Resumen

OBS es un programa open-source que sirve para grabar vídeo o emitir en directo. Actualmente, OBS es el software utilizado por casi todos los usuarios de Twitch para configurar sus emisiones (streamings online). OBS permite mostrar el chat de Twitch en pantalla, mostrar la webcam del usuario (con algún marco u otros recursos visuales), gestionar transiciones o movimientos de cámara, mostrar avisos por suscripciones o donaciones, reproducir música, etc. Es decir, absolutamente todo lo que los usuarios ven en directo se gestiona a través de OBS. Este software, además, presenta muchos recursos en su programa base, pero ofrece mecanismos de extensión funcional a través de plugins. En este proyecto se propone desarrollar una extensión de OBS (a través de un plugin) que ofrezca un recurso visual extra para aquellos usuarios que emiten videojuegos en Twitch (los cuales suman el 83.5% de emisiones según twitchtracker [45]). Este recurso consiste en un marco de webcam cuyo diseño cambia según varía la barra de vida del jugador. Dado que el marco de webcam es uno de los elementos estéticos más populares en los directos y los videojuegos suponen la categoría más emitida en Twitch, este proyecto propone una mejora estética que relacione el marco de webcam con el juego que se está emitiendo y, además, aporte más dinamismo al directo. En la actualidad, existe una solución ya desarrollada por un usuario de la comunidad, pero que utiliza de manera privada en sus retransmisiones. Uno de los objetivos de este proyecto es desarrollar una solución funcional que esté accesible para toda la comunidad, y que sea configurable (en la medida de lo posible) para adaptarse a diferentes juegos y requisitos de emisiones en streaming. En el proyecto se analizarán diferentes estrategias para diseñar y desarrollar una solución a este problema, y se construirá el plugin con el que extender OBS. Se espera obtener un producto con la calidad suficiente para ser distribuible y usable por usuarios finales. Inicialmente, el proyecto se prototipará sobre un videojuego concreto (Apex Legends). Sin embargo, se utilizará como base para poder ser aplicado a otros juegos, pudiendo reconocer la barra de vida de cualquier juego, determinar la cantidad de vida que presenta y mostrar un marco de webcam que refleje esta cantidad. Como objetivo alternativo, y dada la escasa documentación existente sobre el desarrollo de plugins OBS, se espera que este proyecto sirva como referente y como guía a otros desarrolladores que quieran abordar la construcción de plugins para OBS.

Palabras clave: OBS, Twitch, desarrollo basado en componentes, integración de aplicaciones.

Abstract

OBS is an open-source application for video recording and live streaming. Currently, OBS is the software that is used by almost every Twitch user to configure their broadcasts (online streaming). OBS allows users to show the Twitch chat on-screen, to show the user's webcam (with a webcam frame or other visual resources), to manage transitions or camera movements, to show subscription or donation alerts, to play music, etc. In other words, absolutely everything that users watch live is managed with OBS. This software also has many resources itself, but it

offers functional extension mechanisms through plugins. In this project, it is proposed to develop an OBS extension (by creating a plugin) that offers an extra visual resource for those users who broadcast video games on Twitch (who add up to 83.5% of broadcasts according to twitchtracker [45]). This resource consists of a webcam frame whose design changes depending on the player's life bar. Given that the webcam frame is one of the most popular aesthetic elements on streamings and video games are the most streamed category on Twitch, this project proposes an aesthetic improvement that relates the webcam frame to the game that is being streamed and, in addition, it will add more dynamism to the streaming itself. Nowadays, there is a solution already developed by a community user, but he uses it privately at his broadcasts. One of the goals of this project is to develop a functional solution that is accessible to the entire community, and configurable (as much as possible) to suit different game and streaming requirements. In the project, different strategies will be analysed in order to design and develop a solution to this problem, and the plugin which extends OBS will be built. It is expected to obtain a product of sufficient quality to be distributable and usable by users. Initially, the project will be prototyped to process a specific video game (Apex Legends). However, it will be used as a basis to be applied to other games, being able to recognize the life bar of any game, to determine the amount of life it has and to show a webcam frame that reflects this amount. As an alternative objective, and given the scarce existing documentation on the development of OBS plugins, it is hoped that this project will serve as a reference and guide for other developers who want to tackle the development of OBS plugins.

Keywords : OBS, Twitch, component-based development, application integration.

Tabla de contenidos

1 Introducción

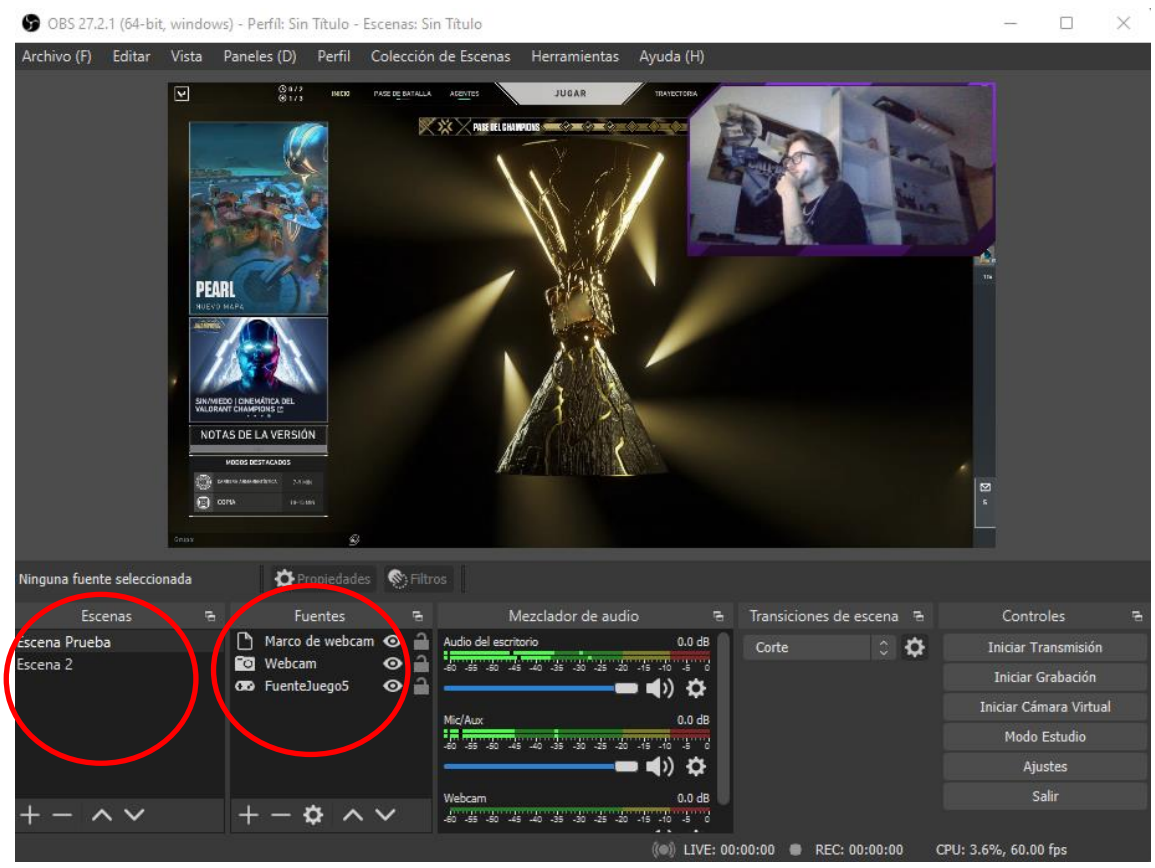
1.1 Motivación

Como muchos alumnos de la ETSINF, soy una persona a la que le encantan los videojuegos y ver a otros disfrutar de ellos. En concreto, adoro ver *streamings* de videojuegos en Twitch.

Twitch tiene 30 millones de usuarios diariamente activos y 9.2 millones de *streamers* mensualmente activos [1]. Es, también, la cuarta red social más usada entre jugadores de videojuegos [2]. Lo que quiero decir con estos datos es que esta plataforma es un gigante en el mundo del *streaming* de videojuegos.

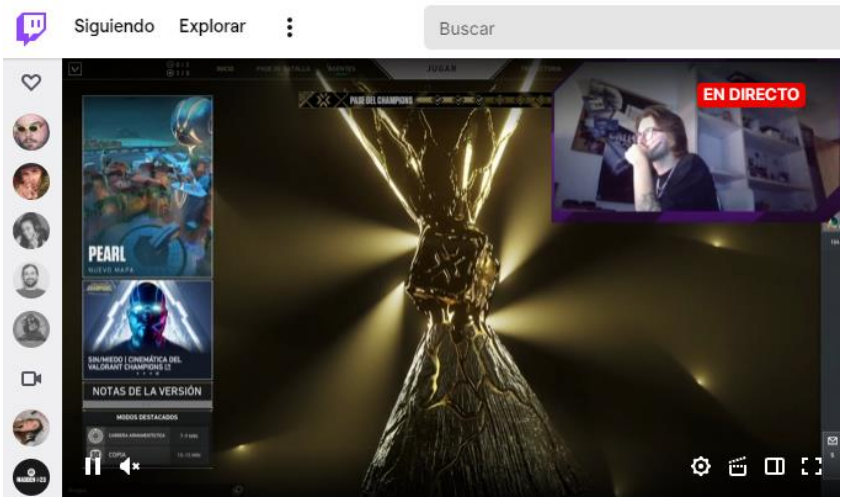
Un aspecto que me encanta de este mundo es OBS (*Open Broadcaster Software*), el cual es un programa de código abierto utilizado para configurar un *streaming* en Twitch.

Ilustración 1: Interfaz de usuario de OBS



En el programa OBS, el usuario asigna las distintas configuraciones de su directo a distintas Escenas. En cada una de estas Escenas, podemos añadir y eliminar Fuentes, las cuales se corresponden a los elementos que vemos en pantalla. En la imagen anterior, observamos 3 Fuentes: un marco para la *webcam*, la misma *webcam* del jugador y la imagen del juego. La disposición de estas Fuentes se encuentra en una Escena llamada “Escena Prueba”.

Ilustración 2: Streaming en Twitch



Aquí se muestra la emisión en directo configurada en la imagen anterior.

No he podido encontrar datos que me permitan expresar lo mucho que OBS supone en la comunidad de Twitch. Únicamente puedo decir que este programa lo podemos encontrar siempre en los *TOP Software* para hacer *streaming* y la propia plataforma de Twitch lo recomienda [3].

Mi afición por OBS y la emisión de videojuegos en directo la comparto con mi amigo Matías Patiño, el cual me planteó desarrollar el proyecto del que se trata esta memoria. El señor Patiño es un diseñador gráfico que ha realizado proyectos para *streamers* como Flexinja y Grimm [4].

El 14 de octubre de 2021, lo recuerdo porque aún tengo guardada la conversación, Matías me envió un vídeo en el que un *streamer* de Fortnite utilizaba un marco de *webcam* que cambiaba según su vida en dicho juego. Esto me pareció increíble, ya que conozco lo suficiente las APIs de videojuegos para afirmar que la vida del jugador no es un parámetro que te faciliten. Además, me impresionó porque el marco estaba siendo utilizado en OBS y dicho programa, antes de terminar este proyecto, me resultaba complicado de entender en cuanto a programación.

A partir del descubrimiento de este vídeo, empezamos a plantearnos cómo funcionaba este dinámico marco de *webcam*. Empezamos a imaginarnos maneras de mejorarlo, de tal forma que no solo creásemos una nueva versión, sino una versión que podrían personalizar los usuarios y reversionar otros desarrolladores. Nuestro deseo, como usuarios de Twitch, es ver al máximo número de personas utilizando esta *feature*, no solo a un único *streamer*.

Asimismo, estaba especialmente interesado en este proyecto ya que quería iniciarme en *open-source* y OBS resultaba un candidato ideal.

Finalmente, Matías y yo nos propusimos crear y hacer pública una extensión de OBS que permitiese añadir un marco de *webcam* el cual cambiase según la vida de

cualquier videojuego. Para el desarrollo del proyecto, Matías diseñaría la parte visual y yo diseñaría la funcional, la cual se encuentra explicada en esta memoria.

Inicialmente, no consideré este proyecto como mi TFG. En su lugar, hablé con un profesor de la facultad para desarrollar uno de sus proyectos, pero uno de mis compañeros me dijo que la extensión de OBS sí que podría ser material de TFG. Así que, dado que deseaba aprender el desarrollo en OBS y que en septiembre ya no tendré tanto tiempo para dedicárselo a proyectos propios debido a mi inicio en la vida laboral, decidí que mi último trabajo en la universidad fuese algo que haya nacido de mi propia iniciativa.

1.2 Objetivos

El objetivo de este TFG es desarrollar una extensión de OBS que añada una nueva Fuente al programa. Dicha Fuente será un marco de *webcam* cuyo color y animación reflejará la cantidad de vida que el jugador presente en el videojuego que esté disfrutando.

Nuestro producto debe ser utilizable en *streaming*, dado que es el escenario para el cual se está creando, y permitir la personalización del mismo por parte del usuario.

1.3 Impacto esperado

Se espera que el producto resultante de este proyecto sea una *feature* muy presente en la comunidad de Twitch, como son Move Transition y Source Record [5, 6]. Las cuales no son un estándar para *streaming*, pero sí que son recomendadas por varios *streamers*.

Es de esperar también que, dado el interés de la comunidad de Twitch por tener más dinamismo en los directos, los usuarios creen y vendan sus paquetes de marcos de *webcam* para añadir a este producto. Al igual que sucede con los logos y animaciones que se pueden comprar en diferentes mercados online.

Finalmente, esperamos que el código del proyecto sirva como referencia para otros desarrolladores que deseen crear productos similares y para programadores que quieran iniciarse en la comunidad de desarrollo de OBS.

1.4 Estructura

Estado del arte: Se expondrá cuál es el contexto de desarrollo del producto y qué otros productos similares existen en este momento.

Análisis del problema: Se analizarán los requisitos y riesgos del sistema a desarrollar, así como su dominio y contexto.

Diseño de la solución: Atendiendo a los requisitos expuestos en el apartado anterior, se redactará el diseño de la solución propuesta.

Desarrollo de la solución: Se presentarán en detalle la implementación de la solución propuesta y las decisiones que se tomaron durante su proceso de desarrollo.

Implantación: Se detallarán las tareas a realizar para preparar un entorno donde trabajar con el código del sistema, si se desea, y las tareas para instalar el producto.

Pruebas: Se evaluará que el sistema funcione correctamente a partir de una serie de pruebas.

Conclusiones: Se presentará, finalmente, una valoración del desarrollo de este proyecto y sus resultados.

1.5 Colaboraciones

Este proyecto ha sido desarrollado por mí con la guía de mi tutor, Joan Josep Fons Cors. En una ocasión, me reuní con el profesor Alberto Sanchis Navarro para consultar unas dudas en relación a este trabajo.

2 Estado del arte

2.1 Contexto de desarrollo

Antes de hablar del estado del arte de nuestro proyecto en concreto, es importante que conozcamos el contexto de la comunidad de desarrollo de OBS.

Actualmente, la comunidad de OBS presenta 204,635 miembros en Discord [7] y su proyecto principal, obs-studio [8], tiene un total de 520 colaboradores. Este proyecto es público y permite que los usuarios lo modifiquen mediante la solicitud y aprobación de *pull requests*. No obstante, editar el código de OBS no es la única manera de aportar una nueva experiencia de usuario. Los usuarios pueden instalar extensiones que cambien el comportamiento del programa o añadan nuevas *features*. Estas extensiones pueden ser creadas y publicadas en la página oficial de OBS por cualquier persona [9].

Pese a que podamos encontrar 695 extensiones desarrolladas y publicadas, el desarrollo en OBS tiene un problema de incertidumbre. En este momento, no encontramos ni reglas ni estándares de desarrollo, por lo que estos proyectos son realmente distintos unos de otros. Este fenómeno lo podemos apreciar al consultar el código de las extensiones ya publicadas en la página oficial.

Además, sumado a este problema, las guías y la documentación de OBS son insuficientes. Esta afirmación la declaramos en base a que la comunidad de desarrolladores de OBS facilita una guía de instalación de los recursos necesarios para crear un plugin [10], pero no una para su proceso de desarrollo.

En el caso de la documentación [11], resulta insuficiente ya que hay recursos que no vamos a encontrar explicados en la misma, como pueden ser algunas de las funciones de Gráficos utilizadas en este proyecto: `gs_texrender_reset`, `gs_texrender_begin`, `gs_texrender_end`, `gs_texrender_get_texture`, etc.

Sin embargo, pese a esta falta de documentación, los desarrolladores de OBS se encuentran disponibles en su servidor de Discord, donde resuelven las dudas de los usuarios a diario.

2.2 Crítica al estado del arte

En este apartado, vamos a exponer el estado del arte actual de nuestro proyecto y cuáles son los problemas que se presentan en el mismo.

Muchos creadores de contenido (*youtubers, streamers*) han desarrollado proyectos cuyo objetivo es aportar más dinamismo a sus vídeos o *streamings*. Proyectos los cuales consisten en generar acciones visuales provocadas por *in-game events*. Uno de los creadores que más tiempo ha dedicado a este tipo de programas es Michael Reeves, usuario de YouTube y Twitch que ha creado toda clase de programas que reaccionan a eventos. Uno de los que más destacaron fue un arma de *airsoft* con *bluetooth* que disparaba al jugador cuando perdía vida en Fortnite.

Como no hay manera de obtener la vida del jugador a través de Fortnite-API, Reeves quería utilizar un *proxy* que leyese el tráfico entre los servidores de Fortnite y el juego en sí. El problema de esta solución es que estos datos están encriptados y no es posible obtener el valor de la vida a través de ellos.

La solución que finalmente ejecutó fue crear un programa en Python que, funcionando en local, captura la imagen de su ordenador y, utilizando una librería de OCR, lee los valores de la vida en Fortnite.

La idea de Reeves se convirtió en un vídeo de 10,577,677 visualizaciones [12]. Asimismo, motivó a otros creadores de contenido a implementar *features* similares en sus directos. Una de las *features* que destacó y la cual motivó el desarrollo de este proyecto fue la presentada por LachlanYT el 7 de enero de 2021.

LachlanYT contrató los servicios de Point Zero, una agencia de diseño gráfico centrada en el entorno de los videojuegos y que forma parte de la compañía Visual By Impulse. Como resultado de esta colaboración, Point Zero desarrolló el Project Lachlan, cuya pieza principal consiste en *"the world's first reactive webcam"* [13, 14].

El *reactive webcam* es una característica estética que Lachlan añadió en sus directos. Se trata de un marco de *webcam* que cambia visualmente cuando la vida del jugador varía. Este proyecto es una extensión de OBS.

El problema de Project Lachlan es que únicamente funciona con Fortnite y su código es privado, solo Lachlan puede disfrutar de esta prestación. Ante la pregunta de los usuarios que desean saber cómo podrían crear una herramienta similar para sus directos, Point Zero declara lo siguiente: *"The secret to Lachlan's facecam is a mixture of computer vision, machine learning and proprietary coding"*.

2.3 Propuesta

Ante el problema de que la prestación de Lachlan sea privada, solo procese un único juego y que su código no se haya publicado como ocurre con el resto de extensiones de OBS, se decidió llevar a cabo este TFG. Este proyecto puede verse, pues, como una mejora de otro ya existente. En su esencia, queremos que nuestro producto sea como el utilizado por Lachlan: un marco de *webcam* que cambie según lo hace la vida del jugador. No obstante, deseamos que sea distinto en el sentido de que suponga una mejor experiencia para el usuario.

Proponemos desarrollar un producto cuyos marcos mostrados van a poder ser establecidos por el usuario, de esta manera el jugador puede cambiar completamente el aspecto visual. Este proyecto, al cual hemos bautizado como REACTIVE FACECAM PLUGIN, se va a caracterizar también porque va a poder procesar múltiples juegos y va a ser escalable, ya que deseamos facilitar que otros desarrolladores puedan expandirlo añadiendo más juegos con los que esta extensión funcione.

