DEPARTAMENTO DE INFORMÁTICA

SIMULACIÓN HÍBRIDA COMO NÚCLEO DE SIMULACIÓN DE APLICACIONES GRÁFICAS EN TIEMPO REAL

INMACULADA GARCÍA GARCÍA

UNIVERSITAT DE VALENCIA
Servei de Publicacions
2004

Aquesta Tesi Doctoral va ser presentada a Valencia el día 23 de Setembre de 2004 davant un tribunal format per:

- D. Roberto Vivó Hernando
- D. Mariano Pérez Martínez
- D. Miguel Chover Selles
- D. Rafael Orts Carot
- D. Emilio Camanort Gurdea

Va ser dirigida per:
D. Marcos Fernández Marín
D. Ramón Mollá Vayá

Depòsit legal:
I.S.B.N.:84-370-6085-0
Edita: Universitat de València
Servei de Publicacions
C/ Artes Gráficas, 13 bajo
46010 València
Spain
Telèfon: 963864115

UNIVERSITAT DE VALÈNCIA

Departamento de Informática

tesis doctoral

Simulación Híbrida como Núcleo de Simulación de Aplicaciones Gráficas en Tiempo Real

Presentada por:

Inmaculada García García

Dirigida por:

Dr. Marcos Fernández Marín

Dr. Ramón Mollá Vayá

Valencia, 2004

Dedicatoria

A Quique y a mis niñas, por tantas ausencias.

Agradecimientos

Esta tesis ha sido el fruto de años de trabajo. No hubiera sido posible sin la ayuda de mi familia, amigos y compañeros. Por ello, quiero dar las gracias:

A Ramón todo el apoyo que me ha dado durante la realización de esta tesis, más allá de lo que es un director de tesis. Sin Ramón esta tesis no habría sido posible.

A Marcos, por haber confiado en nosotros y por haber sacado tiempo de donde no lo había.

A Toni Barella (*pbm*) por todo el tiempo que me ha dedicado y por soportar mis nervios y prisas.

A mis compañeros. A Toni Garrido, por "otra duda de latex". Pero, especialmente a mi amiga Marisa, por todo.

A mi familia y amigos.

Gracias.

Como no sabían que era imposible, lo hicieron.
(Anónimo)

Yo creo bastante en la suerte.
Y he constatado que,
cuanto más duro trabajo,
más suerte tengo.
(Thomas Jefferson)

La inspiración existe, pero tiene que encontrarte trabajando.
(Pablo Ruiz Picasso)

Resumen

Las aplicaciones gráficas en tiempo real siguen un paradigma de simulación continuo acoplado. Este paradigma presenta diversos inconvenientes, entre ellos cabe destacar el bajo aprovechamiento de la potencia de cálculo de la máquina, la imposibilidad de definir la QoS de cada objeto y mantenerla durante la ejecución o el acoplo de los procesos de todos los objetos del sistema, en particular el acoplo del proceso de visualización del sistema con el resto de procesos.

La tesis propone cambiar el paradigma de simulación de estas aplicaciones a un paradigma discreto desacoplado. Este nuevo paradigma permite solucionar los problemas del paradigma anterior. Los objetos definen su propia QoS independientemente del resto del sistema, incluso se permite definir diferentes QoS para diferentes aspectos del propio objeto. Entre estos objetos, está el objeto visualizador, destinado a controlar el proceso de visualización. El objeto visualizador también define su propia QoS.

En el sistema discreto desacoplado cada objeto consume únicamente la potencia de cálculo estrictamente necesaria para llevar a cabo su simulación con la QoS definida. Por ello, la potencia de cálculo del sistema se reparte entre los objetos en función de sus necesidades.

El sistema puede adaptarse dinámicamente, redefiniendo la QoS de los objetos en función de las condiciones de la ejecución del sistema. Los objetos pueden degradar o mejorar su comportamiento durante periodos de la ejecución para evitar colapsos del sistema o para mejorar el comportamiento del sistema.

