Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Monitor de rendimiento de un videojuego empleando un API de simulación de eventos discretos

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Antonio Araque Jiménez

Tutor: Ramón Mollá Vayá

Curso 2020/2021

Resumen

RT-DESK es una librería en C++ orientada a la simulación mediante eventos discretos. El objetivo principal de este proyecto es el desarrollo de un monitor de rendimiento dentro de RT-DESK. Este monitor ayudará a optimizar la gestión de recursos como la memoria o la lógica del paso de mensajes, que son aspectos importantes en el proceso de simulación de programas.

Para el desarrollo del trabajo se utilizarán herramientas como: TortoiseSVN para acceder a los ficheros del motor y poder subir los cambios realizados al repositorio; Visual Studio 2019 para la programación de utilidades y Zotero/Mendeley para gestión de citas bibliográficas.

El objetivo se llevará a cabo gracias a la implementación de un módulo que mediante código obtenga información relevante sobre el estado de la simulación. Por ejemplo, tiempos de espera a la hora de intercambiar mensajes o la cantidad de tipos de eventos diferentes. Posteriormente, se servirá de dicha información para estudiar los cambios a realizar en la simulación a fin de obtener mejoras respecto a versiones anteriores. Este proceso pretende abrir paso para la solución de problemas de rendimiento, mejorando la eficacia y eficiencia del simulador.

Para alcanzar esta meta se trabajará en visual studio desarrollando nuevas secciones de código y nuevos objetos sobre el proyecto de RT-DESK dado. Para esto también será de gran importancia la comprensión del simulador, de cómo se organiza y cómo se comunican sus clases. Al ser el objetivo final de este proyecto la modificación del motor es de gran importancia entender el rol que asume cada una de sus partes.

El trabajo seguirá el siguiente orden:

1. Introducción conteniendo aspectos como por qué se ha elegido este trabajo, qué se pretende conseguir con él y cómo se hará.
2. Planteamiento general de la situación actual en el ámbito de desarrollo del trabajo, tanto tecnológicamente como sobre el entorno de trabajo a utilizar.
3. Análisis de la estructura y funcionamiento interno de RT-DESK.
4. Planteamiento de la estrategia a seguir para conseguir los objetivos propuestos.
5. Implementar el código necesario para poder obtener información de valor durante la simulación.
6. Comprobar el correcto funcionamiento de las nuevas utilidades implementadas mediante pruebas de test unitarias.
7. Pensamientos finales y relación con la carrera.

También se encuentran disponibles al final de este documento numerosos apéndices cuyo objetivo es facilitar una mejor comprensión de aspectos concretos del simulador, tratar temas relacionados con el mismo y exponer soluciones que han sido descartadas durante el desarrollo del proyecto, pero que pueden ser de utilidad u ofrecer valor teórico.

Palabras clave: Simulador, eventos discretos, monitor, información, estructura interna, RT-DESK.

Abstract

RT-DESK is a library in C++ focused on simulation by means of discrete events. The main objective of this project is the development of a performance monitor inside RT-DESK. This monitor will help optimise resource management like memory or logic during message exchange, which are important aspects in the process of program simulation.

For the development of the project numerous tools will be needed: TortoiseSVN to access the engine files and to allow uploading the changes made to the repository; Visual Studio 2019 for the programming of utilities and Zotero/Mendeley for the management of bibliographic citations.

The objective will be carried out by the implementation of a module which with code will obtain information important to the state of the simulation. For example, idle times when exchanging messages or the number of different types of events. After this, this information will be used to study which changes to carry out on the simulation to obtain upgrades in relation to previous versions. With this process it is expected to make way to resolve performance problems, improving the efficiency and efficacy of the simulator.

To accomplish this goal visual studio will be the workspace, developing new code sections and new objects on top of RT-DESK's given project. For this it will also be of great importance to understand the simulator, how it is organized and how its classes communicate. Being the modification of the engine the final objective of this project it is of great importance to understand the role of each of its parts.

The assignment will follow the next order:

1. Introduction containing aspects such as why this project was chosen, what it intends to achieve and how will it be done.
2. General approach to the current situation in the project's development field, as much the technological one as the workspace to be used.
3. Analysis of RT-DESK's structure and internal functioning.
4. Planning of the strategy to follow to achieve the proposed objectives.
5. Implement the needed code to obtain valuable information during the simulation.
6. Verify the correct functioning of the newly implemented utilities through unit testing.
7. Final thoughts and connection with the degree.

There are also many appendices available at the end of this document whose objective is to ease a better comprehension of concrete aspects of the simulator, address matters related to it and present solutions which have been discarded along the development of the project, but which can be of use or offer theoretical value.

Key words: Simulator, discrete events, monitor, information, inner structure, RT-DESK.

Tabla de contenidos

1. Introducción 9
  1.1 Motivación 10
1.2 Objetivos 13
  1.3 Estructura 14
  1.4 Convenciones 15
2. Contexto tecnológico........ 16
3. Análisis del problema........ 20
  3.1 Organización interna de RT-DESK........ 21
  3.2 Análisis de requisitos 23
  3.3 Identificación y análisis de soluciones posibles 25
  3.4Análisis de la solución propuesta……….28
4. Guía de diseño de la solución........ 34
5. Desarrollo de la solución propuesta 39
  5.1 Número de eventos producidos durante la simulación/eventos producidos de cada tipo . 42
  5.2 Tiempo de espera a la memoria dinámica durante la creación de eventos........ 48
  5.3 Diferencia máxima entre tiempo de simulación y tiempo real........ 53
  5.4 Número de eventos recibidos por un objeto 56
  5.5 Solución de errores en el proyecto de pruebas y lanzamiento de tests......... 59
    5.5.1 Solución de errores en *TestGetEventNumber* 61
    5.5.2 Solución de errores en *TestGetEventTypeNumber* 63
    5.5.3 Solución de errores en *TestCreateMsgTime* 64
    5.5.4 Solución de errores en *TestRealSimulationTimeDiff* 65
    5.5.5 Solución de errores en *TestEventsReceivedEntity* 66
6. Conclusiones 68
  6.1 Relación del trabajo desarrollado con los estudios cursados........ 69
7. Trabajos futuros 70
8. Bibliografía 71
9. Apéndice 1: Funcionamiento interno de RT-DESK........ 73
  9.1 Relojes en RT-DESK 73
  9.2 Frecuencia de muestreo 74
  9.3 Funcionamiento de mensajes en RT-DESK........ 75
  9.4 Patrón de diseño en RT-DESK........ 77
  9.5 Organización del código de RT-DESK 78

10. Apéndice 2: Información relacionada 80
10.1 Bibliotecas estáticas y dinámicas 80
11.Configuraciones propias de Visual Studio 82
11.1 Opciones de inclusión de ficheros en Visual Studio 82
11.2 Actualización de la 'Platform toolset' de un proyecto en Visual Studio..... 84
12. Apéndice 3: Desarrollos de la solución descartados/no implementados..... 85
12.1 Implementación de compilación condicional..... 85
12.2 Guía para la incorporación del nuevo motor al entorno de pruebas..... 86

Índice de figuras

Ilustración 1: Ranking de los lenguajes más usados con el cambio en % de su valoración........ 10
Ilustración 2: Ranking de los lenguajes más relevantes según el Institute of Electrical and Electronics Engineers (IEEE)... 10
Ilustración 4: ISFE key facts 2020 12
Ilustración 5: Bucle principal de Doom, Quake y Fly3D.. 17
Ilustración 6: Bucle principal de RT-DESK.. 18
Ilustración 7: Esquema básico de la organización interna de RT-DESK 21
Ilustración 8: Fragmento de código del proyecto 'DemoGameOfLifeRTDesk......................... 26
Ilustración 9: Esquema de relaciones entre cabeceras en RT-DESK 28
Ilustración 10: Vista general RTDeskEngine.. 29
Ilustración 11: Header Files RTDeskEngine.. 29
Ilustración 12: Source files RTDeskEngine 31
Ilustración 13: Diferenciación funciones inline/no inline 34
Ilustración 14: Nueva estructura en RTDeskEngine.h 35
Ilustración 15: Cambio de estructura en RT-DESK.. 36
Ilustración 16: Declaración de nuevas estructuras de datos en engine.................................... 37
Ilustración 17: Uso de constantes en RTDeskEngine.cpp.. 37
Ilustración 18: Definición de tipos de mensajes en RTDeskMsg... 39
Ilustración 19: Función ReceiveMessage() en engine... 40
Ilustración 20: Función ChangePoolManager() de Engine 41
Ilustración 21: Función SetEventCounterTypes en RTDeskMonitor.h 42
Ilustración 22: Parte del código de DispatchMsg() en DispatcherMonitor.............................. 43
Ilustración 23: Función GetEventNumber()... 43
Ilustración 24: Función GetEventNumber(int type).. 43
Ilustración 25: Definición de un nuevo tipo de mensaje cMsgDispatcher en DispatcherMonitor 44
Ilustración 26: Función ReceiveMessage en DispatcherMonitor... 44
Ilustración 27: Implementación de la primera prueba sobre el primer caso de uso en el proyecto de tests 46
Ilustración 28: Bucle de envío de mensajes dentro de SelfMessageLoop() en la clase User...... 47
Ilustración 29: Implementación de la segunda prueba sobre el primer caso de uso en el proyecto de tests 47
Ilustración 30: Orden de llamadas en RT-DESK al solicitar un mensaje 48
Ilustración 31: Método Pop() monitorizado 49
Ilustración 32: Método ReceiveMessage() de PoolManagerMonitor 50
Ilustración 33: Método de prueba para medición de tiempos de Pop() en el proyecto de tests .. 51
Ilustración 34: Método MessagePopLoop de User en el proyecto de tests............................... 51
Ilustración 35: Método RealSimulationTimeDifference() en DispatcherMonitor 53
Ilustración 36: Recepción de mensaje MONITOR_DISPATCHER_TIME_DIFFERENCE en dispatcher monitor.. 54
Ilustración 37: Implementación de la prueba del tercer caso de uso en el proyecto de tests 54
Ilustración 38:Estructura de datos en dispatcher para contar mensajes recibidos por cada entidad 56
Ilustración 39:Segunda modificación sobre el método DispatchMsg 56
Ilustración 40: Función GetEventsReceivedEntity(RTDESK_CEntity) en dispatcher monitor. 57

Ilustración 41: Recepción de MONITOR_DISPATCHER_ENTITY_RECEIVED en dispatcher monitor 57
Ilustración 42: Método de prueba TestEventsReceivedEntity en el proyecto de pruebas........ 57
Ilustración 43: Paso 1 creación proyecto de pruebas 60
Ilustración 44: Paso 2 creación proyecto de pruebas 60
Ilustración 45: Debuggeando métodos de prueba (1)........ 61
Ilustración 46: Debuggeando métodos de prueba (2)........ 61
Ilustración 47: Nueva creación y envío de mensaje en SelfMessageLoop()........ 61
Ilustración 48: Lanzamiento correcto de los dos primeros tests........ 62
Ilustración 49: Modificación de SelfMessageLoop() para TestGetEventTypeNumber........ 63
Ilustración 50: Nueva implementación del método 'DispatchMsg()' monitorizado 66
Ilustración 51: Lanzamiento correcto de todos los test 67
Ilustración 52: Condición de avance de la simulación. 73
Ilustración 53: Ejemplo de diferentes frecuencias de muestreo 74
Ilustración 54: Función DispatchMsg() en RTDeskMsgDispatcher 1 75
Ilustración 55: Función DispatchMsg() en RTDeskMsgDispatcher 2 75
Ilustración 56: Función Pop() en RTDeskMsgPoolManager 1 76
Ilustración 57: Función GenerateNewMsgs() en RTDeskMsgPool........ 76
Ilustración 58: Función Pop() en RTDeskMsgPoolManager 2 76
Ilustración 59: Interfaz de Doxywizard........ 78
Ilustración 69: Menú propiedades en VSTUDIO........ 82
Ilustración 70: Propiedades de configuración en VSTUDIO 82
Ilustración 71: Edición de directorios a incluir en VSTUDIO 83
Ilustración 72:Salida de consola al actualizar el platform toolset 84
Ilustración 73:Definición y uso ejemplo de la constante de monitorización en RTDeskDefCom.h (en este caso desactivada) 85
Ilustración 74: Paso 1 creación de plantilla........ 87
Ilustración 75: Paso 3 creación de plantilla........ 87

1. Introducción

RT-DESK (Real Time Discrete Event Simulation Kernel) es una librería en C++ diseñada para la gestión temporal de eventos en tiempo real. Esta librería logra a la vez ofrecer las ventajas de la simulación discreta y sincronizar la simulación con el tiempo real durante su ejecución[1]. RT-DESK está enfocado tanto al desarrollo de aplicaciones en tiempo real como al modelado de sistemas consistentes en una colección masiva de objetos con interacción entre ellos. Se caracteriza por gestionar mensajes de eventos con marcas temporales y distribuirlos sincronizándolos con el tiempo real.

RT-DESK por lo tanto se encarga principalmente de manejar y entregar mensajes con órdenes y/o información entre objetos de una misma simulación dictando el desarrollo de esta. Además, gracias a estos mensajes y sus atributos permite una mayor libertad a la hora de simular programas.

Al igual que muchas otras herramientas de simulación de eventos RT-DESK se podría beneficiar de un módulo encargado de controlar el rendimiento del programa para evitar fallos de eficiencia. Esto permitiría aprovechar de manera óptima los recursos disponibles y gestionar de manera correcta el tiempo dado. El desarrollo de este módulo de monitorización junto con el estudio del motor que conlleva será el objetivo principal de este trabajo.

¿Por qué es importante el rendimiento en un simulador? Se plantea por ejemplo una simulación de un videojuego, donde el jugador puede observar cientos de entidades con diferentes comportamientos. Cada una de estas entidades tiene su comportamiento dictado por el simulador y, si funciona correctamente, el usuario tendrá una experiencia fluida y agradable. Pero ¿qué ocurre si el simulador se sobrecarga y no es capaz de gestionar el sistema? En este caso se comienzan a dar problemas de rendimiento, que afectan al usuario y enturbian su experiencia.

Este ejemplo es en el mundo de los videojuegos, pero también puede afectar a otros tipos de simulaciones. En otros ámbitos la ineficiencia del simulador puede tener consecuencias peores como obtención de resultados inesperados en una simulación o aumentos excesivos en el tiempo invertido para simular.

A lo largo de este documento se presentarán conceptos tanto propios de RT-DESK como externos que se beneficien de una explicación más extensa. Para estos casos existe al final del documento un apéndice, cuyo objetivo es tratar temas considerados de cierta importancia para el desarrollo del trabajo. Esta información no se considera necesaria para la comprensión del proyecto, pero puede ayudar en la comprensión del mismo.

1: [**RTDESK. RTDESK ai2, Presentation.** Disponible en: http://rtdesk.ai2.upv.es/en/presentation/].

1.1 Motivación

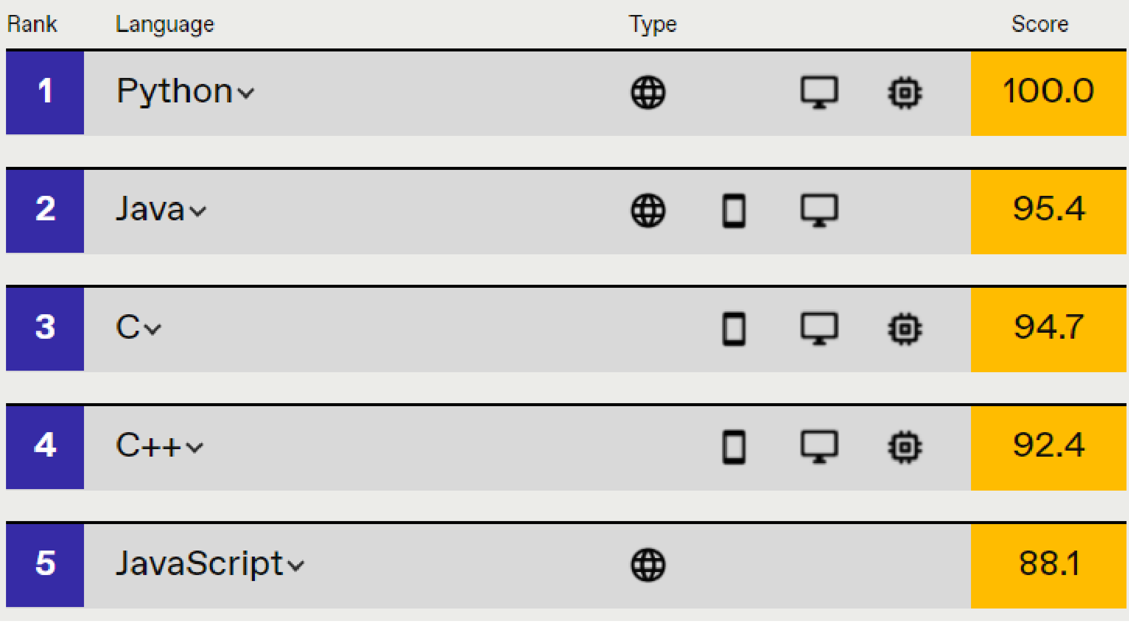
El desarrollo de este proyecto planteó un interés especial por diferentes motivos:

1. Poder trabajar en el lenguaje C++ es de gran interés, ya que ayuda a la comprensión y manejo de este lenguaje, el cual se encuentra entre los lenguajes más utilizados. En la siguiente figura se pueden observar tanto su valoración y posición respecto a otros lenguajes como el cambio en valoración sufrido (13. Kumar y Dahiya 2017):

May 2017	May 2016	Change	Programming Language	Ratings	Change
1	1		Java	14.639%	-6.32%
2	2		C	7.002%	-6.22%
3	3		C++	4.751%	-1.95%
4	5		Python	3.548%	-0.24%
5	4		C#	3.457%	-1.02%
6	10		Visual Basic .NET	3.391%	+1.07%
7	7		JavaScript	3.071%	+0.73%

Ilustración 1: Ranking de los lenguajes más usados con el cambio en % de su valoración

Si se desea comprobar esta información con datos más actualizados se comprobará que, aún tras perder popularidad, C++ continúa siendo uno de los lenguajes más usados[2]:



Rank	Language	Type	Score
1	Python		100.0
2	Java		95.4
3	C		94.7
4	C++		92.4
5	JavaScript		88.1

Ilustración 2: Ranking de los lenguajes más relevantes según el Institute of Electrical and Electronics Engineers (IEEE)

2: [**IEEE Spectrum, Top programming languages 2021.** Disponible en: https://spectrum.ieee.org/top-programming-languages].

El hecho de poder aprender a manejar un lenguaje así de popularizado supone una gran ventaja para un estudiante, pudiendo ser de utilidad para el mundo laboral. Esto es porque se parte de una mejor base y, en caso de acabar utilizando este lenguaje, los conocimientos aprendidos durante su uso serán aplicables. Aún en caso de no utilizarse servirá como ayuda para otros lenguajes o ámbitos de la informática.

2. Al tratar el proyecto del desarrollo de un módulo dentro de un simulador, se podía prever que se trataría de un proyecto muy práctico, enfocado al desarrollo de código. Esto también suponía una motivación por poder experimentar de primera mano el gran campo de la informática que es la programación de software. Esto permitiría un mejor juicio a la hora de decidir el futuro laboral de un estudiante de dos maneras. Ya sea para reconocer la poca motivación de zambullirse en el mundo laboral a través de esta rama o para partir con cierta experiencia en caso de proceder a trabajar en este campo.

3. El tema concreto tratado aquí, la monitorización a fin de lograr una mayor eficiencia, despierta interés debido a la importancia en informática de las buenas praxis y el desarrollo de un código limpio y eficiente. Al tratar de manera directa la eficiencia de un código ya desarrollado se ofrecía la posibilidad de:
    a. Observar cómo se había realizado código de manera correcta. Lo cual supone en sí unas líneas a seguir para desarrollar buenas prácticas.
    b. La posibilidad de mejorar la estructura o comportamiento de dicho código. Resaltando desde un principio la importancia de, no sólo cumplir la función esperada, sino de realizarlo de la mejor manera posible.

4. Frente a otros posibles proyectos este contaba con otra gran ventaja que es su aplicación al mundo de los videojuegos, un mundo que suscita gran interés y deseo por conocerlo. Este es también uno de los motivos más importantes a la hora de haber elegido el proyecto a realizar por los motivos que se plantean a continuación.

El simulador se puede aplicar a muchos ámbitos de estudio o desarrollo, siendo uno de ellos los videojuegos. El mundo de los videojuegos es uno que ha vivido un gran crecimiento en los últimos años como:

- Herramienta de entretenimiento para los usuarios.
- Mercado a explotar por las compañías.
- Mercado laboral activo para trabajadores.

Con la gran oferta laboral que produce, muchas personas se han visto introducidas en él. Esto ha permitido su crecimiento, generando en poco tiempo muchas nuevas formas de desarrollar productos y creando un mercado muy activo.

Otro incentivo a este crecimiento viene dado por la reducción del alto precio inicial de las consolas a lo largo del tiempo.

El mercado de los juegos, en su origen, era un mercado de nicho; una revolución que muchos no recibían con brazos abiertos y que ahora cuenta con una gran base de jugadores. El interés de estos jugadores viene dado por el gran abanico de posibilidades que presentan y el gran impacto que pueden generar avances tecnológicos en este ámbito. Datos más concretos sobre la base de jugadores con la que cuentan se pueden observar en la siguiente imagen:

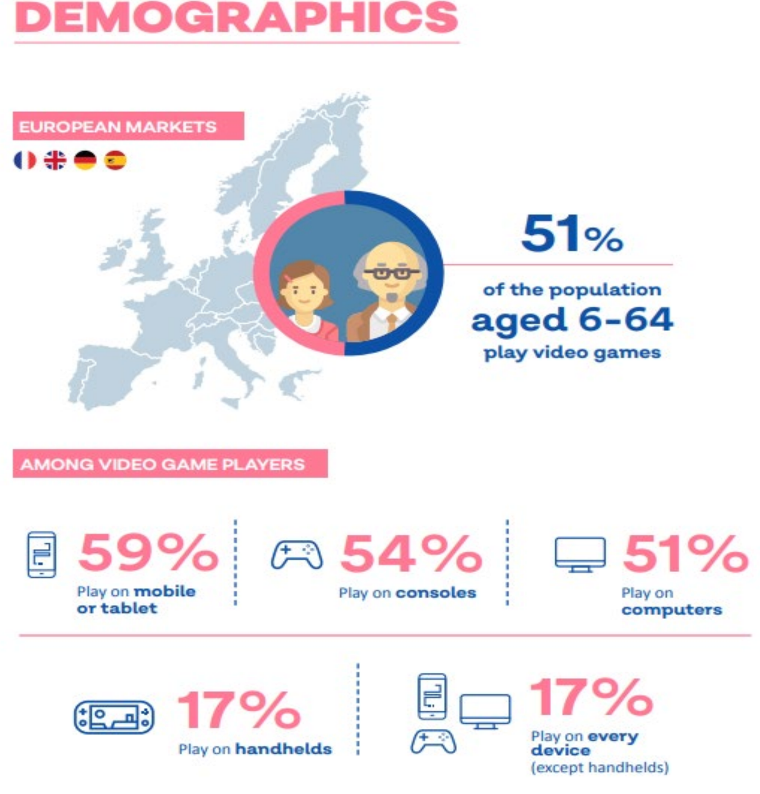


Ilustración 3: ISFE key facts 2020

Según esta información, parte de estudios anuales realizados por ISFE[4], (Interactive Software Federation of Europe) es evidente que una gran parte de la población participa en actividades relacionadas con videojuegos o al menos siente curiosidad por ellos.

Es en gran medida esta curiosidad hacia el mundo de los videojuegos y el cómo funcionan lo que motiva a la producción de este trabajo; tratando de arrojar luz sobre este complejo mundo y aprendiendo durante el desarrollo del trabajo sobre los programas utilizados. Además, se aplicarán a la vez conocimientos aprendidos durante los años de carrera.

La elección de la librería a utilizar viene dada en gran parte por uno de los puntos a favor de los sistemas discretos siendo que las herramientas utilizadas suponen un menor esfuerzo a la hora de ser aprendidas, facilitando la adaptación a ellas y permitiendo más tiempo para el desarrollo (2. García, Mollá y Cabanillas 2002), lo cual ayuda a introducirse en este mundo y aprender de él. Asimismo, como se explicará más adelante, los sistemas continuos poseen diversas desventajas que se evitan al trabajar sobre un sistema discreto.

4: [**ISFE Key Facts. ISFE Key Facts 2020.** Disponible en: https://www.isfe.eu/wp-content/uploads/2020/08/ISFE-final-1.pdf]

1.2 Objetivos

El objetivo principal es diseñar y desarrollar una API de monitorización de rendimiento para videojuegos a partir de la librería de simulación de eventos discretos RT-DESK. Esta monitorización buscará resolver casos de uso concreto para obtener información concreta de utilidad para el motor. A su vez se arrojará luz sobre el funcionamiento de la estructura de RT-DESK. Este monitor contará con funciones como:

- Carga del sistema en un momento dado. Conseguida estudiando los valores de los tiempos de simulación.
- Visualización de la cantidad de eventos diferentes durante la ejecución de la simulación. Gracias al conteo de eventos generados a la hora de simular.
- Memoria ocupada por cada objeto. Estudiando el espacio ocupado por los objetos, en este caso en las piscinas de eventos que forman parte del simulador.

