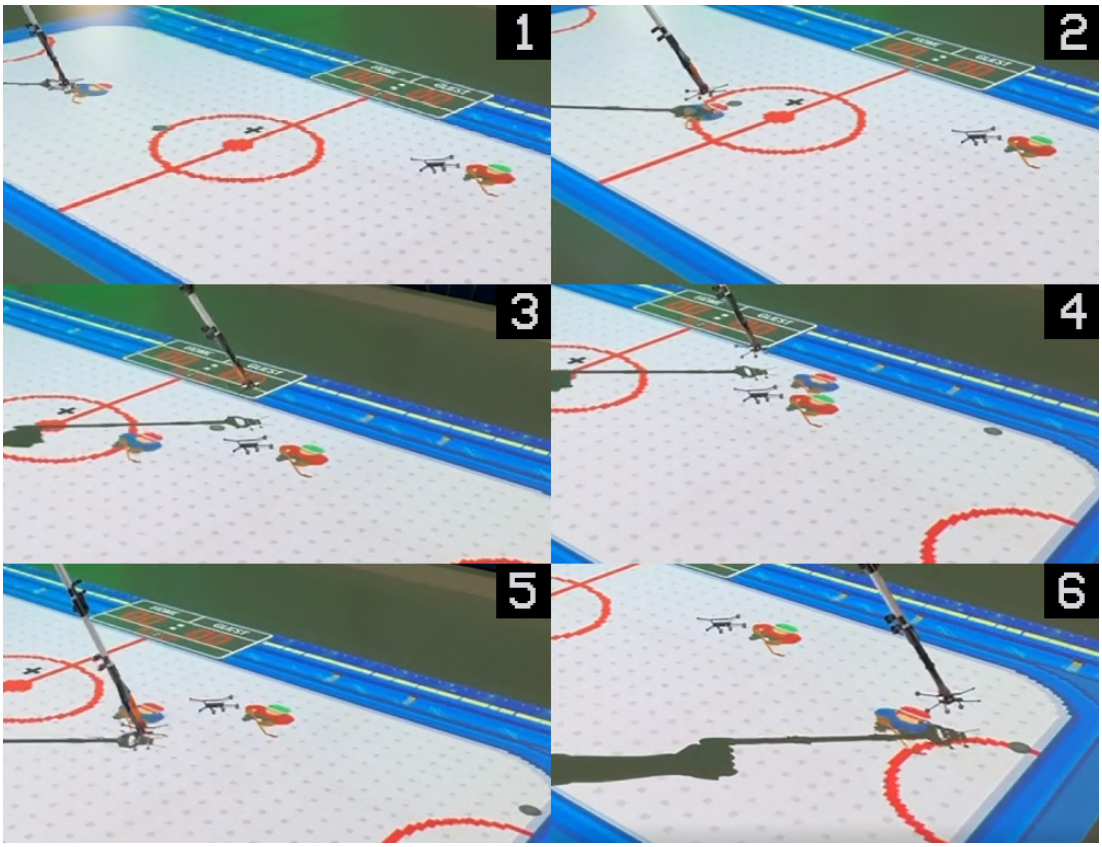
Cuadro 20. Funciones principales de la clase `Game`

Método	Descripción
`Game.__init__(screen)`	Inicializa el juego, incluyendo el espacio físico de Pymunk, fondo, scoreboard, rink, puck, jugadores, porterías, colisiones, UI y cliente MQTT para markers.
`setup_collisions()`	Define los handlers de colisión entre puck, jugadores y porterías, configurando rebotes según velocidad relativa y permitiendo ignorar colisiones fantasma.
`goal_scored(team)`	Marca un gol pendiente para el equipo indicado.
`process_pending_goal()`	Ejecuta el gol pendiente: actualiza marcador, reinicia puck y activa advertencia de reseteo.
`start_marker_thread()`	Inicia el cliente MQTT en un hilo independiente para recibir posiciones de markers y calcula la homografía para mapear al espacio de pantalla.
`mqtt_on_connect(client, userdata, flags, rc)`	Callback de conexión MQTT; suscribe al topic de markers.
`mqtt_on_message(client, userdata, msg)`	Procesa mensajes MQTT, transforma coordenadas de markers y actualiza posiciones de los jugadores.
`map_to_screen_from_marker(x, y)`	Transforma coordenadas de marcador a coordenadas de pantalla usando homografía.
`trigger_reset_warning()`	Activa advertencia de reseteo tras gol, deteniendo puck y jugadores.
`reset_puck()`	Reinicia el puck al centro del rink.
`reset_game()`	Reinicia marcador, temporizador, estado UI y puck.
`check_initial_positions()`	Verifica si los jugadores están en sus posiciones iniciales correctas; activa warnings si no.
`handle_reset_warning()`	Maneja la lógica del estado RESET_WARNING, reposicionando jugadores y puck según warnings.
`handle_victory_input()`	Gestiona entradas del usuario durante la pantalla de 'Continue?'; permite reiniciar o terminar juego.
`finish_game()`	Determina ganador, actualiza resultado y cambia estado UI a FINISHED.
`update(dt)`	Actualiza temporizador, físicas, puck, jugadores, y controla flujo según estado del juego (RUNNING, RESET_WARNING, FINISHED).
`draw()`	Dibuja fondo, scoreboard, puck, jugadores, rink (opcional debug), goals (debug) y overlays de UI.
`run()`	Loop principal del juego: gestiona eventos, READY/GO, actualización y renderizado continuo, incluyendo pantalla de victoria y pausa.

Nota. Resumen de funciones clave de la clase `Game`, incluyendo inicialización, física, control de jugadores mediante markers, UI, colisiones, goles y loop principal de juego.

10.10. Integración del sistema de proyección para la segunda validación completa del sistema

Figura 18. Secuencia de interacción física–virtual del juego de *air hockey* proyectado en el entorno Robotat



Nota. La figura presenta seis subimágenes que muestran la evolución temporal del sistema proyectado. En ellas se observa el movimiento del marcador que controla al jugador 1, mientras que el marcador correspondiente al jugador 2 permanece estático durante toda la secuencia. Entre las subimágenes 2 y 4 se distingue claramente el momento de colisión entre el jugador móvil y el disco virtual, así como el subsecuente cambio de trayectoria del *puck*.

