INSTITUTO TECNOLÓGICO DE BUENOS AIRES – ITBA

ESCUELA DE INGENIERÍA Y GESTIÓN

SUNSHINE

Game streaming en Windows

AUTOR/ES: Raies Tomás (Leg. Nº 56099)

DOCENTE/S TITULAR/ES O TUTOR/ES: Lorenzon, Jorge Alejandro

TRABAJO FINAL PRESENTADO PARA LA OBTENCIÓN DEL TÍTULO DE

INGENIERO EN INFORMÁTICA

Lugar: Ciudad de Buenos Aires

Fecha: 20/11/2019

Resumen

Presentamos SUNSHINE, una nueva herramienta de *software* libre que permite realizar *game streaming* en Windows, sacando provecho de *hardware* especializado presente en las tarjetas de vídeo comerciales para realizar la captura, codificación y decodificación de los cuadros de vídeo, con lo que se consiguió una latencia de procesamiento inferior a 60 ms. Se ha logrado una experiencia de juego comparable a la nativa en un contexto de red LAN hogreña típica utilizando *access points* WiFi de 5.8 GHz. y hasta un 42 % menor *input lag* que Steam Remote Play, una de las aplicaciones de juego remoto comerciales más populares en el mercado.

Para este desarrollo se realizó una investigación profunda acerca de las peculiaridades del sistema operativo que permiten la inyección de código en procesos ajenos mediante funciones de Win32. Utilizando *hooks* que intercepten las llamadas al sistema dentro de los videojuegos ejecutados, se pudo realizar la captura de vídeo directamente desde el *frame buffer* de la aplicación, así como también intervenir en el flujo de mensajes del sistema para permitir la manipulación del programa por parte del usuario remoto. De esta forma, se logro un sistema donde los procesos de captura, codificación, transmisión, recepción, decodificación y reproducción de los cuadros de video toman un tiempo de procesamiento muy bajo.

Índice general

1. Introducción **1**

1.1. Productos comerciales 2

1.2. Objetivos 5

2. Analisis **9**

2.1. Requerimientos 9

2.2. Arquitectura 11

3. Detalles técnicos **13**

3.1. *DLL injection* 13

3.2. *JMP hooking* 15

3.3. Entrada del ratón y el teclado 16

3.4. *Raw input* 21

3.5. Captura de vídeo 24

3.6. Reproducción de vídeo 30

3.7. *Networking* 32

4. Evaluación del sistema **37**

5. Conclusiones **44**

5.1. Trabajo futuro 45

Introducción

Game streaming se refiere a la modalidad de jugar a un videojuego que está siendo ejecutado en un servidor remoto. Esto puede tener ciertas ventajas frente a correr el juego de forma local de la forma convencional, en contraste llamado *juego nativo*. [1]

Primero, permite acceder a videojuegos con una gran fidelidad gráfica en dispositivos con una limitada potencia de procesamiento, como pueden ser consolas portátiles, celulares o *Smart TVs*. Además, en casos de dispositivos móviles energizados por batería, puede ser un método más ahorrativo comparado con ejecutar videojuegos de forma nativa. Permite ahorrar espacio de almacenamiento, y tiempos de instalación y cargas durante el juego. Por otro lado, es importante recordar también que los precios de las PCs de alta gama también pueden ser un factor prohibitivo para muchos posibles consumidores: utilizando *game streaming*, estos costos de *hardware* podrían ser absorbido por los proveedores de los servicios de juego.

Sin embargo, implementar un sistema de *game streaming* presenta múltiples desafíos. En primer lugar, está la necesidad de una conexión de internet sólida y con un buen ancho de banda de bajada. En la opinión del autor, se estima que para la transmisión a una calidad visual aceptable de un videojuego moderno a 60 cuadros por segundo y a una resolución de 1920 x 1080 pixeles, se necesita una velocidad de transferencia de al menos 10MB por segundo.

Además de esto, es muy importante que la latencia total del sistema, popularmente conocida como *lag*, sea mínima. De no ser así, el juego interactivo se vuelve poco agradable o, en el peor de los casos, se vuelve totalmente imposible. La latencia total es resultado del *delay* agregado a partir de la suma de la duración de los varios pasos a lo largo del proceso de captura, codificación, transmisión, recepción, decodificación y reproducción de los cuadros de vídeo del juego, así como el *delay* inherente de los dispositivos de entrada, como el ratón, el teclado o el mando, y de salida como un televisor o el monitor.

Es por eso que resulta crítico que cualquier sistema que quiera brindar un servicio equiparable al juego nativo deba minimizar tanto como sea posible por el *hardware* la latencia acumulada de los procesos, así como la latencia de transmisión sobre la red, para limitar el tiempo de respuesta a un rango aceptable. El objetivo del presente trabajo consiste en demostrar las ganancias que se pueden obtener al integrar de forma óptima los distintos componentes del sistema, limitando la presencia de elementos que suelen introducir demoras de procesamiento, como *buffers* innecesarios y transferencia de memoria de vídeo a memoria del sistema.

Es difícil y subjetivo estimar un tiempo de respuesta aceptable para los usuarios. Para el interés de este proyecto, se considera una latencia total del sistema aceptable de 50ms.

1.1. Productos comerciales

Cloud gaming, o el juego remoto, es un interés de larga data en la industria de los videojuegos. Los primeros servicios consistieron en catálogos de juegos on-demand, monetizado mediante subscripciones mensuales. En el 2010, el servicio On-Live se lanzó en el mercado estadounidense bajo este modelo. Gaikai lo siguió en el 2011. Los usuarios se vieron atraídos a estos servicios por la posibilidad de acceder a un gran

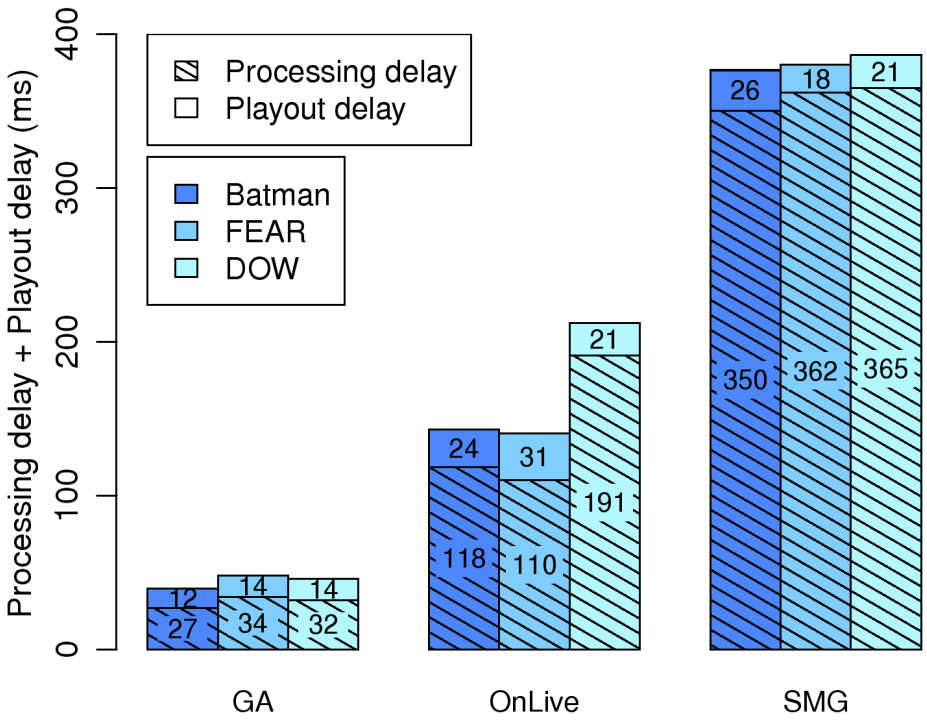

Figura 1.1: *Delay* de procesamiento para los juegos *LEGO Batman: The videogame*, *F.E.A.R 2: Project Origin* y *Warhammer 40,000: Dawn of War II*, usando GamingAnywhere, OnLive y StreamMyGame. [12]

catalogo juegos gráficamente intensivos sin tener que poseer una costosa PC o una consola.

Sin embargo, rápidamente fueron evidentes las fuertes limitaciones de la tecnología de la época, y las críticas y el escepticismo no tardaron en aparecer [2]. La latencia de procesamiento, medida como el tiempo entre el momento en que un usuario interactúa con el videojuego y el momento en que ve su efecto en pantalla, y descontando la latencia propia de la red, era superior a los 150ms, un lapso inaceptable para muchos jugadores de géneros como First Person Shooters, por ejemplo. Además, el rendimiento de los propios juegos estaba seriamente limitado por la codificación de los *streams* de video, con lo que los cuadros por segundo (FPS) se veían fuertemente afectados. Los *hardware encoders* de H264 no estarían presentes en las tarjetas

gráficas *mainstream* hasta finales del 2011, tanto en productos NVIDIA como AMD.

On-Live cerro sus puertas en 2015, mientras que Gaikai fue adquirida por Sony en el 2012 [3, 4]. Desde entonces, las incursiones de nuevas compañías en el sector han sido más bien modestas. Sony lanzo su servicio Playstation Now [5] en el 2014 con un catálogo compuesto por juegos *retro* de Playstation 3, mientras que su consola Playstation 4 incluye funcionalidad que permite el juego remoto desde otros dispositivos, incluyendo dispositivos Android, iOS y Windows.

GeForce NOW [6], de NVIDIA, lleva años en estado beta, mientras que la compañía permite a los usuarios de sus tarjetas GeForce jugar de forma remota desde sus PCs hacia sus dispositivos NVIDIA Shield con una funcionalidad conocida como Gamestream. También existen clientes no oficiales para otros dispositivos y sistemas operativos, producidos bajo el proyecto de *software* libre Moonlight.

Más notablemente, a lo largo del 2019 se han hecho anuncios de peso que anticipan un renovado interés en *Cloud Gaming* por parte de la industria tecnológica. En abril, Google anunció el proyecto Stadia[7], mientras que Microsoft hizo lo propio con el proyecto xCloud[8]. Ambos están actualmente en beta cerrada, y planean ser lanzados en los territorios de norteamerica durante el 2020.

Las circunstancias de estos gigantes tecnológicos son muy diferentes a las que tuvieron aquellos servicios pioneros lanzados al principio de la década. En primer lugar, ambos cuentan con servicios de infraestructura en la nube de los que se pueden apalancar para proporcionar un servicio de calidad superior (Azure y Google Cloud). Además, cuentan con los fondos necesarios para afrontar los costos de cientos de miles de GPUs y CPUs capaces de correr juegos actuales con el nivel de calidad que un usuario puede esperar de un juego de PC moderno. Por otro lado, son favorecidos por el incremento significativo de las velocidades de descarga hogareñas a nivel mundial,

	Q1 2010 (Mbps)	Q1 2017 (Mbps)	Variación (%)
Alemania	12.948	65.6	506.64
Argentina	6.196	40.3	650.42
Brasil	5.043	46.5	922.07
China	2.732	45.9	1680.09
Egipto	4.472	17.0	380.14
España	9.340	84.8	907.92
Estados Unidos	16.207	86.5	533.72
Francia	11.118	49.7	447.02
Hong Kong	29.570	129.5	437.94
Israel	10.922	99.1	912.84
Japón	25.790	94.5	366.42

Cuadro 1.1: Crecimiento de las velocidades más altas por proveedor en promedio, según Akamai[10, 11]

y la proliferación de *access points* WiFi de 5.8 GHz, lo que permite una conexión inalámbrica más estable y fiable. Por lo menos en el caso de Stadia, plantean trabajar con los desarrolladores para lograr una fuerte integración que maximice la utilización de una plataforma de *hardware* fija y fuertemente controlada, por ejemplo, mediante técnicas de predicción de entradas del usuario y *lag compensation*[9].

