UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Sistemas Informáticos y Computación

Universitat Politècnica de València

SIMULADOR DE EVENTOS DISCRETOS PARA UNITY

Proyecto final de máster.

Máster en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

Curso 2018/2019

Autor: Adrián Carboneras Mas
Tutor: Ramón Pascual Mollá Vayá

Resumen

Este proyecto es un estudio de viabilidad de la integración de un motor de simulación de eventos discretos que opera en tiempo real (RT-DESK) dentro del popular motor de videojuegos Unity. Para ello se empleará una biblioteca desarrollada en C++ en un doctorado anterior y una adaptación propia que permita utilizar RTDesk directamente por los videojuegos desarrollados en Unity.

El objetivo es evaluar la viabilidad del uso de un simulador de eventos discretos en Unity y que mejoras puede aportar a los videojuegos ya creados o en desarrollo.

De esta forma a lo largo del proyecto se muestra los pasos necesarios para adaptar RT-DESK al entorno de desarrollo Unity, la comunicación con Unity y las pruebas realizadas en los juegos de muestra.

Para ello se ha seguido una metodología incremental que permitía implementar trozos de RT-DESK y testearlos de forma iterativa para, al final asegurarse de que el simulador al completo funcionaba correctamente.

Al final se aportan pruebas de una mejora del rendimiento, especialmente en la tasa de refresco para la misma carga de computación o bien un mantenimiento de la tasa de refresco al aumentar dicha carga.

Para ello, se ha creado un prototipo de juego rápido con un plugin de creación de juegos comerciales suministrado por el propio fabricante sobre el que se han realizado las pruebas de campo.

Así mismo se ha conseguido mejorar el rendimiento en las aplicaciones que usen el motor y se ha implementado un sistema de mensajes que permite a los usuarios crear sus propios mensajes entre sistemas de un juego, y controlar la frecuencia de muestreo de todos los sistemas.

Palabras clave: Videojuegos, Unity, simulación, motores, eventos discretos, tiempo real.

Índice de Contenido

1. Introducción 5

1.1 Historia de los motores de videojuegos 5

1.2 Funcionamiento de los motores de videojuegos 6

1.2.1 Unity 7

1.2.2 Unreal Engine 4 9

1.2.3 Love 2D 10

1.2.4 Ogre3D 11

1.3 Motivación 12

1.4 Objetivo 13

1.5 Metodología 13

1.6 Estructura de la obra 0

2. Estado del arte 1

2.0.1 Unreal 3

2.0.2 Ogre3D 5

2.0.3 Love2D 6

2.0.4 Unity 8

2.1 Modelos de simulación en videojuegos 10

2.1.1 El modelo simple acoplado 10

2.1.2 Modelo sincronizado acoplado 12

2.1.3 Modelo simple desacoplado 13

2.1.4 Comparaciones 14

2.2 Crítica al estado del arte 16

2.3 Propuesta de mejora 19

2.3.1 Evolución histórica de RTDesk 19

2.3.2 Descripción de las ventajas de RTDesk 19

3 Análisis del problema 21

3.1 Formas de implementar código de C++ en Unity. 22

3.2 Análisis DAFO del proyecto 30

3.3 Solución propuesta 31

3.4 Presupuesto 32

4. Diseño de la solución 33

4.1 Arquitectura del sistema 33

4.2 Diseño detallado 33

4.3 Tecnología utilizada 34

5 Desarrollo de la adaptación de RTDesk a Unity 34
6 Implantación 41
7 Pruebas 42
7.1 Prueba de frecuencia de muestreo 43
7.2 Pruebas de mejoras de físicas 53
8 Conclusión 54
9 Trabajo futuro 55
Anexo 56
Guía de uso 56
Glosario 58
Bibliografía 59

Índice de Ilustraciones

Ilustración 1 Esquema de funcionamiento de un juego 5
Ilustración 2 Videojuego hecho con la Doom engine. Parece que el juego sea en 3D, pero en realidad todo es 2D. 6
Ilustración 3 Logo de Unity 7
Ilustración 4 Dos ejemplos de juegos realizados en Unity, Hands of Fate 2 primero y Cuphead el segundo. Unity permite realizar tanto juegos en 2D como en 3D con una alta calidad 8
Ilustración 5 Logo de Unreal Engine 9
Ilustración 6 Kingdom Hearts 3 es un ejemplo de un juego realizado en Unreal Engine para Playstation 4 y Xbox One 10
Ilustración 7 Logo de Love2D 10
Ilustración 8 Move or Die es un ejemplo de un juego exitoso realizado con Love 2D 11
Ilustración 9 Logo de Ogre3D 11
Ilustración 10 Torchlight 2 es un ejemplo de un juego realizado con Ogre3D 12
Ilustración 11 Ejemplo de modelo incremental [2] 14
Ilustración 12 Diagrama de Gantt del proyecto 1
Ilustración 13 Representación de un bucle de juego, que aparece en el libro Game Programming Patterns [3] 1
Ilustración 14 Ciclo de ejecución de los actores según su grupo de actualización. 4
Ilustración 15 Posible implementación del ciclo de vida del motor Ogre3D [4]. 5
Ilustración 16 bucle de juego de Love2D 6
Ilustración 17 El bucle de juego de Love2D 7
Ilustración 18 Ciclo de un script de Unity 8
Ilustración 19 Esquema que describe el modelo simple acoplado 11
Ilustración 20 Esquema que explica el funcionamiento del modelo sincronizado acoplado 13
Ilustración 21 Esquema de funcionamiento del modelo simple desacoplado 14
Ilustración 22 Esquema de funcionamiento del caso de un juego de peleas 17
Ilustración 23 Esquema de funcionamiento del caso de un juego de peleas en el que un jugador ataca a la vez que otro esquiva. 17
Ilustración 24 Esquema de la propuesta de mejora, en este caso un simulador de eventos discretos desacoplado 20

Ilustración 25 Implementación del código escrito en CLI 22
Ilustración 26 Los scripts de Unity que se comunican con la dll 23
Ilustración 27 Cambios añadidos a la DLL usando el segundo método 25
Ilustración 28 Comunicación de Unity con la dll 27
Ilustración 29 Análisis dafo del proyecto 30
Ilustración 30 Estructura del bucle nuevo que simplifica el de Unity 33
Ilustración 30 Imagen que muestra los archivos que contienen los extern, y como estos solo llaman a funciones de C++ y hacen visibles funciones para C#. 35
Ilustración 31 Función que crea una clase ManagedEntity y la otra función que ejecuta las funciones de C# en C++. 36
Ilustración 32 Una de las reorganizaciones de código 36
Ilustración 33 Pseudocódigo de Unity 38
Ilustración 34 Pointers y funciones que almacena la clase proxy 38
Ilustración 35 Como modificar el tiempo de ejecución de FIxedUpdate, solo cambiar el número de fixed Timestep 39
Ilustración 36 Las 3 colas donde se añaden las funciones y donde se ejecutarán. 39
Ilustración 37 La interfaz creada 40
Ilustración 38 La función que crea un objeto de MyProxyClass y crea los punteros de la función que implemente la interfaz, sin que el usuario haga nada más que heredar de la interfaz 40
Ilustración 39 Como añadir la función Update al array. 41
Ilustración 40 Como crear una clase customizada 42
Ilustración 41 Imagen del juego donde se ven los zombies y como el jugador puede disparar a los zombies. La mano y el sistema de disparos los aporta el plugin UFPS. 43
Ilustración 42 Gráfica que muestra los fps de RTDesk sin mirar a los zombies. 44
Ilustración 43 Gráfica que muestra los fps del mismo juego sim implementar RTDesk 45
Ilustración 44 Gráfica que muestra los frames si cada objeto tiene su propio gestor de mensajes que gestiona los Updates, FIxedUpdate y LateUpdates. 46
Ilustración 45 Framerate cuando la cámara renderiza los zombies con RTDesk y el render pipeline modificado. 47
Ilustración 46 Framerate sin mirar a los zombies de RTDesk con el render pipeline modificado. 48
Ilustración 47 Framerate del juego sin RTDesk con el render pipeline modificado. 49
Ilustración 48 Rendering pipeline en funcionamiento con RTDesk, frame en el que no renderiza nada 50
Ilustración 49 IdleTime de Unity 50
Ilustración 50 Idletime de RTDesk 51
Ilustración 51 Idletime de RTDesk 2, intentando conseguir 60 frames por segundo. 52
Ilustración 52 IdleTime de RTDesk tras subir el tiempo de fixedUpdate a 0.2 52
Ilustración 53 Colisión con el muro exitosa. Cuando RTDesk fallaba, se quedaba la bola al otro lado del muro 53
Ilustración 54 Código para implementar RTDesk 1 57
Ilustración 55 Segunda parte de la implementacion de RTDesk que debe de ser introducida en una funcion Start 58

1. Introducción

Los motores de videojuegos son un conjunto de herramientas que permiten a un usuario emplear todos los componentes de un videojuego [1] tales como el motor de renderizado, el de sonido, físicas, funciones online, la entrada del jugador y la IA sin tener que reescribirlos para cada juego.

El esquema de un entorno de programación de videojuegos sería algo análogo a lo mostrado en la Ilustración 1:

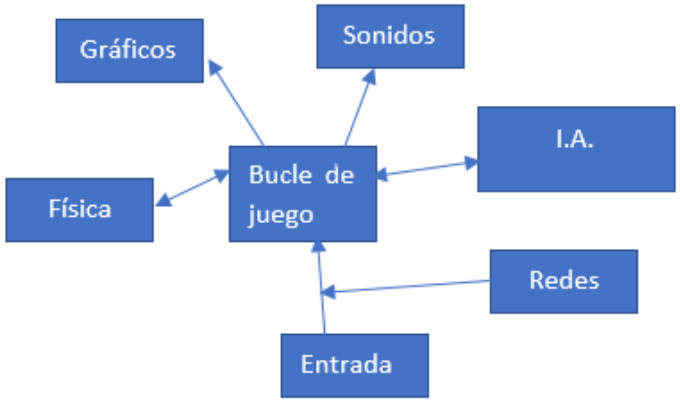

Ilustración 1 Esquema de funcionamiento de un juego

En él se observa que cada motor realiza una tarea específica para la cual está especializado. Existe un núcleo que aglutina a cada aspecto y que presenta la forma de bucle de un juego. Mientras que los motores son agnósticos para cada juego, cualquier cosa creada fuera de los motores tales como los personajes o las acciones realizadas en los juegos, formarían parte del bucle de juego y no del motor y por lo tanto podría ser código más difícil de reutilizar Por esto, los motores son independientes al bucle de juego con el objetivo de ser reutilizados fácilmente, y, en el caso de que los motores sean actualizados, no tener que modificar código del juego.

Debido al coste de crear un motor desde cero, existen pocos estudios que creen sus propios motores. La inmensa mayoría optan por el uso de motores comerciales. Entre ellos se destacan el motor Cryengine, Unity o Unreal Engine.

1.1 Historia de los motores de videojuegos

La primera utilización de un motor de videojuegos conocida consistió en el motor Doom. Este era un motor 2D que fingía renderizar en 3D los juegos, y que, tras la liberación del código en 1997, permitió la creación de mods para el juego Doom de forma muy sencilla, así como el poder trasladar Doom a cada dispositivo. Más tarde, XnGine y Quake Engine en 1995 y 1996 fueron los primeros motores de videojuegos en 3D. Y tras estos, otros motores más populares fueron:

- **GoldSrc**, que evolucionó en la Source Engine de Valve, donde se crearon juegos como el Counter Strike o Half life, y aunque es de libre uso, no es muy usado debido a su complejidad,
- **Unreal Engine**, que fue evolucionando hasta su tercera versión donde se convirtió en un producto comercial y que ya tiene una versión 4.
- **CryEngine** de Crytek, también es un motor comercial pero no muy extendido debido a su complejidad y poca documentación comparado con otros motores. Amazon usó el código de este motor y creo su propio motor llamado Lumberjack para uso interno de sus propios estudios.
- **Unity**, motor más utilizado por los pequeños desarrolladores debido a su sencillez y potencia, tal y como ellos dicen en su web[1].
- Otros motores menos usados, aunque más que Cryengine y Source, pero también importantes son Ogre3D, debido a su antigüedad y al hecho de que sea libre o Love2D debido también a que sea libre y a su sencillez.

Ilustración 2 Videojuego hecho con la Doom engine. Parece que el juego sea en 3D, pero en realidad todo es 2D.

1.2 Funcionamiento de los motores de videojuegos

Cada motor funciona de forma diferente y, aunque explicarlos a fondo variaría demasiado del objetivo del proyecto, sí que se puede repasar por encima su funcionamiento básico, comenzando por Unity, motor elegido en este proyecto, razón que se explicará en el capítulo 2.

[1] https://unity3d.com/unity

1.2.1 Unity

Ilustración 3 Logo de Unity

La sencillez que ha convertido a Unity en el motor más usado, como se ha explicado anteriormente en el apartado historia de los videojuegos, proviene de su lenguaje de programación y de su funcionamiento y sencilla interfaz.

- Primero de todo, Unity carga un archivo .unity que contiene información de la escena. Esta información es un árbol que contiene objetos que pueden estar activos o dormidos, en una jerarquía. Si están activos se representarán en pantalla en una simulación en 3D y si están dormidos su información seguirá en memoria sin borrarse, pero no se representarán para ahorrar costes en la tarjeta gráfica o procesador.
- Estos objetos llamados entidades, son los objetos que Unity recorrerá en un bucle cuando se ejecute. Si estos objetos tienen una acción anclada a ellos, tales como código en un script, información de un sonido, un modelo o imágenes en 2D, Unity ejecutará el script, mostrará el modelo o la imagen y reproducirá el sonido en el orden que el usuario le comunique.

Para ejecutar un script, el cual debe estar escrito en C#, estos deben heredar (13) de una clase llamada Monobehaviour. MonoBehaviour hace uso de la librería .Net, aunque no es compatible con todas las funciones de dicha librería, por lo que no se puede programar en Unity como se programaría en C# y .Net de forma normal. La clase Monobehaviour contiene varias funciones virtuales (14), entre ellas, las más importantes:

- **Awake y Start**. Dos funciones que se ejecutarán una única vez en el primer recorrido de una escena si el script que está anclado al objeto se encuentra habilitado.
- **OnEnable**: Función que se ejecuta cada vez que una Entidad se activa, en el primer recorrido del bucle tras su activación.
- **OnDisable**: Función que se ejecuta cada vez que una Entidad se desactiva, en el primer recorrido del bucle tras su desactivación.
- **Update**: Esta función es una función que se ejecuta en cada recorrido del bucle de juego.
- **LateUpdate**: Esta función se ejecutará siempre después de Update.
- **FixedUpdate**: Esta función tiene un tiempo fijo de ejecución establecido por el usuario en un menú de Unity. Se ejecutará cada vez que pase ese tiempo siempre que Unity pueda.

- Existen más funciones específicas que no tienen nada que ver con este proyecto y que extenderían esta explicación demasiado, por lo que se obvian, pero siempre se puede consultar la Api de Unity si se desea saber cuáles son.

Todas las funciones que se ejecutan dentro de un script de Unity aparecen en la Ilustración 18 y se explicará, junto al bucle del resto de motores, más detenidamente en el capítulo 2. Para más información sobre las funciones de Monobehaviour, se puede consultar la página web de Unity[2].

Ilustración 4 Dos ejemplos de juegos realizados en Unity, Hands of Fate 2 primero y Cuphead el segundo. Unity permite realizar tanto juegos en 2D como en 3D con una alta calidad

[2] https://docs.unity3d.com/ScriptReference/

1.2.2 Unreal Engine 4

Ilustración 5 Logo de Unreal Engine

El siguiente motor para explicar es Unreal, el mayor competidor de Unity en el mundo de los motores comerciales que funciona bastante parecido a este.

Unreal también recorre un grafo de escena para ejecutar scripts adjuntados a entidades. La diferencia es que Unreal permite crear estos scripts con un lenguaje de programación visual con bloques en vez de código, aunque se puede utilizar C++ si se desea.

En el lado de C++, hay que importar la librería de Unreal para utilizar funciones que pertenezcan a AActor, dentro de una entidad. AActor es una clase que contiene todas las funciones predefinidas para una entidad de Unreal.

La función que se ejecuta dentro de cada entidad en un recorrido del bucle se llama Tick en el caso de Unreal y funciona igual que el método Update de Unity. En la documentación de Unreal se explica más a fondo todos los ticks de cada tipo de entidad del motor[3], entre ellos que el orden de ejecución de estos será diferente para cada tipo de entidad. El tipo es asignado por el usuario en el constructor, que es la primera función que se ejecuta en C++, del script de la entidad. El orden en el que se ejecutan se puede ver en la Ilustración 14.

En esta ilustración se observa que, como se ha dicho, primero se asigna un grupo a la entidad. Tras esto, la entidad se encuentra en un bucle, cuya función tick se ejecutará antes, en medio o después de que las físicas se actualicen, según el usuario quiera o no. Tras sus actualizaciones, Unreal realizará operaciones del motor para renderizar y actualizar el mundo y volverá a ejecutar el bucle.

[3] https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Actors/Ticking/index.html

Ilustración 6 Kingdom Hearts 3 es un ejemplo de un juego realizado en Unreal Engine para Playstation 4 y Xbox One

1.2.3 Love 2D

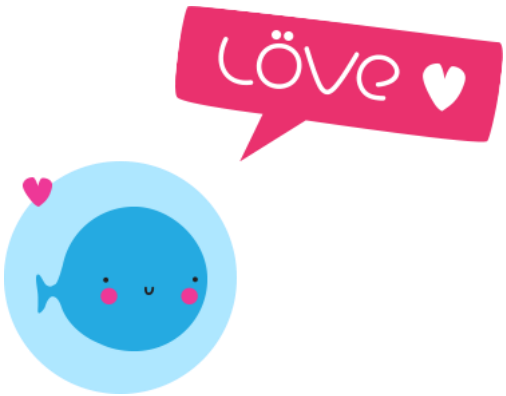

Ilustración 7 Logo de Love2D

Un motor más sencillo es Love2D, el cual se programa en Lua. Este motor es más sencillo de comprender para un programador ya que no hay interfaz ni escenas que recorrer para ejecutar entidades.

Love2D se ejecuta de forma que cualquier clase que importe su librería y tenga una función love.load() se ejecutara y creara el objeto de la clase dejándolo en memoria. La función love.update(dt) es parecida al tick y update de Unreal y Unity respectivamente, con la diferencia de que se le pasa un tiempo entre cada ejecución de la función. Por último en Love2D hay una función necesaria llamada love.draw en cada clase para dibujar los gráficos en la pantalla que representen a este objeto. En esta página web

patrocinada por una wiki de los creadores de Lua se ofrece con más profundidad un ejemplo de cómo crear un juego en Love 2d.[4]

Esta descripción del bucle aparece en la Ilustración 17 y se analizará con más detalle en el capítulo 2.

Ilustración 8 Move or Die es un ejemplo de un juego exitoso realizado con Love 2D

1.2.4 Ogre3D

Ilustración 9 Logo de Ogre3D

Por último, Ogre3D es muy diferente al resto de motores. Ogre3D es tan solo un motor de renderizado escrito en C++ con un paradigma orientado a objetos. Esto significa que de cada clase de Ogre3D se puede instanciar un objeto, no entidades como Unity o Unreal. La diferencia es que las entidades se representan con tablas en un esquema de bases de datos y los objetos son referencias a una estructura de datos en memoria.

Para hacer uso de Ogre3D, primero hay que crear un objeto llamado Root. Este objeto es tan solo un puntero (dirección que apunta a memoria, donde está el objeto) al resto de objetos del sistema e inicializa el motor con una llamada a una función StartRendering.

El siguiente paso sería añadir entidades con el objeto Entity, el cual solo es información de un objeto en el mundo. Almacena la posición y la representación del objeto en 3D, por ejemplo, un modelo en 3D de un coche moviéndose. Estas entidades, tras crearse con Entity::Createentity han de añadirse a un objeto que forma parte de Root llamado SceneManager.

[4] http://sheepolution.com/learn/book/5

Scenemanager contiene una referencia a cada objeto que existe en el mundo en 3D, sea una entidad como se ha definido antes, una luz o una cámara, con el objetivo de no tener que usar una lista con todos los objetos creados para ahorra memoria. Cuando a un objeto se le diga que debe ser renderizado usando una función Render que viene en cada entidad de Ogre, este llamará a SceneManager para ser renderizado sin que el programador tenga que hacer nada.

En cuanto al bucle que revisará si cada objeto ha de ser renderizado, será el bucle de juego y debe ser creado por el usuario. Ogre no tiene ninguna función de actualización de escenas como los otros motores tipo Update. En la página de los creadores se puede buscar más información sobre el api y que función realizan todas las clases mencionadas con más profundidad.[5]

No hay un gráfico que muestre el funcionamiento de Ogre, pero sí que se mostrará un ejemplo de código del bucle de juego que pueda usar este motor en el capítulo 2 en la Ilustración 15.

Ilustración 10 Torchlight 2 es un ejemplo de un juego realizado con Ogre3D

1.3 Motivación

- El principal motivo que me llevó a trabajar en este proyecto fue mi amplia experiencia con Unity. Previamente ya había realizado proyectos, pero ninguno que estuviera relacionado con la modificación del motor en sí.
- Cada vez que creaba un videojuego en el motor aparecía una falta de rendimiento en las plataformas para las que se creaban los proyectos, por lo que la implementación de un simulador que pudiese suplir algunas carencias del motor resultaba interesante tanto personalmente como profesionalmente.
- Además, esto me permitiría aumentar mi experiencia con la cantidad de proyectos diferentes realizados en Unity.

[5] https://www.ogre3d.org/docs/api/1.9/

- Mi interés personal hacia la creación de videojuegos y por lo tanto al estudio de un motor de videojuegos. Este interés me llevo a estudiar, la carrera de diseño y desarrollo de videojuegos en Castellón que posteriormente amplié con el master de inteligencia artificial, reconocimiento de formas e imagen digital. El añadido fue poder escoger un proyecto donde es posible diseccionar y mejorar un motor comercial usado ampliamente en la industria del videojuego.
- El hecho de que el motor estuviera desarrollado en C++ fue también un aliciente dadas mis carencias al programar en el lenguaje C++. Al estudiar el código en el que se basa el simulador de eventos discretos y cómo funciona un gestor de eventos en C++, mi conocimiento del lenguaje también mejoró notablemente.
- El realizar plugins que modifiquen el comportamiento de un motor puede ser un trabajo importante que nunca he realizado y que fue otro factor a favor de la realización de este proyecto.

Por supuesto, implementar un motor de simulación discreto en tiempo real en Unity es una novedad que podría resultar interesante para empresas que quisieran usar este tipo de simuladores y sería una ventaja usar Unity para las simulaciones debido a su facilidad de uso.

1.4 Objetivo

El objetivo de este proyecto consiste en varias fases:

1. Estudiar los distintos estados del arte de los motores de videojuego y, en especial de Unity, para poder averiguar cómo mejorarlos.
2. Estudiar las distintas formas de implementar una librería de C++ en C#, y más concretamente en Unity, debido a que Unity no funciona con todas las funciones de C# y .Net.
3. Al haber definido las soluciones posibles, se escogerá la mejor y se llevará al detalle cómo se implementará. Esto se explicará en el capítulo 4 *4. Diseño de la* solución
4. Tras esto, se implementará el simulador de eventos discretos en C#, usándolo como plugin de C# en Unity. Esto se realizará simplemente realizando los cambios necesarios en el código y compilando el simulador como una dll. Para acabar se implementará la parte del motor de Unity que ha de comunicarse con la dll.
5. La confirmación de que el motor de simulación RTDESK es adecuado y mejora las funciones de Unity. Desde una mejora en la frecuencia de actualización del juego en videojuegos comerciales profesionales o una mejora de las físicas que implementa Unity intentando ser realistas y que se comporten de forma determinista. Para ellos realizaremos un banco de pruebas usando un videojuego oficial.
6. Finalmente, la documentación de todo el trabajo realizado en este proyecto.

1.5 Metodología

La metodología escogida para el proyecto, debido a la experimentación y complejidad es un modelo incremental:

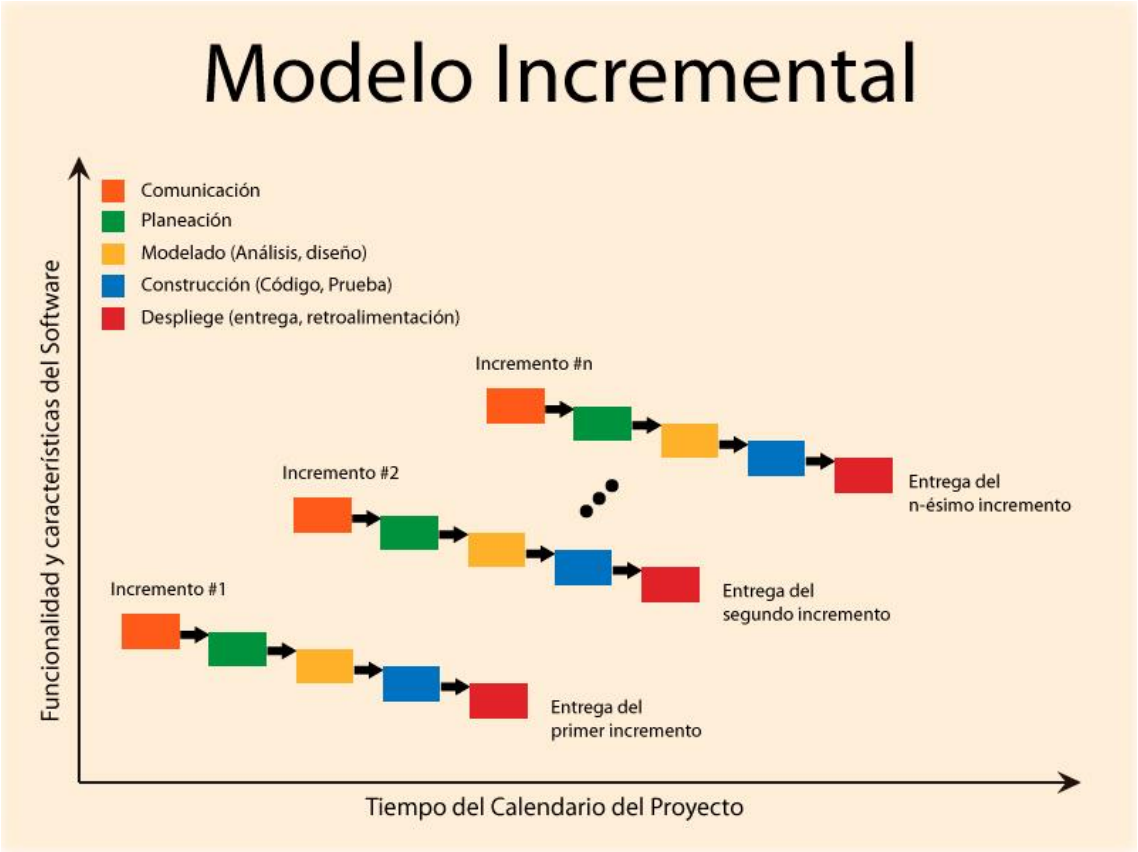

Ilustración 11 Ejemplo de modelo incremental [2]

Debido a que este proyecto lo forma solo una persona, no habrá fase de comunicación. En cuanto al resto de fases:

- En la fase de planeación se investigará las características de RTDESK y cómo implementar esa característica en Unity.
- En la fase de modelado se diseñan las clases que serán creadas o modificadas.
- En la fase de implementación, se programa las clases elegidas y se comprueba que funcionen.
- Durante el despliegue se comprueba que funcionen como es debido y dependiendo de ello se pasa a una nueva función o se modifica.