Una vez validado el funcionamiento del juego en la computadora, se integró nuevamente con el sistema de proyección mediante uno de los puentes disponibles hacia el servidor OBS/GStreamer. Al proyectar sobre la mesa del Robotat la pantalla completa del juego, y utilizar dos marcadores físicos para controlar a los jugadores, se obtuvo una sincronización adecuada entre la posición real de los marcadores y su representación virtual, tal como se muestra en la Figura 18. Cabe mencionar que en las subimágenes 3 y 4 se aprecia un desfase momentáneo entre la posición del marcador activo y la de su jugador virtual, producto de la latencia presente en la red durante la validación. Asimismo, el segundo marcador muestra un leve desfase constante en su posición virtual, atribuible a una ligera falta de calibración en el código fuente al momento de las pruebas.

CAPÍTULO 11

Conclusiones

- La implementación del mosaico de streams demostró que es posible recibir, organizar y proyectar múltiples flujos de video en tiempo real con baja latencia, tolerancia a desconexiones y layouts dinámicos, constituyendo una base estable para la interacción inmersiva sobre la mesa Robotat.
- La aplicación de control del cursor validó la traducción precisa de un marker físico a coordenadas digitales, confirmando la correcta sincronización entre el sistema de captura y la proyección.
- La validación mediante el juego de Air Hockey comprobó la detección y respuesta de múltiples markers de manera simultánea, confirmando la viabilidad del sistema completo para aplicaciones de interacción multiusuario, experimentación tecnológica y entornos educativos.

CAPÍTULO 12

Recomendaciones

- Profundizar en la integración de algoritmos de inteligencia artificial y visión por computadora no solo para el reconocimiento de gestos y comandos, sino también para la interpretación de escenas y objetos dentro del entorno. Esto permitiría que los agentes robóticos actúen de manera más autónoma, estableciendo un ciclo continuo entre percepción, acción y proyección, y ampliando las posibilidades de interacción más allá del uso de marcadores físicos.
- Automatizar el proceso de calibración de coordenadas, actualmente realizado de forma manual, mediante técnicas de detección automática de patrones, estimación de poses y ajuste dinámico de parámetros. Esto reduciría errores operativos, simplificaría la puesta en marcha del sistema y facilitaría su despliegue en distintos entornos o configuraciones de hardware.
- Ampliar la infraestructura del servidor de proyección incorporando soporte de audio y canales de comunicación híbridos entre SRT y WebRTC, con el objetivo de permitir experiencias distribuidas de baja latencia. Esta mejora abriría la puerta a escenarios de realidad aumentada remota, colaboración a distancia y aplicaciones educativas o experimentales que requieran sincronía multimodal.

CAPÍTULO 13

Referencias

[1] D. Kim, H. Chae, Y. Kim, J. Choi, K.-H. Kim y D. Jo, «Real-Time Motion Adaptation with Spatial Perception for an Augmented Reality Character,» *Applied Sciences*, vol. 14, n.º 2, pág. 650, 2024. DOI: 10.3390/app14020650 dirección: https://doi.org/10.3390/app14020650

[2] A. Craig, *Understanding Augmented Reality: Concepts and Applications*. Morgan Kaufmann, 2013, ISBN: 9780240824109. dirección: https://books.google.com.gt/books?id=7_O5LaIC0SwC

[3] C. P. Montoya, «Robotat: un ecosistema robótico de captura de movimiento y comunicación inalámbrica,» Tesis de licenciatura, Tesis de mtría., Universidad del Valle de Guatemala, Guatemala, 2022. dirección: https://repositorio.uvg.edu.gt/handle/123456789/4250

[4] M. Field, Z. Pan, D. Stirling y F. Naghdy, «Human motion capture sensors and analysis in robotics,» *Industrial Robot*, vol. 38, n.º 2, págs. 163-171, 2011. DOI: 10.1108/01439911111106372 dirección: https://doi.org/10.1108/01439911111106372

[5] P. Barrera. «El Robotat, el hábitat donde interactúan los robots en el CIT.» Publicado en Actualidad UVG. dirección: https://noticias.uvg.edu.gt/el-robotat-el-habitat-donde-interactuan-los-robots-en-el-cit/

[6] C. F. A. Valdés, *Pruebas a escala de algoritmos básicos de visión de computadora y control para vehículos autónomos a escala*, Tesis de licenciatura, Guatemala, 2024. dirección: https://repositorio.uvg.edu.gt/xmlui/handle/123456789/5096

[7] L. P. G. Jurado, *Simulación y planificación de rutas para vehículos autónomos en entornos urbanos a escala inspirados en la Ciudad de Guatemala*, Tesis de licenciatura en Ingeniería Mecatrónica, Facultad de Ingeniería, Guatemala, 2024.

[8] M. Ojer et al., «Projection-Based Augmented Reality Assistance for Manual Electronic Component Assembly Processes,» *Applied Sciences*, vol. 10, n.º 3, 2020, ISSN: 2076-3417. DOI: 10.3390/app10030796 dirección: https://www.mdpi.com/2076-3417/10/3/796

[9] L. B. Tabrizi y M. Mahvash, «Augmented reality–guided neurosurgery: accuracy and intraoperative application of an image projection technique,» *Journal of Neurosurgery*, vol. 123, n.º 1, págs. 206-211, 2015. DOI: 10.3171/2014.9.JNS141001 dirección: https://thejns.org/view/journals/j-neurosurg/123/1/article-p206.xml

[10] J. Lee, Y. Kim, M.-H. Heo, D. Kim y B.-S. Shin, «Real-Time Projection-Based Augmented Reality System for Dynamic Objects in the Performing Arts,» *Symmetry*, vol. 7, n.º 1, págs. 182-192, 2015, ISSN: 2073-8994. DOI: 10.3390/sym7010182 dirección: https://www.mdpi.com/2073-8994/7/1/182

[11] H. Benko, R. Jota y A. Wilson, «MirageTable: freehand interaction on a projected augmented reality tabletop,» en *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ép. CHI '12, Austin, Texas, USA: Association for Computing Machinery, 2012, págs. 199-208, ISBN: 9781450310154. DOI: 10.1145/2207676.2207704 dirección: https://doi.org/10.1145/2207676.2207704

[12] Y. S. Chang, B. Nuernberger, B. Luan, T. Hollerer y J. O'Donovan, « Gesture-based augmented reality annotation,» en *2017 IEEE Virtual Reality (VR)*, Los Alamitos, CA, USA: IEEE Computer Society, mar. de 2017, págs. 469-470. DOI: 10.1109/VR.2017.7892383 dirección: https://doi.ieeecomputersociety.org/10.1109/VR.2017.7892383

[13] E. T. Inc., *Mastering MQTT: Your Ultimate Tutorial for MQTT*. China: EMQ Technologies Inc., 2023, Release R24-3, November 14, 2023.

[14] S. Security, *TCP vs UDP — How Are They Different?* https://cheapsslsecurity.com/blog/tcp-vs-udp-how-are-they-different/, Último acceso: 26 de noviembre de 2025, abr. de 2022.

