UNIVERSIDAD POLITECNICA DE VALENCIA

Del videojuego continuo acoplado al discreto desacoplado

Una mejora del ciclo principal del videojuego

Héctor Martínez Cebrián

Dirigido por:
Ramón Mollá

Valencia, 2013

TRABAJO FIN DE MASTER

Máster universitario en inteligencia artificial, reconocimiento de formas e imagen digital.

Departamento de sistemas informáticos

¡Mirad detrás! ¡Un mono de tres cabezas!

Guybrush Threepwood – Monkey Island 2: LeChuck's Revenge

No hay genio sin un gramo de locura.

Aristóteles

Índice

Índice 2
1 Introducción 4
1.1 Motivación 8
1.2 Objetivos 9
1.3 Esquema de la obra 9
2 Estado del arte 11
2.1 Critica al estado del arte 23
2.1.1 Ejecución ordenada de eventos 23
2.1.2 Ejecución de eventos previamente cancelados 25
2.1.3 Frecuencia constante 26
2.2 Modelo continuo acoplado temporizado 28
2.3 MCAT con tiempo de simulación 29
2.4 Propuesta de mejora 29
3 Propuesta de trabajo 30
3.1 Análisis DAFO 33
3.1.1 Modelo continuo acoplado 33
3.1.2 Modelo continuo acoplado temporizado 37
3.1.3 Modelo discreto desacoplado 41
4 RT-DESK 44
4.1 Aportación de la librería 46
4.2 Diseño estático 48
4.2.1 Núcleo 48
4.2.2 Despachador de mensajes 48
4.2.3 Entidad RT-DESK 51
4.3 Funcionamiento dinámico 51
4.3.1 Inicialización 51
4.3.2 Ejecución del simulador 51
4.3.3 Envío de mensajes 52
4.3.4 Recepción de mensajes 52
5 Banco de pruebas 53
5.1 Historia 54
5.2 Diseño previo 55
5.2.1 Diseño estático 55
5.2.2 Funcionamiento dinámico 58
5.3 Diseño actual (Modelo continuo acoplado) 60
5.3.1 Cambios estáticos 60
5.3.2 Cambios dinámicos 60
5.4 Diseño actual (Modelo continuo acoplado temporizado) 61
5.4.1 Cambios estáticos 61
5.4.2 Cambios dinámicos 61
5.5 Diseño actual (Modelo discreto desacoplado) 62
5.5.1 Cambios estáticos 62

5.5.2 Cambios dinámicos....63
6 Experimentación 65
6.1 Rendimiento del sistema 65
6.1.1 Objetivos del estudio....65
6.1.2 Prueba a realizar....65
6.1.3 Resultados 66
6.1.4 Conclusiones....69
6.2 Precisión del estado final 70
6.2.1 Objetivos del estudio....70
6.2.2 Prueba a realizar....70
6.2.3 Resultados 71
6.2.4 Conclusiones....72
6.3 Obtención de tiempo ocioso.... 72
6.3.1 Objetivos del estudio....72
6.3.2 Prueba a realizar....72
6.3.3 Resultados 73
6.3.4 Conclusiones....74
6.4 Exactitud del tiempo de simulación.... 75
6.4.1 Objetivos del estudio....75
6.4.2 Prueba a realizar....75
6.4.3 Resultados 76
6.4.4 Conclusiones....76
7 Conclusiones y resultados.... 78
7.1 Conclusiones finales.... 80
8 Trabajos futuros.... 81
9 Agradecimientos 83
10 Bibliografía 84

1 Introducción

Un videojuego consiste en un juego electrónico en el cual el jugador interactúa mediante uno o varios controladores, denominados *periféricos*, con el sistema y este devuelve información pudiendo estar compuesta por imágenes, sonidos, vibraciones o sensaciones. Los videojuegos se encuentran en las categorías de simuladores visuales y aplicaciones gráficas de tiempo real.

Los videojuegos iniciaron su andadura a finales de la década de los 70. Desde entonces, han evolucionado continuamente y en mayor medida cuanto más tiempo transcurre. Las rudimentarias imágenes compuestas por simples líneas han evolucionado a entornos 3D, los sonidos se vuelven más realistas cambiando los simples pitidos en melodías compuestas por orquestas o sintetizadas por ordenadores, los objetos inteligentes cada vez tienen comportamientos más avanzados, y un sinfín de mejoras. Otro aspecto evolutivo a tener en cuenta es el gran abanico de géneros pues los avances tecnológicos han permitido el desarrollo convirtiendo aquellos primitivos estilos hasta los diversos géneros que se conocen hoy en día. Las videoconsolas, computadores, dispositivos móviles entre otros han evolucionado a la par de los videojuegos. (1)

Los videojuegos deben cumplir algunos requisitos que se pueden encontrar en el denominado **principio de jugabilidad**. En este principio se detallan diversas características que los videojuegos deben cumplir como la atractividad, el aprendizaje, la inmersión o la precisión entre otros. El principio de jugabilidad marca una serie de características que los desarrolladores y analistas deben tener en cuenta para valorar un videojuego. Cuantas más características del principio de jugabilidad cumplan, mayor será la calidad del videojuego y más distará respecto a un Software Interactivo Tradicional (SIT). (2)

Además, existen una serie de normas ISO que contemplan y evalúan la **calidad del software**. En su versión más reciente (véase: ISO/IEC-25010-3, 2009) el proceso de

valoración del software recoge las contribuciones de las normas ISO anteriores y los divide en dos líneas principales: **Calidad del producto software** y **calidad del software en uso**. La división **calidad del producto software** tiene en cuenta la adecuación funcional, confiabilidad, eficiencia del rendimiento, operatividad, seguridad, compatibilidad, mantenimiento y transferencia del software. Estos factores evalúan la ejecución y calidad del software desde el punto de vista del sistema. La división **calidad del software en uso** evalúa el software valorando la flexibilidad en uso, facilidad de uso y protección. Esta división evalúa el sistema desde el punto de vista del usuario y su interacción con el sistema.

Tanto las distintas marcas de videoconsolas como los productores desean que un videojuego asegure un mínimo de calidad. A esto se le denomina calidad de servicio y forman aquellos requisitos que deben cumplir los videojuegos para mejorar los distintos aspectos del principio de jugabilidad (2). A partir de ahora se nombrará como **QoS (Quality of Service)**. Entre las propiedades de la QoS cabe destacar las siguientes:

- *Latencia de respuesta*: Es el tiempo invertido desde que el usuario realiza alguna acción desde un periférico hasta que el sistema ejecuta la tarea correspondiente.

- *Nivel de detalle gráfico*: Los detalles gráficos se determinan mediante la resolución de las mallas poligonales.

- *Frecuencia de muestreo*: Es la cantidad de ejecuciones por unidad de tiempo (habitualmente la unidad de tiempo es el segundo) de cada servicio del sistema.

- *Calidad de la IA*: Nivel de desarrollo de la IA a nivel computacional buscando soluciones más elaboradas. A mayor elaboración, mayor la calidad de la IA.

Los desarrolladores de videojuegos no pueden especificar una QoS concreta dado que no se puede determinar si cada una de las fases se podrá ejecutar de manera correcta con una frecuencia determinada. Además, tampoco pueden especificar una QoS global debido al mismo problema.

Todo sistema que implementa un videojuego gira en torno a un **bucle principal**. Este bucle está dividido en tres fases, la fase de gestión de entrada, la fase de actualización y la fase de presentación. En la **fase de gestión de entrada** se procesan los eventos generados por el usuario como pueden ser la pulsación de una tecla o un movimiento capturado por una cámara. También se procesan los mensajes recibidos de la red o los movimientos del dispositivo mediante la lectura de sus correspondientes acelerómetros. Durante la **fase de actualización**, el sistema varía en función de los eventos realizados por el usuario (teclas presionadas, movimiento del ratón, comando de voz,…) y del estado anterior del sistema aplicando una serie de normas físicas o de comportamiento del sistema entre otras. El estado del sistema hace referencia a la condición de cada uno de los elementos que componen el sistema. En la **fase de presentación** se generan las imágenes, los sonidos, las vibraciones o cualquier sensación de respuesta dirigida al usuario (3). Estas tres fases se ejecutan cíclicamente de manera que el estado del sistema evoluciona a cada ciclo.

El método de ejecución del ciclo principal se puede clasificar por la manera en la que evoluciona el sistema. En esta clasificación se encuentran los simuladores discretos y los simuladores continuos.

Los **simuladores discretos** son aquellos en los que el paso del tiempo de simulación no se corresponde con el tiempo real. Es decir, el tiempo de simulación del sistema puede evolucionar a una velocidad mayor o menor que la realidad. Esta clase de simuladores funcionan mediante eventos o sucesos. Los eventos son generados por las distintas entidades que componen el sistema, por algún periférico de entrada o por alguna comunicación procedente de la red. Cada vez que se genera un evento, se ejecuta una vez el ciclo. Este tipo de simulación se emplea en los casos en los que

prima la calidad frente a la velocidad, realizando cálculos más precisos y completos, pudiendo mostrar los resultados de manera más pausada. Estos simuladores son comúnmente empleados en simulaciones para estudios físicos, químicos, sociales,... (4)

Por el contrario, los **simuladores continuos** son aquellos que evolucionan a la misma velocidad que el tiempo real. En estas simulaciones el avance no lo señala la ejecución de eventos sino que se ejecuta el sistema continuamente avanzando según el coste temporal de las fases empleadas en el ciclo. En estas simulaciones hay que tener precaución en que el coste de las fases no sea demasiado grande como para ralentizar la salida y evitar así una sensación de bloqueos en la simulación. Este tipo de simulación se emplea en los casos en los que prima la velocidad a la calidad, suprimiendo calidad en los cálculos con el fin de que la respuesta del sistema sea lo más continua posible. La simulación continua se emplea en aplicaciones de realidad aumentada, videojuegos,...

La ejecución de manera invariable de las tres fases del ciclo principal se denomina **ciclo acoplado**. Como alternativa, se puede hablar de los ciclos desacoplados. Los **ciclos desacoplados** son aquellos en los que la ejecución de una de las fases del ciclo no implica la ejecución de las fases restantes. Es decir, la frecuencia de las distintas fases no tiene por qué ser la misma de manera que la frecuencia de presentación podría ser menor que la frecuencia de las fases de actualización y gestión de entrada y con ello, se evita cargar la CPU con cálculos innecesarios de presentación.

Los **motores de videojuego** son un conjunto de código de programación que aportan al sistema diversos servicios tales como gráficos, sonidos, manejadores, parsers, lógica, etcétera. La mayoría de los motores han sido diseñados para una única clase de videojuegos de manera que los servicios se encuentran adecuados a las exigencias del género del videojuego. Aun así, también se pueden encontrar motores de videojuegos versátiles que intentan abarcar gran variedad de géneros. Habitualmente, estos motores incluyen su propio ciclo principal de manera que el programador se limite a realizar los mínimos cambios necesarios para adaptar el motor

a los conceptos de su videojuego. El empleo de estos motores evita horas de desarrollo y posterior comprobación además de asegurar una gran compatibilidad e integridad dentro del sistema.

Los motores de videojuegos pueden estar destinados tanto para ser ejecutados en sistemas operativos como para videoconsolas. Los motores destinados a computadores permiten una configuración por parte del usuario para adaptar el videojuego a las especificaciones del sistema dado la gran variedad de computadores del mercado. De esta manera se pueden especificar qué servicios son más importantes y lograr que el sistema se muestre de manera continua. Una mala configuración puede hacer que la ejecución se muestre con pausas e incluso inhabilite el sistema con grandes cálculos. Para el caso de los videojuegos para consolas, las configuraciones vienen determinadas por la empresa desarrolladora. En pocos de estos videojuegos se permite al usuario realizar algunos cambios con respecto a estas configuraciones pero aportan pocas opciones.

1.1 Motivación

Se busca una clase de ejecución que permita asignar una QoS específica para cada una de las secciones que conforman las fases del ciclo principal y por ello, también conforman los videojuegos. Esto implica que es necesario cumplir con una frecuencia mínima para cada sección.

Para comenzar, emplear un ciclo desacoplado permitirá ejecutar aquellas secciones necesarias evitando sobrecargar al sistema con cálculos innecesarios. Por estas ventajas, sería conveniente emplear el ciclo desacoplado.

Si además de un ciclo desacoplado se emplea un ciclo discreto permitirá determinar de manera más exacta la frecuencia de cada sección. Asimismo, la frecuencia puede variarse dinámicamente a lo largo de la ejecución del sistema ajustándose a las condiciones y necesidades. En el peor de los casos, si se desean ejecutar varios módulos al mismo tiempo, el sistema se ralentiza de la misma manera

que en el ciclo continuo. Aun así, el ciclo discreto desacoplado permite el empleo de los recursos del sistema de manera mucho más eficiente y esto conlleva un mejor complimiento de las QoS.

En el año 2012 se realizó un proyecto sobre el empleo de la ejecución discontinua desacoplada en Aplicaciones Gráficas en Tiempo Real (AGTR) obteniendo gratos resultados (5). En esta tesina se pretende ir más allá empleando el nuevo método de ejecución de manera conveniente a una determinada sección de Aplicaciones Gráficas como son los videojuegos.

1.2 Objetivos

El objetivo es mejorar el rendimiento de un videojuego. Para ello se combinará un simulador discreto desacoplado con un videojuego continuo acoplado uniéndolos de manera que se pueda seleccionar si se desea ejecutar el videojuego con un método de ejecución discreto o continuo.

Se analizarán y confrontarán los resultados obtenidos por ambos videojuegos (tanto el videojuego continuo acoplado como el discreto desacoplado) de forma que se observen las ventajas y desventajas de cada uno de los métodos de ejecución. Además, estos resultados se contrastarán con los obtenidos en (5) y con los obtenidos anteriormente en (6) pese a la gran evolución de los sistemas en el tiempo transcurrido.

1.3 Esquema de la obra

En el capítulo 1 se ofrece una introducción a la dualidad de modos de ejecución y la calidad del videojuego. El capítulo 2 refleja el panorama actual sobre los modelos del bucle principal del videojuego. A continuación, en el capítulo 3 se proponen varios aspectos a verificar en este proyecto y un análisis DAFO de cada uno de los modos de ejecución a contemplar. Seguidamente, el capítulo 4 detalla la librería RT-DESK (librería que implementa el método de ejecución discreto desacoplado) describiendo tanto su

funcionamiento como su implementación. Para realizar el banco de pruebas se emplea el videojuego Space Invaders por ello en el capítulo 5 se realiza un repaso a la historia de Space Invaders y su funcionamiento además de detallar la implementación de este. En ese mismo capítulo se informa de los cambios necesarios en el código del videojuego para poder realizar las pruebas pertinentes y formar así un banco de trabajo. El capítulo 6 muestra los resultados de la experimentación para en el capítulo 7 extraer conclusiones. En el capítulo 8 se expresan futuras líneas de investigación por las que se expanden las distintas ramas que aparecen a partir de esta investigación. En el capítulo 9 se agradece el trabajo de aquellas personas que tanto directa como indirectamente han influido en la buena finalización de este proyecto. Por último, en el capítulo 10 se informa de la bibliografía empleada.

2 Estado del arte

En los inicios de este arte, los videojuegos eran codificados íntegramente desde cero y todo el código desarrollado servía únicamente para un videojuego. Este método de programación se pudo utilizar mientras los sistemas no necesitaban grandes ficheros de código pues eran videojuegos sencillos tanto lógica como gráficamente. En 1993 apareció un videojuego que marcaría el inicio de una nueva era. Doom abrió nuevas expectativas. Aunque ya existieran videojuegos en primera persona, ninguno de los anteriores se había desarrollado desde una unidad básica de videojuego al que tras añadir varias especificaciones formase un videojuego completo. Aquella unidad básica de videojuego es lo que hoy se conoce como **motor de videojuego**. Aunque Doom engine presentaba unos gráficos 2D y empleaba algunos trucos para generar sensaciones 3D, pronto aparecerían motores de videojuego que creasen escenarios 3D como Quake engine (7).

En la actualidad la inmensa mayoría de videojuegos emplean un motor de videojuego para manejar todos los eventos y las distintas secciones de los que están compuestos los videojuegos. Además estos motores de videojuego permiten portar los videojuegos a distintas plataformas sin necesidad de realizar grandes cambios. Los motores de videojuego también integran en su núcleo un ciclo principal. En la Tabla 1 se muestran los principales motores y varios videojuegos desarrollados con cada uno de ellos.

Motor	Videojuegos
Unreal Engine 3	Saga Gears of War, Unreal Tournament 3, Saga Mass Effect, Rainbow Six Vegas
Crystal Tools	Final Fantasy XIII, Final Fantasy XIII-2, Final Fantasy XIV, Dragon Quest X
CryEngine 3	Crysis 2 y 3, Sniper: Ghost Warrior 2, Warface
RAGE	Max Payne 3, Red Dead Redemption, Grand Theft Auto IV , V y Episodes From Liberty City

Id Tech 4	Doom 3, Prey, Quake 4, Wolfenstein
Anvil Engine	Saga Assassin's Creed, Prince of Persia: Las Arenas Olvidadas
A4 Engine	Metro 2033
Unity	Battlestar Galactica Online, WolfQuest, Tiger Woods PGA Tour Online
X-Ray Engine	Saga S.T.A.L.K.E.R.
Source Engine	Half-Life 2, Saga Portal, Saga Left 4 Dead, Saga Counter Strike

Tabla 1 – Principales motores y videojuegos desarrollados con dichos motores

En la mayoría de los motores citados anteriormente todo el sistema se basa en la ejecución de un bucle principal continuo y acoplado (8). En este bucle se ejecuta la gestión de entrada, la simulación del sistema y a continuación se genera la presentación de los datos como se ha descrito en el capítulo anterior.

Un videojuego que no cumpla los requerimientos puede hacer que se pierda interactividad y con ello, desaparezca la sensación de ejecución en tiempo real (2). Los diversos motores de videojuego, al igual que los estudios sobre la ejecución de simuladores, han intentado mejorar la ejecución de las fases del ciclo principal y así obtener un mejor rendimiento basándose en los diversos métodos de ejecución que emplean los simuladores.

Los simuladores más primitivos y sencillos de implementar son los que emplean el **modelo simple acoplado** (3). En este caso, las fases se ejecutan cíclicamente de manera continua en un orden especificado en la implementación. Primero se ejecuta la fase de gestor de entrada seguida por la fase de actualización del sistema y finaliza con la fase de presentación. Estas tres fases se ejecutan continuamente sin interrupción alguna de manera que el videojuego se actualiza y muestra información la mayor cantidad de veces posible (Ilustración 1). Si la plataforma en la que está siendo ejecutado el videojuego tiene una potencia mayor a la necesaria, el videojuego

generará muchas imágenes que no se mostrarán en pantalla pues dichas imágenes se generarán a una mayor frecuencia que el refresco del monitor, desperdiciando recursos del sistema y tiempo de procesamiento. Por el contrario, si la plataforma tiene potencia insuficiente el videojuego se ejecutara de manera ineficiente mostrando todo el sistema más ralentizado. Los videojuegos desarrollados por Unreal Engine son un ejemplo de sistema donde se emplea el modelo simple acoplado (9).

Este modelo evolucionó al **modelo sincronizado acoplado** (3). En este nuevo modelo, se sigue ejecutando en el mismo orden la gestión de entrada, la actualización del sistema y a continuación la presentación. La diferencia reside en un temporizador que calcula el tiempo que se requiere para ejecutar el ciclo de manera que se mantenga una frecuencia determinada y con ello deduce el tiempo restante permitiendo al sistema emplear dicho tiempo restante en otras aplicaciones o en gestiones del sistema operativo. Una vez transcurrido el tiempo de espera, el ciclo volverá a ejecutarse (Ilustración 1). Al insertar el temporizador, el ciclo principal se convierte en un sistema discreto. En este modelo se salva el problema generado por una plataforma capaz de ejecutar el sistema a una alta velocidad evitando que se generen datos de presentación que no puedan ser mostrados o reproducidos y actualizaciones del sistema innecesarias. De esta manera, la ejecución en máquinas distintas es similar. Además, el modelo presenta la posibilidad de determinar ciertas QoS globales (10).

La siguiente mejora introducida en los modelos del bucle principal fue la adhesión de un controlador que maneje la ejecución de la fase de actualización. Este es el **modelo simple desacoplado**. El concepto desacoplado conlleva que alguna fase se ejecute con una frecuencia distinta a la del resto de fases. El controlador es el que se encarga de ejecutar la fase de actualización según el tiempo del sistema (Ilustración 1). Con este modelo, además de determinar una frecuencia global del ciclo se logra limitar a una frecuencia menor o igual la ejecución de la fase de actualización. Por este motivo, este modelo permite asegurar una QoS global y una QoS más específica para la fase de actualización (11).