Como ya se ha mencionado, la naturaleza experimental del proyecto, y el hecho de que este basado en una implementación ya realizada en C++, favorece este tipo de metodología en el que se podrá iterar la mejor forma de usar el simulador de eventos discretos dentro de Unity. También ayudara a encontrar la mejor forma de poder ejecutar las funciones Update, LateUpdate y FixedUpdate, ya que, en cuanto se consiga implementar una, las demás se podrán implementar basadas en la iteración anterior. Además de que las pruebas básicas bastarían con solo una de las funciones implementadas.

Las pruebas finales también estarán dentro de este modelo ya que, aparte de la comprobación de la frecuencia de muestreo que será simple, sí que habrá que diseñar e implementar alguna ecuación física para probar si la simulación es correcta.

En la Tabla 1 aparece cómo se repartirán las tareas temporalmente. Esta estimación se basa en el número de créditos requeridos en el proyecto de final de master.

	Agosto 1-14	Agosto 14-28	Agosto 28 - Sept 11	Sept 11-25	Sept 25 – Octubre 9	Octubre 9 -23	Octu 23 – Noviembre 6	Noviembre 6-20	Noviembre 20-30
Investigación	4	4	4	4	4				
Adaptación de código de C++	10	10	14	8	12	8	8		
Programación de la comunicación de la librería con Unity			10	10	14	8	12		
Test de funcionamiento			2	2	2	2	2		
Test de rendimiento					7	7	10	10	6
Escritura de la memoria			7	13	14	8	10	8	10

Tabla 1 Tabla con las horas trabajadas en el proyecto cada dos semanas

Y a continuación el diagrama de Gantt en la Ilustración 12. El diagrama refleja que el código siempre se encuentra en constante evolución, mientras que la memoria, la investigación o las pruebas, se pararán debido a que siempre dependen de la programación, y si ciertos hitos en la programación no se han acabado, no se podrán realizar las tareas.

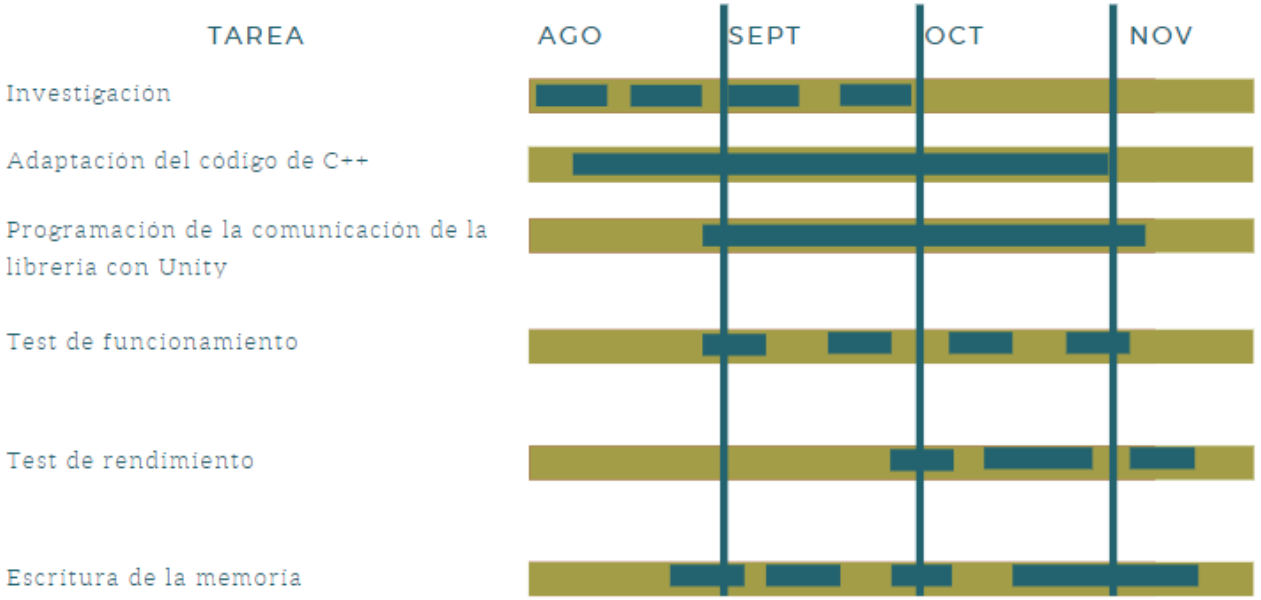

Ilustración 12 Diagrama de Gantt del proyecto

1.6 Estructura de la obra

A continuación, en el documento se describirá el estado del arte actual en los motores de videojuegos junto a una crítica a este y una posible solución. Tras esto se realizará un análisis del problema de implementar RTDesk en Unity, las posibles soluciones existentes y cuál es la elegida, el diseño de la solución y su desarrollo,.

Se ha realizado una mención explícita, a la forma en la que se implantó esta solución en Unity. Se han analizado las dificultades y los pasos seguidos para poder insertar el motor en el código.

Para acabar, se mostrarán las pruebas realizadas y una conclusión sobre los resultados obtenidos tras las pruebas. Unas conclusiones acerca de las dificultades y ventajas que ha aportado implementar RTDesk en Unity y una relación de trabajos futuros completará el contenido de esta memoria.

La descripción de cada uno de los apartados se explica a continuación:

1. El estado del arte, la crítica y una posible solución aplicable aparece explicada en el capítulo 2 *Estado del arte.*
2. Las distintas maneras posibles de implementar la librería de C++ en Unity y la problemática del proyecto se explica en el capítulo 3, *Análisis del problema.*
3. Los detalles de la implementación escogida aparen en el capítulo 4 *4. Diseño de la* solución
4. Las modificaciones realizadas a la librería en la que se basa el proyecto y la implementación del código para comunicar Unity con la librería se explican en los capítulos 5 y 6 *Desarrollo de la adaptación de RTDesk a Unity* e *Implantación*.
5. En el capítulo 7 *Pruebas, se describen las pruebas realizadas para confirmar que el proyecto funciona como debería y sus limitaciones, además de una comparación con cualquier sistema de Unity que se le parezca.*
6. Finalmente, las conclusiones sacadas del proyecto y una resolución aparecen en el capítulo 8 Conclusión, junto a una descripción breve de todos los trabajos posibles que se podrían haber continuado de este proyecto en el capítulo 9 Trabajo futuro.

2. Estado del arte

La enorme variabilidad en la carga de trabajo de un juego no permite asegurar una frecuencia fija de recorrido del buble principal del juego, Esta puede variar según la carga de cualquiera de las fases en las que se divide la ejecución de un videojuego: visualización, audio, simulación física,… y el procesador tiene una potencia de cálculo limitada. El mayor problema proviene de que las instrucciones se suelen ejecutar secuencialmente en un juego. Por ejemplo, el modelo típico de un bucle de juego consiste en:

- Buscar un input de un jugador, tanto online como offline.
- Procesar los eventos del juego en función del input de un jugador, tal y como hacer saltar a un personaje y aplicar las físicas de salto al modelo que represente al jugador.
- Renderizar el resultado en pantalla y ejecutar los sonidos.

Este proceso aparece en el libro *Game Programming Patterns, by Robert Nystrom* [3]*, representado en la* Ilustración 13.

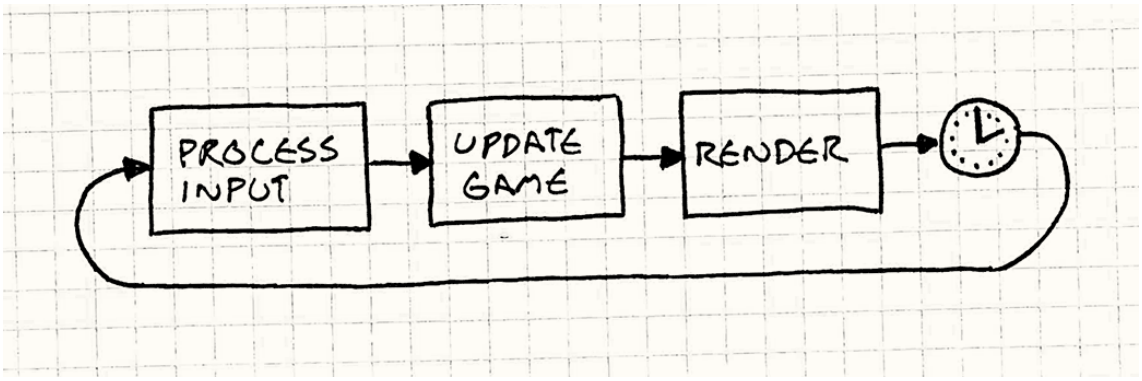

Ilustración 13 Representación de un bucle de juego, que aparece en el libro Game Programming Patterns [3].

Este bucle de juego puede ejecutarse de dos maneras diferentes, según su manera de manejar los eventos del juego:

- **Continuo**: En este bucle, cada cambio de estado de la simulación se actualiza en función del tiempo real transcurrido desde la última iteración del bucle. La mayoría de los motores de videojuegos funcionan así. En cada iteración del bucle se calcula un tiempo transcurrido (delta de tiempo) desde la última vez que se recorrió este bucle y con ese delta calculado, se pasa a la actualización de todos los estados de todos los objetos del juego. En la explicación de la Ilustración 17 aparece este tipo de bucles en forma de código.
- **Discreto**: El tiempo lo dicta la ejecución de los eventos que ocurren en el juego, lo cual significa que el tiempo avanza cuando un evento se produce. Estos eventos se siguen ejecutando de forma continua y pueden estar sincronizados con el tiempo real. SuperHot, un videojuego bastante famoso por su jugabilidad

consiste en que el tiempo de juego no pasa hasta que el jugador se mueve, lo cual es un ejemplo de este tipo de bucles.

La lista de eventos se puede manejar de dos formas diferentes:

- **Ciclo acoplado**: Todas las fases del bucle de juego se ejecutan con la misma frecuencia y siempre pasa la misma cantidad de tiempo entre cada fase, suponiendo que siempre haya suficiente potencia de cálculo.

- **Ciclo desacoplado**: Cada fase puede tener una frecuencia distinta. Por ejemplo, el renderizado puede ejecutarse 60 veces en un segundo mientras que todos los eventos relacionados con las físicas pueden ejecutarse a 120, lo cual produce físicas más precisas al mejorar la precisión de los interpoladores numéricos que resuelven las ecuaciones diferenciales de simulación física.

Como se ha introducido en el capítulo anterior, los motores de videojuegos que se han enseñado en el capítulo 1.2 Funcionamiento de los motores de videojuegos, contienen un bucle donde se recorre una escena. También se ha mencionado que Unity es el motor más usado tanto por estudios pequeños con menos de 10 empleados como grandes estudios que pueden llegar a ser de más de 2000 empleados, debido a su sencillez y potencia frente a los demás motores. Y las diferencias que separan a Unity de los demás motores son:

- Por ejemplo, debido a su uso tan extendido, Unity tiene una comunidad enorme de guías y soluciones a problemas, además de assets,(que son todos los objetos en 3d, imágenes en 2D o música de un juego) extra. Unity es un conjunto de herramientas que incorporan un motor gráfico 3D y 2D, inteligencia artificial, sonido, multijugador y un motor de físicas. Normalmente physx, pero desde su última versión también incluye Havock.
- Unreal por el contrario es otro conjunto de herramientas que contiene los mismos motores que Unity, algunos más avanzados como la inteligencia artificial o el de renderizado, pero a causa de ser más potente también es más complejo usarlo, debido a que usa C++ como lenguaje de programación. Como ventaja frente a Unity, Unreal permite modificar partes del motor escribiendo en C++ las modificaciones, no se permite usar el lenguaje visual para las modificaciones.

- Ogre3D, a diferencia de los dos anteriores, es solo un motor gráfico. Esto significa que el usuario debe fabricarse su bucle de juego e incorporar el motor de Ogre3D dentro. Hay diversos tutoriales de cómo usarlo y debido a que solo es un motor gráfico, su uso permite más libertad a la hora de crear un juego, pero también más dificultad.

- Por último, Love2D, es otro conjunto de herramientas con un motor de sonido y gráfico 2D. Debido a que está creado en Lua (lenguaje interpretado) y solo pensado para el 2D su uso es muy sencillo, además de tener una amplia documentación.

A continuación, evaluaré cada motor según su bucle de juego y cómo funcionan internamente, mostraré una tabla comparativa de estos motores y partiendo de esta tabla explicaré mi elección de motor para el proyecto.

A través de la documentación que ofrecen los desarrolladores de los motores, podemos explicar sus bucles de juego, empezando por Unreal.

2.0.1 Unreal

Según el análisis del código que se encuentra en el github de Unreal y que para tener acceso hay que pedir permiso a Epic:

- Primero de todo se manda un latido a un hilo de diagnósticos. Después de esto actualiza el estado de los objetos que pueden ser renderizados para renderizarlos en el hilo de renderizado. Después se comprueba si el motor se encuentra en modo Idle (si no está haciendo nada), y duerme el tiempo necesario el hilo principal del motor.
- Tras volver del Idle, se comprueba los Inputs de un usuario. Se comprueba el tiempo delta, que es el tiempo que ha pasado desde el último ciclo de bucle y se espera el tiempo necesario para asegurar una frecuencia adecuada.
- Por último, se actualizan todos los objetos de la escena de forma especial. En Unreal, cada método de actualización de un actor se divide en grupos, antes de las físicas, en medio de las físicas y tras las físicas. El anterior a las físicas se ejecutará a la misma vez que la frecuencia de juego, el de las físicas a la misma vez que la frecuencia de las físicas, por lo que puede retrasarse y no estar asegurado su ejecución en cada pase del bucle y el último tras las físicas. Si el usuario ha indicado que algún objeto depende de otro las actualizaciones se ejecutaran uno tras otro. Por último, se renderiza el mundo y actualiza el tiempo delta. Todo esto viene en la página web de la API de Unreal[6]. Esta parte se encuentra explicada en la Ilustración 14

En resumen, las actualizaciones de los objetos del mundo y la simulación física se encuentran acopladas debido a los grupos de actualización de los Tick de Unreal y el rendering está desacoplado.

No se encontró un esquema claro de todo este proceso, debido a que Unreal no tiene tanta documentación. Además, como el código es libre y se puede consultar, los desarrolladores invitan a consultar el código para comprobar funcionamientos internos del motor, y para alguien que no sea muy hábil en C++, esto se puede convertir en una tarea complicada.

[6] Docs.unrealengine.com/en-us/Programming/UnrealArchitecture/Actors/Ticking

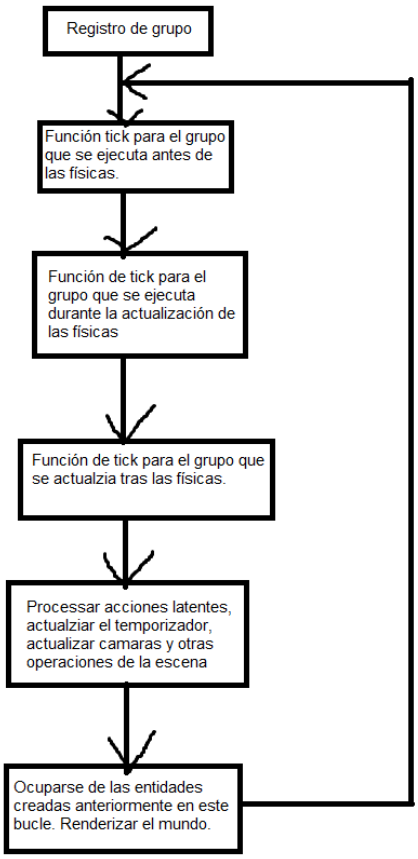

Ilustración 14 Ciclo de ejecución de los actores según su grupo de actualización.

2.0.2 Ogre3D

En Ogre3D, como se ha mencionado antes no hay un bucle de juego, debido a que solo es un motor gráfico, pero en el libro *OGRE 3D. Programming. Pro. Gregory Junker* y mostrado en la Ilustración 15, se da una aproximación al usuario para crear su propio bucle de renderizado si este no desea usar el que viene por defecto, que es donde se podrán actualizas la lógica de los objetos de la escena y se convertirá en un bucle de juego.

Listing 4-8. *Skeleton Example of a Manual Main Rendering Loop in Action*

```
bool keepRendering = true;

// Do all of the Ogre setup we've covered so far in this chapter: loading plug-ins,
// creating render window and scene manager and camera and viewport, and putting
// some stuff in our scene.

while (keepRendering)
{
   // process some network events into engine messages

   // process some input events into engine messages

   // update scene graph (manager) based on new messages

   // render the next frame based on the new scene manager state
   root->renderOneFrame();

   // check to see if we should stop rendering
   // Note: NextMessageInQueue() is completely fictional and used here only
   // for purposes of illustration -- it does not exist in Ogre.

   if (NextMessageInQueue() == QUIT)
   {
      keepRendering = false;
   }
}
```

Ilustración 15 Posible implementación del ciclo de vida del motor Ogre3D [4].

Este bucle que es el de la Ilustración 15 realiza los siguientes pasos:

- Primero procesa los eventos de la red convirtiéndolos en mensajes del motor.
- Después los eventos de Input que haya generado el usuario actualizan el grafo de escena.
- Al haber convertido los mensajes, se actualizan los objetos de la escena según los mensajes que nos lleguen.
- Al finalizar la actualización de los objetos, se renderiza un frame.
- Tras esto, si no hay mensajes que procesar, se deja de renderizar y se termina la aplicación.

2.0.3 Love2D

En Love2D, motor de juegos 2D escrito en Lua, tal y como refleja su nombre, tenemos:

- Primero se comprueba los eventos que han llegado y se ejecutan uno a uno, tales como las pulsaciones de teclas.
- Tras esto se avanza el tiempo de simulación del juego.
- Se actualiza el estado de todos los objetos.
- Se renderizan los gráficos del juego.

Este bucle es extremadamente simple y aparecen explicados en la Ilustración 16 e Ilustración 17, que aparecen explicados en la web de Love2D[7] . Se espera un tiempo en idle para dar tiempo de ejecución de tareas al sistema operativo.

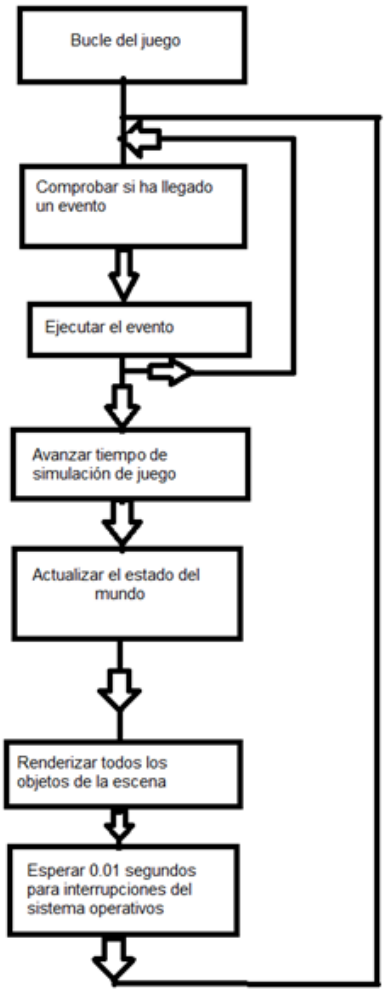

Ilustración 16 bucle de juego de Love2D

[7] https://love2d.org/wiki/love.run

```lua
function love.run()
	if love.load then love.load(love.arg.parseGameArguments(arg), arg) end

	-- We don't want the first frame's dt to include time taken by love.load.
	if love.timer then love.timer.step() end

	local dt = 0

	-- Main loop time.
	return function()
		-- Process events.
		if love.event then
			love.event.pump()
			for name, a,b,c,d,e,f in love.event.poll() do
				if name == "quit" then
					if not love.quit or not love.quit() then
						return a or 0
					end
				end
				love.handlers[name](a,b,c,d,e,f)
			end
		end

		-- Update dt, as we'll be passing it to update
		if love.timer then dt = love.timer.step() end

		-- Call update and draw
		if love.update then love.update(dt) end -- will pass 0 if love.timer is disabled

		if love.graphics and love.graphics.isActive() then
			love.graphics.origin()
			love.graphics.clear(love.graphics.getBackgroundColor())

			if love.draw then love.draw() end

			love.graphics.present()
		end

		if love.timer then love.timer.sleep(0.001) end
	end
end
```

Ilustración 17 El bucle de juego de Love2D[8]

[8] https://love2d.org/wiki/love.run

2.0.4 Unity

Por último, el objetivo de este trabajo, Unity. El bucle de juego de Unity aparece explicado en la documentación del fabricante, en una imagen que se muestra en la Ilustración 18

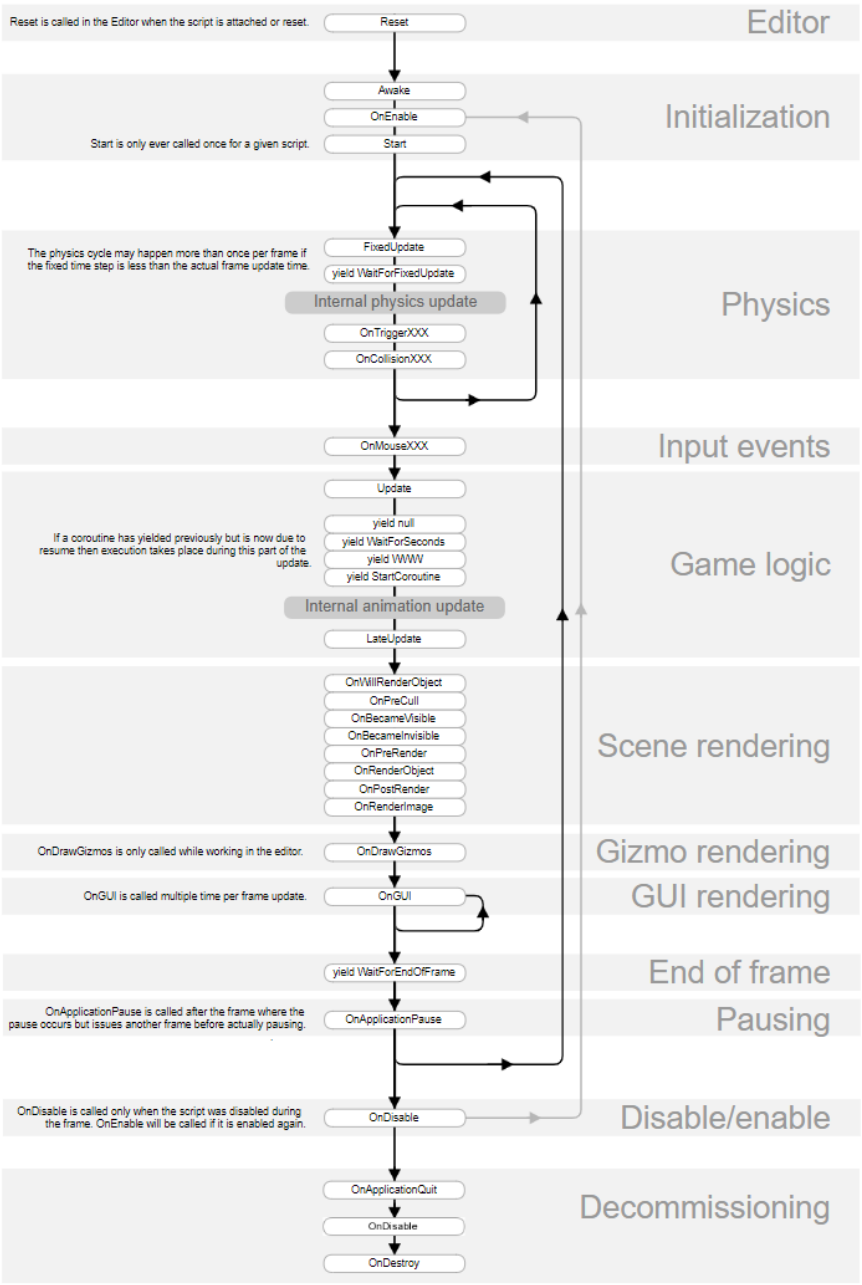

Ilustración 18 Ciclo de un script de Unity[9]

[9]Docs.unity3d.com/es/current/Manual/ExecutionOrder.html

El resumen de dicha ilustración es:

- Unity comienza llamando a todas las funciones Awake, OnEnable y Start que se encuentren dentro de un objeto en la escena.
- Tras esto, actualiza todos los eventos de físicas, actualizando colisiones y llamando a las funciones de los scripts que tengan que ver con estas.
- El siguiente paso es comprobar el input de un jugador y ejecutar la lógica del juego, actualizando el estado del mundo.
- Después se renderiza la escena, dibuja los gizmos(1) si se encuentra en el editor y están habilitados, dibuja la GUI especial de Unity
- Por último, al final del bucle, ejecuta cualquier comando que este en onApplicationPause si se pausa la aplicación, y repite el bucle hasta que se cierre la aplicación.