[15] Jean-Baptiste, *SRT vs RTMP: Which one to choose for your livestream?* https://api.video/blog/video-trends/srt-vs-rtmp/, Último acceso: 26 de noviembre de 2025, mayo de 2024.

[16] K. Fall y W. Stevens, *TCP/IP Illustrated* (Addison-Wesley professional computing series v. 1). Addison-Wesley, 2012, ISBN: 9780321336316. dirección: https://books.google.com.gt/books?id=X-l9NX3iemAC

[17] Pinggy, *How to Set Up Port Forwarding – Even Behind CGNAT*, https://pinggy.io/blog/how_to_set_up_port_forwarding_even_behind_cgnat/, Último acceso: 26 de noviembre de 2025, ago. de 2025.

[18] OptiTrack Documentation, *Camera Placement*, https://docs.optitrack.com/hardware/camera-placement, [Accessed: 2025-11-26], 2025.

[19] OptiTrack Documentation, *Markers*, https://docs.optitrack.com/motive/markers, [Accessed: 2025-11-26], 2025.

[20] A. Kaehler y G. Bradski, *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*, 1st. Sebastopol, CA: O'Reilly Media, 2017, ISBN: 9781491937990.

[21] *Data Streaming – Motive*, Online documentation, Accessed: 2025-11-26, OptiTrack, 2025. dirección: https://docs.optitrack.com/motive/data-streaming

[22] «GStreamer Application Development Manual: Basics,» The GStreamer Project, visitado 26 de nov. de 2025. dirección: https://gstreamer.freedesktop.org/documentation/application-development/introduction/basics.html?gi-language=c

[23] «GtkContainer — GTK 3 Reference Manual,» GTK Project, visitado 26 de nov. de 2025. dirección: https://docs.gtk.org/gtk3/class.Container.html

[24] P. S. Foundation. «Módulo ctypes — Interfaz de llamada a funciones en bibliotecas compartidas.» Documentación oficial de Python, visitado 26 de nov. de 2025. dirección: https://docs.python.org/es/3.10/library/ctypes.html

[25] «SendInput function (winuser.h).» Documentación oficial de la API Win32 para síntesis de eventos de teclado y ratón, Microsoft, visitado 26 de nov. de 2025. dirección: https://learn.microsoft.com/es-es/windows/win32/api/winuser/nf-winuser-sendinput

[26] Python Software Foundation, *Tutorial de Python: Clases*, Consultado el 26 de noviembre de 2025, 2024. dirección: https://docs.python.org/es/3/tutorial/classes.html

[27] V. Blomqvist. «Pymunk — Physics library for Python: Overview.» Documentación oficial de Pymunk, visitado 26 de nov. de 2025. dirección: https://www.pymunk.org/en/latest/overview.html

[28] S. Jain. «Kinematic vs. Dynamic: Choosing the Right Model for Vehicle Simulations.» Artículo en LinkedIn, visitado 26 de nov. de 2025. dirección: https://www.linkedin.com/pulse/kinematic-vs-dynamic-choosing-right-model-vehicle-simulations-jain-g83gf

[29] dodgepong, *How to set up your own private RTMP server using nginx*, Guía publicada en el foro de OBS, mar. de 2014. dirección: https://obsproject.com/forum/resources/how-to-set-up-your-own-private-rtmp-server-using-nginx.50/

[30] E. Wu, *srt-live-server*, GitHub, Repositorio de código, 2025. dirección: https://github.com/Edward-Wu/srt-live-server

[31] E. Bassi. «Who Wrote GTK4.» Blog de desarrollo de GTK, visitado 26 de nov. de 2025. dirección: https://blogs.gnome.org/gtk/2020/12/17/who-wrote-gtk4/

[32] OpenAI. «ChatGPT (GPT-5.1).» Modelo de lenguaje de gran tamaño, visitado 26 de nov. de 2025. dirección: https://chat.openai.com/

CAPÍTULO 14

Anexos

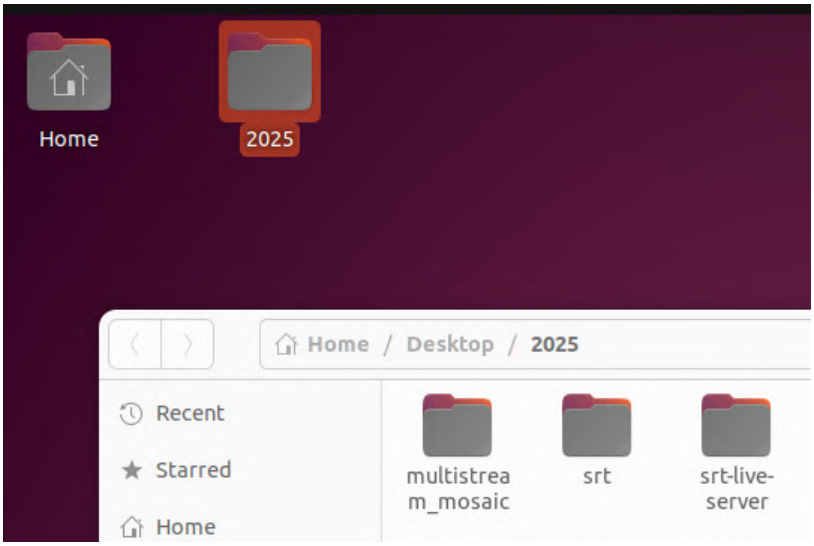
En esta sección se presenta el manual de usuario correspondiente a la metodología de instalación, configuración y uso tanto del servidor como del cliente, considerando los modos de transmisión SRT y UDP. La primera parte detalla el procedimiento para realizar el montaje físico del servidor de proyección, así como la instalación de las dependencias necesarias. La segunda parte describe la configuración del servidor, incluyendo la modificación y compilación de su código fuente, junto con la configuración de las herramientas cliente para establecer transmisiones exitosas. Además, se incluyen posibles fallos derivados de factores externos, como problemas de red o variaciones en la geometría de los marcadores. Finalmente, la tercera parte recopila consideraciones especiales que se deben tomar en cuenta al momento de hacer uso tanto del servidor como del cliente.

14.1. Montaje del servidor de proyección e instalación de dependencias

En caso de que la mini computadora que funciona como servidor del sistema Robotat no se encuentre instalada y operativa sobre la mesa de trabajo, será necesario realizar el montaje y cableado correspondiente. El servidor se conecta y opera igual que un mini PC convencional. El usuario configurado durante el desarrollo de este proyecto es `Robotat`, cuya contraseña se encuentra escrita manualmente en la cubierta de la caja donde normalmente se almacena.