Además, proponemos que tanto este producto como el desarrollo del mismo sean accesibles para cualquier usuario. Alcanzaremos este objetivo publicando el código en su lanzamiento y redactando, en este documento, los aspectos más importantes de su proceso de desarrollo.

3 Análisis del problema

El propósito de este apartado es definir más detalladamente el proyecto a desarrollar mediante la presentación de los requisitos del sistema, los actores que interactuarán con el mismo y los riesgos que tiene su desarrollo. Asimismo, se expondrá el dominio y el contexto del problema utilizando diagramas UML y, finalmente, las soluciones propuestas para desarrollar el producto.

3.1 Especificación de requisitos

Los requisitos de nuestro sistema han sido determinados a partir de las prestaciones que actualmente ofrecen las extensiones de OBS, las expectativas de usuarios que consumen contenido de Twitch y mi propia experiencia como usuario tanto de OBS como de Twitch.

Con el fin de identificar los actores y los requisitos de nuestro sistema, hemos utilizado la técnica personas para identificar *stakeholders*.

3.1.1 Actores

Los actores van a representar los roles que van a tener los diferentes usuarios de nuestro sistema. Dichos actores interactuarán con el sistema a partir de los casos de uso que se expondrán más adelante.

- Llamaremos **Usuario OBS** a aquel que utilice Reactive Facecam Plugin como Fuente en OBS.
- Llamaremos **Usuario API** a aquel que únicamente utilice la API externa desarrollada en este proyecto.

3.1.2 Requisitos

Los requisitos del producto a desarrollar los podemos dividir en dos categorías: No Funcionales y Funcionales.

3.1.2.1 Requisitos No Funcionales

Los requisitos no funcionales van a especificar las características de funcionamiento del sistema.

- Tiempos de respuesta bajos: El marco debe reflejar la cantidad de vida en cada instante. Por lo tanto, la API debe procesar la captura de juego y enviar una respuesta en el menor tiempo posible.

- Interoperabilidad: la API debe poder interactuar no solo con el producto desarrollado, sino con cualquier sistema que desee acceder a ella.

- Escalabilidad: el sistema debe tener la capacidad de ampliarse para procesar más juegos.

- Disponibilidad: el sistema debe continuar funcionando en caso de fallo para no comprometer la calidad del *streaming*.

- Facilidad de uso: el sistema debe presentar una interfaz amigable y fácil de comprender para el usuario.

- Confiabilidad: el sistema debe ser capaz de funcionar bajo condiciones críticas, como puede ser un ordenador con tarjeta gráfica media-baja o atendiendo solicitudes de cientos de usuarios.

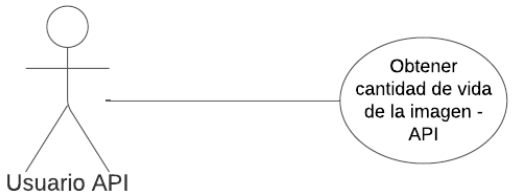
- Mantenibilidad: debemos poder aplicar modificaciones, correcciones y mejoras al sistema con facilidad.

- Eficiencia: el sistema debe funcionar correctamente usando los recursos mínimos.

3.1.2.2 Requisitos Funcionales

En contraposición, los requisitos funcionales van a definir los comportamientos específicos del sistema. Estos requisitos son los casos de uso que determinan la interacción de los actores con nuestro producto.

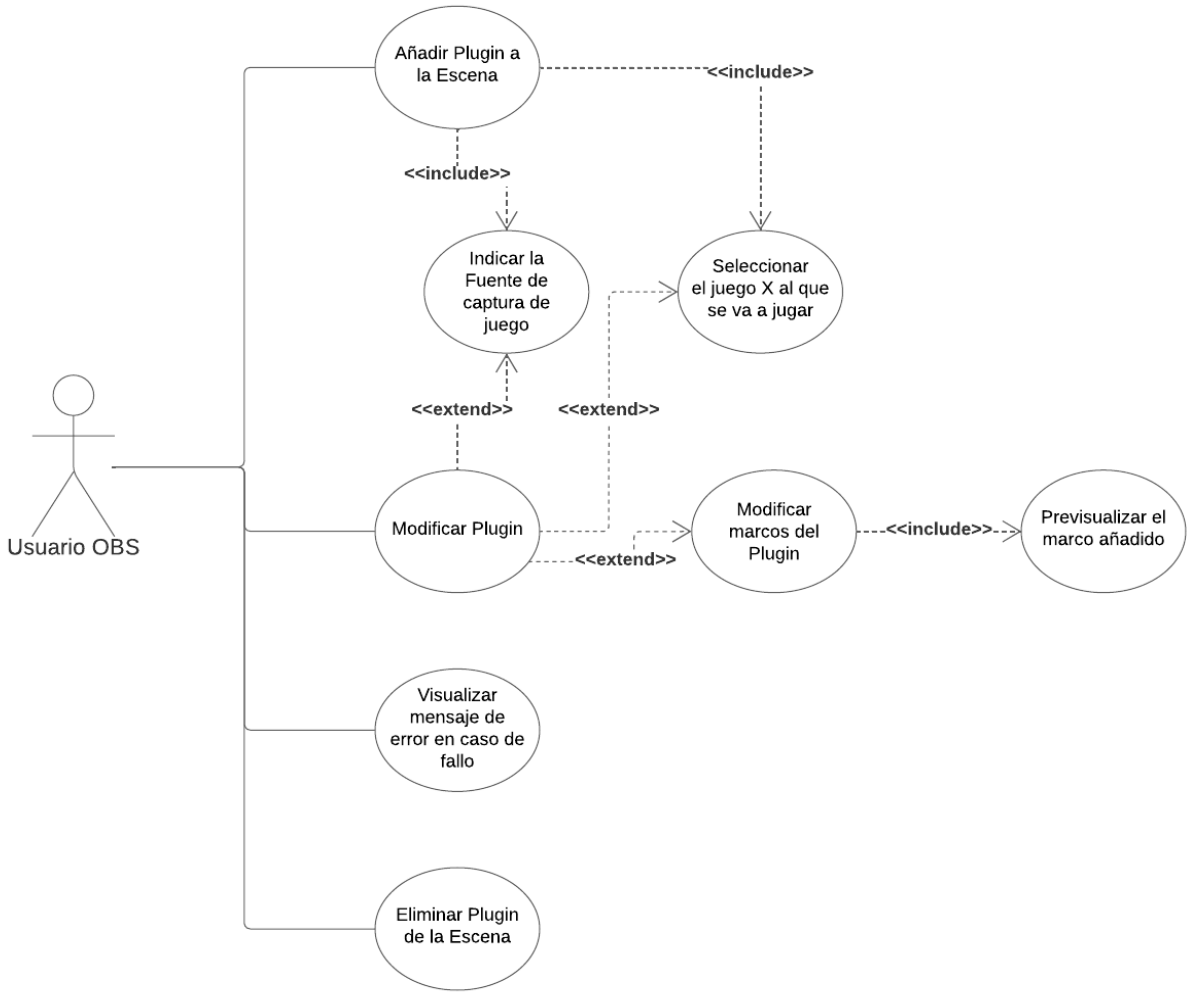
Ilustración 3: Diagrama de casos de uso para Usuario API



Descripción	El usuario envía una solicitud a la API con el objetivo de obtener la cantidad de vida que presenta un jugador en una captura de juego. Si se puede visualizar la vida en la imagen, la API responde con el porcentaje de vida. En caso de error, se responde el mensaje de error.
Actor	Usuario API

Precondición	Ninguna

Ilustración 4: Diagrama de casos de uso para Usuario OBS



Descripción	El usuario añade Reactive Facecam Plugin como Fuente a su Escena. Para posibilitar el correcto funcionamiento del producto, el usuario indicará cuál es la Fuente que está utilizando para mostrar el videojuego y de qué juego se trata.
Actor	Usuario OBS
Precondición	Crear una Escena en OBS

Descripción	El usuario edita las propiedades del producto. Estas propiedades son la captura de juego, el videojuego y los marcos de *webcam* que indican los distintos estados en los que se encuentra la vida del jugador. Si el usuario establece sus propios marcos de *webcam*, estos deberán ser

	previsualizados en el momento en el que se añaden.
Actor	Usuario OBS
Precondición	Tener Reactive Facecam Plugin añadido a la Escena

Descripción	Un mensaje de error es mostrado en pantalla en caso de que ocurra un fallo que comprometa el funcionamiento del producto.
Actor	Usuario OBS
Precondición	Tener Reactive Facecam Plugin añadido a la Escena y que se genere un error en la ejecución del mismo

Descripción	El usuario elimina Reactive Facecam Plugin de su Escena.
Actor	Usuario OBS
Precondición	Tener Reactive Facecam Plugin añadido a la Escena

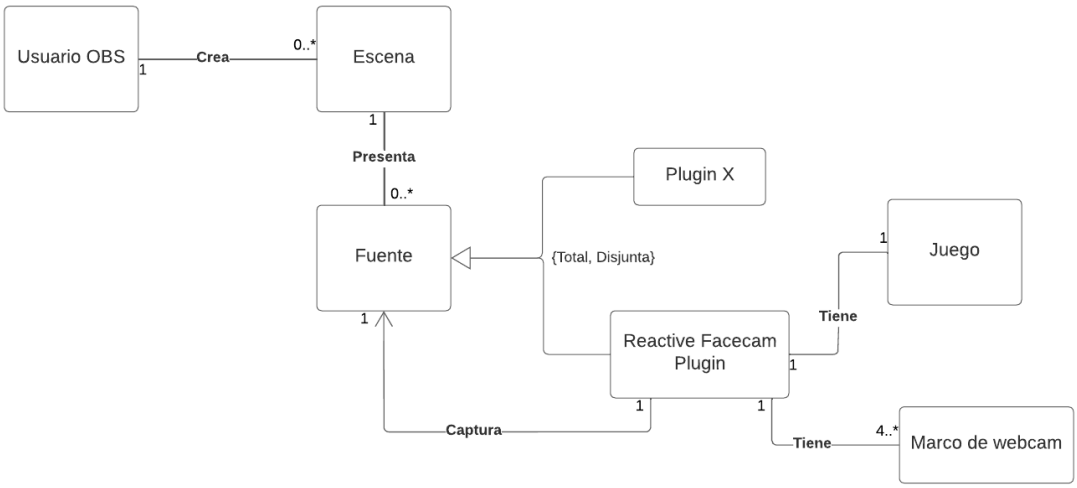
3.1.3 Modelo de dominio

Un modelo de dominio es un diagrama de clases UML que va a expresar las relaciones entre las entidades que forman parte del ámbito del problema. Este modelo es utilizado para representar aspectos del dominio y no del diseño.

En nuestro modelo de dominio, deseamos expresar el rol que tiene Reactive Facecam Plugin dentro del programa OBS.

*Una herencia total disjunta es aquella en la que el padre debe especializarse en alguno de sus hijos y no se permite la especialización en más de un hijo a la vez.

Ilustración 5: Modelo de dominio



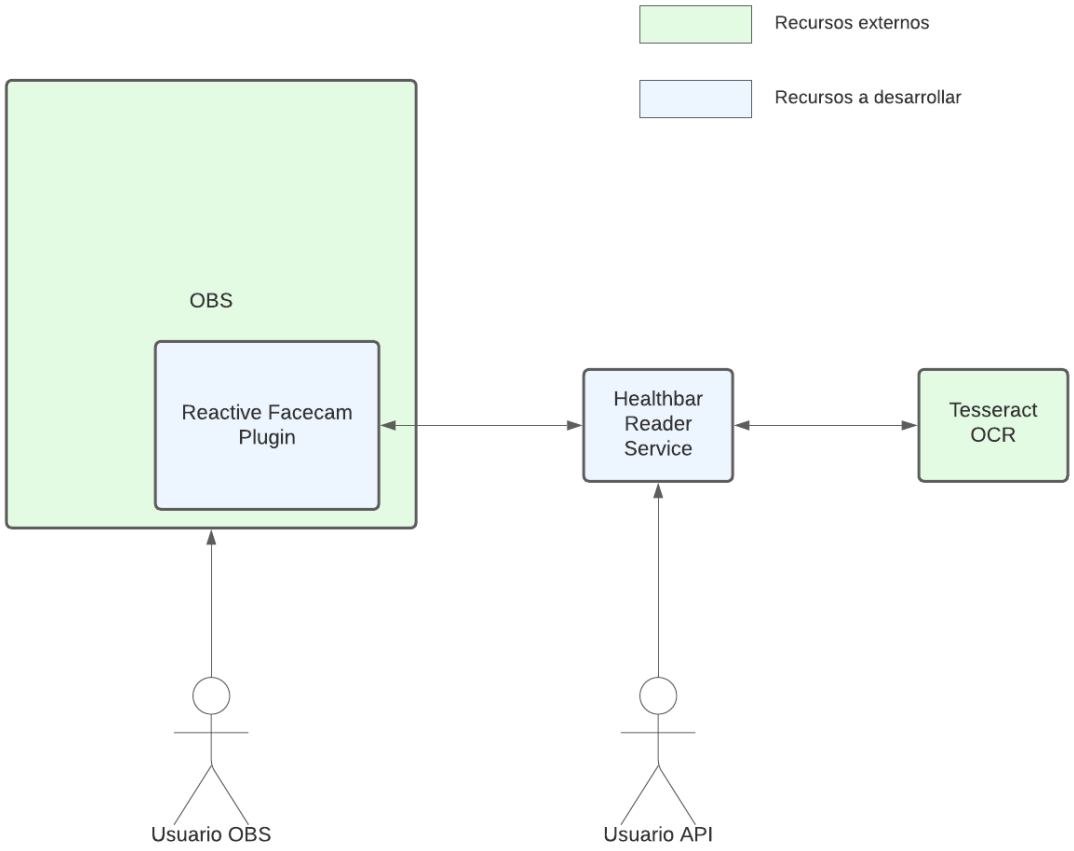
3.1.4 Modelo de contexto

Un modelo de contexto es un diagrama de clases UML que define los límites entre el sistema, o partes del mismo, y el ambiente en el que se encuentra. Este modelo es utilizado para indicar el flujo de información entre el sistema y los componentes externos.

En nuestro modelo de contexto, señalamos las partes del sistema que ya están desarrolladas y no forman parte del proyecto (recursos externos) y aquellas partes que pertenecen a nuestro proyecto y debemos desarrollar (recursos a desarrollar).

He de indicar que Healthbar Reader Service es el nombre de la API a desarrollar en este proyecto y Tesseract OCR es una API de reconocimiento de texto.

Ilustración 6: Modelo de contexto



3.2 Análisis de riesgos

Reactive Facecam Plugin es un producto desarrollado para uso público, por lo que se deben identificar los riesgos con el objetivo de prever y controlar los mismos.

Seguidamente, clasificamos estos riesgos en 3 categorías (del proyecto, técnicos y empresariales) e indicamos qué medidas se tomarán para reducir el impacto de cada uno. Con el objetivo de expresar el impacto que cada riesgo produce en el proyecto, los clasificaremos en una Matriz de Probabilidad e Impacto.

3.2.1 Riesgos del proyecto

- P1. Poco entendimiento de la tecnología

  Medida: Se realizará una investigación exhaustiva de la documentación existente y los proyectos desarrollados.

- P2. Estimación errónea de las tareas a realizar

Medida: Al comienzo del proyecto, se estimará la duración de las tareas que se encuentran en el *backlog*. Al finalizar cada *sprint* (3 semanas), se reestimarán las tareas del siguiente *sprint* teniendo en cuenta los errores de estimación del anterior.

- P3. Falta de motivación

Medida: Se elegirá una metodología de trabajo que proporcione un buen ambiente de desarrollo para el programador. Además, se presentarán *demos* del proyecto con el objetivo de recoger *feedback* de los usuarios.

- P4. Recursos *hardware* insuficientes para el desarrollo

Medida: Se identificarán los programas necesarios para el desarrollo del proyecto y cuáles son los requisitos *hardware* recomendados de los mismos.

3.2.2 Riesgos técnicos

- T1. Alto nivel de complejidad tecnológica

Medida: Como tarea del proyecto, se priorizará el entendimiento del sistema, los subsistemas que lo forman y su comportamiento interno antes de comenzar el desarrollo del producto.

Se establecerá un canal de comunicación con otros desarrolladores de productos integrados en el sistema. De esta manera, podremos mejorar el entendimiento del sistema y consultar dudas a desarrolladores con más experiencia.

- T2. La documentación del sistema con el que se va a trabajar es insuficiente

Medida: Se obtendrá la documentación necesaria utilizando 3 fuentes: la documentación existente del sistema, las extensiones ya publicadas en la página de OBS y el servidor de Discord donde se encuentran los administradores de OBS.

- T3. Dificultad de integración

Medida: Se realizará un análisis tecnológico de los recursos *software* que tienen compatibilidad con el sistema.

- T4. Continuas actualizaciones en el sistema con el que se va a trabajar

Medida: Antes de actualizar las herramientas *software*, se consultarán las notas de la nueva versión para garantizar que no sucederán problemas de compatibilidad con el resto de herramientas integradas y, además, que los recursos utilizados no quedarán obsoletos.

En el caso de que algún recurso quede deprecado, se buscará una alternativa que funcione con la nueva versión de la herramienta.

3.2.3 Riesgos empresariales

- E1. Pérdida del interés de los usuarios por el producto.

Medida: Se compartirán por las redes vídeos con *demos* del producto.

Se darán versiones de prueba a usuarios interesados para que testeen el producto y lo muestren a su comunidad.

Las opiniones de usuarios (tanto *viewers* como *streamers*) deberán ser recogidas y analizadas.

- E2. Los usuarios encuentran el producto difícil de usar

Medida: Se diseñará la interfaz de usuario de acuerdo a los 12 Principios del Diseño.

Se realizarán pruebas de dicha interfaz con usuarios que presenten distintos niveles de conocimiento informático.

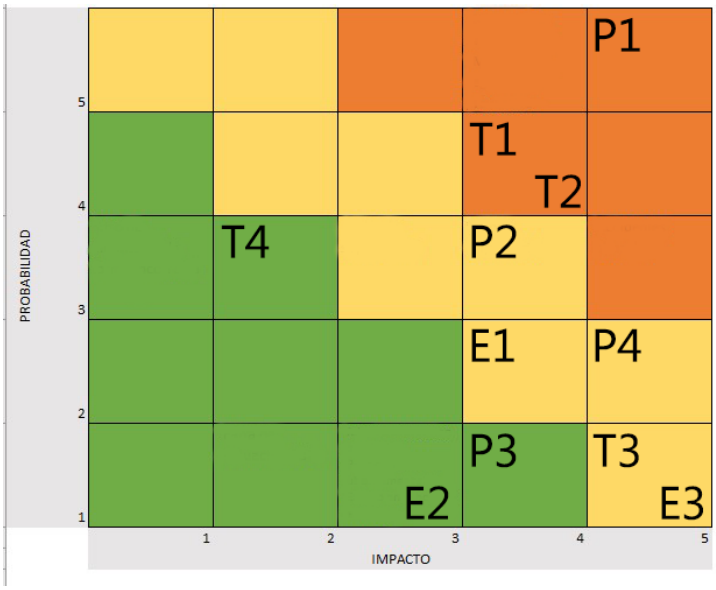
- E3. Los usuarios encuentran el producto visualmente desagradable

Medida: Todo elemento visual del producto será desarrollado por un diseñador gráfico que presenta experiencia en el entorno de diseño para *streaming* de videojuegos. Este diseñador es Matías Patiño.