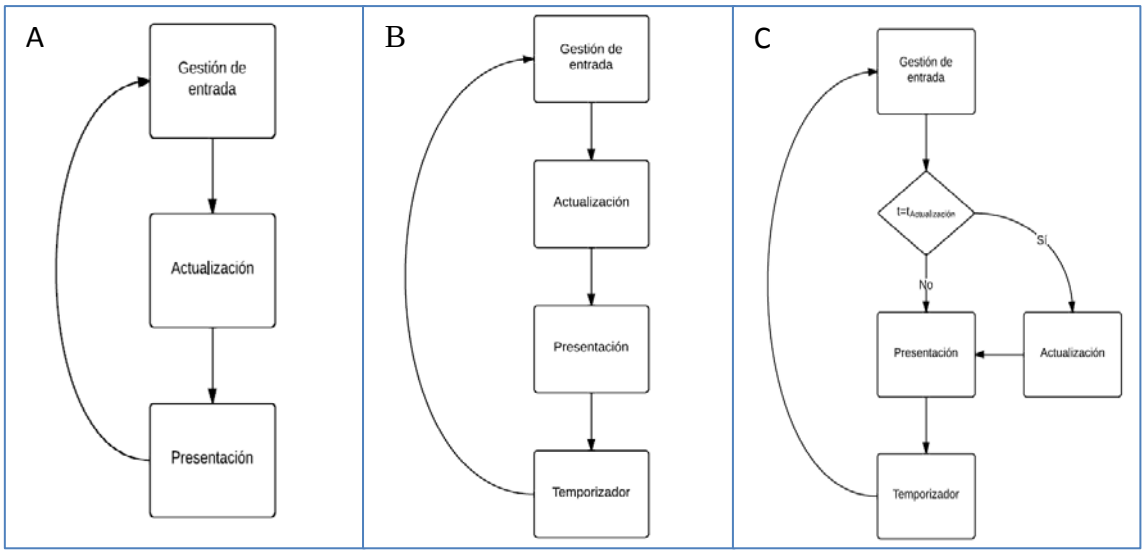


Ilustración 1 – Modelos simples: A – Modelo simple acoplado, B – Modelo sincronizado acoplado, C – Modelo simple desacoplado

Con la evolución de los procesadores se introdujo el concepto multihilo. Pese a que esta característica se encuentra desde 1985, incorporado en el Intel 80386, no se empleó hasta una década más tarde. Un hilo o hebra no es más que una tarea o subproceso que puede ser ejecutada al mismo tiempo que otra tarea. Un proceso está formado por uno o más hilos. El cambio entre hilos de un mismo proceso es más veloz que el cambio entre hilos de diversos procesos. Esto se debe a que los hilos de una misma aplicación comparten datos mientras que entre diferentes procesos los datos cambian por completo por lo que es necesario hacer operaciones de intercambio de datos entre una memoria más lenta, normalmente el disco duro, y una memoria más rápida.

En el **modelo multihilo desacoplado** existen dos hilos distintos, uno principal donde se ejecutan la gestión de entrada, la fase de actualización y un temporizador mientras que en un hilo secundario se ejecutará la fase de actualización (Ilustración 2). En este modelo, el hilo principal se ejecutará con una determinada frecuencia empleando el tiempo restante en la ejecución del hilo secundario. De esta manera, la

presentación se ejecutará con la mayor frecuencia posible evitando el efecto de ralentización (3).

Otro modelo más complejo que emplea la tecnología multihilo es el denominado **modelo desacoplado de frecuencia fija**. En este modelo separan la lógica de la aplicación de la fase de actualización. Este modelo también está compuesto por dos hilos. El hilo principal ejecuta continuamente un bucle en el que se encuentra la gestión de entrada y la lógica de la aplicación. Cuando se alcanza un momento t determinado se ejecuta la fase de actualización del sistema y un temporizador. Cuando el temporizador active la alarma volverá a ejecutarse de nuevo el hilo principal. El hilo secundario ejecuta cíclicamente la fase de presentación al igual que en el modelo multihilo desacoplado (Ilustración 2). Este modelo, además de las ventajas aportadas en el modelo anterior, permite gestionar la frecuencia de la fase de actualización de manera independiente al hilo secundario pero dependerá de la frecuencia del hilo principal (12).

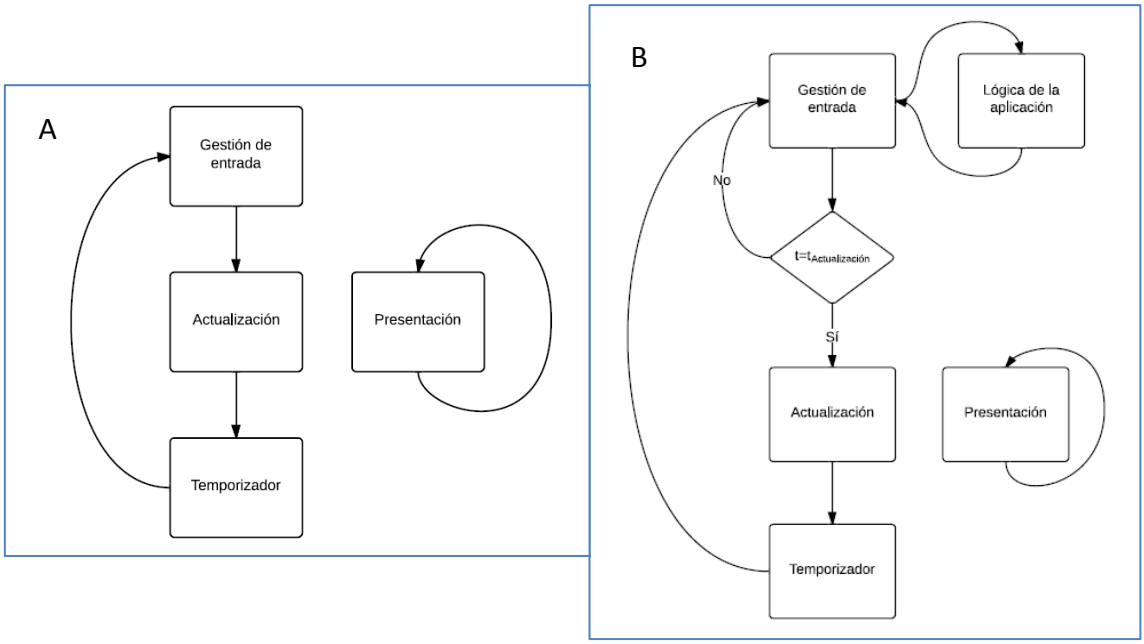


Ilustración 2 – Modelos multihilo: A – Modelo multihilo desacoplado, B – Modelo desacoplado de frecuencia fija

Con el paso de los años, las diversas empresas implicadas en el desarrollo de procesadores fueron incrementando la velocidad de procesamiento mediante nuevas arquitecturas y nuevos materiales de fabricación hasta alcanzar el límite. Al sentirse incapaces de incrementar la velocidad más allá de dicho límite optaron por la solución de incrementar el número de núcleos de procesamiento en cada procesador. Con la invención de los procesadores multinúcleo, se desarrolló de mayor manera las teorías de sistemas paralelos. Esta tecnología permite ejecutar varios procesos al mismo tiempo en el mismo procesador aumentando la capacidad de procesamiento y disminuyendo la sobrecarga del sistema. El paralelismo se puede aplicar de dos maneras, paralelismo respecto a funciones y paralelismo respecto a datos. En el primero, todos los núcleos reciben los mismos datos pero en cada núcleo se ejecutan distintas funciones. En el paralelismo respecto a datos, cada núcleo recibe una porción de los datos completos pero ejecutan el mismo código.

El primer modelo de bucle principal creado a partir de los modelos anteriores y la nueva tecnología fue el **modelo paralelo de funciones síncronas**. En este modelo, además de la lógica de la aplicación, se ha extraído los cálculos físicos de la aplicación de la fase de actualización. Se inicia con la fase de gestión de entrada continuada por la lógica de la aplicación. A continuación el sistema se divide en dos caminos distintos que serán ejecutados en paralelo. El cálculo de las físicas de la aplicación por un lado y la actualización por otro lado, conforman dicha división. Por último, ambos caminos se unen en la fase de presentación (Ilustración 3). Este modelo, permite mejorar el rendimiento del bucle principal primitivo de manera sencilla para casos en los que el procesador no pueda disponer de más de dos núcleos destinados para el videojuego (13).

Para microprocesadores más avanzados, con posibilidad de destinar tres núcleos a la ejecución del videojuego, se presentó el **modelo paralelo de funciones asíncronas**. Este modelo emplea un núcleo de procesamiento para cada fase (gestión de entrada, actualización y visualización) permitiendo que cada fase se ejecute a la

máxima frecuencia posible aunque las frecuencias de las fases no tienen razón para coincidir (14) (Ilustración 3).

Otro de los modelos desarrollados empleando el multiprocesamiento es el **modelo paralelo de datos**. En este caso, la ejecución del bucle se inicia con la fase de gestión de entrada. A continuación, la fase de actualización se ejecuta en todos los núcleos de procesamiento pero los datos son divididos entre dichos núcleos de manera que cada una de las ejecuciones no realizará los mismos cómputos y la unión de resultados será igual que si la fase de actualización fuese ejecutada en un único procesador. Después de esta división se ejecuta la fase de presentación en uno de los núcleos (13)(Ilustración 3).

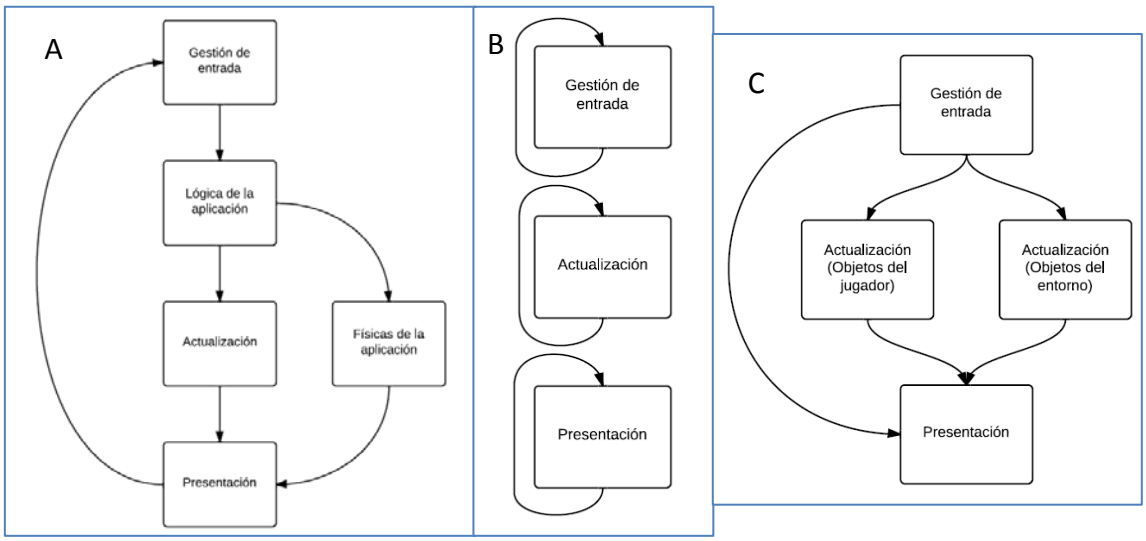


Ilustración 3 – Modelos paralelos: A – Modelo paralelo de funciones síncronas, B – Modelo paralelos de funciones asíncronas, C – Modelo paralelo de datos

Al no progresar en el campo de los microprocesadores se investigaron otros métodos para mejorar las prestaciones de los sistemas. Los investigadores comenzaron a emplear los GPU para sus propios cálculos y las creadoras de tarjetas gráficas apoyaron esta iniciativa desarrollando lenguajes de programación que facilitasen la implementación de código para ser ejecutado en la GPU. La GPU aporta una gran cantidad de unidades funcionales capaces de realizar operaciones en coma flotante.

Además, existe un gran paralelismo en las GPU por lo que una operación es capaz de ser ejecutada paralelamente para computar grandes volúmenes de datos independientemente. Las unidades funcionales de la GPU se pueden dividir en dos tipos fundamentales: los **procesadores de vértices** y los **procesadores de píxeles**. En los primeros, cada unidad funcional obtiene una cantidad de datos fija con la que deberá realizar las operaciones. Como salida pueden devolver tantos resultados como deseen. Esta unidad funcional tiene como principal desventaja que cada unidad no puede conocer información sobre el resto de datos o unidades funcionales, teniendo limitada la entrada de datos. Aun así, tiene la ventaja de poder procesar cada dato independientemente y devolver diversa información seleccionada por el programador. Las prestaciones de los procesadores de píxeles se podría decir que funcionan de manera contraria a los procesadores de vértices. En este caso, la entrada de información es variable para cada unidad funcional pero esta solo puede devolver una cantidad de datos fija. Además, en los procesadores de píxeles sí que se tiene conocimiento del resto de datos de entrada de los demás procesadores de píxeles.

Aunque las operaciones que realice una CPU pueden ser realizadas por una GPU, la estructura de la GPU produce la diferencia entre la ejecución en una CPU o en esta misma. Ciertas operaciones o algoritmos pueden ser óptimos en una GPU mientras que otros pueden ser óptimos en una CPU. Por lo tanto hay que determinar que operaciones se realizarán en cada una de ellas. Las aplicaciones que emplean la GPU como unidad de cálculo se denominan **General Purpouse GPU** (GPGPU).

Los modelos vistos anteriormente se adaptaron para poder emplear la tecnología GPGPU y sus respectivas ventajas. El modelo más sencillo que incorpora una fase de cálculo en la GPU es el **modelo simple acoplado con fase GPGPU**. En este modelo se incorpora una fase de cálculo en la GPU después de la fase de presentación en el modelo simple acoplado (Ilustración 4). De esta manera se destina la GPU para realizar cálculos que ya no se realizarán en la fase de actualización (15).

Otro modelo al que se le añadió una fase de cálculo mediante la GPU fue al modelo sincronizado acoplado. Este nuevo modelo se denominó **modelo sincronizado acoplado con fase GPGPU**. La fase de cálculo mediante la GPU se incorpora tras la fase de presentación pero previamente al temporizador (Ilustración 4). Al igual que en el modelo anterior, parte de los cálculos realizados en la fase de actualización pasan a ser realizados por la GPU (11).

El último modelo en incorporar una fase de cálculos en GPU en el hilo principal fue el **modelo simple desacoplado con fase GPGPU**. Este modelo se diferencia del modelo simple desacoplado en que se agrega una fase de cálculo en GPU tras la ejecución de la fase de presentación pero únicamente se ejecutará con una frecuencia determinada menor o igual a la frecuencia del ciclo (Ilustración 4). La fase de cálculo en GPU se acopla de la misma manera que se acopla la fase de actualización en dicho modelo (11).

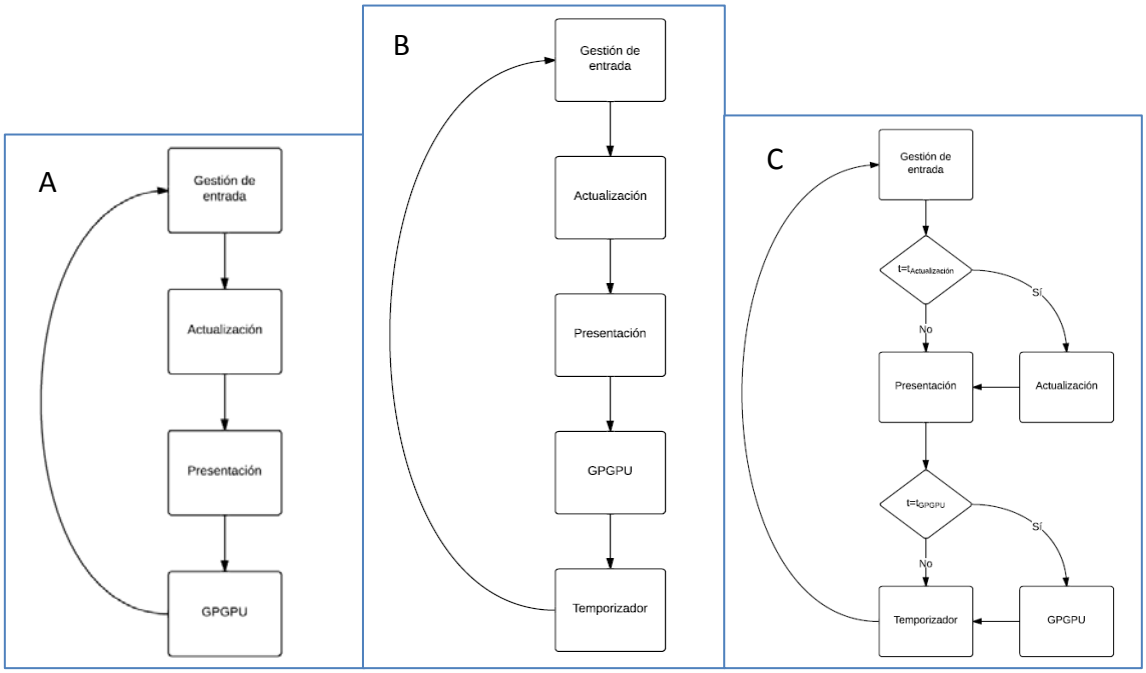


Ilustración 4 – Modelos simples con fase GPGPU: A – Modelo simple acoplado con fase GPGPU, B – Modelo sincronizado acoplado con fase GPGPU, C – Modelo simple desacoplado con fase GPGPU

Además de estos modelos en los que se acopla una fase GPGPU, varios modelos más agregaron esta fase de manera independiente. El primero de ellos fue el **modelo asíncrono con fase GPGPU desacoplada**. El modelo varía del modelo simple desacoplado en que la fase de cálculo con GPU se ejecuta en una hebra distinta de manera que continuamente se está ejecutando de forma aislada al resto de la ejecución (16).

Otro de los modelos en los que se realizó el acoplamiento de una fase GPGPU fue el **modelo síncrono acoplado con fase GPGPU desacoplada**. En este modelo se ejecuta continuamente las fases de gestión de entrada y presentación. La fase de actualización se ejecuta en la GPU y se inicializa desde la fase de entrada. Paralelamente, mientras continúa la ejecución del ciclo de las fases de gestión de entrada y presentación, la GPU realiza la fase de actualización. Tras varios pasos del ciclo, la GPU devuelve los resultados actualizando los anteriores (17).

Por último, el **modelo multihilo desacoplado con hebra GPGPU** es similar al modelo multihilo desacoplado pero añadiendo una hebra donde se ejecutará parte de la fase de actualización en la GPU. La fase de cálculo en GPU se ejecutará de manera continua e independientemente al resto de fases (18).

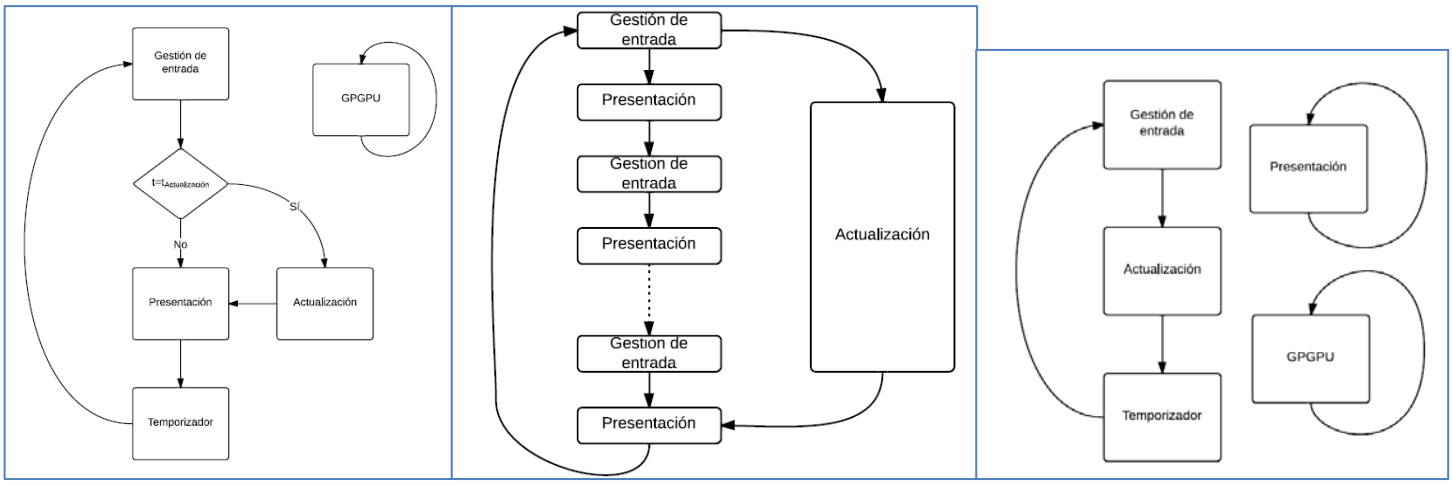


Ilustración 5 - Modelos multihilo: A – Modelo asíncrono con fase GPGPU, B – Modelo síncrono con fase GPGPU, C – Modelo multihilo desacoplado con fase GPGPU

Una alternativa al ciclo principal son los **sistemas multiagente**. En estos entornos, el sistema se desarrolla mediante la comunicación de diversas entidades independientes llamadas agentes. Cada agente tiene su propia funcionalidad y varía sus acciones según sus percepciones del entorno. El agente pese a ser autónomo se comunica con el resto de agentes del sistema y cooperan para llevar a cabo determinadas tareas (19) (20).