En la parte trasera del servidor se debe conectar el cable HDMI proveniente del proyector, utilizando el único puerto disponible. Si el servidor va a operar sobre una red de tipo LAN, también debe conectarse un cable de red proveniente del router o proveedor de red en el puerto Ethernet posterior. Las dependencias de comunicación necesarias deberían encontrarse preinstaladas en el escritorio, dentro de la carpeta `2025`, cómo se observa en la Figura 19, bajo el nombre `SRT`.

Figura 19. Escritorio del servidor Robotat



Nota. Vista general del escritorio del servidor utilizado para el montaje del sistema de proyección.

En caso de que no estuvieran instaladas, o si la carpeta hubiese sido eliminada, se deben ejecutar los siguientes pasos:

1. Actualizar la lista de paquetes:

```
sudo apt update
```

2. Instalar herramientas de compilación y dependencias:

```
sudo apt install -y build-essential cmake pkg-config git libssl-dev
```

3. Clonar el repositorio oficial de SRT:

```
cd ~/Desktop/2025
git clone https://github.com/Haivision/srt.git
```

4. Crear carpeta de compilación y configurar el proyecto:

```
cd srt
mkdir build && cd build
cmake ..
```

5. Compilar e instalar:

```
make -j$(nproc)
sudo make install
sudo ldconfig
```

El servidor SRT utilizado en este proyecto fue originalmente desarrollado por Edward Wu. Si dicho servidor no se encuentra dentro de la carpeta `2025`, será necesario reinstalarlo desde una terminal ubicada en dicha ruta:

1. Actualizar repositorios:

```
sudo apt update
```

2. Clonar el repositorio:

```
cd ~/Desktop/2025
git clone https://github.com/Edward-Wu/srt-live-server.git
```

3. Configurar y compilar:

```
cd srt-live-server
make
```

14.2. Instalación de dependencias y herramientas del cliente

El cliente requiere al menos una herramienta capaz de transmitir su pantalla, una ventana o una fuente multimedia hacia el servidor de proyección, utilizando los protocolos de comunicación SRT o UDP. Para efectos prácticos, esta sección expone el proceso de instalación de *OBS Studio* y *GStreamer*, ambos utilizados como medios de transmisión durante las pruebas del sistema. No obstante, cualquier aplicación compatible con SRT o UDP puede emplearse como alternativa.

14.2.1. Instalación de OBS Studio en Windows

OBS Studio constituye una de las herramientas más utilizadas para la captura y transmisión de video en tiempo real. El procedimiento recomendado para su instalación en sistemas Windows utilizando el instalador oficial es el siguiente:

1. Acceder a la página oficial del proyecto: `https://obsproject.com/es/download`.

2. Seleccionar la opción **Descargar Instalador** para Windows.
3. Ejecutar el archivo descargado.
4. Avanzar por el asistente de instalación hasta llegar a la selección de la ruta de instalación.
5. Elegir la ruta deseada o dejar la opción predeterminada.
6. Hacer clic en **Instalar**.

Una vez completado, OBS Studio estará listo para configurarse como cliente transmisor mediante los protocolos SRT o UDP, según se requiera.

14.2.2. Instalación de GStreamer en Windows

En caso de que se desee utilizar GStreamer como herramienta de transmisión, el proceso de instalación en Windows es similar, aunque requiere atención a ciertos detalles relacionados con las versiones del instalador. Para instalar GStreamer en Windows se recomienda el siguiente procedimiento:

1. Acceder al sitio oficial: `https://gstreamer.freedesktop.org/download/#sources`.
2. Localizar la sección de instaladores para Windows.
3. Descargar el archivo denominado **MSVC 64-bit Installer** correspondiente a la versión estable más reciente.
4. Elegir entre los instaladores:
   a) **Runtime Installer**: contiene solo los binarios necesarios para ejecutar aplicaciones basadas en GStreamer, es decir, si solo se desea emitir una transmisión y nada más se puede usar este instalador
   b) **Development Installer**: incluye además archivos de desarrollo (cabeceras, librerías estáticas, pkg-config). Este instalador es necesario únicamente si se planea programar o compilar aplicaciones que utilicen GStreamer.
5. Ejecutar el instalador elegido.
6. En el tipo de instalación, seleccionar:
   - **Typical** si se utilizará únicamente para enviar video mediante pipelines básicos.
   - **Complete** si se desea acceso a todos los plugins disponibles.
7. Continuar el asistente hasta completar la instalación.
8. Agregar manualmente la ruta de GStreamer a la variable de entorno `PATH`, si el instalador no lo hace automáticamente.

Una vez instalado, el usuario podrá ejecutar pipelines de GStreamer en la terminal de Windows (`cmd` o PowerShell) para enviar flujos de video hacia el servidor de proyección mediante SRT o UDP.

14.2.3. Instalación de GStreamer en Linux

En sistemas Linux, GStreamer suele encontrarse disponible en los repositorios oficiales de la mayoría de distribuciones, lo cual simplifica considerablemente su instalación. Para instalar GStreamer en Ubuntu o distribuciones basadas en Debian se recomienda ejecutar:

1. Actualizar la lista de paquetes:

```
sudo apt update
```

2. Instalar los paquetes principales de GStreamer:

```
sudo apt install -y gstreamer1.0-tools gstreamer1.0-plugins-base \
gstreamer1.0-plugins-good gstreamer1.0-plugins-bad \
gstreamer1.0-plugins-ugly gstreamer1.0-libav
```

3. (Opcional) Instalar elementos de desarrollo si se requiere compilar aplicaciones:

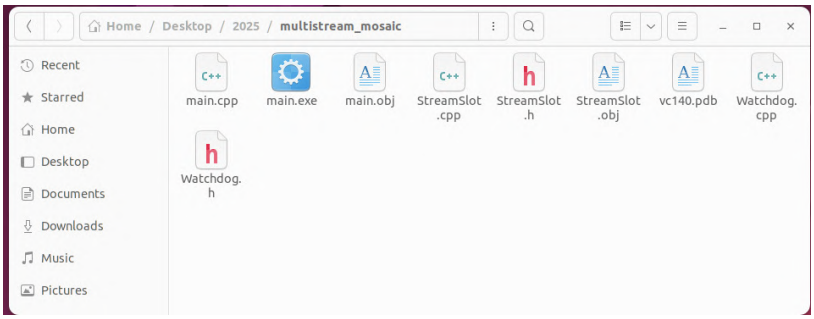
```
sudo apt install -y libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev
```

4. (Opcional) Verificar la instalación ejecutando:

```
gst-launch-1.0 --version
```