3.2.4 Matriz de Probabilidad e Impacto

La Matriz de Probabilidad e Impacto es una herramienta utilizada para establecer prioridades a la hora de mitigar los riesgos del proyecto. Esta matriz se puede construir asignando el grado de probabilidad (del 1 al 5, siendo 5 el más probable) y el grado de impacto (del 1 al 5, siendo 5 el de mayor impacto) a cada uno de los riesgos.

Ilustración 7: Matriz de Probabilidad e Impacto de los riesgos del sistema



Los riesgos críticos son aquellos de elevada probabilidad y mucho impacto. En nuestra matriz observamos que los riesgos P1, T1 y T2 son críticos, por lo que deberemos priorizarlos.

3.3 Identificación y análisis de soluciones posibles

3.3.1 Tipos de extensiones de OBS

Actualmente, existen cuatro tipos de extensiones que podemos crear para OBS: Themes, Tools, Scripts y Plugins.

- Los Themes son modificaciones visuales de la interfaz que no añaden ninguna funcionalidad adicional.

- Las Tools son programas que se ejecutan por separado de OBS, pero mejoran la experiencia de su uso. Por ejemplo, NowPlaying Widget se conecta a tu cuenta de Spotify para obtener la canción que se está reproduciendo y genera una URL, la cual puedes añadir como Browser Source (Fuente Navegador), para mostrar a tus espectadores la música que estás escuchando [15].

- Los Scripts son programas que se ejecutan dentro de OBS extendiendo su funcionalidad. Dichos programas pueden hacer acciones como añadir y eliminar Fuentes y Escenas, obtener información de la configuración de usuario, determinar nuevas opciones en la sección de Ajustes, etc.

  Los Scripts se añaden en la sección de Herramientas>Scripts. Estas extensiones comprenden pequeñas herramientas que funcionan en segundo plano. Por ejemplo, Explorer Select Last Recording es un Script

que abre el vídeo de la última grabación cuando el usuario ha parado de grabar [16].

- Los Plugins son también programas que se ejecutan dentro de OBS extendiendo su funcionalidad. La diferencia principal entre Script y Plugin es que el primero puede interactuar, añadir o quitar recursos que ya están presentes en OBS y el segundo permite eso y, además, puede crear recursos completamente nuevos en el programa, como una nueva Transición, Filtro o Fuente.

  Esta es la opción que nos interesa y, para implementarla, deberemos usar la librería libobs, la cual proporciona el *framework* y las funciones necesarias para desarrollar un plugin. Un ejemplo de plugin es Image Reaction, el cual crea una Fuente nueva que va a presentarse con una imagen u otra dependiendo del volumen del audio [17].

3.3.2 Lenguajes disponibles

Una vez conocemos los cuatro tipos de extensiones que hay y determinamos que un plugin es la más indicada para nuestro proyecto, debemos conocer los lenguajes de programación que nos pueden permitir desarrollar un plugin.

Los plugins publicados los podemos encontrar desarrollados en 3 lenguajes: C++, C y JavaScript/HTML. Aproximadamente el 57,9% están en C++, el 40,5% en C y el 1,6% restante en JavaScript/HTML, según podemos comprobar en la página oficial de plugins [9].

Aquellos que están desarrollados en C++ y en C tienen acceso a toda la interfaz de libobs sin ninguna limitación, por ello constituyen la gran mayoría, el 98,4%. Los plugins en C++ y en C no muestran ninguna diferencia.

En el caso de los programados en JavaScript/HTML, son bastante más sencillos y muestran grandes limitaciones.

La primera limitación es que el usuario los debe añadir mediante una Browser Source, lo cual ya descarta que el plugin pueda ser una nueva Transición, Filtro o Fuente. Y la segunda, que la interacción que pueden hacer con OBS está restringida a recibir eventos, obtener las Transiciones y Escenas disponibles, cambiar de Escena utilizando una Transición, comenzar una emisión en directo, finalizar la emisión y similares [18].

También existe la posibilidad del desarrollo en C#, pero no existe ningún plugin publicado en dicho lenguaje. Únicamente se pueden encontrar dos vídeos de usuarios

explorando esta opción en directo [19, 20] y un C# *wrapper* de libobs que no se encuentra terminado [21].

3.4 Solución propuesta

En cuanto al tipo de extensión que deseamos añadir a OBS, inicialmente se planteó que el producto fuese del tipo Script, pero finalmente se decidió desarrollar un Plugin. Esto se debe a que la alternativa de los plugins es más atractiva para la experiencia de usuario, ya que en lugar de ser una herramienta que corre en segundo plano, son recursos que el usuario tiene disponibles y podrá ver en su interfaz.

Además, un Script puede añadir a la Escena una Fuente multimedia con el marco de *webcam* que queramos mostrar, pero un Plugin puede añadir una nueva Fuente a la lista de las Fuentes disponibles para usar. Así pues, esta nueva Fuente podrá mostrar vídeo, como la Fuente multimedia, y presentar su propia configuración que el usuario podrá modificar.

En cuanto al lenguaje a utilizar, dado que C++ y C son las opciones más populares en la comunidad de desarrolladores y no presentan ninguna diferencia entre ellas, se ha decidido programar este proyecto en C. Esta decisión se toma en base a que la documentación pública de OBS se encuentra escrita en C y por gusto personal.

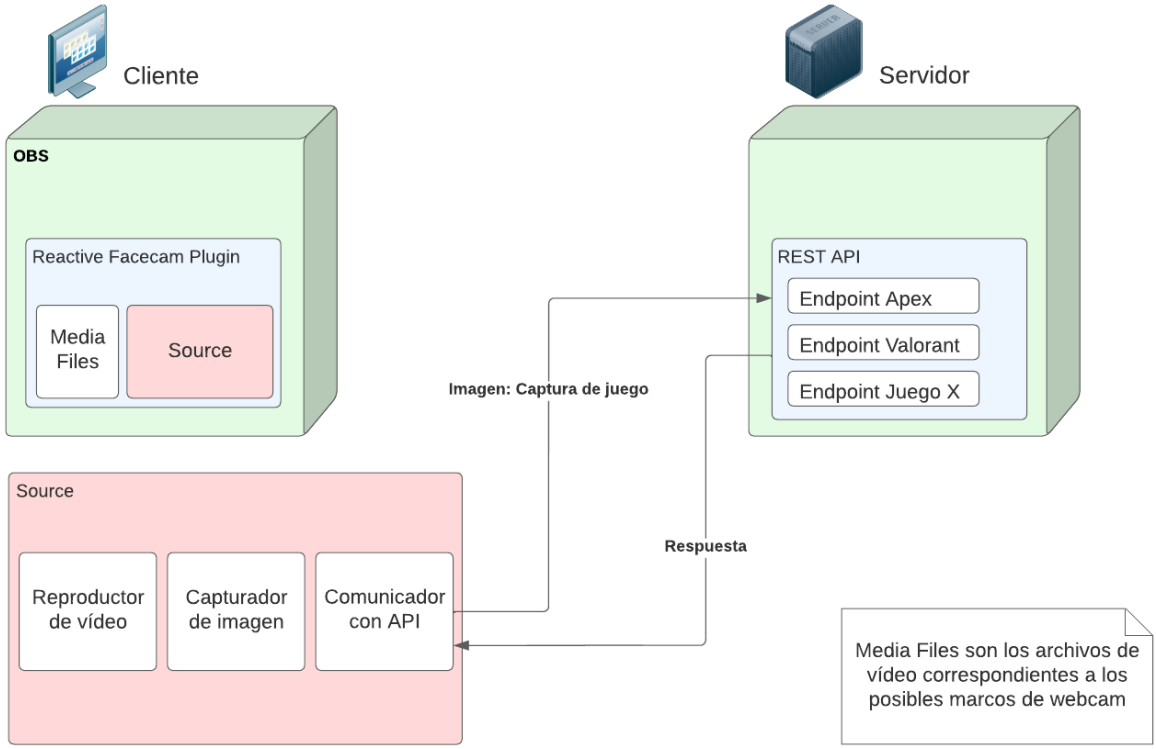
4 Diseño de la solución

4.1 Arquitectura de la solución

Dados los requisitos que debe ofrecer el sistema, la solución presentada debe priorizar la escalabilidad, ya que el proyecto va a procesar cada vez más juegos, mantenibilidad, porque cada juego puede experimentar modificaciones a las que el proyecto tendrá que adaptarse, y tiempo de respuesta, ya que los usuarios esperan una respuesta inmediata a sus cambios en la barra de vida.

Dicho esto, la arquitectura del sistema que se ha escogido para este proyecto es la siguiente:

Ilustración 8: Arquitectura del sistema



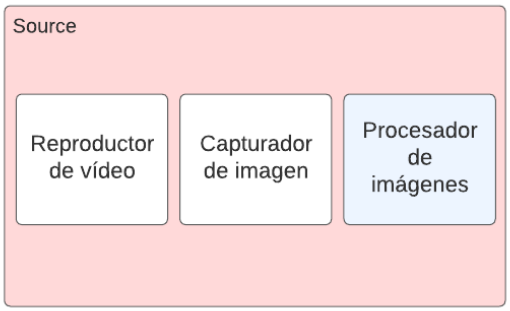
Como podemos apreciar en el esquema, el sistema está basado en una sencilla arquitectura cliente-servidor.

En el plugin de OBS, el cual se encuentra en la máquina del Cliente, se presenta la Source (Fuente) que el usuario puede añadir a su Escena. Esta Source comprende tres módulos: el reproductor de vídeo, el capturador de la imagen de juego y el comunicador con la API. Al presentar esta distribución, podremos actualizar cada uno de los módulos por separado para cumplir con el requisito de mantenibilidad del proyecto.

En la REST API, la cual se encuentra en la máquina Servidor, se presentarán distintos *endpoints* dedicados a cada uno de los juegos que el sistema puede procesar. Esta disposición de *endpoints* nos permite cumplir con el requisito de escalabilidad, de tal manera que se podrán añadir a la API tantos *endpoints* como juegos se desee procesar. Además, esta estructura nos posibilita actualizar cada uno de los *endpoints*, por separado, para perfeccionar la gestión de la imagen del juego correspondiente, así pues, obteniendo un tiempo de respuesta mínimo para cada uno de los juegos.

Por último, respecto al esquema de la solución, me gustaría mencionar un aspecto más. En el diagrama observamos que la lectura de imágenes y obtención del porcentaje de vida del jugador se realiza en la REST API, pero esta organización se puede desarrollar de otra manera. Podríamos plantear una solución en la cual esta tarea se encontrase en un tercer módulo dentro de la Source que se ubica en el plugin.

Ilustración 9: Solución alternativa sin la API



La ventaja de esta estrategia es que el tiempo desde que se captura la imagen hasta que se obtiene el porcentaje de vida del jugador sería mínimo, ya que no tendríamos que sumar el tiempo de envío de imagen y recepción de la respuesta de la API.

La desventaja de esta opción, y motivo por el cual no se ha escogido, es que renunciaríamos al desacoplamiento del Procesador de imágenes, el cual es el componente que más actualizaciones va a presentar.

4.2 Diseño detallado

4.2.1 Diseño REST API

La REST API presentada en la solución surge del desacoplamiento de uno de los tres módulos de Reactive Facecam Plugin. Este módulo era el Procesador de imágenes, cuyo único objetivo era recibir una captura del juego y obtener una respuesta en formato JSON.

Ilustración 10: Ejemplo de respuesta JSON

```
{
  "isImageIdentified": true,
  "errorMessage": "",
  "isLifeBarFound": true,
  "lifePercentage": 0.6
}
```

En esta respuesta se refleja la siguiente información:

- isImageIdentified: si la imagen ha podido ser tratada o no. Una imagen no puede ser tratada cuando el archivo enviado no es una imagen o cuando sus dimensiones no son correctas.
- errorMessage: mensaje de error mostrado en los casos en los que la imagen no puede ser tratada.
- isLifeBarFound: si en la imagen se puede apreciar una barra de vida o no. Un jugador muerto sí que presenta una barra de vida con porcentaje 0, aunque esta no sea visible en pantalla.
- lifePercentage: el porcentaje de vida del jugador.

Con el propósito de que la REST API cumpla este objetivo, se ha decidido seguir una arquitectura de 3 capas: API Layer, Service Layer y Utils Layer.

Como se puede observar, no tenemos una Model Layer, la cual correspondería con la capa donde se almacenarían los modelos de las clases para la base de datos. Esto se debe a que nuestro proyecto no necesita una base de datos, ya que no hay datos que necesiten persistir. El ciclo de vida de una interacción con la API se resume en la recepción de la imagen, la lectura de la información necesaria en la misma y la emisión de una respuesta, así pues no hay ningún dato que la API necesite almacenar para facilitar su próxima interacción.

A continuación, se va a detallar el contenido de cada una de las capas que sí presenta la REST API.

4.2.1.1 API Layer

En esta capa se van a encontrar cada uno de los *endpoints* de la API. En la solución propuesta, encontramos que cada *endpoint* se corresponde con el juego que se va a gestionar. De modo que para X juegos admitidos por el plugin, tendremos X *endpoints* en la API.

Esta capa se comunica directamente con la capa de servicio, la cual presenta la lógica de la aplicación.

4.2.1.2 Service Layer

La Service Layer va a disponer de toda la lógica y, por tanto, de la mayor parte del código de la aplicación. Aquí se validará si la imagen recibida puede ser tratada por esta aplicación, se determinará si la imagen presenta una barra de vida y se obtendrá el porcentaje de vida del jugador, de tal modo que obtendremos la respuesta en formato JSON que se ha indicado anteriormente.

4.2.1.3 Utils Layer

En esta última capa, ubicamos varias funciones auxiliares que únicamente utilizamos para facilitar pequeñas tareas a la hora de programar la aplicación, como funciones *log*, de mostrar la imagen, de mostrar la imagen con líneas de píxeles destacadas, etc.

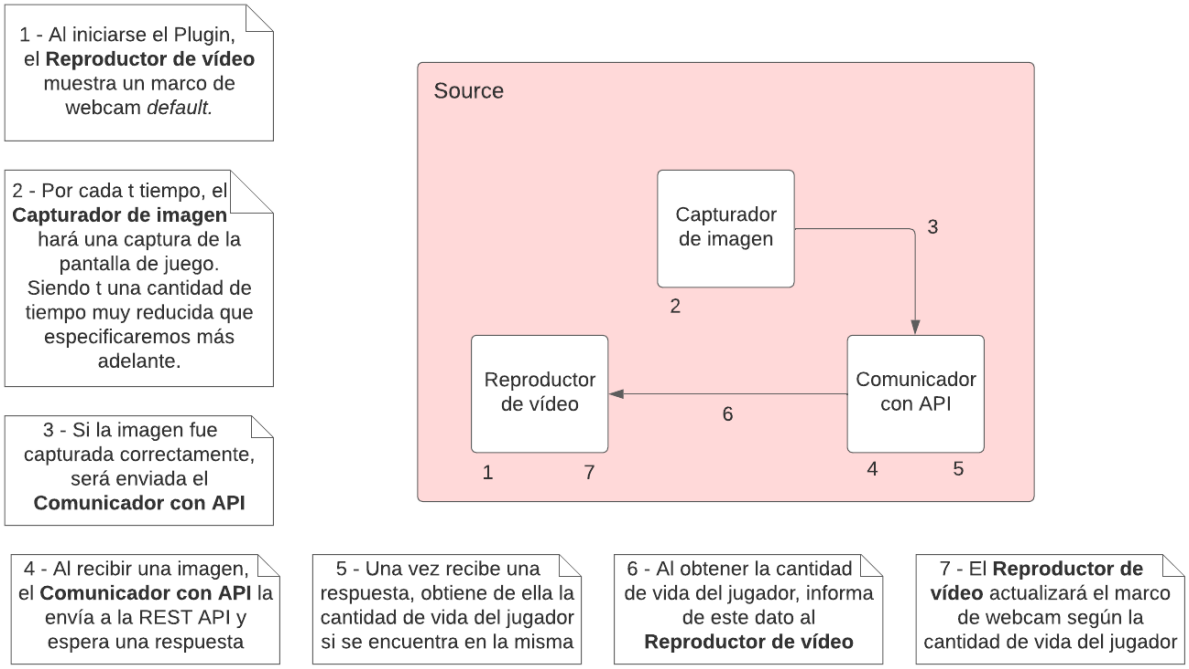
Sumadas a estas, también se encuentran las funciones que dependen de librerías externas, así pues únicamente se deberán hacer cambios en esta capa en el caso de que una actualización de alguna librería pudiera influir en el comportamiento de nuestra aplicación.

4.2.2 Diseño Reactive Facecam Plugin

Al añadir un nuevo componente a OBS, debemos elegir una de las siguientes cuatro implementaciones: Source, Output, Encoder o Service. El plugin a desarrollar presenta una implementación Source pues es la que nos permite mostrar vídeo y capturar la imagen del juego [22].

En la Arquitectura del sistema se puede apreciar que, con el objetivo de cumplir con el propósito del proyecto, este componente va a caracterizarse de tres módulos, cuya comunicación y flujo de información se describe en la siguiente figura.

Ilustración 11: Comunicación entre los tres módulos del plugin



A continuación vamos a describir el comportamiento y las características de cada uno de estos tres módulos.

4.2.2.1 Reproductor de vídeo

Este primer módulo corresponde a la parte del plugin que es palpable para el usuario. Su objetivo es mostrar un marco de *webcam* que sea visualmente atractivo y acorde a la cantidad de vida que el jugador tiene en el momento. Dicha cantidad de vida es facilitada a este módulo gracias al Comunicador con la API.

Si se da la situación en la que no es posible la comunicación con la API o la misma API no ha podido determinar la vida del jugador, se mostrará un marco *default* y se declarará este problema en los *logs* de OBS.

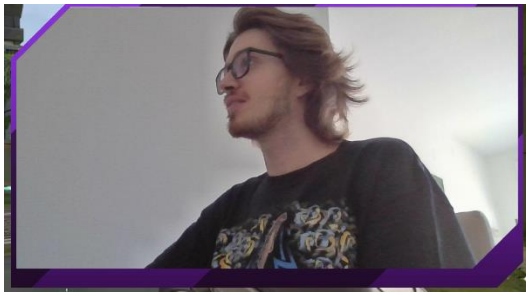
Ilustración 12: Marco de webcam a mostrar para vida al 100%

En esta imagen, el marco es verde ya que el jugador presenta la vida al completo.

Cabe mencionar que la reproducción de este vídeo va a presentar la siguientes características:

- El Reproductor permite mostrar vídeos que presenten un canal alfa, es decir vídeos que contienen elementos transparentes. En consecuencia, los usuarios podrán ubicar el marco de la *webcam* sobre su Fuente *webcam*, como se observa en la siguiente imagen.

Ilustración 13: Marco de webcam con canal alfa



- Este módulo permite la reproducción de vídeo en bucle, pues habrá momentos en los que la vida del jugador no cambie durante un período de tiempo largo y se tendrá que reproducir la animación de un mismo marco de *webcam* repetidas veces.

- El contenido multimedia reproducido debe poder ser administrable por el usuario, esto es que pueda pausar, avanzar y rebobinar el vídeo a voluntad, además de poder gestionar el tamaño que este ocupa en la Escena.

- El Reproductor debe tener acceso a memoria, con el objetivo de obtener los marcos que se encuentran en Media Files y los que desee añadir el usuario.

4.2.2.2 Capturador de imagen

En el caso de este módulo del plugin, se optará por una alternativa que permita renderizar una imagen en su calidad original, es decir con la misma anchura y altura que la resolución del juego, sin exigir demasiada calidad gráfica para la máquina Cliente. Con esto nos referimos a que el ordenador del usuario debe soportar que la

tarea de capturar la imagen suceda cada 2 segundos, máximo, y durante un período de tiempo infinito.