El caso ideal sería que los sistemas incorporasen infinita memoria y fueran capaces de procesar todos los datos de manera inmediata. Sin embargo, este ideal está bien lejos de la realidad en la que los sistemas tienen una memoria limitada y unos pocos procesadores con velocidad de procesamiento limitada. Por ello, se han buscado alternativas de manera que con las limitaciones del sistema se logre ejecutar de manera holgada el videojuego (3).

En el motor XNA se puede emplear un bucle principal continuo y acoplado o un bucle discreto y acoplado. Al desarrollador no se le muestra el bucle principal pero se indica las funciones que debe implementar. Estas funciones son las de inicialización del programa, fase de actualización y fase de presentación entre otros. Cuando se emplea un ciclo discreto y acoplado, para evitar problemas de ejecución en los casos en los que un ciclo dure un tiempo excesivo, el ciclo principal se desacopla de manera que se sacrifica la ejecución de la fase de presentación en un ciclo para emplear dicho tiempo en la ejecución de la fase de actualización del presente y siguiente ciclo. Para informar de la duración excesiva de la fase de actualización se emplea un flag o variable lógica que indica si la ejecución anterior ha sido excesiva. Esa variable no es realmente útil pues aunque informe de que en el ciclo anterior la fase de actualización duró demasiado, no se puede cuantificar de cuanto fue el exceso y por lo tanto no se puede actuar acorde al problema (21).

Otro modelo de ejecución que se emplea en muchos videojuegos es el caso contrario al motor de videojuegos XNA. En este caso, el ciclo se ejecuta de manera continua acoplada pero el programador especifica la frecuencia con la que se debe

ejecutar la fase de actualización de manera que el tiempo restante ejecute la fase de presentación mostrando por pantalla la misma imagen (22).

En (23) se documenta que actualmente en algunas simulaciones continuas se realiza la siguiente técnica: En cada ciclo del bucle se busca qué elementos están en reposo. Si un objeto está en reposo se inhabilita durante varios ciclos activándose por una colisión o un evento de interacción. Aun así, estos sistemas se sobrecargan al emplear la técnica del reposo pues deben calcular las colisiones de estos objetos y en caso de interacción con algún elemento en reposo deben generar un cálculo correcto de comportamiento físico del elemento que se encontraba en reposo.

Aunque la programación paralela está orientada a otras aplicaciones gráficas en tiempo real tales como aplicaciones de realidad virtual, en (24) se muestra que actualmente los videojuegos que emplean programación paralela han sido divididos en sectores para poder ser ejecutados en paralelos. Además, esta división del videojuego hace que sea necesario emplear estructuras de datos complejas además de complicadas funciones para comunicar las distintas divisiones entre sí. Por ello, el rendimiento de los videojuegos no muestra mejoras en cuanto a la velocidad de ejecución. Los desarrolladores de videojuegos para una única consola, con lo que no se debe realizar ningún ajuste de compatibilidad para más de un hardware, evitan esta programación por lo grandes costes tanto computacionales como económicos (25).

Algunos desarrolladores evitan emplear la ejecución por hilos que ofrecen los sistemas operativos y desarrollan sus propios métodos de ejecución y administración de hilos. En estos videojuegos, los distintos hilos ejecutan unos módulos en concreto pero aun así, cada hilo ejecuta sus propias fases de simulación y presentación. Pero los desarrolladores evitan la programación multihilo por complicaciones con la compatibilidad de las configuraciones de hardware que habitualmente tienen los computadores por lo que únicamente se emplea en videoconsolas. En (26) se expone un método para el uso de multihilo en aplicaciones de realidad virtual. Se emplea un

hilo para la ejecución del 3D mientras que el resto (se supone) están ejecutando la lógica del sistema.

2.1 Critica al estado del arte

Tal y como se ha visto, el ciclo principal del videojuego es una técnica heredada de los inicios del desarrollo de videojuegos. Al inicio de esta industria los sistemas únicamente disponían de un lento procesador y pocos recursos para ejecutar todas las instrucciones de las que estaban compuestos los videojuegos. Conforme han evolucionado los sistemas, el ciclo continuo acoplado se ha convertido en una estructura ineficaz pese a que todavía se emplee en la actualidad. Las diversas empresas de la industria del videojuego han intentado evolucionar de alguna manera como se ha observado a lo largo del apartado 2.

El ciclo continuo acoplado se ha empleado exhaustivamente pero no implica que esté exento de inconvenientes. En este capítulo se analizarán algunos aspectos en el que el ciclo continuo acoplado no presenta un buen funcionamiento. La explicación de dichos inconvenientes comenzará con un ejemplo en el que se visualiza el problema para seguidamente argumentar este inconveniente.

2.1.1 Ejecución ordenada de eventos

El primer ejemplo es un videojuego en el que se representa una pelea de disparos en el lejano oeste entre dos jugadores. Cada jugador podrá encontrarse en tres estados distintos, de pie sin realizar ninguna acción, disparar contra el otro jugador o bien esconderse tras un barril. En un instante de tiempo, ambos jugadores escogen el movimiento a realizar presionando para ello su correspondiente botón. El primer jugador ha decidido que su personaje debe disparar. El segundo jugador ha presionado el botón para que su personaje se oculte tras el barril. Contra todo pronóstico, el personaje del segundo jugador cae al suelo muerto y se acaba la partida venciendo el primer jugador.

Como se ha contemplado en capítulos anteriores, el acoplamiento implica que la fase de actualización se ve obligada a recorrer por completo el grafo de escena para comprobar y actualizar las entidades afectadas, por lo tanto se actualizan teniendo cierta prioridad unas entidades frente a otras sin que hayan sido dispuestas para que tengan ningún tipo de prioridad (27).

En el ejemplo anterior se procesó al jugador uno antes que al jugador dos de manera que cuando el primer jugador realizó el disparo el personaje del segundo jugador se encontraba de pie y desprotegido. Por ello, el personaje del primer jugador alcanza con su disparo al personaje del segundo jugador. También podría haberse procesado el segundo jugador antes del primero. Esto viene especificado, por ejemplo, por el grafo de escena (Ilustración 6 e Ilustración 7).

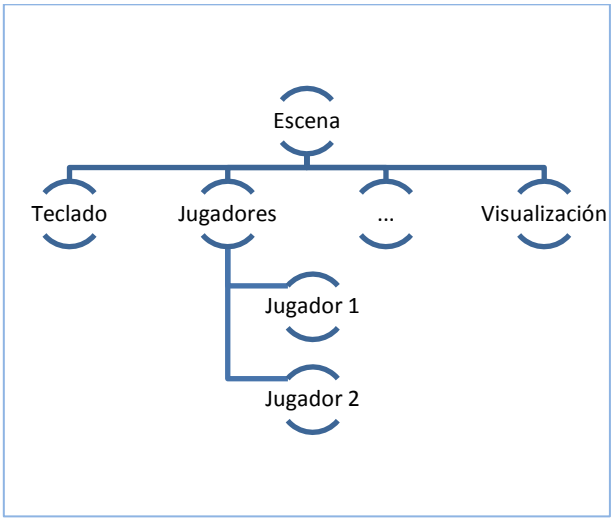
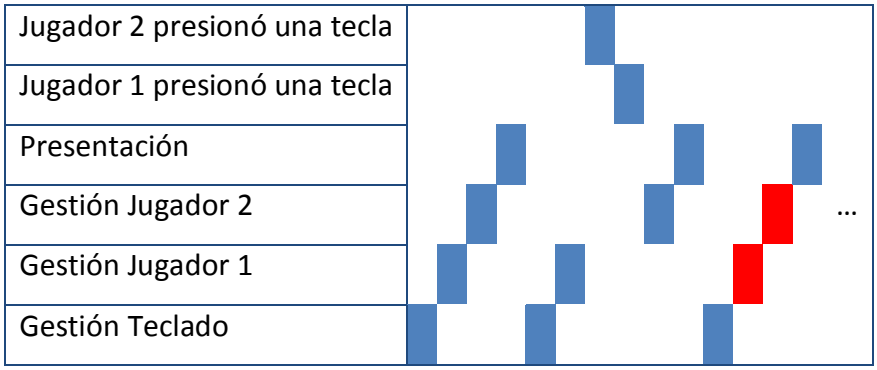


Ilustración 6 – Grafo de escena del ejemplo de ejecución ordenada de eventos.



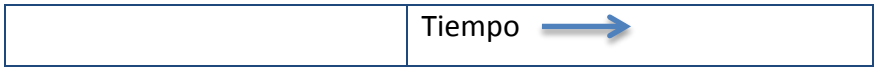


Ilustración 7 – Gráfico temporal de ejecución de las distintas secciones del videojuego. En rojo se muestra cuándo se ejecuta la respuesta a las pulsaciones.

Este inconveniente tan evidente ocurre en la inmensa mayoría de videojuegos de manera imperceptible para el jugador. Dicho problema acarrea una falsa simulación afectando al resultado final del jugador. Además si resultase evidente la falsa simulación, los jugadores se negarían a continuar jugando.

2.1.2 Ejecución de eventos previamente cancelados

En este caso se toma como referencia un videojuego en el que el jugador interviene en un asesinato. El sujeto a disparar se encuentra caminando tras una ventana. El punto de mira se encuentra inmóvil. El jugador realiza un disparo en el momento en que el enemigo se encuentra en el punto de mira. La pantalla muestra que la víctima no muere.

En el ejemplo que se muestra, el disparo debería haber acabado con el enemigo pero en realidad, al ejecutarse un evento previamente cancelado, el resultado difiere de lo que debería haber sucedido. El evento del disparo se almacena en el ordenador en cuanto el jugador presiona el botón. El sistema actualiza la posición del enemigo de manera que actualmente no se encuentra en el punto de mira. A continuación gestiona dichos eventos por lo que el enemigo, al no estar en el mismo lugar al que dispararon, no muere.

Si un evento debe cancelar otro evento y el primero se ejecuta después del segundo se produce una incongruencia. Esto sucede en el bucle continuo acoplado pues como se ha señalado anteriormente, cada evento tiene una prioridad implícita según, por ejemplo, su posición en el grafo de escena.

2.1.3 Frecuencia constante

El ejemplo de videojuego en este apartado es una carrera de bólidos. En este caso, el vehículo que maneja el jugador va a una velocidad muy alta. En el asfalto hay un socavón muy profundo que no ha visto a tiempo. El jugador no es capaz de esquivarlo pero, en el momento en el que el vehículo debería entrar en el agujero, el jugador se encuentra al otro lado del socavón como si no hubiese existido el obstáculo.

Todos los objetos del sistema ejecutan la fase de actualización con la misma frecuencia al tratarse de un sistema continuo. En el ejemplo, en un momento determinado, el vehículo se encuentra frente al socavón. La siguiente vez que se ejecuta la actualización del jugador, es tal la velocidad del vehículo que el tiempo invertido entre la ejecución actual y la ejecución anterior de la fase de actualización es lo suficientemente alto como para posicionar el vehículo al otro lado del socavón y pasar el obstáculo inadvertido para el sistema. La frecuencia de un sistema continuo viene determinada por la complejidad y cantidad de objetos que se encuentran en el sistema.

Para que un objeto se presente correctamente, este debe ejecutarse o visualizarse con una frecuencia superior a la frecuencia crítica de Nyquist. En el teorema de muestreo de Whittaker-Nyquist-Kotelnikov-Shannon se indica que al transformar algo del modelo continuo al modelo discreto este debe ejecutarse con una frecuencia superior al doble de la frecuencia que tenía en el modelo continuo. En caso de no cumplirse esta condición el sistema presentará aliasing que no es más que el efecto de deformación producido al pasar el sistema continuo a discreto.

Un ejemplo sería el efecto que producen las aspas de un ventilador cuando se iluminan con una bombilla cuya frecuencia es menor que el doble de la frecuencia a la que gira el ventilador. El observador verá que el ventilador gira en sentido contrario al que realmente está girando. Otro ejemplo es la visualización de las llantas de un vehículo en marcha. Según la velocidad del vehículo, los radios que componen la llanta

de las ruedas parecen estar girando en sentido contrario al movimiento natural de las ruedas.

En el caso de los videojuegos, este efecto puede inducir en error al usuario pues puede visualizar una animación en sentido contrario o generar duda al jugador respecto a la exactitud del sistema.

2.1.3.1 Submuestreo

La ejecución continua imposibilita la ejecución correcta de aquellos apartados que requieran una frecuencia de ejecución superior a la que el sistema ofrece. Este hecho se llama submuestreo. El submuestreo de las secciones de visualización puede producir perdida en la sensación de realidad en la animación mostrada por pantalla. En el caso de que las secciones con submuestreo estén destinadas a la fase de actualización, puede generar desplazamientos más lentos de los personajes, o imprecisión en los cálculos.

2.1.3.2 Sobremuestreo

También se puede encontrar el punto contrario, aquellas secciones del videojuego que pese a requerir una frecuencia menor que la obtenida por el bucle continuo, se ejecuta más veces de las necesarias. Esta acción se denomina sobremuestreo. Si la sección ejecutada con sobremuestreo forma parte de la fase de visualización se generan imágenes que exceden la capacidad de la pantalla por lo que se desperdician todos esos excesos. Por otro lado, si la sección que se ejecuta con submuestreo forma parte de la fase de actualización los personajes se desplazarán a mayor velocidad y al igual que en el caso anterior, los cálculos serían incorrectos.

2.1.3.3 Ajuste del software al QoS

La ejecución continua del sistema y la imposición de frecuencias no permiten asegurar una QoS global por lo que los desarrolladores nunca estarán completamente seguros de que se cumplan los requerimientos.

2.2 Modelo continuo acoplado temporizado

El modelo continuo acoplado temporizado es una expansión del sistema continuo acoplado para evitar tanto el submuestreo como el sobremuestreo. Además también permite evadir la sensación de acoplamiento aunque realmente el sistema continua acoplado.

Para lograr estas virtudes el modelo emplea un conjunto de temporizadores que se asignan a cada una de las tareas a realizar. Además, a cada una de esas tareas también se les asigna una frecuencia inicial independiente de la frecuencia del resto de tareas. Esta frecuencia puede alterarse en cualquier momento a lo largo de la ejecución del sistema. Cada temporizador almacena en que momento debe ejecutarse la tarea asignada.

En cuanto a la ejecución del bucle principal, el sistema ejecuta las tareas en cuanto comprueba que el tiempo de inicio de ejecución ha sido sobrepasado. Para realizar dicha comprobación, el sistema debe recorrer todas las tareas continuamente.

Respecto al sistema continuo acoplado, este modelo presenta la ventaja de poder determinar una frecuencia para cada sección del videojuego de manera que se evita completamente el sobremuestreo y en gran medida el submuestreo pues el sistema por su propia naturaleza no alcanzará jamás la frecuencia solicitada pero se aproximará a dicha frecuencia sin rebasarla. Otra de las ventajas es la sensación de desacoplamiento, comportamiento generado por la posibilidad de asignar frecuencias a cada sección.

Uno de los puntos negativos de este modelo es la necesidad de emplear recursos para gestionar los temporizadores y la comprobación de estos en cada ejecución del bucle principal.

2.3 MCAT con tiempo de simulación

Otra alternativa al MCAT común es el empleo de tiempo de simulación en vez del tiempo del sistema.

El modelo continuo acoplado temporizado con tiempo de simulación permite detener el tiempo para realizar ejecuciones costosas. Con esta modificación, además de las mejoras obtenidas en el MCAT, permite ejecutar cada sección del videojuego en su momento exacto. En contraste, el coste computacional es mayor que en el caso del MCAT con tiempo real pues en cada ejecución del bucle principal busca entre las distintas secciones a ejecutar cuál de todas es la siguiente. Además hay que andar actualizando el modelo de simulación continuamente para ajustarlo a las necesidades por lo que se requieren unos pocos recursos más.

2.4 Propuesta de mejora

La propuesta de mejora es lograr un modelo de ejecución que permita delimitar la frecuencia de ejecución de las distintas secciones del videojuego. Para ello, se propone un sistema continuo temporizado.

Pese a que este sistema permite proponer una frecuencia de ejecución a cada sección de videojuego, la frecuencia indicada no se cumple, sufriendo cierta varianza durante la ejecución del videojuego. Otro comportamiento negativo de este modelo es que no solo existe varianza en la frecuencia sino que el sistema no es capaz jamás de alcanzar la frecuencia determinada.

Un intento de solucionar estos inconvenientes, sería implantar el sistema continuo acoplado temporizado pero en vez de con tiempo real se emplearía tiempo de simulación. Este nuevo sistema acaba con la inmensa mayoría de los comportamientos negativos del sistema anterior pero no salva la distancia de la ejecución ordenada de eventos. Por ello, en este proyecto se empleará el modelo discreto desacoplado como se verá en próximos capítulos.

3 Propuesta de trabajo

Como se observa en el apartado anterior, el uso de un sistema continuo acoplado realiza algunos problemas que se deben paliar para el caso de un videojuego. El empleo de un sistema discreto aporta la posibilidad de que el sistema operativo empleé los recursos del computador mientras el videojuego se ejecuta. Además el uso de un sistema desacoplado permite ejecutar cada una de las fases indistintamente por lo que no es necesario ejecutar todas las fases, únicamente se ejecutarían aquellas fases que se requieran.

Por ello, las intenciones de esta tesina son:

- Desacoplar todas las fases que implementan un ciclo principal de un videojuego, seccionándolas en **subfases o módulos mínimos** integrando como módulos las distintas entidades u objetos del videojuego e incluso diferentes aspectos de los que se componen.

- La modificación del bucle principal de un videojuego mediante el empleo de un simulador discreto de tal manera que la ejecución de dicho bucle pase de ser un bucle continuo acoplado a uno discreto desacoplado. En este proyecto se evalúa dicha posibilidad de cambio realizando los menores cambios posibles.

Comprobar el rendimiento de cada una de las clases de simulación verificando mediante pruebas algunos de los aspectos más importantes de las distintas clases de simulador.

Con este nuevo modelo de ejecución, es necesario el empleo de algún mecanismo de activación para cada uno de los módulos en los que se dividirá el videojuego. Para ello, mediante el envío de un mensaje al módulo que se desee activar, despertará de su letargo tratando el contenido del mensaje y devolviendo como

resultado ninguno, uno o más mensajes destinados al mismo o a otro módulo. Para poder manejar los distintos módulos que conforman las fases de gestión de entrada, actualización y presentación de manera coherente se empleará RT-DESK. RT-DESK es un simulador discreto de tiempo real basado en mensajes (28). Un mensaje en RT-DESK no es más que una solicitud destinada a un módulo en un tiempo determinado. Cuando un mensaje ha sido enviado por un módulo pero el tiempo de entrega no ha llegado todavía, el mensaje se almacena en una pila interna del RT-DESK. Todos los mensajes son enviados con la misma prioridad y por tanto el orden de ejecución de los módulos se basa únicamente en el tiempo de entrega asignado a cada uno de los mensajes. De esta manera, los módulos son ejecutados únicamente cuando son necesarios evitando así una pérdida de rendimiento.

Con este sistema, no existe frecuencia de muestreo fija ni común a todos los módulos. Cada uno tiene su propia frecuencia de muestreo y esta no tiene porqué ser siempre la misma sino que puede variar dinámicamente según las necesidades. En esta situación, las entidades que tengan un comportamiento rápido tendrán mayor frecuencia que las entidades que tengan un comportamiento más lento. Por lo tanto, las entidades se ejecutan cuando es necesario evitando derroches de computación. Si la potencia de cálculo fuera insuficiente el sistema se ralentiza pero la frecuencia de muestreo se mantiene por lo que la ejecución sigue realizándose de manera correcta. En caso que el sistema tuviese mucha potencia de cálculo, esta potencia no se desaprovecha con cálculos innecesarios como sucedía en el bucle continuo. Cuando no se esté empleando toda la potencia de cálculo se entrega el manejo del procesador al sistema operativo quedando el videojuego a la espera del envío del siguiente mensaje. Independientemente de la potencia de cálculo, las entidades simplemente se ejecutan las veces necesarias. Ni se ejecutan más veces de las necesarias (sobremuestreo) ni se ejecutan menos veces de las que necesite (submuestreo).

Se ha elegido la librería RT-DESK para acoplarla al videojuego y así proporcionar una gestión del paso de mensajes entre módulos. Las principales razones por las que se ha realizado esta selección son las siguientes:

- Gestiona el tratamiento necesario de los mensajes para el correcto funcionamiento del sistema.

- Proporciona un servicio de eficiencia en llamadas al gestor de memoria de manera que si un mensaje dejó de ser empleado, se deja en espera en vez de eliminarlo. Para la creación de un nuevo mensaje, se comprueba si hay algún mensaje inactivo para emplearlo en lugar de generar uno nuevo.

- Este sistema ya ha sido empleado en aplicaciones gráficas en tiempo real de manera exitosa.

- El framework RT-DESK no necesita realizar grandes cambios de código para poder integrarlo.

En la sección 3.1.2 se detallará más a fondo el simulador RT-DESK.

El videojuego al que se le acoplará el simulador discreto será una versión desarrollada en la Universidad Politécnica de Valencia del conocido videojuego Space Invaders en homenaje a la empresa Atari, creadora del videojuego declarada en bancarrota en enero de 2013 (29). Esta versión, emplea tanto entornos 2D como 3D de manera que emplea las capacidades del computador. Otra de las razones por las que se ha decidido emplear este videojuego es porqué se dispone completamente del código fuente y además se conoce todo el funcionamiento interno gracias a la documentación adjunta al código fuente.