Abstract

Real time graphic applications, specifically videogames, follow a paradigm of continuous simulation that couple the simulation phase and the rendering phase. This paradigm can be inefficient or it can produce incorrect simulations. It has disadvantages, some of them are: the inadequate computer power distribution between the graphic application objects, it is not possible to define the Quality of Service (QoS) of each application object, the object QoS can be maintained during the application running, the behavior of all the system objects are coupled (specifically the rendering process and the simulation process).

The proposal is to change the simulation paradigm of real time graphic applications. The new simulation paradigm is discrete and decoupled. The use of a decoupled discrete paradigm avoids the problems of the continuous coupled paradigm and it avoids incorrect simulations, besides it improves the simulation quality and efficiency. The discrete simulation paradigm allows to define a private QoS criterion for each aspect of each object in the videogame. The render object is dedicated to control the application render process. The render object defines its own QoS criteria.

It is possible to define a different sampling frequency for each object aspect in the system. The discrete paradigm allows to define the objects sampling frequency to distribute the computer power adequately among the objects. The computer power consumed executing the application is only the necessary to guarantee the QoS of each object.

The system can be adapted dynamically. The objects QoS can be adjusted to the objects requirements and the whole system requirements, the system load or characteristics. This sampling frequency may change dynamically to adapt the QoS of the object aspect to the real computer power. The result obtained is a discrete system that allows a Smart System Degradation and may redefine dynamically the objects aspects QoS. Objects collect system information and use it to adapt its QoS.

Tabla de contenidos

Índice de figuras

Índice de tablas

Índice de algoritmos

Capítulo 1

Introducción

En el presente capítulo se presenta la motivación que ha originado esta tesis, así como los objetivos perseguidos. Seguidamente se detallan las aportaciones realizadas y finalmente se muestra la estructura general de la tesis que se desarrolla en los siguientes capítulos.

1.1. Motivación

El esquema de simulación tradicional de aplicaciones gráficas en tiempo real sigue un paradigma de simulación continuo. Este esquema tradicional de simulación revela una serie de deficiencias, de entre las cuales destacan las siguientes:

- No se hace un uso eficiente de la potencia de cálculo. La potencia de cálculo se utiliza para simular y visualizar a la máxima velocidad que el sistema es capaz de proporcionar, en vez de tener en cuenta las necesidades de los objetos a la hora de repartir la potencia del sistema. El reparto inadecuado de la potencia de cálculo provoca que los objetos se muestreen inadecuadamente: los objetos que necesiten un frecuencia de muestreo mayor, se submuestrean y los objetos que necesitan menor frecuencia de muestreo, se sobremuestrean. Los objetos submuestreados pueden tener comportamientos incorrectos y degradan la calidad de la simulación. Los objetos sobremuestreados desperdician potencia de cálculo que podría destinarse a los objetos submuestreados.
- No permite definir cual es la QoS de los objetos del sistema. El esquema continuo trata a todos los objetos del sistema por igual y asume que las necesidades de QoS son iguales en todo el sistema (asume una QoS global, no adaptada a las necesidades particulares de cada objeto.
- No permite mantener la QoS de los objetos durante la ejecución. La QoS global del sistema en el sistema continuo varía dinámicamente dependiendo de

la carga del sistema, por lo que un sistema continuo es incapaz de garantizar que se cumpla la QoS global. El programador de la aplicación no puede controlar determinados parámetros de la QoS, como la frecuencia mínima o máxima de muestreo. Otros parámetros si los puede controlar, como sistemas multirresolución.

- El esquema de simulación continuo produce un acople de las fases de simulación y visualización. La frecuencia de muestreo del sistema es la frecuencia de cuadro del sistema. No permite definir la frecuencia de cuadro ni garantizar que se mantenga durante la ejecución. Si bien es cierto que la mayoría de aplicaciones gráficas acoplan las fases de visualización y simulación, existen ciertas aplicaciones que, manteniendo un esquema continuo de simulación, permiten desacoplar estas fases. Pero el resto de la aplicación sigue manteniendo un esquema continuo. Este desacople se lleva a cabo, sobre todo, el aplicaciones de realidad virtual con el objeto de distribuir los procesos del sistema.