Esta información servirá para tener una idea acertada de posibles áreas de la simulación que puedan hacer cuello de botella, es decir, que supongan un detrimento para el rendimiento general de la simulación. Una vez obtenida esta información será posible el tratamiento de los problemas para darles solución ya que se conocerá, con datos específicos para la situación, cuál es el fallo preciso y dónde se sitúa. Esto permitirá en algunos casos mejorar el rendimiento de la simulación al solventar situaciones de ineficiencia por parte del motor.

Para llegar a este objetivo principal se deben cumplir ciertos subobjetivos parte del proceso general como son:

- El estudio del estado del arte en el ámbito de simulación de videojuegos
- La comprensión de la organización y funciones internas de RT-DESK, siendo este conocimiento necesario para poder trabajar con la librería y hallar los aspectos y funciones susceptibles a ser optimizados
- El planteamiento de casos de uso para valorar la información más importante a obtener
- La resolución de los casos de uso implementando utilidades de monitorización en el motor
- Creación de pruebas unitarias que nos permitan comprobar el correcto funcionamiento de las nuevas utilidades en diferentes situaciones.

1.3 Estructura

Este documento se divide en diferentes bloques conteniendo desde información general hasta más específica a lo largo del desarrollo del objetivo presentado. En concreto se dispone de las siguientes partes:

- Introducción: Explica de manera general el problema global a tratar además del por qué se ha elegido este tema para el proyecto, qué se pretende conseguir a lo largo de su desarrollo y cómo se estructurará.
- Contexto tecnológico: Trata de dar una breve idea de la situación tecnológica actual en el campo de trabajo del proyecto, siendo este la simulación de eventos discretos dentro de programas de simulación. Se da especial importancia a la superioridad en algunos aspectos de la simulación discreta frente a la simulación continua.
- Análisis: En el apartado de análisis se expondrá tanto el entorno de trabajo utilizado para el desarrollo del proyecto como el planteamiento realizado para delimitar las futuras funciones del monitor. Se plantearán casos de uso del programa con información a obtener por nuestro monitor.
- Guía de diseño de la solución: Abarca la idea general y la forma de trabajo que se va a seguir durante el desarrollo de la solución. También contiene pasos previos para ajustar la estructura a las nuevas utilidades.
- Desarrollo de la solución propuesta: En este apartado, a partir de los casos de uso previamente planteados se tratará el desarrollo de estos siguiendo el siguiente orden:
    - Exposición de la manera a proceder para desarrollar el código necesario y comprobación de correspondencia con la solución planteada en el anterior apartado 'Análisis de la solución propuesta'.
    - Desarrollo de dicho código.
    - Implementación de los métodos de comunicación que permitan el intercambio de la información obtenida con estas nuevas utilidades.
    - Creación de pruebas para comprobar el correcto funcionamiento de las nuevas funciones.
- Conclusión: Se hablará sobre los objetivos del trabajo y si se han podido cumplir, aquellos objetivos que no se hayan alcanzado y las herramientas o conocimientos que han sido de utilidad durante el desarrollo.
- Bibliografía : Documentos referenciados para la obtención de cierta información utilizada en la memoria.
- Apéndices: En los apéndices se aclaran conceptos específicos del ámbito tecnológico del trabajo. Estos conceptos intervienen durante el desarrollo del proyecto y pueden facilitar la comprensión de este para lectores no iniciados en el ámbito de trabajo.

1.4 Convenciones

A lo largo de la memoria se van a seguir las siguientes normativas de marcado o convencionalismos:

- Las referencias bibliográficas seguirán la norma ISO-690.
- Se utilizará la cursiva en aquellas palabras que representen anglicismos o nombres de atributos parte del código.
- Las comillas simples se aplicarán a citas textuales, nombres de funciones y extensiones de archivos.
- En cuanto a cuerpo y tipo de letras se seguirán las pautas recomendadas:
  - Tipo de letra: Times New Roman
  - Tamaño de letra:
    - Epígrafes, títulos de capítulos o secciones 16 puntos
    - Cuerpo de la memoria 11 puntos
    - Pies de gráficos y notas 9 puntos
- Inicio de los párrafos con sangría
- Para mejor legibilidad se ha decidido no añadir los argumentos de una función cuando se hace una referencia a la misma.
- En las enumeraciones se utilizarán diferentes símbolos según las extensiones de estas.

2. Contexto tecnológico

‘Computer games follow a scheme of continuous simulation, coupling the rendering phase and the simulation phase. That way of operation has disadvantages that can be avoided using a discrete event simulator as a game kernel’ (1. García, Mollá y Morillo 2004).

Los sistemas o entornos de simulación pueden ser clasificados según diferentes criterios, uno de ellos diferencia entre entornos continuos y discretos. Los sistemas de eventos continuos son aquellos con funcionamiento ‘clásico’ en los que se tiene un flujo de cambios y evoluciones que continúa a lo largo del tiempo. Por otra parte, en los sistemas con eventos discretos estos cambios se dan en instancias concretas del tiempo, en este caso dichas instancias vienen dadas por la marca temporal asociada a cada evento.

Actualmente, la mayoría de los videojuegos son implementados como sistemas continuos, uniendo las fases de simulación y renderizado; esto implica desventajas que se pueden evitar con un sistema discreto. Entre estas desventajas caben destacar (4. Cebrián 2013):

- Posibilidad de simulaciones erróneas: Al depender los objetos de su posición en el grafo de escena y por cómo está organizado éste, se produce una ejecución desordenada de eventos en el tiempo.
- Necesidad de inventarse pasos intermedios ficticios para cubrir la posibilidad de ser utilizados.
- No realimentación de eventos dentro del mismo ciclo.
- Cuando los objetos no generan eventos, también consumen recursos.
- Recorrido obligatorio del grafo de escena que sincroniza todos los objetos, esto implica que:
  - Para objetos de comportamiento lento existirá un sobre muestreo provocando un malgasto en la potencia de cálculo
  - Para objetos de comportamiento rápido existirá un submuestreo provocando un posible error de comportamiento

Los motores basados en el paradigma continuo o tradicional acarrean puntos negativos que son mejor tratados en paradigmas discretos, facilitando un mejor funcionamiento del simulador.

Estas desventajas son relevantes por el hecho de que, en el mundo de los videojuegos, uno de los campos a los cuales se puede aplicar el simulador, la eficiencia a la hora de aprovechar el hardware disponible tiene un gran peso. De hecho, esta eficiencia marca muchas veces el grado de satisfacción con la experiencia de juego. Por este motivo se incentiva a los desarrolladores a tratar de refinar el código para dar un mejor uso de los recursos disponibles en las máquinas.

Para llevar a cabo el refinamiento del código es de gran ayuda una herramienta de monitorización de rendimiento. Esta herramienta permite estudiar mejor cuáles son los puntos por mejorar a la hora de simular o renderizar programas como puede ser un videojuego.

Es de gran importancia también remarcar la diferencia de funcionamiento entre un bucle interno de RT-DESK y otro como puede ser el de videojuegos como Doom o Quake. Para poder observar esta diferencia se va a presentar el esquema por el que se rigen para su comparación. Se ha seleccionado el motor de estos videojuegos por ser ampliamente reconocidos y además tener versiones de su código liberado, para redistribución y modificación bajo licencia GNU. Existen otros motivos por los que se seleccionan estos motores para la comparación como la robustez de

estos títulos con un amplio historial de pruebas y el seguimiento de un paradigma convencional de videojuegos.

En la siguiente ilustración se puede observar el funcionamiento del bucle principal de los motores comentados (7. García G. 2004) :

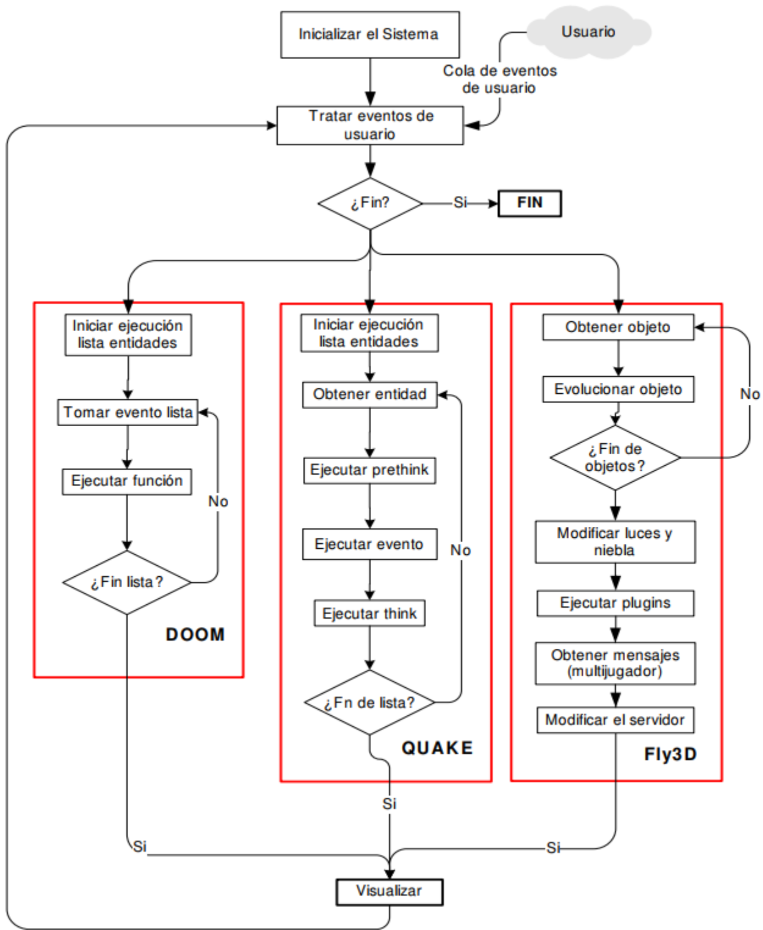


Ilustración 4: Bucle principal de Doom, Quake y Fly3D

El bucle principal de estos videojuegos tiene tres fases (3. García y Mollá 2006):

1. El videojuego recibe los eventos de usuario de la cola y los trata.
2. Fase de simulación: Supone las áreas remarcadas en rojo de la anterior ilustración y consiste en un repaso del grafo de escena en el que se pregunta a los objetos por acciones pendientes que puedan tener. Tras esto los objetos activos son muestreados evolucionando el mundo simulado. Este muestreo de objetos es el que representa que en el sistema se siga un esquema de simulación continua.
3. Visualizar la escena actual. Utilizando el grafo de escena se simula y visualiza la misma.

En contraposición a los motores basados en paradigmas continuos se encuentra el motor RT-DESK, el cual como se ha explicado anteriormente, utiliza un paradigma discreto. El paradigma discreto permite solventar gran parte de los problemas o ineficiencias originados a causa de la programación continua. Por este motivo toma gran importancia este simulador, y es la razón por la cual este trabajo se va a centrar en el refinamiento de varias de sus funciones. Por ello se va a ofrecer una visión más exhaustiva de cómo funciona. A continuación, se muestra cómo se

desarrolla la simulación en RT-DESK gracias a la explicación del bucle principal de dicho programa.

Se puede observar el bucle principal de RT-DESK en la siguiente imagen[5]:

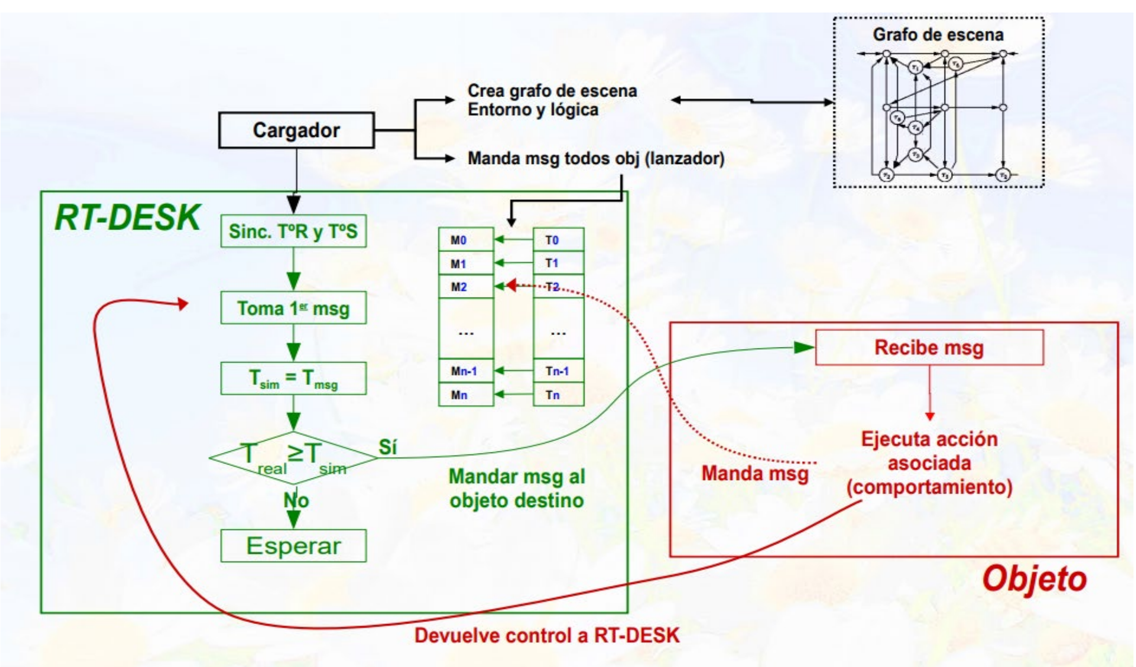


Ilustración 5: Bucle principal de RT-DESK

El bucle principal de RT-DESK consiste en:

1. El cargador crea el grafo de escena, entorno y lógica de la simulación.
2. El cargador manda mensajes a aquellos objetos que lo necesiten, la activación de estos mensajes/eventos depende del tiempo de simulación que en este instante se encuentra congelado. Estos mensajes consisten principalmente en el lanzamiento de la situación inicial para los objetos del universo.
3. Se sincronizan el tiempo real y el tiempo de simulación. El tiempo de simulación será posterior al tiempo real para que cuando se lance la simulación lo haga con el tiempo real.
4. Se toma el primer mensaje con marca temporal igual al tiempo de simulación, es decir, aquel que según su marca temporal debe ser enviado.
5. En este apartado se realiza una comprobación fundamental para el funcionamiento del bucle, se comprueba la relación entre tiempo de simulación (interno del simulador) y tiempo real. Durante la sincronización del motor el tiempo de simulación utiliza como referencia al real, nunca avanzando más que él ya que no debe de anticipar el futuro. Tras realizar esta comprobación se manda el mensaje al objeto destino, el cual tras recibirlo ejecuta la acción asociada y manda el mensaje a su siguiente destinatario.
6. Si el tiempo real no es mayor o igual al tiempo de simulación RT-DESK pasa a la espera.

5: [**RTDESK blogs. Real Time Discrete Event Simulation Kernel, Presentación.** Disponible en: https://rtdesk.blogs.upv.es]

En cuanto a la frecuencia de muestreo del simulador, dentro del apéndice de funcionamiento interno de RT-DESK se puede encontrar más información sobre ella y sobre cómo afecta a las muestras obtenidas.

3. Análisis del problema

En este apartado se hablará de cuál es el enfoque del proyecto, pensado para tratar de dar soluciones a problemas concretos previamente analizados.

El trabajo se enfocará tratando de solucionar casos de uso de los cuales se pueda sacar información útil para tratar alguna forma de ineficiencia o situación no deseable. Por ejemplo, una mala comunicación entre procesos del sistema o una mala organización del tiempo activo de los componentes.

Para ello se necesitará una buena comprensión del funcionamiento y estructura interna de RT-DESK, cómo se organiza internamente, que funciones realiza, etc. Además, se tendrán que plantear las preguntas adecuadas, siendo una pregunta de menor interés menos beneficiosa de contestar que una que pueda tener más peso sobre el rendimiento.

Estos casos de uso se obtendrán planteándose necesidades de conocimiento por parte de los desarrolladores o de uso por parte de los usuarios entre otras. Para este proceso también resultará de utilidad el estudio de otros simuladores tratando de comparar aspectos en común a la hora de la obtención o utilización de valores internos de la simulación.

Las posibles ineficiencias que solucionar o secciones de código que mejorar podrían ser muchas ya que cualquier acción puede acarrear un retraso innecesario de no ser llevada a cabo de la manera óptima. Para reducir esta gran cantidad de posibilidades se plantean los ya nombrados casos de uso. En el proyecto se presentarán casos de uso concretos como el número de eventos de un tipo producidos durante la simulación, así se pueden centrar esfuerzos en dar solución a una ineficiencia concreta.

De antemano no se puede saber cuál será el conjunto de ineficiencias a tratar en el simulador. Por este motivo no se puede hacer aquí una lista detallada, pero tras el estudio de la organización, estructuras de datos y lógica del simulador se podrá saber cuáles son los objetos más afectados. Conociendo estos objetos que conviene monitorizar, se implementarán sobre ellos las funcionalidades extra que permitan obtener los datos necesarios para sacar conclusiones.

3.1 Organización interna de RT-DESK

Para poder realizar un mejor estudio de los datos que se necesitan cuantificar es de gran importancia conocer las características y funciones del motor, así como su organización y clases que lo componen.

RT-DESK tiene como características principales[6]:

- Trabaja en tiempo de simulación sincronizándolo con el tiempo real.
- Es autocontenido, siendo más fácil de integrar en aplicaciones terminadas.
- Los eventos se encuentran ordenados según su marca temporal, la cual representa el instante de tiempo en el que serán ejecutados.
- Emplea un patrón de diseño o fabricación con fábrica de objetos de tipo 'factory method'. En el apéndice se puede encontrar más información sobre la lógica de este patrón de diseño.

Funciones principales:

- Gestiona el ciclo de vida y flujo de los eventos mediante sus estructuras de datos y diversas funciones implementadas:
    - Enviados entre entidades.
    - Entre el almacén de eventos y el sistema.
- Mantiene tanto los mensajes no enviados como los mensajes que todavía no han sido recibidos ordenados por tiempo de ejecución.
- Entrega los mensajes en el periodo de tiempo dictado por la marca temporal.

RT-DESK se organiza según se muestra en el siguiente esquema:

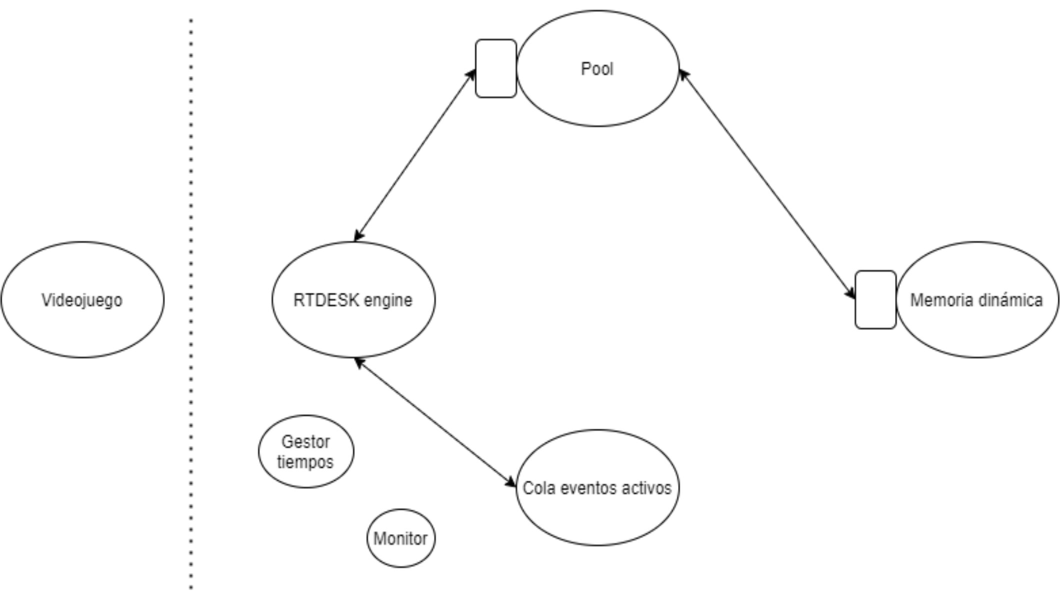


Ilustración 6: Esquema básico de la organización interna de RT-DESK

Según el esquema anterior se pueden distinguir las siguientes partes (3. García y Mollá 2006):

6: [**RT-DESK blogs. Características.** Disponible en: https://rtdesk.blogs.upv.es/caracteristicas/]

- RT-DESK *engine*: Es el recipiente de gran parte de la lógica en el programa como es el comienzo y la finalización de la simulación. También se encarga en su código de incluir y guardar entidades de gran importancia como son el *dispatcher* y el *pool manager*. El objeto *dispatcher* se encarga de:
    - Apoyar en el proceso de paso de mensajes
    - Estar pendiente de los mensajes ordenados por su marca de tiempo para cuando llegue el momento en que deban de ser enviados
    - Enviar mensajes al destinatario correspondiente en el momento adecuado, según dicte el objeto emisor del mensaje.

  Por otra parte, *pool manager* se encarga de la gestión de *pools* y sus atributos.
- *Pool*: La memoria dinámica es llamada cada vez que se necesita un nuevo evento en el sistema. Cuando el evento haya terminado su función y deba salir del sistema la memoria dinámica será llamada de nuevo para destruirlo. Esto es ineficiente por lo que se necesita el *pool* para almacenar aquellos eventos que no se encuentren actualmente en el sistema, es decir, que se encuentren inactivos. De manera que, cuando un evento sale del sistema, en vez de ser destruido pasa a ser almacenado. Esto permite que si el sistema vuelve a requerir de ese evento no tendrá que crearlo y no se deberá llamar a la memoria dinámica.
- Cola de eventos activos: Sirve para almacenar aquellos eventos que se encuentren actualmente en uso en el sistema.
- Memoria dinámica: Es necesaria a la hora de proporcionar eventos al sistema que no se encuentren en el *pool*.
- Gestor de tiempos (temporizadores): Son objetos encargados de realizar mediciones tanto en *ticks* del sistema como en milisegundos. Permiten la devolución de períodos invertidos en realizar dichas mediciones y los correspondientes tiempos medidos entre otros. Además de conteos pueden realizar funciones de alarmas.

En cuanto a la organización del código se puede encontrar más información sobre su estructura, relación de inclusión entre cabeceras y funcionamiento por clase en el apéndice.

3.2 Análisis de requisitos

Para poder enfocar mejor los campos a tratar se plantean casos de uso en los cuales se necesita información a aportar por los objetos de monitorización; de esta forma se podrá saber exactamente que se quiere monitorizar. Durante este proceso se tendrá en cuenta tanto el papel de desarrollador como el de usuario final de la aplicación, dado que según el perfil del usuario sus necesidades cambian y la información deseada puede no ser la misma.

Para ello, primero se hará un listado de posibles preguntas sobre valores internos del simulador que podrán ser obtenidos por el monitor. De este listado de preguntas las consideradas de mayor relevancia pasarán a ser resueltas. La resolución consistirá en que los objetos encargados de monitorizar obtengan la información que solicitan las preguntas. Estos nuevos objetos manejarán los valores de interés a través de mensajes/eventos.

A fin de plantearse las preguntas necesarias, se ha de pensar sobre aspectos con impacto sobre el rendimiento de la simulación, los recursos más interesantes en este aspecto son los mensajes o eventos. Se generan gran cantidad de eventos de diferentes tipos durante el transcurso de la simulación, cargando con importantes funciones y consumiendo recursos de la máquina en el proceso. Por este motivo conocer valores relacionados con la generación y comportamiento de eventos ayudará en gran medida a obtener datos de interés.

Por otra parte, se encuentra la memoria dinámica la cual juega también un papel importante sobre el rendimiento, pudiendo generar tiempos de espera si no se gestiona de manera adecuada. Es por ello por lo que se plantearán preguntas sobre sus tiempos de espera, así como de tiempos de espera de manera global durante la simulación además de aspectos generales sobre la inicialización del cargador.

Entre las preguntas a plantearse se tienen:

1. ¿Cuántos eventos se han producido durante la simulación?
2. ¿Cuántos eventos de cada tipo se han producido?
3. ¿Cuál es el tiempo total de espera a la memoria dinámica?
4. ¿Existen tiempos de espera fuera de la memoria dinámica? ¿Dónde se producen y cuál es la suma total de tiempos de espera?
5. ¿Cuál es el tiempo medio que transcurre desde que un mensaje es enviado hasta que ejecuta su función?
6. ¿Cuánto tiempo se invierte en la generación y gestión de eventos?
7. ¿Cuánto tiempo tardan en mandarse los mensajes de inicialización?
8. Cantidad media y picos de memoria ocupada
9. ¿Cuándo se manda el primer mensaje fuera de la inicialización?
10. ¿Cuánto tarda el cargador en realizar la inicialización previa al envío de mensajes?
11. Una vez un temporizador haya adquirido información solicitada ¿cuánto tiempo se invierte en mandar esta información a su destinatario final?
12. ¿Reusabilidad de mensajes/funciones?
13. ¿Media y desviación media de mensajes enviados por iteración del simulador?
14. Diferencia máxima entre tiempo de simulación y tiempo real
15. Número de eventos recibidos por un tipo de objeto

Estas preguntas han sido planteadas gracias al estudio de diferentes manuales de simuladores (9. Bucy et al. 2008) (10. Huang et al. 2014) que han ofrecido una visión más amplia con ejemplos concretos de datos de interés para los simuladores. También ha sido de ayuda el repaso sobre RT-DESK y su código, valorando las entidades y atributos que maneja y estudiando cuáles de ellas nos pueden ofrecer o poseen valores de utilidad.