Se comprobará la ejecución del videojuego tanto en su versión continua acoplada como en su versión discontinua desacoplada. Además, para que ambas ejecuciones sean realizadas de igual manera, los eventos generados por el jugador quedarán registrados de manera que puedan volver a ser reproducidas tantas veces se deseen con cualquier versión de ejecución.

Para fortalecer las diferencias entre ambas clases de ejecución del bucle principal, se emplearán también sistemas continuos acoplados con temporizadores. Esta nueva clase de ejecución se presenta como el punto intermedio entre las dos clases definidas anteriormente de manera que se intenta aproximar la ejecución continua acoplada a un punto lo más cercano posible al modo discreto desacoplado.

3.1 Análisis DAFO

En este apartado se realiza un análisis DAFO para las ejecuciones continua acoplada, continua acoplado con temporizadores y otro para la ejecución discreta desacoplada.

Los análisis DAFO muestran todas aquellas características y como afectan al objeto analizado. Las características se pueden definir como positivas o negativas y externas o internas. Estas cuatro combinaciones forman lo que se denomina **oportunidades** (Factores positivos generados en el exterior del sistema), **amenazas** (Factores negativos generados en el exterior del sistema), **fortalezas** (Factores positivos generados en el interior del sistema) y **debilidades** (Factores negativos generados en el interior del sistema).

3.1.1 Modelo continuo acoplado

El modelo continuo acoplado (MCA) presenta gran cantidad de factores que se sitúan entre los apartados del análisis DAFO. Por ello, se han seleccionado los más evidentes y fáciles de observar.

Entre las **oportunidades** que ofrece un videojuego ejecutado de manera continua acoplada como base del bucle principal, se encuentra que es la clase de ejecución habitual en este tipo de software. Todos los programadores de la industria conocen y emplean este modelo. Todos los motores de videojuegos emplean este modelo o alguna de sus variantes. Además, el modelo es soportado por todas las plataformas y lenguajes empleados en la codificación de videojuegos. El simple hecho

de no requerir ningún estudio sobre clases de ejecución implica que en muchos videojuegos se emplee este modelo por comodidad. Además, no requiere ningún concepto nuevo ni visualizar el bucle principal desde una perspectiva distinta a la que habitualmente se tiene. Otro factor que se debe remarcar se basa en que cuando el sistema se satura todos los eventos de entrada del usuario se procesan, aunque se procesen a largo plazo.

Por otra parte, entre las **amenazas** se encuentra el bajo rendimiento del videojuego cuando se satura obteniendo una salida de imágenes más pausada y perdiendo la sensación de realidad por parte del jugador. Otra de las amenazas a señalar es la lentitud de las aplicaciones en segundo plano por el escaso empleo de recursos dedicados para el sistema operativo.

En cuanto a los factores internos, la ejecución continua acoplada muestra como principal **fortaleza** la fácil implementación pues simplemente se divide el bucle principal en tres fases y se ejecutan cíclicamente. Otra de las fortalezas que aporta es la comprensión de funcionamiento por parte del programador ya que no varía el modo de ejecución al que las grandes desarrolladoras están acostumbradas. Esta es una de las principales razones por la que la industria del videojuego mantiene la ejecución continua acoplada. Una fortaleza más a tener en cuenta es el escaso gasto computacional requerido para poder realizar una ejecución continua acoplada del sistema ya que únicamente requiere la ejecución de una instrucción que maneje el flujo del programa de manera que las instrucciones mencionadas anteriormente se ejecuten continuamente.

Por último, las **debilidades** que se obtienen del empleo de esta clase de ejecución es el exceso de imágenes por segundo que presenta y también el efecto contrario, la generación de pocas imágenes. Esta debilidad hace que el usuario pierda la sensación de realidad e incluso puede producir malestar en el jugador. Otra de las debilidades a remarcar es la frecuencia impuesta a todas las fases del bucle principal. Cada sección del videojuego debe ejecutarse con la misma frecuencia que el resto de

secciones sin poder determinar siquiera la única frecuencia a la que se ejecuta todo el sistema. Además, se encuentra la debilidad de que el orden de ejecución de las distintas secciones del videojuego viene impuesto por el programador, normalmente por el diseño del grafo de escena, de manera que no se pueden ejecutar secciones alternamente ni ninguna otra combinación a excepción de la ejecución completa manteniendo el orden. La última debilidad que se debe señalar es que el exhaustivo empleo del procesador y otros recursos no permiten al sistema operativo realizar operaciones de mantenimiento u otras funciones internas del mismo.

A modo de resumen, en la Ilustración 8, se puede encontrar un esquema DAFO que ayudará a visualizar lo expuesto anteriormente.

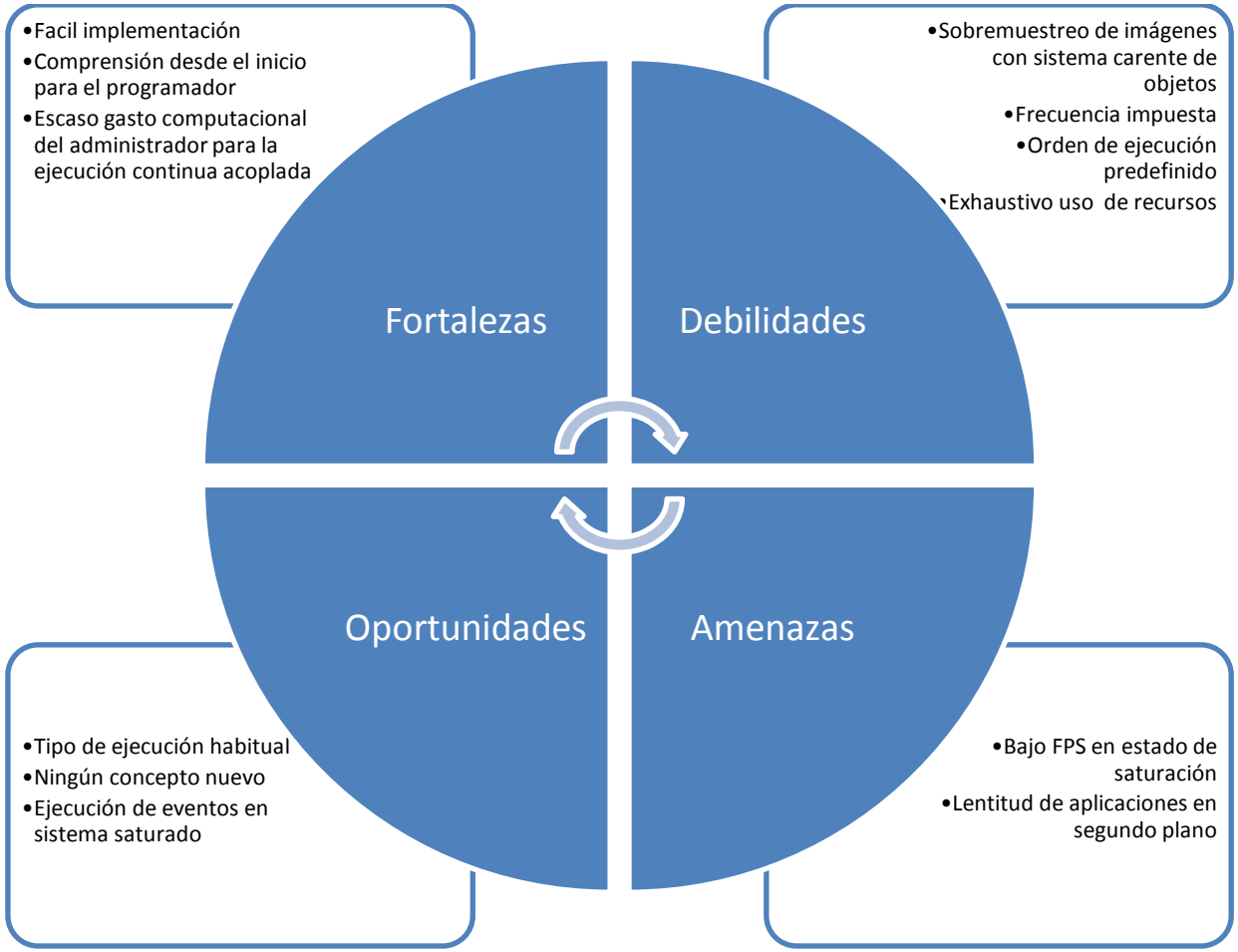


Ilustración 8 – Análisis DAFO de la ejecución continua acoplada

3.1.2 Modelo continuo acoplado temporizado

En este apartado, se muestra el análisis DAFO de la ejecución continua acoplada temporizada (ECAT). Se basa en el empleo de temporizadores para determinar y limitar de algún modo la ejecución de las distintas secciones del bucle principal. Este modelo es aceptable para sistemas con pocas secciones a temporizar, pero en caso de que aumente la cantidad de secciones el sistema se vuelve ineficiente.

Podemos distinguir dos clases, la ejecución continua acoplada temporizada mediante tiempo real y la ejecución continua acoplada temporizada mediante tiempo de simulación. Como las diferencias entre ambas son pocas, se realiza el análisis en conjunto y se especifican aquellos factores que únicamente afectan a una de las dos clases.

Las **fortalezas** que muestra esta ejecución mejoran las vistas anteriormente para la ejecución continua acoplada. Entre ellas podemos encontrar la posibilidad de emplear una frecuencia para cada sección del videojuego de manera que cada sección tenga su propio periodo de ejecución. Además, otra de las fortalezas es que la frecuencia de ejecución se puede alterar a lo largo de la reproducción del videojuego. Esto ofrece la sensación de desacoplamiento aunque es necesario indicar que este desacoplamiento es virtual y no real. Esto se debe a que desaparece el orden de ejecución entre secciones debido a la independencia de frecuencias de ejecución. Otra de las cualidades positivas de este modo de ejecución lo podemos encontrar en la aparición de tiempo ocioso. Pese a que este tiempo en el que el sistema no está realizando operaciones sea corto, aporta un beneficio al sistema operativo sobre el cual se está ejecutando. Por último cabe reseñar la fácil implementación (aunque algo más costosa que en el caso de la ejecución continua acoplada) ya que solo es necesario que cada sección almacene cuando debe volver a ejecutarse y cuando se alcance este punto en el bucle se compruebe si el tiempo actual supera al tiempo en el que se debía ejecutar. Esta fortaleza se presenta únicamente en la ejecución continua acoplada temporizada mediante tiempo real pues la implementación del bucle de manera

continua acoplada temporizada mediante tiempo de simulación es más compleja. Por ello, en las debilidades se volverá a hacer mención. Una fortaleza que presenta la ejecución temporizada mediante tiempo de simulación frente a la ejecución que emplea tiempo real es que la frecuencia de las distintas secciones del videojuego se cumple de manera correcta, es decir, ejecutándose únicamente cuando ha transcurrido exactamente el periodo especificado. Como se puede comprobar de lo expuesto anteriormente se puede deducir la sencillez de conceptos, aportando mayor fuerza a la última fortaleza.

Por contraposición, se encuentran las **debilidades**. Tal y como se comentaba anteriormente, una de ellas es que la desviación típica de las frecuencias de un mismo objeto obtenidas en una ejecución en la que se emplea el tiempo real no es nula. Esto quiere decir que no se puede asegurar el tiempo real que transcurre desde la ejecución de una sección hasta la siguiente ejecución de la misma sección. Además, la frecuencia real siempre es menor que la solicitada pues en el momento en el que se debería ejecutar la sección el programa se encontrará con toda seguridad ejecutando otra de las secciones. Por esta razón, un programador o una desarrolladora no puede asegurar que se mantenga unos FPS (*Frames Per Second*, imágenes por segundo) determinados sino que este fluctuará por debajo de la frecuencia especificada. Otra de las debilidades a reseñar es el consumo de recursos destinados a la gestión del método de ejecución. Además de las variables que deben almacenar cada sección, en cada ejecución hay que realizar la operación de comprobar si esa sección debe ejecutarse o si aún debe esperar. Una debilidad más es que no se puede asegurar que distintos objetos con la misma frecuencia se ejecuten la misma cantidad de veces debido a la fluctuación de frecuencias que anteriormente se ha justificado. Esta debilidad junto a la debilidad de que la ejecución no es completamente exacta, dos ejecuciones bajo las mismas condiciones no tienen por qué devolver los mismos resultados.

Observando las **oportunidades** del sistema se observa que los sistemas basados en este modelo muestran mayor realismo que aquellos que emplean el modelo continuo acoplado. Otra oportunidad importante es que la generación de FPS se

muestra más acorde a lo requerido por los monitores de manera que no se sobrecarga la plataforma en la que se esté ejecutando el sistema.

Entre las **amenazas** presentes indicar que pese a las mejoras respecto a la eficiencia y rendimiento del sistema continuo acoplado, si se alcanza el punto de saturación, acaba teniendo los mismos problemas que el anterior. Por esto, implica mejora respecto al modelo anterior pero existen situaciones en los que se comporta de manera similar.

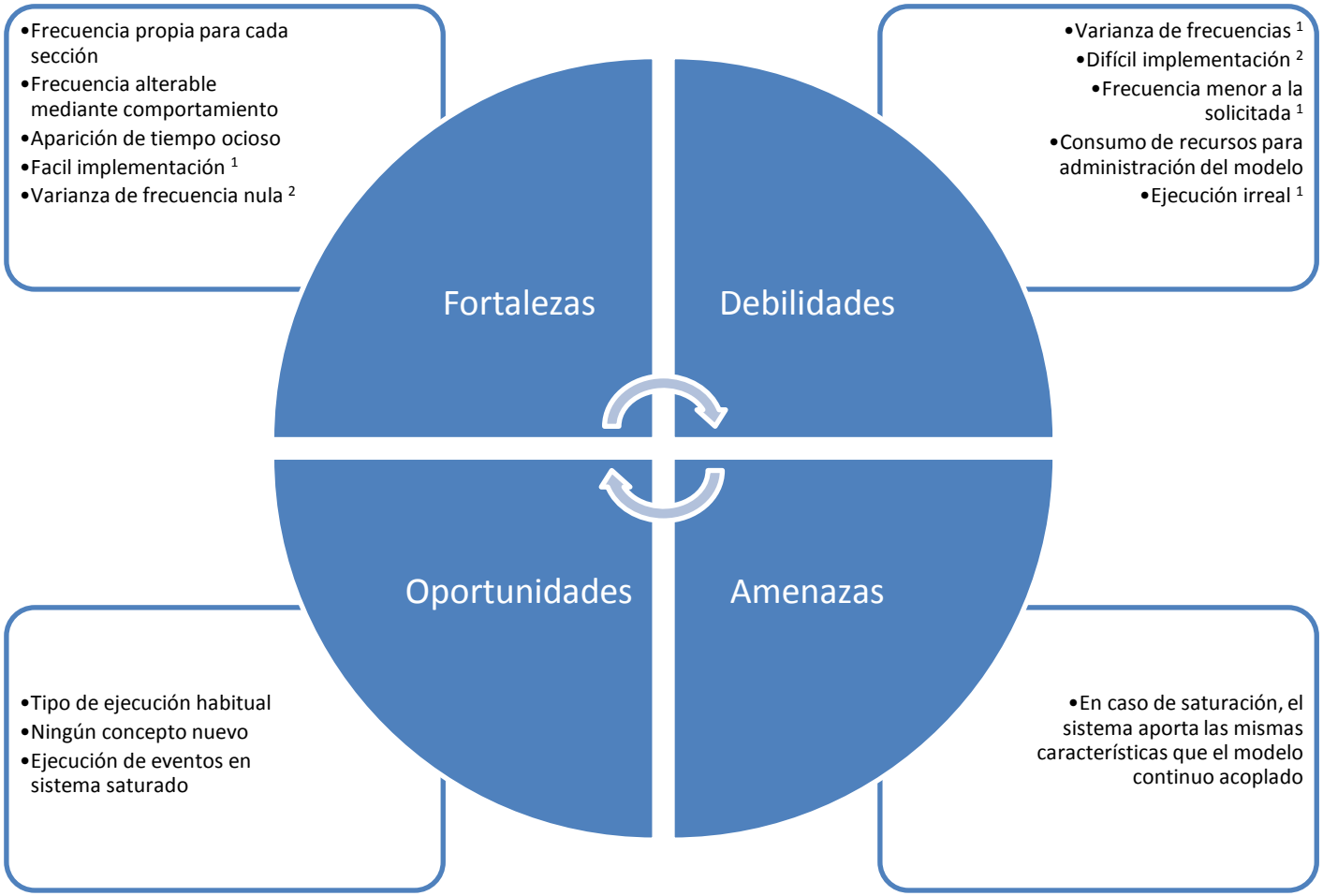


Ilustración 9 – Análisis DAFO de la ejecución continua acoplada temporizada. El subíndice 1 hace referencia a los casos de tiempo real mientras el subíndice 2 indica tiempo de simulación

3.1.3 Modelo discreto desacoplado

En este capítulo se realiza un análisis DAFO del modelo discreto desacoplado (MDD). A continuación se detallan las oportunidades, amenazas, fortalezas y debilidades.

Una de las **oportunidades** que presenta la ejecución discreta desacoplada es la posibilidad de disminuir la velocidad del procesador y detener el disco duro obteniendo un ahorro energético. Además, las QoS se pueden mantener cuando el videojuego se cambia de plataforma en la que se ejecuta pues al existir tiempo ocioso se emplearía ese tiempo para realizar las fases de gestión de entrada, actualización o visualización si fuesen más costosas temporalmente. Otra de las oportunidades que se obtienen es la posibilidad de incrementar las QoS sin necesidad de cambios de hardware en la plataforma. Una oportunidad más que se debe reseñar es que todas las herramientas de desarrollo permiten el empleo de este modelo.

En cuanto a las **amenazas** se encuentra el cambio de tipo de ejecución respecto al tipo de ejecución habitual, lo que supone un rechazo por las empresas desarrolladoras. Estas empresas no desean realizar estudios sobre nuevos métodos de ejecución. Otra de las amenazas que cabe remarcar es que el rechazo de las empresas genera sensación de poca seguridad en esta clase de ejecución.

La ejecución discreta desacoplada contiene un gran número de factores que se pueden encasillar en las **fortalezas**. La fácil adaptación de las librerías de simulación discreta desacoplada en un videojuego es una importante fortaleza. A parte de esta, otra fortaleza importante es que no se desaprovechan recursos en cálculos innecesarios. Esto lleva a hablar de dos fortalezas más que refuerzan la fortaleza anterior. La primera de ellas es que cada sección del videojuego puede tener una frecuencia de ejecución variante e independiente del resto de secciones. La segunda fortaleza es el orden variante de ejecución de las secciones de manera que el orden de ejecución de las secciones no tiene por qué estar fijado. Por último, se presenta la

fortaleza respecto al tiempo ocioso que permite la ejecución de operaciones de mantenimiento del sistema operativo y otras aplicaciones.

En último lugar, se realiza un repaso de las principales **debilidades** de esta clase de ejecución. Los conceptos necesarios para la implantación del sistema son completamente nuevos. Los conceptos no son complejos pero son novedosos y por ello, puede ser complejo adaptarse a ellos. Otra de las debilidades es la necesidad de procesamiento y uso de recursos para la gestión del modo de ejecución.

La Ilustración 10 se presenta a modo de resumen del análisis realizado a la ejecución discreta desacoplada.