Debe suceder cada 2 segundos como mucho porque queremos que la aplicación reaccione lo antes posible a los cambios en la cantidad de vida del jugador y deseamos, también, que se realice esta tarea de forma infinita porque la duración del *streaming* de los usuarios no la podemos estimar. Puede durar 2 horas, 15 horas, 72, etc.

Sumado a esto, es nuestra responsabilidad tener en cuenta que en un *streaming* se suelen encontrar múltiples programas en uso en el ordenador del usuario, por lo que este módulo tiene que usar la menor cantidad de recursos posibles de la máquina Cliente para no dificultar la utilización de dichos programas.

Por motivos de privacidad, debemos garantizar que el Capturador de imagen únicamente acceda a la información de la captura de juego y no de la Escena donde se encuentre el plugin. Esto es porque queremos obtener la imagen del videojuego, no de la pantalla del ordenador personal del usuario, aunque el videojuego se encuentre en ella.

4.2.2.3 Comunicador con la API

En el diagrama anterior se ilustra que el Comunicador con la API comienza a ejecutarse una vez el Capturador de imagen obtiene la imagen que se desea enviar a la API. Para posibilitar el envío de esta imagen, debemos usar una librería que nos permita enviar archivos de tamaño elevado. Recordemos que la imagen capturada va a tener la misma resolución que la utilizada en el juego. 1920x1080 ó 2560x1440 son las dos resoluciones más utilizadas para jugar a videojuegos.

Este módulo debe permitir también la recepción y correcta lectura de datos de la respuesta generada por la API. Asimismo, gestionará los fallos en la conexión con la API y los mensajes de error que reciba de la misma.

5 Desarrollo de la solución

5.1 REST API

En este apartado se van a plantear las decisiones que se han tomado durante el desarrollo de la API, así como los problemas que se dieron. Con el objetivo de presentar el desarrollo de la forma más clara y concisa posible, vamos a comentar cada una de las capas de la API por separado.

5.1.1 API Layer

Como se mencionó anteriormente, esta es la capa en la que se encuentran los *endpoints* de la API.

La primera cuestión que nos vamos a plantear es qué *framework* va a satisfacer las necesidades de nuestro proyecto. Nuestra aplicación va a seguir un modelo cliente-servidor que no utilizará base de datos, por lo que el *framework* que utilicemos para la API no tiene porqué proporcionar una *Data Abstraction Layer* (DAL). Un *framework* bastante popular que destaca por su sencillez es Flask, el cual nos permite crear una RESTful API en la que organizamos nuestro código respecto a los *endpoints* utilizados por el cliente.

Para la primera versión de este proyecto presentamos dos *endpoints*.

El primero de ellos, va a procesar las imágenes del juego Apex Legends con resolución Full HD (1920x1080).

```
@app.route('/healthbar-reader-service/apex/fullhd', methods=['POST'])
def read_apex_image_fullhd():
    ...
```

El segundo, se encargará de las imágenes de Valorant con misma resolución.

```
@app.route('/healthbar-reader-service/valorant/fullhd', methods=['POST'])
def read_valorant_image_fullhd():
    ...
```

El motivo por el que se ha escogido procesar las capturas de juego con este tamaño para la primera versión es porque la resolución Full HD es la más usada, con más del 62% de los usuarios de Steam disfrutándola [23].

Por último, me gustaría mencionar que se ha optado por Flask en lugar de Django y otras alternativas similares, porque este *microframework* está orientado para las

APIs ligeras y las *Service-Oriented Architertures* (SOAs). De tal manera que es la opción más adecuada para la solución del proyecto.

Una vez tenemos los *endpoints* establecidos, gracias a Flask, y recibiendo las solicitudes de los clientes, vamos a enviar estas solicitudes a nuestra Service Layer.

5.1.2 Service Layer

En esta capa se encuentra contenida la lógica de la aplicación y su objetivo consiste en generar un *output* a partir de un *input*.

- El *input* se trata de la solicitud del cliente, la cual llega a esta capa desde la API Layer.
- El *output* es una respuesta JSON que es enviada a esa misma capa y tiene la estructura que se muestra a continuación.

```
{
 'isImageIdentified' : bool,
 'errorMessage' : str,
 'isLifeBarFound' : bool,
 'lifePercentage' : float,
}
```

Para conseguir este objetivo, en esta capa se van a realizar dos acciones: la validación de la imagen y la obtención de la vida del jugador.

5.1.2.1 Validación de la imagen

En Python, encontramos múltiples librerías para manejar archivos de imagen. Entre las más populares encontramos Scikit-image, OpenCV, Pillow y Matplotlib. Ahora bien, necesitamos generar un objeto Image a partir de un *stream* de *bytes* y la que nos permite realizar esta acción directamente desde esta secuencia es Pillow.

Además, Pillow nos va a permitir confirmar los dos aspectos indicativos de que la solicitud del cliente es válida. Estos dos aspectos son que el *stream* de *bytes* sea una imagen RGBA y que la resolución de esta misma imagen sea Full HD.

```
try:
 img = Image.frombytes('RGBA', (1920, 1080), data)
 is_image_identified = True
except IOError as e:
 error_message = str(e)
 is_image_identified = False
```

Una vez validamos estos aspectos, la Service Layer iniciará la obtención de la vida del jugador a partir de esta imagen.

5.1.2.2 Obtención de la vida del jugador

Determinar la cantidad de vida que presenta el jugador, de una manera inmediata y precisa, es la tarea más importante del proyecto y va a ser crucial para presentar esta aplicación como un producto de calidad.

A fin de desarrollar esta tarea, se plantean los siguientes enfoques para la solución.

El primero de estos enfoques, el más orientado a la rama de computación, consiste en entrenar un clasificador por cada estilo con el que aparece la vida en el juego. Las distintas formas en las que los videojuegos muestran la vida se exponen en la siguiente hoja. Esta opción tiene el beneficio de que supone mucho tiempo en el desarrollo inicial, es decir el entrenamiento del clasificador, pero poco tiempo de mantenimiento.

Consulté la posibilidad de usar este enfoque con un profesor con experiencia en el ámbito de la Inteligencia Artificial, Alberto Sanchis Navarro, y me confirmó que resulta ser una alternativa muy ambiciosa que no se podría abarcar en este TFG si pensaba, además, desarrollar el plugin para OBS. En consecuencia, se descartó esta idea para el desarrollo de este proyecto.

El segundo enfoque, el cual es más indicado para este proyecto y es el que se va a utilizar, supone leer la matriz de píxeles de la imagen. En esta solución, la Service Layer va a recorrer unas posiciones específicas de esta matriz de píxeles para comprobar si el elemento que nos indica la vida del jugador se encuentra en dichas posiciones.

Ahora bien, esta opción tiene como inconveniente que el HUD (*Heads-Up Display*) del juego puede verse actualizado cambiando de posición el componente que muestra la vida del jugador, lo que para el proyecto supone actualizar los valores de las posiciones que recorremos en la matriz. Por lo tanto, esta última opción presenta un tiempo de desarrollo inicial mínimo, pero un elevado tiempo de mantenimiento.

> *El HUD está formado por las áreas de la pantalla en las cuales el jugador puede ver información relativa a la partida que está jugando. Cantidad de munición, vida y armadura son ejemplos de información mostrada en el HUD.

Sin embargo, la solución de recorrer la matriz de píxeles nos garantiza la inmediatez y precisión que estamos buscando.

Es inmediata ya que únicamente estamos recorriendo posiciones específicas de una matriz, lo cual resulta en un tiempo de ejecución casi instantáneo porque estamos comprobando el menor número de posiciones posibles.

En el momento en que detecta un píxel cuyo valor difiere del valor que tendría al mostrar la vida en pantalla, el algoritmo para de recorrer la matriz y determina que en esa imagen en concreto no se encuentra la vida del jugador. El escenario con mayor tiempo de ejecución es en el que se recorren las posiciones de la matriz que indican que sí se puede encontrar la vida y se leen aquellos píxeles que determinan su cantidad

exacta. Este tiempo es [0’014, 0’021] segundos para el *endpoint* de Apex y [0’163, 0’223] segundos para el de Valorant.

Y, también, es precisa porque la ubicación de los componentes en el HUD para una determinada resolución de pantalla es invariable. Siempre se van a encontrar en la misma posición de la matriz de píxeles a menos que, como hemos comentado antes, en una actualización del juego se decida cambiar la ubicación de la vida en el HUD.

Una vez determinamos que vamos a usar el segundo enfoque para nuestra aplicación, se nos plantea un último obstáculo en el desarrollo de esta tarea y es que cada videojuego muestra la vida del jugador de forma distinta.

Ilustración 14: Imagen in-game de Apex

Apex Legends presenta una barra de vida blanca en la parte inferior izquierda de la pantalla.

Ilustración 15: Imagen in-game de Valorant

Valorant muestra la vida de forma numérica en la parte inferior central, ligeramente a la izquierda.

Ilustración 16: Imagen in-game de Amnesia

En Amnesia: The Dark Descent la reducción de tu vida viene indicada por el incremento de tonos rojos en pantalla.

Ilustración 17: Imagen in-game de Multiversus

En Multiversus, cada jugador presenta un número bajo su personaje. Cuanto mayor sea ese número, menor es la cantidad de vida del jugador.

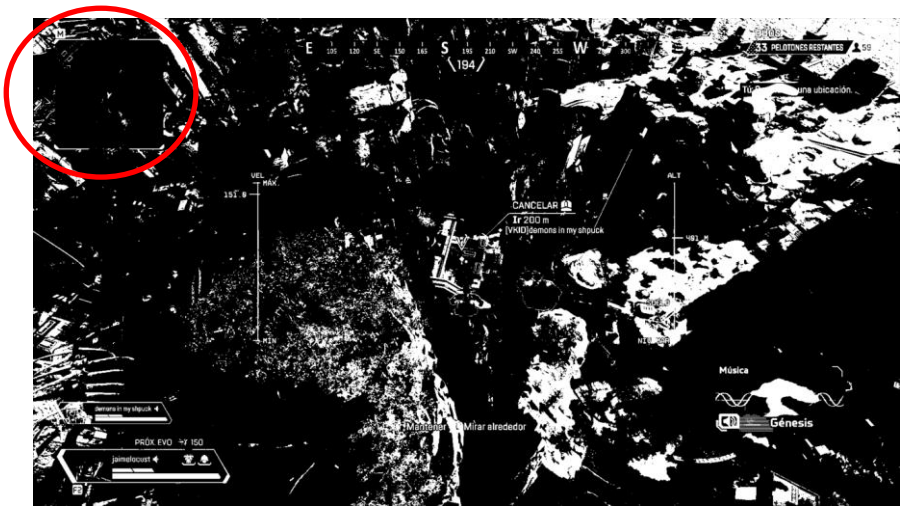
Debido a esta variedad de alternativas, hemos decidido enfocarnos inicialmente en procesar los juegos que indican la vida mediante una barra, como Apex Legends, o de forma numérica, como Valorant.

Seguidamente, se va a exponer cómo la Service Layer obtiene la cantidad de vida para cada una de estos dos casos.

5.1.2.2.1 Caso 1: HUD con barra de vida

En primer lugar, debemos comprobar que el jugador se encuentra jugando una partida antes de leer los *bytes* de la barra de vida. Para ello, la Service Layer va a binarizar la imagen de tal manera que se van a resaltar los elementos del HUD.

Ilustración 18: Imagen in-game de Apex binarizada



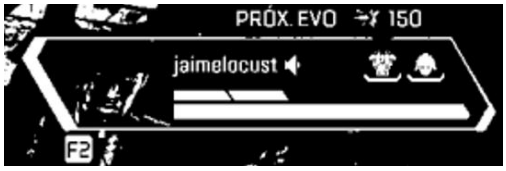
Una vez obtenida esta nueva matriz de *bytes*, vamos a verificar si se presenta algún elemento del HUD en la captura. En el caso de Apex Legends, este elemento es el mini-mapa ubicado en la esquina superior izquierda. Como se puede apreciar, esta es una matriz cuyas celdas contienen únicamente los valores true o false, así que confirmaremos que el mini-mapa está en la imagen haciendo una sencilla comprobación de las celdas que lo definen.

Ilustración 19: Imagen in-game de Apex binarizada, zoom mini-mapa

Estas celdas son las correspondientes a los bordes que delimitan el mini-mapa. En total, en el caso de que sí se presente el mini-mapa en pantalla, 732 celdas van a ser recorridas.

Al determinar que la comprobación ha sido positiva, se va a realizar un segundo y último recorrido de la matriz binarizada. En este recorrido, vamos a obtener la vida que presenta el jugador. Los *bytes* a leer en este caso son los que están en la barra de vida, la cual se encuentra en la esquina inferior izquierda de la imagen.

Ilustración 20: Imagen in-game de Apex binarizada, zoom barra de vida

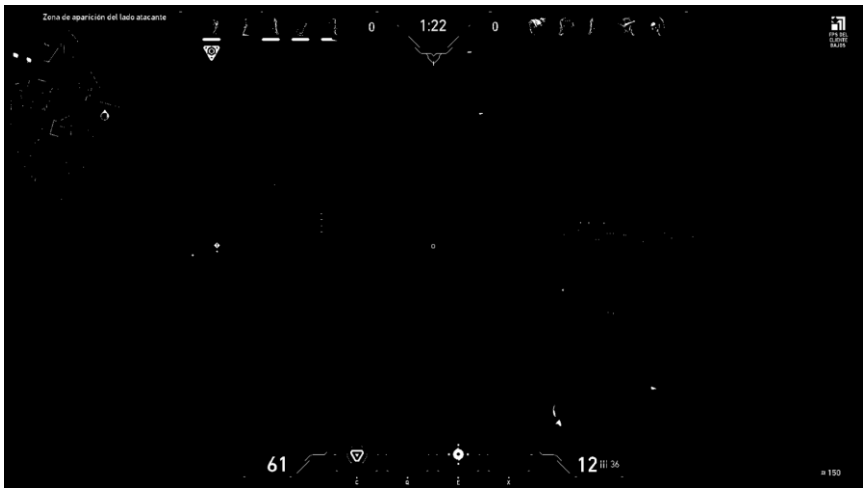


Dicha lectura consiste en leer una de las líneas de celdas que se encuentren en el interior de la barra de vida y hallar el porcentaje de celdas que tengan el valor true. Este porcentaje es la vida del jugador y en esta última acción se recorren 237 celdas.

5.1.2.2.2 Caso 2: HUD con vida numérica

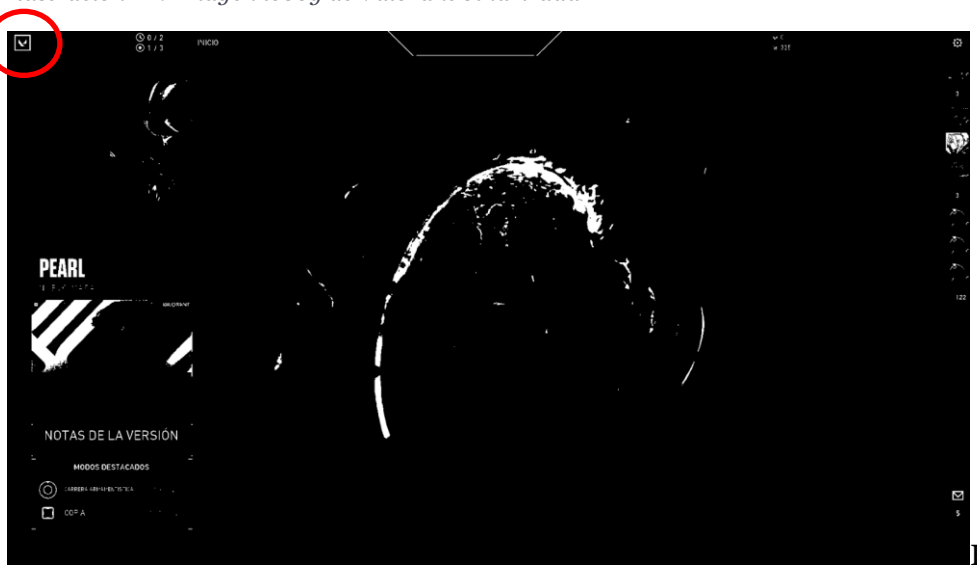
La primera acción que vamos a realizar es la misma que en el Caso 1: binarizamos la imagen para destacar los elementos del HUD y recorremos las posiciones de la matriz que nos permitan verificar que el jugador se encuentra en una partida.

Ilustración 21: Imagen in-game de Valorant binarizada



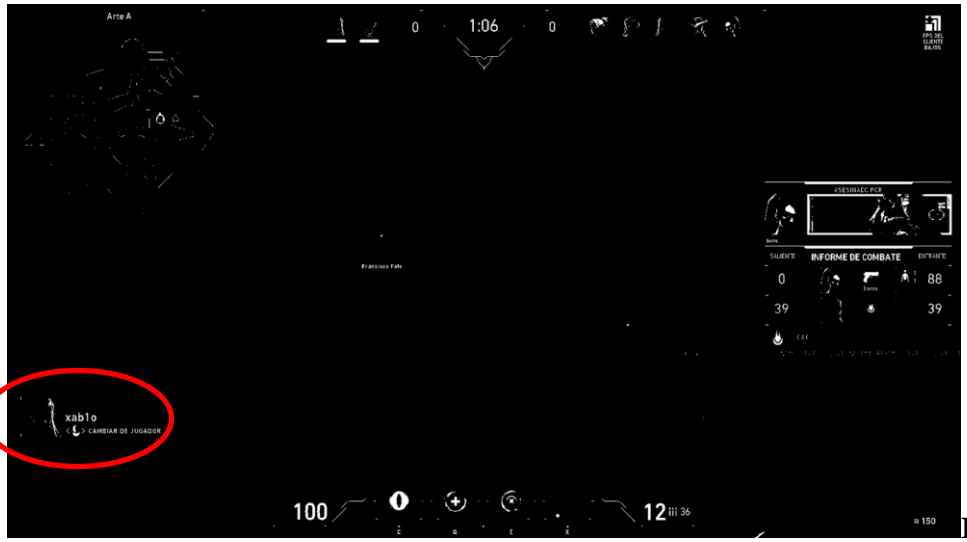
El problema que encontramos en esta situación es que, debido a características como la iluminación del juego y la libertad que tienen los usuarios de editar elementos del HUD, no tenemos la certeza de que estos elementos vayan a estar siempre en una posición concreta. En consecuencia, vamos a determinar que un jugador se encuentra en una partida por descarte. Con esto queremos decir que un jugador estará en partida si NO se encuentra en el *lobby* y NO se encuentra muerto. En el caso de que se encuentre muerto, se entiende que sí que está en partida y su cantidad de vida es 0.

Ilustración 22: Imagen lobby de Valorant binarizada



El jugador está en el *lobby* si localizamos el logo de Valorant en la esquina superior izquierda. En el caso de que sí se encuentre en el *lobby*, se leerán 22 celdas.

Ilustración 23: Imagen de Valorant binarizada, jugador muerto



El jugador está muerto si se muestra a qué otro jugador está espectando en el lateral izquierdo de la pantalla. Si el jugador está muerto, en esta acción se leen 15 celdas.

Inmediatamente después de comprobar que el jugador no se encuentra ni en el *lobby* ni muerto, recortamos la zona de la imagen donde se encuentra la vida del jugador y la enviamos a nuestra tercera capa, Utils Layer, en la que se usará una herramienta de OCR (*Optical Character Recognition*) para determinar qué número aparece en la imagen.

Ilustración 24: Imagen in-game de Valorant binarizada, zoom vida del jugador

5.1.3 Utils Layer

Por último, en Utils Layer, tenemos funciones de utilidad y funciones que acceden a librerías externas.

En esta primera versión del proyecto, únicamente presentamos una función de acceso a librería externa en Utils Layer. Esta función recibe como *input* una imagen y obtiene como *output* un `float` que corresponde con el número que aparece en dicha imagen, ó -1 en el caso de que no halle ningún número. A fin de realizar esta tarea, se debe utilizar una librería de OCR.