1.2. Objetivos

El objetivo de este trabajo es desarrollar las bases de un producto de software libre para el juego remoto que supere en rendimiento y latencia a las alternativas existentes en el mercado. Para ello, se plantea optimizar los procesos de captura y codificación de los cuadros producidos por el juego, aprovechando las posibilidades del *hardware* presente en las tarjetas de vídeo actuales a su máxima capacidad y realizando distintas optimizaciones de rendimiento, como minimizar la presencia de *buffers* intermedios y

los procesos de copia a memoria del sistema.

En primer lugar, es pertinente diferenciar los distintos productos que existen en el mercado bajo el término genérico *"game streaming"*. Por un lado, existen aquellos servicios monetizados mediante una suscripción que ofrecen uso ilimitado de un catálogo de juegos disponibles para ser retransmitidos instantáneamente desde un servidor remoto. En este caso, los videojuegos son ejecutados en los servidores de la compañía, y el usuario tiene acceso a un cliente, frecuentemente multiplataforma, mediante el cual puede disfrutar del título usando una conexión a internet. Ejemplos de esto incluyen servicios del estilo de Playstation Now, GeForce NOW, y los futuros Microsoft Project xCloud y Google Stadia.

Por otro lado, otros productos permiten que el usuario realice una transmisión en la red local de un videojuego que es ejecutado en su PC. Se espera que el usuario instale en sus dispositivos tanto el programa servidor como el cliente. A menudo se ofrecen clientes multiplataformas, incluyendo dispositivos móviles, consolas de juego y PCs, mientras que el software del servidor está limitado a una única clase de dispositivos; generalmente se trata de PCs con ciertas características o consolas de sobremesa. Ejemplos de este tipo de producto son Sony Remote Play, Steam Remote Play, NVIDIA Gamestream y Xbox Game Streaming. Este proyecto, SUNSHINE, es similar a este tipo de productos.

Existe una carencia de alternativas de software libre para este tipo de aplicaciones de *game streaming*. En efecto, los productos disponibles comercialmente son producidos por compañías de peso en la industria, y su compatibilidad está limitada o por las plataformas o dispositivos que estas compañías manejan, como es el caso de Sony y NVIDIA, o por los videojuegos vendidos a través de sus plataformas, como es el caso de Steam.

Gaming Anywhere[12] es el único *software* libre de este tipo que se pudo encontrar que ofrece un sistema completo. Sin embargo, lleva sin actualizarse desde el 2016, y no cuenta con la capacidad de utilizar el *hardware* moderno para hacer *encoding* de vídeo, por lo que el impacto que la captura tiene sobre el rendimiento del juego es significativo. Por otro lado, tiene el beneficio de estar disponible en múltiples plataformas, incluyendo Windows, Linux, Mac, iOS y Android.

Otro proyecto de *software* libre que existe es el anteriormente mencionado Moonlight[13] (de donde el nombre de este proyecto proviene originalmente), un cliente multiplataforma del producto Gamestream de NVIDIA. En este caso, tanto el *encoding* del vídeo como la decodificación suceden en *hardware*, por lo que los cuadros por segundos del juego no se ven reducidos y la latencia de procesamiento es baja. Por desgracia, el *software* del servidor es propietario y parte de los drivers de NVIDIA, y su funcionamiento exige que la PC donde se ejecuta el juego cuente con una tarjeta GeForce.

Además de proveer una herramienta alternativa de código abierto y de alto rendimiento para *game streaming*, se encontró pertinente para este trabajo explorar una funcionalidad interesante que no es provista por ningún *software* comercial hasta el momento: capturar vídeo directamente desde el *back buffer* de la aplicación, y no desde el compositor de ventanas de Windows. Si bien puede ser que el proceso de captura sea más complejo como resultado, hacer esto acompañado por un tratamiento adecuado de las entradas del usuario remoto permitiría que la ejecución del juego pueda ser aislada del comportamiento de la ventana de la aplicación, al menos en cuanto a su visibilidad se refiere. En otras palabras, un usuario puede estar utilizando la PC donde el juego esta siendo ejecutado sin afectarlo ni siendo afectado por su presencia. Tanto la introducción de entradas del ratón y el teclado recibidas por el juego como la visibilidad de la ventana son totalmente independientes de cualquiera

	Servidor				Cliente					
	Windows	macOS	Linux	PS4	Windows	macOS	Linux	iOS	Android	Shield
Steam Remote Play	X									
NVIDIA Gamestream	X*				X**	X**	X**	X**	X**	X
Playstation Remote Play				X	X			X	X	
GamingAnywhere	X	X	X		X	X	X		X	
Parsec	X				X	X	X		X	

Cuadro 1.2: Tabla de compatibilidad de sistemas de *game streaming*. *Utilizando tarjetas gráficas NVIDIA. **Utilizando clientes de Moonlight

sea la ventana activa actual de la PC.

Se considera que esta funcionalidad haría de SUNSHINE una alternativa valiosa en cuanto a la funcionalidad que proporciona, incluso dejando de lado las ganancias de rendimiento obtenidas, porque permite que la operación remota no requiera del uso exclusivo de la PC dónde el juego esta siendo ejecutado, algo que ningún otro producto, comercial o no, permite por el momento.

Analisis

2.1. Requerimientos

El sistema operativo elegido para el desarrollo es Windows, debido a que concentra la gran mayoría de la base de usuarios de juegos de PC. Esto es tanto para la PC que transmite el juego (que llamaremos el SERVIDOR), como la que interactúa el usuario (que llamaremos el CLIENTE). Tanto porque el funcionamiento interno del sistema de captura e ingreso de entradas de teclado y ratón está íntimamente acoplado a Win32, como por el hecho de que el mercado de videojuegos de PC afuera de Windows es, por el momento, ínfimo, no se considera factible ni necesario portar el *software* del SERVIDOR a otra plataforma. En contraste, un CLIENTE multiplataforma si resultaría muy beneficioso, porque permitiría expandir la disponibilidad de los videojuegos a sistemas para los cuales no han sido diseñados. Debido a la dificultad de elaborar clientes para diversas plataformas, se decidió dejar esto como trabajo futuro y enfocarse en Windows por el momento.

Se decidió soportar las dos familias de tarjetas gráficas más populares en el mercado: AMD Radeon y NVIDIA Geforce. Ambas proveen funcionalidades similares que permiten la codificación de video por *hardware*, cada una proveyendo su propia librería de usuario destinada para este fin: AMF[14] y NVENC[15], respectivamente. Se decidió ignorar el tercer fabricante relvevante de GPUs para PCs de escritorio, Intel, porque si bien sus GPUs se encuentran en una gran cantidad de dispositivos,

en realidad se trata de chips integrados con una baja capacidad de procesamiento gráfico, por lo que en la práctica no resultan idóneos para juegos gráficamente intensivos. Esto significa que su relevancia en el mercado en cuanto a este caso de uso en particular es más bien baja.

Como la captura de los *buffers* de vídeo se realiza directamente desde dentro de la aplicación misma, es necesario que el sistema soporte trabajar con texturas de la API gráfica del juego que se busca transmitir. Se eligió soportar los juegos que utilicen DirectX 11 para el presente trabajo, debido a su gran popularidad entre los juegos actuales, así como la compatibilidad con los *encoders* por hardware de AMD y NVIDIA. Sin embargo, esta preferencia puede resultar un tanto anticuada, considerando que varios de los juegos mas populares lanzados en el 2019 utilizan DirectX 12 o Vulkan. Es una prioridad como trabajo futuro incluir soporte para éstas nuevas APIs gráficas.

El SERVIDOR debe realizar la captura del vídeo de un juego, debe codificarlo en formato H264 o HEVC y debe transmitirlo por red al CLIENTE, que debe decodificar el vídeo y reproducirlo en pantalla. Además, el SERVIDOR debe inyectar las entradas del ratón y el teclado que recibe por la red de parte del CLIENTE, de forma que se pueda controlar el videojuego utilizando las entradas producidas por el usuario remoto, y de forma tal que un usuario local del SERVIDOR no pueda introducir entradas en el juego.

En resumen, los requerimientos son:

- Soporte del sistema operativo Windows tanto para el SERVIDOR como el CLIENTE.
- Captura de la imagen de la ventana de la aplicación

- Soporte de codificación de video por *hardware* utilizando GPUs AMD y NVIDIA.
- Soporte de juegos que utilicen DirectX 11.
- Transmisión del video capturado desde el SERVIDOR al CLIENTE por red.
- Transmisión de las entradas de ratón y teclado del CLIENTE al SERVIDOR por red.
- Latencia de procesamiento del sistema menor a 50 ms.

Los siguientes puntos fueron considerados para su incorporación en un primer momento, pero se resolvió abandonarlos para esta entrega por cuestiones de tiempo y alcance. Aún así, se consideran puntos relevantes para el desarrollo de SUNSHINE como un producto útil, y por lo tanto se dejan como trabajo futuro:

- Captura, transmisión y reproducción de audio del juego.
- Soporte de mando.
- Soporte de juegos que utilicen DirectX 9, 10, 12, OpenGL o Vulkan.

2.2. Arquitectura

El proyecto está distribuido en dos partes. La primera, el SERVIDOR, consiste en un ejecutable y un DLL cuya función es iniciar el proceso del juego, capturar los cuadros de vídeo, codificarlos y transmitirlos por la red. Además, debe reproducir las entradas del ratón y del teclado que recibe desde la red para permitir la interacción con el juego.

La segunda parte, el CLIENTE, consiste en un ejecutable que recibe los paquetes de vídeo desde la red, los decodifica y los reproduce en pantalla, mientras que también registra la entrada del usuario y la envía por la red.

Todo el desarrollo se hizo en C++ en el sistema operativo Windows 10. Principalmente, se intentó utilizar únicamente la funcionalidad provista en la API nativa del sistema, Win32, y no depender de librerías externas cuya funcionalidad se pueda reemplazar por código propio, tanto por motivos didácticos como para maximizar el control sobre el proceso total, y poder optimizar el rendimiento tanto como fuera posible. A continuación, se listan las librerías de terceros utilizadas en SUNSHINE:

- Easylogging++, librería de logging.
- AMF y NVENC, SDKs para el encoding por hardware de AMD y NVIDIA, respectivamente.
- EasyHook, herramienta de inyección de código remoto y *hooking*.
- `libavcodec` del proyecto `ffmpeg`, decodificación de *streams* H264 y HEVC
- GLFW, sistema de ventanas para OpenGL.
- Glad, loader de OpenGL 4.