Algunos eventos se ejecutan más veces que otros, como por ejemplo cada evento del apartado de la lógica del juego se ejecutan en cada ejecución del bucle, mientras que las que figuran en las físicas tienen un tiempo fijo de ejecución que puede que no se cumpla y que puede ser establecido por el usuario en las opciones de Unity.

Para terminar este apartado, a continuación, aparece resumido en una tabla las diferencias entre estos motores.

Motores	Documentación	Assets extras	Facilidad de uso	Cantidad de motores	Customización	Bucle de juego	Precio
Ogre	Abundante	Escasos	Difícil de usar	Solo gráficos	Completa debido a que solo es un motor gráfico	Ha de crearse	Gratis
Love2D	Abundante	Escasos	Muy sencillo de usar	Gráficos 2D, sonido inputs de un jugador, físicas y multijugador	Se puede modificar todas las partes del motor con acceso libre al código fuente de este	Contiene uno, aunque puede modificarse	Gratis
Unreal	Escasa pero abundante ayuda del fabricante	Tiene un mercado con abundantes assets extra.	Si se pretende crear una aplicación compleja es difícil de usar, si no contiene programación visual.	Motor 3D con raytracing, físicas, input de jugador, multijugador, audio, físicas, IA	Acceso libre al código fuente del motor, aunque hay que pedir permiso para verlo en github.	Contiene uno y no puede modificarse. Es diferente según se use un thread o varios.	Porcentaje de las ganancias de un juego
Unity	Abundante	Tiene un mercado con abundantes assets extra.	Sencillo de usar	Motor 3D, físicas, input de jugador, multijugador, audio, físicas, IA	Modificaciones del motor son difíciles y escasas	Contiene uno y no puede modificarse.	Pago mensual o gratis si es una empresa pequeña

Tabla 2 Tabla comparativa entre motores de videojuegos

Como muestra la tabla, la razón por la que se ha escogido Unity, aparte de por ser el motor que mejor se usar, es por su sencillez frente a otros motores, su potencia con la cantidad de funciones que tiene (modificar físicas, IA, audio, online) y por su enorme aceptación en la industria de los videojuegos. Aunque modificar el motor sea más difícil que en el resto de los motores, considero que las demás ventajas pesan más a la hora de elegir Unity que las desventajas.

Y tal y como se ha mencionado, los motores incorporan bucles de juegos que pueden ser de distintos tipos, explicados a continuación:

2.1 Modelos de simulación en videojuegos

2.1.1 El modelo simple acoplado

Este es el tipo de bucle más simple que hay y el primero en ser creado en los videojuegos. En este modelo [5] las fases se ejecutan en un orden determinado y sin parar.

El ejemplo de Ogre3D de la Ilustración 15 es exactamente este modelo, en el que se ejecuta el bucle a la máxima velocidad que el procesador pueda, con variaciones en la frecuencia dependiendo de la carga del procesador.

Es fácil de implementar, debido a que solo se ejecuta la actualización de la lógica del juego y el renderizado de los gráficos, sin ningún cálculo de tiempo. Esto es, que, el número de veces que se ejecute el bucle de juego, o lo que es lo mismo, su frecuencia de actualización, dependerá de la velocidad del procesador y su carga.

Al no haber un tiempo de espera calculado entre cada ejecución del bucle, como se explicará después, podrá ocurrir que el bucle funcione demasiado rápido. Si, por ejemplo, se crease un reloj con este tipo de bucle, según la potencia del hardware en el que corriese la aplicación, el reloj funcionaría más o menos rápido. Lo cual no tiene sentido porqué un reloj siempre ha de funcionar igual.

Otro problema sería, al desincronizarse la frecuencia de actualización con la cantidad de veces que un televisor o monitor muestra la imagen en pantalla, término llamado hercio, se pueden observar comportamientos inesperados en la televisión.

Normalmente, la tarjeta gráfica de un dispositivo electrónico estará conectado al televisor. A parte de la frecuencia de actualización del bucle del juego y la de refresco de pantalla, también existe la frecuencia de actualización de renderizado. Esta frecuencia es la cantidad de veces que una tarjeta gráfica puede dibujar la información que se le transfiere en un segundo.

Si, tanto el bucle de juego como la actualización del dibujado de la tarjeta gráfica (rendering), es mayor que los hercios a los que funciona el televisor, puede ocurrir que en la pantalla no se lleguen a mostrar varias de las imágenes generadas por la gráfica. Aquí pueden ocurrir fenómenos definidos por el teorema de Shannon- Nyquist, que dice que el muestreo de una señal ha de ser al menos el doble de la mayor frecuencia de la señal [6]. Por ejemplo, como se enseña en esta página web[10], si el reloj que gira en sentido horario se muestra por pantalla a una frecuencia menor del que se mueve,

[10] https://jackschaedler.github.io/circles-sines-signals/sampling4.html

parece que gire en sentido contrario. En un videojuego puede ocurrir que un enemigo parezca teletransportarse.

Si es tan solo la frecuencia del bucle de juego el que funciona más rápido, las imágenes se mostraran correctamente, pero pueden ocurrir comportamientos extraños en la simulación, tal como que un personaje corra demasiado y acabe atravesando paredes debido a que la frecuencia en la que se consulta las colisiones no era suficientemente rápida. Este fenómeno se llama efecto túnel [7](6).

Sí la frecuencia de actualización del bucle de juego es menor que la frecuencia de actualización de la tarjeta gráfica y el monitor parecerá que el juego funcione muy lento, que las fuerzas sean menores provocando errores en la simulación física y cualquier velocidad que serán más lentas.

Algunos videojuegos antiguos que implementaban este sistema ya no funcionan como deberían, como, por ejemplo, la versión original del videojuego Grim Fandango, que tiene un puzle que obligaba a ralentizar la cpu en los ordenadores más nuevos si se tiene intención de completarse.

Esto es debido a que el puzle tenía que ver con pulsar una tecla cada cierto tiempo, pero el intervalo de tiempo iba demasiado rápido, por lo que los jugadores no podían pulsar la tecla lo suficientemente rápido.

En el libro *Game Engine Architecture, Third Edition* [1], al principio del capítulo *The Game Loop and Real-Time Simulation*, se puede observar este bucle aplicado a un juego de pong que se resumen con este esquema de la Ilustración 19.

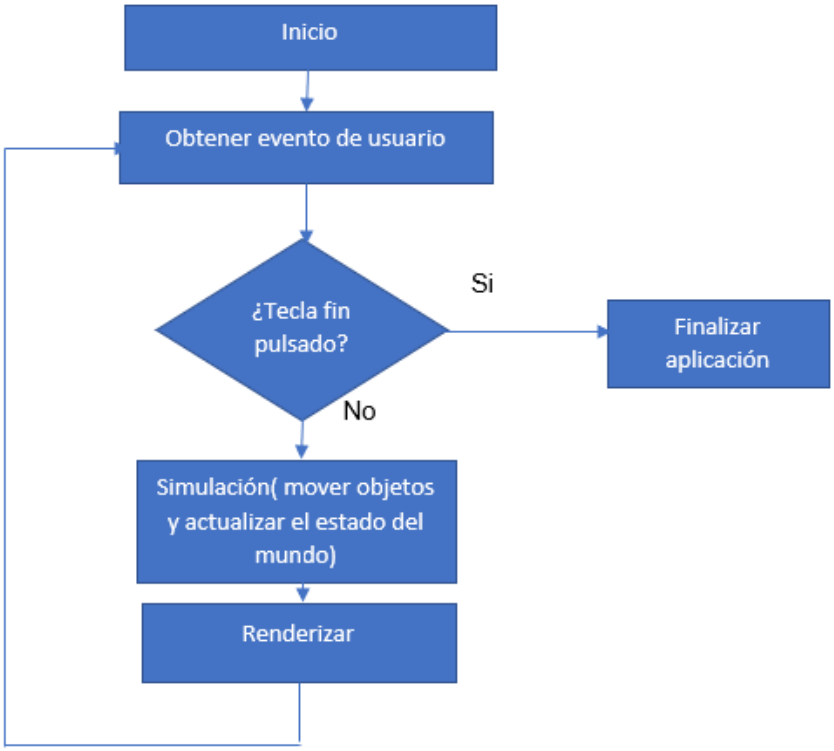

Ilustración 19 Esquema que describe el modelo simple acoplado

2.1.2 Modelo sincronizado acoplado

Tras esto surgió el modelo sincronizado acoplado [5], debido a la necesidad de asegurar una frecuencia de muestreo para que los juegos funcionasen de forma similar en diferentes maquinas.

Las fases se siguen ejecutando del mismo modo, pero ahora hay una gestión del tiempo, y se calcula el tiempo necesario para ejecutar un ciclo. El tiempo sobrante entre cada ciclo se reserva para tareas del sistema u otros programas. Esto es necesario para que cada programa no utilice al 100% del procesador. Si no se utilizase la reserva del tiempo para el sistema operativo, usar programas en multitarea sería imposible porque el procesador estaría ocupado ejecutando un solo programa.

Al insertar este tiempo, el bucle se convierte en un ciclo discreto. De este modo se soluciona los problemas que surgen al ejecutarlos en una plataforma demasiado rápida, como ocurría en el videojuego Grim Fandango. Debido a que ahora se tiene un tiempo entre bucles calculado, se puede enlazar todas las velocidades y fuerzas a ese tiempo, haciendo que los cálculos de fuerzas o los movimientos de los objetos se calculen en función de la frecuencia de ejecución del bucle principal. Así la plataforma que impedía al protagonista avanzar, tendrá siempre la misma velocidad en diferentes hardwares y no habrá necesidad de disminuir la velocidad del procesador. Las ventajas son:

- Ahora se puede limitar a una frecuencia fija para que funcione bien en cualquier dispositivo, sin pérdida de información por mostrar demasiadas imágenes en un segundo.
- Aun así, el problema que surge al estar acoplado la simulación y el renderizado en el caso de una frecuencia baja sigue existiendo.

En el ejemplo con el motor Love en la Ilustración 16 y la Ilustración 17, se observaba que el bucle se ejecutaba cada 0.001 segundos y se manejaba un tiempo. Ese era un ejemplo de este tipo de bucles. En el libro *Game Engine Architecture, Third Edition* [9], existe un ejemplo de esto en el apartado *Measuring Real Time with a High-Resolution Timer, en el que* se muestra un ejemplo de este tipo de bucles. En un esquema sería el de la Ilustración 20., en el que, como se observa, primero se obtiene un evento de usuario. Después si la aplicación no acaba, se realiza la simulación. Luego se renderiza

la simulación en pantalla y una variable llamada tiempo delta, se actualiza con el tiempo que ha pasado desde la actualización de la última invocación del bucle.

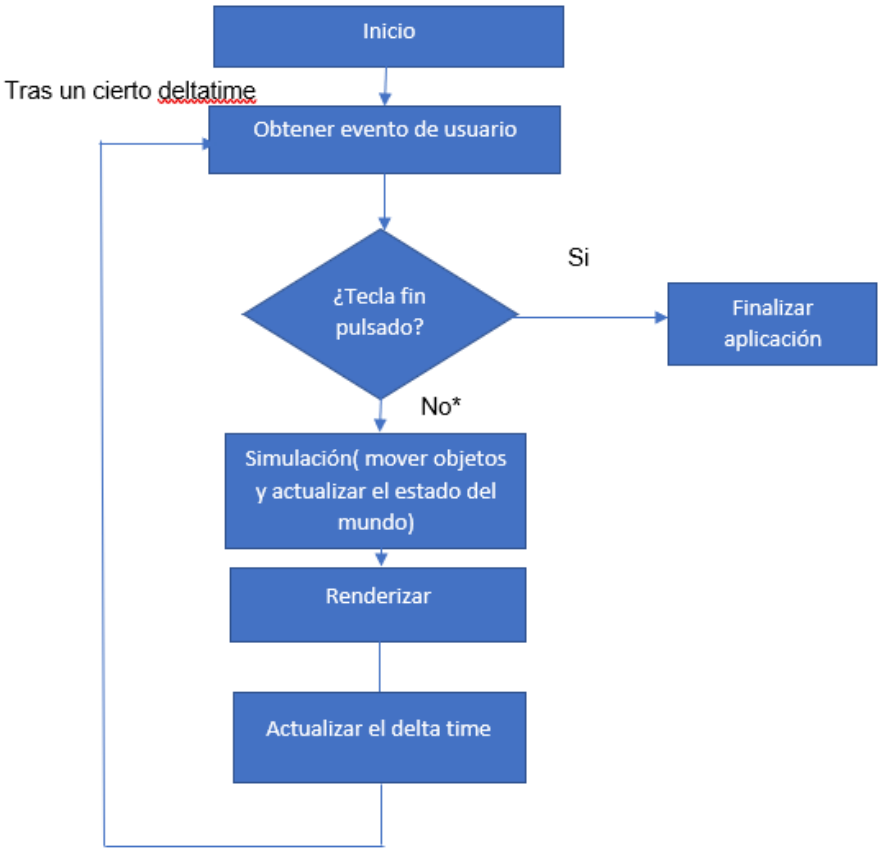

Ilustración 20 Esquema que explica el funcionamiento del modelo sincronizado acoplado

2.1.3 Modelo simple desacoplado

Por último, el modelo simple desacoplado [8]. En este modelo, distintas fases pueden ejecutarse en distintos momentos con una frecuencia de muestreo distinta, por lo que se puede conseguir mejor rendimiento que en los dos modelos anteriores, debido a que podemos repartir la carga al procesador de dos procesos en dos tiempos distintos, a diferencia de los anteriores donde dos procesos se cargan uno detrás de otro en un recorrido del bucle.

El modelo surgió con el objetivo de asegurar un comportamiento determinista para ciertos componentes del bucle en diferentes maquinas, tanto más potentes como más débiles.

El diagrama de este modelo consistiría en el que aparece a continuación, representado en la Ilustración 21, donde además de lo que ocurre en el modelo de la Ilustración 20,

antes de renderizar se comprueba si ha pasado cierto tiempo para ejecutar una función, y si es cierto, esta se ejecuta:

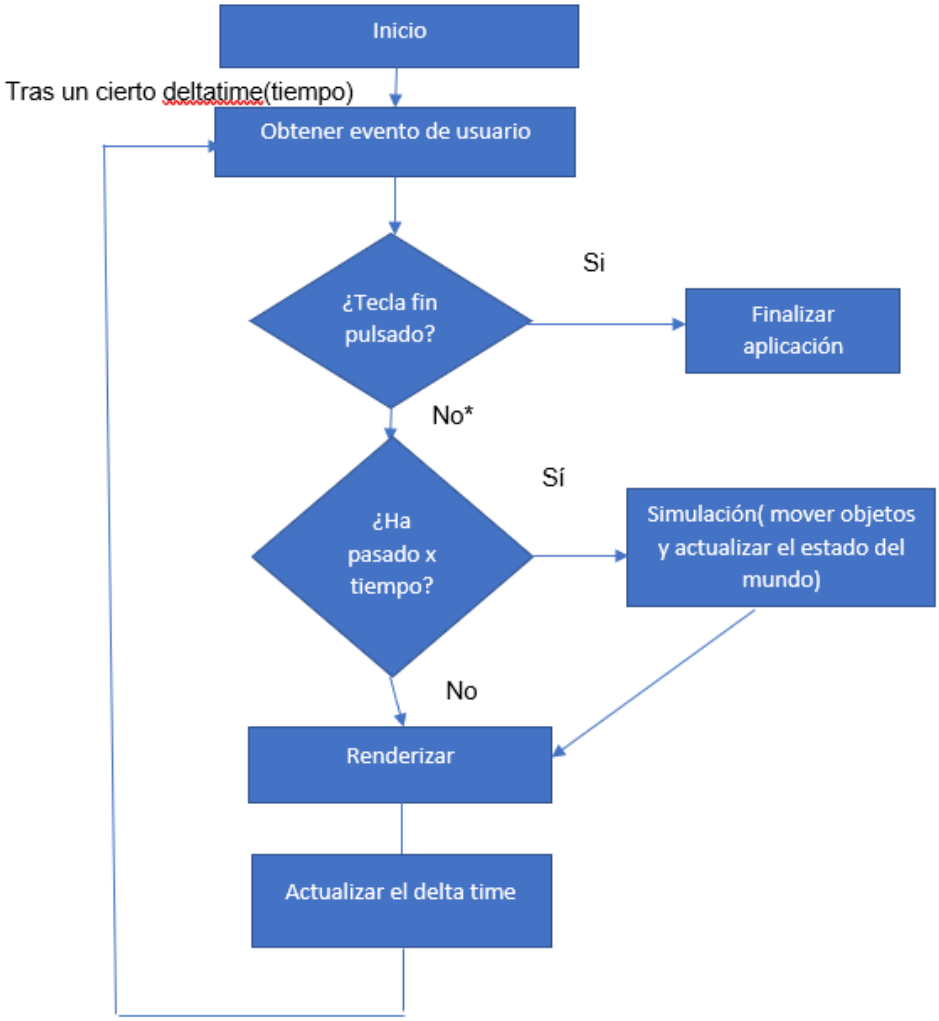

Ilustración 21 Esquema de funcionamiento del modelo simple desacoplado

2.1.4 Comparaciones

Obviamente el primer modelo se dejó de usar debido a los errores que pueden ocurrir en la simulación, lo cual lo hace el peor de todos. Como ya se ha analizado es más lento que el modelo simple desacoplado, y genera errores en la simulación, como se ha comentado con el videojuego Grim Fandango, debido a que no se puede atar las velocidades de las físicas a una frecuencia fija de actualización, ya que genera comportamientos erróneos.

Pero, el modelo sincronizado acoplado aún se usa debido a que limitar la ejecución del bucle a una frecuencia fija soluciona suficientes problemas. Específicamente el comentado con el reloj que gira de forma antihoraria y el problema generado en Grim

Fandango. En cuanto al resto de problemas sobre simulación no determinista en hardware con peores prestaciones se pueden solucionar con un delta time, que como se ha comentado anteriormente al principio del capítulo 2, es el tiempo transcurrido que se guarda entre cada actualización del bucle de juego, el tiempo que ha transcurrido entre cada frame, y realizar las actualizaciones del mundo en base a ese valor, tal como hace el motor Love y se enseña en la Ilustración 17.

Aun así, el modelo deseable sería el modelo simple desacoplado debido a que la posibilidad de fijar la frecuencia de actualización de cualquier fase del bucle principal aporta un comportamiento determinista a la fase.

Existen otros bucles como el que utiliza la Unreal Engine 4 en el que se paralelizan en hilos la simulación y el renderizado, pero, debido a que el proyecto se centra en Unity, y el más parecido a Unity es el modelo simple desacoplado, en el proyecto no se mencionan.

Para ser exactos, en Unity, como se puede apreciar en la Ilustración 18, el más parecido es el modelo simple desacoplado. Unity además tiene un thread independiente donde se procesa el sonido, pero el resto se parece al esquema de la Ilustración 21. Aun así, la fase de actualización y renderizado siguen estando acopladas ylas fases desacopladas son las físicas, el GUI (interfaz de Unity) y los yield(2) de la lógica del juego, ejecutándose con un tiempo diferente al resto.

Existe una particularidad más del bucle de Unity y es que este se ejecuta con eventos[11]. Un evento es un cambio de estado del sistema a raíz de una acción del usuario o del paso de una cantidad de tiempo. El sistema de eventos [9] funciona añadiendo un evento a una cola y luego siendo procesado por un handler. El handler se encarga de ejecutar en el tiempo correcto cada evento, permitiendo así en Unity ejecutar el fixedupdate con un tiempo determinado, aunque puede que no se cumpla. En forma de código, el motor de la Ilustración 17 muestra cómo se procesan los eventos en un motor y luego renderiza el mundo. Como muestra ese ejemplo, en cualquier sistema de eventos, dentro del bucle principal se comprueba si hay un evento en cola, se desencolará y se ejecuta cualquier evento.

Otra técnica que sirve para ahorrar tiempo en las físicas es la desactivación de las colisiones en los objetos que no se usan. Unity[12] hace uso de esta técnica y solo los reactiva si otro objeto colisiona con este, si otro rigidbody(3)unido a éste sufre una colisión, si se modifican sus propiedades o si se le aplica una fuerza. Esta técnica permite mejorar algo el rendimiento del bucle principal, hasta el punto de que viene incorporada una función que se recomienda usar en al comienzo de una escena de Unity que pone cada objeto a dormir para aumentar el rendimiento. Aun así, esta mejora no se notará si es necesario que todos los objetos deban estar despiertos.

Todas las diferencias de los bucles y las particularidades están explicadas en la siguiente tabla:

[11] https://docs.unity3d.com/Manual/EventFunctions.html

[12] http://ws.cis.sojo-u.ac.jp/~izumi/Unity_Documentation_jp/Documentation/Components/RigidbodySleeping.html

Modelo	Funcionamiento en diferentes tipos de hardware	Distintas frecuencias para distintas fases	Implementado en Unity	Ejecución a tiempo de los eventos
Simple acoplado	La simulación se acelera en hardwares más potentes que en los que se ha programado	No	No	No
Sincronizado acoplado	Funciona en todos los ordenadores iguales si se utiliza delta time	No	Fase de actualización y renderizado. La GUI se ejecuta más veces que el bucle principal y hace perder rendimiento al juego. Las físicas pueden ejecutarse en tiempos distintos al bucle principal	Siempre que el resto de las fases del bucle acoplados no carguen el sistema, este podrá ejecutarlos.
Simple desacoplado	Siempre es igual en todos los hardwares	Si	No	Se puede calcular con precisión cuanto tiempo costará ejecutar cada fase y así asegurar el tiempo entre ejecuciones de fases

Tabla 3 Comparación de los modelos

2.2 Crítica al estado del arte

Aunque el modelo sincronizado acoplado se haya mejorado con diversas técnicas, tal y como se ha explicado, aún se podría mejorar más. El uso de un modelo continuo conduce a ciertos errores que se explicarán durante este capítulo.

La primera critica es uno de los problemas inherentes del modelo continuo. Debido a que sus fases se ejecutan una tras otra de forma continua, pueden surgir problemas con el orden de los scripts. Dentro del editor de Unity[13], en la última versión, se permite modificar el orden de los scripts para que el programador intente remediar el problema, pero seguirá ocurriendo de un modo u otro. Por ejemplo, en un juego de lucha, jugando dos jugadores en el mismo dispositivo, si fueran a realizar el mismo ataque con los mismos personajes justo en el mismo instante, el ataque que afectaría sería el del

[13] https://docs.unity3d.com/Manual/class-MonoManager.html

personaje cuyo script estuviese primero en el orden de la escena. El orden exacto de ejecución será que primero se procesará la interacción del jugador 1, luego la del jugador 2, entonces se procesarán los Updates del personaje perteneciente al jugador 1, que atacará, luego los Updates del personaje perteneciente al jugador 2 que atacará, pero cuando se procesen los ataques que se reciben, el jugador 1 recibirá antes el ataque y cancelará su ataque, por lo que el jugador 2 no recibirá daño. Este comportamiento se puede observar en la Ilustración 22

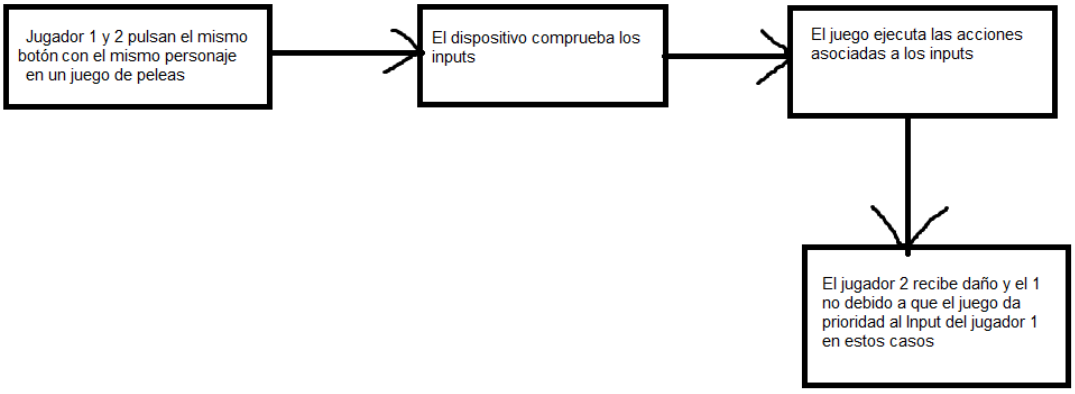

Ilustración 22 Esquema de funcionamiento del caso de un juego de peleas

También puede ocurrir otra variación que comienza con que un jugador 1 realiza un ataque y un jugador 2 lo esquiva en el mismo momento. Si el tiempo de las animaciones y las acciones realizaran el mismo efecto, el jugador 1 tendría prioridad debido a que el evento realizado por el jugador 1 se procesa antes. El comportamiento normal de esta acción debería ser que el esquive debería cancelar el evento de ataque. Es otro efecto negativo del bucle continuo que aparece descrito en la Ilustración 23.