Para responder a las preguntas es necesario añadir funcionalidades y clases extra a RT-DESK que nos permitan obtener información durante la ejecución de cualquier simulación. Tras la resolución de los casos de uso, se agrupará la información para poder ser estudiada con mayor facilidad. En los próximos apartados se tratará el desarrollo del código necesario para poder resolver los casos de uso planteados, estudiando previamente cómo interesa resolver cada uno de ellos.

3.3 Identificación y análisis de soluciones posibles

De las preguntas planteadas anteriormente se realiza una criba excluyendo en el proceso aquellas que resulten de menor interés, posean un nivel de complejidad demasiado elevado o no resulten de utilidad para el simulador empleado, entre otras.

Quedarán descartadas las siguientes preguntas :

12. ¿Reusabilidad de mensajes/funciones?

Siendo el propósito principal de este trabajo la obtención de información para uso futuro, se considera esta pregunta demasiado compleja por lo que a pesar de abarcar el aprovechamiento de recursos se decide excluirla de las preguntas de interés.

6. ¿Cuánto tiempo se invierte en la generación y gestión de eventos? y 4. Dónde se producen otros tiempos de espera y su suma total.

La respuesta a estas preguntas necesita de información durante diferentes momentos de la simulación y de distintos sectores del simulador, convirtiéndose en una pregunta con muchos factores a tener en cuenta. Además, su respuesta aporta una información muy general por lo que se opta por excluirla. Existe otro caso de uso que estudia la espera a la memoria dinámica durante la creación de eventos, lo cual supone obtención de información más concreta en cuanto al tiempo invertido en la generación de eventos.

10, ¿Cuánto tarda el cargador en realizar la inicialización previa al envío de mensajes? y 9. ¿Cuándo se manda el primer mensaje fuera de la inicialización?

Dentro del proceso de inicialización del simulador, se considera más viable hallar y resolver problemas u optimizar rendimiento en el apartado de envío de mensajes iniciales que en los procesos previos. Por ello se decide cerrar esa posibilidad y centrar esfuerzos en el envío de mensajes dentro de la fase de inicialización. Por estos motivos no es de interés recabar información como la duración en el tiempo de esta primera fase.

13. ¿Media y desviación media de mensajes enviados por iteración del simulador?

Esta información puede ser muy útil para observar el estado de ociosidad del simulador en todo momento, pudiendo conocer picos de envíos de mensajes y momentos en el programa en que haya que equilibrar la carga. A pesar de la utilidad de esta información, se considera fuera del alcance de este proyecto, ya que es más viable de primera mano estudiar la cantidad de mensajes en diferentes momentos de la simulación.

7. Tiempo que tardan en mandarse los mensajes de inicialización y 8. Cantidad media y picos de memoria ocupada

La información sobre memoria es obtenida ya en el proyecto *GameOfLife* midiendo el tiempo transcurrido durante diferentes acciones de la aplicación mediante varios temporizadores como 'RTDSKMM_TIMER'. Este temporizador se encarga de controlar el manejo de mensajes y el tiempo de gestión de la memoria. El cálculo ya existente proporciona el % de memoria ocupado, así que no se trata de la misma información. Aun así, la forma de obtención de esta puede ser de ayuda para obtener la información deseada en el caso de uso. Su uso se puede observar en la siguiente ilustración:

```
#ifdef DEF_RTD_TIME
TM.Timers[RTDSKMM_TIMER].InitCounting();
#endif

msg = (cMsgCel*) GetMsgToFill(RTDESKMSG_CELDAS);
msg->msgSubType = MSG_SETDORMIDA;
SendSelfMsg(msg,1.0);

#ifdef DEF_RTD_TIME
EndAccCounting(RTDSKMM_TIMER);
```

Ilustración 7: Fragmento de código del proyecto 'DemoGameOfLifeRTDesk

En ella se observa que se inicializa el conteo antes de la acción o acciones a realizar y se termina al finalizar las acciones a medir. El procedimiento para medir el tiempo invertido en los mensajes de inicialización sería similar al de la ilustración. Al ya encontrarse aquí implementado y para evitar cargar de funcionalidades al monitor, se decide no implementar esta funcionalidad en la versión presente. En caso de querer implementarse bastaría utilizar la estructura de temporizadores ya existentes en RT-DESK junto a la estructura de paso de mensajes implementada.

11. Tiempo invertido en mandar información solicitada a su destinatario final y 5. Tiempo medio que transcurre desde el envío de un mensaje hasta su ejecución

Esta información puede resultar útil para:

- Detectar si existe un retraso a la hora de ejecutarse un mensaje
- Comprobar el tiempo general invertido mandando mensajes, viendo si el tiempo transcurrido hasta la ejecución de su orden corresponde con el tiempo planificado.

A pesar de esto, tras considerar la manera en que se realiza el envío de mensajes dentro del simulador, se opta por no desarrollar estos casos de uso. El motivo es que el tiempo invertido en el envío de un mensaje a partir de su marca temporal es despreciable en situaciones normales. De no ser así existen otras posibilidades

preferibles para el estudio de sobrecarga. Por otra parte, como se ha explicado anteriormente, la cantidad de mensajes enviados depende mayormente de la simulación lanzada, por lo que el tiempo invertido mandando mensajes dependerá mayormente de ésta.

A pesar de que este proyecto no abarque el lanzamiento de las nuevas utilidades dentro de una simulación, se ofrece un punto de vista que puede ayudar a la hora de en un futuro lanzarlas.

Como entorno de simulación para comprobar la respuesta a estas preguntas ya filtradas existirían infinidad de posibilidades. Esto es debido a que existen pocos requisitos necesarios para poder obtener la información deseada, entre ellos: generación de eventos por parte del programa, que de estos eventos existan diferentes tipos, que se haga uso de la memoria dinámica, etc.

Al no poder filtrar los posibles entornos de simulación a través de los requisitos, se debería diferenciar entre las posibles soluciones por complejidad del programa, conveniencia y atractivo visual.

En cuanto a la complejidad, se verían favorecidos aquellos programas cuya simulación sea fácilmente comprensible, pero a la vez aportasen valor al proyecto, es decir entornos que no tratasen campos avanzados y a la vez fuesen fácilmente diferenciables de otros.

Por conveniencia ganarían aquellos programas sobre los cuales se puedan realizar modificaciones a fin de conseguir la información deseada, aprovechando código y simulaciones con RT-DESK ya incorporado.

En cuanto al atractivo visual, se necesitaría de un programa cuya ejecución mostrase información de manera clara y concisa al usuario que la ejecute, es decir, que permitiese ver como la simulación avanza mientras se recaban los datos buscados.

Teniendo en cuenta todos estos factores, se cree que para en un futuro realizar una simulación con el nuevo motor, a fin de obtener datos, se debería escoger como simulación una versión del juego matemático ‘El juego de la vida’. Este juego ya se encuentra implementado en RT-DESK.

3.4 Análisis de la solución propuesta

Para obtener información útil durante la simulación primero se debe plantear qué información concreta es la que se desea obtener y tras ello cómo se pretende obtenerla. En esta situación es cuando resulta de utilidad el planteamiento de los casos de uso, preguntas cuya respuesta se considera información valiosa de utilidad para la simulación.

Será sobre la resolución de estos casos de uso sobre lo que trataran los próximos apartados, tratando que las modificaciones sobre el código de la librería RT-DESK sean lo menos disonantes posibles y siguiendo las reglas de lo previamente implementado. Además, según se realicen modificaciones al código, se acompañarán de comentarios explicativos para que su comprensión sea más fácil. Estos comentarios serán de gran ayuda para modificaciones futuras a realizar por otros desarrolladores, que podrán comprender el código sin trabas mayores.

Para una mejor comprensión de cómo se organizan y relacionan las clases de RT-DESK entre ellas se presenta el siguiente esquema. En él se representan las relaciones de inclusión entre las diferentes clases cabecera pertenecientes a RT-DESK. Esto es, muestra aquellas clases '.h' que hacen uso de otras habiendo lanzado la directiva 'include' previamente para poder utilizar las funciones y datos que ofrece:

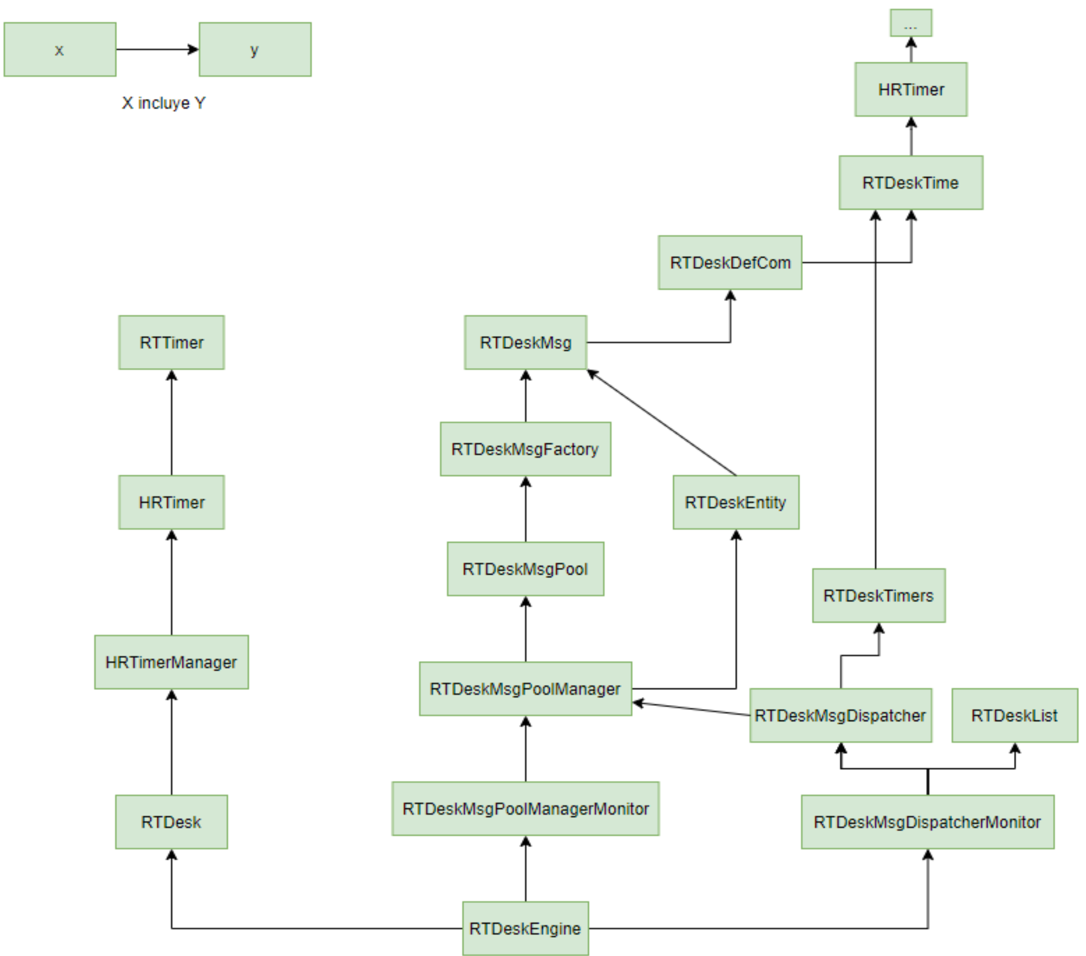


Ilustración 8: Esquema de relaciones entre cabeceras en RT-DESK

En el esquema se puede observar que, según esta organización, *RTDeskEngine.h* se encuentra como la más alta en esta jerarquía; esto es porque es la clase encargada de gestionar los objetos *dispatcher* y *pool manager.* Estos objetos a su vez son los encargados de la mayoría de las funciones llevadas a cabo durante las simulaciones. Por otra parte, *RTDeskEntity.h* también es de gran importancia para estos dos objetos ya que clases necesitan heredar la cabecera *entity* para poder realizar funciones de envío/recepción de mensajes.

Para un objeto siempre habrá un archivo '.h' contenido en *Header Files,* pero puede no existir su pareja de archivo con formato '.cpp' en *Source Files* esto es porque un objeto siempre tendrá características a definir, pero no es indispensable que tenga funciones a implementar.

Se puede observar en la siguiente ilustración una carpeta para dependencias externas del simulador y, por otra parte, las carpetas *Header Files* y *Source Files*. Estas carpetas contienen definiciones de funciones/atributos de los objetos y sus implementaciones respectivamente.

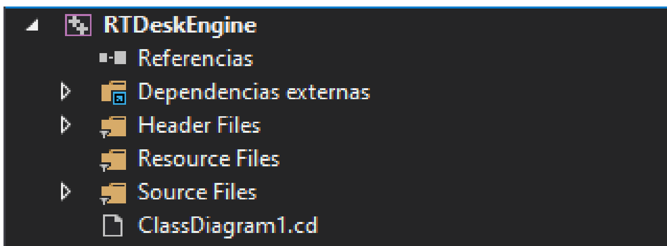


Ilustración 9: Vista general RTDeskEngine

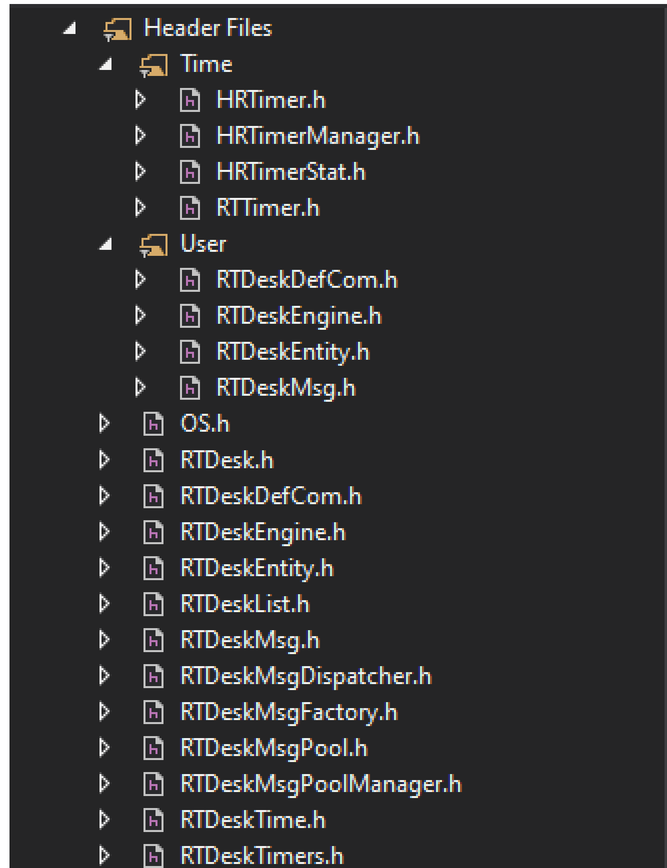


Ilustración 10: Header Files RTDeskEngine

Time, contiene clases relacionadas con la gestión del tiempo*:*

- *HRTimer*: Es la clase cabecera para los relojes de alta resolución. Define constantes que ayudan a conocer el estado de los temporizadores. También define un reloj *cHRTimer* de tipo *cRTTimer* aportando:
    - Atributos de rendimiento derivados de cálculos estadísticos
    - Métodos de manejo de contadores
    - Métodos de ajustes varios para el reloj y sus atributos.
- *HRTimerManager*: Se encarga de gestionar los relojes definiendo acciones de configuración como cambios en la frecuencia de muestreo y del nombre del reloj. También define posibles errores que pueden ocurrir al manejar los relojes y otras funciones relacionadas con los mismos como creación, destrucción y reinicio.
- *HRTimerStat*: Actúa como cabecera de los relojes de alta resolución con análisis estadísticos. Contiene una clase *cHRTimerStat* de tipo *cHRTimer* con atributos relacionados con el rendimiento y funciones relacionadas con el conteo.
- *RTTimer*: Cabecera para los relojes de tiempo real. Contiene atributos y funciones necesarias para dichos relojes. Estos relojes para obtener su medida de tiempo aprovechan el hardware de la máquina y utilizando la función rdtsc obtienen el tiempo real con una gran precisión.

Las clases contenidas en *User* son copias de algunas en *Header Files,* esto es debido a que aquí se contienen clases para el desarrollo, que pueden ser exportadas para su uso en otros proyectos. *User* contiene:

- *RTDeskDefCom*: Definiciones y constantes comunes para RT-DESK.
- *RTDeskEngine*: Núcleo principal de RT-DESK.
- *RTDeskEntity*: Declaraciones sobre funciones de envío y recepción de mensajes.
- *RTDeskmsg*: Interfaz para la clase mensaje. Contiene tanto la declaración de los atributos necesarios para los mensajes como la declaración de métodos de inserción.

Header Files, cabeceras principales del programa*:*

- *OS*: Definición del sistema operativo en el que la API está funcionando.
- *RTDesk*: Cabecera general para cargar definiciones.
- *RTDeskDefCom*: Definiciones y constantes comunes de RT-DESK.
- *RTDeskEngine*: Núcleo principal de RT-DESK.
- *RTDeskEntity*: Declaraciones sobre funciones de envío y recepción de mensajes.
- *RTDeskList*: Lista de mensajes. Declaración de listas y métodos para manejo de mensajes dentro de ellas.
- *RTDeskMsg*: Interfaz para la clase mensaje. Contiene declaraciones de atributos y funciones importantes para el funcionamiento de la lógica de los mensajes.
- *RTDeskMsgDispatcher*: Alberga diferentes atributos y funciones relacionados con la gestión del tiempo y de los relojes para gestionar el movimiento de mensajes para el cual también posee funciones declaradas.
- *RTDeskMsgFactory*: Proporciona declaraciones de métodos para crear mensajes de usuario. Estos métodos deben ser implementados en una clase hija para proporcionar la funcionalidad completa.
- *RTDeskMsgPool*: Un pool para la interfaz de la clase *RTDeskMsg* y sus definiciones.

- *RTDeskMsgPoolManager*: Similar a la anterior, posee declaraciones sobre funciones de manejo y edición de atributos de mensajes.
- *RTDeskTime*: Descripción de la definición global de como representar el tiempo en RT-DESK.
- *RTDeskTimers*: Descripción de los contadores globales de rendimiento necesarios para realizar test.

Por otra parte, en *Source Files* se tiene:

Source Files
Time
HRTimer.cpp
HRTimerManager.cpp
HRTimerStat.cpp
RTDeskEngine.cpp
RTDeskEntity.cpp
RTDeskList.cpp
RTDeskMsg.cpp
RTDeskMsgDispatcher.cpp
RTDeskMsgPool.cpp
RTDeskMsgPoolManager.cpp
RTDeskTimers.cpp

Ilustración 11: Source files RTDeskEngine

Dentro de *Time* se tiene la implementación de aquellas clases relacionadas con la gestión del tiempo:

- *HRTimer*: Implementa la obtención y cambio de nombre del reloj, la actualización de atributos privados y el reinicio de la frecuencia de muestreo con la posibilidad de cambiar su valor.
- *HRTimerManager*: Métodos para el gestor de los relojes. Contiene diferentes funciones relacionadas con la frecuencia de muestreo, tanto su modificación como la obtención de su valor. También implementa el cambio de nombre de los temporizadores, su creación y su obtención.
- *HRTimerStat*: Declaraciones para los relojes de alta resolución. Implementa funciones como recalcular valores de constantes internas, reiniciar contadores y valores estadísticos, extracción de valores dentro de arrays propios, etc.

Source Files, implementación de las clases principales del programa:

- *RTDeskEngine*: Realiza diversas funciones, algunas de ellas:
  - Controla la versión de la librería, subversión y revisión
  - Controla el flujo de la simulación, pudiendo iniciarla, terminarla y/o reiniciarla
  - Permite establecer el máximo tipo de mensajes diferentes posible y cambiar el tipo de mensaje de un pool
- *RTDeskEntity*: Contiene declaraciones básicas relacionadas con el envío/eliminación de mensajes.
- *RTDeskList*: Posee funciones para la gestión de mensajes en colas.
- *RTDeskMsg*: Al igual que *List* posee funciones para la gestión de mensajes en colas.

- *RTDeskMsgDispatcher*: Es el encargado principal del envío, almacenamiento y eliminación de mensajes de RT-DESK. Además, contiene la función que inicia la simulación y la función de sincronización del tiempo de simulación con el tiempo real.
- *RTDeskMsgPool*: Definición de métodos para el pool de mensajes. Permite obtener el primer mensaje de la pila, generar nuevos mensajes, validar el pool realizando test de integridad y eliminar mensajes.
- *RTDeskMsgPoolManager*: Clase envoltorio de *Pool Manager* con definición de métodos.
- *RTDeskTimers*: Variables globales de relojes necesarias para RT-DESK.

Habiendo planteado el entorno y la información a obtener de él, queda por estudiar cómo se obtendrá dicha información durante el programa. Previo a la implementación, se presenta en la siguiente tabla el procedimiento que se piensa seguir para responder cada una de las preguntas durante la ejecución del simulador. Existe la posibilidad de que el curso de acción a seguir presentado aquí diste en algunos casos de la implementación durante la fase de desarrollo. Por lo tanto, esta tabla sirve como un análisis teórico previo, cuyo planteamiento puede ver trabas o ser modificado a la hora de ser implementado en el código:

Preguntas	**Método de resolución**
1. Eventos producidos durante la simulación **2. Eventos producidos de cada tipo**	Se puede obtener información sobre generación de eventos gracias a la clase *RTDeskMsgDispatcher*, ya que ella se encarga de gestión de eventos/mensajes. Además, por cómo funciona el paso de mensajes, en caso de que algún objeto se envíe un mensaje a sí mismo el *dispatcher* es la única clase donde quedaría reflejado. 1. Para saber la cantidad de eventos producidos durante la simulación, se ha de estudiar el número de mensajes que ha mandado el *dispatcher* durante la simulación. Esto podría hacerse incorporando una estructura personalizada compuesta por dos atributos, tipo y contador. Con esta estructura en la función de insertar mensaje se podrá, al llegar un mensaje, obtener su tipo y buscar la coincidencia en nuestra estructura para poder aumentar el contador correspondiente. 2. En cuanto al número de eventos producidos de cada tipo, se pretende utilizar un vector de booleanos. El vector permitirá dejar registro de los diferentes tipos de eventos que se vayan produciendo durante la simulación.
3. Tiempo de espera a la memoria dinámica durante la creación de eventos	3. Para el estudio de la espera a memoria dinámica se realizarán mediciones antes y después de la generación de un nuevo mensaje por parte de la fábrica *RTDeskMsgFactory*. Esta llamada ocurre cuando *RTDeskMsgPool* llama a la función 'pop()' la cual devuelve el primer mensaje de la cola desplazándose a la vez al siguiente mensaje de la cola. Además, en caso de no haber más mensajes, llama a la función 'GenerateNewMsgs()' que entre otras pide a la fábrica la creación de un nuevo mensaje a través de la función 'CreateNewMsg()'.
16. Diferencia máxima entre tiempo de simulación y real	A fin de resolver este caso de uso se necesitará acceder al valor de los relojes *SimulationClock* y *SystemClock* los cuales representan el tiempo de simulación y el tiempo real respectivamente. Estos relojes se encuentran situados en el objeto *dispatcher* y el objeto *manager*.

	Para poder averiguar la diferencia máxima entre ellos durante la simulación se pretende implementar en estos objetos una función que permita comparar dichos valores y guardar su diferencia. Esta diferencia se guardará si es mayor a la diferencia máxima previamente guardada.
17. Número de eventos recibidos por un objeto	Para conocer esta información se ha de localizar o la función encargada del envío de un evento a su destinatario o la función encargada de que dicho destinatario reciba su mensaje. Para el trabajo actual se decide utilizar la función encargada de enviar el evento. En concreto, la función 'DispatchMsg()' del objeto *dispatcher.* Una vez localizada la función se plantean dos posibilidades: 1. Implementar una estructura de conteo similar al primer caso de uso, asignando un contador a cada objeto y aumentándolo al recibir éste un evento 2. Implementar la monitorización del número de eventos recibidos por un objeto concreto a partir de la recepción de un mensaje indicando el objeto del cual monitorizarlo.

4. Guía de diseño de la solución

Previo al desarrollo de funciones y utilidades es importante recalcar que durante el desarrollo del monitor se tratará de hacerlo de la manera lo menos intrusiva posible, tratando por ejemplo de tener el máximo posible de funciones 'inline' o insertadas. Estas funciones[7] pueden agilizar la ejecución del programa ya que eliminan la sobrecarga asociada a las llamadas a funciones normales. Además, las funciones insertadas se benefician de optimizaciones de código de las cuales las funciones normales no pueden beneficiarse. Estas funciones son por lo tanto de gran utilidad para funciones pequeñas como obtención de atributos y devolución de información. En la siguiente ilustración se puede observar la diferencia entre una función inline y una que no lo es:

```
inline void ShutDown    (){Reset();}
void        Delay       (double lMilliseconds);
```

```
133     */
134     void RTDESK_CEngine::Delay(double lMilliseconds)
135
136     {
137         SystemClock->SetAlarm(lMilliseconds);
138
139         //Time polling. Wait until the alarm sounds
140         while (!SystemClock->IsSounding());
141     }
```

Ilustración 12: Diferenciación funciones inline/no inline

En la ilustración se observa la función 'ShutDown()' definida como inline y la función 'Delay()' junto con su implementación en el archivo '.cpp'. 'ShutDown()' se traducirá en tiempo de compilación a una llamada a 'Reset()', mientras que cuando se llame a la función 'Delay()' se tendrá que buscar su implementación en el fichero '.cpp' y después ejecutarla.