Para decidir qué librería de OCR utilizaremos en el proyecto, se ha buscado cuáles son consideradas las mejores en 2022 y llegamos a la conclusión de que la más indicada es Tesseract OCR de Google [24]. Además de que su uso no resulta complicado, se trata de una herramienta con un *error rate* muy reducido. Según [25]: *"Test results on English, a mixture of European languages, and Russian, taken from a random sample of books, show a reasonably consistent word error rate between 3.72% and 5.78%"*. Pese a que este margen de error es muy bajo, siempre podemos incrementar nuestro éxito con Tesseract aplicando ciertas mejoras.

Con el objetivo de obtener el mayor número de aciertos:

- Binarizamos la imagen antes de que sea procesada por Tesseract, lo cual ya se hace previamente en Service Layer.
- Indicamos el Inglés como lenguaje, por el alfabeto latino. `lang='eng'`
- Indicamos que debe procesar la imagen como un solo carácter, es decir un único número. `--psm 10`
- Señalamos que el texto está bien cuadrado y no debe buscar otra orientación para leerlo. `--oem 3`
- Denotamos que los caracteres son únicamente numéricos. `tessedit_char_whitelist=0123456789`

Así pues, obtendremos la cantidad de vida del jugador a partir de una imagen binarizada de la siguiente manera:

```
life_percentage_str = tess.image_to_string(image_bin, lang='eng',
        config='--psm 10 --oem 3 -c tessedit_char_whitelist=0123456789')
```

5.2 Reactive Facecam Plugin

Al igual que hemos comentado las decisiones de desarrollo de la REST API, vamos a hacer lo mismo con la segunda parte y la más importante del proyecto, nuestro plugin de OBS, Reactive Facecam Plugin.

Como ya hemos mencionado en la sección de Estado del arte, no se han podido encontrar tutoriales ni documentación ni guías suficientes para el desarrollo de un nuevo plugin. Sin embargo, los usuarios de la comunidad de OBS generalmente publican el código de sus proyectos, los cuales presentan estructuras muy distintas entre ellos.

Sabiendo esto, aprendimos cómo desarrollar un plugin de OBS utilizando ingeniería inversa a partir del plugin llamado Lightgun Flash Filter [26]. En consecuencia, el proceso de desarrollo que se ha realizado en este proyecto no es el que ha sucedido en todos los plugins existentes, pero consideramos que es el mejor para nuestro proyecto.

5.2.1 Desarrollo básico de un plugin

Para el desarrollo de esta parte del proyecto usaremos el lenguaje C y el compilador CMake.

En primer lugar, lo que debemos hacer al comenzar el desarrollo de cualquier plugin es permitir el acceso a la librería libobs. Para ello, hay que descargar el código fuente de OBS en la máquina donde se vaya a desarrollar y compilar el plugin. Una vez hecho esto, tenemos que indicar en nuestro archivo CMakeLists.txt del proyecto que es necesario el acceso a libobs.

```
find_package(LibObs REQUIRED)

include_directories(
    ${LIBOBS_INCLUDE_DIR}
    "${LIBOBS_INCLUDE_DIR}/../UI/obs-frontend-api"
    ...
)

target_link_libraries(${CMAKE_PROJECT_NAME}
    ${LIBOBS_LIB}
    ${OBS_FRONTEND_LIB}
    ...
)
```

Además, en el proyecto deberá localizarse un archivo FindLibObs.cmake, el cual va a definir las variables `LIBOBS_INCLUDE_DIR` y `LIBOBS_LIB` que vemos en el código de arriba. Este archivo puede encontrarse para uso público en el código de varios plugins [27].

A continuación, será necesario un tercer archivo plugin-main.c el cual cargará los módulos que deseemos añadir a OBS. Estos módulos no son los que se comentaron en la sección de Diseño de la solución sino componentes que nosotros estamos aportando a OBS dentro de nuestro plugin. Cada módulo añadido debe presentar una de las cuatro posibles implementaciones Source, Output, Encoder o Service.

En nuestro caso, vamos a añadir una nueva Fuente a OBS y sabemos que una Fuente es uno de los tipos presentados por la implementación Source, por lo que deberemos cargar un módulo Source.

```
OBS_DECLARE_MODULE()

extern struct obs_source_info reactive_facecam_plugin;

bool obs_module_load(void)
{
    obs_register_source(&reactive_facecam_plugin);
    blog(LOG_INFO, "plugin loaded successfully (version %s)", PLUGIN_VERSION);
    return true;
}
```

Como podemos observar en el código anterior, esta Source es una `struct` que nosotros definimos.

```
struct obs_source_info reactive_facecam_plugin = {
    .id = "reactive_facecam_plugin",
    .type = OBS_SOURCE_TYPE_INPUT,
    .output_flags = OBS_SOURCE_VIDEO,
    .get_name = rfp_getname,
    .update = rfp_update,
    .create = rfp_create,
    .destroy = rfp_destroy,
};
```

Dicha `struct` va a presentar unos atributos básicos y unos métodos que serán ejecutados a lo largo de su ciclo de vida.

`.create` se invocará cuando el usuario añade esta Fuente a su Escena.

`.destroy` cuando la elimine de la Escena o cierre el programa de OBS.

`.update` cuando actualice los ajustes del plugin, si es que el plugin permite ser personalizado.

Uno de los atributos es el identificador, el cual se corresponde con una segunda `struct` que tenemos que crear y donde definiremos todos los atributos que definirán en sí nuestra Fuente. Si se quiere añadir ajustes al plugin, serán añadidos aquí. A continuación, mostramos esta `struct` con los tres atributos básicos que debe tener.

```
struct reactive_facecam_plugin {
    obs_source_t *context;
    uint32_t width;
    uint32_t height;
};
```

`context` es la referencia a nuestra Fuente.
`width` y `height` son la anchura y altura que ocupa la Fuente en la Escena.

Hasta aquí, se ha expuesto lo que debe tener todo plugin de OBS para funcionar. Esperamos que sirva como guía para futuros desarrolladores. De ahora en adelante, vamos a hablar del desarrollo específico del Reactive Facecam Plugin.

5.2.2 Desarrollo específico para este plugin

La Source correspondiente al proyecto Reactive Facecam Plugin va a necesitar una serie de métodos que reaccionen a distintos eventos en su ejecución.

```
struct obs_source_info reactive_facecam_plugin = {
    .id = "reactive_facecam_plugin",
    .type = OBS_SOURCE_TYPE_INPUT,
    ...
    .video_tick = rfp_tick,
    .media_play_pause = rfp_play_pause,
    .media_restart = rfp_restart,
    .media_stop = rfp_stop,
    .media_get_duration = rfp_get_duration,
    .media_get_time = rfp_get_time,
    .media_set_time = rfp_set_time,
    .media_get_state = rfp_get_state,
    .get_properties = rfp_properties,
    .get_defaults = rfp_defaults,
    .missing_files = rfp_missingfiles,
};
```

Estos métodos actuarán como *Listeners* y favorecerán el correcto funcionamiento de los módulos expuestos en Diseño de la solución.

- `.video_tick` es un método invocado por cada *frame* de vídeo. Si grabamos a 60 fps, será invocado 60 veces cada segundo. Este método será utilizado para iniciar el paso 2 de la Ilustración 11.

- Todos los métodos de `.media` permiten que el vídeo mostrado pueda ser controlable por el usuario, es decir, que pueda ser pausado, rebobinado, etc.

Ilustración 25: Elementos de la interfaz para interactuar con el vídeo

- `.get_properties` nos sirve para presentar al usuario los ajustes del plugin que le van a permitir personalizar su experiencia. Y `.get_defaults` tiene como objetivo indicar cuáles son aquellos ajustes

que el plugin tendrá por defecto. Incluidos qué marcos de *webcam* se usarán si el usuario no establece unos distintos.

- `.missing_files` es invocado cuando el plugin no encuentra en memoria alguno de estos marcos de *webcam*, los cuales son en sí archivos de vídeo.

Seguidamente, vamos a explicar el proceso de desarrollo de cada uno de los tres módulos del plugin. Debo indicar que la separación del código en módulos es puramente por organización. Los tres módulos no son clases, sino archivos donde se encuentran por separado los fragmentos de código que van a permitir realizar las tres tareas principales del plugin.

Nótese que no vamos a hablar de todo el contenido que presentan, únicamente comentaremos sus elementos más cruciales.

5.2.2.1 Reproductor de vídeo

La tarea de mostrar contenido multimedia es común entre los plugins desarrollados para OBS. El problema que se nos presenta es que deseamos mostrarlo de una forma muy específica para cumplir con nuestros requisitos.

Recordamos que los marcos de *webcam* van a presentar un canal alfa, tienen que reproducirse en bucle y nuestro proyecto necesita acceso a memoria para obtener los archivos de vídeo. La segunda y tercera condición no suponen ningún problema, pero la primera de ellas no la comparten todos los plugins que reproducen vídeo.

No obstante, OBS ya presenta un plugin por defecto cuya única función es mostrar archivos multimedia y permite la reproducción con canal alfa, FFmpeg Source (Fuente multimedia) [28]. Dicho plugin utiliza el proyecto media-playback [29], el cual fue también creado por los desarrolladores de OBS.

Media-playback es de nuestro interés ya que pone a nuestra disposición una interfaz que nos va a facilitar mostrar contenido multimedia dentro de nuestro plugin. El recurso principal del que depende este proyecto es FFmpeg, un set de librerías de código abierto desarrollado para manejar contenido multimedia. Es usado por productos bastante populares como Audacity, Blender, VLC media player y Google Chrome.

Una vez tenemos acceso a la interfaz del proyecto media-playback, ya podemos desarrollar el contenido del módulo Reproductor de vídeo.

5.2.2.1.1 Atributos a añadir a la Fuente

Primero, añadimos una serie de atributos a la segunda `struct` de nuestro proyecto, la que define la Fuente en sí.

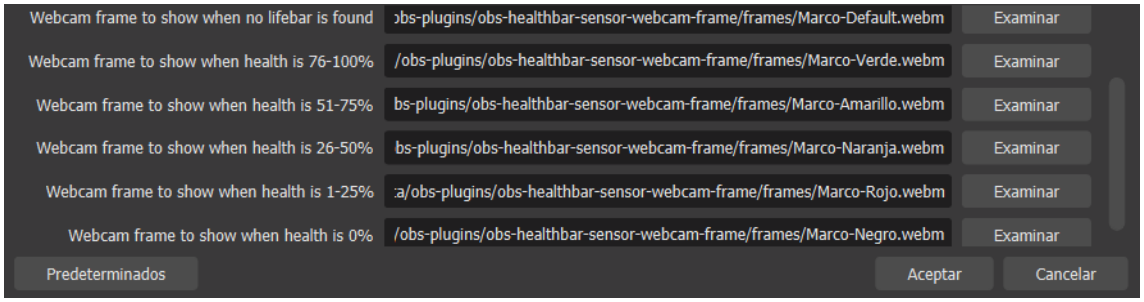
```
struct reactive_facecam_plugin {
    obs_source_t *context;
    ...
    mp_media_t media;
    char *no_lifebar_frame_path;
    char *high_health_frame_path;
    char *medium_high_health_frame_path;
    char *medium_low_health_frame_path;
    char *low_health_frame_path;
    char *zero_health_frame_path;

    int current_webcam_frame;
};
```

El atributo `media` se trata de una referencia al reproductor de vídeo que nos proporciona el proyecto de media-playback. Esta referencia junto a una colección `mp_media_info`, que comentaremos a continuación, nos servirán para mostrar el vídeo que nosotros deseemos en la Escena.

Los 6 atributos siguientes son las direcciones de memoria que indican dónde se encuentran los vídeos correspondientes a los 6 estados que puede tener el marco de *webcam*. Dichos estados son *default* y cinco situaciones en las que se presenta la vida del jugador: 0% de vida, de 1 a 25%, de 26 a 50%, de 51 a 75% y de 76 a 100%. El valor de estos atributos puede cambiarse en los ajustes del plugin.

Ilustración 26: Ajustes del plugin, ubicaciones de los marcos de webcam



Además, debe añadirse un atributo `current_webcam_frame` que usaremos para indicar cuál es el marco de *webcam* activo en la Escena.

5.2.2.1.2 Métodos que aporta este módulo

Los métodos principales que están contenidos en el Reproductor de vídeo se corresponden a las siguientes tareas que realiza: abrir y reproducir el vídeo, mostrar una *preview* del mismo, cambiar el vídeo que se está mostrando y facilitar al usuario unas *hotkeys* para controlar el vídeo.

*Las *hotkeys* son “teclas rápidas” que permiten asignar una acción a una tecla específica.

```
static void rfp_media_open(struct reactive_facecam_plugin *plugin);
```

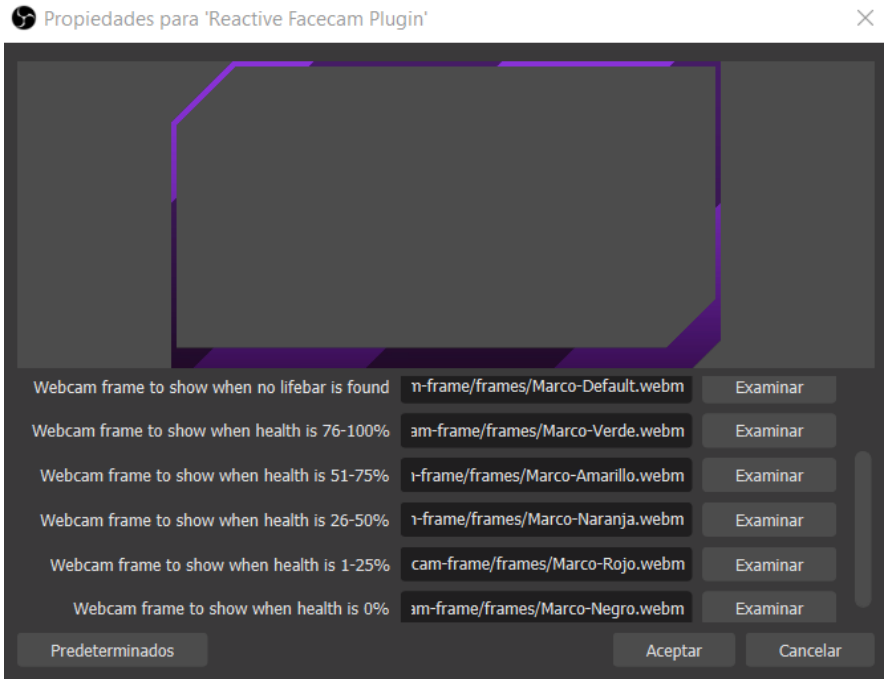
Con el fin de abrir y reproducir el vídeo, creamos `rfp_media_open` que utilizará el atributo `media` y la colección `mp_media_info` para mostrar el vídeo que deseemos en la Escena. Esta colección es un tipo de datos definido en el proyecto media-playback y va a contener toda la información del contenido multimedia a mostrar.

```
static void rfp_media_start(struct reactive_facecam_plugin *plugin);
```

Crearemos, también, el método `rfp_media_start`. Tanto este método como el anterior serán invocados al iniciar el ciclo de vida del plugin y al cambiar el marco de *webcam* que se desea mostrar. `rfp_media_start` nos va a permitir indicar, usando media-playback, que vamos a comenzar la reproducción de un vídeo, lo que quiere decir que deben mostrarse en la interfaz de OBS los elementos de control de vídeo que aparecen en Ilustración 25.

Seguidamente, se va a señalar que el vídeo debe aparecer también como *preview* en los ajustes del plugin.

Ilustración 27: Ajustes del plugin, preview del marco de webcam



```
void change_webcam_frame(struct reactive_facecam_plugin *plugin,
                char *new_frame_path);
```

El tercer método que será crucial en el Reproductor de vídeo es `change_webcam_frame`, el cual es invocado por el tercer componente que mencionamos en Diseño de la solución, el Comunicador con la API, para indicar el nuevo marco de *webcam* que se debe exponer. Este método es, pues, el punto en el que se relacionan el Reproductor de vídeo y el Comunicador con la API.

`change_webcam_frame` va a responsabilizarse de hacer una gestión óptima de la memoria utilizada por el Reproductor de vídeo, esto es liberar las referencias al reproductor proporcionado por media-playback y a la ubicación donde se almacena el marco de *webcam*, y va a usar los dos métodos anteriores para mostrar el nuevo vídeo.

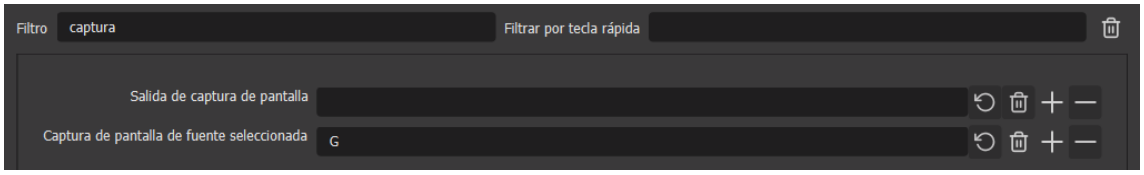
```
static void rfp_restart_hotkey(void *data, obs_hotkey_id id, obs_hotkey_t
*hotkey,
                bool pressed);
static bool rfp_play_hotkey(void *data, obs_hotkey_pair_id id,
                        obs_hotkey_t *hotkey, bool pressed);
static bool rfp_pause_hotkey(void *data, obs_hotkey_pair_id id,
                         obs_hotkey_t *hotkey, bool pressed);
static void rfp_stop_hotkey(void *data, obs_hotkey_id id,
                        obs_hotkey_t *hotkey, bool pressed);
```

Por último, con el objetivo de mejorar la experiencia del usuario, añadimos los métodos *hotkey*, mostrados más arriba, cuya invocación se va a asignar a las teclas que indique el usuario si este lo desea. Como se aprecia en el código, estas teclas pueden abarcar los eventos *restart*, *play*, *pause* y *stop* y harán las mismas acciones que sus correspondientes métodos `.media` utilizados para controlar el vídeo.

5.2.2.2 Capturador de imagen

En una versión previa del proyecto que se va a entregar en este TFG, este módulo no existía. En su lugar se tomaban las capturas de pantalla mediante el método `obs_frontend_take_source_screenshot` perteneciente a obs-frontend-api [30]. Esta función tenía la misma ejecución que la *hotkey* de "Captura de pantalla de fuente seleccionada", la cual guarda una captura en la misma carpeta donde se almacenan las grabaciones.

Ilustración 28: Ajustes programa OBS, hotkey para hacer captura de una fuente



```
11:02:44.052: RFP - T EJ SCREENSHOT: 0.001000 secs
11:02:45.036: Saved screenshot to 'C:/Users/jaime/Videos/Screenshot 2022-08-
13 11-02-44.png'
```

Sabiendo la información de los *logs* aquí presentados, determinamos que el tiempo de ejecución de `obs_frontend_take_source_screenshot` es casi inmediato y podemos deducir el nombre con el que se guarda la imagen conociendo la ruta donde se almacenan las grabaciones y la fecha en la que se ha tomado la captura.

El problema de utilizar esta función es que no recibimos ningún tipo de respuesta por parte de OBS. No sabemos si se ha conseguido tomar el *screenshot* o no. Aunque deduzcamos el nombre, no tenemos la certeza de que acertemos. Y tampoco sabemos cuándo ha sido finalmente almacenado en memoria porque no recibimos ningún tipo de *feedback* por parte de la función. Ante esta incertidumbre y falta de garantías, se decidió explorar una segunda opción.

Tras preguntar por el Discord de desarrolladores de OBS y distintos foros, encontramos una respuesta a un usuario que también estaba buscando una manera para tomar capturas de pantalla de una Fuente concreta. Afortunadamente, en 2019 el usuario de GitHub synap5e creó un Filtro de OBS que permite tomar *screenshots* de la Fuente a la que este filtro esté asignada [31]. Pasó un tiempo hasta que encontramos este proyecto ya que su código está en GitHub, pero el plugin no está publicado en la página oficial donde se encuentran el resto de plugins.