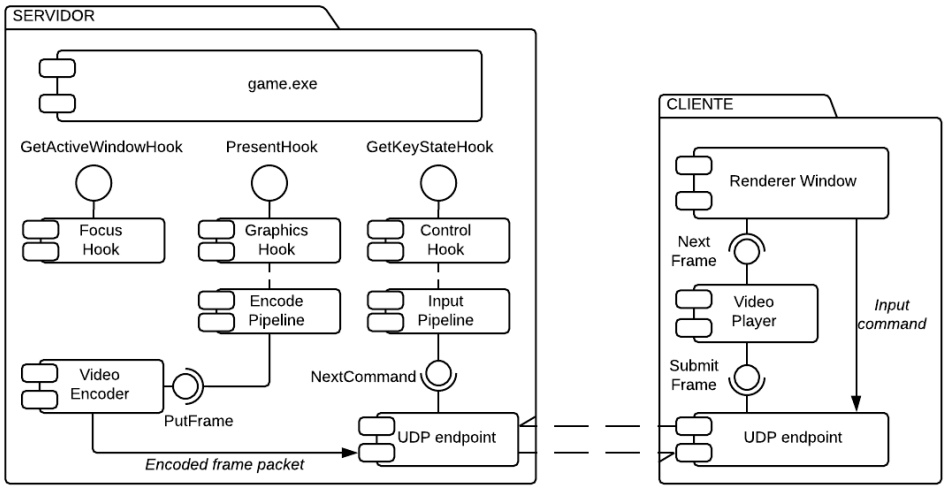

Figura 2.1: Diagrama de componentes del proyecto

Detalles técnicos

3.1. *DLL injection*

En Windows, existen tanto bibliotecas estáticas como dinámicas, estas últimas llamadas DLLs[16] o *Dynamic Link Libraries*. Los archivos DLL son un componente fundamental de Windows. Tan es así, que los servicios de la API Win32 están distribuidos a través de múltiples archivos `.dll` disponibles para las aplicaciones, tales como `kernel32.dll`, `user32.dll` y `gdi32.dll`.

Una propiedad interesante de los DLLs es que pueden ser cargados dinámicamente, mediante una llamada a la API `LoadLibrary`[18]. Esto se conoce como vinculación dinámica en tiempo de ejecución, y tiene la ventaja que la aplicación no necesita haber sido *linkeada* con el archivo `.lib` correspondiente al `.dll` cuando fue creado el ejecutable; en otras palabras, `LoadLibrary` es una forma de ejecutar código de un DLL arbitrario desde una aplicación que no conoce necesariamente los detalles de esa librería.

A continuación, se explica el proceso que realizado para explotar la naturaleza de los archivos `.dll` en Windows, así como ciertas APIs presentes en Win32 para inyectar código de forma remota en un proceso ajeno[17].

- Encontrar el identificador del proceso que buscamos inyectar mediante `EnumProcess`, buscando con el nombre del archivo ejecutable.

- Obtener un `HANDLE` del proceso víctima mediante `OpenProcess` con el *flag* `PROCESS_ALL_ACCESS`.
- Asignar suficiente memoria en el proceso víctima para la cadena del *path* completo al DLL que contenga el código que queremos inyectar, utilizando `VirtualAllocEx`.
- Escribir el *path* completo al DLL en la memoria que acabamos de asignar, utilizando `WriteProcessMemory`.
- Obtener una referencia a la llamada del sistema `LoadLibrary` utilizando la función `GetProcAddress`.
- Crear un hilo de ejecución remoto que llame a la funcion `LoadLibrary`, mediante `CreateRemoteThread`.

En el archivo DLL a asignar se debe incluir el código arbitrario que se desee ejecutar en la función `DllMain` que sirve como punto de entrada para el DLL. En este caso, el código arbitrario consiste en la instalación de *hooks* en los subsistemas gráficos y en las colas de mensajes de las ventanas, así como también la inicialización de *threads* para la transmisión y recepción de paquetes por la red. Para obtener una referencia a una función de Win32 que se desea *hookear*, basta con utilizar una llamada a `GetProcAddress`, pasando como parámetro el *handle* del módulo del que pertenece la función.

Existe documentación de esta técnica al menos desde el 2003, y ha sido utilizada con éxito en software legítimo en productos como *OBS Studio* y *Reshade*, por nombrar algunos ejemplos de código abierto. Este tipo de técnicas es particularmente frecuente en *sofware* que interactúa con juegos, ya sea de forma útil como puede ser el caso de clientes de *chat*, herramientas de captura de imágenes y sistemas de notificaciones, como de forma maliciosa en el caso de *cheats* en juegos multijugador. Como ciertos programas de *anti-cheat* pueden detectar la inyección de código, por ejemplo, listando

los módulos cargados en el proceso mediante `EnumProcessModules`, no se recomienda el uso de SUNSHINE en juegos multijugador competitivos, a menos que el usuario no este preocupado por las sanciones que puede llegar a sufrir.

3.2. *JMP hooking*

Un *hook* es simplemente una función que tiene como propósito ser invocada en lugar de alguna otra función, y de esta forma secuestrar el control del hilo de ejecución y proporcionar algún comportamiento que no fue previsto por el programa original.

En este caso, fue utilizado *JMP hooking*[19], una técnica que consiste en interceptar las llamadas que hace la aplicación a distintas funciones dentro de DLLs conocidos, tales como `user32.dll` y `d3d11.dll`, y redireccionarlos hacia funciones propias que provean funcionalidad adicional. El método consiste simplemente en reemplazar la llamada de la primera instrucción de la función a interceptar con una instrucción JMP incondicional hacia un *hook*, y luego de su ejecución, redirigir a la instrucción de retorno de la funcion original.

Este es un método estable y confiable, popularizado por la propia Microsoft con su librería Detours[20] en el año 2002, hoy de código abierto. Se decidió por utilizar una librería de terceros para implementar esta funcionalidad; si bien el método no es demasiado complejo, presenta ciertas dificultades y casos bordes a la hora de tratar con código de 64 bits.

Utilizando en conjunción las técnicas de *DLL injection* y *JMP hooking*, se puede inyectar código de forma remota en un juego, instalar *hooks* en puntos claves de su ejecución que permitan proveer de funcionalidad adicional que no fue planificada por los desarrolladores originales, y correr servidores en *threads* dentro del proceso del juego, ocupando el mismo espacio de memoria. De esta forma, se tienen todas las herramientas necesarias para realizar la captura de los cuadros del juego, así como el *encoding* del vídeo, la transmisión y la inyección de los mensajes de entrada del ratón

y el teclado.

3.3. Entrada del ratón y el teclado

Uno de los puntos claves del sistema consiste en la reproducción de los comandos de entrada del ratón y el teclado que son recibidos de parte del cliente remoto. El importante para el correcto funcionamiento del sistema que la entrada recibida desde el cliente no sea simplemente insertada dentro de las colas de entrada del sistema operativo, utilizando alguna de las APIs comunes de Win32 como `SendInput`: se quiere que el cliente remoto no pueda interactuar con ninguna otra aplicación que esté siendo ejecutada en el servidor aparte del juego, permitiendo así que otro usuario del servidor pueda seguir utilizando la PC de forma local sin verse afectado por la transmisión remota. En cambio, los mensajes son inyectados directamente a la aplicación mediante una combinación de mensajes sintetizados y *hooks* en puntos clave de la API Win32.

Para empezar, cabe enumerar las distintas formas que una aplicación puede ser notificada acerca de la entrada de ratón y del teclado[21, 22]. Una de las complicaciones más grandes que se encontraron en este trabajo estuvo en que esta información puede ser obtenida de diversas formas por parte de las aplicaciones, varias de las cuales son utilizadas en la actualidad por distintos videojuegos populares con patrones de uso completamente diferentes, algunos incluso usando simultáneamente métodos de entrada que pueden parecer suplementarios. Es por eso que se debió realizar un extenso trabajo de ingeniería reversa, acompañado de un seguimiento detallado de la documentación de Microsoft, para descifrar tanto las sutilezas que existe en la implementación de varios videojuegos, como los errores comunes presentes en ellos que pueden llevar a que una imitación imperfecta de las API de Entrada de Windows dé como resultado un juego imposibles de usar por la incorrecta detección de los movimientos del ratón, e incluso errores que lleven a la aplicación a abortar.

Un ejemplo de este tipo de disparidad entre las implementaciónes por parte de las aplicaciones y la documentación oficial de Windows está en el incorrecto manejo por parte de los videojuegos actuales de las actualizaciones de la posición del ratón mediante posiciones absolutas. Si bien existe este tipo de entrada por parte de los ratones en la documentación de Windows, su falta de uso en la práctica lleva a que muchos juegos tengan dificultades a la hora de procesar este tipo de entradas, requiriéndose en cambio actualizaciones que usen posiciones relativas. Resultó importante para la realización de este trabajo, por lo tanto, prestar atención al uso real de las APIs de Windows, y no depender únicamente de la documentación oficial.

La forma más simple que tiene que una aplicación de recibir entradas es mediante la cola de mensajes de sus ventanas. En Windows, todas las ventanas reciben notificaciones del sistema mediante mensajes que puede recibir llamando tanto la funcion `GetMessage` (bloqueante) como `PeekMessage` (no bloqueante). Una vez recibido un mensaje, la aplicación puede llamar a `DispatchMessage` para decirle al sistema operativo que realice una llamada a la `Window Procedure`, un *callback* definido por la aplicación para el procesamiento de mensajes.

Para las entradas del ratón y el teclado, los mensajes que recibe la aplicación son `WM_KEYDOWN` y `WM_KEYUP` al presionar y soltar una tecla, `WM_MOUSEMOVE` al realizar un movimiento del cursor y el conjunto de `WM_LBUTTONDOWN`, `WM_LBUTTONUP`, `WM_RBUTTONDOWN`, `WM_RBUTTONUP`, `WM_MBUTTONDOWN`, `WM_MBUTTONUP`, etc, al presionar los botones del ratón.

```
static LRESULT CALLBACK TempWndProc(
    HWND hWnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam
) {
    switch (message) {
        case WM_ACTIVATE:
          if (wParam == WA_ACTIVE || wParam == WA_CLICKACTIVE) {
            LOG(INFO) << "Window activated.";
            break;
          }
          else {
            LOG(INFO) << "Window inactivated";
            return DefWindowProcW(hWnd, message, wParam, lParam);
          }
          break;
    case WM_KILLFOCUS:
      LOG(INFO) << "Kill focus";
      return DefWindowProcW(hWnd, message, wParam, lParam);
    case WM_SETFOCUS:
      LOG(INFO) << "Set focus";
      break;
    case WM_SYSCOMMAND:
      LOG(INFO) << "SYSCOMMAND";
      break;
    }
    TranslateJustDummyValue(message);
    return ORIGINAL_WND_PROC(hWnd, message, wParam, lParam);
}
```

Listing 3.1: *Hook* de la `Window Procedure`

Resulta interesante tanto evitar que la aplicación reciba eventos de entrada provenientes de la misma PC que esta ejecutando el juego, como también que reciba correctamente las entradas provenientes del usuario remoto del sistema. Para lograr esto, se instalaron *hooks* para `GetMessage` y `PeekMessage` para que filtren los mensajes que notifican las entrada del ratón o del teclado. Esto evita que un usuario local pueda afectar el juego en curso. Por otro lado, cada vez que un comando proveniente del cliente remoto es recibido a través de la red, se envía a todas las ventanas de la aplicación un mensaje *custom* del tipo `DV_<nombre del mensaje>`, que es convertido

dentro de los *hooks* de `GetMessage` y `PeekMessage` al mensaje que corresponde al tipo de entrada que haya sido recibida. Estos *"dummy values"* utilizan el rango de mensajes definidos a partir de `WM_APP`, un rango reservado para el uso interno de las aplicaciones. En teoría, podría existir un conflicto si el videojuego mismo utiliza estos mensajes para su uso interno; sin embargo, no se encontró que esto sea el caso en la práctica.

```
#define DV_INPUT       (WM_APP + 200)
#define DV_MOUSEMOVE   (DV_INPUT + 1)
#define DV_KEYUP       (DV_INPUT + 2)
#define DV_KEYDOWN     (DV_INPUT + 3)
#define DV_LBUTTONDOWN (DV_INPUT + 4)
#define DV_LBUTTONUP   (DV_INPUT + 5)
#define DV_RBUTTONDOWN (DV_INPUT + 6)
#define DV_RBUTTONUP   (DV_INPUT + 7)
#define DV_MBUTTONDOWN (DV_INPUT + 8)
#define DV_MBUTTONUP   (DV_INPUT + 9)
#define DV_XBUTTONDOWN (DV_INPUT + 10)
#define DV_XBUTTONUP   (DV_INPUT + 11)
#define DV_MOUSEWHEEL  (DV_INPUT + 12)
```

Listing 3.2: Dummy values

Utilizando este método, se logró que la aplicación procese las entradas recibidas por parte del cliente remoto como si fueran mensajes de Windows legítimos.