Ahora se ha de plantear la estrategia general que se seguirá para desarrollar la solución de cada uno de los casos de uso. Tras descartar otras posibles soluciones, desarrolladas en el apéndice, se ha decidido afrontar el problema de la siguiente manera:

Actualmente en RT-DESK los objetos que necesiten de un *MsgDispatcher* o *MsgPoolManager* poseen un puntero al mismo heredado de *entity*. Tras las modificaciones a realizar en este apartado:

- Se dispondrá de dos versiones de estos objetos, la estándar y la preparada para monitorización.
- Los objetos no poseerán referencias a *Dispatcher* o *PoolManager*, sino que tendrán una referencia a *Engine*.

7: [**Microsoft. Microsoft docs, Funciones insertadas** 09/10/2018. Disponible en: https://docs.microsoft.com/es-es/cpp/cpp/inline-functions-cpp].

- En *engine* se guardarán un vector de *Dispatcher* y uno de *Manager* para almacenar las diferentes versiones de los objetos.
- También en *engine* se tendrá un puntero para el *Dispatcher* en uso y otro para el *PoolManager* en uso. Se puede observar la definición de estos objetos en la siguiente ilustración:

```
//Vector to store both PoolManager and MsgDispatcher versions, monitored and not monitored
std::vector<RTDESK_CMsgPoolManager*>    PoolManagers;
std::vector<RTDESK_CMsgDispatcher*>     MsgDispatchers;

//Current Message Pool Manager and Dispatcher
RTDESK_CMsgPoolManager* MsgPoolManager;
RTDESK_CMsgDispatcher*  MsgDispatcher;
```

Ilustración 13: Nueva estructura en RTDeskEngine.h

A pesar de esta nueva estructura de objetos no será necesario sustituir las referencias directas a *Dispatcher* o *PoolManager.* Esto es porque, ya que *entity* pasa a heredar de *engine*, cualquier objeto considerado entidad continuará teniendo estas referencias a través de *engine*. Cuando se deba producir un cambio de versión, por ejemplo, por la llegada de un mensaje que así lo indique, simplemente se cambiará a que objeto de los vectores de almacenamiento apunta el puntero de objeto activo.

Esta nueva forma de trabajo supone diversas ventajas frente a la alternativa de implementar una compilación condicional mediante directivas '#ifdef':

- Permite cambiar cómodamente y en cualquier momento entre versiones monitorizadas y no monitorizadas de estos objetos. Al no utilizar una compilación condicional se podría cambiar el estado de monitorización en tiempo de ejecución gracias a la recepción de mensajes por parte del motor. Por otra parte, la compilación condicional fuerza al programa a iniciarse o bien en modo monitorizado o sin monitorizar, solo pudiendo cambiar el modo previo a la compilación.
- Se evita tener código duplicado dentro del proyecto. Aquellas funciones o atributos que puedan ser reutilizados por las versiones monitorizadas serán heredados directamente del objeto original. Por otra parte, en la implementación de compilación condicional el código original de los ficheros '.cpp' a monitorizar se encuentra replicado. Aunque esto no suponga un detrimento para el rendimiento del programa, pues solo una de las implementaciones de los objetos será compilada, supone una mayor cantidad de código escrito en el proyecto. Estos ficheros extra causarían que el proyecto pesase más y se convirtiese en una construcción farragosa, enmascarando lo que realmente estaba ocurriendo.
- Disponer de una estructura de almacenaje de *dispatcher/pool manager,* como son los vectores por declarar en *engine,* puede suponer un mayor nivel de complejidad que el programa base. A pesar de esto, una vez superada esta primera fase de desconocimiento se abren diversas posibilidades, permitiendo con mayor facilidad la implementación de nuevos módulos y/o funcionalidades. Además, resultará más cómodo el desarrollo de estas nuevas partes.
- En cuanto a las estructuras de datos presentes en el simulador, según como se van a organizar las clases de los objetos implicados, se mantendrá el estado de dichas estructuras. D esta manera, aún después de cambiar entre modo no monitorizado y

monitorizado no se perderán datos relevantes de la simulación. Esto es debido a que los atributos y estructuras base son heredados por las clases monitorizadas y por lo tanto los valores almacenados no se pierden. La referencia a los datos es la misma, se trate de un objeto monitorizado o no. Naturalmente esto implica que aquellos atributos declarados dentro de las clases monitorizadas sí sean exclusivos de las mismas. Esto no supone un problema ya que dichos datos no eran necesarios desde un principio para la ejecución normal del simulador.

La versión preparada para monitorización consistirá en una clase que herede de la versión estándar, es decir, una clase 'hija' que contendrá:

- Las funciones y atributos extra añadidos a fin de contribuir en el proceso de monitorización
- Los atributos y funciones no modificados respecto de la clase original, los cuales no es necesario hacer explícitos en la clase hija ya que son heredados.

Los objetos o funciones declarados como *private* en la clase padre no podrían ser utilizados por sus clases derivadas o hijas. En este caso no hay código en la clase *dispatcher* declarado como privado por lo que no se produce pérdida de información en el proceso de herencia.

A fin de ofrecer una visión más clara del cambio de estructura se ofrece la siguiente ilustración donde queda resumida:

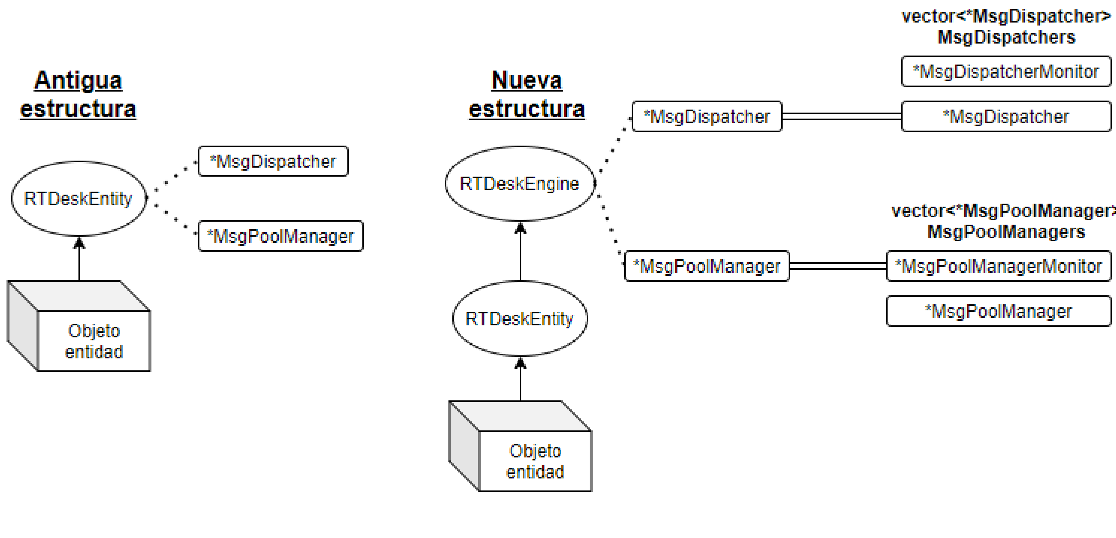


Ilustración 14: Cambio de estructura en RT-DESK

Para completar el proceso de creación de un nuevo *dispatcher,* un nuevo *pool manager*, el manejo de estos por parte del *engine* y otras preparaciones necesarias para la nueva estructura se habrán de hacer los siguientes cambios:

1. Crear las clases *RTDeskMsgDispatcherMonitor.h*, *RTDeskMsgDispatcherMonitor.cpp*, *RTDeskMsgPoolManagerMonitor.h* y *RTDeskMsgPoolManagerMonitor.cpp.*

2. Modificar las relaciones de inclusión por parte del *engine* para adaptarse a la nueva situación, incluyendo nuevos ficheros como son el *pool manager* y *dispatcher* monitorizados.
3. Declarar en *engine* dos vectores que contendrán punteros a *pool manager* y *dispatcher* respectivamente. Estos vectores serán los encargados de almacenar dichos objetos con el fin de facilitar el cambio de modo de funcionamiento del motor. Se mantendrán los punteros originales que apuntan al objeto en uso:

```
std::vector<RTDESK_CMsgPoolManager*>    PoolManagers;
std::vector<RTDESK_CMsgDispatcher*>     MsgDispatchers;

//Current Message Pool Manager and Dispatcher
RTDESK_CMsgPoolManager* MsgPoolManager;
RTDESK_CMsgDispatcher*  MsgDispatcher;
```

Ilustración 15: Declaración de nuevas estructuras de datos en engine

4. Después, se han de modificar los métodos presentes en engine para que actúen en base a esta nueva estructura de vectores. Se han de modificar, o crear de no existir previamente, las funciones: 'SetMsgDispatcher()', 'GetMsgDispatcher()', 'SetPoolManager()' y 'GetPoolManager()' así como la implementación a dichas funciones en el fichero *RTDeskEngine.cpp*.
5. La función 'Simulate()' de *engine,* previamente vacía, apunta ahora a la función de mismo nombre en *dispatcher* que se encarga del envío de mensajes y de realizar mediciones de tiempos.
6. En *PoolManager* se realizará también un pequeño cambio en la estructura de datos. Se definirá un vector de punteros a *Pools* frente al vector de *Pools* que existía previamente, con los cambios menores en funciones que esto implica.
7. Preparar, en los objetos monitorizados, las definiciones de aquellas funciones a incorporar o a modificar respecto de la implementación original, como es el caso para la función 'pop()' en *PoolManagerMonitor*.
8. Definición de constantes en el fichero 'RTDeskDefCom.h'. Su utilidad consiste en almacenar los valores numéricos para marcar la posición del objeto no monitorizado/monitorizado en los vectores almacén y el tamaño de estos. Su uso se puede observar en la siguiente ilustración:

```
PoolManagers.resize(RTDESK_TOTAL_MON);
MsgDispatchers.resize(RTDESK_TOTAL_MON);

PoolManagers[RTDESK_NO_MONITORING] = new RTDESK_CMsgPoolManager();
MsgDispatchers[RTDESK_NO_MONITORING] = new RTDESK_CMsgDispatcher();

PoolManagers[RTDESK_MONITORING] = new RTDesk_CMsgPoolManagerMonitor();
MsgDispatchers[RTDESK_MONITORING] = new RTDESK_CMsgDispatcherMonitor();
```

Ilustración 16: Uso de constantes en RTDeskEngine.cpp

9. La ilustración corresponde con el código de la función 'Startup()'. Esta función crea los vectores almacén de tamaño *RTDESK_TOTAL_MON* y crea el *manager* no monitorizado

y el monitorizado en las posiciones cero y uno respectivamente. La creación de los *dispatcher* sigue el mismo proceso.

10. Cambios en el funcionamiento de *engine;* en concreto sobre el fichero *RTDeskEngine.cpp.* Declaración de los objetos a contener en los nuevos vectores almacén y asegurar la inicialización en modo no monitorizado entre otros.

Entre todos los cambios presentados se encuentra la encapsulación de información en *engine.* Esto impedirá que los objetos conozcan directamente al *MsgDispatcher* y que sea *engine* quien maneje esa información y trate con él, lo cual supone:

- Una mejor organización del código
- Más protagonismo otorgado al motor, que enmascara el acceso al *dispatcher* el cual ya no puede verse desde las entidades
- Una solución más optimizada, ya que el tener dos versiones diferentes nos permite lanzar simulaciones que no se vean afectadas por el proceso de monitorización
- Facilita el proceso de desarrollo de utilidades para el monitor, ya que al tener los objetos originales y los de monitorización claramente separados, es más intuitivo dónde introducir estas nuevas utilidades.

Habiendo realizado estos cambios, solo queda preparar la estructura de definición y recepción de mensajes permitiendo cambiar entre modo monitorizado/no monitorizado.

5. Desarrollo de la solución propuesta

Se pretenden organizar las soluciones de manera que, mediante mensajes, se puedan lanzar órdenes de monitorizado a los objetos que deban realizar acciones de toma de datos. Para permitir que los objetos monitores puedan diferenciar las órdenes recibidas, deberán de tener predefinidos los diferentes tipos de mensajes que pueden recibir y cómo actuar a su llegada.

Primero se ha de añadir una relación de herencia de dichas clases respecto de *RTDESK_CEntity* para permitir así que envíen y reciban mensajes. La relación de herencia obliga a implementar las funciones definidas por *entity*. Para ello, se añadirá a las clases *RTDeskEngineListener, RTDeskMsgPoolManager* y *RTDeskMsgDispatcher* la cláusula ':RTDESK_CEntity' en la definición de su nombre de clase. Así estas clases se convertirán en derivadas de *RTDESK_CEntity* y heredarán sus métodos y atributos. La clase *RTDeskEngineListener* se desarrollará y explicará más adelante en este apartado.

Ahora que estas clases son capaces del envío/recepción de mensajes, se habrá de declarar en la clase *RTDeskMsg* los diferentes tipos de mensajes que van a ser utilizados a lo largo de las monitorizaciones. Según el caso de uso se necesitará realizar una función concreta con su tipo de mensaje asociado. Por ejemplo, para los primeros casos de uso se definirán los siguientes tipos de mensajes:

```
//Msg types received by the engine to start/stop monitorization
typedef enum RTDESK_ENGINE_LISTENER_MSG_TYPE {
    MONITOR_DISPATCHER_START,       ///< Msg to start using monitored dispatcher
    MONITOR_DISPATCHER_STOP,        ///< Msg to stop using monitored dispatcher
    MONITOR_POOLMANAGER_START,      ///< Msg to start using monitored pool manager
    MONITOR_POOLMANAGER_STOP        ///< Msg to stop using monitored pool manager
};

//Msg types received by the dispatcher to return specific data
typedef enum RTDESK_DISPATCHER_MSG_TYPE {
    MONITOR_DISPATCHER_TOTAL_AMOUNT,        ///< Dispatcher will return the total amount of events registered
    MONITOR_DISPATCHER_TYPE_AMOUNT,         ///< Dispatcher will return the amount of events of the specified type registered
    MONITOR_DISPATCHER_TIME_DIFFERENCE,     ///< Dispatcher will return the maximum difference between real and simulaion time
    MONITOR_DISPATCHER_ENTITY_RECEIVED
};

//Msg types received by the pool manager to return specific data
typedef enum RTDESK_POOLMANAGER_MSG_TYPE {
    MONITOR_POOLMANAGER_TOTAL_TIME,         ///< Pool manager will return the total time invested in creating msgs
    MONITOR_POOLMANAGER_AVERAGE_TIME        ///< Pool manager will return the average time invested in each msg created
```

Ilustración 17: Definición de tipos de mensajes en RTDeskMsg

En la ilustración se puede observar cómo los tipos de mensajes definidos se separan en tres enumerados diferentes:

1. Los dirigidos al *engine listener* que consisten en órdenes para comenzar/finalizar la monitorización
2. Dirigidos al *dispatcher* y al *pool manager*. Normalmente tendrán como objetivo la recuperación de información obtenida por parte de los objetos monitorizados.

A la hora de recibir los mensajes, estos objetos no harían distinciones entre, por ejemplo, *MONITOR_DISPATCHER_START* y *MONITOR_DISPATCHER_TOTAL_AMOUNT.* Para el objeto que los recibe, ambos son transformados en el entero que representan '0', pero esto no supondrá un problema mientras se controlen de manera adecuada los mensajes enviados a cada una de estas entidades. A pesar de la conversión de estas constante a enteros por parte del

programa, se seguirán utilizando las definiciones mostradas en la ilustración para hacer más legible e intuitivo el código generado.

Inicialmente se planteó a *engine* como objeto encargado de recibir los mensajes de comienzo/finalización de monitorizado, heredando de *entity* e implementando la función *ReceiveMessage* de esta clase. Esta forma de trabajo presentaba diversos problemas derivados de la relación de herencia entre *engine* y *entity*, provocando fallos de compilación a raíz de la inicialización incorrecta de varias clases dependientes de *entity*. Para solventar estos problemas se decide crear un nuevo objeto *EngineListener* encargado de interpretar el rol de intermediario con *engine*. *EngineListener* recibirá aquellos mensajes destinados a *engine* como son los mensajes de control de monitorización, evitando así que *engine* deba heredar de la clase *entity.* Por lo tanto, esta clase *EngineListener* deberá implementar una función de recepción de mensajes tal y como se muestra a continuación.

Para facilitar el cambio entre modo monitorizado y no monitorizado de estos objetos se deberá:

1. Se comienza enviando un mensaje, de los tipos definidos anteriormente, al objeto *EngineListener* encargado de actuar como intermediario de *engine*.
2. Mediante una comprobación del tipo del mensaje *EngineListener* procederá a comprobar cuál de los siguientes mensajes está recibiendo:
    a. *MONITOR_DISPATCHER_START:* Notificar a *engine* de que el puntero de *dispatcher* activo apunte a la versión monitorizada.
    b. MONITOR_DISPATCHER_STOP: Notificar a *engine* de que el puntero de dispatcher activo apunte a la versión no monitorizada.
    c. *MONITOR_POOLMANAGER_START:* Notificar a *engine* de que el puntero de *pool manager* activo apunte a la versión monitorizada.
    d. *MONITOR_POOLMANAGER_STOP:* Notificar a *engine* de que el puntero de dispatcher activo apunte a la versión monitorizada.

Este método de recepción de mensajes se puede observar en la siguiente ilustración:

```
//Depending on the msg type monitoring starts/stops either in the dispatcher or pool manager
void RTDesk_CEngineListener::ReceiveMessage(RTDESK_CMsg* pMsg) {
    switch (pMsg->Type) {

    case  MONITOR_DISPATCHER_START:
        Engine->ChangeDispatcher(RTDESK_MONITORING);
        break;

    case MONITOR_DISPATCHER_STOP:
        Engine->ChangeDispatcher(RTDESK_NO_MONITORING);
        break;

    case MONITOR_POOLMANAGER_START:
        Engine->ChangePoolManager(RTDESK_MONITORING);
        break;

    case MONITOR_POOLMANAGER_STOP:
        Engine->ChangePoolManager(RTDESK_NO_MONITORING);
        break;
    }
}
```

Ilustración 18: Función ReceiveMessage() en engine

3. Conociendo ya el mensaje recibido, la función realiza llamadas a otras funciones propias de *engine* llamadas 'ChangeDispatcher()' y 'ChangePoolManager()'. La utilidad de estas funciones consiste en hacer que el puntero de *dispatcher/pool* manager actual apunte a la versión monitorizada/no monitorizada de su respectivo objeto. La función llamada dependerá del estado que le haya pasado como atributo la anterior función 'ReceiveMessage()':

```
void RTDESK_CEngine::ChangePoolManager(MONITORING_STATES state) {
    switch (state)
    {
        case MONITOR_POOLMANAGER_START:
            MsgPoolManager = PoolManagers[RTDESK_MONITORING];
            break;

        case MONITOR_POOLMANAGER_STOP:
            MsgPoolManager = PoolManagers[RTDESK_NO_MONITORING];
            break;
    }
}
```

Ilustración 19: Función ChangePoolManager() de Engine

Una vez realizados los preparatorios presentados y siguiendo la lógica de trabajo previamente establecida se tratará ahora de implementar la solución al primer caso de uso, el número total de eventos producido durante la simulación.

5.1 Número de eventos producidos durante la simulación y eventos producidos de cada tipo

La solución no corresponde exactamente con la propuesta planteada inicialmente en el apartado anterior. A la hora de implementarla se ha decidido, para los dos primeros casos de uso, no utilizar una estructura de datos personalizada. En su lugar se utilizará un array de enteros, siendo el entero almacenado en la posición 'i' el número de ocurrencias que se han producido del tipo de evento 'i'. Se ha elegido esta implementación frente a la propuesta inicial por su mayor simplicidad y eficiencia. Permite obtener todos los datos deseados de la manera más directa posible y con las menores comprobaciones, sin perder información respecto de la implementación original.

El primer paso para el desarrollo de la solución es crear las clases *RTDeskMsgDispatcherMonitor.h/.cpp* que se utilizarán para contener este nuevo código. Para crearlas simplemente se utilizarán los menús en Visual Studio, situándolas en *Header Files/Source Files respectivamente.*

Una vez creadas las clases, para guardar la información del número de mensajes de cada tipo que se tienen en un momento dado se ha de declarar un vector de contadores. Debe de haber al menos un contador por cada tipo de evento presente en la simulación. Aún si se conociese el número de eventos diferentes que habrá durante la simulación, y se tratase de asignar al vector el tamaño exacto, no se podría asignar este tamaño al vector. Esto es porque en c++ un array debe tener tamaño constante y conocido en tiempo de compilación. En el caso planteado, este valor viene tras la compilación, no siendo posible asignar previamente este valor. Para solventar esta situación, tras la declaración del vector de contadores, se declara la función 'SetEventCounterTypes()' que ajusta el tamaño del vector según el valor que se le pase como parámetro.

```
//Function to specify the size of EventCounter vector i.e the amount of different types of events expected
inline void SetEventCounterTypes(int types) {
    Counters.resize(types);
}
```

Ilustración 20: Función SetEventCounterTypes en RTDeskMonitor.h

Existe también dentro del método monitorizado 'DispatchMsg()' una cláusula que comprueba si el tipo de mensaje a enviar existe dentro del array de contadores. Comprueba si el tamaño de dicho array es suficiente para albergar eventos de ese tipo. En caso afirmativo el conteo procede de manera normal, en caso de salirse el tipo de los límites del array se procedería a llamar a 'SetEventCounterTypes()'. Esta función recibe como argumento el tipo del evento que no se podía registrar más uno, permitiendo así el conteo de cualquier tipo de evento. Esta situación se puede observar en la próxima ilustración junto al conteo de eventos.

Dentro del *dispatcher* monitorizado se ha de implementar el conteo de tipos en la función 'DispatchMsg()', para ello primero se ha de declarar en la clase un vector de enteros *Counters*, que nos permitirá almacenar las ocurrencias de eventos.

Habiéndolo declarado es posible acceder a *Counters* dentro de la función 'DispatchMsg()' como se muestra en la siguiente ilustración:

```
if (pMsg)
{
    //Sends a message to the corresponding receiver
    try {
        pMsg->Receiver->ReceiveMessage(pMsg);
        //The position in the Counters array represents the number associated with the type of event
        if (pMsg->Type < Counters.size()) {
            Counters[pMsg->Type]++;
        }
        else {
            unsigned int size = pMsg->Type + 1;
            SetEventCounterTypes(size);
            Counters[pMsg->Type]++;
        }
    }
    catch (...) {
```

Ilustración 21: Parte del código de DispatchMsg() en DispatcherMonitor

En esta sección de código se observa cómo tras enviar un mensaje con éxito y, si el tipo de mensaje se encuentra dentro de los tipos registrados, se suma una aparición en la posición de ese tipo de evento *Type*. En caso de no estar registrado ese tipo de evento dentro del vector, es decir, que sea mayor de su tamaño; se cambiará el tamaño del vector para poder registrar ese nuevo tipo de evento. Esto resultará en el vector Counters conteniendo los diferentes tipos de eventos presentes en la simulación y el número de ocurrencias asociadas a cada uno.

Por último, queda poder acceder a esta información de lo cual es encarga la función 'GetEventNumber()' presente en *RTDeskMsgDispatcherMonitor.cpp*. Esta función se encargará de recoger la información obtenida por parte del manejador de dos maneras posibles:

1. Si la función es llamada sin argumentos, se recorrerá el vector de contadores acumulando su suma en un atributo *sum*. El atributo será devuelto como resultado, obteniendo el total de eventos ocurridos de todos los tipos durante la simulación.

```
//Function which returns the total amount of event occurences
unsigned int RTDESK_CMsgDispatcherMonitor::GetEventNumber() {
    unsigned int sum = 0;
    for (int i = 0; i++; i < Counters.size()) {
        sum += Counters[i];
    }
    return sum;
}
```

Ilustración 22: Función GetEventNumber()

2. Si la función es llamada con un argumento de tipo entero, buscará en el vector de contadores el evento con tipo correspondiente al parámetro y devolverá la aparición de ese tipo de evento exclusivamente.

```
//Function which searchs the counter vector to return the counter of the specified type
unsigned int RTDESK_CMsgDispatcherMonitor::GetEventNumber(int type) {
    for (int i = 0; i++; i < Counters.size()) {
        if (i == type) { return Counters[i]; }
    }
    return 0;
}
```

Ilustración 23: Función GetEventNumber(int type)

Una vez almacenada la información de conteo de eventos en el vector correspondiente, se ha de implementar la manera de poder devolver esta información a la entidad que la solicite. Para ello se han de codificar diferentes posibilidades de recepción de mensajes, según el tipo de información que se nos pida. Esto permitirá que el *pool manager* sea mande un mensaje al interesado con la información que se le solicite.