La diferencia principal entre el proyecto de synap5e y nuestro proyecto es que nosotros queremos capturar la imagen de la Fuente en la que se encuentre el videojuego, pero esto no supone un gran problema.

Seguidamente, se expone el contenido desarrollado para el Capturador de imagen.

5.2.2.2.1 Atributos a añadir a la Fuente

En primer lugar, añadimos los siguientes atributos a la `struct` que define nuestra Fuente.

```
struct reactive_facecam_plugin {
    obs_source_t *context;
    ...
    obs_source_t *game_capture_source;
    sem_t mutex;

    gs_texrender_t *texrender;
    gs_stagesurf_t *staging_texture;
};
```

El primero de los atributos a añadir es `game_capture_source`, el cual almacena la referencia a la Fuente en la que se encuentra el videojuego. El valor de este atributo es establecido por el usuario en los ajustes del plugin.

También va a ser necesaria la inclusión de un semáforo, `mutex`, entre los atributos de la Fuente. Como el Capturador de imagen y el Comunicador con la API se ejecutan ambos en un mismo hilo en paralelo al hilo principal de ejecución, necesitamos un semáforo que garantice que no se inicie la ejecución de más de un hilo en paralelo. Esto lo veremos con más detalle en el siguiente apartado.

Ahora bien, `texrender` y `staging_texture` son los atributos que nos van a posibilitar obtener el *screenshot* deseado. El primero de ellos lo usaremos para renderizar las texturas de la Fuente donde se encuentre el juego y el segundo para copiar estas texturas a la memoria RAM. De nuevo, lo veremos en el siguiente apartado.

5.2.2.2.2 Métodos que aporta este módulo

Los métodos que contiene el Capturador de imagen se corresponden a las siguientes tareas que realiza: obtener la referencia de la Fuente donde se encuentra el juego, gestionar el hilo en paralelo que comprende la ejecución de este módulo y el Comunicador con la API y obtener la imagen de captura de juego.

```
void check_for_game_capture_source(struct reactive_facecam_plugin *plugin);
```

El primero de los métodos que este módulo engloba es `check_for_game_capture_source`. El recurso que necesita esta función para ejecutarse correctamente es el nombre de la Fuente donde se presenta el juego.

Ilustración 29: Ajustes del plugin, Fuente donde se muestra el juego

Dicha información es administrada por el usuario en los ajustes del plugin. Así pues, con este dato el Capturador de imagen obtiene la resolución a la que el plugin deberá renderizar la imagen y la referencia de la Fuente que va a capturar, `game_capture_source`.

`check_for_game_capture_source` va a ejecutarse al comienzo del ciclo de vida del plugin y cada vez que este ajuste sea modificado por el usuario.

```
static void rfp_tick(void *data, float seconds);
```

Nuestro segundo método a desarrollar es `rfp_tick`, el cual será invocado automáticamente por cada *frame* de vídeo. Dado que este método va a ser ejecutado continuamente hasta el fin del ciclo de vida del plugin, es ideal para realizar procesos que tengan infinitas iteraciones. En nuestro caso, el proceso que queremos que esté ocurriendo siempre es la ejecución paralela del hilo en el que se procede a capturar la imagen de juego, enviar dicha imagen a la API y recibir la respuesta.

```
static void rfp_tick(void *data, float seconds)
{
    ...
    sem_getvalue(&plugin->mutex, &value);

    if (value == 1) {
        pthread_t thread;
        pthread_create(&thread, NULL, thread_take_screenshot_and_send_to_api,
(void*) plugin);
    }
}
```

El hecho de que este hilo suceda paralelamente a la ejecución principal favorece que nuestro plugin no se quede "congelado" en el caso de que la API se atrase al enviar la respuesta u otros problemas similares. Como se aprecia en el código anterior, es en esta función en la que el semáforo `mutex` va a regular que únicamente se ejecute un hilo en paralelo si el hilo anterior ya ha concluido.

```
bool render_game_capture(struct reactive_facecam_plugin *plugin);
```

Por último, comentaremos la tercera función que el Capturador de imagen aporta al proyecto. Esta función es `render_game_capture`, la cual va a ejecutarse al comienzo del hilo `thread_take_screenshot_and_send_to_api` y tiene como único objetivo capturar la imagen de `game_capture_source`. Para ello, vamos a utilizar las funciones de gráficos y renderizado de texturas que nos facilita OBS [32, 33].

El resultado de su ejecución va a ser un *stream* de *bytes* que almacenaremos en memoria y cuya referencia guardaremos en uno de los atributos de nuestra Fuente.

5.2.2.3 Comunicador con la API

En esta parte del código del proyecto se va a enviar el *stream* de *bytes*, obtenido en el módulo anterior, a la REST API y se va a extraer la información de la respuesta JSON. Para ello, emplearemos Curl, una librería y herramienta de comandos utilizada para transferir datos por, prácticamente, todo el mundo en Internet. Curl proporciona una sencilla interfaz para enviar datos mediante una sintaxis de URL.

A continuación, exponemos los elementos del código que pertenecen a este módulo.

5.2.2.3.1 Atributos a añadir a la Fuente

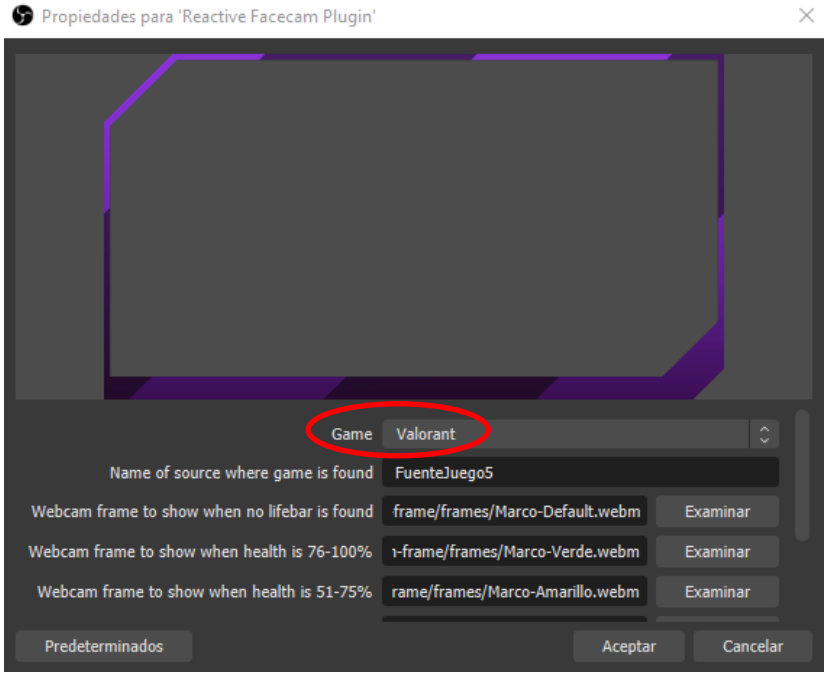
Los atributos más esenciales que se han añadido a nuestra Fuente durante el desarrollo del Comunicador con la API son los siguientes.

```
struct reactive_facecam_plugin {
    obs_source_t *context;
    ...
    int game;
    CURL *curl;
};
```

El primer atributo a añadir es `game` y va a indicar cuál es el juego al que el usuario se dispone a jugar. Este atributo es necesario debido al diseño que presenta la REST API en el que tenemos un *endpoint* por cada juego, por tanto `game` va a determinar qué *endpoint* se debe utilizar.

Al igual que ocurre con `game_capture_source` y la dirección de memoria de los marcos de *webcam*, `game` es un ajuste que establece el usuario en las propiedades del plugin. De esta manera, puede crear una instancia del plugin para jugar a Valorant y otra para jugar a Apex o puede usar una misma instancia y cambiar manualmente este ajuste según el juego que esté emitiendo.

Ilustración 30: Ajustes del plugin, videojuego



El segundo atributo que añadimos es `curl`. Este va a actuar como *handle* que usaremos como *input* en nuestras funciones Curl.

5.2.2.3.2 Métodos que aporta este módulo

Los métodos principales que se encuentran en el Comunicador con la API se corresponden con las tareas de enviar una solicitud a la API y recibir una respuesta de la misma.

```
bool send_data_to_api(struct reactive_facecam_plugin *plugin);
```

En primer lugar, desarrollamos el método que realiza el envío de la solicitud a la API. Este método es `send_data_to_api`, el cual se ejecuta en el hilo de `thread_take_screenshot_and_send_to_api` y sucede inmediatamente después de que `render_game_capture` concluya, en el caso de que se haya obtenido la captura de juego correctamente.

La primera alternativa que se desarrolló para el envío de la imagen fue intentar enviar el *stream* de *bytes* en un `string` o en un JSON a la API. Al no obtener ningún éxito con esta alternativa, optamos por la solución provisional de convertir el *stream* de *bytes* en un archivo *raw* y enviar dicho archivo, siguiendo el ejemplo expuesto en la página oficial de Curl [34].

```
curl_easy_setopt(plugin->curl, CURLOPT_READDATA, file);
...
curl_easy_setopt(plugin->curl, CURLOPT_READFUNCTION, read_callback);
curl_easy_setopt(plugin->curl, CURLOPT_WRITEFUNCTION, got_data_from_api);
```

Además, en `send_data_to_api` debemos indicar el archivo que deseamos enviar y las funciones responsables de leer este archivo y de recibir la respuesta de la API respectivamente.

```
size_t read_callback(char *ptr, size_t size, size_t nmemb, void *userdata);
```

La función que lee el archivo a enviar es `read_callback` y debe presentarse en el código porque es un requisito que establece Curl en Windows.

```
size_t got_data_from_api(char *buffer, size_t itemsize, size_t nitems, void*
ignorethis);
```

Por último, el segundo método principal y la función responsable de recibir la respuesta de la API es `got_data_from_api`. En este método, en su primer argumento, `buffer`, es recibido el JSON de donde extraemos la cantidad de vida del jugador. Una vez obtenida esta información, se va a evaluar si es necesario cambiar el marco de *webcam* que se muestra en la Escena. Si es así, el método `change_webcam_frame` será invocado indicando el nuevo marco a mostrar.

6 Implantación

En este apartado se va a explicar cómo preparar el entorno de explotación que nos permitirá desarrollar el código de la aplicación y ejecutar la misma. Primero, se va a comentar la instalación de los requisitos mínimos para el entorno de desarrollo del proyecto y, posteriormente, se indicarán los pasos de instalación para el disfrute del usuario.

6.1 Preparación

6.1.1 REST API

Para poder desarrollar y ejecutar correctamente la API, debemos garantizar que las siguientes librerías se encuentren en el entorno donde va a estar activa. Estas librerías se pueden instalar usando el comando `pip install`.

- flask, versión >= 2.1.2.
- Pillow, versión >= 9.1.1.
- numpy, versión >= 1.22.4.
- matplotlib, versión >= 3.5.2.
- pytesseract, versión >= 0.3.9.

Además, para asegurar el correcto funcionamiento de las funciones de pytesseract, es necesario instalar Tesseract OCR en nuestra máquina.

6.1.2 Reactive Facecam Plugin

Con el fin de preparar un entorno en el que poder desarrollar nuestro plugin de OBS, se deben seguir los siguientes pasos:

1. Instalar Visual Studio Code, CMake y MinGW [35]. Se recomienda añadir la extensión de CMake a Visual Studio Code.

2. Debemos hacer una *build* de OBS desde su código fuente [36, 37], el cual encontramos en GitHub [8].

   Al completar los pasos que se indican en el vídeo tutorial de instalación, o en la guía, tendremos dos nuevas carpetas en nuestra máquina. En la primera de ellas, se encuentra el código de OBS y nos referiremos a la misma como *OBS-STUDIO* al mencionarla a continuación. En la segunda, están las dependencias descargadas durante la instalación y nos referiremos a la misma como *OBS-STUDIO-DEPS*.

Se insiste en que este paso se complete previamente a hacer la *build* de nuestro proyecto, ya que dicha *build* requiere de archivos descargados y generados en este mismo paso.

3. Hacemos la *build* de nuestro proyecto desde el código fuente, también encontrado en GitHub [38].

   Si el plugin se encuentra con un nombre distinto a Reactive Facecam Plugin es porque este nombre es nuevo y el proyecto en GitHub aún está pendiente de renombrar.

   En el momento en que realicemos la *build* utilizando CMake, será necesario añadir las siguientes variables desde CMake antes de generar los *binaries* del proyecto. Nótese que el sistema operativo usado en este caso es Windows de 64 bits.

   LIBOBS variables

   LIBOBS_INCLUDE_DIR, cuyo valor es "*OBS-STUDIO*/libobs".

   LIBOBS_LIBRARY, con valor "*OBS-STUDIO*/build/libobs/Release/obs.lib".

   OBS DEPS variables

   DepsPath, "*OBS-STUDIO-DEPS*/win64".

   OBS_FRONTEND_LIB, "*OBS-STUDIO*/build/UI/obs-frontend-api/Release/obs-frontend-api.lib".

   CURL variables

   CURL_INCLUDE_DIR, "*OBS-STUDIO-DEPS*/win64/include".

   CURL_LIBRARY, "*OBS-STUDIO-DEPS*/win64/bin/libcurl.lib".

   JSON-C variables

   Con el objetivo de facilitar la descarga de json-c, en el proyecto se ha añadido una carpeta comprimida donde se encuentran los archivos necesarios para ejecutar esta librería. Esta carpeta comprimida se encuentra dentro de *deps* en la estructura de nuestro proyecto. Asumimos que esta carpeta es nombrada *JSON-C-FOLDER* al descomprimirla.

   JSON_C_INCLUDE_DIR, "*JSON-C-FOLDER*".

   JSON_C_LIBRARY, "*JSON-C-FOLDER*/build/libjson-c.a".

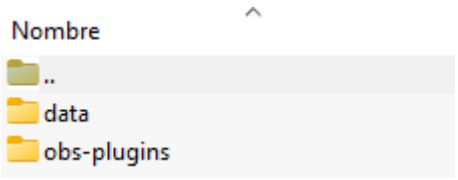
Una vez hayamos añadido las variables, ya podremos modificar el código del proyecto y generar los archivos binarios del mismo desde CMake. Estos archivos serán usados a continuación para instalar el plugin en OBS.

6.2 Instalación

El proceso de instalación de Reactive Facecam Plugin en OBS es sencillo e idéntico a la instalación de cualquier otro plugin.

Nuestro plugin, listo para instalarse, se puede encontrar en el apartado de *Releases* de su página en GitHub con el número de versión 0.0.1. Dicha versión se trata de una carpeta comprimida donde se encuentran los archivos que definen nuestro plugin y están ordenados de una manera muy específica.

Ilustración 31: Carpetas versión Release del plugin



En la primera carpeta, *data*, se disponen aquellos recursos del plugin que no son código. Estos son archivos donde se encuentran las traducciones de textos que presenta el proyecto, como los campos de "Ajustes" que el usuario puede editar, y otros archivos, como puede ser multimedia que usa el plugin. En nuestro caso, es aquí donde almacenamos los marcos de *webcam*, los cuales facilitamos al usuario en el caso de que no utilice marcos propios, y el archivo *raw* en el que se escribe el *stream* de *bytes* de la imagen antes de ser enviada a la API.

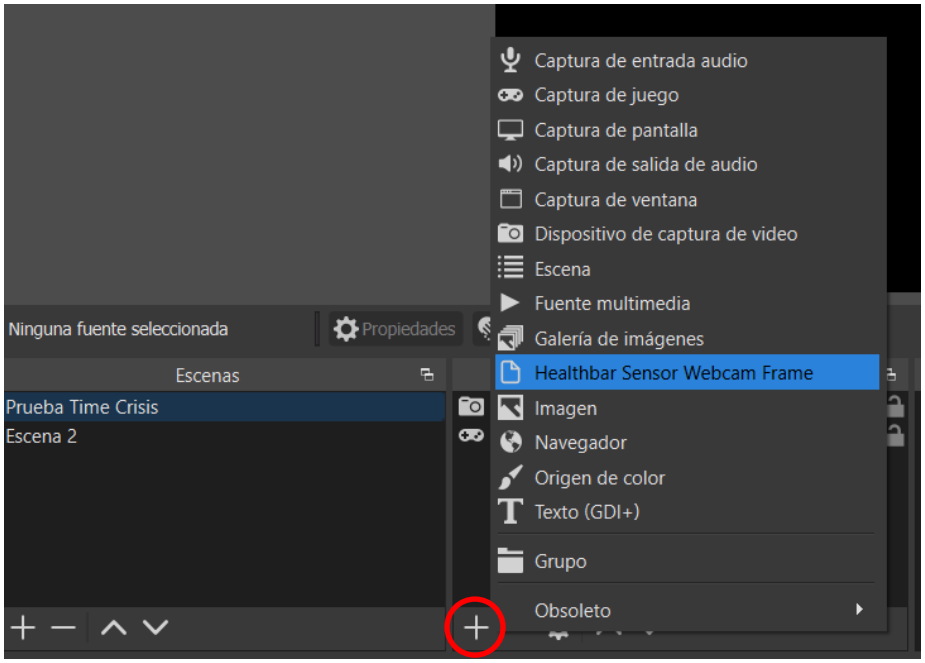
La segunda carpeta, *obs-plugins*, contiene el binario generado con CMake en el paso 3 del apartado anterior. Este binario es *libobs-healthbar-sensor-webcam-frame.dll* y se trata del plugin en sí. De nuevo, el archivo está pendiente de renombrar a *libreactive-facecam-plugin.dll*.

Para la instalación del plugin debemos arrastrar ambas carpetas a un directorio específico dentro del programa OBS. Si estamos utilizando la versión de OBS construida a partir del código fuente, este directorio es la carpeta *Release* y se sitúa en "*OBS-STUDIO*/build/rundir/Release". Por el contrario, si usamos OBS descargado directamente de su página oficial, el directorio donde arrastraremos las carpetas será "Program Files/obs-studio".