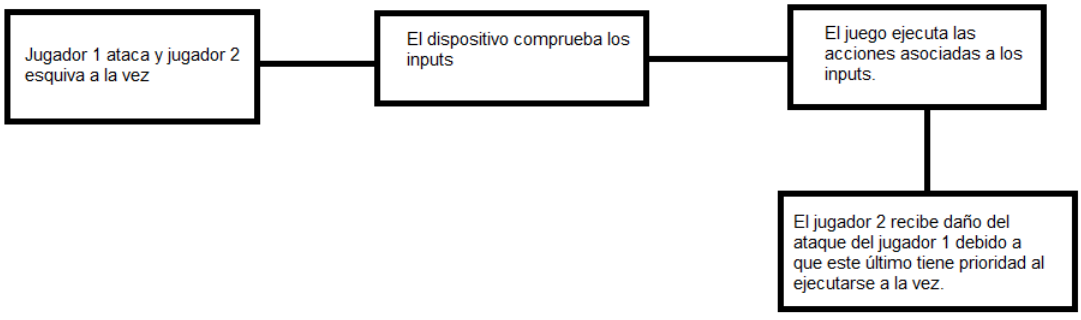

Ilustración 23 Esquema de funcionamiento del caso de un juego de peleas en el que un jugador ataca a la vez que otro esquiva.

Otro problema que aparece en el bucle de Unity es la simulación de físicas. Con los parámetros que vienen por defecto, si el objeto va demasiado deprisa, este puede atravesar paredes, tal y como se ha mencionado en el apartado anterior. En la página web de Unity aconsejan formas para atajar este problema, pero con ciertas condiciones[14]:

- Una consiste en cambiar el modo de colisiones de los objetos de discreto a continuo. En discreto los objetos se mueven en tiempos discretos de simulación, por ejemplo, el instante 0, 0.5, 1. En continuo la comprobación de las colisiones se ejecuta en todos los pasos del movimiento de un cuerpo físico. Según la guía esto puede hacer que no ocurran colisiones exactas.
- El otro consejo es aumentar la frecuencia de llamada de FixedUpdate en ajustes de Unity. Esto implica también una reducción del rendimiento del juego en función a los cálculos de FixedUpdatesi estos son muy costosos.

Además, en estudios realizados previamente [9] , donde se pretendía usar Unity como simulador de un ascensor, se vio que era suficientemente bueno, pero tenía problemas al aumentar o reducir el tiempo de simulación, ya que las físicas para cada velocidad de la simulación eran diferentes lo cual creaba simulaciones completamente distintas.

Un problema que FixedUpdate podría crear si se ejecuta demasiadas veces, es que puede generar problemas de visualización descritos por el teorema de muestreo de Nyquist-Shannon, el cual dice que, para representar correctamente una señal, su frecuencia de muestreo debe ser el doble que la original, si no se produce aliasing en la señal.

Por lo que, si se hace girar una rueda a una velocidad de rotación muy alta, pero la frecuencia de muestreo es muy baja (frecuencia de llamadas al método virtual FixedUpdate por unidad de tiempo), puede que la rueda gire en el sentido contrario, o que las animaciones que se reproduzcan no sean fieles a la realidad. Para esto Unity está preparado y se puede activar una opción llamada interpolación en todos los rigidbody que lo necesiten, que impide la aparición de submuestreo

Una excepción de este comportamiento es todo aquel movimiento no manejado por Unity, como el que podría crear un usuario dentro de una función personalizada o en Update, debido que, al ejecutarse el mismo número de veces que el renderizado de frames, el objeto se moverá más o menos rápido según la potencia de la máquina y el número de frames por segundo que se rendericen Para solucionar este problema, Unity aporta el tiempo que ha ocurrido entre frames y puede usarse para variar los movimientos dependiendo de este tiempo, denominado **delta time**. Además, este procedimiento es más deseable que el ejecutar cualquier movimiento que no dependa de físicas en FixedUpdate debido a que usar la función FixedUpdate podría sobrecargar las físicas que pueden resultar imprecisas.

Por último, la función Update de Unity, puede que se ejecute sin mucha optimización con el objetivo de proteger al usuario final. En este blog[15] realizaron un experimento para remplazar el método de llamada a los Updates y consiguieron una mejora notable de 5 milisegundos en una función Update donde solo se incrementaba una variable.

[14] https://docs.unity3d.com/Manual/class-Rigidbody.html
[15] https://blogs.unity3d.com/es/2015/12/23/1k-update-calls/

2.3 Propuesta de mejora

La propuesta de mejora de este proyecto es la implementación de RTDesk para ejecutar las funciones del bucle principal que sean posibles, usando el simulador de eventos discretos para llamarlas y evitar que sean llamadas desde Unity.

2.3.1 Evolución histórica de RTDesk

RTDesk fue una evolución de GDesk [10]que a su vez fue una evolución de Desk. Desk es un kernel de simulador de eventos en C++ realizado durante una tesis de máster [11]. Más tarde GDesk nació de una tesis doctoral y, tras unas evoluciones, nació RTDesk.

GDesk contenía:

- Un gestor de tiempos, el cual se actualiza si el tiempo real ha alcanzado el tiempo del evento a ejecutar. El gestor de tiempos es una variable con el tiempo que se ejecutó el último evento. Cuando en el bucle de juego se comprueba si un evento ha de ejecutarse comprobando su tiempo con el tiempo real, el gestor de tiempos se actualiza con el tiempo del evento actual. Y así es como avanza el tiempo de simulación.
- Una definición de entidades. Las entidades en GDesk son los objetos que se intercambian mensajes.
- La definición de los mensajes. Los mensajes contienen la función de la entidad que hay que ejecutar.
- La finalización de la simulación a través de un evento.

RTDesk añadió un temporizador interno de alta resolución, un monitor del núcleo y optimizaciones y adiciones de funciones no existentes previamente. RTDesk es tan solo un gestor de mensajes y no ejecuta ningún código de por sí. En el apéndice B, después del apéndice de sobre las nomenclaturas particulares del proyecto, existe una pequeña guía sobre cómo funciona la API.

2.3.2 Descripción de las ventajas de RTDesk

Como se ha introducido, RTDesk es un simulador de eventos discretos. Un simulador de eventos discretos consiste en, a diferencia de la manera de programar continua que pregunta constantemente al sistema por los cambios en el estado de un evento, que los cambios en el estado activan el evento, por lo cual se requiere de menor capacidad de procesamiento frente al modelo continuo.

En este tipo de simulación el paso del tiempo debe ser constante. RTDesk funciona en tiempo de simulación, que es sincronizado constantemente con el tiempo real, por lo que el paso del tiempo es el mismo que en la vida real, pero en este tipo de simulaciones se podría usar cualquier velocidad de tiempo

El esquema de funcionamiento es el siguiente, que corresponde a la Ilustración 24:

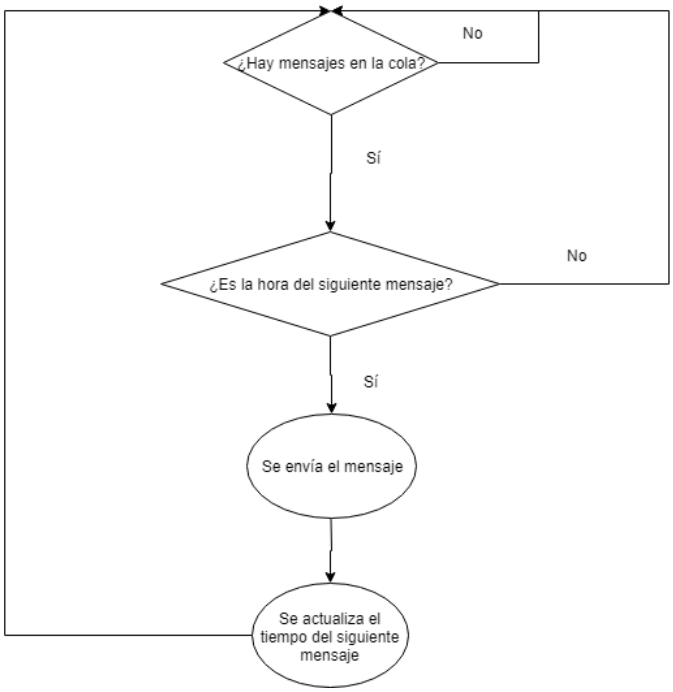

Ilustración 24 Esquema de la propuesta de mejora, en este caso un simulador de eventos discretos desacoplado

El bucle consiste en la comprobación de si hay eventos (llamados mensajes) en la cola. Si el tiempo de simulación en el que han de ser ejecutados es superado por el tiempo real, entonces, se ejecutan. Este bucle puede ir en un hilo diferente a la renderización o incluso el bucle puede implementar la renderización como si fuera un mensaje más del sistema, y todo seguiría desacoplado, ya que el objetivo de este bucle es ejecutar los mensajes en tiempos distintos. El tiempo contra el que se comprueban los mensajes avanza cada vez que se produce un evento.

En cuanto al funcionamiento de añadir los mensajes a la cola, cada entidad o actor, es decir, cualquier objeto fuera de RTDesk, puede generar un evento y enviarlo a una cola con el tiempo en el que debe ejecutarse.

El tiempo en el que debe enviarse se comprobará contra el tiempo actual y se ejecutará cuando el del mensaje sea menor o igual al actual.

Respecto a sí este modelo puede solucionar los problemas de Unity mencionados anteriormente:

- El orden de ejecución de los eventos se basa en el tiempo determinado por el evento como se ha explicado antes. Por lo que, si se da el caso de los dos jugadores golpeándose a la vez que se daba como ejemplo, debido a que los dos eventos se ejecutarán en el mismo ciclo de bucle, efectivamente ambos jugadores se dañarán, por lo que la simulación será más fiable.
- En cuanto al segundo ejemplo del ataque y del esquive, debido a que seguirá existiendo un orden y los dos eventos se auto cancelan, el orden seguirá importando por lo que pueden seguir ocurriendo errores. Por lo que esto seguirá a merced del programador que deberá decidir de alguna forma lo que ocurre. La forma de calcular las físicas es fácilmente solucionable con el nuevo paradigma. Si, por ejemplo, en un juego de disparos, una bala sale con una fuerza, se puede

calcular matemáticamente la trayectoria y ver cuando colisionará con un objeto, tiempo en el que se activará el mensaje. Esto es una mejora considerable frente al ciclo continuo donde hay que calcular todo el rato si el objeto choca con algo y si puede generar el efecto túnel. Aun así, esto solo funcionaría con objetos que no se mueven. En el caso de los objetos que se mueven habría que calcular cada cierto tiempo de un mensaje la posición en la que se encuentra para saber si colisiona en la posición calculada. La ventaja de este método es quese elimina el coste de los cálculos en cada ciclo de los objetos que no se mueven. Para el resto de los objetos, se puede modificar la frecuencia en la que cada objeto comprueba la colisión con la bala, en función de su cercanía a la bala. Este método ahorraría cálculos ya que no habría que comprobar la colisión todo el rato. Sería una leve mejora frente al ciclo de Unity ya que Unity utiliza las físicas desacopladas con un tiempo fijado por el usuario, aunque no pueden garantizar que se ejecuten siempre y todos los objetos tienen la misma frecuencia.

- RTDesk [12] permitirá a Unity ejecutar los eventos de forma desacoplada y asegurar una frecuencia de muestreo exacta para cada evento implementado. Además de abrir el sistema de mensajería de RTDesk a los usuarios finales para que estos puedan crear sus propios mensajes y eventos con un reloj en tiempo real fiable. Este sistema mejorará al de Unity debido a que no hay ningún sistema específico en Unity que permita enviar a un usuario mensajes en un tiempo preciso determinado.
- Y, por último, el caso de la cantidad de veces llamadas a Update se puede controlar perfectamente y variar en función del rendimiento de la aplicación, por lo que es superior al modelo tradicional simple acoplado.

Para implementar el simulador de eventos discretos, se realizará una adaptación de RTDesk en Unity para ejecutar las funciones del bucle principal que sean posibles, usando el simulador de eventos discretos para llamarlas y evitar que sean llamadas desde Unity. Como ya se ha mencionado en los pasos anteriores, RTDesk añadirá ventajas a Unity debido a la simulación discreta frente a la continua de Unity y permitirá realizar una simulación más fiable debido al reloj en tiempo real.

En cuanto los detalles del funcionamiento de la librería RTDesk, los eventos se transfieren entre entidades utilizando un sistema de mensajes. Estos mensajes vienen asociados a un tiempo, y una cola de mensajes los ordena en función de cuando deben ser ejecutados de menor tiempo a mayor.

3 Análisis del problema

En este apartado se realizará un análisis de la problemática de implementar un simulador de eventos discretos para Unity.

Al usar RTDesk como simulador, del cual se realiza un análisis en el anexo 2, pueden aparecer estos problemas.

- No es posible acceder al código fuente de Unity y, por lo tanto, no se puede modificar directamente ninguna función.
- El simulador RTDesk está escrito en C++, y el lenguaje disponible en Unity es C#, por lo que habrá que lograr un modo de adaptarlo.

- Una vez insertado el simulador, cómo reaccionará con las demás clases de Unity que no pudieron ser implementadas.
- Debe mejorar las simulaciones que se pueden realizar por defecto en Unity y también incrementar el rendimiento de un juego, por lo que el código debe de estar bien adaptado.
- Por último, se deben aportar razones para usar este simulador mejorando las capacidades de Unity y haciendo que su uso sea todo lo sencillo posible.

Existen diversas formas para implementar una DLL de C++ en C#. Por lo tanto, en este análisis de cuál es la mejor forma para implementar RTDesk, existen varias soluciones.

3.1 Formas de implementar código de C++ en Unity.

La 1º consistiría en el uso de CLI(9). CLI es un tipo de lenguaje para manejar plugins de C++ creado por Microsoft. De esta forma se pueden crear plugins para Net.

La otra forma es usando externs, función de C para exponer clases de C++ a otros lenguajes. Esta solución tiene un problema y es que las clases virtuales no están creadas aun, y no se pueden añadir en un extern.

Por lo tanto, los dos UML del comunicador con RTDesk serian:

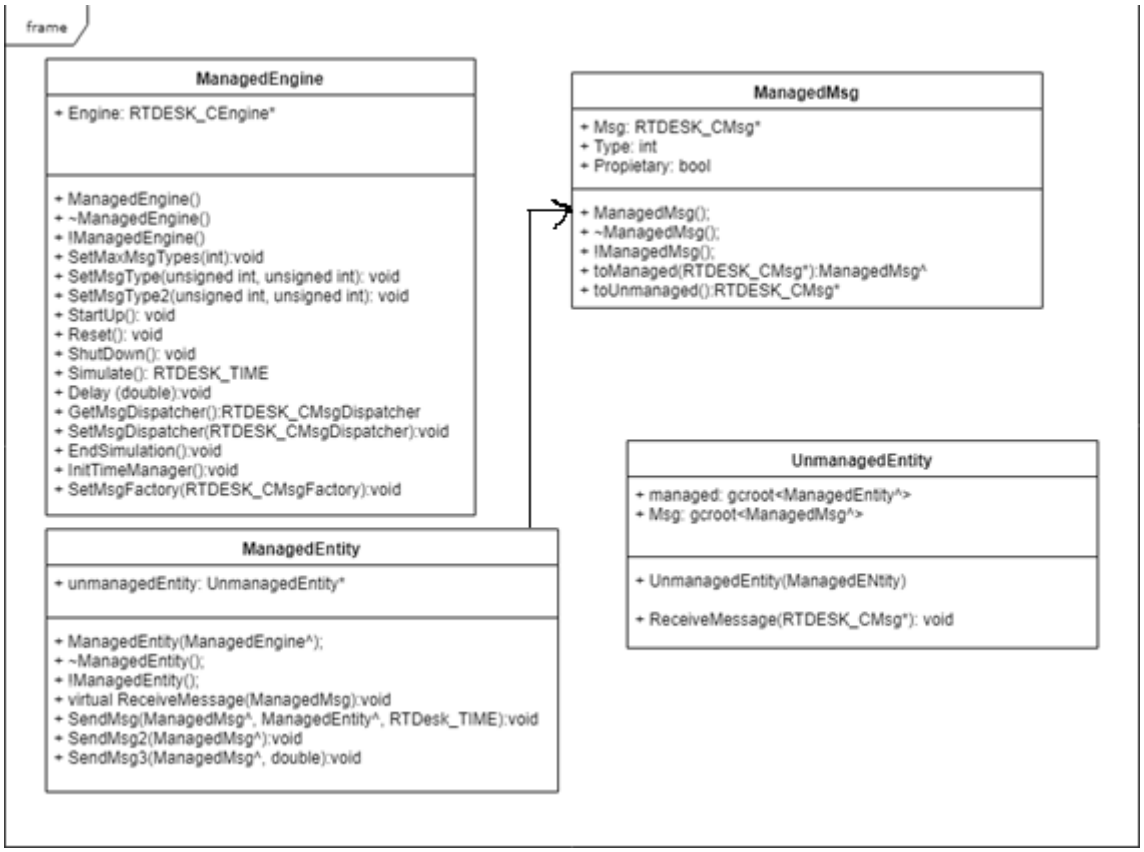

Ilustración 25 Implementación del código escrito en CLI

En la Ilustración 25 se observa que habrá cuatro clases ManagedEngine, ManagedEntity, ManagedMsg y UnmanagedEntity:

- **ManagedEngine** contiene el código que maneja en memoria el motor de RTDesk. También contiene el bucle de juego y todas las funciones necesarias para iniciarlo, pararlo, reiniciarlo, iniciar y parar la simulación y crear los tipos de mensajes necesarios.
- **ManagedEntity y UnmanagedEntity**: ManagedEntity contendrá un puntero a UnmanagedEntity, debido a que UnmanagedEntity seguirá consultando objetos de C++ por ser una clase abstracta. ManagedEntity también contendrá, a parte de la función virtual para recibir mensajes, todas las funciones para mandar los mensajes, tanto en un tiempo determinado, como la función para enviar un mensaje inmediatamente.
- **ManagedMsg** será la clase que contendrá las definiciones de los mensajes. También contendrá dos funciones para pasar un objeto mensaje creado de puntero a C++ al objeto mensajes de C# y viceversa.

Y la parte de Unity:

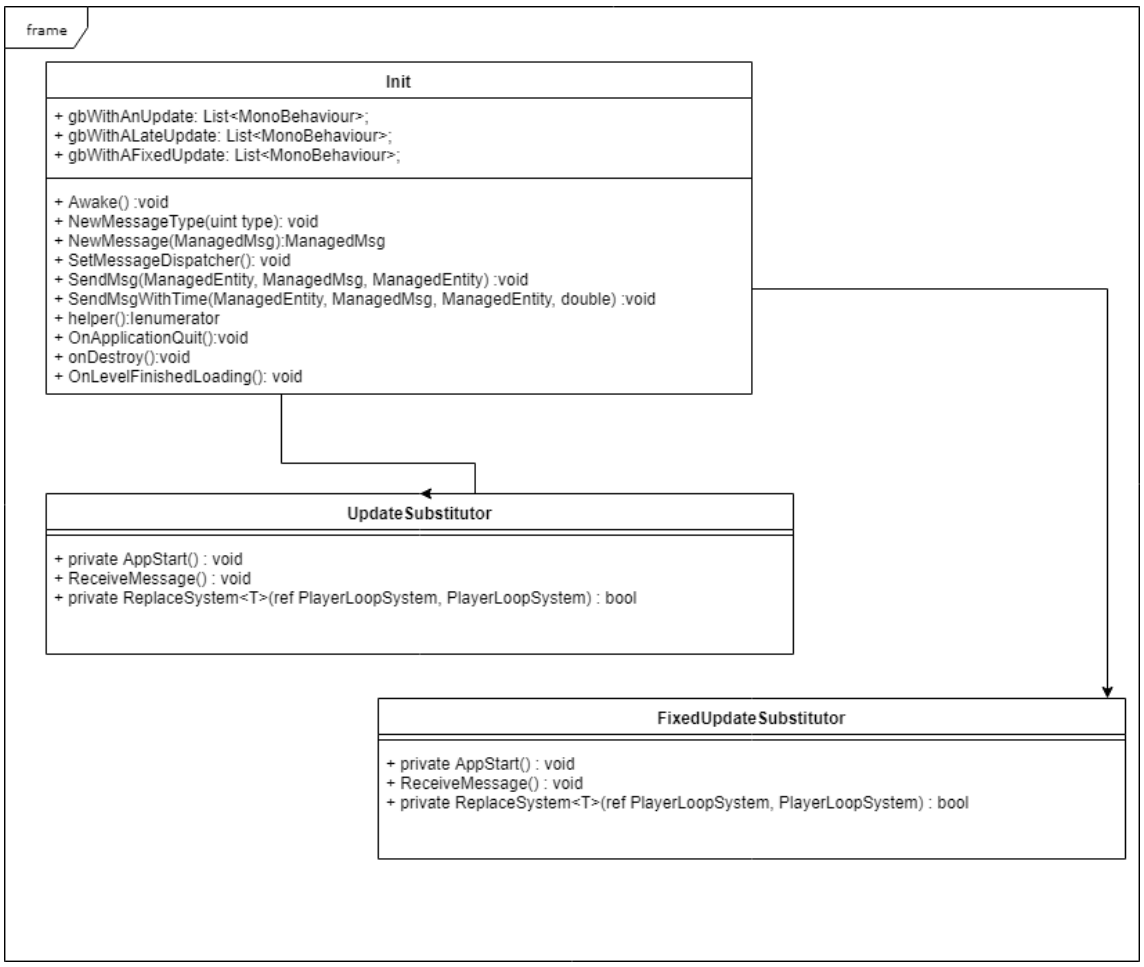

Ilustración 26 Los scripts de Unity que se comunican con la dll

En la Ilustración 26 vemos las clases de C# que forman parte de Unity. Este es más sencillo de explicar:

- **FixedUpdateSubstitutor y UpdateSubstitutor**: Estas dos clases solo contendrán una función que ejecutará las funciones originales definidas en el capítulo 1.2.1 Unity, FixedUpdate y Update. Cuando les llegue un mensaje de que deben ser ejecutadas, estas serán ejecutadas y mandarán otro mensaje con el tiempo siguiente en el que deberán ejecutarse otra vez.
- **Init**: Este script será el que una entidad de Unity posea. Así sus funciones serán llamadas por el bucle principal de Unity. Esta clase contendrá una Lista con las funciones Update de todos los scripts, que FixedUpdateSubstitutor y UpdateSubstitutor ejecutarán, la funciones para inicializar objetos de estas dos clases y la función para crear mensajes y entidades.

Mientras, con el segundo método, surgiría este diseño de la Ilustración 27:

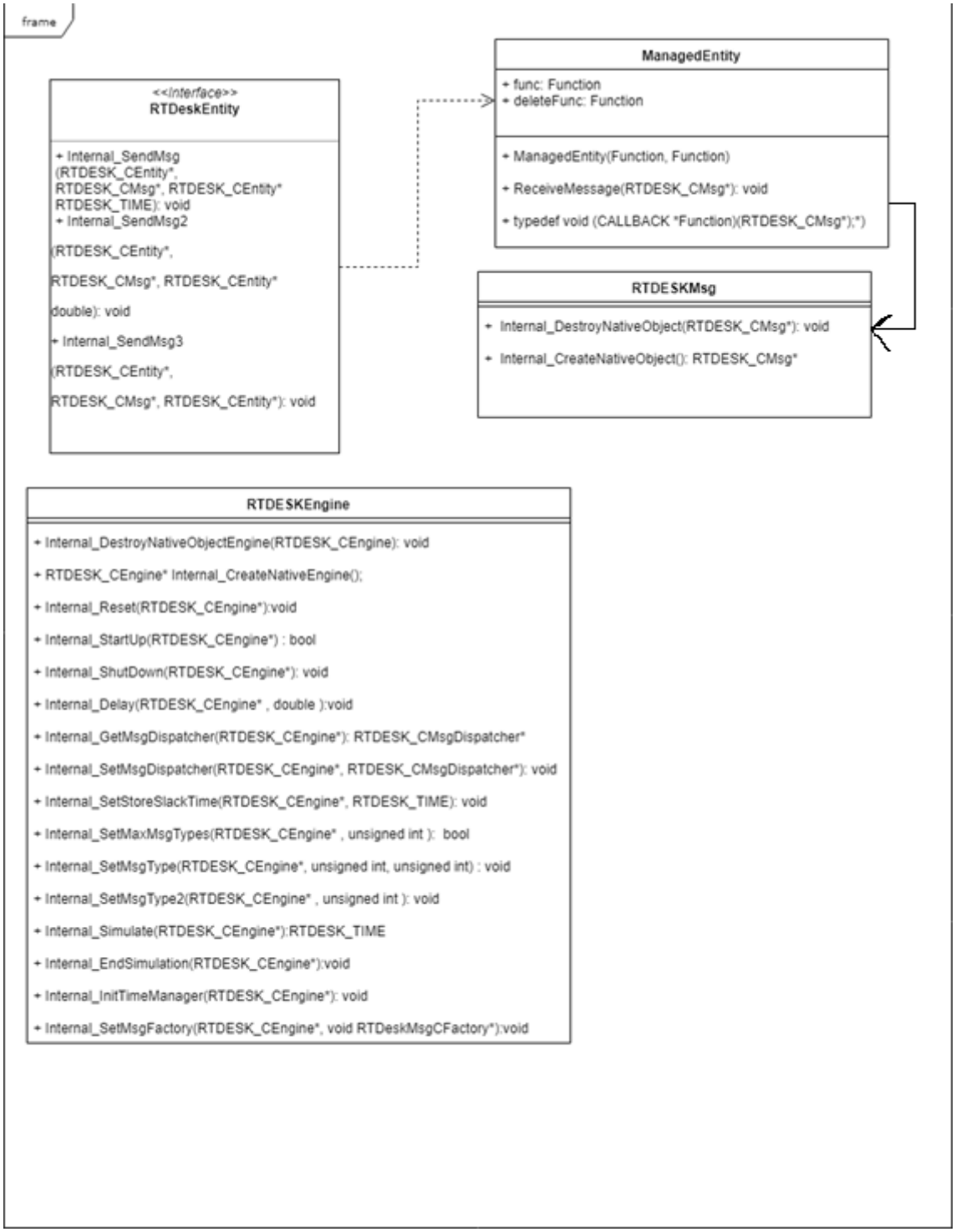

Ilustración 27 Cambios añadidos a la DLL usando el segundo método

En esta ilustración se modifican algunas clases del código original de RTDesk, para añadir todas las funciones Internal posibles. Estas serán las funciones que estarán dentro de un bloque extern en C++ para que C# pueda leerlas. Las tres clases mostradas harán los siguiente:

- **RTDeskEntity**: Esta es la clase original de entidades de RTDesk. Se expondrán las tres funciones que sirven para enviar mensajes. Dos de ellas en un tiempo determinado y otra para enviar el mensaje inmediatamente.
- **ManagedEntity**: Esta clase hereda de la clase abstracta RTDeskEntity. En C# se crearán objetos de esta clase para que las entidades en C# contengan la función de recibir mensajes, que ha de quedarse en C++. Además, contiene un objeto que referencia a una función. Cuando en C# se cree un objeto de este tipo, se le pasará una función al constructor, la cual se almacenará en C++. Luego, se llamará en la clase ReceiveMessage de ManagedEntity a esta función almacenada y entonces será ejecutada en C#, dentro de Unity.
- **RTDeskMsg**: Solo contiene dos funciones expuestas, una para destruir un objeto de tipo mensaje y otra para crearlos.
- **RTDeskEngine**: A parte del constructor y destructor del motor, al igual que la clase RTDeskMsg, esta clase expone las funciones necesarias para iniciar el motor, pararlo, reiniciarlo, iniciar la simulación y pararla y crear los tipos de mensajes necesarios.