Sin embargo, esto no es suficiente para que la mayoría de las aplicaciones procesen los mensajes que son enviados. Los juegos suelen escuchar a las notificaciones acerca del estado de la ventana mediante mensajes[24], y tomar acciones como poner el juego en pausa. Para que el uso de otras aplicaciones del sistema no afecte el juego remoto, se necesita que estos mensajes sean filtrados de la cola de mensajes de la ventana del juego.

Las ventanas reciben un mensaje `WM_ACTIVATE` con el parámetro `WM_INACTIVE` cuando dejan de ser la ventana activa, y el mensaje `WM_KILLFOCUS` cuando dejan de tener el foco del teclado. Estos mensajes también son filtrados en los *hooks* de `GetMessage` y `PeekMessage`, para permitir que el estado de la ventana no afecte el uso remoto. De esta forma, se consigue "engañar" al juego para que piense que su ventana es siempre la que tiene foco, y no deje de procesar las entradas que van siendo enviadas.

Por completitud, también es necesario instalar *hooks* para las funciones de Win32 que permiten consultar directamente el estado del sistema en un momento dado. Es importante que las respuestas de estas funciones coincidan con la información provista por los mensajes, porque de lo contrario el funcionamiento del juego podría ser incorrecto o, incluso, la aplicación podría abortar. En la práctica, se encontró que los videojuegos rara vez utilizan este tipo de funciones, y en cambio manejan la entrada mediante el sistema de mensajes. Sin embargo, sí son utilizado en ciertos casos para detectar, por ejemplo, si se presionan teclas "modificadoras" como `SHIFT`, `ALT` o `CTRL`.

Las funciones *hookeadas* son:

- `GetFocus` y `SetFocus` permiten consultar y asignar la ventana con el foco del teclado.
- `GetActiveWindow`, `SetActiveWindow`, `GetForegroundWindow` y `SetForegroundWindow` permiten consultar y asignar la ventana activa.
- `SetCursorPos`, `GetCursorPos`, `SetCursorPhysicalPos` y `GetCursorPhysicalPos` permiten consultar y asignar la posición del cursor del ratón.
- `ClipCursor` limita el movimiento del cursor del ratón dentro de una región determinada.

- `GetKeyState`, `GetAsyncKeyState` y `GetInputState` permiten consultar el estado del teclado.

3.4. *Raw input*

Las aplicaciones de Windows tienen un método alternativo para obtener entradas del usuario directamente desde los dispositivos de entrada conectados, incluyendo la entrada del ratón y del teclado, conocido como *raw input*[23]. La ventaja que tiene este método frente al descripto anteriormente es que la entrada no tiene ningún procesamiento por parte del sistema operativo, sino que viene directamente del controlador del dispositivo de entrada. Esto quiere decir, por ejemplo, que el movimiento del cursor no es sometido a la aceleración típica de Windows.

Esto presenta un desafío para la reproducción de los comandos de entrada recibidos de forma remota, dado que significa que se deben reproducir también los efectos de este método de entrada para el correcto funcionamiento de los juegos que dependan de ello. En la práctica, se encontró que la mayoría de los juegos contemporáneos utilizan *raw input* para procesar la entrada, ya sea parcial o totalmente.

Para recibir *raw input* de un dispositivo, la aplicación debe primero registrar las clases de dispositivos en los que está interesada mediante `RegisterRawInputDevices`. Para obtener una lista de los dispositivos conectadosse puede llamar a `GetRawInputDeviceList`, y a `GetRawInputDeviceInfo` para obtener información más detallada.

Para leer las entradas provenientes de estos dispositivos, la ventana primero debe recibir un mensaje `WM_INPUT` que notifique que un dispositivo de entrada tiene información que necesita ser leída. Este mensaje está acompañado por parámetro que especifica un handle que hace referencia al dispositivo de entrada en cuestión.

Existen dos métodos para leer la nueva entrada del dispositivo: mediante `GetRawInputData`, o mediante `GetRawInputBuffer`. La diferencia está en que el segundo permite obtener varias entradas de forma simultánea con una única llamada a la API, mientras que el primero permite obtener información de un único dispositivo a la vez. En la práctica, no se encontró ningún juego que utilice `GetRawInputBuffer`, por lo que este método fue temporalmente ignorado.

`GetRawInputData` es una función que espera un `HRAWINPUT`, un *handle* que puede ser obtenido únicamente del mensaje `WM_INPUT`. Este *handle* sirve de referencia opaca para una estructura `RAWINPUT` mantenida por el sistema, que representa un comando de entrada proveniente de un teclado, de un ratón o de un mando.

Para simular entradas recibidas por parte del usuario remoto, además de los métodos anteriormente mencionados, se envían mensajes del tipo `DV_INPUT`, que son a su vez convertidos a mensajes `WM_INPUT` en los *hooks* de `GetMessage` y `PeekMessage`. Estos mensajes van acompañados por un valor en el lugar del `HRAWINPUT` que indica el índice en un *buffer* circular de los comandos de entrada recibidos de parte del cliente remoto.

```
static void FillMouseRawInputHeader(
    RAWINPUTHEADER& header,
    size_t rawInputSize,
    LPARAM handle
){
  header.dwType = RIM_TYPEMOUSE;
  header.wParam = RIM_INPUT;
  header.hDevice = (HANDLE)handle;
  header.dwSize = rawInputSize;
}

inline static void FillKeyRawInput(
    RAWINPUT& rawInput,
    InputCommand command,
    LPARAM handle
){
```

```
  FillKeyRawInputHeader(rawInput.header, sizeof(RAWINPUT), handle);
  rawInput.data.keyboard.VKey = command.val1;
  rawInput.data.keyboard.MakeCode = command.val2;
  rawInput.data.keyboard.Reserved = 0;
  if (command.event1 == (int16_t)ButtonEventType::BUTTON_EVENT_DOWN) {
    rawInput.data.keyboard.Flags = RI_KEY_MAKE;
    rawInput.data.keyboard.Message = WM_KEYDOWN;
  }
  else {
    //ButtonEventType::BUTTON_EVENT_UP
    rawInput.data.keyboard.Flags = RI_KEY_BREAK;
    rawInput.data.keyboard.Message = WM_KEYUP;
  }
}

static UINT WINAPI HookGetRawInputData(
  HRAWINPUT hRawInput,
  UINT      uiCommand,
  LPVOID    pData,
  PUINT     pcbSize,
  UINT      cbSizeHeader
){
  LPVOID buff;
  if (uiCommand == RID_INPUT) {
    RAWINPUT rawInput;
    ZeroMemory(&rawInput, sizeof(rawInput));
    LPARAM handle = (LPARAM) hRawInput;
    if (pData != nullptr) {
      if (handle >= 128) {
        auto controller = MouseShadowController::GetInstance();
        RAWMOUSE rawMouse = controller->GetRawMouse(hRawInput);
        FillMouseRawInput(rawInput, rawMouse, handle);
      }
      else {
        auto controller = KeyboardShadowController::GetInstance();
        InputCommand command = controller->GetInputCommand(hRawInput);
        FillKeyRawInput(rawInput, command, handle);
      }
    }
    buff = &rawInput;
    *pcbSize = sizeof(RAWINPUT);
  }
    ...
```

Listing 3.3: *Hook* de `GetRawInputData`

La API `GetRawInputData` es *hookeada*, de forma que al recibir un `WM_INPUT`, la aplicación hará una llamada a `GetRawInputData` con un `HRAWINPUT` que servirá para popular una estructura `RAWINPUT` a utilizando la información proveniente del comando de entrada recibido. Al mantener los comandos de entrada en un *buffer* circular de una dimensión considerable, podemos garantizar que reiteradas llamadas a `GetRawInputData` con el mismo `HRAWINPUT` retornara el mismo `RAWINPUT`, lo cual es beneficioso porque consultar un mismo `HRAWINPUT` de forma consecutiva es un patrón de uso que fue encontrado en juegos reales.

3.5. Captura de vídeo

Existen diversas alternativas a la hora de realizar capturas de pantalla en Windows 10. La mayor parte de los programas de *game streaming* disponibles en la actualidad hacen uso de una API presente en Windows desde la versión 8, conocida como `Desktop Duplication API`[25]. Esta API tiene múltiples ventajas:

- Provee la captura de pantalla directamente en una textura de DirectX 11, manteniéndola en memoria de video.
- El resultado de la captura mantiene siempre el mismo formato de pixeles sin importar la aplicación siendo capturada ni la configuración del monitor.
- Tiene un impacto bajo en el rendimiento del juego.
- Es el método oficial de captura soportado por Microsoft.
- Es simple de utilizar.

Sin embargo, también trae aparejado una serie de limitaciones en cuanto a su funcionalidad. En primer lugar, la captura se realiza sobre el *frame buffer* de la pantalla por completo, no limitándose a la visibilidad de una aplicación en particular. Algunos programas de captura recortan esta imagen para que solo la aplicación de

interés sea visible por el usuario remoto, pero aun así la ventana capturada debe ser visible en la pantalla de la PC servidor. Esto es perjudicial a nuestras intenciones de permitir que el uso del juego sea posible independientemente del estado de la ventana de la aplicación, incluyendo el caso en el que la ventana capturada se encuentre minimizada u ocluida por otras aplicaciones.

En lugar de utilizar esta API, SUNSHINE realiza la captura de video directamente desde la ventana correspondiente a la aplicación, utilizando *hooks* en las APIs gráficas[26] en lugar de llamar a la `Desktop Duplication API`. Este tipo de *hooks* es usado frecuentemente en *plugins* relacionados con los videojuegos, ya sea para realizar captura de la pantalla, como para dibujar *overlays* que agreguen funcionalidad útil incluyendo aplicaciones de *chat* y contadores de cuadros por segundo.