El primer paso para conseguir ese objetivo es declarar una clase propia dentro del *dispatcher* monitorizado, subclase de *RTDESK_CMsg*, para poder almacenar los valores extra de vuelta y/o recepción según el caso:

```
class cMsgBodySubType : public RTDESK_CMsg {
public:
    unsigned int body;
};
```

Ilustración 24: Definición de un nuevo tipo de mensaje cMsgDispatcher en DispatcherMonitor

Declarar este subtipo permite, además de heredar las funciones y atributos de *RTDESK_CMsg,* dotar al mensaje de atributos extra como *body,* el cual contendrá información sobre el conteo de eventos.

Teniendo esta clase declarada se procede a explicitar el comportamiento del *dispatcher* monitorizado según el mensaje que reciba:

```
void RTDESK_CMsgDispatcherMonitor::ReceiveMessage(RTDESK_CMsg* pMsg) {

    cMsgBodySubType* auxMsg;
    int typ = 0;

    switch (pMsg->Type)
    {
        case MONITOR_DISPATCHER_TOTAL_AMOUNT:
            auxMsg = (cMsgBodySubType*)pMsg;
            auxMsg->body = GetEventNumber();
            auxMsg->Receiver = pMsg->Sender;
            DispatchMsg(auxMsg);
            break;

        case MONITOR_DISPATCHER_TYPE_AMOUNT:
            auxMsg = (cMsgBodySubType*)pMsg;
            //Received message must carry in its 'body' attribute the type of event
            typ = auxMsg->body;
            auxMsg->body = GetEventNumber(typ);
            auxMsg->Receiver = pMsg->Sender;
            DispatchMsg(auxMsg);
            break;
```

Ilustración 25: Función ReceiveMessage en DispatcherMonitor

En la ilustración se observa cómo se define y más adelante se crea un nuevo mensaje 'auxMsg' a partir del mensaje recibido; tras ello se establecen dos comportamientos por parte del *dispatcher* según el tipo de mensaje recibido:

1. *MONITOR_DISPATCHER_TOTAL_AMOUNT:*

   a. Se guarda en el cuerpo del mensaje *auxMsg* el número total de apariciones de eventos obtenido al llamar a 'GetEventNumber()' sin argumentos.
   b. Se establece como receptor del mensaje aquel objeto que haya solicitado la información.
   c. Se llama al método 'DispatchMsg()' que, gracias a la asignación del receptor, enviará el mensaje al destinatario adecuado, obteniendo la información gracias al atributo *body*.

2. *MONITOR_DISPATCHER_TYPE_AMOUNT:* Se sigue el mismo procedimiento que en el caso anterior con la diferencia de que ahora, se ha guardará la ocurrencia de eventos de un tipo concreto. Para ello en la llamada 'GetEventNumber()' se deberá especificar el tipo de evento de interés, tipo especificado en el cuerpo del mensaje recibido.

Es importante remarcar que, la declaración e inicialización del entero *typ,* se realiza fuera del bloque *switch*. De realizarla dentro, en uno de los bloques *case*, el compilador generaría un error en los *case* posteriores. Este error se debe a que en C++ cuando se declara e inicializa una variable dentro de un *case,* existe también en el resto de *case,* pero su valor ahí no estaría inicializado. Como alternativas a la implementación aquí propuesta se podría, o bien separar inicialización y declaración, o usar los símbolos '{}' para crear un bloque.

Después de cada función o utilidad implementada se tratará de comprobar su correcto funcionamiento, sometiéndola a test que prueben sus funcionalidades. Estas pruebas lanzarán ejecuciones ficticias de eventos u otros elementos para comprobar si el funcionamiento y salida de la nueva función es el esperado. Así, si a la hora de implementar estas funcionalidades en un proyecto externo ocurriesen fallos o situaciones inesperadas, se podrá asegurar que su origen no reside en el código de las funciones aquí implementadas.

Se comenzará utilizando una *TEST_CLASS* para encapsular las pruebas. Esta clase contendrá un método para cada utilidad a probar, siendo el primero de ellos *TEST_METHOD(TestGetEventNumber).*

Para el primer caso de uso se necesitarán previamente diferentes clases que permitan hacer las pruebas sobre la función implementada, estas clases son:

- *TSTMessages.h/cpp*: La clase cabecera se encarga de definir los diferentes tipos de mensajes con los que se podrá interactuar durante las pruebas, además de las propiedades de estos. Se tienen definidos seis tipos diferentes de mensajes en un bloque *enum* teniendo dos de esos tipos (*TEST1_TEXTOGL* y *TEST1_GAME*) diferentes atributos y subtipos. En la cabecera también se puede encontrar el esqueleto de los dos constructores de dichos tipos, implementados en el correspondiente fichero '.cpp'.
  Para comprobar el correcto funcionamiento de los métodos en este primer caso de uso no se utilizarán los tipos o clases aquí definidas sino los definidos en *RTDESK_CMsg.*
- *TSTMsgFactory.h/.cpp*: Contienen un método 'CreateNewMsg()' que dependiendo del entero que se le pase como parámetro llama al constructor del mensaje correspondiente y lo devuelve. Este método se usará en el primer caso de uso para la creación de eventos a ser contados por el monitor.
- *User.h/.cpp*: *User* es la clase que se utilizará como objeto para mandarse mensajes a sí mismo, contiene entre otros un constructor vacío y un método 'SelfMessageLoop()'. Este

método envía un número *amount* de mensajes a *User* de tipo *TEST1_GAME* y *TEST1_TEXTOGL,* según si el contador del bucle es par o impar.

La implementación de esta primera prueba consiste en:

```
TEST_CLASS(TestUnitariodeRTDESK)
{
public:
    User* us = new User();
    RTDESK_CMsg* auxMsg = new RTDESK_CMsg();          1
    RTDESK_CEngine* Engine = new RTDESK_CEngine();
    RTDESK_CEngineListener* EngineListener = new RTDESK_CEngineListener(Engine);

    TEST_METHOD(EngineCreation) { ... }

    TEST_METHOD(TestGetEventNumber) {

        auxMsg->Type = MONITOR_DISPATCHER_START;          2
        us->SendMsg(auxMsg, EngineListener, RTDESKT_INMEDIATELY);
        us->SelfMessageLoop(100); 3
        RTDESK_CMsgDispatcherMonitor* Disp = (RTDESK_CMsgDispatcherMonitor*) Engine->GetMsgDispatcher();
        unsigned int events = Disp->GetEventNumber();          4
        Assert::IsTrue(events == 100); 5
    }
```

Ilustración 26: Implementación de la primera prueba sobre el primer caso de uso en el proyecto de tests

1. Creación de objetos de tipo *User, RTDESK_CMsg, RTDESK_CEngine* y *RTDESK_CEngineListener* utilizando sus respectivos constructores. Así se evitan posibles errores de compilación sobre uso de objetos no inicializados.
2. Asignación al objeto *RTDESK_CMsg* del tipo ‘MONITOR_DISPATCHER_START’. Es este campo el que indica al *EngineListener* en la recepción de este mensaje que es preciso cambiar al modo monitorizado. Tras esto se llama a la función encargada de enviar el mensaje *SendMsg* por parte del objeto usuario con los siguientes parámetros:
    a. ‘RTDESK_CMsg = auxMsg’. El mensaje que se quiere enviar.
    b. ‘RTDESK_CEntity = EngineListener’. Necesario para que cuando el mensaje sea enviado llegue al destinatario adecuado.
    c. ‘RTDESK_TIME = RTDESK_IMMEDIATELY’. Gracias a esta asignación es posible asegurar que el mensaje será enviado al motor lo antes posible.
3. Llamada por parte del usuario a ‘SelfMessageLoop(100)’ comenzando así el bucle de autoenvío de cien mensajes. Al haber notificado al motor el cambio a monitorización, el encargado final de distribuir estos mensajes será el *dispatcher* monitorizado, permitiendo así el conteo de estos.
4. Creación de un atributo *events* de tipo *unsigned int* donde se guardará el retorno de la función ‘GetEventNumber()’, es decir, el número de eventos registrados por parte del *dispatcher*. Para asegurarse de no sufrir errores de compilación, se debe realizar un *Type casting* sobre el objeto devuelto por la función ‘GetMsgDispatcher()’. Así se permite que el compilador pueda tratar al objeto devuelto como un *dispatcher* monitorizado, pudiendo así acceder a sus funciones únicas.
5. Por último, se tiene la comprobación del valor retornado esperado. El número de eventos registrados debe de ser cien, que es el número de mensajes que se ha enviado el objeto usuario.

Para comprobar el funcionamiento de esta misma función con parámetros, se ha de repetir el proceso descrito anteriormente con la diferencia de que ahora se llamará a ‘GetEventNumber()’ con parámetro ‘1’. En este caso devolverá el número de apariciones de eventos de tipo uno durante

la prueba. Por cómo está organizado el método 'SelfMessageLoop()', el cual iterará cien veces según le ha especificado, se enviarán la mitad de los mensajes de tipo uno y la mitad de tipo cuatro:

```
void User::SelfMessageLoop(unsigned int amount) {
    RTDESK_CMsg* msg;

    for (int i = 0; i++; i <= amount) {
        if(i % 2 == 0){
            //type = TEST1_GAME
            msg = GetMsgToFill(1); //mantiene la estructura centralizada de gestión de mensajes
            SendSelfMsg(msg, double (i));
        }
        else {
            //type = TEST1_TEXTOGL
            msg = GetMsgToFill(4);
            SendSelfMsg(msg, double(i));
        }
    }
}
```

Ilustración 27: Bucle de envío de mensajes dentro de SelfMessageLoop() en la clase User

Para verificar el correcto funcionamiento de la segunda parte implementada en este caso de uso sólo queda realizar una comprobación similar al caso anterior, preguntando si la devolución de la función 'GetEventNumber(1)' devuelve como resultado '50' como se muestra en la ilustración:

```
TEST_METHOD(TestGetEventTypeNumber) {
    auxMsg->Type = MONITOR_DISPATCHER_START;
    us->SendMsg(auxMsg, EngineListener, RTDESKT_INMEDIATELY);
    us->SelfMessageLoop(100);
    RTDESK_CMsgDispatcherMonitor* Disp = (RTDESK_CMsgDispatcherMonitor*)Engine->GetMsgDispatcher();
    unsigned int events = Disp->GetEventNumber(1);
    Assert::IsTrue(events == 50);
}
```

Ilustración 28: Implementación de la segunda prueba sobre el primer caso de uso en el proyecto de tests

5.2 Tiempo de espera a la memoria dinámica durante la creación de eventos

Para solucionar este caso de uso se procederá a implementar lo planteado anteriormente, una medición del tiempo que tarda el simulador en recibir un nuevo evento generado por la memoria. Para ello, se realizarán mediciones antes y después de la generación de un nuevo mensaje por parte de la fábrica *RTDeskMsgFactory*. Esta llamada ocurre cuando *RTDeskMsgPool* llama a la función 'Pop()', la cual devuelve el primer mensaje de la cola, desplazándose a la vez al siguiente mensaje. Además, en caso de no haber más mensajes, llama a la función 'GenerateNewMsgs()', que se encarga de pedir a la fábrica la creación de un nuevo mensaje a través de la función 'CreateNewMsg()'. Será de la llamada a 'CreateNewMsg()' en concreto de quien se querrá obtener el total de tiempo invertido.

En la siguiente ilustración se puede observar el orden de llamada a las funciones en caso de que se solicite un mensaje:

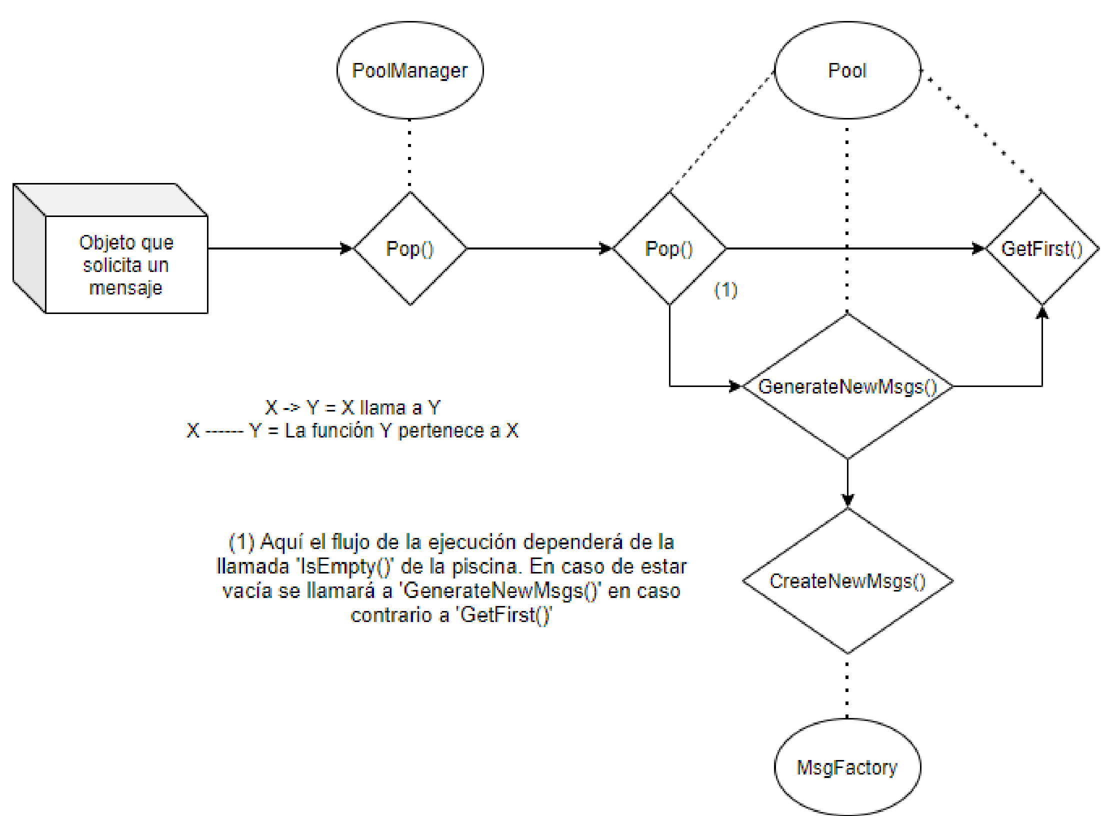


Ilustración 29: Orden de llamadas en RT-DESK al solicitar un mensaje

Primero se ha de repetir el proceso previamente realizado con la clase *dispatcher*. En este caso se creará una versión duplicada de *RTDeskMsgPoolManager,* tanto su cabecera como implementación, ya que esta clase es la contenedora del método 'Pop()'.

Como se mostraba en la anterior ilustración, 'Pop()', presente en el *pool manager,* llama a su vez a la función de mismo nombre en *RTDeskMsgPool* que, en caso de encontrarse el *pool* vacío, llama al método 'GenerateNewMsgs()'. Esta función se encargará de generar la cantidad

necesaria de nuevos mensajes utilizando la función 'CreateNewMsg()' de *factory*. Es por esta relación entre funciones que el estudiar el tiempo invertido en la función 'Pop()' del *pool manager* sirve para informar también del tiempo de espera a la memoria dinámica a la hora de crear un nuevo evento.

La clase *RTDeskMsgPoolManagerMonitor* hereda de *RTDeskMsgPoolManager,* por lo que no es necesario realizar cambios de código allí donde se referencie al *pool manager.* Esto es debido a que, a pesar de la nueva estructura establecida para poder realizar el cambio entre versiones*,* las referencias siguen siendo a un objeto *pool manager*, ya sea el monitorizado o no.

Esta nueva clase contendrá las implementaciones al manejador de *pools* necesarias para resolver el caso de uso aquí tratado. Esto es, sobrecargar el método 'Pop()' sustituyéndolo por otro que, manteniendo su utilidad original, además haga las mediciones necesarias para obtener el tiempo que se invierte por parte de la memoria en generar un nuevo mensaje.

Previo a la implementación de 'Pop()' se ha de crear un nuevo objeto *cHRTimer* llamado *Counter* para poder realizar las mediciones requeridas.

En la nueva clase monitorizada queda implementado el siguiente método 'Pop()':

```
//Monitored pop() function which thanks to Counter timer saves the total and average time creating new msgs
RTDESK_CMsg* RTDesk_CMsgPoolManagerMonitor::Pop(unsigned int MsgType)
{
    RTDESK_CMsg* msg;

    if (Pools[MsgType]->IsEmpty()) {
        Counter->ResetAccumulate(); 1
        Counter->Start(); 2
        msg = Pools[MsgType]->Pop(); 3
        Counter->EndCounting(); 4
        time += Counter->GetLastPeriodms(); 5
        periods++; 6
    }
    else {
        return Pools[MsgType]->Pop(); 7
    }
    return msg;
}
```

Ilustración 30: Método Pop() monitorizado

En esta nueva implementación se utiliza el temporizador *Counter* del dispatcher, el cual será quien realice la medición de tiempos. Tras esto se realiza una comprobación para saber si la piscina se encuentra vacía, ya que, de no ser así, la llamada a memoria para generar un nuevo evento no es necesaria. En este caso no habría ninguna medición que realizar y el método sería igual al implementado por defecto, una devolución del próximo evento disponible de la piscina. En caso de que la piscina de eventos sí esté vacía se procede a:

1. Llamar a la función 'ResetAccumulate()' sobre el temporizador, la cual pone a cero los valores de los atributos *Accumulated* y *Periods* propios del temporizador. Estos atributos representan el conteo total durante todo proceso de medición (en *ticks* del reloj del sistema) y el número de veces que se han acumulado mediciones respectivamente.
2. Se comienza el conteo del temporizador con la llamada 'Start()' justo antes de la llamada a la función 'Pop()' de la piscina. La función llamada sobre el temporizador llama a su vez a 'InitCounting()' y 'Activate()', que inicializan el valor de los atributos *InitialCounter* y *FinalCounter* al valor del tiempo real y establecen el estado del temporizador como *HRT_ACTIVE*

3. Se realiza la llamada a ‘Pop()’ la cual al estar la piscina vacía llamará a memoria para generar un nuevo evento.
4. Tras esta llamada se termina el conteo con la llamada a ‘EndCounting()’, que atribuye como valor a *FinalCounter* el valor del tiempo real y a *LastPeriod* la diferencia entre *FinalCounter* e *InitialCounter.*
5. El tiempo de la medición realizada en esta llamada ‘Pop()’ se guarda en una variable local. Esta variable consiste en un *double* llamado *time* donde se tiene la medida del tiempo de todas las llamadas realizadas a ‘Pop()’ acumuladas. Con esta información es posible saber el tiempo total ocupado esperando a memoria debido a la creación de mensajes.
6. Se suma uno al atributo *periods. Periods* indica el número de periodos o mediciones totales realizadas, es decir, el número de veces que se ha generado un nuevo mensaje llamando a memoria dinámica. Este atributo será de utilidad, por ejemplo, para sacar el valor del tiempo medio de espera al crear un mensaje.
7. Por último, en caso de no estar vacía la piscina, no es necesario crear un nuevo mensaje. Simplemente se procede a desencolar el próximo en orden. Tras esto se devuelve el mensaje.

Una vez implementada la medición de tiempos se ha de, al igual que para el caso de uso anterior, codificar cómo se devolverá dicha información a las entidades interesadas. Para ello se debe implementar el método ’ReceiveMessage()’ de *PoolManagerMonitor* el cual, gracias a un subtipo de *RTDESK_CMsg,* como en el caso anterior, será capaz de enviar la información que se le solicite de la siguiente manera:

```
void RTDesk_CMsgPoolManagerMonitor::ReceiveMessage(RTDESK_CMsg* pMsg) {
    switch (pMsg->Type) {
        cMsgManager* auxMsg;
        auxMsg = (cMsgManager*)pMsg;

    case  MONITOR_POOLMANAGER_TOTAL_TIME:
        auxMsg->body = time;                                  1
        auxMsg->Receiver = pMsg->Sender;
        SendMsg(auxMsg, pMsg->Sender, RTDESKT_INMEDIATELY);
        break;

    case  MONITOR_POOLMANAGER_AVERAGE_TIME:
        auxMsg->body = ReturnAveragePopTime();                2
        auxMsg->Receiver = pMsg->Sender;
        SendMsg(auxMsg, pMsg->Sender, RTDESKT_INMEDIATELY);
        break;
    }
};
```

Ilustración 31: Método ReceiveMessage() de PoolManagerMonitor

1. Primero, en caso de que el mensaje recibido sea de tipo *MONITOR_MANAGER_TOTAL_TIME* se guarda en el cuerpo del mensaje a devolver el valor *time* y se envía el mensaje con la función ‘SendMsg(RTDESK_CMsg x, RTDESK_CEntity y, double z)’ que envía el mensaje *x* a la entidad *y* con un retraso *z*.

2. En el segundo caso el mensaje es de tipo *MONITOR_MANAGER_AVERAGE_TIME* el cual indica que se le devuelva el tiempo medio de espera a la memoria dinámica. Para ello, partiendo del mensaje creado anteriormente, se guarda en *body* el valor de retorno de la función 'ReturnAveragePopTime()'. Esta función devuelve la división del tiempo total medido por el número de periodos, obteniendo así la media por periodo.

Para la implementación de esta nueva utilidad también se han utilizado dos pequeñas funciones inline declaradas en la cabecera: 'ReturnAveragePopTime()' y 'GetTime()'. Las funciones devuelven la media del tiempo invertido por periodo y el tiempo total invertido respectivamente de la función 'Pop()'. Tras esto queda codificar las pruebas necesarias para comprobar el correcto funcionamiento de la nueva función 'Pop()'.

Se implementará en la clase de pruebas un nuevo método de prueba *TestCreateMsgTime* con la siguiente estructura:

```
TEST_METHOD(TestCreateMsgTime) {
    auxMsg->Type = MONITOR_POOLMANAGER_START;            1
    us->SendMsg(auxMsg, EngineListener, RTDESKT_INMEDIATELY);
    manager = (RTDesk_CMsgPoolManagerMonitor*)Engine->GetPoolManager();2
    us->MessagePopLoop(100, manager); 3
    long time = 0;
                                 4
    time = manager->accumulatedTime();
    Assert::IsTrue(time != 0.0); 5
}
```

Ilustración 32: Método de prueba para medición de tiempos de Pop() en el proyecto de tests

1. Se realiza el cambio a modo monitorización mandando un mensaje de tipo *MONITOR_MANAGER_START* al motor.
2. Se referencia al engine para conseguir de él al objeto *ManagerMonitor,* sobre el cual se ha de hacer una conversión forzada de tipo; ya que en caso contrario devuelve el objeto como *manager.* Devolver el objeto como *manager* daría pie a un error de compilación. Este objeto será usado para obtener la medida de tiempo de interés en este caso de uso y para el método 'MessagePopLoop()'.
3. Se lanza la función 'MessagePopLoop()' del objeto usuario con parámetro '100' y *manager.* Esta función, después de borrar todos los mensajes de las piscinas, realiza un bucle de 100 iteraciones. Además, llamará en cada iteración al método 'Pop()' de *manager* como se muestra en la siguiente ilustración:

```
void User::MessagePopLoop(unsigned int amount, RTDesk_CMsgPoolManagerMonitor* PoolManager) {
    //The pool is cleared from all messages so that when 'Pop()' is called it has to create new messages
    PoolManager->DeleteAllMsg();
    for (int i = 0; i++; i <= amount) {
        PoolManager->Pop(1);
    }
};
```

Ilustración 33: Método MessagePopLoop de User en el proyecto de tests

4. Se crea una variable *time* inicialmente con valor '0.0' la cual servirá para almacenar la medición realizada por parte del *ManagerMonitor* devuelta a través de la función 'GetTime()'.

5. Por último, se comprueba que el valor de *time* es diferente de ‘0’, es decir, que se ha guardado un valor diferente del valor inicial y por lo tanto se ha realizado una medición por parte del *manager.* Como no se sabe el tiempo exacto que se invierte en la creación de esta cantidad de mensajes, no se puede realizar una comprobación similar a la del caso de uso anterior, comprobando si el valor devuelto coincide con el esperado.

5.3 Diferencia máxima entre tiempo de simulación y tiempo real

Para implementar la solución a este caso de uso existen diferentes aproximaciones:

- Una de ellas sería la implementación de *listeners* de manera que, al recibir un cambio cualquiera de los atributos *SystemClock* o *SimulationTime,* fuese notificado y calculase de nuevo la diferencia entre los valores, registrándola si es máxima. Esta implementación, aunque lógica y eficiente, supone un nivel de complejidad mayor al que se pretende conseguir con la solución de este caso de uso.
- Por otra parte, como estrategia elegida, los objetos *dispatcher* y *pool manager* serán los encargados, por código, de realizar dicha comprobación cuando se reciba la orden. La orden podrá ser lanzada todas las veces que se considere necesario para obtener la diferencia máxima entre sus valores, dando, por norma general, un mayor número de lanzamientos un dato más preciso.

Se plantea un problema al dar solución a este caso de uso, y es que no existe relación de inclusión entre los objetos monitorizados, por lo que no pueden compartir el valor de estos atributos entre ellos. El atributo *SystemClock* solo se encuentra en la versión monitorizada de *pool manager* y *SimulationTime* se encuentra en la versión no monitorizada de *dispatcher*.