Y su parte de Unity:

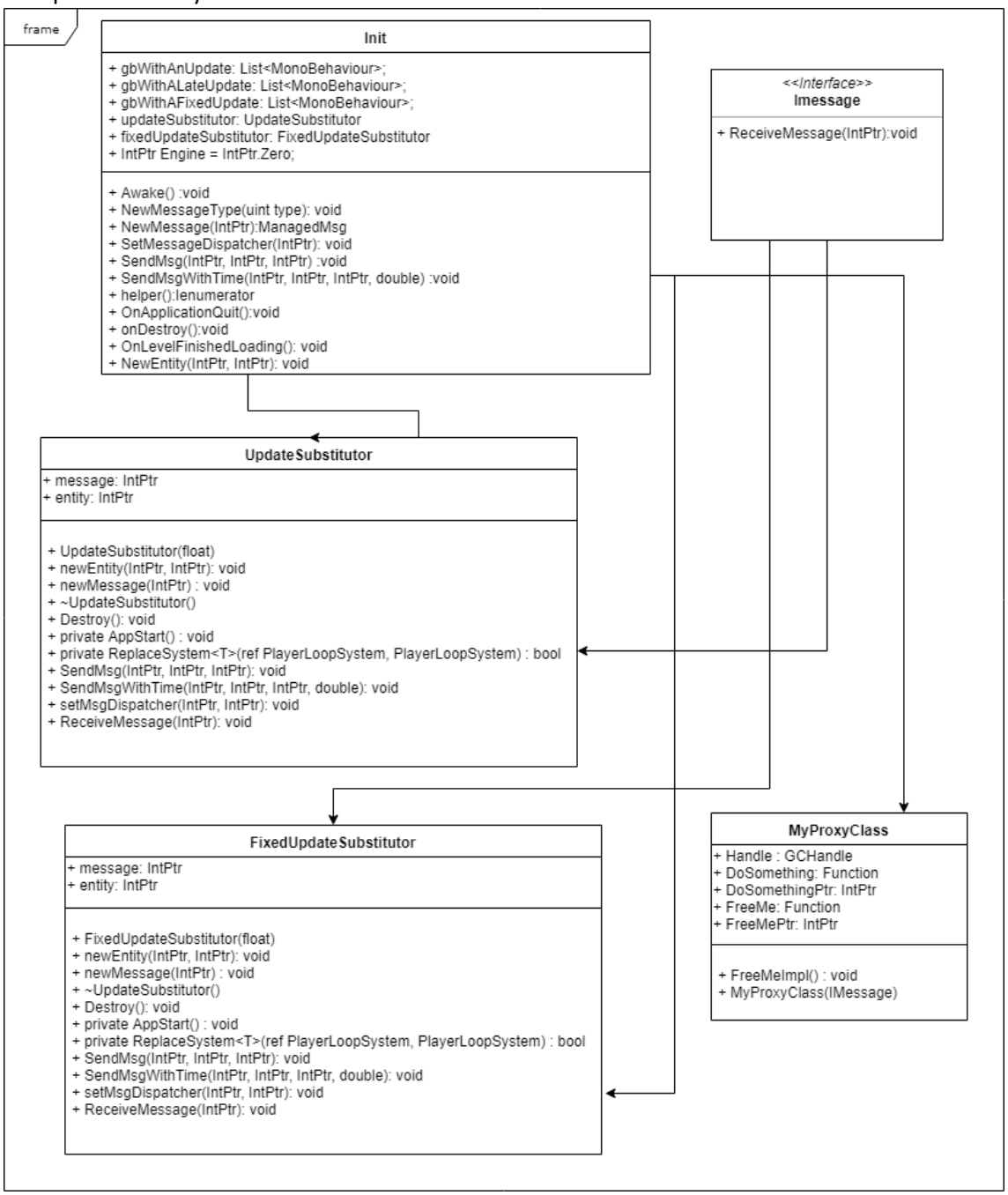

Ilustración 28 Comunicación de Unity con la dll

En la Ilustración 28 se observa todo lo que ha de crearse que forma parte de Unity:

- **FixedUpdateSubstitutor y UpdateSubstitutor:** Funcionarán igual que los de la Ilustración 26. Con el añadido de que hay que crear las clases que se comunicarán con el lado de C++ que crearán los mensajes y entidades y le pasará la función ReceiveMessage a C++, para que se ejecute como se ha explicado. Además de que tendrá como objetos punteros a los objetos de C++ creados de las clases ManagedEntity y RTDeskMsg.
- Init: También hará los mismo que en la Ilustración 26, con la diferencia de tener punteros a los objetos C++ de RTDeskEngine.

- **Imessage**: Esta interfaz servirá únicamente para que los usuarios que hereden de ella sepan que han de implementar la función ReceiveMessage, para que esta pueda ser ejecutada al recibir un mensaje.
- **MyProxyClass**: Esta clase sirve para que los punteros que almacenan las clases anteriores al lado de C++ no sean borrados por el colector de basura de C#. Para ello, todas las funciones de C# que han de pasarse a C++ han de ser almacenadas en una clase de este tipo, y al almacenarlas aquí, el colector de basura nunca las tocará.

Si uno de estos modelos de UML se siguen se espera que la implementación de RTDesk sea un éxito. Como se ha presentado en el análisis, existen dos posibles soluciones para implementar RTDesk. Cada una tiene sus pros y sus contras.

La solución de los UML de la Ilustración 25 y la Ilustración 26 tiene las siguientes ventajas:

- El colector de basura funciona correctamente, gracias a gcroot(12).
- Hay menos código por duplicado.
- La interfaz se implementa de forma más directa.

Por contrario las desventajas:

- Unity requiere circunstancias muy específicas para usar CLI, ya que no es del todo compatible.

Mientras que en la implementación de la Ilustración 27 y la Ilustración 28, sus ventajas:

- Más compatible con Unity, debido a que, como se ha definido anteriormente, Unity tiene su propia definición del lenguaje C#.

Las desventajas:

- Hay que manejar personalmente el colector de basura para los punteros que se pasan a C++ (para eso está la clase myproxyclass).
- No se puede trasladar la interfaz a C#, por lo que se tiene que buscar otra manera (se crea una clase que implementa la interfaz y cada vez que se necesite usar la interfaz se crea una instancia de la clase de C++).
- Para llamar a las clases de C++ hay que encapsular las clases en unos extern C, comando que, lo único que hace, es llamar a las clases de C++, por lo que se genera mucho código por duplicado.

Repasados los problemas, el mayor problema es la compatibilidad de Unity con CLI. Cuando se diseñó la forma de implementarlo se encontró que el compilador ha de estar programado en Net 2.5, el cual solo se encuentra disponible en visual studio 2008 y ha de tener el parámetro CLI:Safe al compilar, comando del compilador de visual studio que obliga a generar errores por cada componente no verificable. La lista de componentes no verificables se encuentra en la página de Microsoft[16]. Esta orden no era compatible con gran parte del código de RTDesk. Este problema era muy grande como para ignorarlo porque suponía usar tecnología no disponible y reescribir gran parte del código de RTDesk. Así que se decidió usar la estructura del segundo UML.

En cuanto a las dificultades encontradas realizando el análisis DAFO, que son:

[16] https://docs.microsoft.com/en-us/previous-versions/ykbbt679(v=vs.140)

- La aparición de los threads (**¡Error! No se encuentra el origen de la referencia.**) en Unity con el sistema de Jobs (es un thread en Unity, pero con el api de monobehaviour limitada para no cruzar threads) y el modo de programar ECS (entity component system) (10) pueden dejar obsoleto la futura API.
- La complejidad que puede llegar a aportar tener que implementar la API en un juego.

La primera, técnicamente no es un problema que surgiría realizando este trabajo de final de máster, pero dejaría obsoleto el trabajo realizado, y debido a que merece la pena tener un simulador de eventos discretos desacoplados, estaría bien actualizar la API al sistema de threads cuando sea lanzado oficialmente.

Y en cuanto a la complejidad, se intentará que el usuario final que utilice la API tenga que realizar el mínimo trabajo posible al implementarla en su juego. Por supuesto, para implementar mensajes personalizados el usuario deberá programarlos, pero para la ejecución de Update, FixedUpdate y LateUpdate se intentará que RTDesk los ejecute sin que el usuario deba hacer nada.

3.2 Análisis DAFO del proyecto

En cuanto al uso de RTDesk frente al bucle por defecto de Unity, un análisis DAFO puede ayudar a comprender él porque será mejor usar RTDesk.

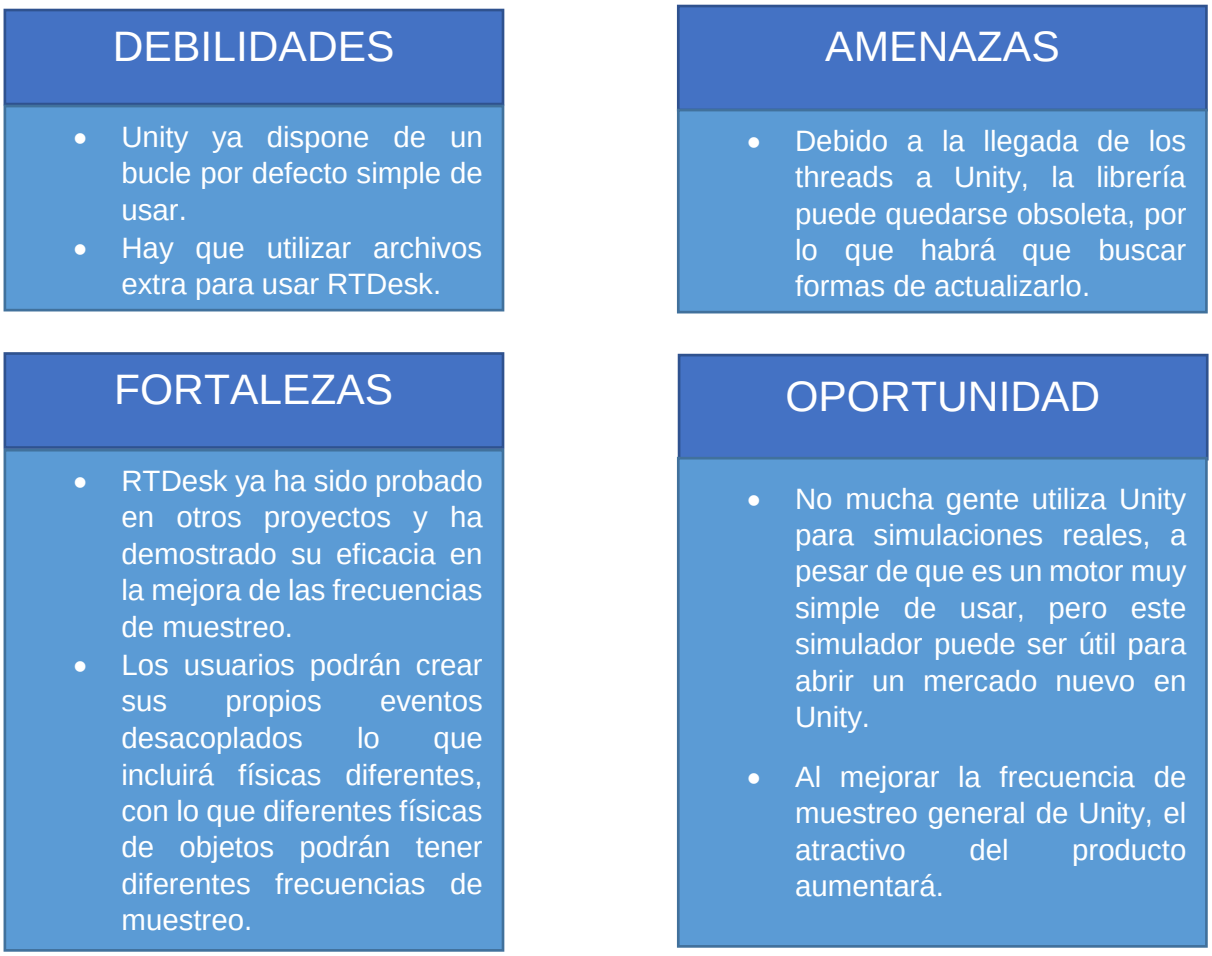

Ilustración 29 Análisis dafo del proyecto

En la Ilustración 29 se realiza el análisis DAFO del proyecto:

- Debilidades:
  - Como ya se ha explicado cuando se ha introducido Unity, este ya contiene un bucle de juego, con un temporizador que recorre la escena y ejecuta las funciones correspondientes a su tiempo. Este bucle estaba descrito en la Ilustración 18.
  - Por supuesto, como es lógico, incorporar cualquier cosa que no sea parte de Unity supone incorporar archivos extra, con lo cual hay que realizar una sencilla guía para los usuarios sobre como instalar los archivos y que estos no sean nada intrusivos.

- Amenazas:
  - La nueva aparición de los threads a Unity, que cambiará el api completamente puede hacer que la librería entera quede obsoleta, por lo que se debe de continuar con el proyecto una vez esto ocurra. La razón por la que cambiará el api es porque los threads, como se describió en el

subcapítulo anterior, es una forma de ejecutar dos partes diferentes de un código a la vez en dos partes diferentes de un procesador. Esto significa que, si dos threads acceden a la vez a una misma variable, pueden ocurrir errores de lectura, como que primero un thread debería modificar una variable y luego el otro leerla, pero ocurre a la vez y por lo tanto el código no se comporta como se esperaba. Para solucionar esto Unity utiliza un api completamente diferente en el que aún se está trabajando, por lo que no se puede definir. Los actuales avances se pueden consultar en la página web de los Jobs de Unity[17].

- Fortalezas:
    - RTDesk ya ha sido probado en proyectos anteriores y ha demostrado una mejora del rendimiento en las aplicaciones donde se ha implementado [13] [14]. Por lo que conseguir una mejora de la frecuencia a la que se actualiza el juego es un aspecto importante para considerar en el proyecto.
    - Los usuarios podrán crearse sus propios mensajes. Debido a que el sistema de mensajes de Unity no funciona con tiempo[18], que los usuarios puedan crearse sus propios mensajes es un incentivo para usar Unity con RTDesk. Un mensaje importante puede ser la creación de físicas realistas y poder mejorar las de Unity, gracias a la precisión del reloj de RTDesk, lo que atraería a más usuarios a Unity, no solo desarrolladores de videojuegos.
- Oportunidades:
    - Como ya se ha comentado en fortalezas, el poder crear sistemas nuevos con el sistema de mensajes gracias al reloj en tiempo real, puede ayudar a que profesionales que no podían usar Unity debido a cualquier problema con la precisión temporal, se replanteen su uso.
    - La mejora de la frecuencia de actualización puede mejorar videojuegos anteriormente realizados en Unity sin realizar trabajo nuevo, lo cual haría RTDesk muy apetecible para desarrolladores experimentados en Unity.

3.3 Solución propuesta

Como se ha comentado, la solución escogida es la representada en el análisis de la Ilustración 27 y la Ilustración 28.

En esta solución lo que se realizará es:

1. En primer lugar, se realizará el código que representa la **¡Error! No se encuentra el origen de la referencia.**. Este consistirá en exponer todas las funciones básicas de Engine, Message y Entity invocándolas desde un bloque extern C.
2. Se creará la clase en C++ que implementará la interfaz de entity. Esta contendrá como propiedad un puntero a la función de C# que ha de ser llamado desde C++. Así al implementar el método virtual en este objeto, se llamará siempre a una función de C#.
3. En C# se implementarán las llamadas a las funciones de C usando p/invoke, se realizará la prueba de que Startup (función de RTDesk para

[17] www.docs.unity3d.com/Manual/JobSystem.html
[18] www.docs.unity3d.com/ScriptReference/GameObject.SendMessage.html

arrancar el simulador) de la Engine funciona y que Simulate funciona también.

4. Después se probará a enviar un mensaje a la propia entidad a la que pertenece el mensaje y comprobar que se ejecuta la función receivemessage en C#.
5. Después se probará a enviar un mensaje con un tiempo determinado.
6. Y por último en las pruebas, se probará a enviar varios mensajes.
7. Tras esto, se acabará de construir el código perteneciente a la Ilustración 28 y se sustituirá el método de ejecución de Update.
8. Se probará que haya una mejora en la frecuencia de muestreo.
9. Se acabará de remplazar el método de ejecución de LateUpdate y FixedUpdate copiando la clase que reemplaza Update.
10. Se realizarán las pruebas finales donde se probará a fondo si hay mejora en la frecuencia de muestreo, que el renderizado no interfiera en esta frecuencia y que se puedan conseguir mejoras en las físicas con mejor rendimiento que en Unity e intentar sobrecargar con código basura todo lo que se pueda los métodos para realizar mejor las pruebas.
11. Probar la facilidad de implantación en un videojuego real usando plugins de Unity que se usarían en juegos oficiales o si se pueden conseguir, juegos.

3.4 Presupuesto

Primero de todo, se obtuvo una colección de plugins de una agrupación de software para Unity barata de una página web que consiste en la venta de software, libros y juegos en grupos para recaudar dinero para caridad. Este costo 13 euros y se obtuvo un plugin de creación de juegos de disparos en tercera persona que usa las funciones Update y FixedUpdate y probará la facilidad de la implementación de RTDesk en Unity.

En cuanto al resto del presupuesto consistiría en la contratación de un programador y un tester, ya que este proyecto solo consiste en la programación de código y no tiene nada de arte u otras ramas.

Por esto, contratar a 1 programador 4 meses y a un tester 2 consistirían en 1300/1500 euros por persona y unos 7800/9000 euros en total dependiendo de la negociación del salario.

4. Diseño de la solución

4.1 Arquitectura del sistema

La arquitectura del sistema es la siguiente:

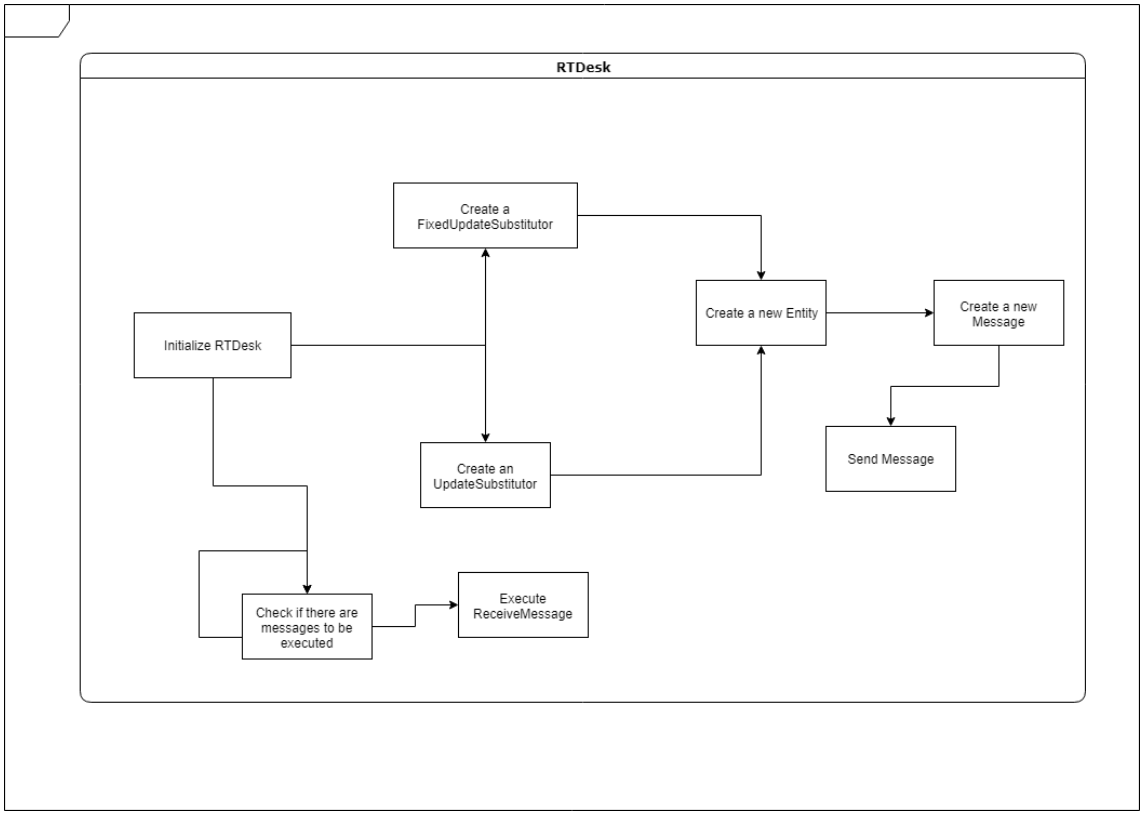

Ilustración 30 Estructura del bucle nuevo que simplifica el de Unity

En la Ilustración 30 primero se inicializa el motor de RTDesk, después se crean los objetos que sustituirán las funciones de Unity, se crea la Entidad en ellos y un mensaje, se envía el mensaje, mientras en un bucle se comprueba si hay mensajes en la cola y se ejecutan si su tiempo de ejecución supera al tiempo real.

4.2 Diseño detallado

En cuanto al diseño detallado, el UML de la solución esta descrito en la Ilustración 27 y la Ilustración 28. En el capítulo 5 Desarrollo de la adaptación de RTDesk a Unity se explicará por qué se utilizaron estos diseños y porque el otro diseño falló.

Estructuradamente, el primer UML representa un DLL, perteneciente al código de C++ y el segundo otro DLL, que pertenece al código de C# de RTDesk, que se encontrarán dentro de la carpeta Assets de Unity. A la hora de crear las DLL, la estructura de RTDesk se mantendrá y se usará la carpeta source para los ficheros C++ y en la carpeta header los headers. En cuanto a los ficheros del segundo DLL, se encontrarán todos en la carpeta raíz debido a que no hay muchos archivos que conforman el proyecto.

En cuanto a la función específica de cada parte de RTDesk, se explica a continuación:

- RTDeskEngine: Esta clase se encarga de almacenar el gestor de tiempo de alta resolución, la lista de mensajes y el gestor de la lista de mensajes.
- RTDeskEntity: Es la interfaz de la que heredan los objetos que se encargan de recibir los mensajes y contienen la función que se ejecutará al recibir el mensaje.
- RTDeskMsg: Contiene la definición de un mensaje, incluyendo la entidad a la que pertenece, la entidad a la que debe ser enviado y su posición en la lista de mensajes.
- RTDeskMsgDispatcher: Aquí se encuentra el bucle que envía los mensajes cuando deben ser enviados, además del reloj de simulación.
- RTDeskMsgPool: Contiene la lista de mensajes.

En la parte de Unity:

- Init: Aquí se crea el objeto de la clase engine, se arranca y se crean las entidades que contienen las funciones que reemplazaran a las llamadas de Unity. También tiene definidos las funciones de p/invoke necesarias para que alguna persona externa cree sus propios mensajes y entidades. Además de tener una función que limpia la memoria para los cambios de escena.
- FixedUpdateSubstitutor y UpdateSubstitutor: Contienen las funciones necesarias para reemplazar las funciones que llevan su nombre.
- MyProxyClass: Clase necesaria para manejar manualmente el colector de basura.

4.3 Tecnología utilizada

Entre los softwares utilizados están:

- Unity: Por supuesto, Unity debe ser utilizado ya que el foco del proyecto es implementar un simulador de eventos discretos en Unity
- Word: Para escribir la memoria se ha decidido usar Microsoft Word por la simplicidad de uso y porque la universidad regala la licencia.
- Visual Studio 2017: Para modificar el código y compilar se ha decidido usar Visual Studio 2017, ya que tiene el compilador tanto de C#, como de C++, además que permite hacer debug del código en C# de Unity. Otros factores son que Unity está finalizando el soporte de mono y Visual Studio viene por defecto en la instalación.

5 Desarrollo de la adaptación de RTDesk a Unity

Debido a que, como ya se había mencionado anteriormente, el desarrollo de este proyecto era iterativo, parte del proyecto ya se fue implementando para probar lo que funcionaba y se comentó en sus respectivas secciones. Por lo que en esta sección se expandirá sobre lo ya comentado en temas anteriores explicando en profundidad los problemas que se han encontrado al intentar comunicar Unity con la API de RTDesk.

Primero de todo, para realizar la adaptación de RTDESK a Unity, se tuvo que compilar una dll de 64 bits de la API. Para ello, en RTimer.h, la clase de RTDesk que conseguía

consultar el reloj de alta resolución en tiempo real usando código de ensamblador, se tuvo que cambiar el código escrito en ensamblador y moverlo a un archivo separado ya que C++ de 64 bits no admite este tipo de programación. Esta función dio bastantes problemas como el borrado de distintas variables, por lo que se acabó utilizando una función nativa de C llamada __rdtsc(), la cual hace lo mismo que el código que se usaba en ensamblador, devolver los ciclos de la cpu para tener un reloj en tiempo real.