Un problema que tiene utilizar esta técnica es que es necesario realizar un método de *hooking* específico para las distintas APIs gráficas que puede utilizar el juego. Para simplificar la discusión de este trabajo, se decidió implementar únicamente soporte para DirectX 11, considerando que es la API más utilizada por los juegos de los últimos años. Es importante como trabajo futuro incorporar soporte de las demás APIs relevantes en Windows: DirectX 9, 10 y 12, y OpenGL y Vulkan.

Para realizar la captura, es necesario interceptar las llamadas a `IDXGISwapChain::Present`. Esta API se utiliza para presentar el cuadro final al usuario despues de haber sido *renderizado*, intercambiando el *front buffer* de la aplicación por el *back buffer*. Dentro de este *hook*, se pasa la textura del *back buffer* directamente al *encoder* de vídeo correspondiente según la tarjeta gráfica de la PC. Esto permite evitar la copia del *back buffer* a memoria del sistema, lo que resultaría en una operación costosa y bloqueante, que causaría una degradación en el rendimiento del juego.

...

```
auto hMod = GetModuleHandle(TEXT("d3d11.dll"));
... // error checking
auto pd3d = (CREATE_DEVICE_AND_SWAP_TYPE) GetProcAddress(
    hMod,
    "D3D11CreateDeviceAndSwapChain"
);
...
//  Create temporary window
WNDCLASSEX wc = { sizeof(WNDCLASSEX), CS_CLASSDC, TempWndProc,... };
RegisterClassEx(&wc);
auto hWnd = CreateWindow(...);
    ...
//  Set Swap chain description, just for initialization,
//  does not really affect anything.
DXGI_SWAP_CHAIN_DESC sd;
ZeroMemory(&sd, sizeof(sd));
sd.BufferCount = 1;
sd.BufferDesc.Width = 640;
sd.BufferDesc.Height = 480;
sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
sd.BufferDesc.RefreshRate.Numerator = 60;
sd.BufferDesc.RefreshRate.Denominator = 1;
sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
sd.OutputWindow = hWnd;
sd.SampleDesc.Count = 1;
sd.SampleDesc.Quality = 0;
sd.Windowed = TRUE;
IDXGISwapChain * pSwapChain;
ID3D11Device * device;
ID3D11DeviceContext * context;
D3D_FEATURE_LEVEL levels = D3D_FEATURE_LEVEL_11_0;
D3D_FEATURE_LEVEL  FeatureLevelsSupported;
HRESULT d3d = pd3d(...,&levels, 1, D3D11_SDK_VERSION, &sd,
    &pSwapChain, &device, &FeatureLevelsSupported, &context);
...
DestroyWindow(hWnd);
//  Getting pointer to EndScene.
uintptr_t * pInterface = (uintptr_t*) * (uintptr_t *)pSwapChain;
//  Present: 8 (Source: Google)
presentSwap = (PRESENT_SWAP_FUNC) pInterface[8];
//  Install hook
this->InstallHook("SwapChainPresent", presentSwap, HookPresent);
    ...
```

Listing 3.4: Instalación del *hook* de `DXGISwapChain::Present`

Para poder instalar el *hook*, es necesario obtener una referencia a la dirección de memoria de la función original. A diferencia de los otros *hooks* que instalamos para manejar la entrada, en este caso no es suficiente utilizar `GetProcAddress` porque la función en cuestión es un miembro de la clase `IDXGISwapChain`. En cambio, se puede obtener una referencia a la `VTable` de esa clase instanciando una *Swap Chain* auxiliar. Sabiendo el *offset* de la función de antemano, es posible obtener así una referencia a la dirección de memoria de la función y proseguir con la instalación del *hook*.

Dentro del *hook* del método `Present`, el *back buffer* actual se puede obtener a partir del método `IDXGISwapChain::GetBuffer`, accedido a través del puntero del objeto *Swap Chain* que realiza la llamada. Una vez obtenido una textura con el *back buffer* de la aplicación, se le pasa esta textura al *encoder* de vídeo de la tarjeta gráfica para que realice la codificación de manera bloqueante, y se encola el paquete resultante para que luego sea enviado por la red al cliente desde un hilo de ejecución separado. Aunque la llamada al *encoder* de vídeo sea bloqueante dentro del *thread* de renderizado del juego, la realidad es que esto no incurre en un costo importante que afecte notablemente los cuadros por segundo, y se evita de esta madera tener que copiar el *back buffer* en una textura auxiliar, lo que sería necesario si la codificación de vídeo se realizase en un *thread* aparte.

```
static HRESULT WINAPI HookPresent(
  IDXGISwapChain * This,
  UINT SyncInterval,
  UINT Flags)
{
  //  Perform backbuffer capture and encoding
  auto hook = D11Hook::GetInstance();

  ID3D11Device * device;
  This->GetDevice(__uuidof(device), (void **)&device);
  auto pipeline = hook->GetEncodePipeline(device);

  //  Get backbuffer.
  ID3D11Texture2D * backbuffer;
  This->GetBuffer(0, __uuidof(backbuffer), (void **)&backbuffer);
  if (!pipeline->Call(backbuffer)) {
    LOG(ERROR) << "EncodePipeline failed.";
    //  ExitProcess(1);
  }
  return hook->GetPresent()(This, SyncInterval, Flags);
}
```

Listing 3.5: *Hook* de `DXGISwapChain::Present`

A la hora de elegir el códec de video a utilizar por el sistema, las opciones están fuertemente limitadas por la capacidad del *hardware* disponible en las tarjetas de vídeo comerciales en la actualidad. Si bien existen alternativas de códecs libres de patentes que funcionan con una *performance* competitiva en la actualidad, como pueden ser VP9 o AV1[27], en la práctica la disponibilidad de estos códecs es muy limitado en los *encoders* de *hardware* presentes en las tarjetas gráficas más populares. Esto nos deja como únicas opciones a los códecs H264 y su sucesor más moderno, H265 o HEVC.

La configuración que resulta más interesante para éste caso de uso tiene que ver con el *rate control mode*[28], esto es, el modo en que el códec decide la cantidad de compresión aplicada sobre un cuadro de vídeo. En nuestro caso, la configuración más relevante es el modo CBR, o modo de *bitrate* constante, que nos permite fijar la cantidad de datos por segundo que se obtienen del proceso de codificación. Esto nos

GPU	H.264 (AVCHD) YUV 4:2:0		H.264 (AVCHD) YUV 4:4:4		H.264 (AVCHD) LOSSLESS		H.265 (HEVC) YUV 4:2:0		H.265 (HEVC) YUV 4:4:4		H.265 (HEVC) LOSSLESS	
	MAX Color	MAX Res.	MAX Color	MAX Res.	MAX Color	MAX Res.	MAX Color	MAX Res.	MAX Color	MAX Res.	MAX Color	MAX Res.
Kepler	8-bit	4096 x 4096	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Maxwell (1st Gen)*	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	N/A	N/A	N/A	N/A	N/A	N/A
Maxwell (2nd Gen)	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	N/A	N/A	N/A	N/A
Maxwell (GM206)	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096
Pascal	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	10-bit	8192 x 8192**	10-bit	8192 x 8192**	10-bit	8192 x 8192**
Volta	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	10-bit	8192 x 8192	10-bit	8192 x 8192	10-bit	8192 x 8192
Turing	8-bit	4096 x 4096	8-bit	4096 x 4096	8-bit	4096 x 4096	10-bit	8192 x 8192	10-bit	8192 x 8192	10-bit	8192 x 8192

Figura 3.1: Códecs disponibles en tarjetas gráficas NVIDIA

permite evitar la congestión en la red, sacrificando parte de la calidad visual de la imagen final obtenida como resultado.

Por otra parte, es importante reducir al el *bitrate* tanto como es posible para minimizar la congestión en la red y no incurrir en pérdida de paquetes de vídeo, lo que implicaría necesidad de reenvío y por lo tanto significaría introducir latencia al sistema.

Existen distintos tipos de cuadros que son producidas durante la codificación de un *stream* de vídeo H264 o HEVC. En primer lugar están los cuadros I, que son expresados de forma independiente con respecto a los demás cuadros del video, de forma que no se necesita de cuadros de referencia para que puedan ser decodificados correctamente.

Los cuadros P, por otro lado, necesitan un cuadro de referencia anterior para ser decodificados, lo que significa que requieren una cantidad significativamente menor de datos para ser representados. Finalmente, los cuadros B necesitan información tanto de cuadros de referencias anteriores como futuros para que su decodificación sea correcta.

Debido a que los cuadros B necesitan información futura para ser decodificados, solo sirven si se tiene un *buffer* de vídeo que almacene cuadros futuros, y por lo tanto sólo son utilizables si la existencia de latencia no tiene efectos negativos; debido al interés de este sistema de minimizar al máximo la latencia de procesamiento, los cuadros B no son utilizables para este caso de uso. En cambio, como se busca reducir el ancho de banda requerido, se configura el *encoder* para que la estructura del GOP (*group of pictures*) sea de la forma IPP, y su tamaño infinito. Esto significa que solo el primer cuadro es de tipo I, mientras que todos los siguientes son del tipo P. Así, se reduce el *bitrate* substancialmente sin incurrir en latencia adicional, pero viene aparejado del problema de que un paquete perdido hace irreproducible el *stream* a partir de ese punto, lo que es significativo teniendo en cuenta que se planea utilizar un protocolo de red no confiable en un medio propenso a perdidas como lo son las conexiones WiFi.

3.6. Reproducción de vídeo

Una función clave del CLIENTE es la reproducción de video que fue capturado por el SERVIDOR. Para esto, se hizo uso de la librería `libavcodec` del proyecto `ffmpeg` para decodificar los paquetes de H264 o HEVC. Esta librería incluye una variedad de *encoders* y *decoders* disponibles en distintas plataformas y que utilizan distintos tipos de *hardware*, y es un paquete de *software* estándar en la industria.

Se decidió utilizar un *decoder*[29] de *software* u, opcionalmente, un *decoder* en *hardware* compatible con tarjetas gráficas NVIDIA. Se encontró que la latencia de procesamiento del decodificador de vídeo de *sofware* se ve beneficiada si se configura para que utilice un único hilo de ejecución para la decodificación, siendo los resultados obtenidos con esta configuración comparables a los resultados obtenidos usando decodificación por *hardware*. Sin embargo, si se empieza a incrementar el *bitrate* de la transmisión de vídeo, el *decoder* por *software* empieza a ser un cuello de botella para

la reproducción del vídeo. Por eso es recomendable utilizar el *decoder* por *hardware* si está disponible, o reducir el *bitrate* por debajo de los 10 MB si no lo es; en cualquier caso, la calidad visual es idéntica para el mismo *bitrate*.

Una vez que se decodifica el paquete, se obtiene un cuadro de vídeo cuyo formato de píxeles depende del *decoder* utilizado. En el caso del *decoder* por *software*, el formato de pixeles es YUV420, mientras que el *decoder* de *hardware* de NVIDIA tiene NV12 como formato de pixeles de salida. La información del color es representada de una forma equivalente; la diferencia está en que en el caso de YUV420 las componentes U y V son almacenadas en *buffers* de memoria separados, mientras que en NV12 las componentes están intercaladas en un mismo *buffer*.

En el sistema de color YUV, la componente Y codifica información acerca de luma, es decir, la luminosidad del píxel, mientras que los componentes U y V codifica la crominancia del píxel, es decir, el color. Este tipo de sistema de colores es típico de las transmisiones de vídeo, y es el formato de salida convencional utilizado por los códecs de vídeo.