Finalmente, ejecutamos nuestro plugin en OBS de la siguiente forma:

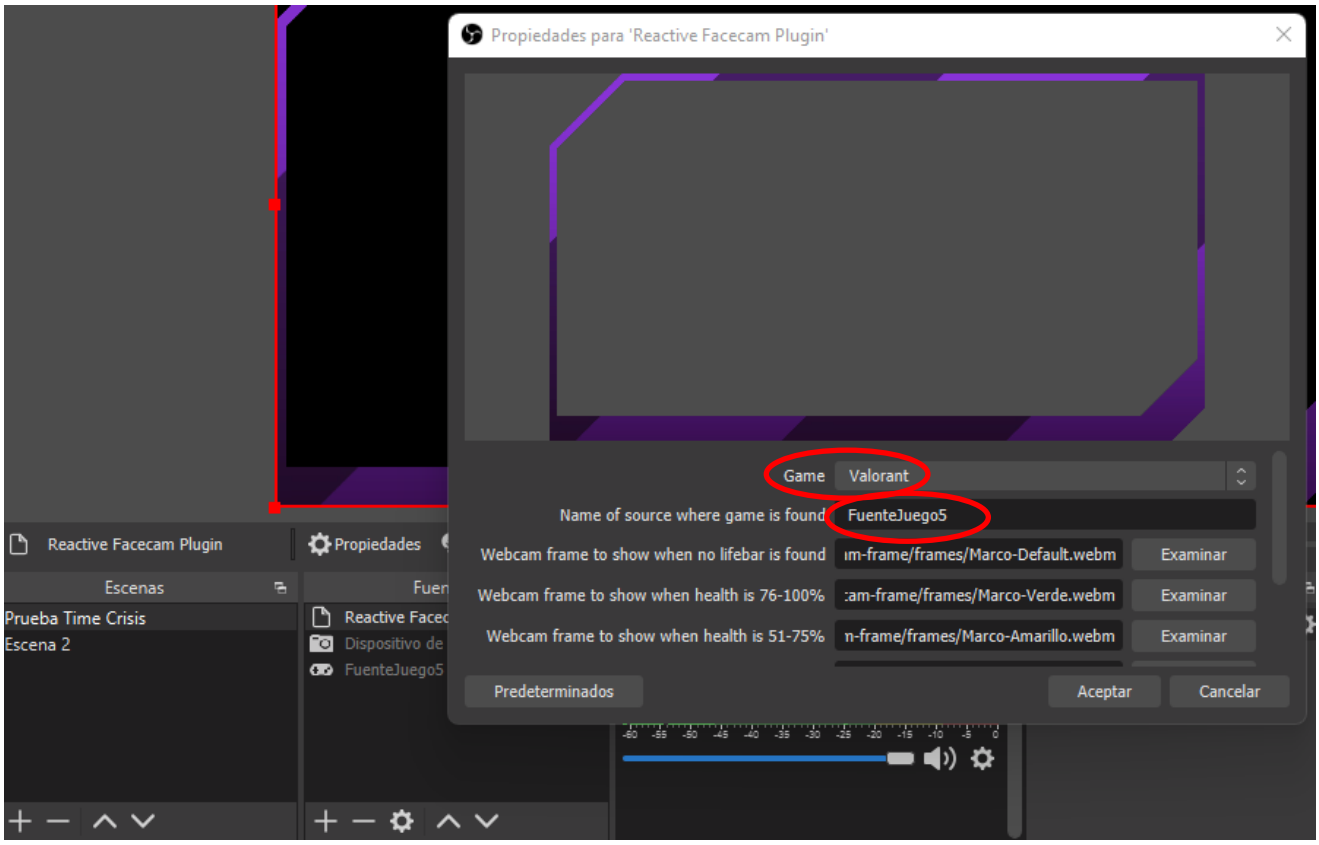
1. Añadimos el plugin como una Fuente más en nuestra Escena.

Ilustración 32: Plugin ubicado en la lista de Fuentes disponibles



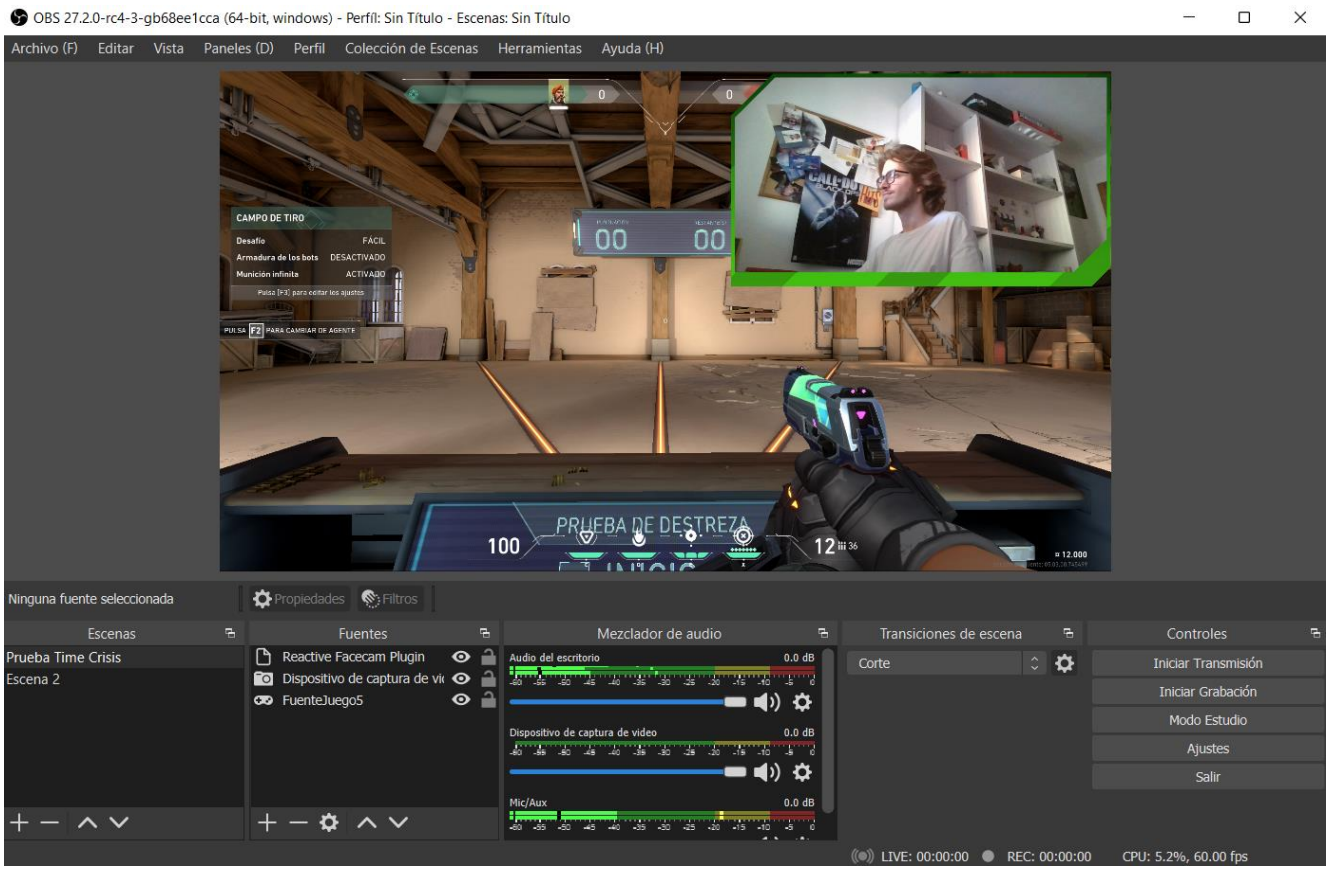
2. Configuramos el plugin estableciendo el juego que nos disponemos a jugar y la Fuente donde se puede ver dicho juego.

Ilustración 33: Ventana de ajustes del plugin



3. Ajustamos el marco de *webcam* en la Escena para que tenga la ubicación y el tamaño que deseemos.

Ilustración 34: Plugin ubicado en una Escena



En la máquina donde queramos ejecutar la API, únicamente tenemos que cumplir los requisitos expuestos en el apartado de Preparación y ya podremos ejecutarla con Python. Téngase en cuenta que el plugin cambiará el marco de la *webcam* de acuerdo a la vida del jugador siempre que la API esté activa.

7 Pruebas

En este apartado se van a exponer las pruebas que se han realizado en el proyecto y las que se propone realizar en el futuro. Dichas pruebas tienen como objetivo garantizar que el plugin cumple los requisitos establecidos y que el correcto funcionamiento del mismo se cumple en las próximas versiones.

7.1 Pruebas realizadas

Las pruebas que se han llevado a cabo para esta primera versión no han sido automatizadas. Se tratan, pues, de pruebas de rendimiento y varios *performance test* con 3 usuarios cuyas máquinas presentan características distintas.

Para obtener los datos que se van a presentar a continuación, grabamos una partida de 40 minutos de Valorant y dos partidas de 7 y 8 minutos de Apex Legends. En estas partidas, los 3 usuarios y yo jugamos utilizando Reactive Facecam Plugin y ejecutando cada uno la REST API en local.

7.1.1 Pruebas de rendimiento

Diremos que Reactive Facecam Plugin tiene un buen rendimiento si se cumplen los requisitos no funcionales de eficiencia y tiempos de respuesta bajos. Esto es que el plugin debe utilizar la cantidad mínima de recursos de la máquina del usuario y que, frente a un cambio en la vida del jugador, se debe reaccionar lo antes posible.

En primer lugar, es de nuestro interés garantizar la eficiencia del plugin dado que una falta de la misma puede ocasionar la reducción de la calidad del *streaming* del usuario y la limitación en el uso de otros programas que desee utilizar. Con el objetivo de comprobar el cumplimiento de este requisito, se debe medir la cantidad de recursos que consume OBS mientras ejecuta el plugin.

Generalmente, en una ejecución en la que tenemos 3 Fuentes (Captura de juego, Webcam y nuestro plugin) en una Escena, se ha obtenido que OBS consumiría alrededor de 275 MB de memoria RAM usando los ajustes óptimos para *streaming*. En cuanto al uso de CPU y GPU, debemos monitorizarlo en la siguiente versión del proyecto porque ahora mismo no podemos determinar su uso exacto. Sin embargo, los usuarios que han probado el plugin declaran que el uso de CPU y GPU de OBS es "ligeramente mayor" al utilizar Reactive Facecam Plugin, pero no compromete la calidad del *streaming*.

Y, segundo, tenemos que asegurar que los tiempos de respuesta en la comunicación que hay del plugin a la API son bajos porque los usuarios desean que el marco de *webcam* cambie tan pronto como cambie su barra de vida en el juego. Seguidamente, se muestran los tiempos de respuesta mínimo, máximo y medio en una partida de Valorant y en una partida de Apex respectivamente.

Valorant

- T. respuesta mínimo = 0,02626 segundos
- T. respuesta máximo = 1,49366 segundos
- T. respuesta medio = 0,22564 segundos

Apex

- T. respuesta mínimo = 0,02892 segundos
- T. respuesta máximo = 1,24893 segundos
- T. respuesta medio = 0,09897 segundos

De estos datos podemos observar que el tiempo de respuesta mínimo para ambos juegos es similar, ya que la manera en la que la API determina si debe leer la vida o no es idéntica. Sin embargo, los tiempos máximo y medio de Valorant son mucho mayores que los de Apex porque, para leer la vida en una captura del primero, utilizamos Tesseract, lo cual retrasa un poco la respuesta.

Igualmente, los usuarios que han probado el plugin declararon que los cambios del marco de *webcam* en Valorant "no son tan inmediatos" como se esperaría. Por lo tanto, se debe insistir en la reducción del tiempo de respuesta para Valorant en futuras versiones del plugin.

7.1.2 Performance test

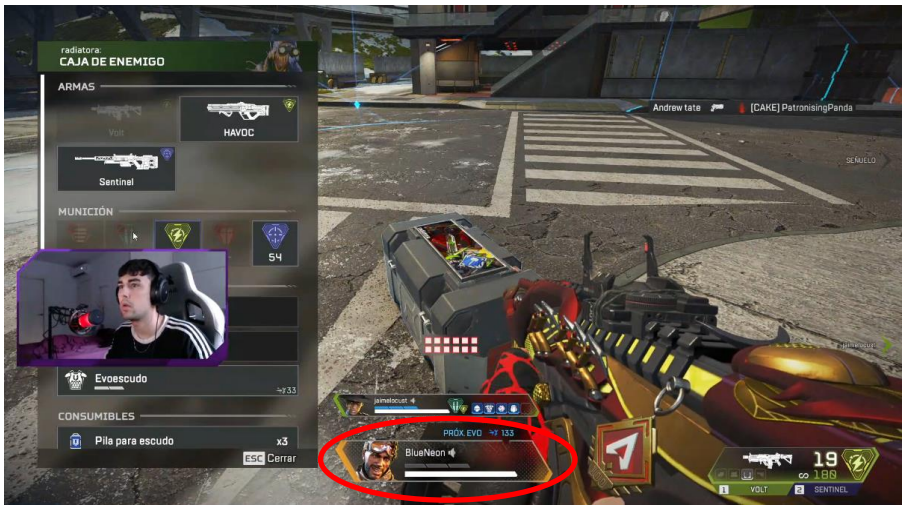
Tras comprobar el rendimiento del proyecto, probamos el plugin en el escenario para el cual se desarrolló, *streaming* de videojuegos, y analizamos las grabaciones de su *performance*. Con el fin de evaluar el funcionamiento de Reactive Facecam Plugin, hemos apuntado los fallos encontrados en dichas grabaciones. Estos fallos pueden ser de dos tipos: primero, no hay motivo para que el marco cambie y se cambia, y segundo, el marco debe cambiarse y no se cambia.

El primero de ellos se debe a una incorrecta lectura de la imagen por parte de la API. Este fallo lo podemos encontrar en Valorant en aquellas situaciones en las que deseamos leer elementos de la pantalla que, al binarizar la imagen, no podemos distinguir.

Ilustración 35: Performance test, fallo al no identificar modo espectador

En esta imagen, por ejemplo, el jugador está muerto pero el marco ha cambiado interpretando que la vida del jugador se encuentra al 33%. La lectura incorrecta de la vida se ha debido a que no se ha podido distinguir la marca que indica que está en modo espectador. También, podemos encontrar este primer tipo de fallo en Apex Legends. No obstante, este fallo ocurre en Apex solo en aquellas situaciones en las que el HUD presenta un cambio *in-game* y no lo hemos tenido en cuenta.

Ilustración 36: Performance test, fallo al cambiarse el HUD

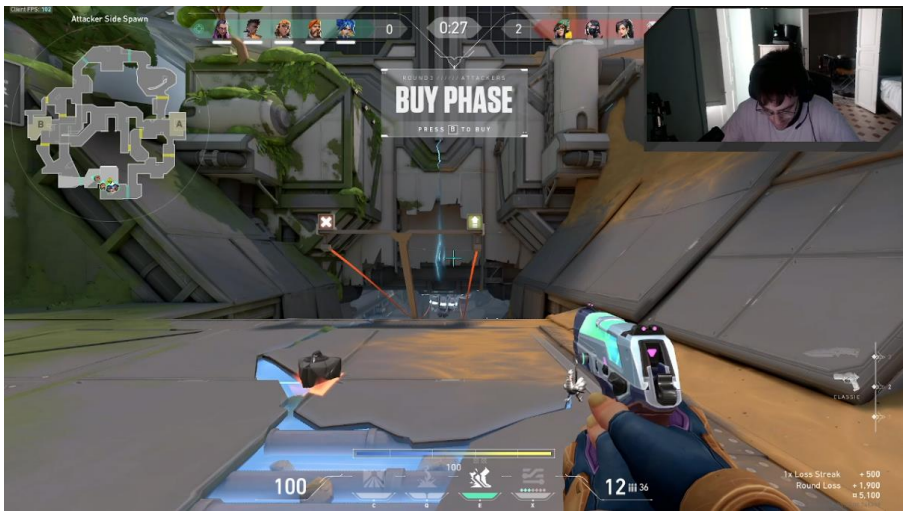


En la imagen anterior, el marco cambió al color morado indicando que no se ha podido encontrar la barra de vida en pantalla. Dicha situación se ha dado, sencillamente, porque la barra de vida cambia de ubicación cuando ves el equipamiento de un jugador muerto.

Al visualizar los vídeos de cada una de las partidas grabadas, determinamos que suceden alrededor de 12 de estos fallos (el marco no debe cambiarse y se cambia) en una partida de 40 minutos de Valorant y 1 fallo en una partida de 8 minutos de Apex.

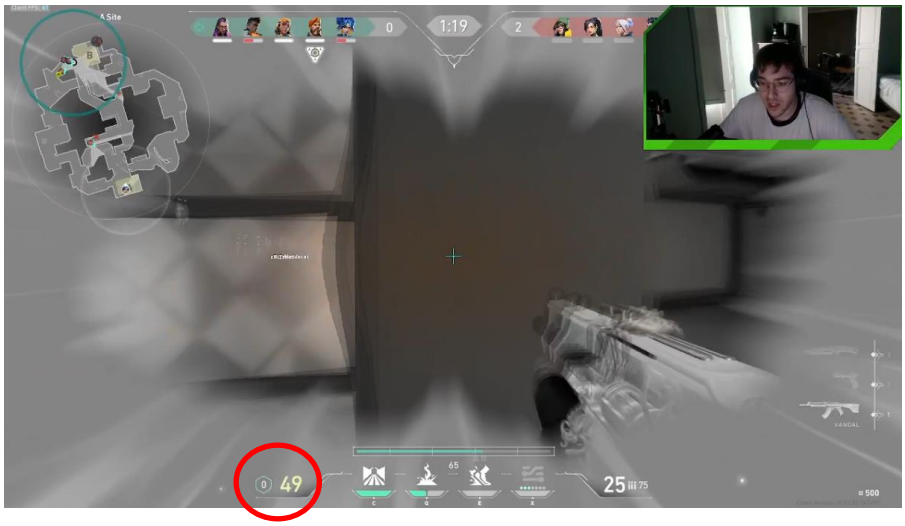
El segundo tipo de fallo que podemos encontrar, el marco debe cambiarse y no se cambia, se puede dar por varios motivos. En Valorant, generalmente, sucede debido a que el tiempo de respuesta de la API es demasiado elevado, lo cual genera situaciones como la que se muestra en la imagen a continuación.

Ilustración 37: Performance test, fallo causado por tiempo de respuesta API



En la imagen podemos ver el marco indicando que el jugador está muerto, sin embargo el jugador ya ha reaparecido y tiene la vida al completo. Otro motivo por el cual se puede dar este tipo de fallo es que la imagen enviada a Tesseract no es suficientemente buena. Esta situación se da cuando la imagen se ve alterada por elementos del juego, como pueden ser explosiones.

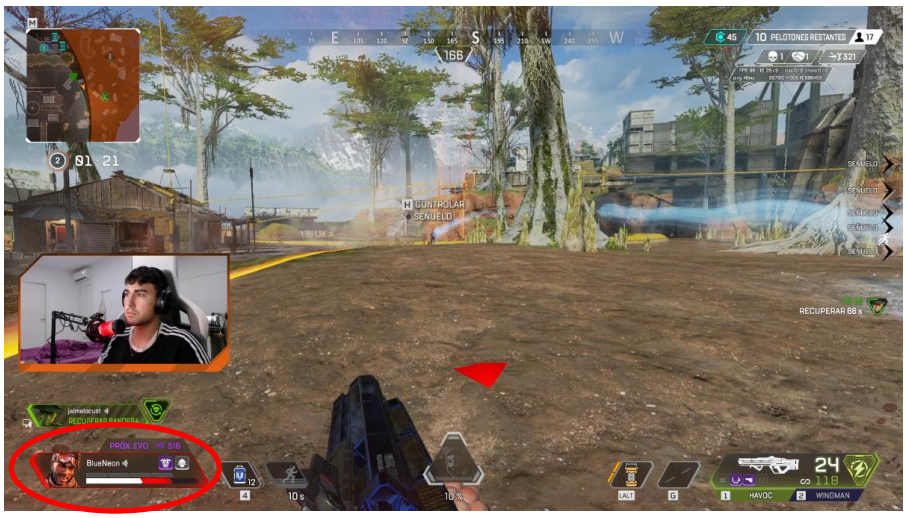
Ilustración 38: Performance test, fallo causado por distorsión de imagen



Aquí el marco debe cambiar para indicar la nueva cantidad de vida del jugador, pero no se ha producido el cambio debido a que Tesseract tiene dificultades para identificar el número que aparece en pantalla.

Afortunadamente, en Apex, este segundo tipo de fallo no ha ocurrido en ninguna de las partidas grabadas. En la siguiente imagen observamos como el marco del jugador está reflejando la nueva cantidad de vida justo después de ser disparado en un combate.

Ilustración 39: Performance test, resultado esperado



En conclusión, en base a los fallos recogidos al visualizar las grabaciones, podemos interpretar que la *performance* del plugin en juegos cuya vida se muestra con una barra es buena, pero mejorable ya que hay situaciones que no hemos apreciado en las que la barra sí está en pantalla. En cuanto a los juegos con vida numérica, la *performance* del plugin ha sido pobre. Como ya se ha mencionado anteriormente, el tiempo de respuesta debe ser reducido y, además, se debe mejorar la manera en la que la API determina si la vida se encuentra en pantalla o no.