Después de no conseguir hacer funcionar el código del UML de la Ilustración 24, por incompatibilidades con Unity, se pasó a la implementación del código relacionado con el UML de la Ilustración 27 y la Ilustración 28. Esta implementación consistió en crear, dentro de bloques externs C que sirven para exponer a otros lenguajes clases de C++, llamadas a las clases de C++, implementadas como se muestra en la Ilustración 31. Esto funcionó perfectamente hasta llegar a la clase Entity, la cual es virtual y, por lo tanto, como no existía una definición del método receivemessage (método llamado cada vez que se recibe un mensaje), no se podía exponer. El motivo por lo que esta función es virtual, es porque crearla así permite la llamada a la función desde la clase padre mientras tenemos una definición propia de que debe hacer la función. Así cada entidad puede realizar una función diferente sin tener que reescribir mucho código.

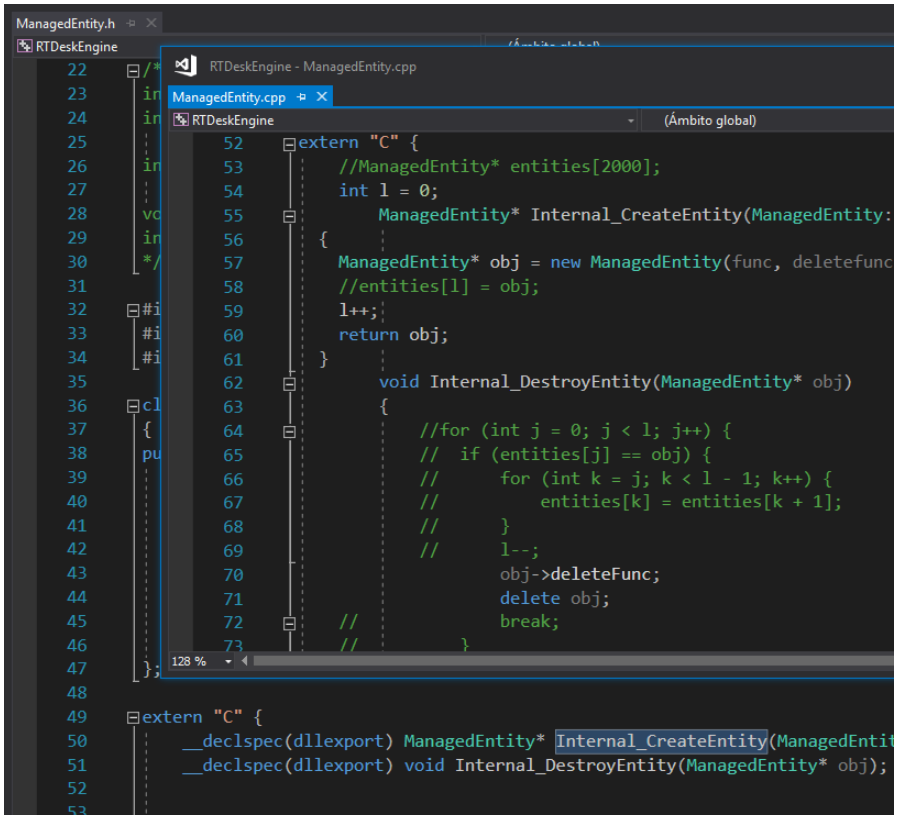

Ilustración 31 Imagen que muestra los archivos que contienen los extern, y como estos solo llaman a funciones de C++ y hacen visibles funciones para C#.

Para solucionar esto, es para lo que se creó la clase ManagedEntity, en C++. Lo que se hizo fue heredar de Entity en esta función, y definir una variable que fuera una función. Cuando se crea un objeto ManagedEntity en C#, se le pasa una función y esta queda asignada a la variable. Después, en el receivemessage que esta implementado en ManagedEntity en la parte de C++, se llama a esta función guardada, tal y como muestra la Ilustración 32.

```
void ManagedEntity::ReceiveMessage(RTDESK_CMsg * pMsg)
{
    func(pMsg);

}
extern "C" {
    //ManagedEntity* entities[2000];
    int l = 0;
        ManagedEntity* Internal_CreateEntity(ManagedEntity::Function func, ManagedEntity::Function deletefunc)
  {
    ManagedEntity* obj = new ManagedEntity(func, deletefunc);
    //entities[l] = obj;
    l++;
    return obj;
  }
```

Ilustración 32 Función que crea una clase ManagedEntity y la otra función que ejecuta las funciones de C# en C++.

A parte de exponer las funciones normales y los constructores, también se tuvo que exponer los destructores para manejar la memoria, ya que C# no puede destruir los objetos de C++.

A lo largo de las pruebas con el api, Unity dejaba de funcionar con muchas de las funciones debido a varios errores. A parte del error que provenía del código de ensamblador comentado previamente, hubo que reordenar el código que almacena los primeros y últimos mensajes, ya que eliminaban a los demás mensajes de la cola.

Así el código quedó como en la Ilustración 33, cuando antes se hacía la inserción del mensaje como última parte del código, lo que provocaba un error en Unity.

También en el message dispatcher, se reorganizó el código del bucle para que, después de enviar el mensaje, se actualizara el tiempo del siguiente mensaje, ya que antes ocurría que si, en el envío se almacenaba un mensaje A que debía ejecutarse antes que un mensaje B, el mensaje A no se ejecuta hasta que el plazo del mensaje B no se cumpliera. Así que se añadió el código de la Ilustración 33 para solucionar el problema.

```
*/
inline void InsertFirstMsg(RTDESK_CMsg *pMsg) { pMsg->InsertSingleMsg(this); First = pMsg; Last = pMsg;  };
```

Ilustración 33 Una de las reorganizaciones de código

En cuanto a la implementación de la librería de Unity que se comunica con el código en C++, lo primero fue crear la clase que implementaba el nuevo método Update, llamada UpdateSubstitutor. En esta, hay un comando nuevo, de Unity 2017, *[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.AfterSceneLoad)],*
el cual permite a un script ejecutar una función sobre la que esté escrito, aunque el script no se haya añadido a la escena, antes de cargar una escena. Si se desea cargarlo tras cargar la escena, solo hay que cambiar el afterSceneLoad por BeforeSceneLoad. En esta función, lo primero que se añadió es código para que Unity no ejecutara la función Update.
Tras esto, se creó un script llamado Init que heredaría de MonoBehaviour (4). Dentro de la función que se ejecuta cuando se ha cargado la escena, la que estaba precedida por el RuntimeInitializeOnLoadMethod, lo añadí a un objeto de la escena de Unity. Tras ver que esto funcionaba, que Update no se ejecutaba y que el script se añadía correctamente a la escena, comencé a implantar el DLL de RTDesk.

Por lo que de momento tenemos en el orden de las funciones RuntimeInitilizeOnLoad(desactivar update, crear gameObject al que se le adjunta el script Init) -> ejecución de la función Awake de Init.

Comencé por, en el Awake de Init, primer método que ejecuta Unity, inicializar la clase Engine de RTDesk, habiendo programado los p/invoke (5)necesarios. Aquí es donde observé el primer problema con el código de ensamblador y pude testear hasta que el código de ensamblador se ejecutaba sin problemas. A continuación, acabe de programar UpdateSubstitutor, la cual sería la clase que almacenaría la substitución de la función Update de Unity, implementando todos los p/invoke para crear la clase de managedEntity, crea los mensajes, enviarlos y asignar un dispatcher, el cual se encarga de enviar los mensajes del motor de RTDesk a sus respectivas entidades. Entonces, desde Init se creó un objeto UpdateSubstitutor y dentro de este una Entity y un mensaje. Al crear la Entity se le pasa la función CustomUpdateFunction, que es la nueva función update, y se probó si funcionaba implementando en una corutina que se ejecutaba cada 0 segundos el bucle de simulación.

Así, el esquema desde la ejecución de Awake queda como:

- Se ejecuta la función Awake de la clase Init, donde se crea el objeto engine de RTDesk, se inicia llamando a la función Start de RTdesk, que se encarga de inicializar las variables necesarias en C++ y se crea un tipo de clase UpdateSubstitutor
- En UpdateSubstitutor se crea una Entidad que, como se ha expresado anteriormente, almacena la función a ejecutar en C++.
- Se crea un mensaje, que será el objeto que se envíe desde RTDesk a Unity y viceversa.Después se le pasa una función creada por nosotros a esta Entidad. Esta función se encontrará en UpdateSubstitutor.
- Para acabar, por separado, una función que se ejecuta cada x segundos llama a la función de simulación de RTDesk, que ejecuta el bucle de juego asignado a RTDesk y llama a la función creada por nosotros con el mensaje de la entidad como parámetro.

Y en pseudocódigo se reflejaría así:

```
Class updateSubstitutor {
        var Entity;
        var Message;
        [RunTimeInitializeOnLoad]
        {
                Desactivar update;
                Añadir Init a un GameObject;
        }
        Function newUpdate(Message) {
                Mandar mensaje con x segundos
        }
}
Init() {
        Awake() {
                Engine = new RTDeskEngine.
                Engine.Start;
                customUpdateClass = new CustomUpdateClass;
                customUpdateClass.entity = new Entity;
                customUpdateClass.message = new Message;
                //El que despatcha los mensajes reside en la clase Engine
                customUpdateClass.setMessageDispatcher(Engine);
```

```
            Ejecutar corutina;
            Mandar mensaje de customUpdateClass con x segundos
        }
        Corutina() {
            While(true) {
                Engine.Simulate();
                Devolver Control a Unity;
            }
        }
}
```

Ilustración 34 Pseudocódigo de Unity

Aquí ocurrió un error que rompía Unity, ya que, después de ejecutar una vez CustomUpdateFunction, los objetos referenciados desde C++ a C# en la clase managedEntity, que eran la variable entidad y el mensaje del pseudocodigo de Ilustración 34, se borraban, por lo que tuve que investigar cómo solucionarlo.

Al final encontré una forma de controlar el colector de basura y de ahí surgió MyProxyClass, clase que tiene un objeto de la clase CustomUpdateFunction, y un puntero al objeto y otro objeto de tipo freePointer, el cual es una función que se crea en la clase MyProxyClassy libera los punteros, y un puntero a esta función. En la Ilustración 35 se observa como el objeto DoSomething es del tipo CustomUpdate, que era una función. DoSomethingPtr es el puntero a esta función. Después se crea la función FreeMe que libera el puntero de la memoria, y FreemePtr es el puntero a esta función. Al añadir los punteros usando el objeto GCHandle conseguimos que los objetos que dependen de las funciones no se borren.

```
private GCHandle Handle;
private CustomUpdate DoSomething { get; set; }

// You pass this function pointer to C++. Its signature is:
// bool(__stdcall *DoSomething)(int32_t, int32_t)
public IntPtr DoSomethingPtr { get; private set; }

private FreeMeDelegate FreeMe { get; set; }

// You have to pass this function too to C++. C++ code must use it
// to free MyProxyClass. Its signature is:
// void (__stdcall *FreeMe)(void)
public IntPtr FreeMePtr { get; private set; }
```

Ilustración 35 Pointers y funciones que almacena la clase proxy

Así, se creaba un objeto de la clase MyProxyClass que hace de proxy, y en el lado de C++ se pasa el puntero a la función CustomUpdateFunction de MyProxyClass.

Así se pudo realizar la prueba de que RTDesk enviaba y recibía mensajes. Al añadir dos mensajes a la cola, obtuve el error de que el mensaje anterior se borraba comentado anteriormente y que la función Update se ejecutaba varias veces al solo haber un mensaje en la cola, solucionado con la reorganización del dispatcher comentada anteriormente.

Tras esto, implementar FixedUpdate fue copiar la clase anterior en una nueva y cambiar los nombres, y LateUpdate fue ejecutar los LateUpdates después de los Updates en la función CustomUpdateFunction. La ejecución del tiempo de FixedUpdate se puede modificar a través de Unity de la misma manera que se hacía en con el FixedUpdate por defecto, ya que, como tiempo de ejecución del mensaje, se manda la variable que lo controlaba y que reside en, dentro de Unity, Edit->Project Settings->Time, como enseña la Ilustración 36.

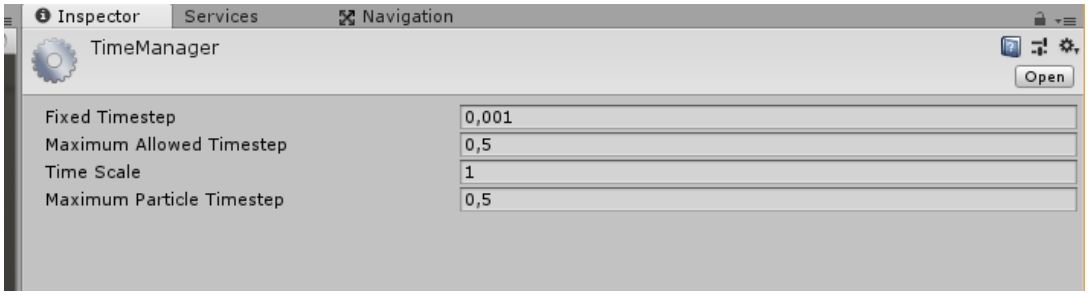

Ilustración 36 Como modificar el tiempo de ejecución de FIxedUpdate, solo cambiar el número de fixed Timestep

Al cambiar de escena y parar la reproducción del juego, Unity se rompía. Esto era debido a que los objetos que se habían dejado permanentes en memoria no se destruían y además la clase que los contenía sí que se destruía. Esto se solucionó haciendo permanente la clase Init, con una simple función llamada DontDestroyOnLoad(), que pone en una escena separada el objeto para que no se destruya y añadiendo dos funciones onApplicationQuit y onFinishedLevelLoading, las cuales ejecutan la función que libera de la memoria las entidades, el freeptr que hemos visto antes, acción que arregló ambos errores en Unity.

Para finalizar se deseaba que el usuario final no tuviese que realizar ningún esfuerzo a parte de colocar las dll en la carpeta de Assets de Unity. Para esto se implementó un método en la creación de las entidades donde se buscaban todos los objetos que heredarán de MonoBehaviour y los añade a una cola. Luego, al ejecutar la función Update, esta cola se recorre y se ejecutan las funciones update de cada objeto. Esto resultó ser horrible para el rendimiento ya que los objetos que no contenían update mostraban una advertencia que hacía perder rendimiento a la aplicación, ya que realizar un print es muy costosa. Además, no había forma de saber si un objeto contenía un método update, ya que Reflect (clase utilizada en C# para obtener el tipo de los atributos de una clase que se le pasa a Reflect[19]) de C# devolvía siempre que sí tenían una función Update debido a que heredan de MonoBehaviour.

Por lo que, al final, se optó por hacer públicas las colas y pedir a los usuarios que añadan los objetos que contienen un Update, LateUpdate y FixedUpdate a sus respectivas colas manualmente en el método Start().

```
public static List<Action> GbWithAnUpdate { get => gbWithAnUpdate; set => gbWithAnUpdate = value; }
public static List<Action> GbWithALateUpdate { get => gbWithALateUpdate; set => gbWithALateUpdate = valu
public static List<Action> GbWithAFixedUpdate { get => gbWithAFixedUpdate; set => gbWithAFixedUpdate = v
```

Ilustración 37 Las 3 colas donde se añaden las funciones y donde se ejecutarán.

Como extra, para que los usuarios pudiesen implementar sus propios mensajes, se creó una interfaz que solo contiene la definición de receivemessage, como en la Ilustración

[19] https://www.tutorialspoint.com/csharp/csharp_reflection

38. Luego, en la clase MyProxyClass, se permitió crear un objeto de esta clase que tuviese un puntero a la función receivemessage de las clases que implementarán la interfaz, tal y como muestra la Ilustración 39. Así, los usuarios pueden crear el mensaje y la entidad con un ejemplo aportado y no tienen por qué preocuparse por gestionar la memoria, tan solo llamar a liberarla cuando se cierre la aplicación.

```
namespace RTDESKUNITYDLL
{
    public interface IMessage
    {
        void ReceiveMessage(IntPtr passedMsg);
    }
}
```

Ilustración 38 La interfaz creada

```
public MyProxyClass(RTDESKUNITYDLL.IMessage obj)
{
    Handle = GCHandle.Alloc(this, GCHandleType.Normal);

    // This will implicitly create a reference to obj. GC can't
    // collect MyProxyClass because we have a GCHandle on it,
    // and can't collect obj because there is a delegate that
    // has a reference on it.
    DoSomething = obj.ReceiveMessage;
    DoSomethingPtr = Marshal.GetFunctionPointerForDelegate(DoSomething);

    FreeMe = FreeMeImpl;
    FreeMePtr = Marshal.GetFunctionPointerForDelegate(FreeMe);
}
```

Ilustración 39 La función que crea un objeto de MyProxyClass y crea los punteros de la función que implemente la interfaz, sin que el usuario haga nada más que heredar de la interfaz

Al final de todo se intentó implementar el sistema de Jobs en la dll, que consiste en usar threads con objetos de Unity, pero debido a que aún están en beta, no podía usar la librería de donde se heredan dentro de la dll porque no la reconocía.

Un cambio importante que no se refleja en los anteriores apartados, es el añadido de el scriptable render pipeline(11) con el objetivo de que RTDesk pueda manejar el rendering de Unity. Así se creó un rendering pipeline customizado, donde al principio hay un booleano que comprueba si el mensaje de renderizar de RTDesk ha llegado. Si no ha llegado, no se rendriza nada. Se comprobó que funcionaba ya que, en los datos de debug del editor de Unity, la mayoría de las veces decía que renderizaba 0 triángulos en una prueba donde el mensaje de renderizado era menos frecuente que la frecuencia de actualización del bucle.

6 Implantación

A la hora de añadir la librería de RTDesk a un videojuego comercial, se realizó la prueba con un plugin comercial llamado UFPS[20], mencionado en los presupuestos, en un juego rápido creado por mí. Esto es debido a que obtener código de un juego a la venta es imposible porque los desarrolladores no quieren dar gratis el código. También se puede probar en los ejemplos de programación de Unity, pero el plugin es más complejo que estos y se utiliza de verdad en juegos comerciales, por lo que fue mejor idea implementarlo en un juego rápido y probar la implementación de RTDesk.

Para comenzar, como ya se ha mencionado anteriormente, solo hay que añadir los dos DLL a la carpeta Assets de Unity. Al arrancar el juego la clase Init se añadirá automáticamente a la escena y así RTDesk comenzará a funcionar.

Para hacer que los objetos que contienen Updates, FixedUpdates y LateUpdates se ejecuten, hay que ir script por script y en la función onEnable escribir el código:

RTDESKUNITY.Init.[array].Add(this).

Siendo array los objetos de Init GbWithAnUpdate, GbWithAFixedUpdate o GbWithALateUpdate, y en onDisable de Unity lo mismo, pero en vez de Add, Remove, tal y como se refleja en la Ilustración 40.

```
private void OnEnable()
{
    RTDESKUNITY.Init.GbWithAnUpdate.Add(Update);

}
private void OnDisable()
{
    RTDESKUNITY.Init.GbWithAnUpdate.Remove(Update);

}
```

Ilustración 40 Como añadir la función Update al array.

Sí una persona desea crearse sus propios mensajes, solo debe crear una clase nueva que implemente la interfaz IMessage, que tenga dos objetos de tipo IntPtr (punteros), siendo estos la entidad y al menos, un mensaje.

Luego en un gameobject hay que crear un objeto de este script. Después, un objeto de la clase MyProxyClass pasándole el primer objeto al constructor de la clase y crear la entidad y el mensaje, que son funciones de la clase Init, escribiendo como parámetros de entrada a la entidad los punteros creados por MyProxyClass . Después, llamar a las funciones de Init newMessageType según la cantidad de mensajes, asignar el messagedispatcher a la entidad con setMessageDispatcher de entity, y tras esto, ya se puede enviar el mensaje con tiempo o sin tiempo según se quiera usando una de las dos funciones de la clase Init que lo permiten.

[20] https://assetstore.unity.com/packages/templates/systems/ufps-ultimate-fps-2943

```
entity2 = new RTDESKUNITY.colliderEntity(initGB);
proxyClass = new RTDESKUNITY.MyProxyClass(entity2);
entity2.entity = initGB.NewEntity(proxyClass.DoSomethingPtr, proxyClass.FreeMePtr);
entity2.messageCollider = initGB.NewMessage(proxyClass.DoSomethingPtr);
entity2.setPositionPlayer(ball.transform.position);
entity2.setPositionzombie(this.transform.position);
entity2.setHit(hitfunct);
entity2.askForPosition = askForPosition;
initPosition = ball.transform.position;
initGB.SetMessageDispatcher(entity2.entity);
initGB.SetPropietaryMessage(entity2.messageCollider, true);
entity2.sendMsg();
```

Ilustración 41 Como crear una clase customizada

Como se puede ver en la Ilustración 41, colliderEntity es la clase ejemplo con la función customizada. Se crea una entidad nueva, un mensaje nuevo, se asigna la función, se asigna el dispatcher, se asigna el mensaje como propietario y se envía el mensaje. El resto de las instrucciones son propiedades que no tienen nada que ver con RTDesk.

Para implementar el rendering pipeline, hay que crearlo como Unity menciona en sus ejemplos de la Scriptable Rendering Pipeline, que no atañen a este proyecto, y añadirlo en el apartado de gráficos del editor de Unity, como se ha comentado en la sección anterior. Después solo hay un semáforo, el cual seabre o cierra cuando llega un mensaje de RTDesk antes de comenzar el pipeline de renderizado.

7 Pruebas

Como ha aparecido en capítulos anteriores, las pruebas se realizan sobre un juego que implementa el plugin de UFPS para comprobar que la implementación funciona sobre cualquier juego comercial. Aunque antes, para hacer una comprobación segura sobre la frecuencia de muestreo también se intentó realizar una prueba sobre la demo creada por Unity de Survival Shooter, pero en la versión descargada faltaban objetos en la demo y no funcionaba bien, por lo que se acabó optando por 2D RogueLike otra demo de Unity. El rendimiento fue exactamente el mismo debido a que solo había dos métodos update. Al final se optó por Roll-A-Ball, un simple juego en el que se mueve una bola en una mesa y se recolectan 10 objetos. Demasiado simple para unas pruebas a fondo, pero lo suficiente como para valorar que funciona la API.

Las pruebas definitivas consistían en un juego con RTDesk y otro sin RTDesk. En la versión de RTDesk se probó a crear un mensaje que llamase a todos los métodos Update, otro a todos los fixedUpdate y otro a todos los métodos lateUpdate. Otra prueba fue crear un mensaje para cada función de cada objeto y la última prueba de rendering fue probar a modificar el rendering pipeline y que cada 16.66 milisegundos se renderizara tras un mensaje de RTDESK en vez de permitir que Unity renderice en cada frame.

El juego creado especialmente en este proyecto consiste en un juego de disparos usando el plugin UFPS, donde grandes hordas de zombies siguen a un jugador y hay que dispararlas. Para comenzar, el juego crea 300 zombies en la escena. Cada zombie estaba formado por 9000 triángulos. En la Ilustración 42 se muestra una captura del juego creado.

Ilustración 42 Imagen del juego donde se ven los zombies y como el jugador puede disparar a los zombies. La mano y el sistema de disparos los aporta el plugin UFPS

7.1 Prueba de frecuencia de muestreo

La primera prueba consiste en ejecutar el juego con RTDesk implementado encargándose de las funciones Update, LateUpdate y FixedUpdate y ejecutar el mismo juego sin implementar RTDesk. Los resultados son los siguientes:

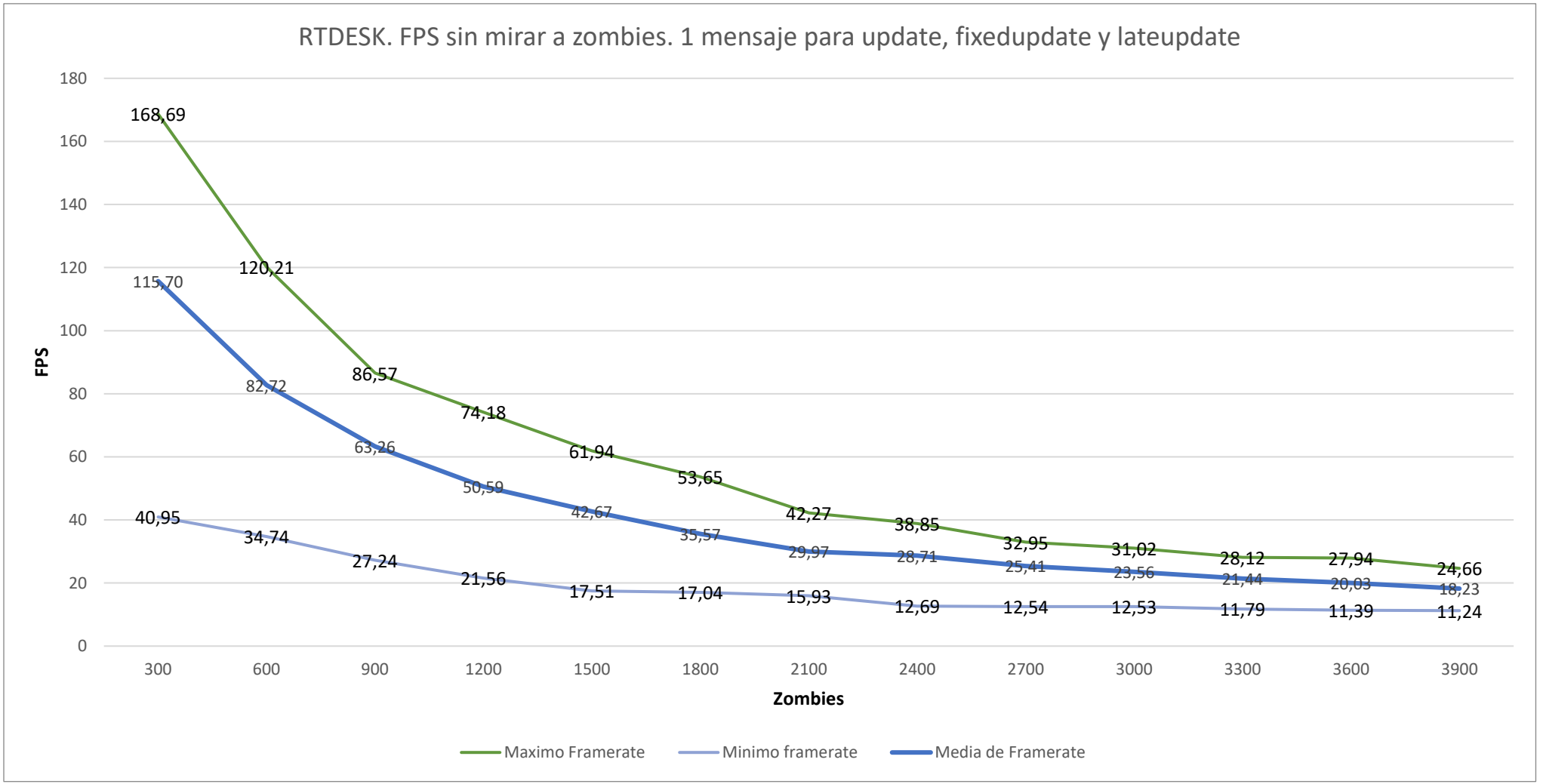

Ilustración 43 Gráfica que muestra los fps de RTDesk sin mirar a los zombies.