Para poder mostrar los cuadros de vídeo en pantalla, es necesario convertir el cuadro desde el sistema YUV a RGB, que es el formato de píxeles que espera OpenGL para sus *frame buffers*. Para maximizar el rendimiento de la reproducción de vídeo, esta conversión se realiza mediante *shaders* GLSL dentro del *pipeline* gráfico de la aplicación, y no en tiempo de CPU mediante la lectura y copia del cuadro de vídeo como sale del proceso de decodificación. El proceso consiste en cargar los *buffers* obtenidos en la estructura `AVFrame` obtenida del *encoder* en texturas de OpenGL, y luego suministrarlas como parámetro al *fragment shader* que producirá la imagen final que se ve en el reproductor.

Un *shader*, o sombreador, es un programa especializado que extiende la funcionalidad de las tarjetas gráficas, también conocidas como GPUs (*graphical processing units*). En particular, un *pixel shader*, en terminología de OpenGL conocido como *fragment shader*, se encarga de computar los atributos de color que corresponden a la textura de salida del *pipeline* gráfico. En nuestro caso, podemos utilizar el *shader* para realizar la conversión de color a RGB de una forma que no afecte negativamente el rendimiento.

Las fórmulas para la conversión de formato YUV a RGB son:

$$R = Y + 1,14V$$

$$G = Y - 0,21482U - 0,38059V$$

$$B = Y + 2,1978V$$

3.7. *Networking*

Existen diversos protocolos de capa de aplicación que se utilizan en la actualidad para el *streaming* de contenido de vídeo en tiempo real. Los más utilizados en la actualidad son RTSP (en conjunción con RTP para la transmisión de los contenidos multimedia), RTMP (utilizado por apliaciones web) y WebRTC.

Sin embargo, para este proyecto se decidió no utilizar protocolos existentes, y en cambio utilizar un protocolo binario diseñado específicamente para esta aplicación en particular, montado directamente sobre uno de los protocolos de capa de transporte conocidos. La decisión fue tomada en consideración de las necesidades específicas del proyecto. Se consideró en primer lugar que la implementación completa de un protocolo de reproducción de vídeo sería un proceso largo e innecesario en cuanto a los objetivos de este proyecto. Esto es porque el contenido de vídeo que se transmite no está pensado para ser reproducido por reproductores multimedia disponibles y

preexistentes, sino en un cliente hecho a medida para esta aplicación en particular. Esto permite obviar una gran parte de la funcionalidad presente en estos protocolos, como la capacidad de controlar la reproducción del vídeo, incluyendo las operaciones de pausar, retroceder y acelerar la transmisión, así como las tareas de negociación acerca de las propiedades del contenido y el códec utilizado.

Además, instrumentar un protocolo propio permite tomar decisiones acerca de las propiedades en cuanto a la confiabilidad de la transmisión de distintos paquetes que varíen según el tipo de contenido que es transportado. En este caso particular, tratar el contenido de vídeo como una transmisión multimedia de la misma forma que otros protocolos de reproducción de vídeo es contraproducente: como lo más importante para nuestro sistema es que la latencia total sea mínima, los problemas relacionados con el *timing* de los cuadros de vídeo, conocido como *jittering*, pasan a un segundo plano. En general, las aplicaciones de reproducción de video "en vivo" suelen tener una latencia que va desde unos pocos segundos a unos 100-200 ms. Esto no es aceptable para este caso de uso si se quiere otorgar una experiencia de juego comparable a la nativa.

A la hora de decidir que tipo de *sockets* se utilizarían en este proyecto, se consideraron los dos tipos proporcionados por los sistemas operativos[30]. TCP es un protocolo de capa de transporte que proporciona comunicación confiable basada en conexión. Esta confiabilidad está basada en la retransmisión de paquetes perdidos, cuya recepción o falta de ella es notificada por el receptor al emisor mediante el envío de paquetes de *acknowledgment* (ACK) por parte del receptor del paquete. Además de esto, proporciona funcionalidades de control de congestión, y se ocupa de forma transparente de la segmentación de los datos enviados en paquetes menores al MTU (*minimum transfer unit*). También garantiza que el orden de llegada de los paquetes sea el correcto.

Por otro lado, UDP es un protocolo de red no confiable basado en datagramas. Esto quiere decir que no provee garantías en cuanto a la recepción del paquete, ni tampoco que el orden de llegada de los paquetes sea correcto. Es responsabilidad del usuario dividir los mensajes enviados en paquetes de un tamaño menor al MTU para evitar la fragmentación.

El protocolo de transporte elegido fue UDP. Aunque TCP puede parecer una buena idea en principio, existen varios problemas asociados a su uso en aplicaciones *real time*, como es el caso de este proyecto. En primer lugar, tener que esperar a la llegada de un ACK por parte del cliente antes de comenzar el reenvío introduce latencia en el sistema cuando se incurre en una pérdida de paquetes. Aún peor, TCP conserva en una cola los paquetes más recientes mientras reenvía los paquetes que no han sido confirmados con un ACK, lo que se conoce como head of line blocking, y significa la introducción de aún más latencia en el sistema. Otro aspecto negativo desde un punto de vista práctico es que las opciones de configuración disponibles para TCP son bastante limitadas debido a que su implementación es parte del sistema operativo, y no puede ser modificada por los usuarios con facilidad.

Por otro lado, el uso de UDP presenta distintas problemáticas que necesitan ser abordadas. En primer lugar, UDP necesita que el contenido enviado sea particionado en paquetes menores al MTU, y aun peor, no garantiza que los paquetes lleguen de forma ordenada a destino, algo necesario para la correcta decodificación de los cuadros de vídeo. Esto se puede solventar incluyendo un identificador y un numero de secuencia en el header de los paquetes correspondientes al mismo cuadro, así como un indicador de si se trata del último paquete. Con esta información se puede mantener una tabla de *hash* que permita acumular los datos de los distintos cuadros hasta que estén completos, y recién en ese momento pasarlos al decodificador.

Para la correcta reproducción de un *stream* de vídeo H264 o HEVC es necesario que todos los cuadros sean decodificados en orden y sin saltos, porque la codificación de los cuadros P requiere de información previa. Saltearse un cuadro en la decodificación de vídeo no suele ser fatal para la reproducción del vídeo, pero se prosigue con errores.

En lugar de recurrir al reenvío de paquetes en caso de pérdida, el CLIENTE notifica al SERVIDOR en cuanto cierta cantidad de cuadros han llegado antes de que el próximo cuadro a decodificar. Esto significa que al menos un paquete con información del cuadro actual ha sido perdido. Cuando esto sucede, se le solicita al *encoder* que el próximo cuadro sea del tipo IDR, un tipo similar a I, pero que además impide que los próximos cuadros codificados hagan referencia a cuadros anteriores. Esto en efecto reinicia el stream.

El protocolo es muy simple, y solo tiene 3 tipos de mensajes:

- Paquetes de vídeo, que incluyen un *payload* con parte de un cuadro codificado.
- Paquetes de entrada, que incluyen una lista de comandos de entrada del teclado y del ratón.
- Paquetes de reinicio de codificación, enviados desde el CLIENTE al servidor cuando detecta pérdida.

El cliente envía los últimos 20 comandos de entrada de forma redundante cada vez que el usuario introduce una entrada nueva, es decir, presiona una tecla o mueve el puntero del ratón. Esta redundancia sirve para aumentar la confiabilidad del protocolo sin recurrir a la retransmisión de paquetes, y aprovechar el tamaño máximo del paquete enviado.

Aunque el comando puede nunca llegar al destino, se considera que si se da este caso lo más probable es que, de cualquier forma, la entrada ya sería demasiado tardía

de cualquier forma. Al ser una aplicación en tiempo real, las llegadas tardías de los comandos de entrada podrían tener un efecto indeseado en el juego, por lo que su falta de procesamiento puede ser mejor que su retransmisión a destiempo. Este tipo de efecto perjudicial de la latencia se puede entender mejor si se imagina una situación en un videojuego de plataformas donde el personaje salta a destiempo, o un juego de acción donde la animación del ataque que llega tarde deja al personaje expuesto al enemigo.

Evaluación del sistema

Para evaluar el éxito del proyecto, es necesario realizar una comparación objetiva entre SUNSHINE y los demás productos de *software* disponibles, considerando como métricas importantes tanto la latencia total del sistema como su rendimiento, medido según los cuadros por segundo del juego capturado.

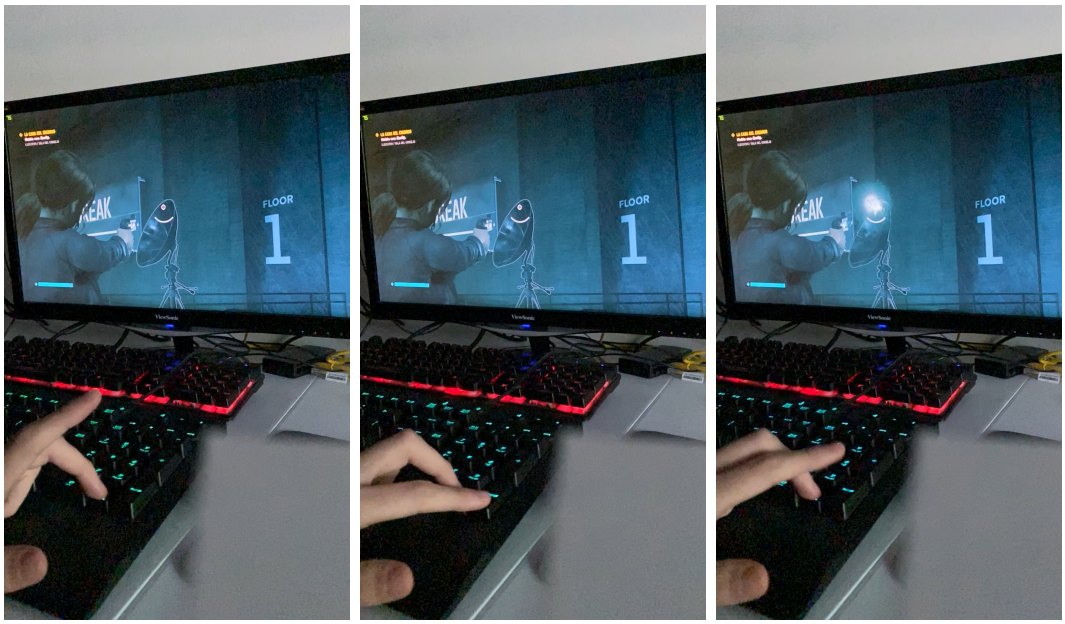

Figura 4.1: Proceso de medición de latencia utilizando el videojuego *Control*

Para medir el tiempo de respuesta total de cada sistema, se dispuso de una cámara fija filmando simultáneamente la pantalla del juego y el dispositivo de entrada que lo controle. El proceso de medición consiste en capturar interacciones repetidas que efectúen un cambio en el juego visible en pantalla, y luego contar la cantidad de

	Servidor	Cliente
CPU	Intel(R) Core(TM) i5-7600 CPU @ 3.50GHz, 3504 Mhz, 4 Core(s), 4 Logical Processor(s)	Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, 2801 Mhz, 4 Core(s), 8 Logical Processor(s)
GPU	NVIDIA GeForce RTX 2060 6GB	NVIDIA GeForce GTX 1050 2GB
RAM	16GB	8GB
Monitor	1920x1080@60Hz	1920x1080@60Hz

Cuadro 4.1: *Hardware* utilizado para las pruebas

cuadros de vídeo entre el accionar en el dispositivo de entrada y su efecto en el juego.