7.2 Pruebas futuras

Dado que no se han desarrollado pruebas automatizadas, en este apartado vamos a proponer un Plan de Testing inicial para nuestro proyecto.

7.2.1 REST API

Primeramente, proponemos que se desarrollen las siguientes pruebas para comprobar el correcto funcionamiento de la REST API.

Como se sugiere en la documentación de Flask [39], usaremos las librerías pytest y coverage para desarrollar tests unitarios y asegurar la mayor cobertura posible. Además, para garantizar que se cumplen las expectativas del usuario, se harán tests de caja negra utilizando una galería de imágenes y comprobando que se obtienen las respuestas esperadas. El objetivo de utilizar una galería de imágenes es plantear

situaciones en las que la API tiene dificultades para determinar la vida del jugador, como las destacadas en el apartado *Performance test*.

7.2.2 Reactive Facecam Plugin

En segundo lugar, sugerimos que se realicen las siguientes pruebas para garantizar la calidad del plugin desarrollado. Es cierto que se ha comprobado manualmente el funcionamiento del proyecto en múltiples situaciones, pero es necesario el desarrollo de test unitarios y de integración.

Ilustración 40: Ejemplo de uso del plugin en Valorant, vida al 36%

Ilustración 41: Respuesta de la API obtenida con Ilustración 40

```
13:01:00.029: HSWF - got_data_from_api: {
13:01:00.029:   "errorMessage": "",
13:01:00.029:   "isImageIdentified": true,
13:01:00.029:   "isLifeBarFound": true,
13:01:00.029:   "lifePercentage": 0.36
13:01:00.029: }
```

Aquí observamos como la vida del jugador se encuentra al 36%, por lo que el JSON recibido de la API presenta *lifePercentage* con el valor 0.36 y el plugin muestra el marco anaranjado.

Ilustración 42: Ejemplo de uso del plugin en Apex, vida al 0%

Ilustración 43: Respuesta de la API obtenida con Ilustración 42

```
13:08:50.662: HSWF - got_data_from_api: {
13:08:50.662:   "errorMessage": "",
13:08:50.662:   "isImageIdentified": true,
13:08:50.662:   "isLifeBarFound": true,
13:08:50.662:   "lifePercentage": 0.0
13:08:50.662: }
```

En esta imagen el jugador ha sido derribado, por lo que el JSON recibido de la API presenta *lifePercentage* con el valor 0.0 y el plugin muestra un marco negro.

En el caso de los test unitarios a desarrollar, se propone testear cada uno de los módulos del plugin por separado siguiendo la técnica de caja blanca. De esta manera, obtendremos una cobertura completa respecto a las ramas de ejecución del programa. Un *framework* que nos puede servir para escribir test que midan la cobertura de rama es Cmocka [40].

Asimismo, se deben testear las tareas de las que se encarga cada módulo. Por ejemplo, debemos comprobar que el Reproductor de vídeo:

- Al inicio de ejecución, muestra el marco *default*.
- Al invocarse `change_webcam_frame`, se cambia el marco que se esté reproduciendo por el indicado.
- Al establecer nuevos marcos el usuario, los valores de los parámetros `frame_path` del plugin deben cambiar.
- Debemos asegurar el correcto funcionamiento de las funciones de pausar y reanudar el vídeo.
- Debemos asegurar que las funciones para establecer *hotkeys* de pausar y reanudar modifican los ajustes de OBS como se espera.

En el caso del desarrollo de test de integración, podemos tomar el subdirectorio de tests de OBS y las pruebas del plugin obs-gstreamer como referencia [41, 42]. En los tests de estos dos proyectos, se comprueba únicamente si el plugin puede ser añadido y

eliminado de la Escena. Nuestro objetivo en los test de integración es confirmar esto mismo y, también, que los módulos del plugin se comuniquen entre ellos y con nuestra API.

8 Conclusiones

Para concluir, me gustaría decir que este proyecto me ha traído mucha felicidad. Ha sido todo un orgullo desarrollar este producto y ver cómo ha gustado a varios usuarios cuando lo compartí por redes sociales. Es más, como programador y usuario de Twitch, no hay nada que me agrade más que ver algo que he construido suponiendo una pequeña mejora en la experiencia de *streaming* de otra persona.

Desearía destacar la ayuda y guía que recibí de la mano de mi tutor, Joan Josep Fons. El profesor Fons ha hecho posible que un pequeño *hobby*, al que le dedicaba algo de tiempo, se convierta en un trabajo de fin de grado. Le estoy muy agradecido porque este proyecto no habría visto la luz este año de no haber sido por su apoyo.

Respecto al cumplimiento de objetivos del proyecto, podemos afirmar que han sido alcanzados. El resultado de este trabajo es la extensión de OBS lista para usarse en *streaming*. Además, cabe destacar la sencillez con la que podemos añadir un juego más que el producto pueda procesar, ya que inicialmente solo se iba a poder usar para Apex, pero en tan solo dos días se pudo añadir Valorant a la lista de juegos permitidos.

Sin embargo, el producto exige que la API se esté ejecutando para funcionar y, en estos momentos, la API no se está hosteando en la nube. El motivo por el que la API no se encuentra publicada es porque el tamaño de las solicitudes recibidas es demasiado grande. Su elevado tamaño se debe a que necesita recibir capturas de juego, así que se tiene que tratar este problema en Trabajos futuros.

Finalmente, se han de abordar las dificultades encontradas y el aprendizaje que ha supuesto este trabajo.

En cuanto a dificultades, me gustaría señalar las siguientes:

- Falta de guías oficiales para preparar el entorno de desarrollo en OBS.
- Escasez de documentación en cuanto a plugins se refiere.
- Falta de un estándar de programación, lo cual dificulta la lectura del código de distintos proyectos.
- Falta de una plantilla a partir de la cual se pueda crear una nueva Fuente. Sé que este último punto no es necesario, pero creo que facilitaría mucho la iniciación de nuevos desarrolladores en OBS.

En resumen, las dificultades de este proyecto han supuesto que el desarrollo del mismo exija tiempo y paciencia. No obstante, como ingeniero he aprendido lo que es entender un sistema por ti mismo, ya que no hay ninguna guía que te lo explique, y a investigar de forma exhaustiva para descubrir las modificaciones que el sistema te permite realizar.

Desconozco si este es el caso que se da en el resto de proyectos *open-source*. Igualmente, recomiendo a todo amante de la programación que alguna vez participe en el desarrollo de un proyecto de código abierto. Dicha experiencia es enriquecedora en cuanto aprendizaje, ya que es bueno verse en situaciones en las que la solución que buscas no se encuentre en Google, y en cuanto a la comunidad de desarrolladores que lo mantienen. En el caso de OBS, los miembros de su comunidad son realmente amables.

8.1 Relación del trabajo desarrollado con los estudios cursados

A lo largo de la carrera, se han adquirido una serie de conocimientos que han sido vitales para el desarrollo de este trabajo. Los estudios cursados, tanto la materia directamente relacionada con el código como la aprendida para que el alumno tenga un pensamiento crítico, han marcado una diferencia en el proyecto.

En primer lugar, todo el contenido del apartado de Análisis del problema ha sido redactado atendiendo a lo aprendido en AER, donde se enseña a especificar requisitos, analizar riesgos y plantear los modelos de dominio y de contexto. Además, han resultado de gran ayuda las asignaturas de CSO, para la identificación de *stakeholders*, y GPR, para plantear las etapas del proyecto.

Segundo, durante la etapa de desarrollo cabe destacar la utilidad de CSD, en la cual se aprendió a manejar correctamente varios hilos de ejecución en paralelo, IEI, donde se imparten los conocimientos para el desarrollo de APIs, y las asignaturas de PIN y PSW, donde tenemos contacto directo con la gestión del proceso de desarrollo de un proyecto.

Finalmente, por cuestión de tiempo en nuestro trabajo no hemos podido realizar la etapa de testing. No obstante, las pruebas planteadas en el apartado Pruebas futuras, así como las librerías para crearlas, han sido propuestas según los conocimientos de testing adquiridos en MES, AVD e ISW.

8.2 Trabajos futuros

En este apartado, se van a enumerar las tareas restantes para dar el proyecto por terminado.

- La API debe hostearse en algún servicio en la nube como es AWS EC2 [43].

  Previamente, se ha intentado utilizar AWS Lambda, pero este servicio limita el tamaño de las solicitudes y respuestas a 6 MB [44].

  En el caso de que la API no funcione correctamente en EC2, se reducirá el tamaño de las imágenes enviadas a la misma o se acoplarán las tareas realizadas por la API al mismo plugin.

- Se deben terminar las refactorizaciones de la REST API y de Reactive Facecam Plugin.

- El procesado de la API en juegos cuya vida es numérica debe mejorarse, teniendo en cuenta los fallos destacados en el apartado de Pruebas realizadas.

- Se deben desarrollar los test indicados en el apartado de Pruebas futuras.

- Reactive Facecam Plugin debe monitorizar la cantidad de CPU y GPU que está consumiendo en la máquina cliente.

- Se deben monitorizar la cantidad de peticiones respondidas con éxito por la API, así como su tiempo de respuesta.

- Los marcos del plugin deben presentar animaciones de transición al cambiar de un marco a otro.

- Se deben añadir nuevos marcos de *webcam* más atractivos que los sencillos marcos presentados actualmente por el plugin.

- Se debe actualizar la página de GitHub del plugin. Especialmente, el README que encontramos en nuestro código.

9 Bibliografía

[1] Jason Wise, «Twitch Statistics 2022: How many people use Twitch?» 06 09 2022. Available: https://earthweb.com/twitch-statistics/#:~:text=Key%20Twitch%20Statistics%20for%202022

[2] Brian Dean, «Twitch Usage and Growth Statistics: How Many People Use Twitch in 2022?» 05 01 2022. Available: https://backlinko.com/twitch-users

[3] Twitch, «Recommended Software for Broadcasting» 2022. Available: https://help.twitch.tv/s/article/recommended-software-for-broadcasting

[4] Matías Patiño, «Blue Design on Behance» 14 12 2021. Available: https://www.behance.net/MatBlue

[5] Exeldro, «Move transition» 28 03 2020. Available: https://obsproject.com/forum/resources/move-transition.913/

[6] Exeldro, «Source Record» 13 03 2021. Available: https://obsproject.com/forum/resources/source-record.1285/

[7] Discord, «OBS Community» 2022. Available: https://discord.com/invite/obsproject

[8] obsproject, «obs-studio» 06 09 2022. Available: https://github.com/obsproject/obs-studio

[9] OBS Studio, «Resources» 2022. Available: https://obsproject.com/forum/resources/

[10] OBS Studio, «Getting Started With OBS Studio Development» 2022. Available: https://obsproject.com/wiki/Getting-Started-with-OBS-Studio-Development

[11] OBS Studio, «Welcome to OBS Studio's documentation!» 2022. Available: https://obsproject.com/docs/index.html

[12] Michael Reeves, «A Robot Shoots Me When I Get Shot in Fortnite» 30 08 2022. Available: https://www.youtube.com/watch?v=D75ZuaSR8nQ

[13] Point zero, «Project Lachlan – Case Study» Winter 2020. Available: https://pointzero.design/work/lachlan

[14] Ben Sledge, «Fortnite streamer Lachlan's new facecam will reflect his in-game health» 08 01 2021. Available: https://www.theloadout.com/fortnite/lachlan-facecam-mod

[15] HetIsJoey, «NowPlaying Widget [Spotify/StreamElements]» 16 11 2020. Available: https://obsproject.com/forum/resources/nowplaying-widget-spotify-streamelements.1138/

[16] dustractor, «Explorer Select Last Recording» 18 01 2021. Available: https://obsproject.com/forum/resources/explorer-select-last-recording.1309/

[17] scaled, «Image Reaction» 20 08 2021. Available: https://obsproject.com/forum/resources/image-reaction.1342/

[18] obsproject, «JS Bindings» 11 04 2022. Available: https://github.com/obsproject/obs-browser/blob/master/README.md#js-bindings

[19] MattPlaysTV, «Coding Live: OBS / TS2016 Cab Plugin #1» 06 05 2016. Available: https://www.youtube.com/watch?v=Sf4ySfbjjv4

[20] Mark Miller, «e459 - KidzCode - Controlling OBS in C#» 25 05 2021. Available: https://www.youtube.com/watch?v=fg1ynwXxO8I

[21] VidGrid, «libobs-sharp» 16 11 2020. Available: https://github.com/ilosvideos/libobs-sharp

[22] OBS Studio, «OBS Studio Backend Design» 2022. Available: https://obsproject.com/docs/backend-design.html

[23] Dustin Bailey, «Barely 6% of Steam users play above 1080p» 02 10 2018. Available: https://www.pcgamesn.com/steam-hardware-survey-resolution

[24] LibHunt, «Top 23 OCR Open-Source Projects» 2022. Available: https://www.libhunt.com/topic/ocr

[25] Ray Smith, Daria Antonova, Dar-Shyang Lee, «Adapting the Tesseract Open Source OCR Engine for Multilingual OCR» 25 07 2009. Available: https://static.googleusercontent.com/media/research.google.com/es//pubs/archive/35248.pdf

[26] mooware, «Lightgun Flash Filter» 26 09 2021. Available: https://obsproject.com/forum/resources/lightgun-flash-filter.1366/

[27] mooware, «FindLibObs.cmake» 13 08 2021. Available: https://github.com/mooware/OBS-Studio-Flash-Filter/blob/master/external/FindLibObs.cmake

[28] obsproject, «obs-ffmpeg-source.c» 28 07 2022. Available: https://github.com/obsproject/obs-studio/blob/master/plugins/obs-ffmpeg/obs-ffmpeg-source.c

[29] obsproject, «media-playback» 01 08 2022. Available: https://github.com/obsproject/obs-studio/tree/master/deps/media-playback

[30] OBS Studio, «OBS Studio Frontend API» 2022. Available: https://obsproject.com/docs/reference-frontend-api.html

[31] synap5e, «obs-screenshot-plugin» 18 03 2022. Available: https://github.com/synap5e/obs-screenshot-plugin

[32] OBS Studio, «Core Graphics API» 2022. Available: https://obsproject.com/docs/reference-libobs-graphics-graphics.html

[33] OBS Studio, «Rendering Video Effect Filters» 2022. Available: https://obsproject.com/docs/graphics.html#rendering-video-sources

[34] cURL, «libcurl example - fileupload.c» 17 05 2022. Available: https://curl.se/libcurl/c/fileupload.html

[35] geek_eng, «Set up VS Code and Debug C/C++ application on Windows using VScode, CMake and MinGW» 10 09 2021. Available: https://www.youtube.com/watch?v=6QVjIZTDG8c

[36] OBS Studio, «Install Instructions» 2022. Available: https://obsproject.com/wiki/install-instructions

[37] TroubleChute, «Building OBS from Source Code | Complete Guide» 24 08 2021. Available: https://www.youtube.com/watch?v=p8TV9-MdMoY

[38] manjavacasjaime, «obs-healthbar-sensor-webcam-frame» 18 08 2022. Available: https://github.com/manjavacasjaime/obs-healthbar-sensor-webcam-frame

[39] Flask, «Test Coverage» 2022. Available: https://flask.palletsprojects.com/en/2.0.x/tutorial/tests/

[40] cmocka, «cmocka – unit testing framework for C» 2022. Available: https://cmocka.org/

[41] obsproject, «test.cpp» 09 03 2020. Available: https://github.com/obsproject/obs-studio/blob/master/test/win/test.cpp

[42] fzwoch, «obs-gstreamer» 17 07 2022. Available: https://github.com/fzwoch/obs-gstreamer/blob/master/test/main.c

[43] Amazon, Inc, «Amazon EC2» 2022. Available: https://aws.amazon.com/es/ec2/

[44] Amazon, Inc, «Lambda quotas» 2022. Available: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

[45] TwitchTracker, «TWITCH GAMES STATISTICS» 2022. Available: https://twitchtracker.com/statistics/games

10 Índice de ilustraciones

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	**Alto**	**Medio**	**Bajo**	**No Proceed e**
ODS 1. **Fin de la pobreza.**				**X**
ODS 2. **Hambre cero.**				**X**
ODS 3. **Salud y bienestar.**				**X**
ODS 4. **Educación de calidad.**				**X**
ODS 5. **Igualdad de género.**				**X**
ODS 6. **Agua limpia y saneamiento.**				**X**
ODS 7. **Energía asequible y no contaminante.**				**X**
ODS 8. **Trabajo decente y crecimiento económico.**				**X**
ODS 9. **Industria, innovación e infraestructuras.**		**X**		
ODS 10. **Reducción de las desigualdades**.				**X**
ODS 11. **Ciudades y comunidades sostenibles**.				**X**
ODS 12. **Producción y consumo responsables.**		**X**		
ODS 13. **Acción por el clima.**				**X**
ODS 14. **Vida submarina.**				**X**
ODS 15. **Vida de ecosistemas terrestres.**				**X**
ODS 16. **Paz, justicia e instituciones sólidas.**				**X**
ODS 17. **Alianzas para lograr objetivos.**				**X**

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

Los ODS (Objetivos de Desarrollo Sostenible) tienen como meta acabar con la pobreza, proteger el planeta y mejorar las vidas y las perspectivas de las personas. Para ello, se proponen una serie de objetivos cuyo cumplimiento harían la vida más fácil a todo el mundo.

Como ingenieros informáticos, tenemos el deber de seguir estos objetivos en aquellos proyectos que desarrollemos. En nuestro caso, hemos generado un proyecto software que no va a resultar especialmente influyente en los ODS, dado que se trata de una extensión de un programa para el cual únicamente hemos necesitado un ordenador en su creación y se necesitará un ordenador para su consumo. Sin embargo, nuestro proyecto ha tenido un pequeño impacto en los siguientes ODS.

El OBS 9 (Industria, innovación e infraestructura) señala que necesitamos promover la innovación, el crecimiento tecnológico, y el desarrollo para dar rienda suelta a las fuerzas económicas, dinámicas y competitivas que generan ingresos.
Hemos marcado un grado Medio de relación porque uno de los objetivos del proyecto es favorecer el progreso tecnológico en un entorno muy específico. De hecho, parte de la motivación de este proyecto es dar acceso a todo el mundo a un producto que solo una persona podía utilizar. Así pues, la detallada redacción de este trabajo y la publicación del código del mismo se han hecho con el fin de promover el desarrollo de otros proyectos similares.
El grado marcado ha sido Medio en lugar de Alto, dado que el entorno donde se está promoviendo el desarrollo es muy específico.

El OBS 12 (Producción y consumo responsables) defiende el uso eficiente de los recursos, las infraestructuras y la energía para, así, alcanzar un consumo y una producción sostenibles.
Hemos indicado que el proyecto presenta un grado Medio de relación con el mismo porque uno de nuestros requisitos es hacer un uso eficiente de los recursos de la máquina cliente. El cumplimiento de este requisito no solo favorece una mejor calidad del producto, sino que reduce el consumo energético del usuario que lo utilice y, debido a su ejecución eficiente, favorece un mayor tiempo de vida del ordenador donde se esté usando.
El grado marcado ha sido Medio en lugar de Alto, ya que de forma directa se reduce el consumo energético del cliente, pero no se está promoviendo el uso de energías renovables ni las infraestructuras ecológicas ni nada por el estilo.

En conclusión, todo proyecto software va a tener una influencia, directa o indirecta, en los ODS y nuestro objetivo debe ser que esta influencia sea positiva. Para ello, es aconsejable que el desarrollador identifique aquellos objetivos en los que el sistema puede influir, considere su impacto y establezca unas medidas para hacer que su influencia sea positiva, de forma similar al proceso de Análisis de riesgos.
Además, debemos tener estos objetivos en mente a la hora de utilizar software de terceros como son las librerías externas. Es nuestra responsabilidad preguntarnos si los productos que estamos utilizando también ejercen una influencia positiva en los ODS.