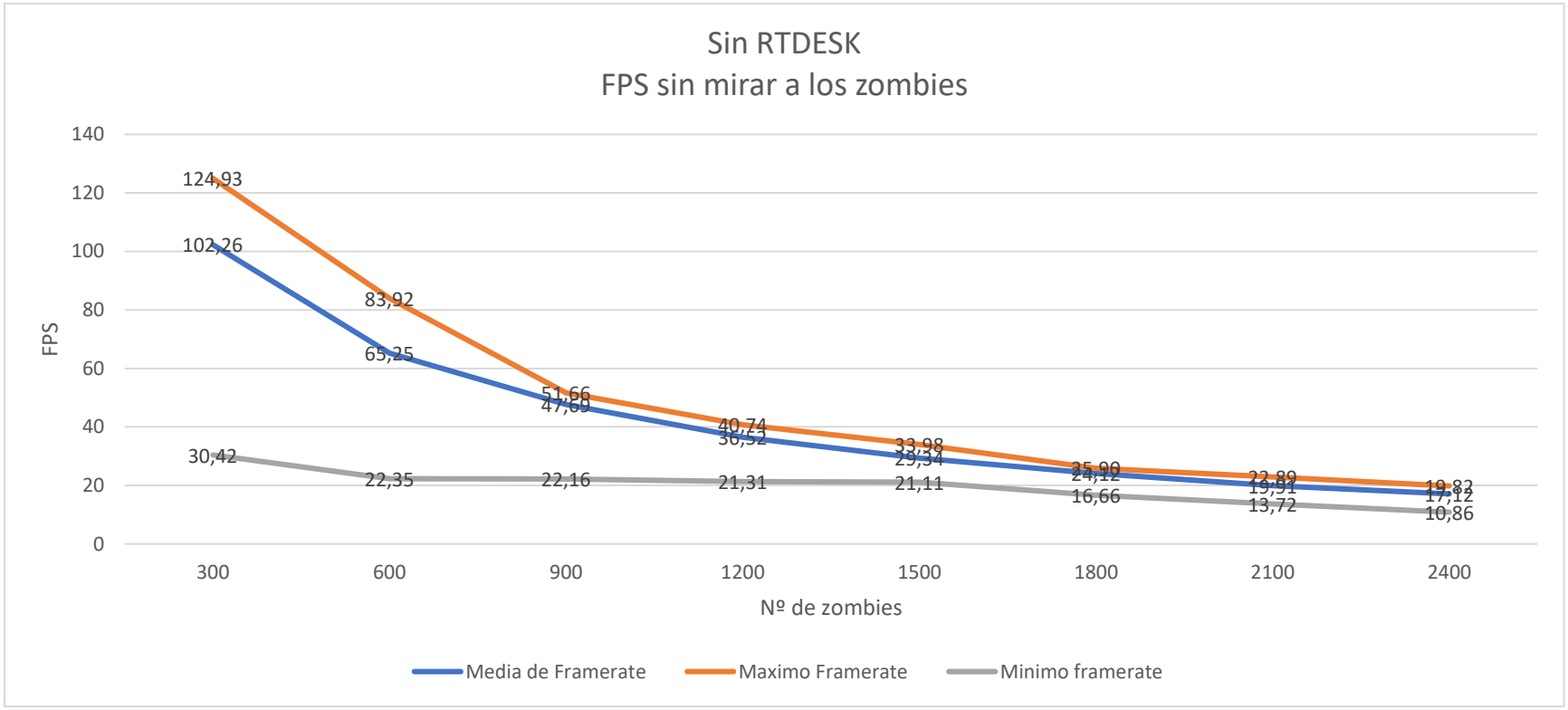

Ilustración 44 Gráfica que muestra los fps del mismo juego sim implementar RTDesk

Lo cual claramente otorga una mejora a la solución con RTDesk a pesar de todos los problemas generados para implementarlo.

Primero de todo se aprecia que la prueba sin RTDesk tiene menos zombies creados en la escena que en el modo RTDesk. Esto es debido a que a partir de los 2100 zombies ya teníamos sobre los 10 frames por segundo, lo cual hace que dificulta el control del juego por parte del jugador, y la animación de la simulación no se vea fluida lo cual no es ideal. RTDesk tiene más zombies con la misma cantidad de frames. Esto puede ser debido a una mejora en el uso de las arrays que contienen los métodos Update, FixedUpdate y LateUpdate, una mejora al ejecutar los métodos desde C++ o incluso una mejora debido a poder ejecutar las funciones en el tiempo correcto gracias a RTDesk. Debido a que ya se prueba que el juego con RTDesk funciona mejor, los siguientes test no usan los juegos sin RTDesk o son más limitados. La media de mejora en los frames por parte de RTDesk es de un 13%. Este 13% no es uniforme, como se puede observar, en la primera hay un 13% de diferencia, pero en la segunda, con 600 zombies, hay 17% de diferencia en el rendimiento.

A continuación, se estudia si asignar un mensaje para cada objeto de la escena, que cada objeto llame a su propio método Update, podría suponer una mejora:

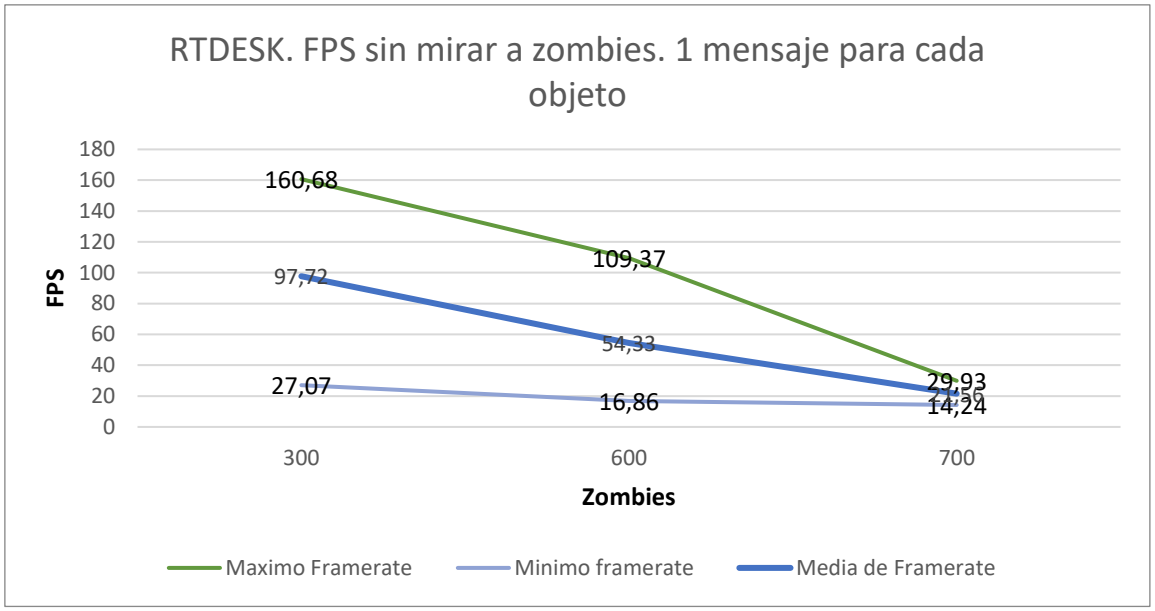

Ilustración 45 Gráfica que muestra los frames si cada objeto tiene su propio gestor de mensajes que gestiona los Updates, FIxedUpdate y LateUpdates.

A partir de 600 objetos el juego sufre una caída enorme en la frecuencia de actualización debido probablemente al coste que supone traducir cada función asignada de C# a C++. Por lo que esta idea se descartó. La última prueba fue tras descubrir como manipular el renderizado de Unity, ver la mejoría en la frecuencia de actualización con el render pipeline modificado y sin modificar:

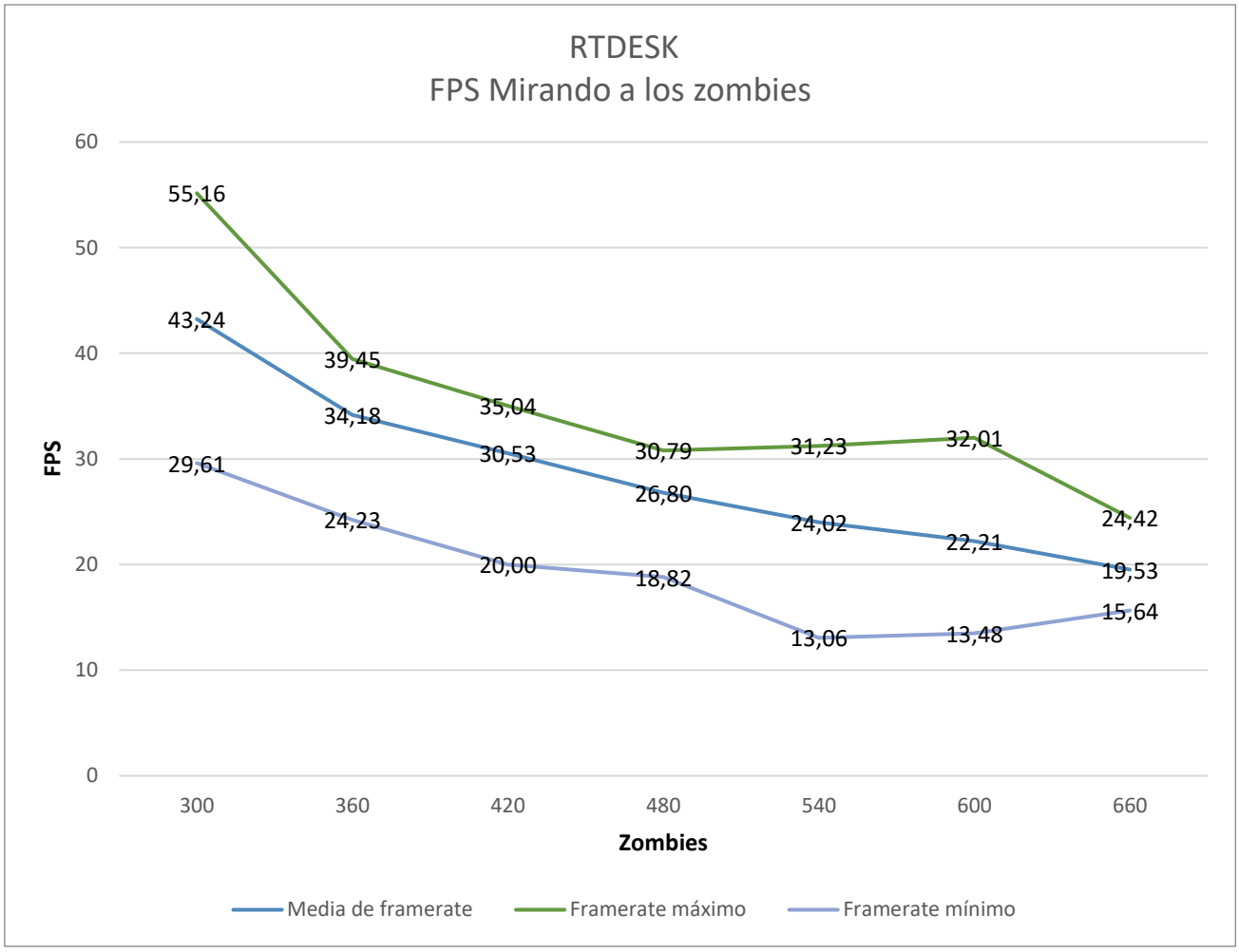

Ilustración 46 Framerate cuando la cámara renderiza los zombies con RTDesk y el render pipeline modificado.

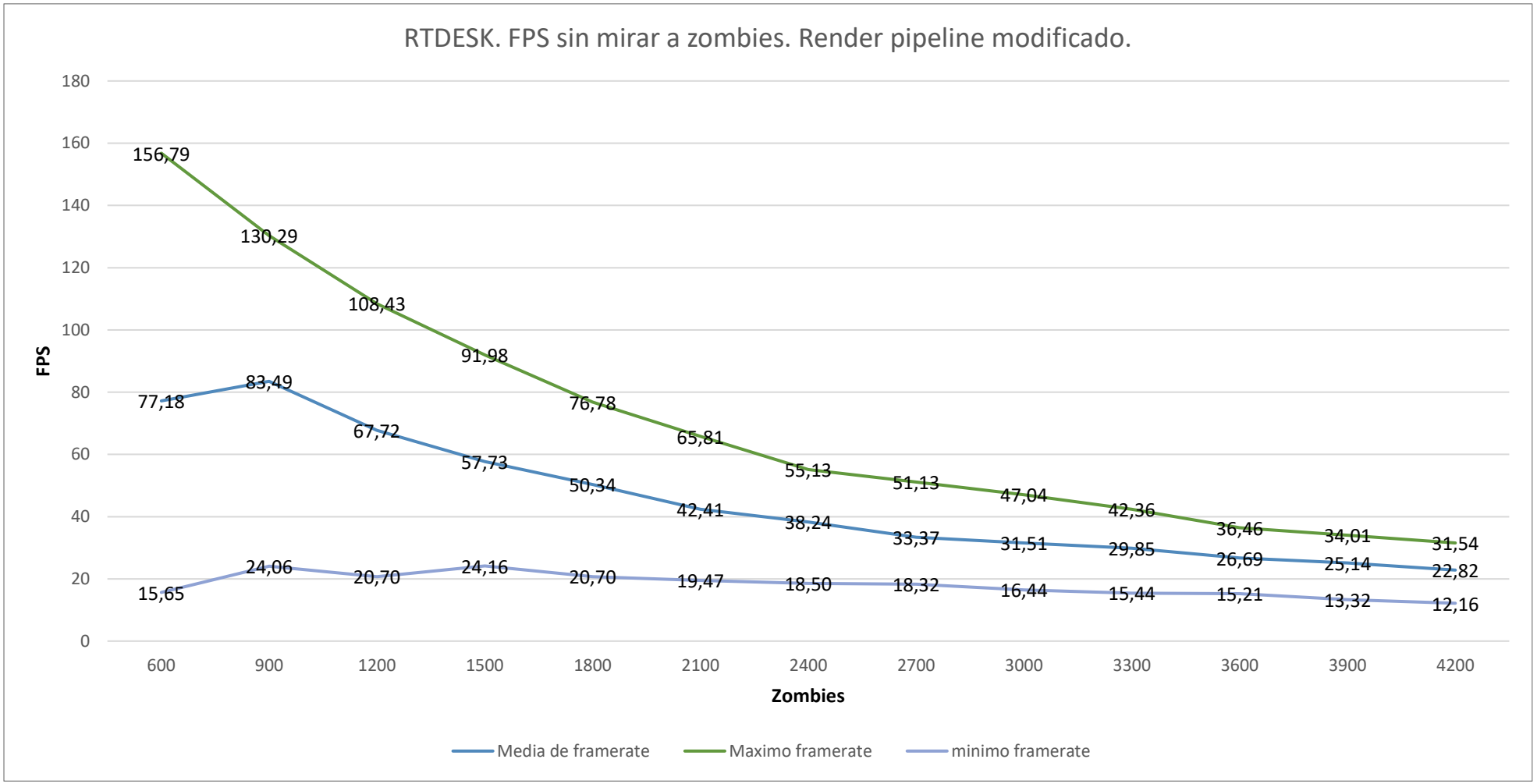

Ilustración 47 Framerate sin mirar a los zombies de RTDesk con el render pipeline modificado.

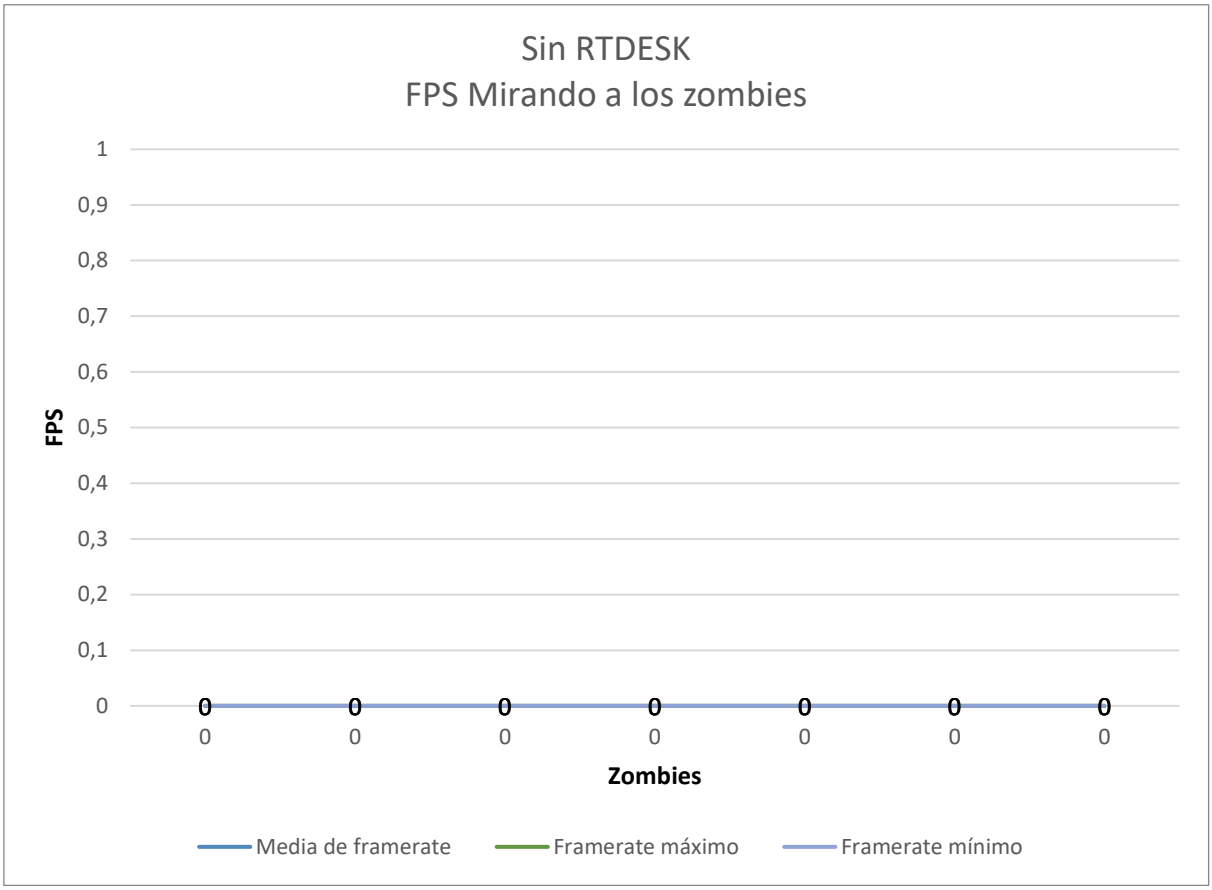

Ilustración 48 Framerate del juego sin RTDesk con el render pipeline modificado.

En esta ocasión se encuentran más igualados debido a que el trabajo de renderización de los zombies probablemente es más costoso y RTDesk no puede ayudar tanto a un proceso que probablemente ocupe la mayoría de los frames por segundo. La razón por la que no se han incluido más zombies en la prueba sin RTDesk en la Ilustración 48 es debido a que ya quedaba probado que el juego con RTDesk funcionaba mejor y solo había que probar que mejoras introduce el scriptable render pipeline. En cuanto a la Ilustración 47, la razón por la que el framerate con 600 zombies es tan bajo, es porque este dato se obtuvo junto a los datos mirando a los zombies, lo cual pudo insertar datos falsos mientras limpiaba la memoria de los objetos en 3D. La mejora de frames media es de un 11% frente al RTDesk sin modificar el render pipeline. Este valor tampoco es uniforme a través de toda la gráfica, ya que, como se puede ver, hay un 20% de diferencia con los 900 zombies creados. También hay que destacar que en la Ilustración 46 con 540 zombies hay una bajada de zombies mínima, la cual en 600 zombies se mantiene, lo cual puede ser debido a una carga de algún objeto que Unity ha debido de realizar en el momento de 540 zombies, y que da un mínimo que solo debería haber ocurrido con los 600 zombies. Y en el siguiente el framerate máximo se mantiene debido al descanso de esta tarea. En la Ilustración 49 se observa el funcionamiento del

Rendering Pipeline y RTDesk. Framerate limitado a 60 frames por segundo, tenemos frames en los que no se dibuja nada.

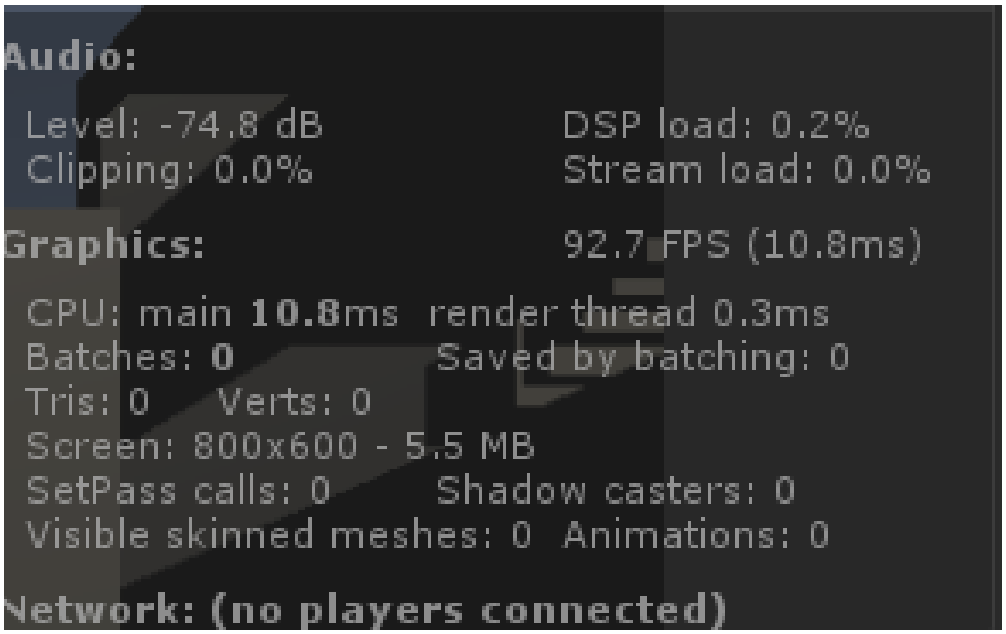

Ilustración 49 Rendering pipeline en funcionamiento con RTDesk, frame en el que no renderiza nada

Por último, para medir el tiempo de idle, se va a limitar a Unity a funcionar a 60 frames por segundo con vsync, técnica para limitar los frames de la pantalla a la misma tasa que el monitor puede mostrarlos. A la vez, también en Unity sin RTDesk y usando el scriptable render pipeline, se usará RTDesk para renderizar la escena. El tiempo que tarda cada uno en renderizar será el idle time de cada uno.