El juego seleccionado para este *test* fue *Control(2019)*, un videojuego de disparos en tercera persona desarrollado por *Remedy*. Se consideró que un disparo en el juego es una buena acción para ser medida, debido a que tiene un efecto inmediato en pantalla. Esto es en contraste a un movimiento del personaje, que en cambio es difícil determinar el momento exacto cuando se procesa la acción debido al *blending* de las animaciones. También se intento que el accionar del botón sea lo más corto posible para minimizar la incerteza en cuanto al momento de activación.

Para cada sistema de *game streaming* se tomaron 20 muestras, de las cuales 10 fueron con la sincronización vertical activada. La sincronización vertical es una configuración que hace que el juego sincronice los cuadros que *renderiza* con la frecuencia de refresco del monitor, de forma de evitar *tearing*, un artefacto visual que surge cuando se intenta dibujar varios cuadros en un mismo ciclo del monitor. Un efecto secundario conocido de la sincronización vertical es la introducción de *input lag*, por lo que se optó por realizar mediciones separadas para medir su efecto.

La captura del vídeo se realizó con una cámara de 240 cuadros por segundo, incluída en un dispositivo iPhone 11, mientras que el monitor capturado tenia una tasa de refresco de 60 Hz. El *frame time* para esa frecuencia es de 4.16ms, con lo cual la latencia medida es el resultado de contar la cantidad de cuadros en el vídeo desde el accionar del botón hasta que aparece el primer destello del disparo en la pantalla,

multiplicado por el *frame time* del vídeo, 4,16 milisegundos.

Todos los experimentos fueron realizados utilizando una conexión WiFi de 5.8 GHz. El *ping* entre las dos máquinas era fue menor a los 1ms. En todos los casos, el códec utilizado fue H264 configurado para codificar con un *bitrate* de 5MBps, y tanto el *encoding* como el *decoding* fue hecho por *hardware* en todos los programas excepto por GamingAnywhere, que no incluye esa funcionalidad.

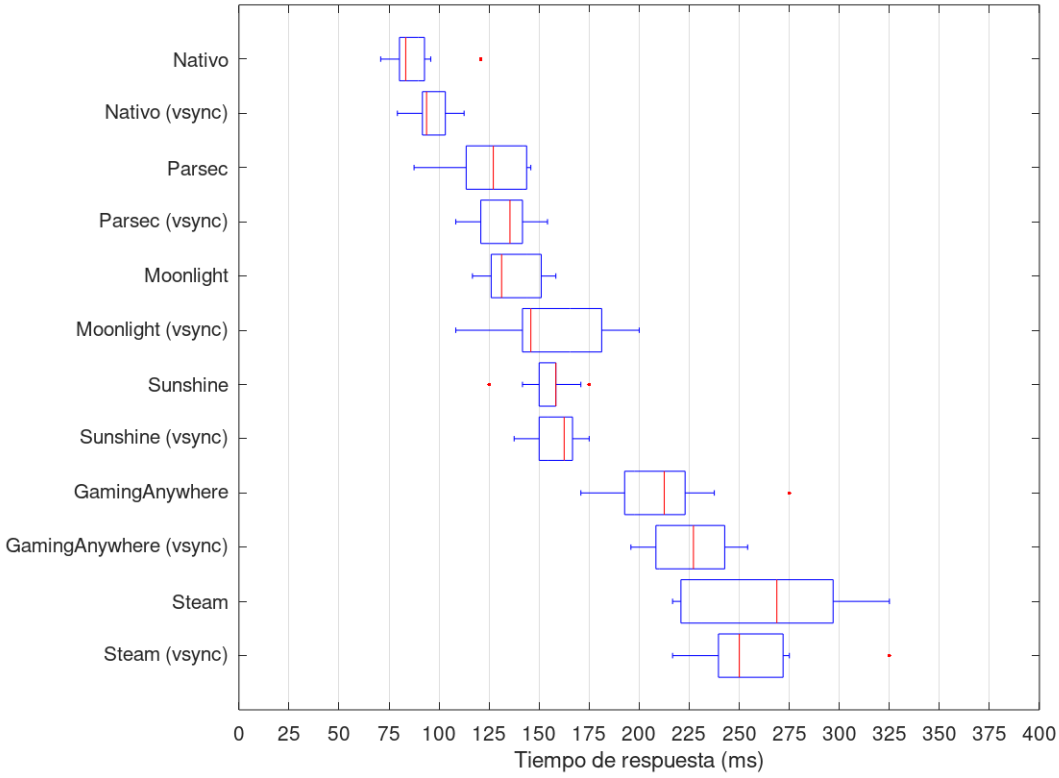

Figura 4.2: Latencia total del sistema medida al efectuar un disparo en *Control*

Además de los programas de *streaming*, se realizaron mediciones del juego siendo jugado en modo Nativo, es decir, de manera local y no remoto. Esto sirve de control para realizar comparaciones contextualizadas de los resultados obtenidos.

Se puede observar que el *software* con menor tiempo de respuesta es Parsec, seguido de cerca por Moonlight y SUNSHINE, y muy lejos de GamingAnywhere y Steam Remote Play. Fue inesperado en *input delay* del juego incluso al ser ejecutado de forma nativa, pues supera los 70 ms, o en otras palabras, aproximadamente 4 cuadros del juego. Es importante considerar esta base al hablar de la latencia agregada por los procesos de captura y transmisión que añade el juego remoto.

Es notorio el caso de Parsec, que sólo cuenta con unos 35.36ms adicionales de latencia en promedio. Esto lo ubica firmemente en el primer puesto entre los productos testeados, mientras que Moonlight y SUNSHINE se disputan el segundo puesto: Moonlight cuenta con menores promedios, mientras que SUNSHINE tiene resultados más consistentes. La latencia adicional que introdujo Moolight en promedio fue 58.24ms, y SUNSHINE, 61.152ms.

GamingAnywhere y Steam, por el contrario, tienen una latencia adicional de 128.96 ms y 173.472 ms, respectivamente. Esto hace que su uso sea considerablemente menos satisfactorio.

Con estos resultados se puede comprobar que SUNSHINE se ubica entre los principales programas disponibles disponibles a día de hoy, y por lo tanto se encuentra dentro del estado del arte en cuanto a lo que tiempo de respuesta se refiere. Aún así, la diferencia que existe entre SUNSHINE y el mejor *software* medido, Parsec, es importante, con lo que aún queda espacio para optimizar aún más los procesos de codificación y captura.

Otro aspecto importante a tener en cuenta a la hora de evaluar un programa de *game streaming* es el efecto que su uso tiene en el rendimiento del juego capturado. Idealmente, gracias a que la codificación se puede hacer por *hardware* especializado presente en las tarjetas gráficas, este efecto debería ser mínimo.

Para evaluar esto se utilizó la herramienta de *benchmark* presente en el videojuego *Rise of the Tomb Raider*, que consiste en una navegación automatizada por distintos niveles del juego y culmina con un reportaje de los cuadros por segundos obtenidos durante la ejecución. Nuevamente, el códec utilizado fue H264, y el *bitrate* 5MB por segundo.

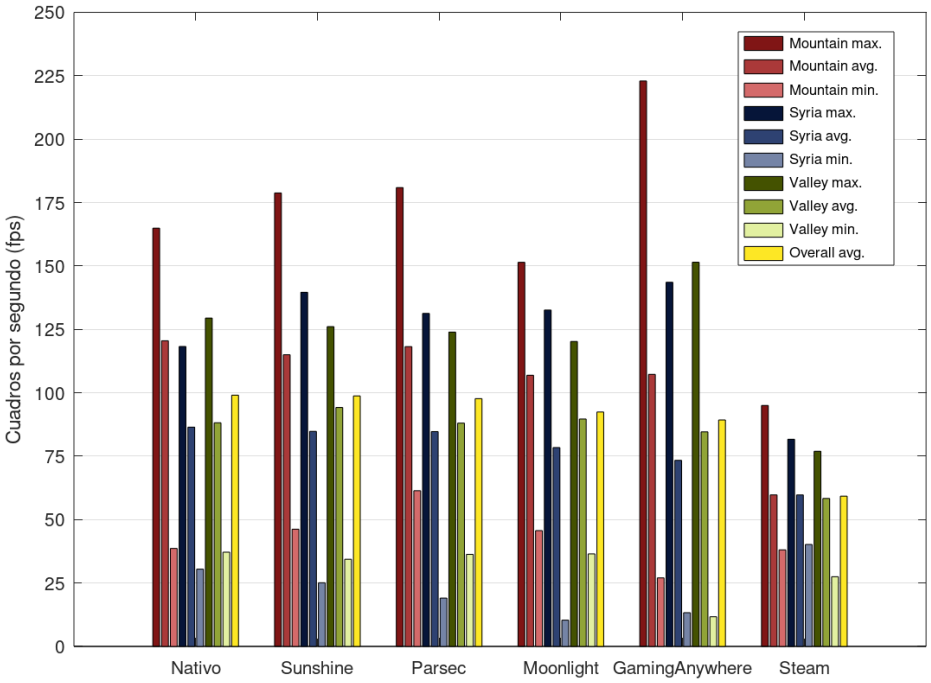

Figura 4.3: Resultados de la herramienta de *benchmark* del juego *Shadow of the Tomb Raider* siendo capturado a 5MBps

En efecto, se pudo ver que el rendimiento se vió muy mínimamente afectado por aquellos programas que utilizan *encoding* por *hardware*, mientras que en el caso de GamingAnywhere, que utiliza la CPU, el rendimiento se disminuyó un 10 %. Aún más importante, el comportamiento del *frame rate* se volvió completamente errático, algo que se puede ver en la gran diferencia entre los mínimos y los máximos medidos.

Por otro lado, parece que Steam Remote Play aplica un *cap* de 60 cuadros por segundo a los videojuegos que captura, así que los resultados que se obtuvieron no resultan de mucha utilidad.