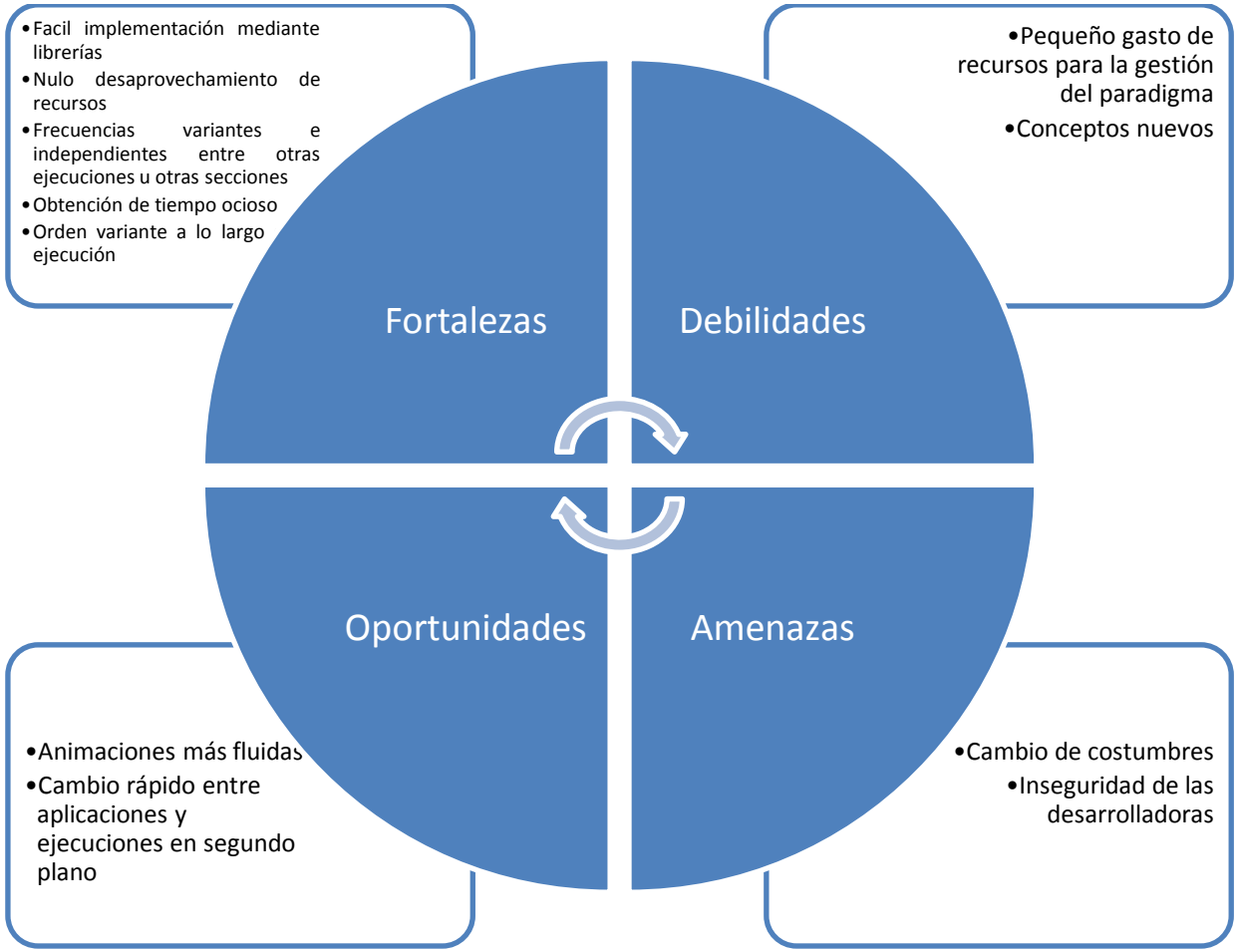


Ilustración 10 – Análisis DAFO de la ejecución discreta desacoplada

4 RT-DESK

RT-DESK es el acrónimo de Real Time Discrete Event Simulation Kernel. Es un núcleo de simulación discreta dirigida a aplicaciones de tiempo real. En este sistema las distintas entidades se comunican entre sí mediante mensajes. Una entidad reacciona a un mensaje recibido generando cambios en el sistema y generar ningún, uno o varios mensajes. Cada mensaje tiene un tiempo de entrega determinado. En el caso de no haber alcanzado el tiempo de entrega, este queda en espera hasta que dicho tiempo es alcanzado. Las entidades además de comunicarse con cualquier entidad del sistema también se pueden comunicar consigo mismas. Además, los mensajes enviados pueden ser recibidos instantáneamente (28).

Inicialmente el sistema comenzó como DESK. Este núcleo de simulación discreto funcionaba con dos entidades básicas: los clientes y las estaciones de servicio. Los clientes circulan de estación de servicio en estación de servicio realizando cambios en el sistema. Las estaciones de servicio pueden ser fuentes, servidores y recursos con lo que el simulador se completa con todas las clases de entidades necesarias para ejecutarse correctamente y ofrecer todos los servicios que se pretendía (30).

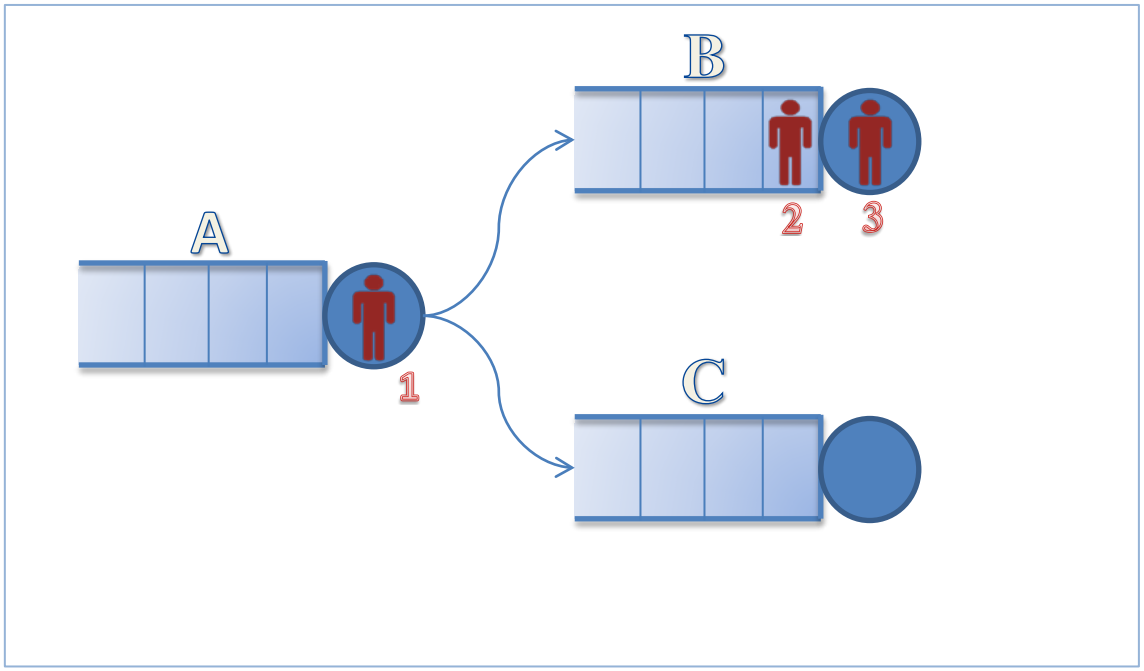


Ilustración 11 – Representación de un sistema DESK

En la Ilustración 11 se representa una ejecución ejemplar del sistema DESK. Los objetos señalados con las letras 'A', 'B' y 'C' son estaciones de servicio. La silueta roja representa un único cliente y la numeración que le acompaña indica la posición del cliente en varios momentos temporales. El cliente inicialmente se encuentra en la estación de servicio 'A' (Posición 1). Tras obtener los servicios ofrecidos por la estación 'A', el cliente puede decidir entre dirigirse a la estación de servicio 'B' o 'C'. El cliente decide automáticamente dirigirse a la estación 'B'. Esta decisión la toma comprobando el estado interno del cliente y los servicios que ofrecen las estaciones 'B' y 'C'. Una vez escogida la siguiente estación de servicio, el cliente pasa de la estación de servicio 'A' (Posición 1) a la cola de espera de la estación de servicio 'B' (Posición 2). Una vez introducido en la cola de espera, estará detenido hasta que el cliente que se encuentre dentro de la estación de servicio y todos los clientes que se encuentren por delante en la cola hayan sido servidos, en caso de haber algún cliente más en la cola o en la estación de servicio. A continuación, el cliente se traslada de la cola de espera (Posición 3) a la estación de servicio (Posición 3). El cliente continuará accediendo de estación en estación hasta que finalice su tarea.

A continuación se generó JDESK que no es más que una extensión de DESK para aplicaciones web. El sistema está desarrollado en Java y permite el empleo de módulos generados en Java para ser utilizados por el simulador. Por lo tanto, el usuario no necesita aprender un nuevo lenguaje de programación pudiendo emplear el lenguaje JAVA (31).

Por último, el sistema avanzó hasta lo que hoy se conoce como RT-DESK (anteriormente conocido como GDESK). Las antiguas estaciones de servicio son las actuales entidades y los clientes se han transformado en mensajes (6).

En caso de que el sistema necesite más mensajes tanto el dispatcher como el gestor de memoria aumentan su capacidad para poder manejar un mayor número de mensajes. Tanto el gestor de memoria como el gestor de mensajes tienen siempre el mismo tamaño y se corresponde a la cantidad total de mensajes que existen en el sistema.

4.1 Aportación de la librería

Los distintos servicios que ofrece RT-DESK son:

- Proporciona a las aplicaciones gráficas en tiempo real las estructuras de datos y funciones necesarias para modelar el mecanismo de paso de mensajes.

- Gestionar la comunicación de las entidades mediante paso de mensajes:

    - Gestionar el proceso de envío de mensajes.
    - Mantener los mensajes enviados y todavía no recibidos por la entidad destino ordenados por tiempo de ocurrencia.
    - Enviar los mensajes a las entidades destino en el instante de tiempo indicado.

- Garantizar que los mensajes son enviados a las entidades receptoras en el tiempo indicado y ordenados por tiempo.
- Reciclaje de mensajes para realizar menos llamadas al gestor de memoria.

RT-DESK únicamente gestiona la comunicación mediante mensajes entre entidades. No realiza ninguna tarea que los mensajes lleven asociada. La entidad receptora será la encargada de ejecutar la función que la propia entidad tenga asociada al contenido del mensaje. La entidad tras procesar el mensaje teniendo en cuenta de que entidad proviene y el contenido del mensaje, se encargará de ejecutar todas las funciones necesarias para generar una respuesta al mensaje. Todas las entidades y las susodichas funciones pertenecen a la aplicación donde se integre RT-DESK y no del propio RT-DESK (5).

RT-DESK se emplea para gestionar la comunicación entre entidades en un videojuego pero también puede ser empleado en cualquier aplicación que requiera una comunicación temporizada entre varias entidades sin que tenga que estar destinada a los videojuegos o a aplicaciones visuales e interactivas.

Los objetivos aportados por RT-DESK son:

- Gestionar los mensajes de manera discreta.

- Ser autocontenido de manera que sea fácilmente integrable en otras aplicaciones delegando parte de su funcionalidad a dichas aplicaciones.

- Permitir una gestión de mensajes transparente para el usuario de manera que el desarrollador del videojuego interfiera lo mínimo posible en el funcionamiento del núcleo.

Con RT-DESK las aplicaciones pasan de ser un sistema continuo a ser uno discreto. A continuación se propone un ejemplo para mostrar la diferencia. Se desea que en el instante 5 segundos detone unos explosivos en nuestro videojuego. En un sistema continuo se comprobaría ininterrumpidamente cuanto tiempo falta para llegar al instante. Una vez pasados los 5 segundos detonará los explosivos. Sin embargo, el sistema discreto de RT-DESK se activaría automáticamente a los 5 segundos haciendo detonar los explosivos sin necesidad de hacer comprobaciones continuamente.

RT-DESK se presenta como una API en forma de una colección de cabeceras que presentan las funcionalidades a las que pueden acceder. Además, se agrega un DLL que suministra el código de dichas cabeceras por lo que los desarrolladores de videojuegos no pueden acceder al código interno de RT-DESK administrando los mensajes de manera transparente como se comentó anteriormente. Este simulador discreto está implementado en C++ por lo que es compatible con todos los tipos de programación y plataformas que acepten código escrito en C++ o C#.

4.2 Diseño estático

En este apartado se muestran las distintas secciones que conforman la librería RT-DESK. El entendimiento de estas secciones aporta una ayuda para la comprensión de su funcionamiento.

4.2.1 Núcleo

El núcleo es el punto de acceso al sistema y el gestor de todos los recursos que forman la librería. Es el responsable de iniciar, pausar, detener o reanudar el simulador discreto. Además contiene un enlace al despachador de mensajes para poder gestionar de manera correcta el simulador.

4.2.2 Despachador de mensajes

El **gestor de mensajes o "Dispatcher"** es la principal estructura de datos del simulador y su función es controlar el envío y la recepción de mensajes. Todos los

mensajes enviados por las entidades los gestiona el dispatcher. Realmente, cuando una entidad envía un mensaje a otra entidad, el mensaje es recibido por el dispatcher quien más tarde enviará el mensaje a la entidad destino. El dispatcher es transparente para las entidades. Las entidades reciben los mensajes como si realmente hubiese sido enviado por la entidad origen. El dispatcher es el encargado de guardar los mensajes hasta que el tiempo de ocurrencia haya pasado y entonces enviar el mensaje correspondiente a la entidad destino. Para ello, el dispatcher convierte el tiempo de ocurrencia a tiempo global del simulador para poder ordenar los mensajes que almacena de manera que el mensaje cuyo tiempo de entrega sea más próximo al tiempo actual quede primero a la espera de ser enviado. Una vez alcanzado el tiempo de entrega, el dispatcher envía el mensaje a la entidad destino ejecutando la función de recepción de dicha entidad. La función de recepción es implementada por el programador de modo que el programador es quien escoge el comportamiento de la entidad como respuesta al mensaje. El dispatcher está compuesto por el reloj de simulación y una estructura de almacenamiento de mensajes. El reloj de simulación indica el tiempo actual del sistema. Este se actualiza cada vez que el dispatcher envía un mensaje por lo que en realidad muestra el tiempo de la última entrega. La estructura de almacenamiento de mensajes no es más que un heap y contiene los mensajes que las entidades han enviado y todavía no ha vencido su tiempo de entrega.

4.2.2.1 Mensaje

Los **mensajes** son elementos que permiten las comunicaciones entre entidades. El mecanismo de paso de mensajes modela el comportamiento de cada entidad y del sistema en general. Los atributos que contienen los mensajes se pueden clasificar en dos grupos. El primero de ellos son parámetros de gestión de RT-DESK. Entre estos parámetros se incluyen las referencias a las entidades de origen y destino del mensaje, el tiempo que debe transcurrir para hacer la entrega del mensaje y un enlace del mensaje al pool que más adelante se describirá. Las entidades de origen y destino pueden ser la misma entidad. En cuanto al tiempo de ocurrencia, puede convertirse en una entrega inmediata si se determina un tiempo 0. Para modificar estos parámetros

el programador debe emplear unas funciones destinadas a ello de manera que no se tiene acceso directo a dichos parámetros para el correcto funcionamiento de RT-DESK. El segundo de los grupos es el de parámetros destinados a la aplicación. Este grupo lo conforma una serie de atributos añadidos por el programador a cada mensaje para aportar toda la información necesaria desde la entidad emisora hasta la entidad destino para que esta última realice las tareas pertinentes además de la clase de mensaje. El comportamiento del receptor respecto a un mensaje viene determinado por la entidad de origen, el tipo de mensaje, los parámetros del grupo destinado al correcto funcionamiento de RT-DESK y el estado actual de la entidad.

4.2.2.2 Cola de mensajes

La cola de mensajes almacena todos aquellos mensajes enviados por las distintas entidades y que están esperando a ser despachados. Están ordenados temporalmente para localizar cual es el primer mensaje que se debe despachar y en que instante además de permitir un acceso directo. Cada vez que un mensaje es enviado por una entidad, se introduce en la cola de mensajes de manera ordenada por el momento temporal en el que se debe realizar la entrega.

4.2.2.3 Administrador de pool de mensajes y pool de mensajes

El pool de mensajes es un almacén de mensajes en estado ocioso. Existe un pool por cada tipo de mensaje. El pool no tiene ningún tipo de orden. Su objetivo es reducir las llamadas al gestor de memoria. Si se necesita un mensaje y no queda ninguno en el pool, se crea un conjunto de mensajes en vez de un único mensaje para disminuir la cantidad de llamadas al gestor de memoria.

El administrador se encarga de organizar todos los pools del sistema. Cuando al sistema se le indica la cantidad de tipos distintos de mensajes que se van a emplear, el administrador genera un nuevo pool para cada uno de ellos. Cuando se necesita un mensaje para ser enviado por una entidad el administrador es el encargado de acceder al pool correspondiente y solicitar uno de los mensajes ociosos que allí se encuentra.

Toda la gestión temporal de RT-DESK viene determinada mediante el número de ticks en vez de segundos de manera que los envíos de mensajes se ejecutan en un momento temporal más específico.

4.2.3 Entidad RT-DESK

Las **entidades** son objetos creados por parte del desarrollador que heredan de una clase de RT-DESK para que incorporen las funcionalidades de enviar y recibir mensajes. Estas entidades forman cada uno de los objetos del videojuego tanto objetos físicos (personajes, disparos, escudos,..) como objetos lógicos (contadores, monitores, IA,…). También se incluye como entidad aquellos objetos lógicos dirigidos a la generación de imágenes, reproducción de sonidos y demás.

4.3 Funcionamiento dinámico

En este apartado se esclarece el funcionamiento y el flujo de datos para comunicarse con RT-DESK.

4.3.1 Inicialización

Antes de comenzar con la ejecución del sistema, se debe inicializar la librería. Primero hay que crear una instancia del motor y ejecutar su arranque. A continuación se debe indicar al administrador de pools de mensajes los distintos tipos de mensaje que se van a emplear en las comunicaciones entre entidades. Después se debe enlazar cada entidad con el despachador de mensajes. Una vez finalizado el enlazado de las entidades con el despachador de mensajes se generan los mensajes iniciales necesarios para ejecutar o despertar las entidades que deban ejecutarse inicialmente.

4.3.2 Ejecución del simulador

La ejecución del simulador se realiza mediante una llamada al motor de la librería. El despachador de mensajes se ocupa de enviar los mensajes que están a la espera y cuyo tiempo de entrega ya ha vencido. Cuando el despachador de mensajes

no tiene más mensajes que entregar por el momento, devuelve los mandos al sistema indicándole cuanto tiempo debe pasar hasta la siguiente entrega de mensajes.

4.3.3 Envío de mensajes

Una entidad puede enviar un mensaje llamando a la función destinada a ello implementada en la clase Entidad de la que hereda cada una de las entidades. A esta función hay que pasarle tres parámetros. El primero de ellos es el mensaje a enviar. El segundo es la entidad destino del mensaje. Por último, el tercer parámetro es el tiempo que debe pasar hasta la entrega del mensaje.

4.3.4 Recepción de mensajes

Las entidades también tienen una función heredada de la clase Entidad que se encarga de la recepción y procesamiento de mensajes. El programador debe codificar esta función. Esta función recibe únicamente el mensaje. El remitente se obtiene gracias a uno de los atributos del mensaje. Según el tipo de mensaje, el programador habrá decidido el comportamiento que debe tener la entidad y la respuesta al mensaje. La recepción de un mensaje puede modificar el estado interno de una entidad y puede realizar uno o más envíos de mensajes a otras entidades o a sí misma.

5 Banco de pruebas

Para generar un banco de pruebas era necesario localizar un sistema en el que se pudiesen realizar los cambios necesarios para poder ejecutar las pruebas y nos mostrará la realidad de un videojuego.

Space Invaders es el videojuego escogido para realizar este proyecto ya que se dispone de acceso al código fuente por completo. Es uno de los videojuegos más conocidos del mundo y también es uno de los precursores en el mundo de los videojuegos. Tanto infantes, adultos e incluso ancianos disfrutan de este sencillo videojuego. Se clasifica como shoot'em up.

El jugador maneja una nave situada en la parte inferior de la pantalla. Esta nave siempre apunta hacia arriba. En la parte superior se encuentra una horda de alienígenas organizados que desean conquistar la tierra. El jugador deberá acabar con todos los alienígenas. Las naves tanto alienígenas como la del jugador tienen la capacidad de disparar. El usuario está protegido por cuatro bunkers. Los bunkers se desintegran al recibir disparos del jugador y del enemigo. Las naves enemigas se desplazan horizontalmente y bajan su posición cuando alcanzan uno de los laterales de la pantalla. Después de descender, vuelven a desplazarse horizontalmente pero esta vez en sentido contrario. Las naves continuaran realizando este movimiento hasta que alcancen al jugador en cuyo caso se dará como finalizada la partida. Si el jugador es capaz de acabar con todas las naves alienígenas se proclamará vencedor.

Ilustración 12 – Captura de pantalla del juego original

5.1 Historia

El videojuego se creó en 1978 para la empresa Taito Corporation. Su creador, Toshihiro Nishikado era un diseñador japonés de la empresa. El videojuego se llamó Space Invaders (スペースインベーダー). Al comienzo, los enemigos eran soldados humanos pero como no era bien visto se cambiaron por alienígenas. El videojuego se comercializó y la productora Midway compró los derechos para su reproducción en máquinas recreativas creadas en USA. Además, Atari compró también una licencia y distribuyó el videojuego para la consola Atari 2600. Más adelante, el videojuego fue portado por Nintendo para una consola del momento, NES (Nintendo Entertainment System). A lo largo de los años, el videojuego ha sido remasterizado y clonado de diversas maneras para plataformas tan diversas como PS2 (Space Invaders Anniversary), Nintendo DS (Space Invaders Revolution y dos ediciones de Space Invaders Extreme) o móviles.

Este videojuego ha supuesto un gran punto de partida para muchos artistas. Shigeru Miyamoto, creador de la franquicia Mario, valora el Space Invaders como el videojuego que revolucionó la industria. Otras empresas han tomado el relevo y han creado videojuegos similares como Galaga o Asteroids. También es fuente inspiración de músicos, científicos e incluso de artistas gráficos y cineastas.

5.2 Diseño previo

En este apartado se expondrá como era el sistema previamente a los cambios necesarios para generar un banco de pruebas que aportase la plataforma en la que realizar las pruebas correspondientes.

El videojuego Space Invaders empleado en este proyecto es utilizado como práctica de la asignatura Animación por Computador y Videojuegos. En estas prácticas los alumnos realizan modificaciones en el código para agregar nuevas funcionalidades al sistema. Se decidió emplear este videojuego porque, como se ha indicado anteriormente, se tiene pleno acceso al código sin ningún tipo de restricción.