- La QoS de los objetos no puede adaptarse dinámicamente a las condiciones del sistema.

Las deficiencias del esquema de simulación continuo se estudian con mayor profundidad en el apartado 2.5.

Por tanto, este paradigma de simulación está obsoleto y deben buscarse nuevos paradigmas que solucionen las deficiencias. La búsqueda de un nuevo esquema de simulación de aplicaciones gráficas en tiempo real lleva obliga a buscar en otros campos o aplicaciones como se ha resuelto este problema. Un precedente importante es estudiar las técnicas utilizadas en el campo de simulación de sistemas.

Una posible clasificación de los simuladores de sistemas es dividirlos en sistemas discretos y continuos, como técnicas de simulación diferentes. Los simuladores discretos evitan el muestreo de los objetos, ejecutando sólo los eventos que generan los objetos y en el instante de tiempo de simulación en que estos se generan.

La técnica de simulación de eventos discretos puede adaptarse a aplicaciones gráficas en tiempo real para introducir un paradigma de simulación discreto que podría resolver los problemas del paradigma continuo. Durante el estudio realizado no se han encontrado aplicaciones gráficas en tiempo real que funcionen siguiendo un esquema discreto.

El esquema discreto podría mejorar los problemas planteados en el esquema continuo, permitiendo que:

- Sólo los objetos que generan eventos consumen potencia de cálculo. La potencia de cálculo consumida depende del número de eventos generados y del coste de

ejecución de dichos eventos. Por tanto la potencia se reparte entre los objetos que lo necesitan y en la medida en que lo necesitan.

- La QoS de un objeto depende, en parte, de la generación de eventos, por tanto el programador puede controlar la QoS del objeto controlando la frecuencia de generación de eventos. La QoS de cada objeto es independiente del resto de objetos.
- La QoS puede mantenerse durante la ejecución del sistema, pues todos los eventos se ejecutan y consumen el mismo tiempo de simulación, con independencia de la carga del sistema. Cada objeto controla su proceso de generación de eventos, por tanto, puede ser capaz de obtener información del sistema para, si es necesario, redefinir su QoS para adaptarse a la carga del sistema o las necesidades de otros objetos.
- La visualización del sistema es un proceso independiente del resto, cuyo comportamiento también se modela mediante la generación de eventos. Por tanto mantiene su QoS y puede adaptarse dinámicamente.

La tesis propone el cambio del esquema de simulación continuo a un esquema de simulación discreto, para comprobar si este nuevo esquema resuelve las deficiencias del esquema tradicional. La tesis no se centra en aspectos de aplicaciones gráficas como técnicas de visualización, inteligencia artificial, comportamiento de los personajes o detección de colisiones. Estos aspectos son los originales de la aplicación gráfica utilizada para probar el cambio de paradigma. Se centra en las metodologías de simulación.

1.2. Objetivos

El objetivo de la tesis es comprobar si el cambio de paradigma de simulación propuesto realmente mejora la simulación de aplicaciones gráficas en tiempo real. Para ello se propone la integración de un simulador de eventos discreto en una aplicación gráfica en tiempo real. Este nueva aplicación tendrá un comportamiento discreto.

La tesis comienza con un estudio del esquema de simulación de las aplicaciones gráficas en tiempo real. Se comprueba que siguen un esquema de simulación continuo y que presentan los problemas indicados anteriormente.

Para comprobar las hipótesis sobre el cambio de paradigma se debe cambiar el esquema de simulación de una aplicación gráfica en tiempo real. Se debe seleccionar una aplicación gráfica en tiempo real y un núcleo de simulación para integrarlo en la aplicación. La búsqueda de las aplicaciones a integrar produce los siguientes resultados:

- Como aplicación gráfica en tiempo real se elige el núcleo de videojuegos Fly3D. Esta aplicación tiene el código liberado, altamente estructurado y documentado. Es una aplicación monoprocesador representativa de la tecnología actual. Las aplicaciones creadas usando Fly3D está completamente separadas del núcleo, lo que permite crear diferentes tipos de aplicaciones para comprobar los resultados.