14.3. Configuración del servidor de proyección

Figura 20. Archivos de la aplicación mosaico ubicados en la carpeta del proyecto

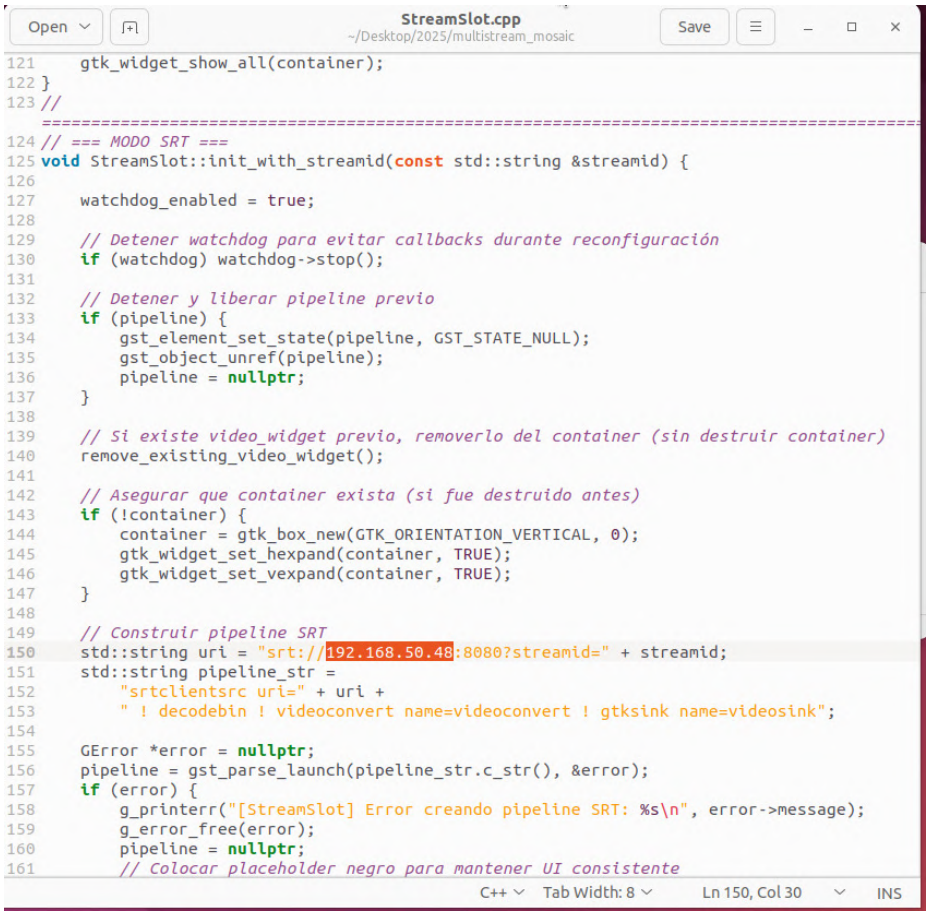


Nota. La carpeta 2025 contiene los ejecutables, código fuente y dependencias necesarias para el funcionamiento de la aplicación de mosaico de transmisiones.

La aplicación de mosaico de transmisiones se encuentra en el escritorio dentro de la carpeta `2025`, como se observa en la Figura 20. Cuando opera en modo SRT, la aplicación debe conectarse a la dirección IP del servidor SRT, el cual debe estar en ejecución antes de iniciar el mosaico. A menos que se modifique el archivo `sls.conf`, el servidor SRT se monta automáticamente sobre la red a la cual el mini PC esté conectado, ya sea Wi-Fi o LAN, utilizando la dirección IP asignada por dicho entorno.

14.3.1. Modificación de IP o ID de la aplicación mosaico de transmisiones para SRT

Figura 21. Fragmento del código responsable de la construcción del pipeline SRT



Nota. Esta sección del archivo `StreamSlot.cpp` define la dirección IP, puerto y *streamid* utilizado para cada conexión SRT.

Para modificar la dirección IP a la que se conecta la aplicación mosaico en modo SRT, debe accederse al archivo `StreamSlot.cpp` y localizar el siguiente bloque de código encargado de construir el pipeline SRT (el cual también se puede observar en la Figura 21):

```
// Construir pipeline SRT
std::string uri = "srt://172.23.193.99:8080?streamid=" + streamid;
std::string pipeline_str =
"srtclientsrc uri=" + uri +
" ! decodebin ! videoconvert name=videoconvert ! gtksink name=videosink";
```

Dentro de este fragmento es posible modificar:

- La dirección IP del servidor SRT;
- El puerto (valor por defecto: `8080`);
- La estructura completa del pipeline, en caso de requerir configuraciones específicas para latencia o decodificación.

Si se desea modificar los *stream IDs* utilizados por los clientes (por ejemplo, transmisiones enviadas desde OBS), es necesario editar el archivo `main.cpp` dentro del método `rebuild_pipelines`, donde se encuentra el siguiente arreglo:

```
std::vector<std::string> stream_ids = {
 "live.sls.com/live/stream1",
 "live.sls.com/live/stream2",
 "live.sls.com/live/stream3",
 "live.sls.com/live/stream4",
 "live.sls.com/live/stream5",
 "live.sls.com/live/stream6"
};
```

Cada elemento del vector corresponde a un espacio fijo dentro del mosaico, por lo que debe asignarse un *streamid* por transmisión.

14.3.2. Modificación de puertos o pipeline de la aplicación mosaico de transmisiones para UDP

En el modo UDP no es necesario especificar una IP de servidor, ya que cada cliente transmite directamente a un puerto del receptor. Para modificar los puertos utilizados, debe editarse el archivo `main.cpp` y ubicar el siguiente bloque dentro del método `rebuild_pipelines`:

```
std::vector<std::string> udp_ports = {
 "5000", "5001", "5002", "5003", "5004", "5005"
};
```

Si se desea ajustar el pipeline utilizado en modo UDP, puede hacerse desde el archivo `StreamSlot.cpp` en la sección identificada como `// MODO UDP`. Dependiendo de los requerimientos de latencia, pueden utilizarse pipelines optimizados como el siguiente:

```
gst-launch-1.0 -v udpsrc port=5000 buffer-size=200000 \
! application/x-rtp,media=video,encoding-name=H264,payload=96 \
! rtpjitterbuffer latency=20 drop-on-latency=false \
! rtph264depay ! h264parse ! avdec_h264 ! videoconvert ! autovideosink
```

El parámetro `drop-on-latency` puede configurarse en `true` para obtener menor latencia, aunque esto incrementa el riesgo de artefactos o fallas en caso de inestabilidad en la red, pudiendo provocar cierres inesperados de la aplicación. La barra invertida solo es un separador en este manual pero debe omitirse al utilizar cualquier pipeline de GStreamer.