El código con el que se inició este proyecto necesitaba una exhaustiva depuración ya que los alumnos no se aseguraban de un correcto funcionamiento del videojuego. Además, el código se encontraba en proceso de refactorización por lo que también se invirtió tiempo en finalizar la refactorización para obtener un código ejemplar de un videojuego.

Con la finalidad de comprobar la complejidad y el entendimiento de las distintas partes del código inicial para más tarde realizar las adaptaciones pertinentes para incorporar los distintos modelos, se introdujo una nueva nave (Circleship).

5.2.1 Diseño estático

Para realizar un análisis del videojuego es necesario explicar primeramente las estructuras de datos que componen el sistema y más tarde, la interacción entre dichas estructuras de datos.

5.2.1.1 Núcleo

El **núcleo** del videojuego contiene todos los objetos que el videojuego necesitará para su correcto funcionamiento. Está compuesto de parsers necesarios para una correcta lectura de ficheros y posteriormente convertir dichos datos en

objetos del videojuego. También incluye toda la información del jugador, enemigos y resto de elementos de los niveles.

Esta sección es el punto de acceso al videojuego y es el encargado de integrar las distintas secciones y que todo funcione correctamente. Además contiene el bucle principal que resulta de gran interés en este proyecto.

5.2.1.2 Parsers

Esta versión del Space Invaders incorpora **parsers** para la lectura de ficheros con diversa información. Existen ficheros de entrada que configuran la inicialización del sistema, es decir, las configuraciones estándares de los diversos objetos y las propiedades principales del sistema. También existen parsers para la lectura de ficheros de niveles. Cada nivel está especificado mediante un fichero HTML que refleja cada objeto y la posición entre otros datos de cada entidad que se pueda encontrar en el nivel. Otro de los parsers más importantes es el parser dedicado a las FSM. Las FSM (término inglés para Maquina de Estados Finito) son unos modelos de comportamiento en la que se definen los distintos estados y las conexiones entre estados que puede adoptar una entidad. Las FSM se emplean en el videojuego para definir la inteligencia artificial de las distintas naves. Como se ha podido comprobar, los parsers son unas implementaciones necesarias para los videojuegos.

5.2.1.3 Player

La clase **Player** implementa tanto los gráficos para el jugador como el manejo a la respuesta del jugador o la cantidad de vidas que el jugador dispone entre otros. El jugador únicamente se desplaza horizontalmente y puede disparar a los enemigos.

5.2.1.4 Navy

La clase **Navy** es un gestor de naves enemigas que permite la actualización y la visualización de todas las naves de manera cómoda. Además, se encarga también de la inicialización de dichas naves.

Los distintos tipos de naves que componen el objeto Navy son: Ship, Supplyship y Circleship.

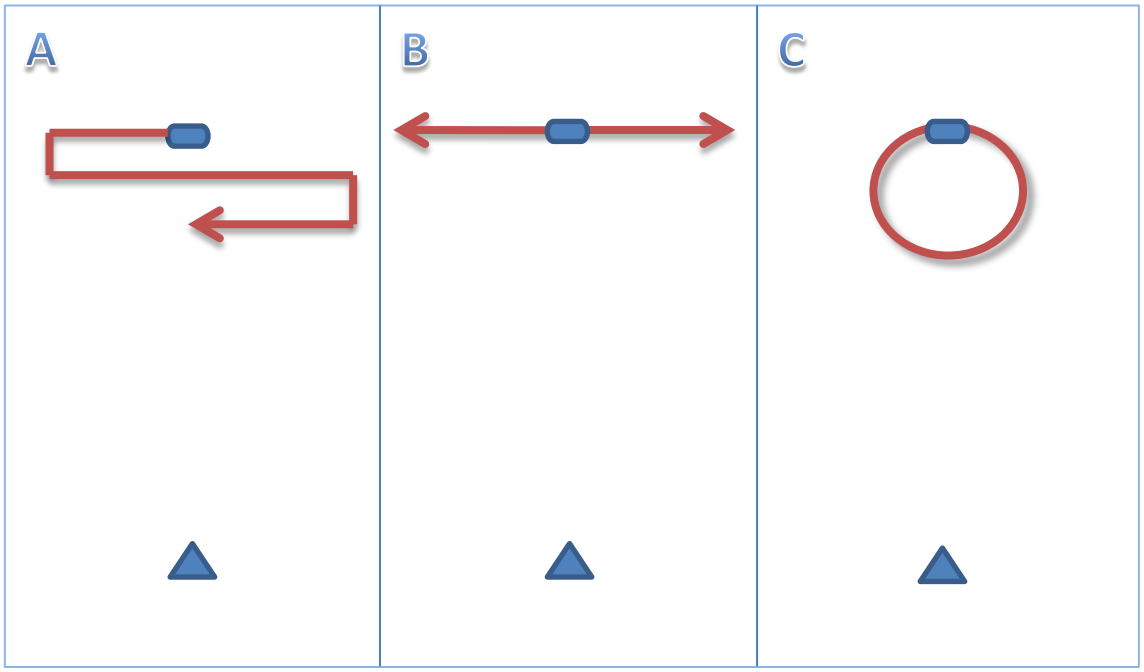


Ilustración 13 – Representación del movimiento de las naves Ship (A), Supplyship (B) y Circleship (C)

Los **ship** son los enemigos más sencillos del videojuego. Se presentan en gran cantidad en cada uno de los niveles. Suelen estar en formación rectangular y se desplazan horizontalmente. Cuando las naves alcanzan uno de los laterales de la pantalla, descienden levemente e invierten la dirección de desplazamiento. Estas naves disparan y avanzan hacia el jugador por lo que son un enemigo potencial. Además, conforme descienden su velocidad incrementa, aumentando la dificultad de acertar los disparos por parte del usuario.

Los **Supplyship** son unas naves algo mayores que las Ships. Se desplazan en horizontal hasta alcanzar uno de los laterales de la pantalla y retorna en sentido contrario al anterior. Esta nave, a diferencia de las Ships, no desciende por la pantalla ni aumenta su velocidad. Lo que si realiza al igual que las Ships es que ambas disparan.

5.2.1.5 Circleship

Las naves Circleship realizan un movimiento en sentido circular ya sea en sentido horario o antihorario e incluso en un sentido y cambiando de dirección cuando ha completado un movimiento circular. Esta nave se desplaza siempre alrededor de un punto fijo a una distancia fija. Además esta nave también dispara al jugador. Esta clase fue creada a lo largo del proyecto actual pero forma parte de las actuaciones previas de mantenimiento del código inicial.

5.2.1.6 ShootsManager y Shoot

La implementación **Shoot** no es más que la representación de los disparos. En esta clase se encuentra la dirección, velocidad y procedencia de cada disparo. En cada fase de actualización la posición de la nave se renueva y se detecta si se ha realizado alguna colisión.

El manejador de disparos, el **Shootsmanager**, gestiona todos y cada uno de los disparos que se pueden ver en pantalla. Cuando un disparo desaparece de pantalla o colisiona con algún objeto, el Shootsmanager almacena el disparo en un pool o almacén de disparos inactivos. Cuando se realiza un disparo por parte de alguna nave, el manejador extrae un disparo del pool o en caso de que el pool esté vacío, el manejador se encargará de crear un disparo nuevo.

Las **API** representan un gran porcentaje de la aplicación y permiten cargas de texturas y mallas 3D, carga y ejecución de sonidos, algoritmos de colisiones, representación de bunkers, además de otra serie de funciones que permiten la ejecución y la integración de todos los componentes.

5.2.2 Funcionamiento dinámico

En esta sección se muestra el flujo de datos además del funcionamiento interno del videojuego.

5.2.2.1 Inicialización

La **inicialización** comienza con una configuración gráfica de la aplicación. A continuación se inicializan las variables internas del sistema mediante el parser de inicialización. A continuación se ejecuta el parser de carga de niveles y se carga el primero de ellos. A partir de este momento se inicia el proceso más importante de un videojuego, el bucle principal.

5.2.2.2 Bucle Principal

En el caso actual en el **bucle principal** se ejecutan las dos fases principales: la fase de actualización y la fase de presentación. Además, en cada ejecución del bucle se ejecuta una tercera fase, un procesamiento de los mensajes que el sistema operativo envía a la aplicación.

5.2.2.3 Actualización

En cada ejecución de la fase de actualización se realiza la actualización de Navy. Después de actualizar la posición y el estado de la IA del enemigo, el Player es la siguiente sección en ser actualizado. Por último se actualiza el Shootsmanager y con este, las colisiones de las balas con su entorno. Antes de abandonar la fase de actualización se comprueba si el nivel ha terminado.

5.2.2.4 Visualización

En esta fase, se generan las llamadas a la librería OpenGL para generar las imágenes que se muestran por pantalla. A lo largo de la fase de visualización se completa el buffer de imagen para al final de la fase realizar su presentación mediante la pantalla.

5.2.2.5 Cambio de nivel

Cuando el nivel llega a su fin se realiza un cambio de nivel a excepción de que el nivel jugado fuera el último. Para cambiar el nivel, el paso inicial es liberar de memoria

todos aquellos aspectos u objetos del nivel finalizado. A continuación se requiere una carga del nuevo nivel mediante su respectivo parser. Después de realizar esta carga, se inicializan los objetos del nivel y la ejecución del sistema. Previo a todos estos pasos, se genera y se presenta una imagen estática informando del cambio de nivel.

5.3 Diseño actual (Modelo continuo acoplado)

En la sección actual se presentan los cambios tanto estáticos como dinámicos que se han producido en el sistema con la finalidad de adaptar el banco de pruebas a un sistema continuo acoplado.

5.3.1 Cambios estáticos

Los niveles o fases vienen determinados por unos ficheros de configuración. En estos ficheros se pueden especificar la cantidad, posición y velocidad de las distintas naves. Estos ficheros de configuración y sus correspondientes parsers han sido modificados para poder introducir mediante estos ficheros el tiempo de refresco que deben emplear para enviarse a ellos mismos los mensajes de actualización o visualización. De esta manera se pueden especificar más variables y con ello se obtiene una mayor flexibilidad para poder realizar distintas pruebas.

Adicionalmente se han incorporado unos temporizadores para poder realizar los cálculos respecto al tiempo invertido en actualización, visualización y tiempo improductivo del sistema. Estos temporizadores son de alta precisión por lo que los resultados obtenidos son lo más rigurosos posible.

5.3.2 Cambios dinámicos

En cuanto al plano dinámico, la evolución del videojuego en este aspecto no dista mucho de la ejecución original. Cada vez que el videojuego accede a la fase de actualización o de visualización el sistema registra el tiempo empleado en ejecutar dichas fases y cuando la ejecución del videojuego finaliza, el sistema almacena todos los datos recogidos por los temporizadores.

5.4 Diseño actual (Modelo continuo acoplado temporizado)

En el capítulo actual se informa de los cambios realizados para adaptar el modelo anterior al modelo continuo acoplado temporizado necesario para ejecutar las pruebas necesarias en el banco de pruebas.

5.4.1 Cambios estáticos

A nivel del parser, se han realizado varios cambios para poder importar la frecuencia de cada uno de los objetos. Además, se han empleado los temporizadores incorporados anteriormente para la gestión de las frecuencias de cada sección del videojuego. Para el correcto funcionamiento de estos temporizadores, también se ha creado un método en el que se inicializan los temporizadores y se asigna a cada objeto su frecuencia correspondiente

5.4.2 Cambios dinámicos

Inicialmente realiza la carga del fichero correspondiente al nivel o prueba actual. Entre otros datos, se obtienen las frecuencias de los objetos que se asignan a cada objeto a continuación

En el caso de una simulación empleando tiempo real, se ejecuta de manera imparable el bucle principal. En cada ejecución del bucle principal se recorren todos los objetos en búsqueda de aquellos que se deban actualizar. Tras comprobar el temporizador del objeto se decide si debe o no debe actualizarse. En caso de actualizarse, se vuelve a poner a cero el temporizador. Después de comprobar que objetos requieren actualizarse, se comprueba el temporizador asignado a la visualización para determinar si se debe generar una nueva imagen del sistema. El sistema ejecuta estas operaciones hasta el final de la prueba o de la partida.

Por otro lado, en el caso de emplear el tiempo de simulación, cada vez que se ejecuta el bucle principal se realizan las mismas operaciones. Primeramente se busca entre todos los temporizadores asignados a cada sección de actualización de los

objetos así como al temporizador de presentación del sistema cual se debe ejecutar en primer lugar. Si el tiempo de ejecución de dicha sección aún no ha llegado, se pasa a la ejecución de operaciones del sistema operativo hasta que dicho momento de ejecución llegue. Una vez iniciada y finalizada una sección, se actualiza el temporizador y se vuelve a iniciar en bucle principal.

5.5 Diseño actual (Modelo discreto desacoplado)

Por último, se exponen los cambios realizados a los modelos anteriores para poder incorporar el modelo discreto discontinuo al banco de pruebas.

5.5.1 Cambios estáticos

Para incorporar RT-DESK al videojuego y así lograr una implementación del Space Invaders con el modo de ejecución discreto desacoplado, fue necesario modificar la implementación de cada objeto que requiriese comunicación e independencia en RT-DESK. Dichas modificaciones fueron muy leves. Cada uno de dichos objetos fue redefinido como clase heredera de RTDESK-cEngine (Ilustración 14).

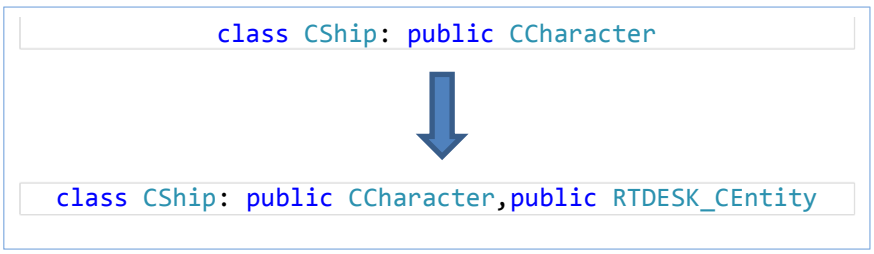


Ilustración 14 – Conversión de clases a entidades RT-DESK

Al heredar de RTDESK_cEngine, era necesario declarar y definir una función que gestionase la recepción de mensajes para cada objeto. Cuando un objeto reciba un mensaje, esta función deberá realizar una serie de acciones definidas según el tipo y los atributos de dicho mensaje. Esta clase se declara en RT-DESK como ReceiveMessage (Ilustración 15).

```
void CShip::ReceiveMessage(RTDESK_CMsg *pMsg){…}
```

Ilustración 15 – Método ReceiveMessage empleado para la- recepción de mensajes por parte de las entidades RT-DESK

Para que esta función pueda realizar su cometido, es necesario declarar una variable que le indique al objeto la frecuencia con la que debe re-enviarse un mensaje de actualización (Update). Por ello se declaró la variable correspondiente en cada uno de los objetos.

Los objetos que requieren dicha autonomía y comunicación con el sistema RT-DESK son:

- SupplyShip (para realizar su actualización)
- CircleShip (para realizar su actualización)
- Navy (para realizar la actualización de los Ships)
- SIGame (para realizar la visualización del Videojuego)
- ShootManager (encargado del movimiento de los Shoots)

Además de las modificaciones en cada uno de dichos objetos, es necesaria una inicialización de RT-DESK y de todas aquellas variables de los objetos para que funcione en RT-DESK.

A parte de las modificaciones de los objetos, su inicialización y la puesta en marcha del sistema, era necesario modificar el bucle principal. En el lugar en el que se ejecutarían las fases de actualización y de visualización en el caso de un modelo básico, se modifica por una ejecución del motor RT-DESK.

5.5.2 Cambios dinámicos

Lo primero es inicializar el sistema RT-DESK para poder ejecutar el sistema con el método discreto desacoplado y que pueda manejar los mensajes de inicialización de cada objeto (Ilustración 16). A continuación se configura el despachador de mensajes y

se enlaza con todos los objetos (Ilustración 17). Por último se generan los mensajes iniciales con los que se debe ejecutar el sistema de manera que los objetos comiencen sus actividades (Ilustración 18).

```
void CSIGame::LoadRTDESK(){

        NextSimulation=0;
        saveAccepted=false;

        RTDESK_Engine=new RTDESK_CEngine();
        RTDESK_Engine->Reset();
        RTDESK_Engine->StartUp();
```

Ilustración 16 – Método de inicialización. Sección de arranque del sistema.

```
RTDESK_Engine->MsgDispatcher->MsgPoolManager
        ->SetMaxMsgTypes(RTDESKMSG_MAX_TYPE);

RTDESK_Engine->MsgDispatcher->MsgPoolManager->SetMsgType();

for(unsigned int i=0;i<Navy.CircleShip.size();i++){
      Navy.CircleShip[i]->SetMsgDispatcher(RTDESK_Engine->MsgDispatcher);
}
```

Ilustración 17 – Método de inicialización. Sección de configuración y enlazado del despachador de mensajes.

```
for(unsigned int i=0;i<Navy.Ship.size();i++){
      Navy.Ship[i]->msg=Navy.Ship[i]->GetMsgToFill(RTDESKMSG_UPDATE);
      Navy.Ship[i]->SendMsg(
         Navy.Ship[i]->msg,
         Navy.Ship[i],
         Navy.Ship[i]->TimeToUpdateShip);
}
```

Ilustración 18 – Método de inicialización. Sección de envío inicial de mensajes del sistema.

En cada entrada al bucle, el motor RT-DESK realizará el envío de mensajes pendientes y cuando entre en espera, devolverá al sistema el tiempo (medido en ticks de reloj) que deberá transcurrir hasta la siguiente dispensación de mensajes.

6 Experimentación

Con este experimento se quiere mostrar las diferencias entre los modelos que se han presentado anteriormente.

Para valorar las implementaciones de los distintos modelos de ejecución se medirán y analizaran distintas características incluidas entre todas las listadas en la ISO/IEC previamente presentada (ISO/IEC-25010-3, 2009). Los experimentos se centrarán en la división **Calidad del Producto Software** ya que la calidad de esta división es la que varía en caso de cambiar la implementación del sistema.

Las pruebas que se realizaran durante la experimentación serán las siguientes:

- Rendimiento del sistema
- Precisión del estado final
- Obtención de tiempo ocioso
- Exactitud del tiempo de simulación

6.1 Rendimiento del sistema

6.1.1 Objetivos del estudio

Este estudio tiene por objetivo el comprobar la capacidad de los modelos para obtener una tasa de FPS acorde a unos requisitos solicitados mediante las QoS. Una de las características que pueden llevar a usar uno de los modelos puede ser el cumplimiento de exigencias gráficas.

6.1.2 Prueba a realizar

Para poder comprobar si los modelos anteriores son capaces de cumplir una tasa de FPS se va a reproducir una batería de pruebas en las que el número de naves se incrementa desde las 50 naves hasta las 2000. El incremento se producirá con un paso de 50 naves. Esta batería de pruebas se va a ejecutar en cuatro entornos en los que variaremos de gráficos 2D a 3D y la frecuencia de imágenes de 60Hz a 30Hz.

6.1.3 Resultados

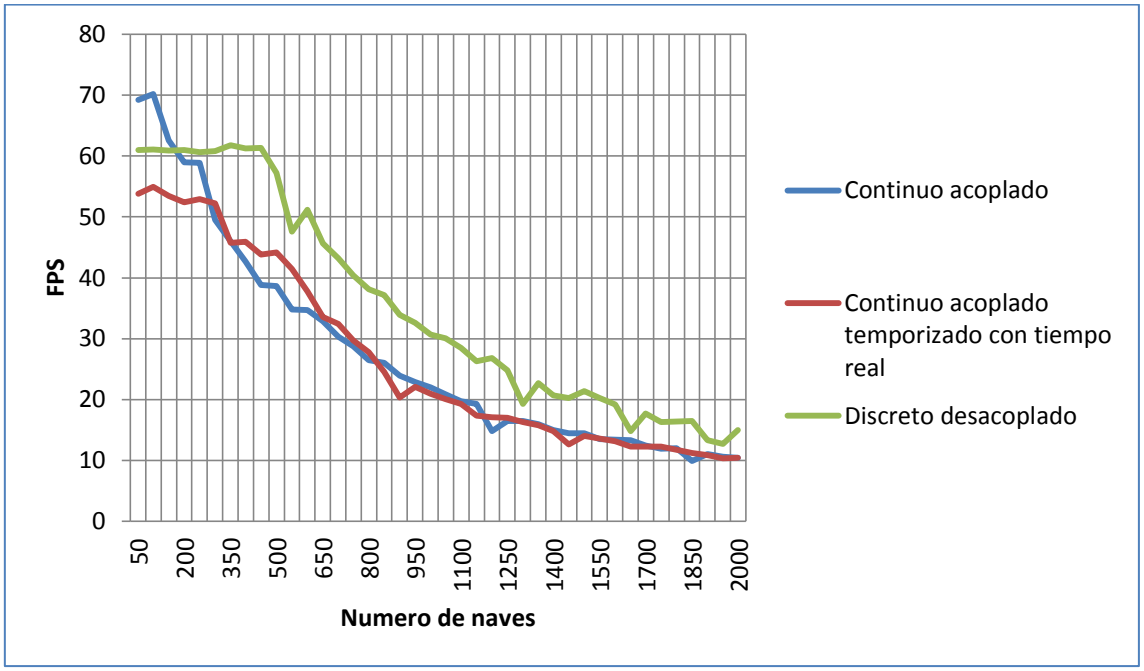


Ilustración 19 – FPS de los distintos modelos incrementando el número de naves con un paso de 50 naves. Se solicita una frecuencia de 60 Hz y unos gráficos 2D

En la ilustración 19 se muestran los resultados obtenidos al realizar la batería de pruebas a los modelos continuo acoplado, continuo acoplado temporizado y discreto desacoplado. En este caso, se ha impuesto una frecuencia de 60 Hz y gráficos 2D.