- Como núcleo de simulación se ha buscado entre los simuladores de eventos discretos existentes. Muchos de ellos no tienen el código liberado. Los que lo tiene liberado, o no tienen la potencia necesaria o está fuera de la corriente de la programación. Por ello se ha optado por crear un núcleo de simulación propio: DESK.

Inicialmente DESK se crea como simulador de eventos discreto independiente, con el objetivo de poder compararlo con otros simuladores a nivel de potencia, coste temporal de la simulación y flexibilidad. Se comparan las prestaciones de este simulador con los otros simuladores de eventos discretos representativos para comprobar que se han alcanzado los objetivos propuestos.

Posteriormente DESK sigue dos líneas paralelas:

- Simulación vía web: JDESK.

- Núcleo de simulación de aplicaciones gráficas en tiempo real: GDESK.

DESK se ha adaptado a la simulación vía web, creando JDESK. Este simulador sigue una línea propia de investigación adaptándose para la simulación vía web, de igual forma que han evolucionado otros simuladores. La ejecución vía web de un simulador supone una mayor accesibilidad y facilita el acceso de los usuarios desde cualquier navegador, lo cual es una ventaja respecto a la simulación tradicional.

Por otro lado, DESK se ha adaptado a un núcleo de simulación de aplicaciones gráficas en tiempo real: GDESK. GDESK sigue teniendo las mismas prestaciones de DESK, pero adaptadas a las necesidades de una aplicación gráfica. La tesis incluye un capítulo donde se muestran los fundamentos del simulador y la adaptación llevada a cabo.

Se integra GDESK en Fly3D para obtener DFLy3D, es decir Fly3D discreto desacoplado. Los procesos de Fly3D que no están directamente relacionados con el mecanismo de simulación del sistema se han mantenido como en el sistema original (como detección de colisiones, visualización o mapas de luces), para que los resultados obtenidos en la comparación del sistema discreto y continuo estén en las mismas condiciones. En los apéndices se muestra la integración detallada de algunos

objetos para que sirvan como guía para quien pudiese estar interesado en adaptar el simulador a otra aplicación gráfica.

Se ha comparado la aplicación original continua acoplada Fly3D con la aplicación discreta desacoplada DFLy3D. Se han realizado diferentes pruebas que han permitido constatar las hipótesis teóricas. Algunos de los parámetros medidos han sido: tiempos de simulación y ejecución, número de simulaciones, número de visualizaciones, otros criterios de QoS de los objetos o variación o adaptación de la QoS de los objetos.

Se muestran los resultados teóricos de este estudio que se consideran extrapolables a otras aplicaciones gráficas en tiempo real y los correspondientes resultados numéricos que apoyan los resultados teóricos.

1.3. Aportaciones

La tesis estudia un nuevo paradigma de simulación de aplicaciones gráficas en tiempo real, la simulación discreta desacoplada. Para la realización de la tesis se crea un simulador de eventos discretos, DESK [Garcia:1997] [Garcia:2000], que sirve como base al núcleo de simulación de aplicaciones gráficas en tiempo real. El simulador de eventos discreto se adapta para su ejecución en web: JDESK [Garcia:2003] [Garciab:2003]. Ambos simuladores son herramientas de simulación de sistemas de libre distribución.

DESK se adapta para servir como núcleo de simulación de aplicaciones gráficas en tiempo real. Se crea GDESK [Garcia:2002] [Garcia:2004]. GDESK, a pesar de ser un núcleo monolítico, no puede funcionar si no se integra en una aplicación gráfica. En esta tesis se ha integrado en el núcleo de aplicaciones gráficas Fly3D, creando DFLy3D [Garciab:2004] [Garciac:2004] [Garciad:2004] [Garciae:2004].