14.3.3. Compilación de la aplicación desde la terminal

Después de realizar cualquier modificación en la dirección IP, puerto o pipeline, se deben guardar los cambios y abrir una terminal dentro de la carpeta que contiene los archivos fuente. Para compilar la aplicación y generar el ejecutable `main.exe`, se utiliza el siguiente comando (omitir barra invertida y dejar espacio en su lugar):

```
g++ main.cpp StreamSlot.cpp Watchdog.cpp -o main.exe \
$(pkg-config --cflags --libs gtk+-3.0 gstreamer-1.0 gstreamer-video-1.0)
```

El archivo ejecutable generado puede utilizarse posteriormente según las instrucciones establecidas en la sección correspondiente de este manual.

14.3.4. Consideraciones especiales para la red del servidor

Por defecto, la aplicación mosaico utiliza la dirección IP que el servidor recibió mediante DHCP al conectarse a la red utilizada durante las pruebas del sistema. Debido a que DHCP asigna direcciones dinámicas que pueden cambiar entre reinicios o reconexiones, es necesario actualizar la IP configurada en la aplicación cada vez que el mini PC se conecte a una red distinta, a menos que dicha IP se configure como una dirección estática.

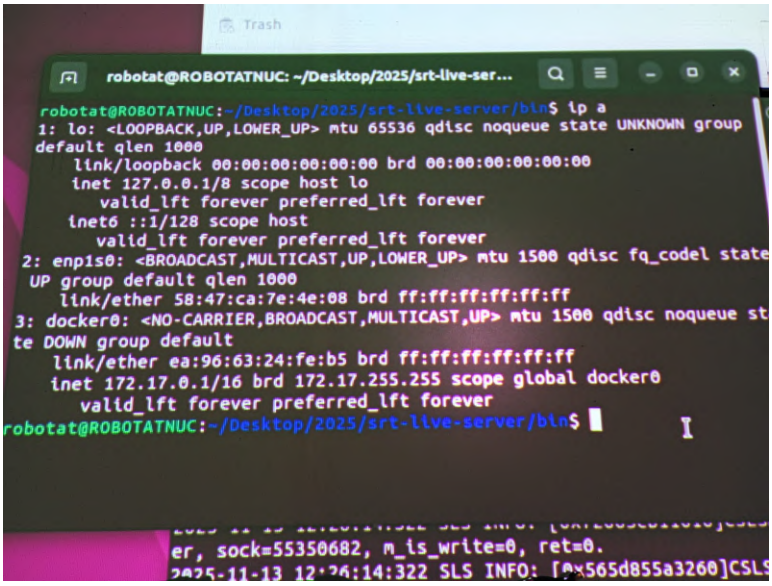
Para conocer la IP actual del servidor, puede ejecutarse el comando:

```
ip a
```

Debe tenerse en cuenta que si el servidor se encuentra en una red cuyos rangos pertenecen a direcciones privadas, podría no ser accesible desde clientes externos o desde OBS configurado fuera de dicha LAN. Las direcciones privadas abarcan:

- Clase A: 10.0.0.0 a 10.255.255.255
- Clase B: 172.16.0.0 a 172.31.255.255
- Clase C: 192.168.0.0 a 192.168.255.255

Figura 22. Dirección IP privada asignada al servidor en una red LAN

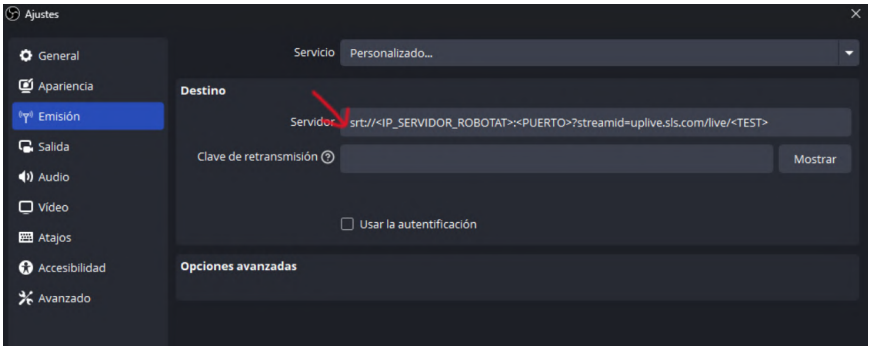


Nota. Ejemplo de dirección IP privada obtenida mediante el comando `ip a`, la IP corresponde a la dirección 172.17.0.1.

Durante las pruebas del sistema, se intentó conectar OBS Studio a una dirección perteneciente a estos rangos (Figura 22) y la transmisión nunca logró establecerse, debido a que el servidor se encontraba dentro de una red de área local no enrutable desde el exterior y, muy probablemente, el router carecía de una regla de *port forwarding* configurada para permitir el acceso externo.

14.4. Configuración del cliente para iniciar una transmisión

Figura 23. Selección del servicio personalizado en la configuración de emisión de OBS Studio



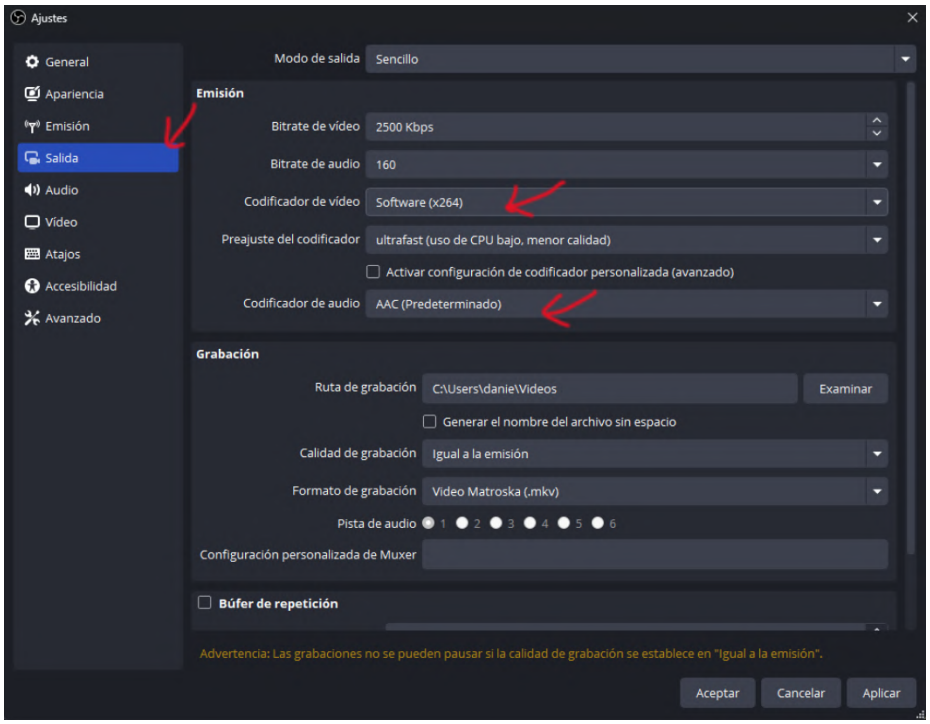
Nota. Para utilizar SRT con el servidor de proyección, es necesario configurar OBS Studio en modo de servicio personalizado.