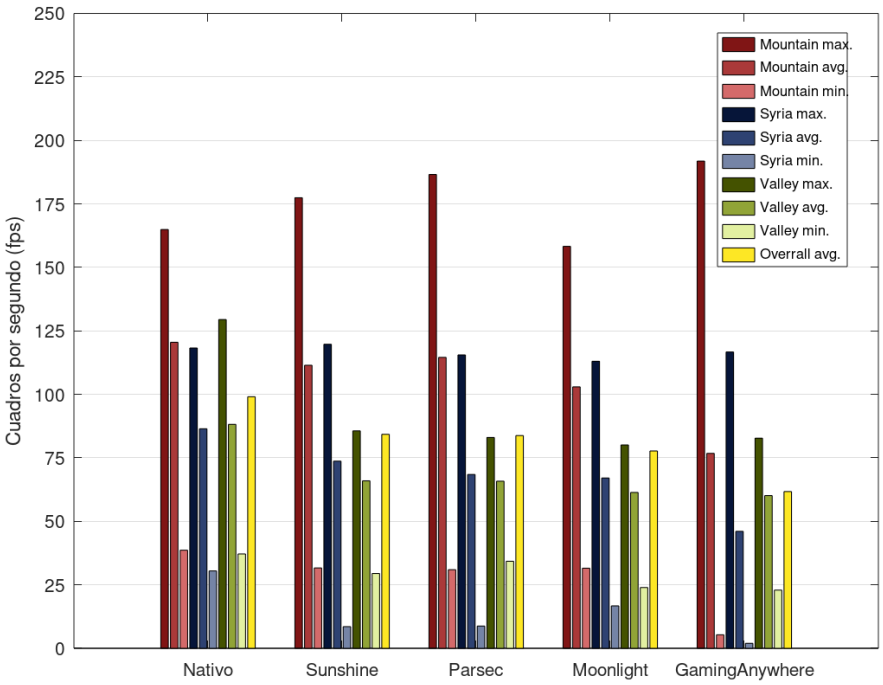

Figura 4.4: Resultados de la herramienta de *benchmark* del juego *Shadow of the Tomb Raider* siendo capturado a 10MBps

Se realizó una última prueba para evaluar el efecto que tiene incrementar el *bitrate* en el rendimiento. Se repitieron los *benchmarks*, esta vez con un *bitrate* de 10MB por segundo. Se pudo notar que esto sí tuvo un efecto en el rendimiento incluso en aquellos sistemas que utilizan *encoders* por *hardware*, mientras que GamingAnywhere resultó directamente inviable, considerando que se midieron cuadros por segundo mínimos de un sólo dígito.

Se puede afirmar que es necesario contar con un márgen de rendimiento cercano al 10 % si se quiere jugar a tráves de *game streaming* manteniendo los 60 cuadros por

segundo.

Conclusiones

Con SUNSHINE se pudo lograr el objetivo de producir una herramienta de *game streaming* de código abierto que permita una experiencia de juego con una latencia total reducida y una calidad visual aceptable. Además, se pudo llegar a un resultado con mejores características de rendimiento que otros productos comerciales que cuentan con una madurez de varios años en el mercado y el respaldo de algunas de las principales compañías en la industria de los videojuegos. No era el propósito de este trabajo reproducir toda la funcionalidad que estos productos le proporcionan a sus usuarios, sino servir como exploración de las posibilidades que existen para implementar una solución holística para este caso de uso utilizando los recursos disponibles en el *hardware* moderno, y funcionar como punto de partida a ser expandido en el futuro.

Se pudo comprobar la importancia que tiene cada paso de la cadena de adquisición, codificación y transmisión de vídeo cuando el objetivo es reducir los tiempos de espera entre cada uno. Al optimizar cada segmento de forma individual, se pudo llegar a un resultado final que cumple con las expectativas de latencia propuestas.

Durante el desarrollo del proyecto, se tocaron distintos subsistemas de *software*, cada uno relacionado con distintas materias cursadas durante la carrera de grado, incluyendo Sistemas Operativos, Protocolos de Comunicaciones y Videojuegos Avanzados. Sin embargo, también hubo mucha investigación en áreas de la computación

que no fueron abordadas a lo largo de la carrera. En particular, fue necesario adquirir un conocimiento a fondo de ciertos aspectos del sistema operativo Windows, incluyendo, entre otras cosas: el manejo de sus procesos; las capacidades presentes en su API principal, Win32; el sistema de mensajes de las ventanas; y el manejo de la entrada del ratón y el teclado. Se encontró que esta experiencia permitió tener una visión más global de los sistemas operativos en uso en la actualidad, y sirvió para tener un entendimiento más claro de que es posible lograr en una PC si se trabaja directamente con el sistema operativo, y no a través de capas de abstracciones como pueden ser el ecosistema Web o motores de videojuegos.

Para finalizar, se encontró que el proyecto resulto ser un trabajo multifacético y complejo, que requirió un largo proceso de iteración y refinamiento para lograr obtener resultados satisfactorios. Las ideas abordadas en este trabajo tienen un gran potencial para dar frutos en un producto valioso para miles de usuarios en todo el mundo, pero implementarlas en un sistema realmente funcional y usable costará aún muchas más horas de dedicación, esfuerzo y perseverancia. Esto sirvió como una lección de humildad, y de apreciación a tantos desarrolladores de *software* que hacen la vida moderna posible.

5.1. Trabajo futuro

Para que SUNSHINE sea un producto realmente usable por el público general, primero es necesario solventar las falencias que fueron relegadas por falta de tiempo en este trabajo.

En primer lugar, es muy importante el soporte de audio del juego. Eso requiere un tratamiento similar a la captura y codificación de vídeo, con la diferencia de que, en este caso, la codificación se haría enteramente en CPU. Para la captura, hay que tomar una decisión de diseño importante: realizarla mediante *hooks*, de forma similar

a la captura de vídeo, o utilizando un dispositivo virtual, como otras aplicaciones de *game streaming*. Además, seria interesante si el audio del juego no se reproduce localmente en la PC que está ejecutando el juego. Tampoco es trivial la tarea de sincronización necesaria entre el *stream* de video y el *stream* de audio.

Otro aspecto importante que esta incompleto actualmente es el soporte de las demás APIs gráficas. Si bien DirectX 11 es la API más utilizada en la actualidad, eso esta cambiando rápidamente: tanto *Call of Duty: Modern Warfare* como *Red Dead Redemption 2*, dos de los lanzamientos más esperados del 2019, no soportan DirectX 11. En cambio, utilizan DirectX 12 y Vulkan, respectivamente. Asimismo, seria muy interesante agregar soporte para el vasto catálogo de juegos más antiguos que utilizan DirectX 9 y OpenGL.

Agregar soporte para mando sería propicio también. Por un lado, es una buena forma de camuflar la latencia del sistema, considerando que el dispositivo de entrada alternativo es el ratón, cuyo movimiento en pantalla el usuario espera que imite exacta e inmediatamente el movimiento físico de su brazo. Esta expectativa hace que el uso del ratón sea particularmente sensible al efecto de la latencia.

Además del cliente de Windows, implementar clientes para otras plataformas y sistemas operativos como pueden ser iOS, Android, Chromecast y Linux, proporcionaría una forma de disfrutar de juegos de gran calidad visual en sistemas que de otra forma no podrían hacerlo, ya sea por falta de capacidad de procesamiento o porque no existen *ports* de esos juegos en esos sistemas operativos.

Por último, seria muy interesante incorporar una capa de red más robusta, utilizando técnicas de NAT traversal y UDP hole punching para permitir el uso en internet de forma *peer to peer*[31]. Este uso permitiría un sistema en que los usuarios puedan compartir sus videojuegos y tiempo de uso de la capacidad de procesamiento de sus

computadoras de forma controlada y limitada a otros usuarios, con lo que se obtendría una funcionalidad no vista hasta ahora en otros productos similares ya disponibles.

Bibliografía

[1] Kuan-Ta Chen, Yu-Chun Chang, Hwai-Jung Hsu, De-Yu Chen, Chun-Ying Huang and Cheng-Hsin Hsu. *On the Quality of Service of Cloud Gaming Systems.*

[2] Richard Leadbetter, *GDC: Why OnLive Can't Possibly Work*, Eurogamer https://www.eurogamer.net/articles/gdc-why-onlive-cant-possibly-work-article

[3] OnLive shutdown announcement, https://web.archive.org/web/20160324144108/http://blog.onlive.com/2015/04/02/a-bright-future-for-cloud-gaming-at-sony/

[4] *Sony Computer Entertainment to Acquire Gaikai Inc., a Leading Interactive Cloud Gaming Company* https://web.archive.org/web/20120703201844/http://www.sacbee.com/2012/07/01/4604219/sony-computer-entertainment-to.html

[5] PS Now website, https://www.playstation.com/es-es/explore/playstation-now/

[6] GeForce NOW beta, https://www.NVIDIA.com/en-gb/geforce/products/geforce-now/

[7] Richard Leadbetter, *Google Stadia hands-on: streaming analysis and controller impressions*, https://www.eurogamer.net/articles/digitalfoundry-2019-hands-on-with-google-stream-gdc-2019

[8] Project xCloud preview, https://www.xbox.com/en-US/xbox-game-streaming/project-xcloud

[9] Adam's Savage Tested Youtube Channel, *Hands-On with Google Stadia and Technology Deep Dive!*, minuto 9:10, https://www.youtube.com/watch?v=eouqoIur5c4

[10] Akamai, *State of the internet report Q1 2010*, p. 28, https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/akamai-q1-2010-state-of-the-internet-connectivity-report.pdf

[11] Akamai, *State of the internet report Q1 2017*, p. 54-55 https://www.akamai.com/de/de/multimedia/documents/state-of-the-internet/q1-2017-state-of-the-internet-connectivity-report.pdf

[12] GamingAnywhere, http://gaminganywhere.org/perf.html

[13] Moonlight, https://moonlight-stream.org/

[14] Advanced Media Framework, https://github.com/GPUOpen-LibrariesAndSDKs/AMF

[15] NVIDIA VIDEO CODEC SDK, https://developer.NVIDIA.com/NVIDIA-video-codec-sdk

[16] Windows Dev Center, *Dynamic-Link Libraries*, https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-libraries

[17] Ivo Ivanov, *API hooking revealed*, 2002, https://www.codeproject.com/Articles/2082/API-hooking-revealed

[18] Windows Dev Center, *LoadLibrary*, https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya

[19] MinHook, *How it works*, https://www.codeproject.com/Articles/44326/MinHook-The-Minimalistic-x-x-API-Hooking-Libra

[20] *Microsoft Research Detours Package Overview*, https://github.com/microsoft/detours/wiki

[21] Windows Dev Center, *About Keyboard Input*, https://docs.microsoft.com/en-us/windows/win32/inputdev/about-keyboard-input

[22] Windows Dev Center, *About Mouse Input*, https://docs.microsoft.com/en-us/windows/win32/inputdev/about-mouse-input

[23] Windows Dev Center, *About Raw Input*, https://docs.microsoft.com/en-us/windows/win32/inputdev/about-raw-input

[24] Windows Dev Center, *About Messages and Message Queues*, https://docs.microsoft.com/en-us/windows/win32/winmsg/about-messages-and-message-queues

[25] Windows Dev Center, *Desktop Duplication API*, https://docs.microsoft.com/en-us/windows/win32/direct3ddxgi/desktop-dup-api

[26] Justin Stenning, *C# – SCREEN CAPTURE AND OVERLAYS FOR DIRECT3D 9, 10 AND 11 USING API HOOKS* http://spazzarama.com/2011/03/14/c-screen-capture-and-overlays-for-direct3d-9-10-and-11-using-api-hooks/

[27] Pinar Akyazi and Touradj Ebrahimi, *Comparison of Compression Efficiency between HEVC/H.265, VP9 and AV1based on Subjective Quality Assessments*

[28] Rate control, *NVENC Video Encoding Programming Guide*, p. 9-10

[29] libavcodec *hardware decode sample*, https://github.com/FFmpeg/FFmpeg/blob/master/doc/examples/hw_decode.c

[30] Gaffer on Games, *UDP vs. TCP*, https://gafferongames.com/post/udp_vs_tcp/

[31] Bryan Ford, Pyda Srisuresh and Dan Kegel, *Peer-to-Peer Communication Across Network Address Translators*, https://bford.info/pub/net/p2pnat/