Como se puede comprobar, cuando el número de naves es inferior a 150 el modelo continuo acoplado excede la frecuencia solicitada y a partir de este punto la tasa desciende de manera inversamente logarítmica. Mientras tanto el modelo discreto mantiene una tasa aproximada de 60 FPS. Hasta las 450 naves, el sistema discreto desacoplado permanece en la misma tasa de gráficos solicitados. A partir de ese momento, la tasa del sistema desciende de la misma manera que el modelo continuo acoplado pero en todo caso la tasa del modelo discreto desacoplado se mantiene por encima. En el caso del modelo continuo acoplado temporizado, jamás alcanza la tasa solicitada tal y como se puede extraer por su naturaleza.

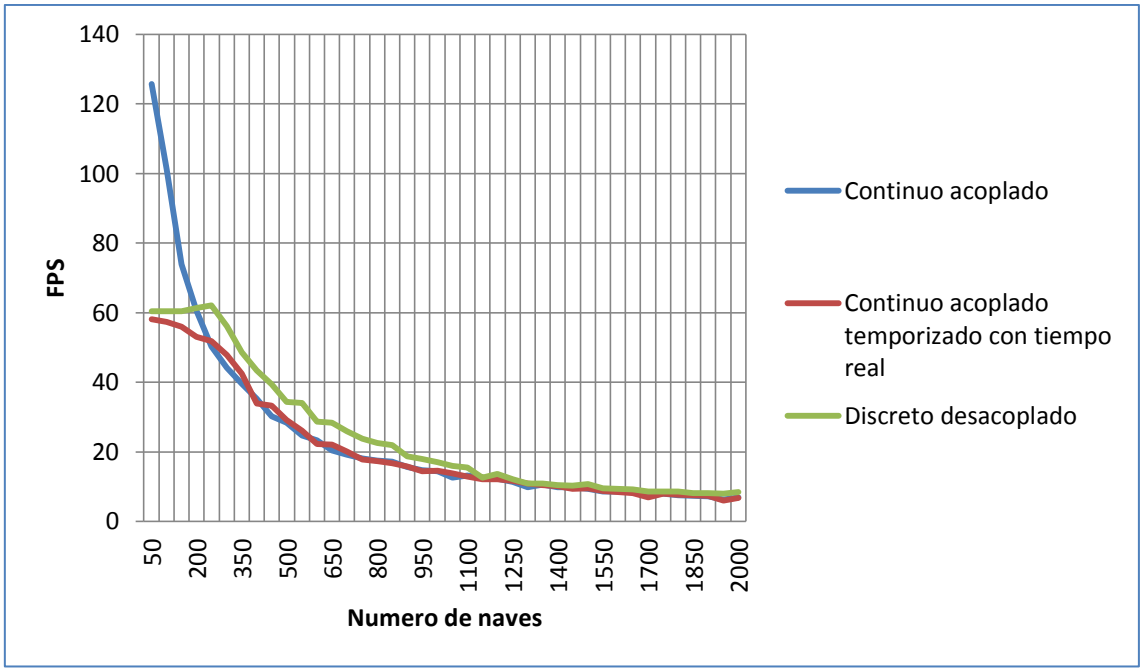


Ilustración 20 – FPS de los distintos modelos incrementando el número de naves con un paso de 50 naves. Se solicita una frecuencia de 60Hz y unos gráficos 3D

La Ilustración 20 presenta los resultados obtenidos de ejecutar la batería de pruebas con gráficos 3D y la frecuencia solicitada vuelve a ser de 60Hz.

En este caso se observa el mismo suceso que anteriormente, el modelo continuo acoplado supera los FPS del modelo discreto desacoplado por la baja carga del sistema al inicio de la batería de pruebas. El modelo discreto desacoplado mantiene la tasa hasta las 250 naves. A partir de este valor, la tasa desciende al igual que anteriormente de manera inversamente logarítmica. El modelo continuo acoplado temporizado vuelve a mostrar una vez más la incapacidad de alcanzar la frecuencia determinada.

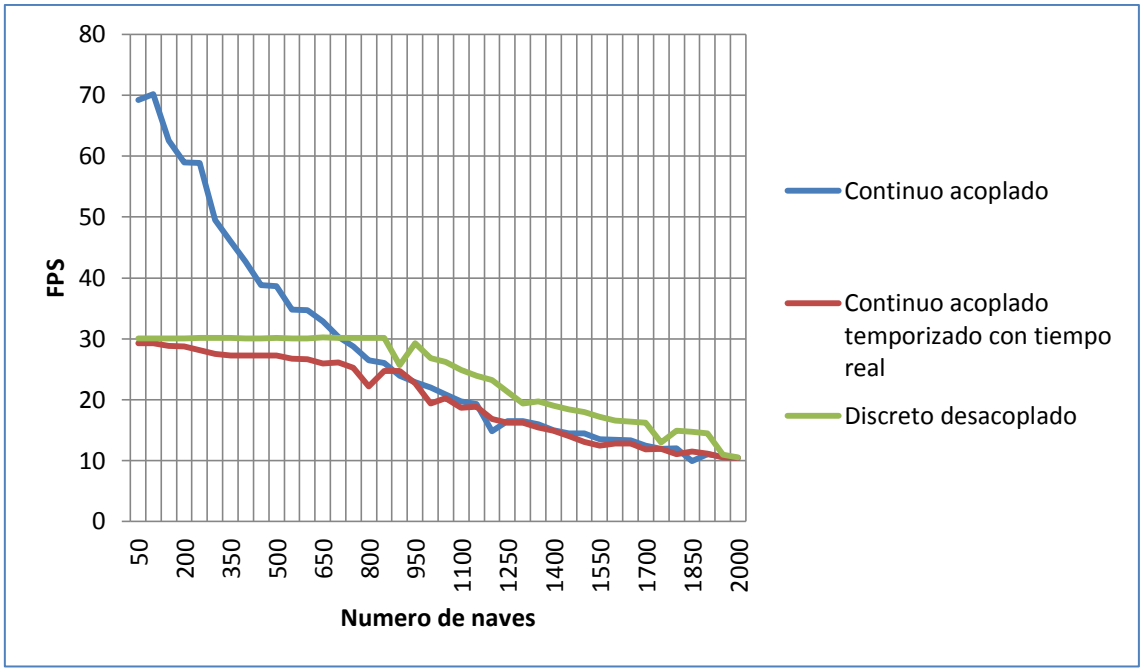


Ilustración 21 – FPS de los distintos modelos incrementando el número de naves con un paso de 50 naves. Se solicita una frecuencia de 30Hz y unos gráficos 2D

En este caso, la ilustración 21 muestra el número de FPS obtenidos por los sistemas en una batería de pruebas donde se solicita una frecuencia de 30 FPS y gráficos 2D. Una vez más, el modelo continuo acoplado supera inicialmente la tasa solicitada. En esta ocasión, se mantiene por encima del resto de modelos hasta las 700 naves. El modelo discreto desacoplado mantiene la tasa hasta las 850 naves mientras que el modelo continuo acoplado temporizado sigue por debajo de la tasa y a su vez, por debajo que el resto de modelos.

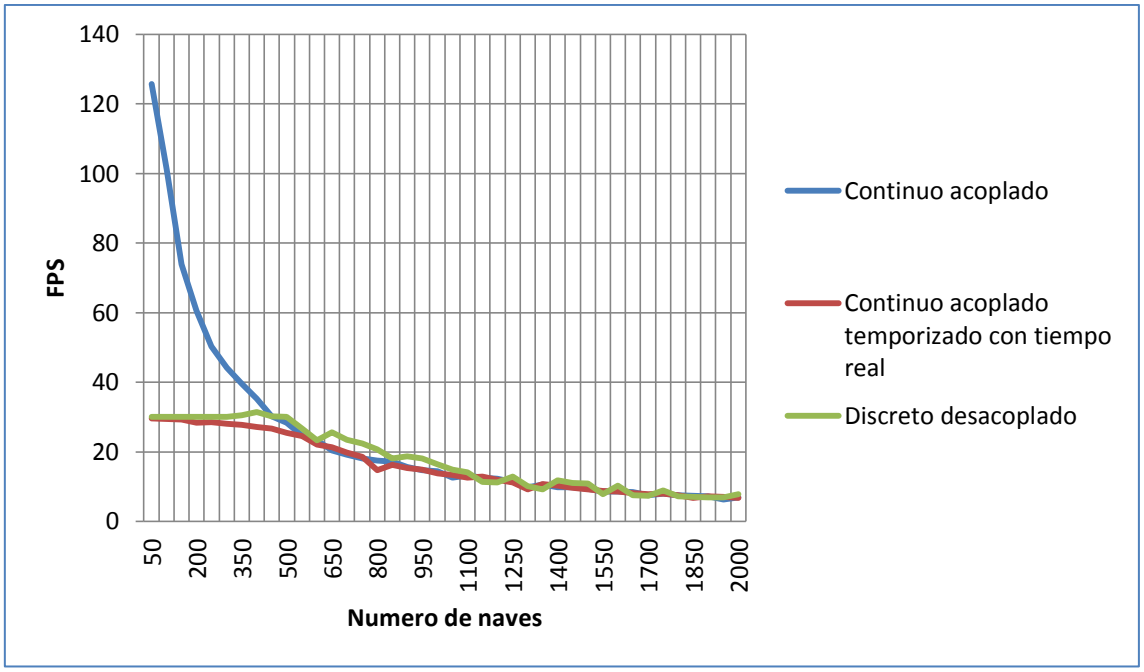


Ilustración 22 – FPS de los distintos modelos incrementando el número de naves con un paso de 50 naves. Se solicita una frecuencia de 30Hz y unos gráficos 3D

Por último entre las pruebas de este estudio se encuentra la prueba de los modelos imponiendo una frecuencia de 30 Hz empleando gráficos 3D. Al igual que en los tres casos anteriores, el modelo continuo acoplado supera con creces la tasa solicitada. El modelo discreto desacoplado mantiene la tasa gráfica hasta las 400 naves. A partir de este punto, ambos modelos disminuyen por debajo de la tasa solicitada. El sistema continuo acoplado temporizado se mantiene muy cercano al sistema discreto desacoplado a excepción de algunos casos en los que el sistema continuo supera al discreto.

6.1.4 Conclusiones

Gracias a los resultados obtenidos y mostrados anteriormente se puede señalar que el modelo continuo acoplado temporizado se presenta como la peor alternativa pues en ningún caso ha sido capaz de alcanzar la tasa solicitada. En cuanto al modelo continuo acoplado, muestra tanto excesos como defectos en la tasa gráfica. Pese a que con un exceso de imágenes se cumple el teorema del muestreo se desperdician

muchos recursos. En contraposición se puede encontrar el modelo discreto desacoplado. El sistema se mantiene en la frecuencia impuesta con una ligera varianza si el sistema no se encuentra saturado. En cuanto el sistema se satura desciende al igual que el resto de modelos aunque presentan una mayor tasa de imágenes que estos.

Desde la experiencia subjetiva del usuario el cambio de modelos no se aprecian diferencias en cuanto a la fluidez del videojuego mientras exista suficiente potencia de cálculo. Pero si se encuentran diferencias en casos próximos a la saturación del modelo discreto desacoplado donde este es capaz de mantener los FPS solicitados frente al resto de modelos que se encuentran con dificultades en realizar sus cálculos.

Por estas razones, el modelo que mejor rendimiento gráfico muestra es el modelo discreto desacoplado.

6.2 Precisión del estado final

6.2.1 Objetivos del estudio

El estudio actual se realiza para medir la precisión de resultados entre los distintos modelos puestos a examen en este proyecto. Para ello, se realiza la misma ejecución para cada modelo variando únicamente los gráficos entre 2D y 3D. Al variar los gráficos, la fase de presentación dura más tiempo obligando al sistema a realizar un cómputo mayor, pero las colisiones se ejecutan de la misma manera y por ende, el número de naves en pantalla al final de la ejecución debe ser igual en ambos casos.

6.2.2 Prueba a realizar

Se realiza una batería de ejecuciones incrementando el número de naves desde 50 hasta 2000 naves. El incremento es de 50 naves en cada paso. Para cada modelo se ejecuta la batería con gráficos 2D y se vuelve a ejecutar la batería con gráficos 3D. El entorno es el mismo para todas las ejecuciones.

6.2.3 Resultados

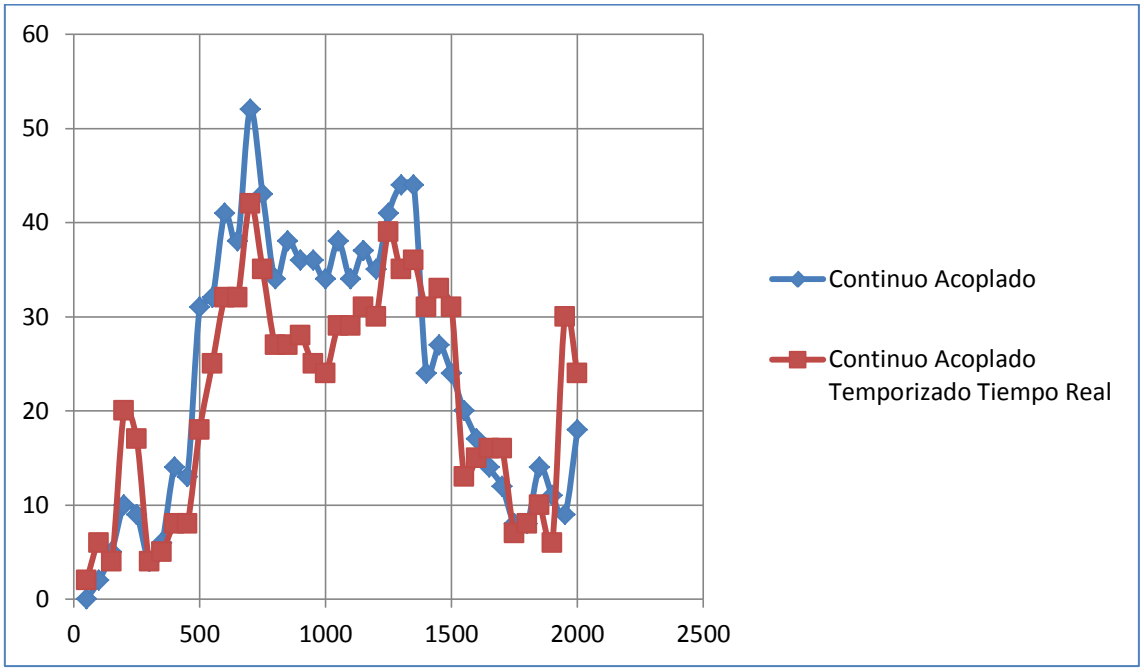


Ilustración 23 – Número de naves restantes al ejecutar las pruebas con distintos tipo de ejecución

En la Ilustración 23 se contempla la diferencia de naves restantes al final de cada ejecución entre ejecuciones con gráficos 2D y 3D. Los resultados devueltos por el sistema discreto desacoplado no aparece reflejado en la gráfica pues por definición, al ejecutarse todos y cada uno de los mensajes, pese a que el sistema deba realizar más cálculos para la ejecución con gráficos 3D en comparación con la ejecución con gráficos 2D, el tiempo de simulación no avanza y por tanto las distintas partes del código se ejecutarán en el momento indicado obteniendo una diferencia nula. Como se puede observar, en la ejecución del modelo continuo acoplado muestra diferencias entre las ejecuciones con gráficos 2D y 3D pues por la propia definición del sistema hace que todas las naves ejecuten la fase de actualización sin una frecuencia fija e induciendo a fallos. Por otro lado el modelo continuo acoplado temporizado también muestra una diferencia entre las ejecuciones con gráficos 2D y 3D ya que las naves no están sincronizadas debido a que la frecuencia de actualización siempre será menor a la indicada y variará levemente a lo largo de la ejecución.

6.2.4 Conclusiones

Los modelos continuo acoplado y continuo acoplado temporizado muestran no se ajustan al requisito de precisión que se recomienda tener tanto en videojuegos como en simulaciones de cualquier otro calibre. Sin embargo, el modelo discreto discontinuo, por su propia naturaleza realiza ejecuciones exactas de manera que se convertiría en el modelo más adecuado resultante del presente estudio.

La experiencia subjetiva del usuario muestra indiferencia entre modelos siempre y cuando no se realice una ejecución donde el usuario no intervenga empleando un modelo discreto desacoplado y otra ejecución con cualquier otro modelo en cuyo caso la diferencia de resultados puede ser muy evidente.

6.3 Obtención de tiempo ocioso

6.3.1 Objetivos del estudio

En el presente estudio se desea comprobar la eficiencia del sistema mediante la obtención de tiempo ocioso en las ejecuciones. El tiempo ocioso permite reanudar ejecuciones del sistema operativo o ejecutar aplicaciones en segundo plano. Otra de las razones por la que resulta interesante estudiar la eficiencia de los modelos es hacer una plataforma más eficiente energéticamente.

6.3.2 Prueba a realizar

Se realiza una batería de ejecuciones incrementando el número de naves a paso de 50 naves desde 50 hasta 2000 naves. Se reproduce la batería en los modelos continuo acoplado, continuo acoplado temporizado y discreto desacoplado. Todas las ejecuciones se producen con una frecuencia gráfica de 60 Hz. Además las pruebas se realizan tanto con gráficos 2D como 3D. En cada ejecución se registra el porcentaje de tiempo de ejecución se dedica a tiempo ocioso.

6.3.3 Resultados

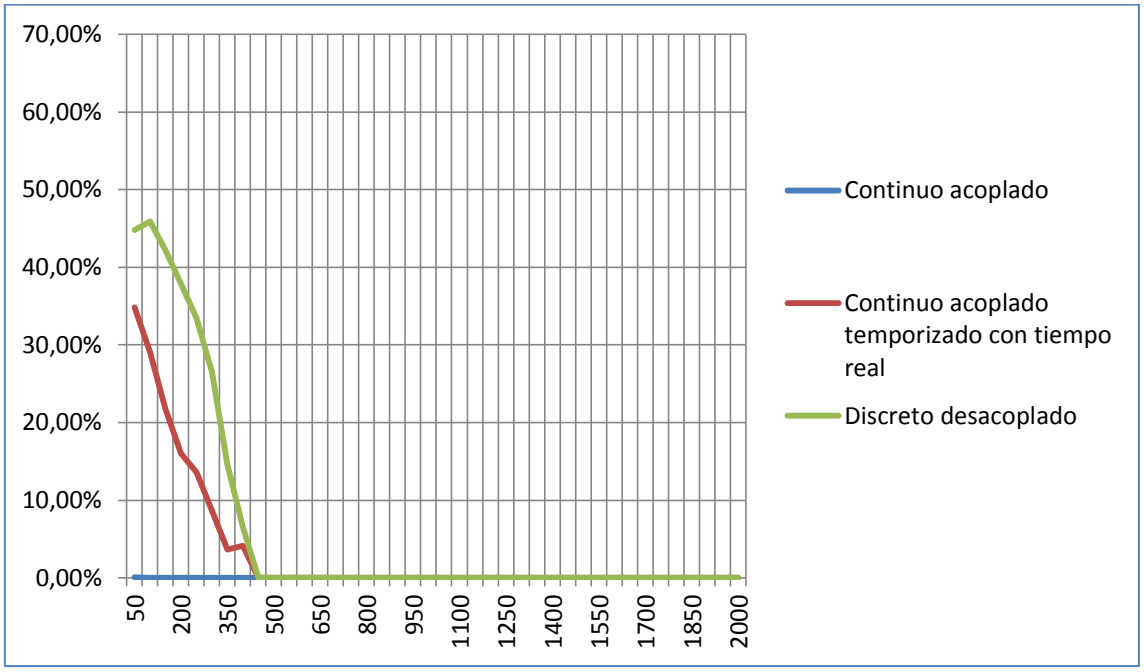


Ilustración 24 – Gráfica mostrando el porcentaje de estado ocioso para los distintos modelos. Ejecución de la fase de presentación a 60 Hz y ejecución 2D

En la ilustración 24 se comprueba como el modelo continuo acoplado no ofrece ningún tiempo ocioso sean cuales sean las condiciones. El sistema continuo acoplado temporizado ofrece tiempo ocioso pero el porcentaje obtenido se mantiene más bajo que para el caso del sistema discreto desacoplado. Este último supera alrededor de un 25% el porcentaje de tiempo ocioso ofrecido por el sistema continuo acoplado temporizado. Por ejemplo, en la prueba con 50 naves el sistema continuo acoplado presenta un 0% frente al 35% del sistema continuo acoplado temporizado. El sistema discreto desacoplado ofrece un 45% por lo que el sistema discreto desacoplado ofrece un 28% más que el sistema continuo acoplado temporizado. Los sistemas que presentan tiempo ocioso se igualan cuando el sistema se satura, desapareciendo completamente este tiempo.