Para iniciar una transmisión desde el cliente utilizando OBS Studio, se debe acceder al menú `Archivo` y seleccionar `Ajustes`. En la sección `Emisión`, elegir el modo *Servicio Personalizado* (véase Figura 23) y en el campo correspondiente al servidor ingresar la siguiente dirección:

```
srt://<IP_SERVIDOR_ROBOTAT>:<PUERTO>?streamid=uplive.sls.com/live/<ID>
```

En las redes Robotat y Robotat 5G, la dirección IP utilizada debe ser la del servidor de proyección, no la del servidor central del Robotat. Por lo tanto, el usuario debe consultar al administrador cuál es la IP asignada al mini PC del sistema de proyección y reemplazar el campo `<IP_SERVIDOR_ROBOTAT>`. De igual forma, el puerto del servicio SRT debe ser proporcionado por el administrador. No debe omitirse el carácter : antes del número de puerto. Finalmente, el administrador debe asignar un identificador único para el campo `<ID>` (ubicado después de `live/`) para evitar colisiones o transmisiones duplicadas.

Figura 24. Ajustes de salida para transmisión mediante SRT en OBS Studio



Nota. Tanto en modo sencillo como avanzado, es importante seleccionar el codificador `x264` para evitar conflictos con el servidor SRT.

Posteriormente, en la sección `Salida`, deben configurarse los siguientes parámetros (modo sencillo):

- **Codificador de video:** `Software (x264)`
- **Codificador de audio:** `AAC` (si no estuviera seleccionado por defecto)

En caso de utilizar el modo avanzado, deben mantenerse las mismas opciones: codificador `x264` para video y `AAC` para audio, tal como se presenta en la Figura 24. En situaciones donde exista un error de conexión o incompatibilidad, puede intentarse el uso de `Hardware (H.264)`, aunque esto podría generar conflictos en el servidor SRT, tales como duplicación de transmisiones. Las configuraciones adicionales pueden ajustarse según la calidad deseada, tomando en cuenta que una mayor calidad aumenta la latencia.

14.5. Configuración del cliente para modificar la calibración del segundo plano de detección

Como se explicó previamente en este documento, cuando la pantalla proyectada cambia su distribución—por ejemplo, al dividirse en múltiples transmisiones—es necesario recalibrar el plano de detección para que corresponda con el nuevo tamaño de pantalla. Cada aplicación de validación (control del mouse y juego de airhockey) contiene métodos que aplican esta calibración mediante parámetros recibidos por MQTT.

En la aplicación del juego de airhockey, la configuración MQTT se encuentra en el archivo `game.py` dentro de la clase `Game`:

```
# ================================================================
# --- CONFIGURACIÓN MQTT ---
# ================================================================
BROKER = "192.168.50.200"
PORT = 1880
TOPIC = "mocap/all"
TARGET_ID = "69"
```

Los marcadores utilizados por la aplicación se procesan dentro del método `mqtt_on_message`. La asignación de cada marcador a su jugador correspondiente está definida mediante condiciones como las siguientes:

```
if identifier == "65":  # Player 1
   self.player1_pos = (x_screen, y_screen)
   self.new_data_p1 = True
elif identifier == "69":  # Player 2
   self.player2_pos = (x_screen, y_screen)
   self.new_data_p2 = True
```

14.5.1. Homografía entre el segundo y el tercer plano

Para proyectar correctamente las posiciones detectadas dentro del entorno de juego, se aplica una transformación por homografía entre el segundo plano (plano calibrado en Robotat) y el tercer plano (pantalla de juego renderizada por Pygame). Esta configuración se encuentra en `game.py` en el siguiente bloque:

```
# ============================================================
# --- CONFIGURACIÓN DE PANTALLA ---
# ============================================================
self.screen_w, self.screen_h = self.screen.get_size()

# ============================================================
# --- DEFINIR PLANOS Y HOMOGRAFÍA ---
# ============================================================
A = np.array([-853, -1583])  # Inferior derecha en proyección
B = np.array([854, -1583])   # Superior derecha en proyección
C = np.array([854, 1475])    # Superior izquierda en proyección
D = np.array([-853, 1475])   # Inferior izquierda en proyección
original = np.array([A, B, C, D, A])  # <-- Primer Plano

width = 870    # mm
height = 1696  # mm
screen_w, screen_h = self.screen_w, self.screen_h  # <-- Tercer Plano

...

center = np.array([4, 41])  # <-- Posición del segundo plano (mm)
scale = 1
angle_deg = 180
theta = np.deg2rad(angle_deg)
```

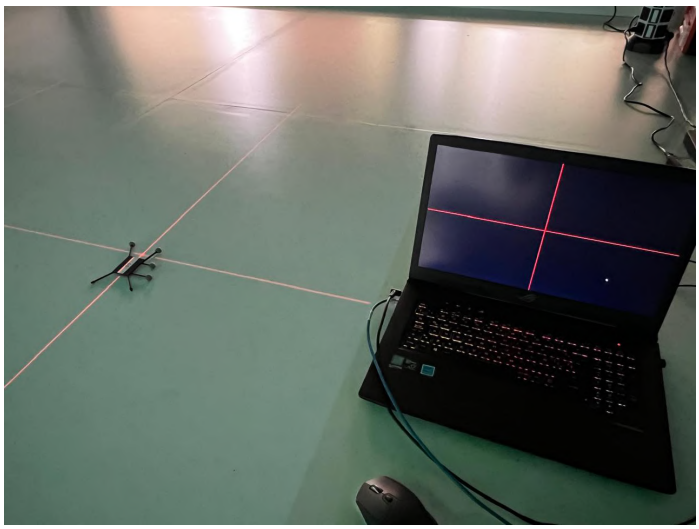
Los valores de `width` y `height` están expresados en milímetros y corresponden a las dimensiones físicas de la sección proyectada sobre la superficie Robotat. Estos deben medirse con una cinta métrica cada vez que se modifique la división en pantallas.