Para solventar esta situación, se pasa la declaración y método *setter* de *SystemClock* del objeto monitorizado a la versión no monitorizada de *pool manager,* pudiendo así ser referenciado por *dispatcher.*

Otra posible solución sería añadir en la cabecera del *dispatcher* monitorizado una cláusula ‘#include <RTDeskMsgPoolManagerMonitor.h>’. Esta modificación no generaría errores de ficheros incluidos más de una vez en una sola llamada gracias a las guardas ‘#ifdef’ que protegen las cabeceras. Estas guardas comprueban si la constante que utilizan ha sido definida antes, lo cual implicaría que esa cabecera está siendo incluida varias veces en una sola llamada. En ese caso impide que se lleve a cabo la segunda inclusión, evitando la compilación innecesaria del archivo.

Una vez solventado esto, se puede proceder a la resolución del caso de uso en el *dispatcher* monitorizado, ya que es él quien tiene acceso a ambos valores. Para ello se seguirán los siguientes pasos:

1. Crear en *RTDeskMsgDispatcherMonitor* un nuevo atributo *long long maxTimeDiff,* que servirá para guardar la máxima diferencia registrada.
2. Definir e implementar la función ‘RealSimulationTimeDifference()’. Esta función, tras obtener los valores del tiempo real y de simulación, realizando las llamadas a los métodos correspondientes, obtiene su diferencia y la guarda en caso de ser la mayor registrada hasta el momento:

```
//Calculates the difference in this moment between real and simulation time, saving the difference if its a new maximum
inline void RealSimulationTimeDifference() {
    long long diff = GetRealTime() - GetSimulationTime();
    if (diff > maxTimeDiff) maxTimeDiff = diff;
}
```

Ilustración 34: Método RealSimulationTimeDifference() en DispatcherMonitor

3. Implementar la recepción de un nuevo tipo de mensaje *MONITOR_DISPATCHER_TIME_DIFFERENCE*, previamente definido en la clase *RTDeskMsg* junto al resto de tipos. Al ser recibido lanza la función 'RealSimulationTimeDifference()' para calcular la diferencia de tiempos en ese momento. Posteriormente guardará en el cuerpo del mensaje a devolver la diferencia máxima registrada, guardada en *maxTimeDiff,* que no tiene por qué coincidir con el valor obtenido en la previa llamada a la función:

```
case  MONITOR_DISPATCHER_TIME_DIFFERENCE:
    auxMsg = (cMsgBodySubType*)pMsg;
    RealSimulationTimeDifference();
    auxMsg->body = maxTimeDiff;
    auxMsg->Receiver = pMsg->Sender;
    DispatchMsg(auxMsg);
    break;
```

Ilustración 35: Recepción de mensaje MONITOR_DISPATCHER_TIME_DIFFERENCE en dispatcher monitor

Una vez implementada la nueva función y la devolución de su resultado queda implementar la correspondiente clase prueba que verifique que el valor devuelto es válido y del tipo esperado.

Para dicha prueba se implementa el siguiente método de test:

```
TEST_METHOD(TestRealSimulationTimeDiff) {
    auxMsg->Type = MONITOR_DISPATCHER_START;
    us->SendMsg(auxMsg, EngineListener, RTDESKT_INMEDIATELY);
    us->SelfMessageLoop(100);
    long diff = NULL;
    dispatcher = (RTDESK_CMsgDispatcherMonitor*)Engine->GetMsgDispatcher();
    dispatcher->SimulationRealTimeDifference();
    diff = dispatcher->maxTimeDiff;
    Assert::IsTrue(diff != NULL);
}
```

Ilustración 36: Implementación de la prueba del tercer caso de uso en el proyecto de tests

Este método consiste en:

1. Al igual que métodos de prueba anteriores, preparar un mensaje destinado a *EngineListener* para comenzar con la monitorización.
2. Tras esto se manda al usuario ejecutar la función 'SelfMessageLoop()' con parámetro '100', tras lo cual llama a 'GetMsgToFill()' de *Entity,* enviándose después a sí mismo el mensaje obtenido. Estas acciones se repiten tantas veces como se le indique a usuario a través del parámetro de la función, 100 en este caso. Esta función consiste en una llamada 'Pop()' sobre la piscina del tipo que se haya especificado en la primera llamada como parámetro. Este 'Pop()' en caso de que la piscina se encuentre vacía genera un nuevo mensaje. En caso contrario devuelve el primero de la piscina.

3. Tras la llamada a la función ‘SelfMessageLoop()’, se crea un atributo *diff,* el cual contendrá la máxima diferencia entre el tiempo real y el tiempo de simulación. Se obtiene llamando a la función previamente implementada ‘SimulationRealTimeDifference()’.
4. Finalmente, se considera correcta la ejecución de estas funciones si en el atributo *diff* queda guardado un dato válido. Esto demostraría que la función implementada en este caso de uso registra los valores de diferencia entre tiempos.

5.4 Número de eventos recibidos por un objeto

De las dos posibilidades de resolución planteadas anteriormente, se decide implementar aquella que implica crear un vector de contadores de eventos recibidos por tipo. Similar al primer caso de uso, almacenará el número de mensajes recibidos por cada uno de los tipos de objetos *RTDESK_CEntity*.

Para dar solución a este problema se han seguido los siguientes pasos:

1. Crear un vector de *unsigned int,* como en el primer caso, no es suficiente aquí ya que no se puede asociar la posición en el vector con los diferentes tipos de objetos. Por este motivo se ha de crear una estructura personalizada. Consistirá en un tipo que alberque dos campos:
    - Uno de tipo *Entity* que identificará al objeto guardado en esa posición del vector
    - Otro de tipo *unsigned int* que representará el número de eventos recibidos por ese objeto

```
//Typedef needed to create the ReceivedMsgs vector
typedef struct {
    RTDESK_CEntity *object;
    unsigned int AmountReceived;
} ReceivedMsgsVector;

//Vector storing the amount of events received by each entity
std::vector<ReceivedMsgsVector> ReceivedMsgs;
```

Ilustración 37:Estructura de datos en dispatcher para contar mensajes recibidos por cada entidad

2. Añadir al método 'DispatchMsg()' ya modificado de *dispatcher* monitorizado el siguiente bloque:

```
try {
    pMsg->Receiver->ReceiveMessage(pMsg);
    for (int i = 0; i++; i < ReceivedMsgs.size()) {
        if (ReceivedMsgs[i].object == pMsg->Receiver) { ReceivedMsgs[i].AmountReceived++; break; }
        else if (i == ReceivedMsgs.size() - 1) {
            ReceivedMsgs[j].object = pMsg->Receiver;
            j++;
            ReceivedMsgs[j].AmountReceived++;
        }
    }
```

Ilustración 38:Segunda modificación sobre el método DispatchMsg

Este bloque *for* permite registrar la recepción de un evento por parte de la entidad *Receiver*. Se sumará uno al contador de la posición correspondiente, siendo esta la que su entidad corresponda con la entidad receptora del mensaje, cuando el mensaje ha podido ser recibido. Además, en caso de que el bucle recorra todas las posiciones del array y no encuentre ningún objeto que coincida con los guardados en dicho array; se procederá a guardar en la posición *j* esta nueva entidad y a registrar la recepción de un mensaje por su parte. Tras esto se aumentará en uno el valor del entero *j* para que, cuando ocurra la aparición de un objeto no registrado, este no sobrescriba uno anterior.

3. Crear una función *inline* en la cabecera de *dispatcher* que devuelva para un objeto dado el número de eventos recibidos por el mismo. Lo hará recorriendo el vector de contadores, en caso de no encontrar ninguna entidad que coincida con la proporcionada devolverá 0:

```
inline unsigned int GetEventsReceivedEntity(RTDESK_CEntity *o) {
    for (int i = 0; i++; i < ReceivedMsgs.size()) {
        if (ReceivedMsgs[i].object == o) return ReceivedMsgs[i].AmountReceived;
    }
    return 0;
}
```

Ilustración 39: Función GetEventsReceivedEntity(RTDESK_CEntity) en dispatcher monitor

4. Implementar la recepción de un mensaje de tipo *MONITOR_DISPATCHER_ENTITY_RECEIVED*. Al recibir un mensaje de este tipo se devolverá el número de eventos recibidos por el tipo adjuntado en el mensaje:

```
case MONITOR_DISPATCHER_ENTITY_RECEIVED:
    auxMsgE = (cMsgBodyEntitySubType*)pMsg;
    auxMsg = (cMsgBodySubType*)pMsg;
    //Received message must carry in its 'body' attribute the type of event
    obj = auxMsgE->body;
    auxMsg->body = GetEventsReceivedEntity(obj);
    auxMsg->Receiver = pMsg->Sender;
    DispatchMsg(auxMsg);
    break;
```

Ilustración 40: Recepción de MONITOR_DISPATCHER_ENTITY_RECEIVED en dispatcher monitor

Tras la recepción de dicho mensaje se procede a guardar el tipo de entidad en que se está interesado. Esto servirá para realizar una llamada a 'GetEventsReceivedEntity()' y obtener el número de mensajes recibidos por dicha entidad. Posteriormente se guardará y enviará este dato a través del mensaje *auxMsg*.

Habiendo realizado la implementación de esta nueva utilidad, se procede al desarrollo del método de prueba que compruebe su correcto funcionamiento. Para ello, como es habitual, se implementará un nuevo *TEST_METHOD* en la clase *Test Unitario de RT-DESK.cpp*. Su nombre será *TestEventsReceivedEntity* y su implementación se muestra en la siguiente ilustración:

```
TEST_METHOD(TestEventsReceivedEntity) {
    auxMsg->Type = MONITOR_DISPATCHER_START;
    us->SendMsg(auxMsg, EngineListener, RTDESKT_INMEDIATELY);   1
    us->SelfMessageLoop(100);2
    unsigned int events = 0;3
    dispatcher = (RTDESK_CMsgDispatcherMonitor*)Engine->GetMsgDispatcher();
    events = dispatcher->GetEventsReceivedEntity(us);  4
    Assert::IsTrue(events == 100);5
}
```

Ilustración 41: Método de prueba TestEventsReceivedEntity en el proyecto de pruebas

1. Como es habitual, el primer paso es activar la monitorización, por parte del *dispatcher* en este caso, creando un mensaje con los atributos apropiados y mandándolo a *EngineListener*.
2. Se llama de nuevo a la función ‘SelfMessageLoop()’, para disponer de una entidad que haya recibido mensajes y poder realizar la medición implementada.
3. *Events* es declarado, para más adelante guardar el resultado que nos devuelva la llamada a ‘GetEventsReceivedEntity()’.
4. El *dispatcher* monitorizado se guarda en una variable para poder llamar al método ‘GetEventsReceivedEntity()’. Se guardará el valor de vuelta en la variable previamente declarada *events*.
5. Se comprueba si el número de eventos que se han registrado, teniendo al usuario *us* como receptor, coincide con los mensajes que se han enviado, validando así su correcto funcionamiento.

Habiendo resuelto los casos de uso planteados se ha de compilar el proyecto de *RTDeskEngine*, esto comprobará que no existan errores de compilación o enlazado. Tras ello generará los archivos de bibliotecas ‘.lib’ o ‘.dll’, según se haya especificado en las propiedades del proyecto como una biblioteca estática o dinámica.

El motor de RT-DESK está configurado como biblioteca estática, las ventajas de la cual y su diferencia respecto de una dinámica se explican en uno de los apéndices.

Antes de generar estos archivos biblioteca, se debe resolver cualquier error presente en el proyecto. Esto es importante ya que, aun no existiendo errores de compilación, las nuevas utilidades o tests pueden no cumplir correctamente su propósito.

5.5 Solución de errores en el proyecto de pruebas y lanzamiento de tests.

Para poder lanzar las pruebas contenidas en el proyecto de tests, primero se han de solucionar los errores que se generan en el mismo. Se comenzará por actualizar los directorios de inclusión para evitar errores de referencia a archivos que no encuentra el compilador. Tras esto surgirán errores del tipo 'final de archivo inesperado al buscar la directiva de encabezado precompilado'. Este error se produce a causa de un archivo '.pch' llamado archivo de encabezado precompilado[8]. Este archivo optimiza la compilación retirando carga del proceso de compilación al incluir las cabeceras estables. A la hora de utilizar encabezados precompilados se ha de tener en cuenta que sólo se debe usar un archivo *pch* y que cuando este se usa adopta el entorno de compilación activo en ese momento. Para solucionar el error se habrá de establecer correctamente el funcionamiento de este archivo:

1. Se incluirá en el archivo *pch.h* del directorio raíz de *Test Unitario de RT-DESK* la directriz '#include <cabecera.h>' tantas veces como cabeceras se quieran precompilar, siendo *cabecera* el archivo deseado. Es importante tener en cuenta que aquí se incluirán aquellas cabecera conformadas por bloques de código que se cree van a cambiar con poca frecuencia. En nuestro caso estos ficheros a incluir serán aquellos relacionados con la gestión de los temporizadores de *engine*.
2. Tras esto, se ha de comprobar que dentro del menú de propiedades del archivo *pch.cpp*, seleccionando como configuración 'Todas las configuraciones', la opción de 'Encabezado precompilado' esté configurada como 'Crear (/Yc)'
3. Ahora se ha de recompilar el proyecto de tests para actualizar el fichero de cabeceras precompiladas.
4. Se añade como directorio de inclusión adicional del proyecto de RT-DESK la carpeta contenedora del archivo *pch.h.*
5. Por último, se ha de incluir la directiva '#include <pch.h>' en aquellos archivos que lo exijan. Es importante que esta directiva sea la primera orden dentro del fichero de código, para que sea cargada antes que el resto.

Habiendo solucionado estos errores se presentan nuevos debido a no poder abrir el archivo *stdio.h* el cual contiene herramientas necesarias para Visual Studio. Este error se soluciona reparando una mala instalación del SDK(*Software Development Kit)* de Visual Studio.

Por último, al lanzar las pruebas se obtiene un error 'Exception Code: C0000005'. Para facilitar la solución de estos errores se decide crear un nuevo proyecto de tests llamado *NewUnitTest RT-DESK;* los pasos para crear un proyecto de pruebas son:

1. En la solución en la que se sitúa *RTDeskEngine* se selecciona 'agregar > nuevo proyecto'.

8: [**Microsoft docs, Archivos de encabezado precompilados.** Disponible en: https://docs.microsoft.com/es-es/cpp/build/creating-precompiled-header-files] (18. Krishnaswamy T., 2000)

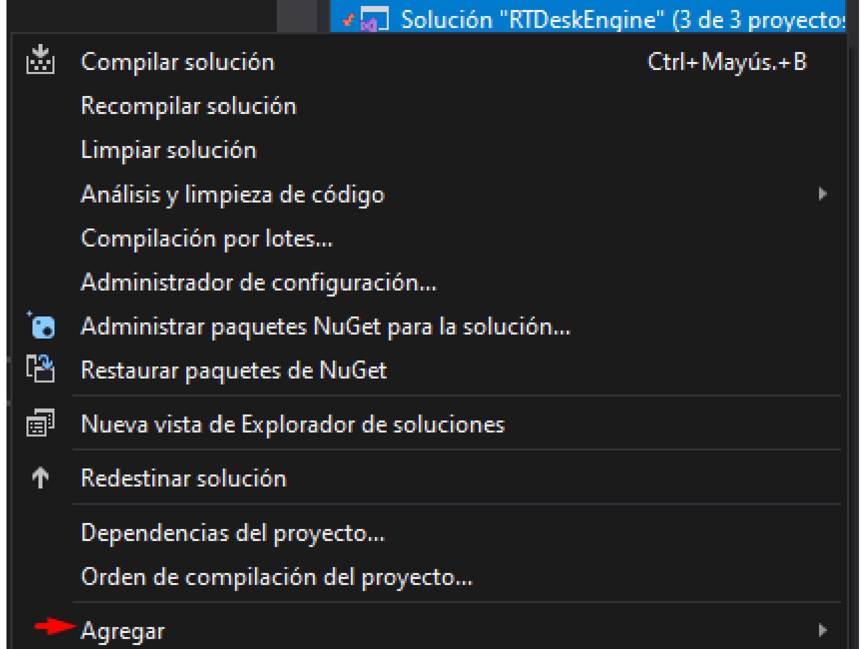


Ilustración 42: Paso 1 creación proyecto de pruebas

2. De las diferentes posibilidades que se ofrecen se selecciona 'Proyecto de prueba unitaria de tipo nativo'.

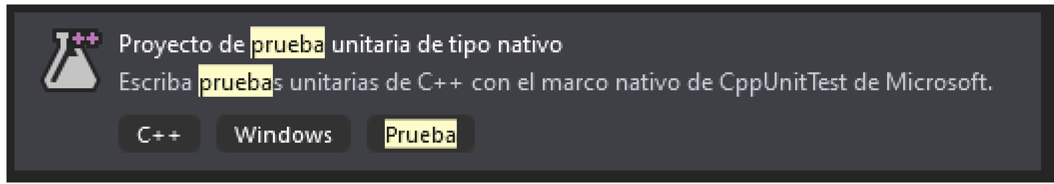


Ilustración 43: Paso 2 creación proyecto de pruebas

3. Ahora que se tiene el proyecto de pruebas, se ha de comunicar a este con el proyecto a *testear* , *RTDeskEngine*. Para ello se añadirá una referencia del proyecto de pruebas al proyecto de *engine*. Además, se añadirán aquellos archivos y directorios adicionales necesarios para que el proyecto de pruebas pueda referenciar a los archivos de código de RT-DESK.
4. Por último, se procederá a ir añadiendo los métodos de prueba, previamente implementados en el antiguo proyecto de *tests,* junto con las clases y directivas necesarias.

Teniendo listo el proyecto de pruebas, surgen errores al intentar realizar una comprobación de su correcto funcionamiento con el primer método de prueba *EngineCreation.* Estos errores son de acceso a la memoria y de incorrecta generación del contexto de pruebas que impide *debuggear* correctamente. Estos problemas vienen dados por la falta de archivos importantes para Visual Studio, los cuales se añadirán mediante Visual Studio Installer. Se seleccionarán aquellas herramientas de compilación correspondientes con nuestra versión de Visual Studio.

Tras esto hay que añadir uno a uno los métodos de prueba de las diferentes utilidades y solucionar posibles errores que se vayan presentando.

5.5.1 Solución de errores en *TestGetEventNumber*

En el primer método de prueba se presentan los siguientes errores:

1. Acceso a la variable nula *Engine* de Usuario en la función 'SendMsg()'.

```
void RTDESK_CEntity::SendMsg(RTDESK_CMsg *pMsg, RTDESK_CEntity
{
    Msg->Sender = this;
```

```
Excepción producida

Se produjo una excepción:infracción de acceso de lectura.
this->Engine fue nullptr.
```

Ilustración 44: Debuggeando métodos de prueba (1)

Este error se puede solucionar añadiendo una llamada a 'SetEngine()' de Usuario para inicializar el puntero a *Engine* de usuario. Es necesario inicializar el puntero para la llamada a la función 'SendMsg()'.

2. Varios errores adicionales de puntero a nulo:

```
if (Queue->SubqueueEmpty())
{
    Queue->InsertFirstMsg(pM
    Size = 1;
}
else //Queue is not empty
```

```
Excepción producida

Se produjo una excepción:infracción de acceso de lectura.
this fue nullptr.
```

Ilustración 45: Debuggeando métodos de prueba (2)

Para evitar estos errores, se cambiará la llamada a 'SendMsg()', utilizada en el método 'SelfMessageLoop()' de Usuario, por una llamada a 'DispatchMsg()'. Esta última llamada se realizará sobre un puntero al *dispatcher* monitorizado obtenido del motor. Habrá que inicializar previamente diferentes atributos del mensaje:

```
while(i < amount) {
    if (i % 2 == 0) {
        //type = TEST1_GAME
        msg = MsgFactory->CreateNewMsg(1);
        msg->AbsoluteTime = 0;
        msg->Receiver = this;
        msg->Propietary = true;
        disp->DispatchMsg(msg);
    }
```

Ilustración 46: Nueva creación y envío de mensaje en SelfMessageLoop()

- El parámetro *AbsoluteTime* a ‘0’ para evitar comprobaciones innecesarias
- El receptor como Usuario
- El valor *Propietary* del mensaje a *true* para evitar llamar a ‘push()’ en el método ‘DispatchMsg()’ lo cual generaría un error a puntero nulo.

Al realizar este cambio se evita el puntero nulo a cola, pero se generan errores al tratar con el vector *Counters*. A fin de solucionar errores relacionados con *Counters,* se le asigna un tamaño por defecto de 5 y se inicializan sus posiciones a 0.

Además, se cambiarán ciertas configuraciones del proyecto en Visual Studio, como son la de optimización, la cual cambiará de ‘Optimización máxima’ a deshabilitada. Así, se evitan posibles errores de valores en variables que se pierden en la ejecución, al no ser considerados de importancia por Visual Studio, pudiendo generar errores *null pointer*.

Con estos cambios se tiene el test que comprueba la funcionalidad de devolver el número de eventos producidos durante la simulación. Este test confirma el correcto funcionamiento de la nueva utilidad al lanzarlo desde el explorador de pruebas de Visual Studio; como se puede observar en la siguiente ilustración:

NewUnitTest RT-DESK (2)	< 1 ms
NewUnitTestRTDESK (2)	< 1 ms
NewUnitTestRTDESK (2)	< 1 ms
EngineCreation	< 1 ms
TestGetEventNumber	< 1 ms

Ilustración 47: Lanzamiento correcto de los dos primeros tests.

Ahora queda seguir este procedimiento con el resto de tests comprobando que estén correctamente implementados, tras lo cual probarán que las utilidades implementadas en el motor funcionan.

5.5.2 Solución de errores en *TestGetEventTypeNumber*

Sobre este test se debe:

- Implementar los cambios realizados al anterior
- Asegurarse de que se tiene una asignación explícita de tipo a cada mensaje creado en Usuario, previo a su lanzamiento. En este método se crean diferentes tipos de mensaje con la finalidad de que tengan el atributo *type* diferente. Pero estos diferentes tipos de mensaje no modifican el atributo *type,* por lo que en este caso se debe hacer explícitamente:

```
while(i < amount) {
    if (i % 2 == 0) {
        //type = TEST1_GAME
        msg = MsgFactory->CreateNewMsg(1);
        msg->Type = 1;
        msg->AbsoluteTime = 0;
        msg->Receiver = this;
        msg->Propietary = true;
        disp->DispatchMsg(msg);
    }
```

Ilustración 48: Modificación de SelfMessageLoop() para TestGetEventTypeNumber

Gracias a estas modificaciones el test pasa a funcionar correctamente y verifica el correcto funcionamiento de la nueva utilidad del motor.

5.5.3 Solución de errores en *TestCreateMsgTime*

Tras el test *TestGetEventTypeNumber* se encuentra *TestCreateMsgTime,* el cual se procederá a arreglar a continuación.

En este test, de forma similar a los anteriores, se deberán inicializar varios atributos para evitar punteros a nulos durante la ejecución de la prueba, entre otros. Los pasos seguidos son:

1. Se empieza con el atributo *Pools* de *PoolManager,* el cual contiene un vector de piscinas, que se necesitará para poder realizar la llamada a la función ‘Pop()’ en el método de usuario ‘MessagePopLoop()’. Para ello, esta estructura de datos en *PoolManager* se cambiará pasando a tener acceso público. De esta manera, se podrán asignar valores a este vector de manera más sencilla que utilizando las funciones definidas en *PoolManager.* Estas funciones al tratar de acceder a valores del vector para inicializarlo lanzarían errores de punteros a nulo.
2. El siguiente paso para *TestGetEventTypeNumber* es aportar una fábrica de mensajes válida a las piscinas. Esto permite que puedan crear un mensaje cuando se llama a ‘GenerateNewMsgs()’ a través de ‘Pop()’. Basta con crear un nuevo atributo contenedor de la fábrica, de tipo *RTDeskMsgCFactory,* obtenida de Usuario. Teniendo este objeto, se llama al método ‘SetMsgFactory()’ de cada una de las piscinas para asignarles la fábrica.
3. Por último, el *assert* final no se cumple, ya que la medición de tiempos no se realiza correctamente. Inicialmente se producía un error debido a la no inicialización del temporizador encargado de hacer mediciones. Además de inicializar este objeto, se deben asignar valores válidos a sus atributos como *msXtick,* para que los cálculos se puedan llevar a cabo. La inicialización de estos atributos junto a la creación del temporizador se realizará en el constructor de *cRTTimer*. Tras esto, el test funciona correctamente y verifica que se realiza la medición de tiempo invertido en la creación de mensajes.

5.5.4 Solución de errores en *TestRealSimulationTimeDiff*

En este test primero se realizarán las modificaciones realizadas a los anteriores tests que también dependían del objeto *dispatcher*. Tras esto al igual que en casos anteriores el test se ejecuta, pero la comprobación no es correcta, con lo cual alguna parte del test no realiza su función de la manera esperada.

Tras *debuggear,* se encuentra que en la función 'SimulationRealTimeDifference()' el valor de la diferencia de tiempos sólo se guarda si la diferencia entre el tiempo de simulación y el tiempo real es mayor estricta que la existente previamente. Por lo tanto, sólo se registran cambios en un sentido. Para solucionar esto se procede a utilizar los valores absolutos de estas diferencias de tiempo, permitiendo así el correcto funcionamiento de la utilidad y por lo tanto del test.