Esta tesis se ha desarrollado con el apoyo y financiación de dos proyectos de investigación:

- OCYT Generalitat Valenciana CTIDIB/2002/344.
- MCYT TIC2002-04166-C03-01.

Las publicaciones a las que ha dado lugar la tesis se muestran en el apartado 6.3 del capítulo 6. Cabe destacar cinco publicaciones en congresos internacionales (una de ellas para LNCS y otra para IEEE), dos publicaciones en revista internacional y dos congresos nacionales.

1.4. Contenidos

La tesis se estructura en los siguientes capítulos:

1.4.1. Estado del Arte

Este capítulo contiene los siguientes apartados:

1. Motores de videojuegos.
2. Visualización y simulación de aplicaciones gráficas en tiempo real.
3. Simulación de eventos discretos.
4. Conclusiones.
5. Crítica.
6. Propuesta de mejora.

La presente tesis se centra en el estudio de los videojuegos como ejemplo de aplicación gráfica en tiempo real. En este capítulo se exponen las razones por las que se han elegido estas aplicaciones gráficas en tiempo real.

El estudio previo de la tesis comienza con el estudio de motores de videojuegos para comprobar que el esquema de simulación utilizado es continuo y acoplado. Se analizan los problemas que este paradigma genera. El término motor de videojuegos incluye motores de visualización.

Posteriormente se estudia como otros sistemas han solucionado los problemas encontrados en los videojuegos. En concreto, las aplicaciones de realidad virtual resuelven el problema del acople de las fases de visualización y simulación, separándolas con el objetivo de distribuir o paralelizar ambas fases.

Para la creación del simulador de eventos discreto se estudian los simuladores de eventos discretos existentes y los fundamentos básicos de la simulación de eventos discretos, con el objetivo de crear un simulador de eventos que reúna las características necesarias para poder integrarse en una aplicación gráfica en tiempo real.

De estos tres estudios se extraen las conclusiones de las que parte el trabajo realizado en la tesis. En el apartado de crítica se exponen los problemas encontrados en el estudio previo: los videojuegos siguen un esquema de simulación continuo acoplado. La tesis propone comprobar las mejoras del cambio de paradigma de simulación. Para ello se elige Fly3D como motor de videojuegos para realizar la integración. Se decide también crear un simulador de eventos discretos para integrarlo en Fly3D, DESK. El resultado de la integración de DESK en Fly3D es DFly3D. DFly3D es un núcleo de videojuegos discreto desacoplado.

1.4.2. Núcleo de Simulación de Eventos Discretos

En este capítulo se describe el simulador de eventos discreto creado: DESK. Se define su estructura interna y la forma en que permite modelar y simular un sistema. Posteriormente DESK se especializa para núcleo de aplicaciones gráficas en tiempo real, creando GDESK.

Como un trabajo paralelo a la creación de GDESK, se crea JDESK. JDESK es el simulador de eventos discreto DESK adaptado para su ejecución en web. Se muestra la importancia de la simulación en web para procesos de e-learning.

En este apartado se definen los aspectos básicos de GDESK y su arquitectura.

1.4.3. DFly3D: Fly3D Discreto

En este capítulo se muestra la integración de GDESK en FLy3D, obteniendo DFly3D. La integración de ambos sistemas supone realizar tanto variaciones en el núcleo de Fly3D como en las aplicaciones creadas a partir de este.

Se define la nueva estructura de Dfly3D, las estructuras de datos utilizadas para la nueva gestión de eventos (mensajes y objetos) y la nueva forma de simular los objetos del sistema y de los videojuegos creados. Se hace especial mención del mecanismo de paso de mensajes que utilizan los objetos para comunicarse y modelar su comportamiento. Entre los elementos añadidos a Fly3D cabe destacar el monitor del sistema y los mecanismos de adaptación dinámica del sistema.