La variable `center` representa la posición del centro del segundo plano, también en milímetros. Para obtener este valor, se recomienda colocar un marcador en el centro exacto de la pantalla proyectada. Para simplificar este proceso, puede proyectarse una imagen guía con líneas que marquen el centro, tal como se muestra en la Figura 25.

Es importante considerar que el punto físico que el sistema utiliza como referencia en el marcador no siempre coincide con el centro geométrico visible del marcador, lo cual puede introducir desplazamientos si no se corrige adecuadamente.

En caso de que se requiera ajustar la velocidad del disco en el juego de *air hockey*, se debe modificar directamente los parámetros físicos definidos en las clases `Player` y `Puck`. Entre dichos parámetros se encuentran la elasticidad, la fricción y los coeficientes de corrección de impulso. Ajustarlos permite modificar el comportamiento dinámico del disco y la sensibilidad a colisiones.

Figura 25. Imagen proyectada utilizada para la calibración del segundo plano



Nota. Esta imagen permite identificar el centro exacto del área proyectada a partir de líneas guía. Esto facilita determinar la posición del marcador utilizado para ajustar la homografía entre planos.

14.6. Manejo del servidor de proyección y de la aplicación mosaico de transmisiones

14.6.1. Comandos para ejecutar el servidor SRT y la aplicación mosaico

Una vez instalado el servidor SRT y conectado el equipo a la red correspondiente, se debe abrir una terminal dentro de la carpeta `bin` del proyecto `srt-live-server` y ejecutar el siguiente comando:

```
./sls -c ../sls.conf
```

Posteriormente, una vez compilada la aplicación del mosaico de transmisiones, se debe abrir una terminal dentro de la carpeta `multistream_mosaic` y ejecutar:

```
./main.exe
```

Es importante mencionar que la aplicación de mosaico no requiere que el servidor SRT esté en funcionamiento si los clientes utilizan exclusivamente transmisiones por UDP mediante GStreamer.

14.6.2. Teclas para el manejo de la aplicación mosaico de transmisiones

Una vez la aplicación esté ejecutándose en la mini computadora del servidor de proyección, es posible manipular la distribución de transmisiones y los modos de funcionamiento mediante un teclado conectado al dispositivo. Es importante considerar que, en algunas ocasiones, se debe hacer clic sobre la ventana del mosaico para que los eventos de teclado sean recibidos correctamente, en lugar de permanecer con el foco en la terminal.

La aplicación responde a los siguientes comandos:

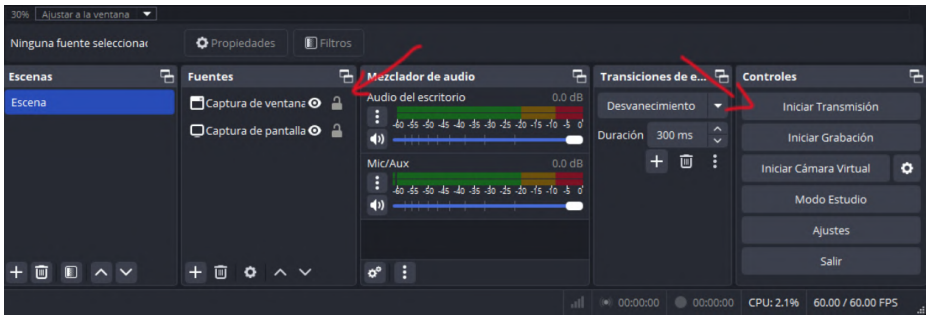
- **L** — Activa o desactiva el bloqueo del *layout*.
- **V** — Activa el modo `UDP_SAFE` (latencia normal, mayor estabilidad).
- **U** — Activa el modo `UDP_FAST` (latencia ultra baja).
- **M** — Activa el modo `SRT_MOSAIC`.
- **F11** — Alterna entre modo de pantalla completa y modo ventana.
- **1–6** — Cambia la cantidad de espacios activos en el mosaico (solo si el *layout* no está bloqueado).

Si se desea modificar, agregar o eliminar eventos de teclado se debe modificar el método `on_key_press` del archivo `main.cpp` y posteriormente es necesario compilar la aplicación nuevamente como se especifica en la sección 14.3.3 del presente manual.

14.7. Manejo del cliente para transmisiones y control de las aplicaciones de validación

14.7.1. Envío de transmisiones por parte del cliente (SRT y UDP)

Figura 26. Configuración de transmisión en OBS Studio



Nota. Desde OBS Studio únicamente es necesario seleccionar la fuente y presionar "Iniciar transmisión" para enviar video mediante SRT al servidor.

En el caso de comunicación mediante SRT utilizando OBS Studio, basta con iniciar la transmisión una vez configurado el servidor y el *stream ID* como se observa en la Figura 26.

Para el caso de transmisión por UDP con GStreamer, desde la terminal del cliente debe ejecutarse el siguiente pipeline. Es importante seleccionar correctamente la dirección IP del equipo que ejecuta la aplicación mosaico y un puerto libre (omitir barra invertida y dejar espacio en su lugar):

```
gst-launch-1.0 -v d3d11screencapturesrc ! \
video/x-raw,framerate=60/1 ! queue ! videoconvert ! \
video/x-raw,format=NV12 ! \
nvh264enc rc-mode=cbr bitrate=4000 zerolatency=true ! \
h264parse config-interval=-1 ! \
rtph264pay config-interval=1 pt=96 ! \
udpsink host=192.168.72.32 port=5000
```

14.7.2. Teclas para el control de las aplicaciones de validación: control del cursor y juego de Air Hockey

Las aplicaciones de validación desarrolladas para este proyecto (control del cursor y juego de *Air Hockey*) utilizan atajos de teclado para gestionar estados de ejecución, depuración y reinicio del sistema. A continuación se listan dichos comandos:

Control del cursor

- **ESC (mantener 5 segundos)** — Detiene la ejecución de la aplicación.

Si se desea modificar, agregar o eliminar eventos asociados al control del cursor, se debe editar la función `esc_monitor()` o el bloque principal del archivo `MouseControl2.py` correspondiente a esta aplicación, y posteriormente ejecutar nuevamente el script.

Juego de Air Hockey

- **Q** — Agrega un punto al jugador 1 (utilizado para depuración).
- **W** — Agrega un punto al jugador 2 (utilizado para depuración).
- **D** — Activa o desactiva el modo de depuración visual.
- **ESC** — Pausa o reanuda el juego.
- **R** — Reinicia la partida, reestablece posiciones y estados del sistema.

Si se desea modificar, agregar o eliminar eventos del juego, se debe modificar el método `run()` de la clase `Game` dentro del archivo `game.py`. Para que los cambios tengan efecto basta con guardar el archivo y ejecutar nuevamente la aplicación.