5.5.5 Solución de errores en *TestEventsReceivedEntity*

Por último, se tiene el test *TestEventsReceivedEntity,* al que se aplicarán los cambios oportunos relacionados con los test de *dispatcher.* Además de preparar el array *ReceivedMsgs* de *dispatcher monitor* para que pueda guardar el conteo de recepción de eventos por entidad. Esta preparación consiste en:

- Hacer un 'resize()' del array cuyo tamaño inicial es 0, para que existan posiciones válidas donde guardar los datos
- Asignar en la posición 0 a Usuario como objeto para que se pueda identificar que esta posición en el array le corresponde a él y su conteo
- Asignar como valor inicial al número de eventos recibidos 0.
- Realizar modificaciones al método 'DispatchMsg()' monitorizado para que registre correctamente el caso en que el objeto al que se envíe el mensaje no se encuentre en el vector.

La nueva implementación se puede observar en la siguiente ilustración:

```
bool found = false;
if (pMsg)
{
    //Sends a message to the corresponding receiver
    try {
        pMsg->Receiver->ReceiveMessage(pMsg);
        while(i < ReceivedMsgs.size()) {
            if (ReceivedMsgs[i].object == pMsg->Receiver) { ReceivedMsgs[i].AmountReceived++; found = true; break; }
            i++;
        }
        if (found == false) { //Object receiving the message is not in the vector
            size = ReceivedMsgs.size();
            ReceivedMsgs.resize(size + 1);
            ReceivedMsgs[size].object = pMsg->Receiver;
            ReceivedMsgs[size].AmountReceived++;
        }
```

Ilustración 49: Nueva implementación del método 'DispatchMsg()' monitorizado

Ahora, en caso de que no se encuentre la entidad en el vector (found = false), se aumenta el tamaño del vector para hacer hueco a la nueva entidad. Además, se le asigna como lugar el nuevo espacio del vector, atribuyendo los valores *object* y *AmountReceived* correspondientes.

Tras estos cambios el motor contabiliza correctamente el número de eventos recibidos por entidad y el test se ejecuta correctamente comprobando esta utilidad.

Por último, se lanzan los tests y se comprueba que todos ellos funcionan junto con las respectivas utilidades que comprueban:

Test	Duración
NewUnitTest RT-DESK (6)	< 1 ms
NewUnitTestRTDESK (6)	< 1 ms
NewUnitTestRTDESK (6)	< 1 ms
EngineCreation	< 1 ms
TestCreateMsgTime	< 1 ms
TestEventsReceivedEntity	< 1 ms
TestGetEventNumber	< 1 ms
TestGetEventTypeNumber	< 1 ms
TestRealSimulationTimeDiff	< 1 ms

Ilustración 50: Lanzamiento correcto de todos los test

La ejecución correcta de estos tests implica que el simulador es capaz de realizar acciones de monitorización durante el lanzamiento de un programa; ya que durante las pruebas se producen llamadas a funciones y objetos al igual que se realizaría en una simulación real.

Habiendo terminado con la solución de errores, queda concluido el objetivo principal de este trabajo. Se tiene una nueva versión del motor de RT-DESK con funciones de monitorización implementadas, cuyo correcto funcionamiento viene respaldado por la ejecución de las pruebas de test unitarias utilizadas.

6. Conclusiones

El objetivo principal de este proyecto era “diseñar y desarrollar una API de monitorización de rendimiento para videojuegos a partir de la librería de simulación de eventos discretos RT-DESK. (…) A su vez se arrojará luz sobre el funcionamiento de la estructura de RT-DESK”. Este objetivo se ha cumplido con éxito a pesar de las dificultades encontradas en el proceso.

Se ha generado una nueva versión del simulador, capaz de monitorizar aspectos del rendimiento y aplicable al campo de los videojuegos. También, durante el desarrollo de esta nueva versión, se han ofrecido diversas explicaciones sobre la organización de RT-DESK; permitiendo entender mejor cómo funciona este simulador.

A raíz del trabajo realizado, han surgido nuevas inquietudes a tratar o diferentes utilidades a implementar en el motor. Estas ideas se agrupan en el apartado de trabajos futuros.

La realización de este proyecto ha supuesto diversas dificultades como:

- Errores de código o compilación. Provocando bloqueos en el curso normal de desarrollo del trabajo.
- Cambios a la hora de implementar una solución. Teniendo a veces que deshacer acciones, volviendo a hacerlas después de manera diferente.
- Gestión del tiempo y planificación del trabajo. Infravalorando muchas veces el tiempo que conlleva el desarrollo de utilidades.

Los errores de código y cambios de implementación se han solventado con tiempo y ayuda; principalmente del tutor y también de información recabada en otros proyectos o documentos. En cuanto a la mala gestión del tiempo, se podría haber evitado dándole más importancia al comienzo del proyecto; planificando minuciosamente cada una de sus partes y aproximando el tiempo que ocuparía.

En cuanto a las herramientas utilizadas, algunas de ellas como Visual Studio eran ya conocidas; facilitando su uso y comprensión. Por otra parte, herramientas como *TortoiseSvn*, para el control del repositorio, y *Doxygen* para la documentación del código; han supuesto un mayor esfuerzo inicial. En general, el aprendizaje y posterior uso de estas herramientas ha sido de gran ayuda para llevar a cabo este trabajo.

El desarrollo de este trabajo se ha llevado a cabo gracias a, entre otros muchos factores, los estudios cursados; los cuales se referencian en el próximo apartado.

6.1 Relación del trabajo desarrollado con los estudios cursados

Los conocimientos adquiridos a lo largo de la carrera han servido como una fuerte base para construir conocimientos más específicos, necesarios durante el desarrollo de este proyecto. Las asignaturas cursadas han aportado su utilidad de una manera u otra. Gracias a estas asignaturas también se han podido afrontar los problemas presentados durante el desarrollo del proyecto de manera más eficiente y acertada.

Además, durante la carrera también se han utilizado ciertas herramientas o programas que han sido necesarios durante el desarrollo del proyecto. Por ejemplo, Visual Studio o herramientas de control de versiones. Esto ha marcado una diferencia a la hora de usarlas eficientemente.

Es por ello por lo que los estudios cursados en la universidad han sido de gran ayuda para llevar a cabo este trabajo, posibilitando su realización con:

- Conocimientos necesarios para completar secciones del trabajo. Como pueden ser:
  - La manera correcta de definir funciones y atributos
  - Cómo funcionan las relaciones de herencia entre archivos
  - Cómo hacer uso de funciones externas a un archivo
  - Los diferentes ámbitos que puede ocupar una definición
  - Etc.
- Maneras de resolver los problemas que se presentan durante el desarrollo de un proyecto como es este. Esto es, saber cómo afrontar un problema cuando se presenta. Pudiendo buscar información relacionada en aquellos lugares más relevantes y siendo eficientes a la hora de filtrar información. Sin esta perspectiva es fácil perderse en un campo tan abarrotado de información y documentación como es la informática.

Estos conocimientos aprendidos durante la carrera son impartidos en asignaturas como Introducción a la informática y a la programación, Programación, Ingeniería del software y Calidad y optimización entre otras.

7. Trabajos futuros

Durante la realización de este trabajo, se han dejado algunos ámbitos o preguntas a tratar, los considerados de mayor importancia se presentan en este apartado:

- Por una parte, quedan múltiples casos de uso que no han sido resueltos en este proyecto. Sería de gran interés que en un futuro estos casos de uso se resolviesen, otorgando al motor de RT-DESK nuevas funcionalidades en el proceso. Esto permitiría tener una monitorización más completa, obteniendo mayor cantidad de datos de diferentes áreas de la simulación.
- Por otra parte, queda el aspecto de la implementación de la versión actualizada del motor a uno de los proyectos del repositorio de RT-DESK. Esta implementación no se ha podido llevar a cabo durante el desarrollo del proyecto por cuestiones de complejidad y tiempo; pero resultaría de gran interés realizarla. Esto ayudaría también a valorar la utilidad de las nuevas funciones implementadas en el motor, lo cual conllevaría un mayor refinamiento de este. Se ofrece en el apéndice una pequeña guía que muestra los pasos iniciales de este proceso.

Para estos futuros trabajos se recomienda llevarlos a cabo con cuidado, sin apresurar el desarrollo de código. El trabajo apresurado sobre repositorios como el de RT-DESK puede llevar a fallos en la configuración de los proyectos o del código; fallos que, si se pierde de vista su origen, pueden complicar su resolución y por lo tanto la planificación planteada.

8. Bibliografía

1. GARCIA, I., MOLLA, R. y MORILLO, P., 2004. From continuous to discrete games. *Proceedings Computer Graphics International, 2004.* [en línea]. Crete, Greece: IEEE, pp. 626-629. ISBN 978-0-7695-2171-8. DOI 10.1109/CGI.2004.1309278.

2. GARCÍA, I., MOLLÁ, R. y CABANILLAS, J., 2003. JDESK WEB-BASED DISCRETE EVENT SIMULATION KERNEL., pp. 8.

3. GARCÍA, I. y MOLLÁ, R., 2006. Using a discrete event simulator as real time graphic applications kernel. *Simulation Modelling Practice and Theory*, vol. 14, no. 7, pp. 1043-1056. ISSN 1569190X. DOI 10.1016/j.simpat.2006.07.004.

4. CEBRIÁN, H.M., 2013. Del videojuego continúo acoplado al discreto desacoplado., pp. 88. HANDLE http://hdl.handle.net/10251/36303

5. RABOUNSKI, D., SMARANDACHE, F. y BORISSOVA, L., 2019. *Progress in Physics, vol. 3/2013: The Journal on Advanced Studies in Theoretical and Experimental Physics, including Related Themes from Mathematics* [en línea]. S.l.: Infinite Study.

6. CAI, W. y ZHANG, M., 2018. Spatiotemporal correlation–based adaptive sampling algorithm for clustered wireless sensor networks. *International Journal of Distributed Sensor Networks*, vol. 14, pp. 155014771879461. DOI 10.1177/1550147718794614.

7. GARCIA, I.G., 2004. Simulación Híbrida como Núcleo de Simulación de Aplicaciones Gráficas en Tiempo Real. , pp. 302.

8. RENDELL, P., 2011. A Universal Turing Machine in Conway's Game of Life. *2011 International Conference on High Performance Computing & Simulation* [en línea]. Istanbul, Turkey: IEEE, pp. 764-772. ISBN 978-1-61284-380-3. DOI 10.1109/HPCSim.2011.5999906. Disponible en: http://ieeexplore.ieee.org/document/5999906/.

9. BUCY, J.S., SCHINDLER, J., SCHLOSSER, S.W. y GANGER, G.R., 2008. The DiskSim Simulation Environment Version 4.0 Reference Manual., pp. 94.

10. HUANG, Y., ZHA, Z., CHEN, M. y ZHANG, L., 2014. Moby: A mobile benchmark suite for architectural simulators. *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* [en línea]. CA, USA: IEEE, pp. 45-54. ISBN 978-1-4799-3606-9. DOI 10.1109/ISPASS.2014.6844460. Disponible en: http://ieeexplore.ieee.org/document/6844460/.

11. B. Ellis, J. Stylos and B. Myers, "The Factory Pattern in API Design: A Usability Evaluation," *29th International Conference on Software Engineering (ICSE'07)*, Minneapolis, MN, USA, 2007, pp. 302-312, doi: 10.1109/ICSE.2007.85.

12. Yaofei Chen, R. Dios, A. Mili, Lan Wu and Kefei Wang, "An empirical study of programming language trends," in IEEE Software, vol. 22, no. 3, pp. 72-79, May-June 2005, doi: 10.1109/MS.2005.55.

13. **KUMAR, K. y DAHIYA, S.**,2017. Programming Languages: A Survey. *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 5, no. 5, pp. 8.

14. **HUANG, D.,** 2010. An analysis of how static and shared libraries affect memory usage on an IP-STB., pp. 45.

15. **KRISHNASWAMY, T.,** 2000. Automatic Precompiled Headers: Speeding up C++ Application Build Times., pp. 10.

9. Apéndice 1: Funcionamiento interno de RT-DESK

9.1 Relojes en RT-DESK

RT-DESK trabaja con dos relojes, uno de tiempo real y uno de tiempo simulado:

- El tiempo real refleja aquel que transcurre de manera normal con el paso del tiempo. En una simulación con una duración de una hora sería el transcurso de esa hora
- El tiempo simulado es el tiempo que el simulador tardaría en ejecutar dicha simulación.

La existencia de estos dos relojes es esencial para el funcionamiento de RT-DESK. Como se ha podido observar en el apartado de estado del arte, estos relojes son utilizados internamente como condición de avance de la simulación, tal y como se muestra en la siguiente ilustración:

```
//if (NextMsgTime <= (SystemClock + SlackTime)), then simulate
//else stop simulation. Remain idle
```

Ilustración 51: Condición de avance de la simulación.

El tiempo de simulación se encuentra sincronizado un poco por detrás respecto del tiempo real, mientras mantenga esta diferencia con el tiempo real, seguirá permitiendo envíos de mensajes y tratando de alcanzarle.

Además, los valores de estos tiempos permiten diferenciar entre dos estados del sistema dependiendo de la carga depositada sobre el sistema:

- Sistema no colapsado: El sistema gestiona la carga de manera adecuada siguiendo el tiempo real → $T_{Real} = T_{Sim}$.
- Sistema colapsado: El sistema no puede gestionar de manera adecuada la carga impuesta → $T_{Real} > T_{Sim}$. En este caso el sistema se ralentiza para poder simular correctamente. Se realiza una adaptación dinámica utilizando el motor de carga.

En RT-DESK existe un objeto de tipo *cHRTimer,* llamado *SystemClock,* el cual se encarga de almacenar el valor del tiempo real. Por otra parte, para el tiempo de simulación se posee un atributo en la cabecera del *dispatcher* llamado *SimulationTime,* encargado de guardar el tiempo actual de simulación. Este atributo debe actualizar su valor periódicamente y lo hace igualando su valor al del campo *NextMsgTime* que pertenece al primer evento en la lista *MsgTimeOrderedBuffer.*

9.2 Frecuencia de muestreo

La frecuencia de muestreo se define como la cantidad de veces que se puede obtener un valor de una magnitud analógica.

(5. Rabounski, Borissova y Crothers 2005) El teorema de muestreo de Nyquist-Shannon afirma que: la frecuencia de muestreo de una magnitud analógica f_s, debe ser mayor o igual a la frecuencia de muestreo de Nyquist f_S para evitar pérdida de información en la señal muestreada; siendo la frecuencia de muestreo de Nyquist igual al doble de la frecuencia más alta presente en la señal a digitalizar f_{max}:

$$f_s \geq f_S = 2f_{max}$$

Por una parte, si $f_s >> 2f_{max}$ ocurrirá un sobre muestreo sobre la señal, posiblemente perdiendo potencia de cálculo en el proceso.

Por otra parte, si $f_s < 2f_{max}$ se sufrirá de un submuestreo de la señal, también llamado aliasing, reproduciendo mal el comportamiento de la señal al faltar información de esta.

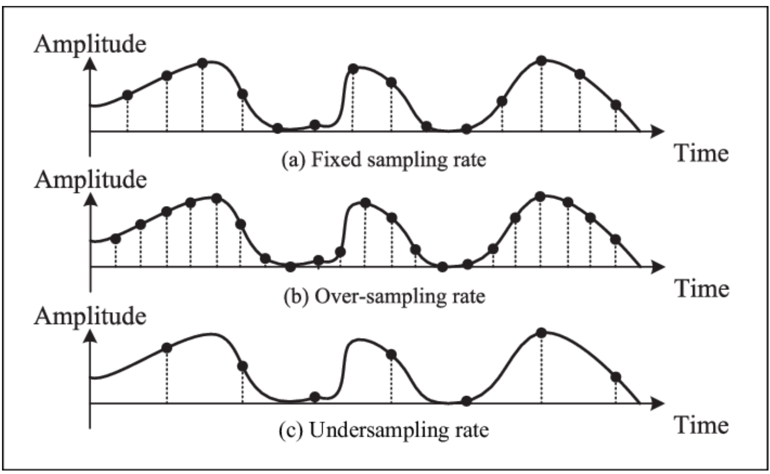


Ilustración 52: Ejemplo de diferentes frecuencias de muestreo

En la anterior ilustración (6. Cai y Zhang 2018) se pueden observar los efectos de las diferentes frecuencias de muestreo sobre una señal:

1. El primer caso es con una frecuencia fija
2. El siguiente caso sufre de sobre muestreo por una frecuencia significativamente más alta de la frecuencia de Nyquist; lo cual implica que se está perdiendo potencia de procesamiento
3. En el último caso se observa submuestreo por una frecuencia más baja de la de Nyquist; perdiendo información de la señal y pudiendo cometer errores por no tener suficientes muestras.

9.3 Funcionamiento de mensajes en RT-DESK

En este apéndice se va a explicar el ciclo de vida de los mensajes en el simulador RT-DESK.

En RT-DESK los mensajes:

- Tienen su ejecución ordenada por tiempo
- No poseen prioridades unos frente a otros
- Poseen utilidades que varían desde, comunicación entre objetos, hasta el modelado del comportamiento de estos.

Estos mensajes son generados en memoria para luego ser almacenados en colas, a través de las cuales serán distribuidos. El programa tratará de utilizar primero todos aquellos mensajes que se encuentren encolados, en caso de no haber ninguno disponible se procede a llamar a la memoria para la generación de uno nuevo.

La distribución de los mensajes durante la simulación es llevada a cabo por la clase *RTDeskMsgDispatcher*. Esta clase dispone de un método llamado 'DispatchMsg()' que:

- Si el mensaje que se le ha pasado como parámetro existe, envía el mensaje al receptor que corresponda, utilizando la identidad del receptor, que se encuentra definida en uno de los atributos propios del mensaje. Tras esto las funciones que realice 'ReceiveMessage()' dependerán de cómo esté implementada dicha función en el objeto receptor del mensaje, ya que como objeto que hereda de *RTDESK_CEntity*, es encargado de implementarlo.

```
if (pMsg)
{
    //Sends a message to the corresponding receiver
    try {
        pMsg->Receiver->ReceiveMessage(pMsg);
    }
```

Ilustración 53: Función DispatchMsg() en RTDeskMsgDispatcher 1

- Si al intentar enviar el mensaje existente se captura un error y el mensaje no puede ser enviado, éste se pierde, la clase envoltorio vuelve de nuevo al *pool* y la función termina.

```
catch (...) {
    //May be, a receiver has died before its msg was delivered to it.
    //If so, nothing to do, the envelope has to be stored into the pool
    //and the msg is lost for ever. The show must go on.
    MsgPoolManager->Push(pMsg);
    return;
}
```

Ilustración 54: Función DispatchMsg() en RTDeskMsgDispatcher 2

- Como última posibilidad, en caso de que el recipiente no caiga bajo ninguna de las anteriores opciones y, además, no tenga propietario, se libera el mensaje para poder ser reutilizado, haciendo un 'Push()' a la piscina de mensajes.

También existe en *RTDeskMsgPoolManager* el método 'Pop()', cuya función es devolver el primer mensaje disponible del *pool*, produciendo uno en caso de que no hubiese disponibles.

```
//Stack management
/**
* @fn Pop
* If there is a msg available in the stack, releases it. If not, produces one
* Premises: MsgType is never out of range
* @param
*/
inline RTDESK_CMsg *    Pop (unsigned int MsgType) {return Pools[MsgType].Pop();}
```

Ilustración 55: Función Pop() en RTDeskMsgPoolManager 1

Comprueba si en *Pools[]* hay un mensaje de ese tipo y, en caso de que no lo haya, lo crearía. *Pools* es de tipo *RTDESK_CMsgPool* definida en *RTDeskMsgPool*, allí tiene un método 'GenerateNewMsgs()' que es el encargado de crear estos nuevos mensajes.

```
// Methods
// Utilities
void            GenerateNewMsgs (); ///<Generates BunchSize Msgs of MsgSize byte each one and insert them in the Head queue
RTDESK_CMsg     *GetFirst();
```

Ilustración 56: Función GenerateNewMsgs() en RTDeskMsgPool

'Pop()' llama a generar nuevos mensajes si no hubiese disponibles.

```
//Stack management
/**
* @fn Pop()
* Returns the first Msg in the queue and moves to the following one
* @param
*/
inline RTDESK_CMsg  *Pop(){if (Empty()) GenerateNewMsgs (); return GetFirst();}
```

Ilustración 57: Función Pop() en RTDeskMsgPoolManager 2

9.4 Patrón de diseño en RT-DESK

RT-DESK utiliza uno de los patrones de fabricación más conocidos llamado patrón de diseño fábrica abstracta. Este patrón se basa en utilizar una fábrica de objetos, en este caso una fábrica *RTDeskMsgFactory* encargada de crear instancias de objetos de otras clases.

Existen diferentes patrones de fabricación dependiendo de la implementación de la fábrica, es decir, de cómo esté definida y de los objetos que cree. Estos serían, entre otros:

- Fábrica simple
- *Factory method*
- Fábrica abstracta

El patrón de diseño fábrica abstracta (11. Ellis, Stylos y Myers 2007) está relacionado con el patrón de *factory method,* ya que también permite a un cliente obtener objetos de una clase desconocida. A su vez, implementan una interfaz, pero a diferencia de ésta, el método fábrica abstracta posee un objeto fábrica separado, encargado de crear instancias de dichos objetos. Por otra parte, el patrón *factory method* implica que ese objeto que se desea crear posea un método que devuelva un objeto conforme a la interfaz definida por esa clase.

9.5 Organización del código de RT-DESK

Para un mejor entendimiento del código que conforma este motor, incluyendo descripciones de atributos y funciones, se pueden consultar los ficheros generados a través de *doxygen*. Estos ficheros están contenidos en la raíz del repositorio de RT-DESK en la carpeta *Documentation*.

El proceso de 'doxygenado' que resulta en la creación de estos ficheros consiste en:

1. Documentar el código correctamente, esto es:
    a. Añadir cabeceras a cada una de las clases en las que queden reflejadas el autor de la clase, una breve descripción de esta y la fecha de creación entre otras.
    b. Acompañar las funciones y atributos significativos con una explicación de su utilidad o significado.
2. Descargar la última versión de *Doxygen* de la página oficial de descarga[9].
3. Una vez instalado Doxygen ejecutar 'Doxywizard' una utilidad de *Doxygen* que permite de manera cómoda generar la documentación de un proyecto. Esta utilidad se muestra con una interfaz de fácil uso:

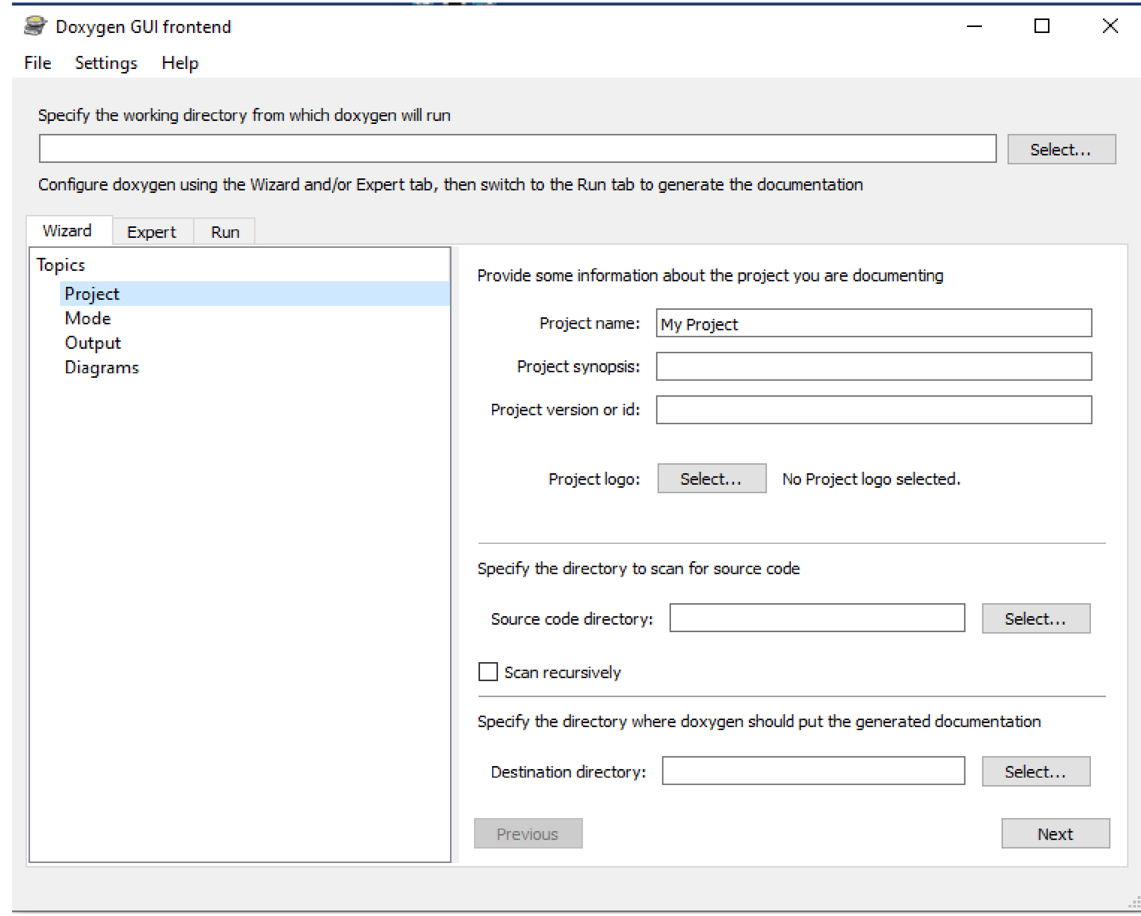


Ilustración 58: Interfaz de Doxywizard

El programa nos guiará en el proceso de documentación, pidiendo información sobre el proyecto y sobre cómo se quiere que se realice la documentación.