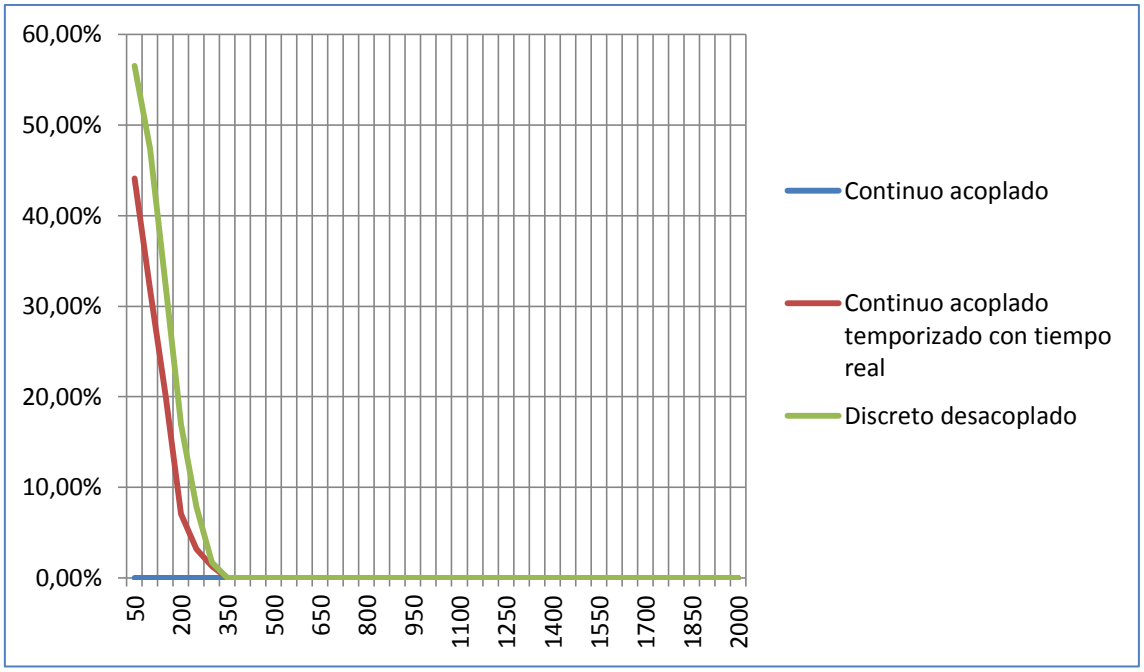


Ilustración 25– Gráfica mostrando el porcentaje de estado ocioso para los distintos modelos. Ejecución de la fase de presentación a 60 Hz y ejecución 3D

En la Ilustración 25 se contempla el rápido decrecimiento del tiempo ocioso y como desaparece dicho tiempo en cuanto el modelo se encuentra saturado. El modelo continuo acoplado nunca entra en estado ocioso ya que está en continua ejecución. Mientras tanto, el modelo continuo acoplado temporizado con tiempo real sí que entra en estado ocioso pero este modelo invierte menos tiempo que el modelo discreto desacoplado.

6.3.4 Conclusiones

Con los resultados observados anteriormente, el modelo continuo acoplado queda completamente descartado desde el punto de vista de la eficiencia. Respecto a los dos modelos restantes, se puede comprobar que el modelo discreto desacoplado ofrece un mayor porcentaje de tiempo ocioso frente al modelo continuo acoplado temporizado.

La experiencia subjetiva del jugador es de indiferencia entre los distintos modelos a la hora de valorar el tiempo ocioso del sistema. Es decir, las tareas en

segundo plano o la gestión del sistema operativo sucede de manera transparente para el usuario por lo que no son capaces de observar cambios.

6.4 Exactitud del tiempo de simulación

6.4.1 Objetivos del estudio

En el presente estudio se desea reflejar el distanciamiento entre tiempo real y tiempo de simulación que existen en los modelos continuo acoplado temporizado con tiempo simulado y el modelo discreto desacoplado. El distanciamiento entre tiempos puede representar la exactitud del sistema pues el tiempo se dilata (aumenta el distanciamiento) para poder procesar los eventos necesarios en su momento exacto.

6.4.2 Prueba a realizar

Se ponen a prueba el sistema continuo acoplado temporizado mediante tiempo simulado y el sistema discreto desacoplado. Para contemplar dos posibles ejecuciones del sistema continuo acoplado, se realizará la prueba con dos variantes. En una de las variantes los eventos se ejecutan en orden temporal mientras que en la otra variante los eventos se ejecutan según el orden implícito en el grafo de escena. Se pretende obtener resultados de un sistema no saturado por tanto se realiza una ejecución 2D con 10 naves que simula una partida habitual. La frecuencia gráfica es de 30 Hz.

6.4.3 Resultados

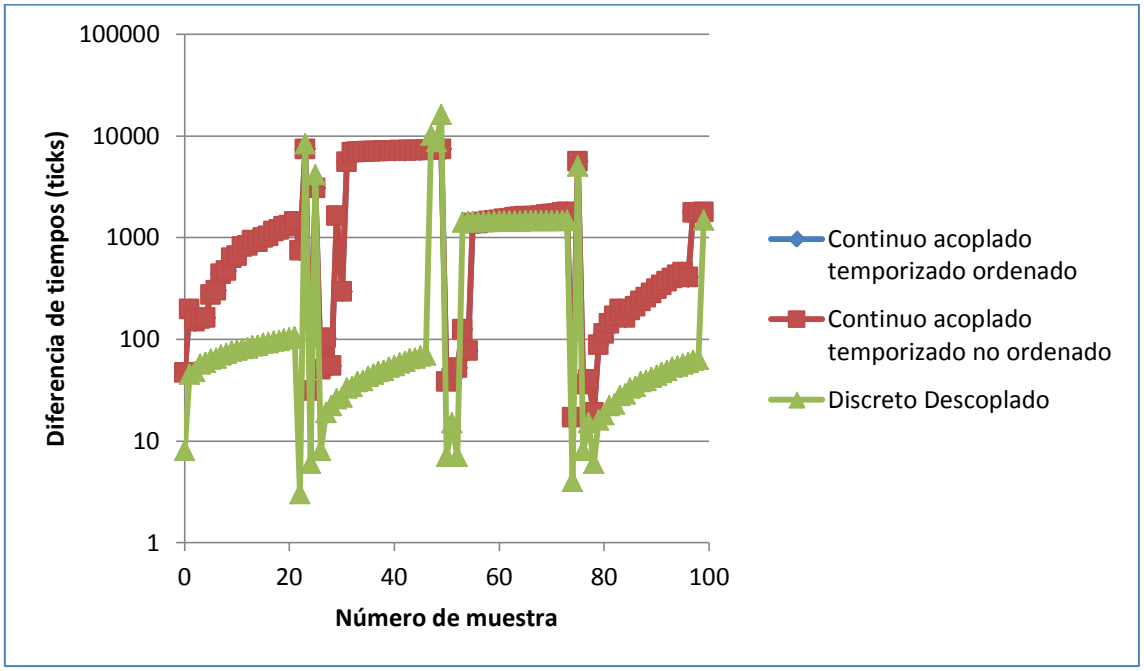


Ilustración 26 - Diferencia entre el tiempo real y el tiempo de simulación en una partida habitual

En la ilustración 26 se presentan las primeras 100 muestras de una ejecución habitual. El eje de coordenadas se muestra en escala logarítmica. La muestra del modelo continuo acoplado temporizado ordenado queda oculta tras la del modelo continuo acoplado temporizado no ordenado. Como se puede comprobar se muestra la elasticidad del tiempo simulado respecto al tiempo real. Se puede comprobar como el modelo discreto desacoplado es capaz de mantenerse más ajustado al tiempo real mientras que el resto de modelos se muestran más alejados.

6.4.4 Conclusiones

Pese a que existen diferencias entre los modelos continuo acoplado temporizado con tiempo de simulación y discreto desacoplado, son nimias las diferencias si se compara con la duración total de cualquier ejecución. Por ello, las conclusiones son que es indiferente el empleo de uno u otro modelo en situaciones donde se empleé tiempo de simulación.

Desde la subjetiva experiencia del jugador, no se observan diferencias entre ejecuciones empleando ambos modelos por lo que también resulta indiferente desde el punto de vista del usuario.

7 Conclusiones y resultados

El problema inicial era la búsqueda de una alternativa al paradigma continuo acoplado con el que se viene implementando el bucle principal de los videojuegos. Este paradigma presenta varios inconvenientes entre los que se encuentran los que se enlistan a continuación:

- La fase de actualización y la fase de visualización se ejecutan una tras otra de manera continua ejecutándose con la misma frecuencia ambas fases.

- Dentro de la fase de actualización todos los objetos se actualizan en el mismo instante de tiempo y se actualizan ordenadamente, es decir, el orden de ejecución de la fase de actualización de todos los objetos no varía y se ejecuta la actualización de todos los objetos independientemente de que realmente necesiten ejecutarse.

- Esta ejecución continua no permite determinar unas QoS y por tanto se hace muy difícil el control de cualquier frecuencia de ejecución.

- Al encontrarse acoplado y no poder ejecutar el código necesario en el preciso momento los resultados son habitualmente inexactos.

- La ejecución continua de las fases de actualización y presentación imposibilita que el sistema operativo ejecute código en segundo plano en el caso de que cuando finaliza una ejecución del bucle no se devuelva el control al sistema operativo.

- Además, la ejecución continua y acoplada puede producir imágenes que jamás serán mostradas pues el número de imágenes generadas por segundo pueden ser mayores a la tasa de refresco de la pantalla.

La solución propuesta es el empleo del paradigma discreto desacoplado substituyendo al clásico paradigma continuo acoplado. Este paradigma elimina gran cantidad de los inconvenientes listados anteriormente.

Para aplicar el paradigma discreto desacoplado se ha empleado RT-DESK. RT-DESK es una librería acompañada de sus respectivas cabeceras codificado en C++ que implementa un sistema de gestión de mensajes entre objetos permitiendo el paradigma discreto desacoplado. Contiene un núcleo gestor, un despachador de mensajes y un almacén de mensajes.

El núcleo gestor de RT-DESK contiene un reloj de simulación. Este reloj nunca superará el tiempo real y se mantendrá lo más cercano a este. Todo sistema que empleé RT-DESK deberá realizar sus cálculos en base al reloj de simulación.

El despachador de mensajes se basa en el tiempo de simulación marcado por el núcleo gestor repartiendo los mensajes en el tiempo especificado en el mensaje. De esa manera, las ejecuciones de las distintas partes que componen el videojuego pueden ejecutarse en un orden no preestablecido y pueden mantener distintas frecuencias.

Por último, el almacén de mensajes almacena todos aquellos mensajes que no están siendo empleados evitando la creación y destrucción con su consecuente coste en memoria y ejecución, de todos aquellos mensajes que se encuentran parados.

El uso de mensajes aporta flexibilidad al sistema. Además de permitir que el sistema se ejecute de manera discreta desacoplada, permite que parte o todo el sistema se ejecute de manera continua e incluso de manera acoplada. Por tanto, el empleo de RT-DESK en un videojuego implica una gran versatilidad ajustándose a las QoS de distintas plataformas. Gracias a esta flexibilidad se puede cambiar el paradigma en tiempo de ejecución e incluso ejecutar unos objetos con un paradigma y el resto con otro.

7.1 Conclusiones finales

De la experimentación se puede extraer que el paradigma discreto desacoplado ofrece mayor calidad del software frente a los modelos continuo acoplado y continuo acoplado temporizado tanto con tiempo real como con tiempo de simulación. Este paradigma ofrece al usuario una mayor exactitud en la ejecución, es fácilmente adaptable, y aporta mejoras tanto en el rendimiento del software como en la interacción con el usuario. Además, el paradigma discreto desacoplado puede ofrecer los mismos servicios para simulaciones científicas ya que los videojuegos son una clase de simulación y este paradigma se puede extender a todo tipo de simulación asegurando una correcta ejecución.

Desde mi experiencia personal, la integración de RT-DESK en un videojuego no supone ninguna complicación y aporta grandes mejoras a la calidad y rendimiento del videojuego además de apreciarse mejoras visuales desde la visión del jugador. El esfuerzo necesario para aplicar esta tecnología se encuentra en comprender las diferencias entre el clásico paradigma continuo acoplado y el discreto desacoplado, y lo que conlleva estas diferencias a la hora de programar el videojuego. Una vez salvado este problema, el programador observa las oportunidades que presenta el modelo discreto desacoplado frente al resto de modelos.

8 Trabajos futuros

Una línea de investigación extraíble de este trabajo puede orientarse a la monitorización del sistema discreto desacoplado de manera interna y automática. Este monitor podría ajustar las frecuencias de cada objeto e incluso bajando la potencia de procesamiento del procesador e incluso la frecuencia de las memorias, disminuyendo la energía necesaria y haciendo que el programa sea eficiente energéticamente hablando. Para ello puede emplear ACPI para llevar a cabo el manejo energético de los distintos recursos que se pueden localizar en una plataforma. Además podría detener momentáneamente el sistema para poder gestionar los mensajes proporcionados por el sistema operativo.

Otra línea de investigación sería realizar un estudio sobre diversos métodos de división de la fase de presentación para poder ejecutar de manera más eficiente el videojuego pues no sería necesario realizar más cálculos de visualización si no hace falta. Podría estudiarse el empleo de grafos de escena en este proceso o la disección y recomposición de los buffers de imágenes.

Se puede extender una nueva línea de investigación sobre el empleo de la GPGPU en aplicaciones que empleen el paradigma discreto desacoplado o en la gestión interna de una librería como RT-DESK. Esta adición supondría un aumento tanto de velocidad como de capacidad de procesamiento de manera que el videojuego funcionaría de manera más holgada. Esta tecnología se podría llevar a cabo mediante CUDA desarrollado por Nvidia o mediante C++ AMP (C++ Accelerated Massive Parallelism) desarrollado por Microsoft e incorporado en DirectX 11.

Una posible expansión de la librería RT-DESK podría ser el empleo y gestión de creación de hilos para su ejecución en diversos núcleos del procesador. Esta técnica se conoce como Multi-Threading on Multi-Processors.

Un

a última línea de investigación se desprende de este proyecto. La posibilidad de portar la librería RT-DESK a otros lenguajes de programación, convirtiendo la librería en un recurso más versátil que el conocido hasta estos momentos.

9 Agradecimientos

Agradezco a la Universidad Politécnica de Valencia así como al Departamento de Sistemas Informáticos y Computación la aceptación de esta tesina y la oportunidad que dicha aceptación me brinda. También quisiera agradecer la dirección de mi tesina a cargo de Ramón Pascual Mollá Vayá quien me encaminó en el transcurso de este proyecto y a Vicente Broseta Toribio por compartir las experiencias con RT-DESK. Un especial agradecimiento a M. Carmen Juan Lizandra por el interés mostrado y el deseo de que este proyecto llegase a buen término.

Un fuerte agradecimiento a mi familia y amigos por estar cerca en los momentos difíciles, apoyándome para que lograra avanzar y mostrando en repetidas ocasiones que ellos son los que realmente han hecho posible la creación de esta tesina.

Muchas gracias a todos.

10 Bibliografía

1. **Schultz, Charles P, Bryant, Robert y Langdell, Tim.** *Game testing all in one.* Boston : Thomson Course Technology, 2005.

2. **J. L. Gonzalez Sánchez, N. Padilla Zea, F. L. Gutiérrez, M. J. Cabrera.** *De la Usabilidad a la Jugabilidad: Diseño de Videojuegos Centrado en el Jugador.*

3. *Real Time Game Loop Models for Single-Player Computer Games.* **Valente, Luis, Conci, Aura y Feijó, Bruno.** s.l. : IV Brazilian Symposium on Computer Games and Digital Entertainmen, 2005.

4. *Inside discrete-event simulation software: How it works and why it matters.* **Schriber, Thomas J. y Brunner, Daniel T.** 2011. Proceedings of the 2011 Winter Simulation Conference. págs. 80-94.

5. **Broseta Toribio, Vicente.** *Cambio de paradigma en el control de aplicaciones gráficas en tiempo real.* DSIC (Departamento de Sistemas Informáticos y Computación). 2012. Trabajo fin de máster.

6. **García, Inmaculada García.** *Simulación híbrida como núcleo de simulación de aplicaciones gráficas en tiempo real.* DSIC (Departamento de Sistemas Informáticos y Computación), UPV (Universidad Politécnica de Valencia). 2004. Tesis doctoral.

7. *Game Engines in Scientific Research.* **Lewis, Michael y Jacobson, Jeffrey.** 1, s.l. : Communications of the ACM, January de 2002, Vol. 45, págs. 27-31.

8. **Gregory, Jason.** *Game Engine Architecture.* Massachusetts : A K Peters, 2009.

9. **Jason Busby, Zak Parrish, Jeff Wilson.** *Mastering Unreal Technology.* Indianapolis : Sams Publishing. Vol. 1.

10. **LaMothe, A.** *Tricks of the Windows Game Programming Gurus.* Indianapolis : Sams, 1999.

11. *An architecture with automatic load balancing for real-time simulation and visualization systems.* **Joselli, Mark, y otros, y otros.** 1, s.l. : Journal of Computational Interdisciplinary Sciences, 2010, Vol. 3, págs. 207-224.

12. *Core Techniques and Algorithms.* **Sanchez, D. y Dalmau, C.** s.l. : New Riders Publishing, 2003.

13. **Mömkkönen, Ville.** Multithreaded Game Engine Architectures. [En línea] 6 de Septiembre de 2006. [Citado el: 1 de Marzo de 2013.]

14. *Game engineering for a multiprocessor architecture.* **El Rhalibi, Abdennour, Costa, Steve y England, David.** s.l. : DIGRA Conference, 2006.

15. **Nyland, Lars, Harris, Mark y Prins, Jan.** Fast N-Body Simulation with CUDA. *GPU Gems 3.* 31, págs. 677-695.

16. *Automatic dynamic task distribution between cpu and gpu for realtime.* **Joselli, M., y otros, y otros.** s.l. : IEEE Proceedings of the 11th International Conference on, 2008, págs. 48-55.

17. *A new physics engine with automatic.* **Joselli, Mark, y otros, y otros.** 2008, Sandbox 08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games, págs. 149-156.

18. *A game loop architecture.* **Zamith, Marcelo P.M., y otros, y otros.** 42, s.l. : Computers in Entertainment, 2008, Vol. 6.

19. *Agent-based simulation of urban infrastructure asset management activities.* **Osman, Hesham.** 2012, Automation in construction, Vol. 28, págs. 45-57.

20. *Agent-Based Simulation Tutorial - Simulation of emergent behavior and differences between agent-based simulation on discrete-event simulation.* **Wai Kin,**

Victor Chan y Young-Jun, Son. 2010. Proceedings of the 2010 Winter Simulation Conference. págs. 135-150.

21. **Microsoft.** Application Model Overview. [En línea] Microsoft, 2013. http://msdn.microsoft.com/en-us/library/bb203873(v=xnagamestudio.20).aspx.

22. **Sanchez-Crespo, Daniel.** *Core Techniques and Algorithms in Game Programming.* s.l. : New Riders Publishing, 2003. ISBN: 0-1310-2009-9.

23. *Accelerating physics in large, continuous virtual environments.* **Daniel J. Tracy, Sheldon Brown.** 24, 2011, Concurrency and computation: practice and experience, págs. 125-134.

24. *Enriching 3-D Video Games on Multicores.* **Romain Cledat, Tushar Kumar, Jaswanth Sreeram, Santosh Pande.** 2011. International Parallel & Distributed Processing Symposium. págs. 176 - 187.

25. *Beyond Gaming: Programming the Playstation 3 Cell Architecture for Cost-Effective Parallel Processing.* **Rabbah, Rodric.** s.l. : Hardware/Software Codesign and System Synthesis (CODES+ISSS),2007 5th IEEE/ACM/IFIP International Conference, 2007.

26. *A Cross-platform Integration Method for Virtual Reality Applications.* **Cheng-jun, Chen, Yun-feng, Wu y Yi-qi, Zhou.** 2010. Third International Conference on Information and Computing. págs. 140-142.

27. *Implementation of Parallel Game Tree Search on a SIMD System.* **Wu, Dan, y otros, y otros.** 2010. WASE International Conference on Information Engineering. págs. 62-66.

28. *GDESK: Game discrete event simulation kernel.* **García, I., Mollá, R. y Barella, A.** 12, s.l. : Journal of WSCG, 2004, Vol. 1, págs. 121-128.

29. **Europa Press.** Atari se declara en bancarrota. [En línea] 21 de Enero de 2013. http://www.europapress.es/portaltic/empresas/noticia-atari-declara-bancarrota-20130121171630.html.

30. *DESK Discrete Events Simulation Kernel.* **García, I., y otros, y otros.** s.l. : Eccomas conf. proc, 2000.

31. *JDESK web-based discrete event simulation kernel.* **García, I., Mollá, R. y Cabanillas, J.** 2003.