1.4.4. Resultados

Los resultados mostrados en este capítulo son:

- Resultados del simulador de eventos discretos:
  - Comparativa de los resultados de DESK con el simulador de eventos discretos SMPL. Se comparan los resultados temporales de la simulación de diferentes modelos de sistemas representativos.
  - Comparativa de los resultados temporales de DESK con JDESK.
- Resultados de los modelos continuo acoplado y discreto desacoplado:
  - Comparativa teórica de los modelos continuo y discreto. Estos resultados son comunes a la mayoría de aplicaciones gráficas en tiempo real.
  - Comparativa de la aplicación continua acoplada Fly3D con la aplicación discreta desacoplada DFLy3D. Se muestran resultados que permiten

corroborar lo expuesto en el apartado anterior sobre aplicaciones concretas y con resultados numéricos. Se comprueba que el paradigma de simulación discreto desacoplado ha cumplido los objetivos propuestos.

1.4.5. Conclusiones y Trabajos Futuros

Por último, en este capítulo se exponen las conclusiones finales de la tesis, obtenidas de la creación de de un simulador de eventos discreto (DESK), la adaptación a aplicaciones gráficas en tiempo real (GDESK), de la integración en una aplicación gráfica (Fly3D) para obtener una aplicación gráfica en tiempo real discreta (DFLy3D).

En este apartado se muestran cuales podrían ser las líneas futuras de investigación y las publicaciones a las que ha dado lugar la tesis.

Se obtiene resultados numéricos de la comparación de Fly3D y DFly3D. Estos resultados permiten comprobar la certeza de los resultados teóricos. Los resultados teóricos son extrapolables a la mayoría de aplicaciones gráficas en tiempo real.

1.4.6. Apéndices

La tesis contiene los siguientes apéndices:

1. Ejemplos de modelado de sistemas que varían dinámicamente con DESK.
2. Manual de usuario de JDESK.
3. Algoritmos ejemplo de DESK, JDESK y SMPL (se incluyen los algoritmos de SMPL porque son ejemplos de los modelos utilizados para realizar la comparativa con DESK y porque sirven para comprobar la facilidad de implementación del modelo en DESK).
4. Estudio del motor de videojuegos Fly3D.
5. Ejemplos de la integración de GDESK y Fly3D. Se muestra como se han integrado dos partes representativas de DFLy3D: el objeto *consola* y el objeto del videojuego *bola*.

Capítulo 2

Estado del Arte

Esta tesis estudia el núcleo de simulación de las aplicaciones gráficas en tiempo real, otros aspectos de estas aplicaciones como visualización o inteligencia artificial no se consideran. Para ello se analiza el paradigma de simulación de estas aplicaciones. El objetivo es mejorar la calidad y la eficiencia de la simulación de las aplicaciones gráficas en tiempo real.

Entre la diversidad de aplicaciones gráficas en tiempo real existente se ha elegido estudiar el campo de los videojuegos por varias razones:

- Existe una gran diversidad de videojuegos con código abierto: multitud de juegos realizados por aficionados y videojuegos comerciales con código liberado (entre ellos destacan *Doom* y *Quake*).
- Existe una amplia comunidad de programadores interesada en ellos [DoomW], [Gameprogrammer], [Gamedev], [Gametutorials], [Gamasutra], [Gdmag], [Gdconf], [Cgri],...
- Se ejecutan habitualmente en un único procesador, pues son aplicaciones destinadas al mercado doméstico.

Las conclusiones obtenidas para videojuegos son extrapolables a otras aplicaciones gráficas en tiempo real.

Durante la realización de la tesis se han estudiado:

1. El código de gran cantidad de videojuegos: el estudio se ha centrado en la gestión de eventos, para estudiar los mecanismos de simulación utilizados.
    - Videojuegos sencillos: tanto antiguos como modernos. Corresponden a juegos no comerciales realizados de forma altruista por entusiastas. Estos videojuegos adolecen de estructuración interna y emplean técnicas

de simulación muy rudimentarias. En internet pueden encontrarse multitud de bibliotecas de juegos no-comerciales [Cdx], [Cfx], [Idevgames], [Sourceforge],...

- Videojuegos complejos: habitualmente videojuegos comerciales.