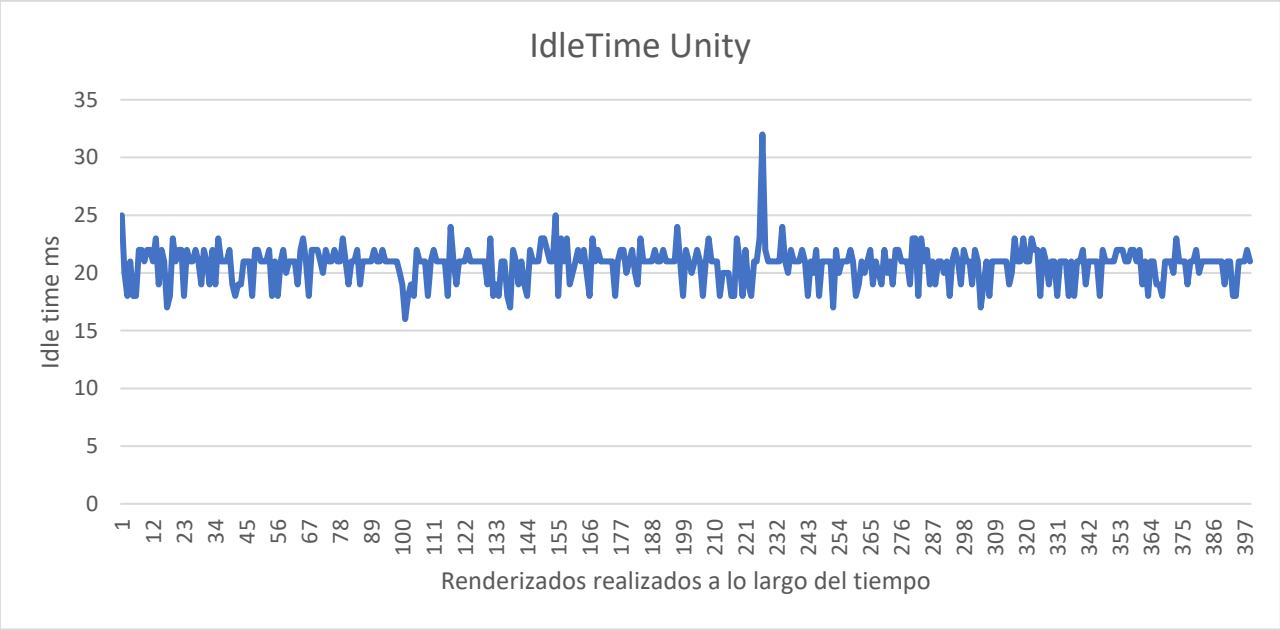

Ilustración 50 IdleTime de Unity

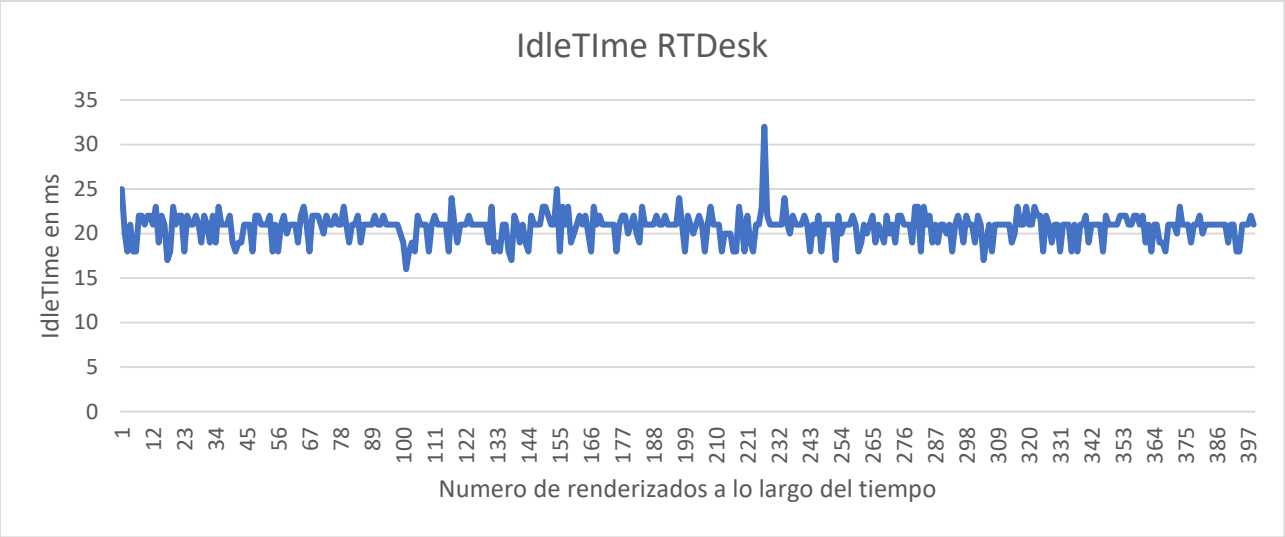

Ilustración 51 Idletime de RTDesk

En estas dos ilustraciones se puede ver como Unity es consistente mientras que RTDesk no. Aun así, se puede observar que RTDesk salta entre idle times de 60 fps y 30, probablemente debido al problema de tener que ejecutarse dentro de Unity. La explicación de que esto ocurra es que, Unity se está saltando 1 frame exacto a veces en la ejecución de RTDesk. Por lo que se decidió realizar una prueba y doblar los frames con el objetivo de obtener 60 frames por segundo y obtener el idle time deseado. En la Ilustración 52 podemos observar que la prueba fue un éxito y que efectivamente, Unity se salta a veces un frame exacto en el que debería ejecutar el núcleo de RTDesk. Al final ambas gráficas son iguales, lo cual nos dice que el idleTime de Unity por defecto es bastante fiable, y que deja el tiempo necesario para renderizar a tiempo, pero no para ejecutar subrutinas. Por lo que si pudiésemos ejecutar RTDesk fuera de Unity debería funcionar correctamente, y la gráfica sería incluso más plana que la de Unity. Los frames por debajo de 30 ocurren en ambos y puede ser debido a cualquier tarea que la tarjeta gráfica tenga que ejecutar, por lo que no se deben tener en cuenta.

Después me fijé que Unity tenía el fixedDeltaTime a 0.001 y estaba ejecutando la función fixedUpdate aún sin tener ninguna función escrita de este tipo en la escena. Tras esto subí el número a 0.02 y la tabla para el idle time con 16.66 ms comenzaba a variar entre 22 milisegundos y 16, como se puede ver en la Ilustración 53, pero en la misma frecuencia que en la Ilustración 51. Ante la relación de ambos datos, se puede deducir que es Unity quien no da prioridad a la función que ejecuta RTDesk y que ejecuta una función fixedUpdate a pesar de que no hay ninguna entidad con esta función asociada

a un script.Este puede ser un motivo por el que RTDesk funcione mejor que Unity normal, RTDesk no ejecuta ninguna función que no tenga ninguna entidad.

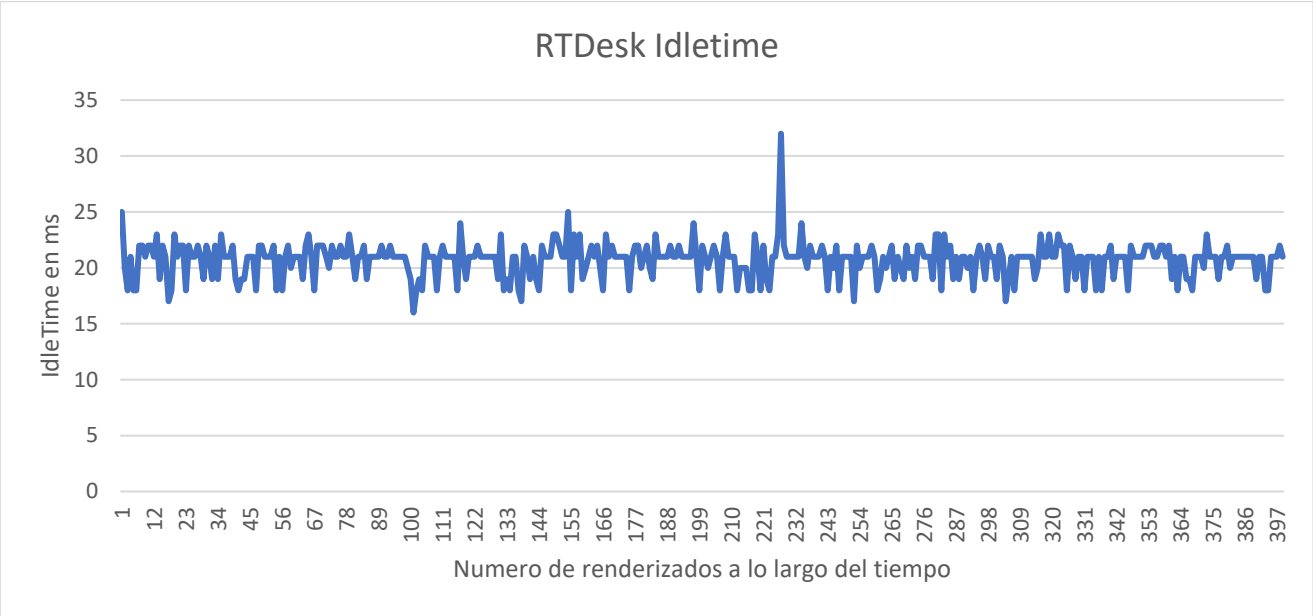

Ilustración 52 Idletime de RTDesk 2, intentando conseguir 60 frames por segundo.

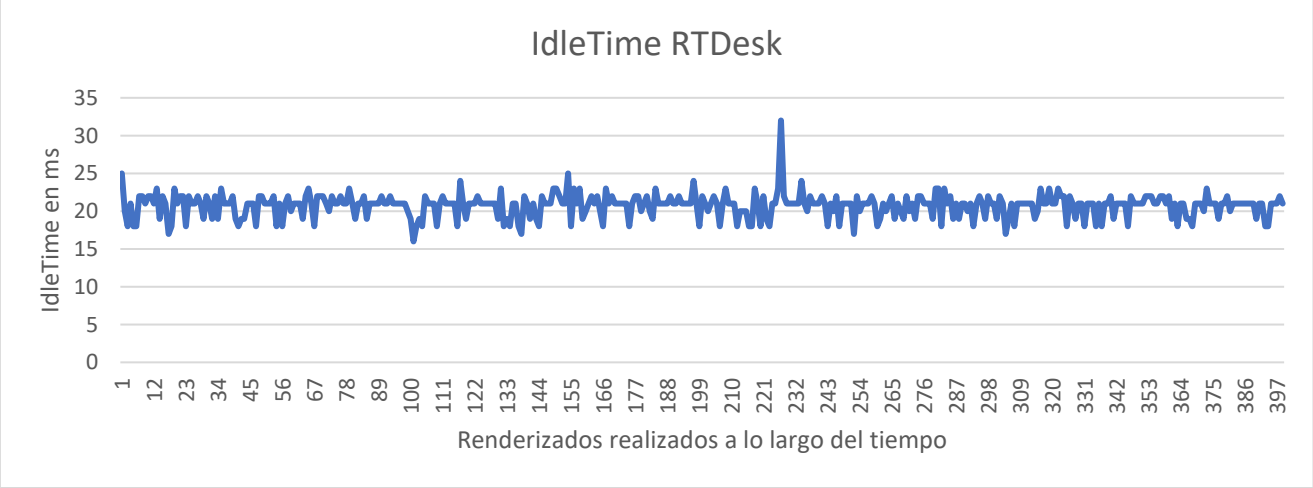

Ilustración 53 IdleTime de RTDesk tras subir el tiempo de fixedUpdate a 0.2

7.2 Pruebas de mejoras de físicas

A continuación, se realizan dos pruebas para averiguar las mejoras en las físicas que pueden aportar RTDesk. Esta prueba puede fallar debido a que Unity implementa Physx para las físicas, api que es un motor físico creado por Nvidia muy optimizado en sus funciones.

Para empezar, se probará a colocar diversos zombies lejos del jugador. Se consultará en un mensaje si están cerca del jugador y si pueden colisionar. Dependiendo de la distancia de los zombies, el tiempo de mensaje será mayor o menor. En la prueba sin RTDesk, se ejecutará los colisionadores por defecto.

En la siguiente prueba se implementó una bola disparada de una posición y se visualizó si el efecto túnel ocurría. Primero se empezó por otorgar a la bola 1 unidad de fuerza, con 100 milisegundos en la frecuencia de actualización de la función que comprueba si la bola se choca. Después se subió a 5, 10, 50, 100, 500, 1000, 5000, 10000, 50000, y 100000. También se tuvo que bajar la frecuencia de actualización de la función a 10 milisegundos.

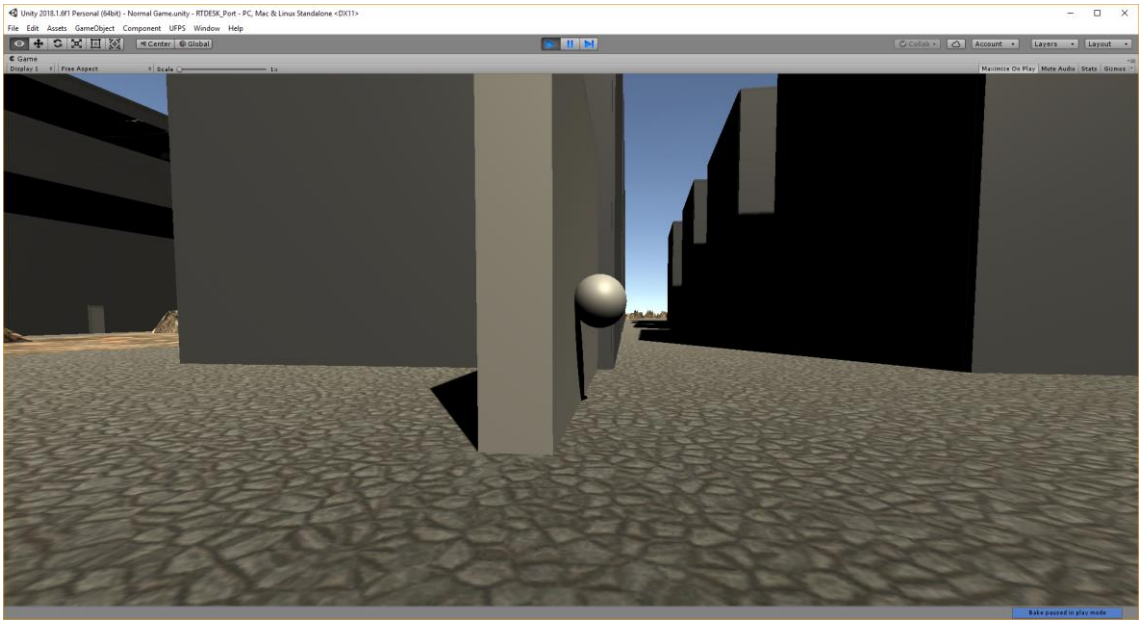

Ilustración 54 Colisión con el muro exitosa. Cuando RTDesk fallaba, se quedaba la bola al otro lado del muro

En la primera prueba no se encontró diferencias en la simulación.

En cambio, en la segunda, tras aplicar una fuerza de 500000 en el eje x, y tener que ejecutar la función que comprueba la colisión cada 1 milisegundo, la simulación se comportaba de manera errática. Esto es debido a que, la función donde Unity ejecuta la simulación de RTDesk no puede ejecutar esta función cada milisegundo. En las pruebas, esta función se ejecutaba a tiempo siempre que estuviese por encima de 10 milisegundos.

Esto indica que, si tuviésemos acceso a un sitio fuera del bucle de Unity que nos permitiese ejecutar cada milisegundo el bucle, no debería existir ningún problema con la simulación, pero estamos limitados por la API de Unity.

En cuanto a la simulación de Unity, fue siempre exacta y no ocurrió el efecto túnel en ninguna de las pruebas. En RTDesk se tuvo que crear un sistema propio de físicas para conseguir físicas deterministas.

Los desarrolladores del motor no permiten ejecutar ninguna función fuera de este que se comunique con la api de Monobehaviour, por lo que se vio imposible ejecutar fuera RTDesk. El único remedio para conseguir una precisión exacta en el tiempo de RTDesk fue usar un thread que cuenta el tiempo, pasársela a una función IEnumerator (8) y hacer que la función IEnumerator que contiene el núcleo de simulación de RTDesk se ejecutara cada vez que el thread devolvía que el tiempo para el siguiente mensaje había llegado.

Como añadido si se implementara en C# RTDesk completo se podría hacer una prueba que manejará el mensaje de colisión para cada objeto manteniendo una frecuencia de actualización correcta, pero seguiría teniendo el problema de que la corutina donde se ejecutará la simulación no puede ejecutarse en periodos demasiado cortos de tiempo.

Como anotación final, no hay comparación de tiempo de espera del motor debido a que Unity se ejecuta siempre al máximo tiempo que puede y no tiene tiempo de espera o no se puede sacar el tiempo alojado para tareas del sistema operativo, por lo que para la comparación del Idle Time se tuvo que usar lo mostrado en las ilustraciones de la Ilustración 50 a la Ilustración 53.

8 Conclusión

A lo largo de la memoria, se ha podido apreciar las dificultades de implementar RTDesk en un juego de Unity, pero también como se creó una API que permitió el uso de RTDesk de una manera sencilla.

En cuanto al funcionamiento de este, se han podido observar estas características:

- En las pruebas se ha podido observar que RTDesk funciona mejor que el propio Unity en cuanto a frecuencia de actualización. Reescribir el motor en C# puro podría solucionar un problema que surgió cuando en la escena aparecen muchas instancias de objetos con mensajes de RTDesk, pero se ha observado que esto se consiguió remediar con un gestor que recibiera los mensajes de RTDesk y ejecutara individualmente estos mensajes.
- Lo que sí que no se consiguió remediar es, que al tener que ejecutar el API dentro del flujo de juego de Unity, si Unity no tiene tiempo para ejecutar RTDesk, puede saltarse ejecuciones del simulador, lo que se probó en la prueba de renderizado. En cambio, si hay tiempo, RTDesk puede ser igual o más fiable y manipulable que Unity, tal y como enseñaba la Ilustración 52.

Existen varias formas de solucionar este problema:

- Uno sería si Unity permitiese ejecutar threads con sus objetos. El sistema de Jobs de Unity que aún se está implementando pretende hacer esto, pero utilizará objetos diferentes y cualquier modificación en estos se realiza cuando el thread acaba, pasando los parámetros de los objetos que son seguros usar en un thread a los no seguros. Para ciertos mensajes esto sería ideal, pero si se quiere sustituir las funciones de Update, FixedUpdate y LateUpdate ya existentes para realizar una conversión de juegos antiguos sin ningún esfuerzo

a RTDesk, esto es inviable. Por lo que un trabajo futuro podría crear un RTDesk para Unity que se encargara solo de ciertos tipos de mensajes que se pudiesen realizar con threads.

- La otra opción sería tener acceso al código para poder ejecutar RTDesk cuando debería junto al bucle principal de Unity, e incluso poder implementar todas las funciones del bucle principal de Unity en RTDesk. Pero no es posible acceder al código, por lo que es inviable. Esto sería especialmente útil para los mensajes de renderizado, ya que podríamos acceder a la renderización de cada objeto por separado y decidir cuando deberían renderizar o también a la función de renderizado general, ya que el método que se utilizó no es ideal, como ya se comentó en la implementación.

El desarrollo de este trabajo de final de máster ha permitido observar cómo funciona un motor de videojuegos profesional como Unity, sus ventajas, desventajas, fortalezas y debilidades más a fondo de lo que se da durante el máster. También ha permitido aplicar los conocimientos de programación de motores y gráficos obtenidos durante la realización del máster tales como la programación en C++ de un motor de gráficos y la implementación de un motor de simulación discreto (en este caso RTDesk) a Unity. Por último, se ha aplicado conocimientos otorgados en el máster como el uso de Unity o aplicaciones gráficas en general.

9 Trabajo futuro

Trabajos pendientes que podrían ser posible estudiar son, principalmente:

- Una implementación de RTDesk pura en C#, para solucionar el problema de coste al pasar a C++. Esto puede traer un reloj menos preciso.
- Intentar usar el sistema de Jobs y el ECS de Unity para comprobar el rendimiento. No se ha implementado en este proyecto debido a que el sistema está aún en beta e incompleto, pero si se lanzara de forma que fuera perfecto para el objetivo de este proyecto, solucionaría todos los problemas de RTDesk, al poder ejecutarse fuera de Unity, con un thread seguro.

Anexo

Guía de uso

La utilización de RTDesk en Unity es la siguiente:

1. Importar los dll del proyecto.
2. Añadir el tag RTDESK a la lista de tags.
3. Cambiar el nombre a las funciones FixedUpdate y LateUpdate por FIxedUpdate2 y LateUpdate2.
4. Añadir las funciones Update, FIxedUpdate Y lateUpdate a las arrays de RTDesk (para añadir la función update habría que escribir en la función onEnable del mismo script Init.GbWithAnUpdate.Add(Update); y cambiar add por remove en onDisable. Para fixedUpdate el array sería GbWithAFixedUpdate y para LateUpdate GbWithAFixedUpdate.
5. En el caso de que deban crearse nuevos tipos de mensajes, habría que realizar dos pasos:
    a. Crear un script con la estructura de la Ilustración 55.
    b. Añadir en el Start de un objeto de Unity la estructura de la Ilustración 56.

Y así RTdesk estaría implementado en un software creado en Unity. Es una forma muy sencilla de añadirlo ya que la sencillez era uno de los objetivos del proyecto.

```
using UnityEngine;
namespace RTDESKUNITY
{
    public class colliderEntity : RTDESKUNITYDLL.IMessage
    {

        public IntPtr entity = IntPtr.Zero;
        public IntPtr message = IntPtr.Zero;
        Init initObject;
        Action functionToCall;

        public colliderEntity(Init initGB)
        {
            initObject = initGB;
        }
        public void Destroy()
        {

            if (entity != IntPtr.Zero)
            {

                initObject.DestroyMessage(message);
                message = IntPtr.Zero;


                initObject.DestroyEntity(entity);
                initObject = null;
                functionToCall = null;
                entity = IntPtr.Zero;
            }
        }
        public void ReceiveMessage(IntPtr passedMsg)
        {

            functionToCall();



            initObject.SendMessageWithTime(entity, passedMsg, entity, 10);
        }
    }
}
```

Ilustración 55 Código para implementar RTDesk

```
initGB = GameObject.Find("RTDESKGB").GetComponent<RTDESKUNITY.Init>();
entity2 = new RTDESKUNITY.colliderEntity(initGB);
proxyClass = new RTDESKUNITY.MyProxyClass(entity2);
entity2.entity = initGB.NewEntity(proxyClass.DoSomethingPtr, proxyClass.FreeMePtr);
entity2.message = initGB.NewMessage(proxyClass.DoSomethingPtr);
entity2.functionToCall = hitfunct;
initGB.SetMessageDispatcher(entity2.entity);
initGB.SetPropietaryMessage(entity2.message, true);
initGB.SendMessageWithTime(entity2.entity, entity2.message, entity2.entity, 10);
```

Ilustración 56 Segunda parte de la implementación de RTDesk que debe de ser introducida en una función Start

Glosario

1. Gizmo: Ayuda visual para corregir la escena o mostrar información.
2. Yield: Instrucción que se introduce en las corrutinas (funciones con una parada en la función) para que la corrutina pare su ejecución y reanude tras la acción indicada en el yield. Por ejemplo, se usarían dentro de una corrutina como yield return EsperarunSegundo.
3. RigidBody: Se pueden definir como cuerpos rígidos que Unity utiliza para aplicar físicas en las escenas. Sin estos no hay colisiones.
4. MonoBehaviour: Api creada por Unity para programar en C# Unity. Sin este, no se puede añadir ningún script a la escena.
5. p/invoke: Acción para poder traducir una función de C++ a C# o viceversa.
6. Efecto túnel: efecto que ocurre en una simulación cuando un objeto traspasa un sólido con el que debería colisionar.
7. API: conjunto de librerías en un programa que exponen funciones extra que el programador puede usar.
8. IEnumerator: En Unity son funciones que actúan como corutinas. Las corutinas sirven para interrumpir el funcionamiento de una función tras una orden yield.
9. CLI: Command line interface, es un lenguaje derivado de C# para implementar funciones de C++ en el framework de C# .NET.
10. ECS: es un tipo de programación mayormente usado en videojuegos que se basa en que todos los objetos de una escena son entidades y todas las entidades han de estar formadas por un componente. Unity está en proceso de incorporar a su scripting este método de programación.
11. Scriptable render pipeline: Una nueva función de Unity que permite modificar el modo en el que una escena se renderiza. Gracias a esto también se incorpora un nuevo método para programar shaders basados en nodos similar al de otras herramientas como Blender o Unreal.
12. Gcroot: clase de CLI que facilita el manejo de clases.
13. Heredar en una clase: Cuando se crea una clase en script en otro script se pude referenciar a la primera clase para decir que esta hereda de ella y puede usar sus funciones.
14. Función virtual: función dentro de una clase padre, cuyas clases heredadas pueden sobrescribir con su propia función, manteniendo el nombre de esta.

Bibliografía

[1] J. Gregory, Game engine architecture., AK Peters/CRC Press., (2017).

[2] N. M. A. &. G. A. Munassar, « A comparison between five models of software engineering. International Journal of Computer Science Issues (IJCSI), 7(5), 94.,» *International Journal of Computer Science Issues (IJCSI), ,* pp. 7(5), 94., (2010).

[3] R. Nystrom, Game Programming Patterns.

[4] P. G. Junker, «OGRE 3D. Programming.».

[5] M. &. Z. M. &. C. E. &. L.-T. R. C. &. M. A. &. V. L. &. F. B. &. P. P. Joselli, «An architecture with automatic load balancing for real-time simulation and visualization syste,» 2010.

[6] K. Noland, «High frame-rate television: sampling theory, the human visual system and why the Nyquist-Shannon theorem doesn't apply».

[7] J. Rojas, «Getting Started with Videogame Development,» de *26th Conference on Graphics, Patterns and Images Tutorials (SIBGRAPI-T)*, Arequipa, Peru, 2013.

[8] L. C. A. &. F. B. Valente, « Real time game loop models for single-player computer games.,» *In Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment (Vol. 89, p. 99).,* (2005)..

[9] J. &. H. P. &. K. P. &. L. M. Polcar, «Using Unity3D as an Elevator Simulation Tool.,» (2017).

[10] R. M. T. B. Inmaculada García, «GDESK: Game Discrete Event Simulation Kernel».

[11] R. M. Inmaculada García, «Simulación Desacoplada de Eventos Discretos en Videojuegos».

[12] V. Broseta Toribio, «Cambio de Paradigma en el Control de Aplicaciones Gráficas en Tiempo Real.,» (2012).

[13] D. Pérez Climent, «Olympic Punch. Creación de un videojuego en XNA utilizando RT-Desk».

[14] J. Barrachina Verdía, «Development and performance analysis of a videogame using the "Real Time Discrete Simulation Kernel"».

[15] N. P. Z. F. L. G. M. J. C. J. L. Gonzalez Sánchez, «De la Usabilidad a la Jugabilidad: Diseño de Videojuegos Centrado en el Jugador.».

[16] K. Murnane, «From Pong To Playstation: The 40-Year Evolution Of Gaming Processing Power».

[17] J. McCall, «https://teachinghistory.org/history-content/ask-a-historian/25764,» [En línea].

[18] W. Higinbotham, « Tennis for two.,» (1958).Retrieved September, 9, 2015..

[19] R. Nystrom, Game Programming Patterns, 02 Nov 2014.