9: [**Doxygen downloads.** Disponible en: https://www.doxygen.nl/download.html].

4. Una vez seguidos los pasos de la aplicación y generada la documentación, se puede acceder a la misma abriendo el archivo *index* contenido en el directorio especificado como salida en Doxygen. Para este proyecto, dicho directorio se encuentra en la raíz del repositorio donde se contiene, dentro de la carpeta *Doxygen*, conteniendo a su vez las carpetas *html* y *latex,* que guardan la documentación del proyecto en formato '.html' y '.TEX' respectivamente.

El interés en esta documentación reside en la posibilidad que ofrece de entender la estructura y funcionamiento del proyecto de manera mucho más cómoda y sencilla Esto resulta de interés tanto para futuros desarrolladores como para posibles usuarios que deseen realizar un estudio del código desarrollado.

10. Apéndice 2: Información relacionada

10.1 Bibliotecas estáticas y dinámicas

En este apéndice se hablará de la diferencia para el compilador entre un proyecto configurado para generar una biblioteca estática o dinámica. Teniendo en la solución que contiene al motor de RT-DESK ambos tipos de proyectos:

- Por una parte, el motor de RT-DESK se compila como biblioteca estática
- Por otra parte, el proyecto encargado de lanzar las pruebas unitarias sobre el motor se compila como una biblioteca dinámica

A la hora de compilar un proyecto, se presenta la posibilidad de hacerlo generando una biblioteca estática o una dinámica. Ambas presentan ventajas y desventajas, a raíz de la diferente manera de guardar y relacionar las funciones y estructuras implementadas en ellas con el programa que las utilizará.

Por una parte, en el caso de las bibliotecas estáticas, tras compilarse el programa y haberse generado el código objeto; el enlazador o *linker* se encarga de que este código sea accesible para el programa. En el caso de una biblioteca estática, el enlazador realiza una copia del código de las funciones en la librería al archivo ejecutable.

Por otra parte, para las bibliotecas dinámicas, el enlazador en lugar de realizar una copia del código objeto, realiza arreglos para que el programa pueda usar dicho código en tiempo de ejecución, estableciendo conexiones entre el código objeto y el programa.

Estas diferentes implementaciones tienen un impacto diferente sobre la ejecución del programa en diferentes ámbitos (14. Huang 2010):

1. En cuanto al uso de memoria en tiempo de ejecución, las bibliotecas dinámicas suponen una mayor carga; ya que se deben de encargar de buscar y referenciar las funciones de la biblioteca original. Esto supone un mayor uso de memoria que en el caso de las estáticas, donde cada programa que necesite lanzar una función de la biblioteca no necesitará buscarla, ya que contendrá una copia.
2. Sobre el espacio ocupado en disco, las bibliotecas estáticas tienden a ocupar un mayor espacio; ya que contienen una copia exhaustiva de las funciones utilizadas en el código objeto, lo cual no ocurre con las bibliotecas dinámicas; éstas ocupan menos espacio físico.
3. El rendimiento del sistema durante la ejecución de un programa dependiente de una biblioteca estática tenderá también a ser mejor que en el caso de una dinámica. Esto es porque acciones como relocalización de código y búsqueda de símbolos, entre otras, se realizan en el momento del enlazado; mientras que, con una biblioteca dinámica, se realizan muchas de estas acciones en tiempo de ejecución.
4. En cuanto a compatibilidad y adaptación a actualizaciones de las bibliotecas; los programas asociados a bibliotecas dinámicas podrán recibir automáticamente las actualizaciones correspondientes de la biblioteca. Esto puede suponer un problema si las nuevas incorporaciones son incompatibles con el uso del programa. Esta situación se evita con bibliotecas estáticas que, aunque exijan un recompilado de las bibliotecas al sufrir

estas cambios para que entren en acción, no provocan este tipo de problemas, ya que todo el código está contenido en la aplicación.

11.Configuraciones propias de Visual Studio

11.1 Opciones de inclusión de ficheros en Visual Studio

Durante el desarrollo del proyecto surgieron problemas debido a la inclusión de ficheros de código no deseados en lugar de los ficheros a incluir. Esto es debido a que algunos de estos ficheros como puede ser *RTDeskMonitor.h* se encuentran repetidos en diferentes carpetas, tanto en la raíz del proyecto, donde se trabaja sobre ellos, como en una carpeta llamada *include* que se encuentra en un nivel superior en este caso.

Esta duplicidad de archivos daba lugar en ocasiones a la inclusión por parte de algunos ficheros de código de otros ficheros no deseados, incluyendo la versión almacenada en *include* y no la deseada de la raíz del proyecto con la cual se deseaba trabajar. Para solucionar este error los pasos a seguir son los siguientes:

1. Acceder al menú 'Propiedades' del proyecto.

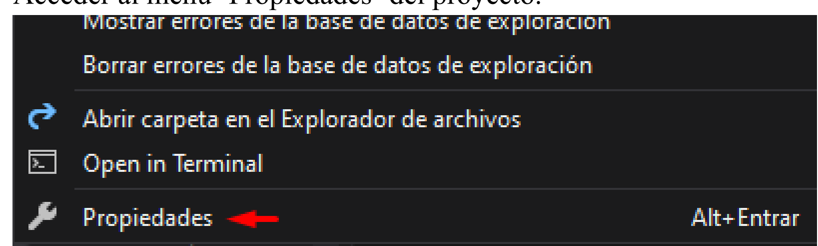


Ilustración 59: Menú propiedades en VSTUDIO

2. Dentro de 'Propiedades de configuración' acceder a 'Directorios de VC++'.

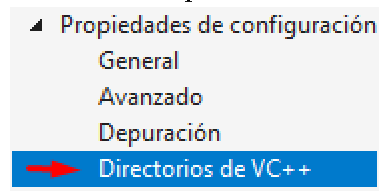


Ilustración 60: Propiedades de configuración en VSTUDIO

3. En 'Directorios VC++' sobre el campo 'Directorios de archivos de inclusión' seleccionar la opción editar, que continúa con siguiente ventana:

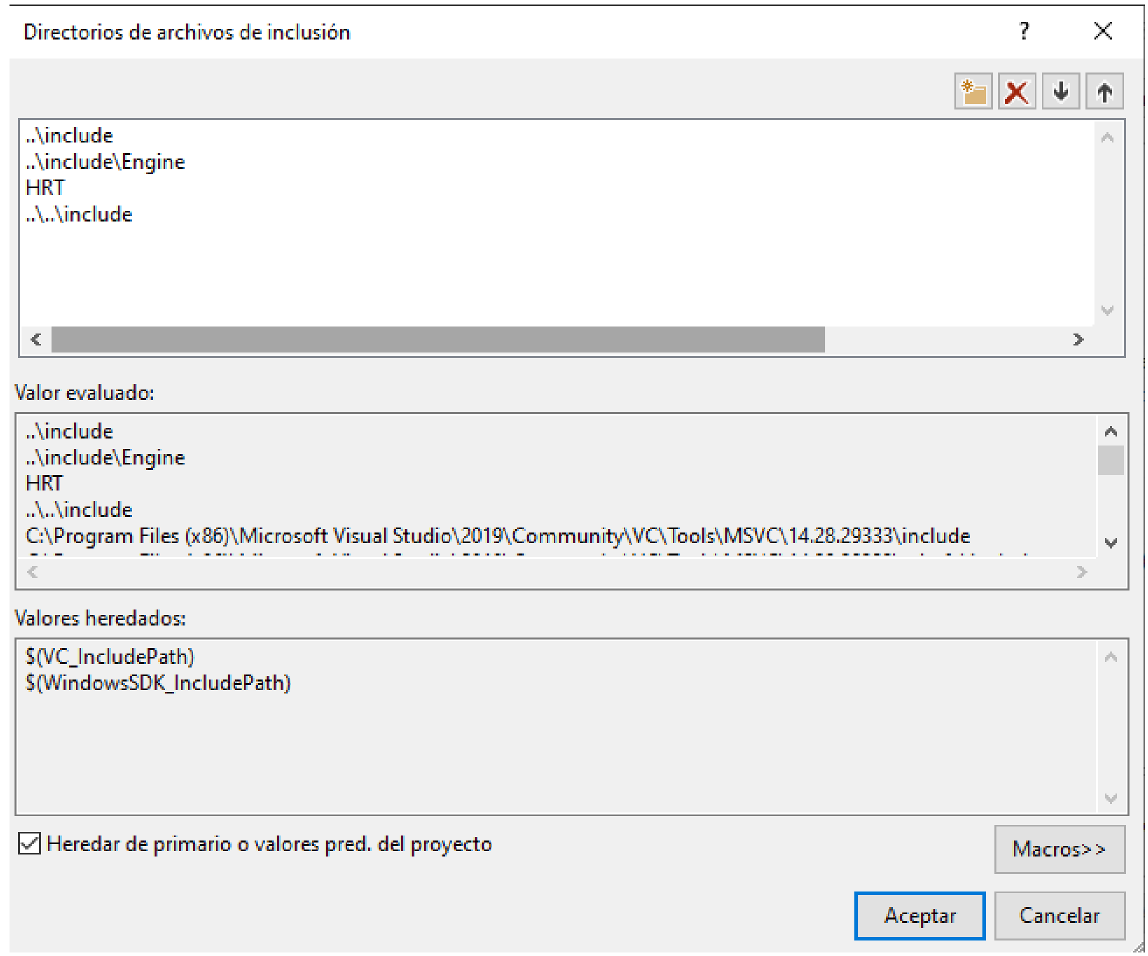


Ilustración 61: Edición de directorios a incluir en VSTUDIO

En ella se puede, en la parte superior, añadir rutas de ficheros. Así, el proyecto tendrá en cuenta a estos directorios a la hora de inclusión de ficheros en código. También se permite cambiar el orden de estos directorios y eliminar establecidos.

11.2 Actualización de la 'Platform toolset' de un proyecto en Visual Studio

El siguiente procedimiento se ha aplicado sobre el proyecto ya que disponía de un conjunto de herramientas desactualizado. Para ello se seguirán los siguientes pasos:

1. Cambiar el *Platform toolset* a la versión deseada, en este caso Visual Studio 2019 (v142), esto provocará una gran cantidad de errores a la hora de compilar el código.
2. Para solventar más fácilmente dichos errores se desactivan primero dos opciones del entorno, /*permissive* y *code analysis situadas en Project > Properties > Configuration Properties > C/C++ > Language y Project > Properties > Configuration Properties > Code Analysis* respectivamente.
3. Para solventar errores al enlazar archivos con diferentes versiones (_MSC_VER), se ha de actualizar la versión del simulador cambiando momentáneamente el *Platform toolset* de nuevo a la versión antigua de Visual Studio 2012 en el proyecto de RT-DESK. Tras ello, encima de la solución del proyecto, dentro de su menú desplegable, se accede a la opción 'redestinar proyecto' lo cual nos permitirá actualizar el conjunto de herramientas de la plataforma.

```
Actualizando el proyecto 'DemoJuegoVida_RtDesk'...
    Configuración 'Debug|Win32': cambiando el conjunto de herramientas de la plataforma a 'v142' (era 'v110').
    Configuración "Debug|x64": no se requiere actualización del conjunto de herramientas de la plataforma. El conjunto de herramientas de la plataforma es "v142".
    Configuración "Release|Win32": no se requiere actualización del conjunto de herramientas de la plataforma. El conjunto de herramientas de la plataforma es "v142".
    Configuración "Release|x64": no se requiere actualización del conjunto de herramientas de la plataforma. El conjunto de herramientas de la plataforma es "v142".
Redestinando extremo: 1 completado, 0 dio error, 0 omitido
```

Ilustración 62:Salida de consola al actualizar el platform toolset

12. Apéndice 3: Desarrollos de la solución descartados/no implementados

12.1 Implementación de compilación condicional

Esta estrategia de trabajo consiste en:

Para aquellas clases que jueguen un papel en la obtención de datos de interés, disponer de dos implementaciones de una misma cabecera. Una de las implementaciones correspondería a la clase original con sus funcionalidades básicas; mientras que la segunda versión sería la que posea funcionalidades extra para llevar a cabo la obtención de datos necesarios en la monitorización. Mientras tanto, las modificaciones a la cabecera necesarias para la monitorización, si las hubiese, se definirían en la misma, pero contenidas en una cláusula sólo activada en caso de haberse especificado el modo de monitorización como el actual.

Los pasos que seguir para preparar esta mecánica de trabajo serían los siguientes:

- Codificar la definición de una constante *RTDESK_USE_MONITORING,* la cual estará activa si se desea que el simulador se lance en modo de monitorización e inactiva en caso contrario.
- Incorporar cláusulas '#ifdef RTDESK_USE_MONITORING' a aquellas partes del código o ficheros de código que sólo deban lanzarse en modo de monitorización; reservando así el coste de compilación y ejecución adicional de las funciones de monitorización para la versión monitorizada. Además, se mantiene el rendimiento normal de la versión original del simulador.

```
//#define RTDESK_USE_MONITORING

//For exporting functionality through a lib file
#define RTDESKENGINE_LIB

//Copy this everywhere you need monitoring
#ifdef RTDESK_USE_MONITORING
#else
#endif
```

Ilustración 63:Definición y uso ejemplo de la constante de monitorización en RTDeskDefCom.h (en este caso desactivada)

Así se conseguiría por lo tanto disponer de dos versiones de un mismo objeto, una monitorizada y otra no monitorizada. Estas versiones se podrían alternar gracias al estado de la constante *RTDESK_USE_MONITORING*. Esta implementación fue descartada porque se consideraba la alternativa implementada en este proyecto más intuitiva, eficiente y escalable.

12.2 Guía para la incorporación del nuevo motor al entorno de pruebas

A grandes rasgos, para poder incorporar el motor RT-DESK (4. Cebrián 2013):

1. Primero se ha de generar el archivo biblioteca correspondiente
2. Tras esto se han de incluir los archivos de RT-DESK que necesitará el programa simulación
3. Siendo accesibles estos archivos se deberán introducir cabeceras de algunos de los archivos de código de la simulación y hacer uso de la nueva estructura de RT-DESK; cambiando la interacción normal que habría con el motor con aquella que permita la monitorización y devolución de datos de interés.

Antes de realizar este proceso sobre el proyecto *DemoGameOfLifeRTDesk,* se ha decidido realizar una copia de este llamada *DemoGameOfLifeMonitorRTDesk1*. Será sobre esta copia sobre la cual se realizará la implementación del motor y el lanzamiento de pruebas para obtención de datos. Esta decisión se ha tomado para evitar la posibilidad de generar errores en un proyecto que funciona sin generar errores; además, es recomendable utilizar esta versión anterior de RT-DESK como un punto de referencia muy útil al que retroceder en caso de generar errores en la nueva implementación.

Para realizar esta copia sería posible directamente, desde el explorador de archivos de nuestro sistema, copiar la carpeta contenedora del proyecto y pegarla en otro directorio. Tras de entrada probar a realizar la copia de esta manera, se generaron diversos errores provenientes de archivos siendo usados por ambos proyectos, no siendo estos totalmente independientes. A fin de evitar esta situación y realizar una copia de manera más limpia, se decide crear una plantilla a partir del proyecto *Game of Life* base. Será a partir de esta plantilla de la que se creará un nuevo proyecto *DemoGameOfLifeMonitorRTDesk1*, los pasos seguidos son:

- Ir a Visual Studio > Proyecto > Exportar plantilla

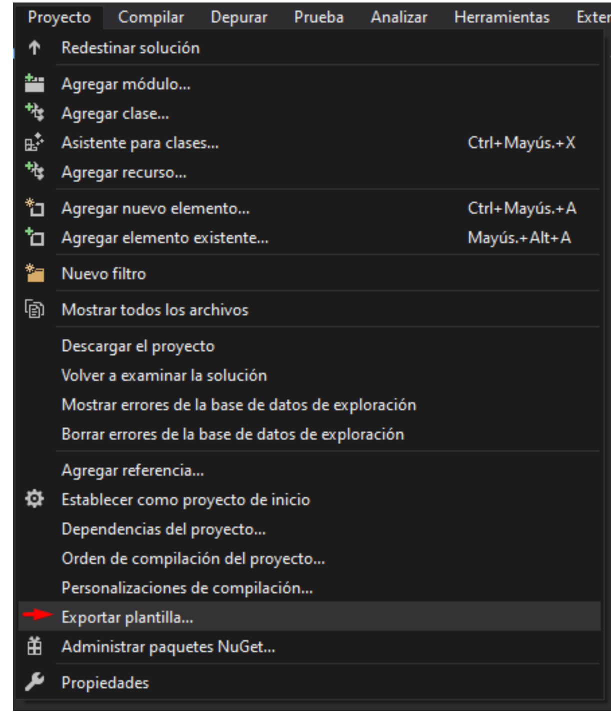
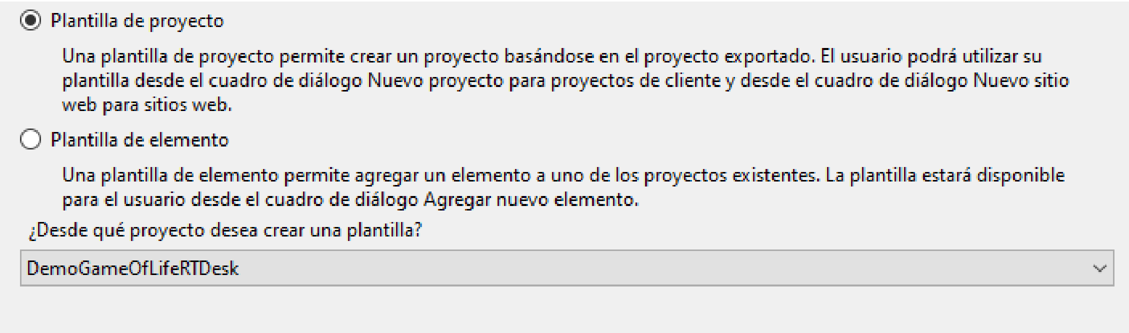


Ilustración 64: Paso 1 creación de plantilla

- Generar plantilla de proyecto sobre *Game of L*ife



Paso 2 creación de plantilla

- Crear un proyecto a partir de la plantilla, seleccionando esta como tipo de proyecto a crear.

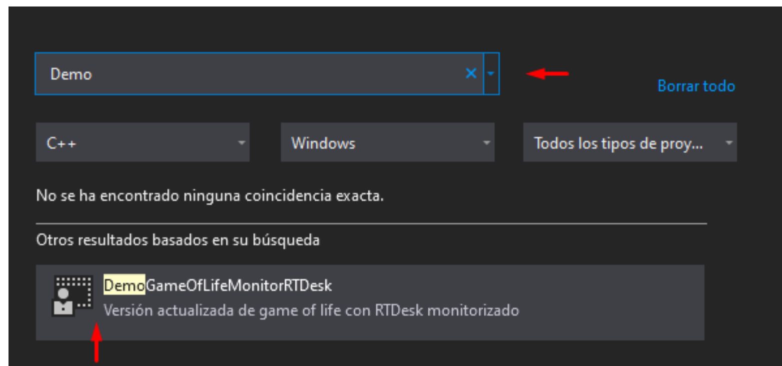


Ilustración 65: Paso 3 creación de plantilla

- Por último, incorporar ficheros excluidos de la replicación. Al tratar de compilar el proyecto replicado este nos avisará de la falta de algunos archivos importantes de RT-DESK a través de errores en la búsqueda de cabeceras. Tras añadir al directorio correspondiente del proyecto los archivos necesarios el proyecto, queda compilado con éxito.

Disponiendo del proyecto donde implementar el motor, se abre en una solución a parte del original para evitar referencias a versiones antiguas de los archivos. Ahora, sabiendo el curso de acción general a tomar, se realizan los siguientes pasos concretos para implementar la nueva versión de RT-DESK al proyecto *DemoGameOfLifeMonitorRTDesk1;* estos pasos se han realizado gracias al conocimiento de las modificaciones realizadas en el motor; comparaciones con el proyecto *Game of Life* original y las utilidades ejecutadas en él del antiguo motor. Además, ha resultado de ayuda la información contenida en la documentación de RTDESK[10]:

1. Incluir la carpeta usuario en el proyecto. Esta carpeta contiene las cabeceras necesarias para realizar la simulación a través de RT-DESK, además de las librerías *RTDeskEngine.lib* y *RTDeskEngine.dll* y de los archivos que contienen los temporizadores. En el caso de este proyecto las carpetas para incluir a RT-DESK se encuentran organizadas de manera diferente que en la carpeta *user* del motor; se respetará la organización de las carpetas de *Game of Life* para evitar algunos de los errores de referencia a directorios.
2. Tras esto, en el proyecto surgirán gran número de errores debido al cambio de lugar de algunos de los nuevos ficheros implementados. Para solventar estos errores se debe editar la configuración del proyecto, en concreto se tendrá que acceder al menú del proyecto *Propiedades > Propiedades de configuración > C/C++ > General > Additional Include Directories* aquí se añadirán como directorios de inclusión *.\RT-DESK\HRT, .\RT-DESK* y *.\RT-DESK\user.*
3. Habiendo solucionado los errores de referencias se presentan nuevos errores, entre ellos:
    a. Errores de acceso a variables protegidas como es el caso de *Timers* en *cHRTimerManager.* Se cambiará el acceso directo a la variable *Timers* por una llamada al método 'GetTimer()', que nos devuelve un puntero al temporizador solicitado, por lo que también se tendrán que cambiar las acciones que se hagan sobre estos temporizadores por acciones sobre punteros.
    b. Errores a la llamada 'SetMsgDispatcher()'. Previamente *RTDeskEntity* contenía un puntero a *MsgDispatcher*, por lo que todo objeto que heredase de entity poseía por herencia este puntero y una función para iniciarlo. Con la nueva estructura este atributo ha desaparecido y ha sido sustituido por un puntero a *Engine* que es el que contiene dichos punteros junto a funciones para definirlos y devolverlos. Por lo tanto, en el caso de que los errores se originen en un intento de establecer el puntero de estos objetos a *Dispatcher,* estas llamadas se pueden eliminar. Cuando se quiera acceder a dicho puntero se hará a través de *Engine*. Además, habrá que modificar aquellos métodos que tomen como parámetro un puntero a *Dispatcher,* eliminando este parámetro, así como modificar las llamadas que se hagan a estas funciones.

10 :[**RTDESK Documentation, User manual.** Disponible en: /RTDESK/Documentation/User manual.docx]

c. Errores por intentar acceder a los atributos *Dispatcher* y *PoolManager* de *Engine* de manera directa. Basta por cambiar estos accesos por llamadas al método *Get* del atributo correspondiente.

d. Accesos al atributo *Configuration* de *Engine* sustituidos por la llamada a la función correspondiente ahora contenida en *Engine* como es para 'RTDESK_Engine->Configuration->StoreSlackTime = x' por 'RTDESK_Engine->SetStoreSlackTime(x)'.

e. Borrar llamadas de establecimiento de tipos innecesarias sobre *PoolManager* , eliminando también llamadas a 'SetMsgType()' sobre *PoolManager* con tres argumentos, ya que esta sobrecarga de la función fue eliminada del motor.

f. Cambiar llamadas a 'DestroyTimers()' de *cHRTimerManager* por llamadas a 'DestroyAllTimers()'.

g. Cambiar el atributo contenedor de los nombres de temporizadores a *HRT_String*.

h. Tras esto, aparecerán errores de enlazado, se comenzará solucionando los generados por una diferencia de configuración entre proyectos. El motor de RT-DESK está configurado de manera que su biblioteca en tiempo de ejecución está especificada como DLL de depuración multiproceso (/MDd) mientras que la del proyecto *Game of Life* está definida como depuración multiproceso (/MTd). Para solucionarlo se cambia la configuración en el proyecto a monitorizar.

i. Errores relacionados con la creación de mensajes. Estos mensajes se deben a la sustitución del archivo *UserMessage.h,* para solucionarlos se ha de recuperar la versión de esta cabecera perteneciente al proyecto *Game Of Life* y cambiarla por la nueva o sustituir el código de esta por el original.

Tras haber solucionado estos errores, al tratar de lanzar el programa, que aún no da uso de las nuevas funciones monitorizadas, se produce una excepción de infracción de acceso al leer una ubicación de la memoria al tratar de crear un nuevo *Engine.* Para solucionar esta situación es de gran ayuda *debuggear* el código para encontrar la fuente del error. Para ello primero se ha de cambiar una configuración en el proyecto de pruebas donde se solucionará la base del error; esta configuración se encuentra en Propiedades de este proyecto > Propiedades de configuración > Vinculador > Depuración > Generar información de depuración y seleccionar 'Generar información de depuración optimizada para compartirla y publicarla (/DEBUG:FULL)'. Esto permite que a la hora de generar los archivos de símbolos .pdb necesarios para la depuración se generen documentos completos con todos estos símbolos en vez de enlazar con los ficheros originales como haría '(/DEBUG:FASTLINK)'. Para finalizar con la incorporación del nuevo motor a este proyecto se deberían solucionar los errores de lanzamiento y posteriormente asegurarse de que se hace uso de las nuevas utilidades por parte del programa.