2. Núcleos de visualización: como *Cristal Space*, *OpenGL Performer*, *Artist* o *Open Scene Graph*.

2.1. Motores de Videojuegos

2.1.1. Introducción Histórica

Ralph Baer creó en 1951 el primer videojuego [Baer:1999]. Se creó para jugar en televisores como demostración de su potencial. El primer videojuego para ordenador se creó en 1958. William Higinbotham (BNL) creó una máquina con una versión electrónica de un juego parecido al tenis, al que denominó *Tennis for two* [PongStory]. BNL es un laboratorio de investigación nuclear del gobierno de EEUU. Para demostrar que el laboratorio era seguro se realizaron visitas guiadas al laboratorio. Higinbotham decidió hacer algo diferente para la exposición, demostrando lo que sus equipos eran capaces de hacer, creando un videojuego. Para su creación utilizó un computador analógico y un osciloscopio.

La evolución de los videojuegos está íntimamente ligada a la evolución del hardware de las computadoras. Los juegos se desarrollan para tres plataformas diferentes: arcade, consola y ordenador personal. Los arcade y las consolas son máquinas especializadas para videojuegos, por lo que, hasta que el ordenador personal estuvo lo suficientemente preparado (a mediados de los 80), los videojuegos se desarrollaron fundamentalmente para estas plataformas.

2.1.1.1. Arcades

Los mainframes eran máquinas demasiado caras para utilizarlas comercialmente, por lo que se crean máquinas cuya utilidad es únicamente jugar. Estas máquina se denominan arcades por el nombre de la primera compañía que diseñó un videojuego de lucha callejera. La evolución de los videojuegos para arcade es meteórica, el hecho de disponer de un hardware específico de videojuegos y un software adaptado al hardware hace que estos puedan evolucionar rápidamente. En la tabla 2.1 se muestra la evolución de los juegos de arcade.

Actualmente los arcades de mayor éxito son los de lucha callejera, aunque existen todo tipo de temáticas. Cada vez son de mayor espectacularidad y ofrecen más posibilidades al jugador.

Año	**Videojuego**	**Hito**
1971	*Computer Space*	Primer juego de arcade (problemas)
1974	*Tank*	Primer juego en usar un chip de ROM
1975	*Pong*	Primer juego de arcade completo
	Indy 800	Primer juego en color
	Gunfight	Primer juego en usar un microprocesador
1976	*Night Driver*	Primer juego de carreras con perspectiva en primera persona
1977	*Space Wars*	Primer juego con gráficos vectoriales
1978	*Space Invaders*	Capaz de almacenar los marcadores de los jugadores
1979	*Asteroids*	Primer juego que almacena las iniciales de los jugadores junto con los marcadores
1980	*Battlezone*	Primer entorno tridimensional
	Berzerk	Incluye un sintetizador de voz de 30 palabras
	Defender	Primer juego en incluir un mundo artificial
	Pac Man	Uno de los juegos más famosos de la historia
1981	*Tempes*	Primer juego de gráficos vectoriales en color
	Donkey Kong	Clásico
	Galaga	Clásico
1983	*Dragon's Lair*	Primer videojuego en utilizar la tecnología Laser Disc, con formato de película interactiva.
1986	*Out Run*	Primer juego en utilizar Scaling y uno de los primeros juegos de 16 bits
1987	*Double Dragon*	Primer juego de lucha callejera
	Street Fighter	Juego de lucha con un estilo diferente
1989	*Final Fight*	Marca las pautas de los juegos de lucha callejera
1991	*Street Fighter II*	Permite seleccionar entre 8 luchadores
1992	*Virtual Racing*	Utiliza polígonos rellenos
	Mortal Kombat	Gráficos de los luchadores digitalizados
...	*SNK Millennium Fight*	Juegos actuales [Adventure]
	The King of Fighters	
	Metal Slug 4	
	Virtua Cop 3	
	Dragon Treasure	
	F-Zero AC	

Tabla 2.1: Historia de los videojuegos tipo arcade [DotEaters] [CStory] [Flyer]

2.1.1.2. Sistemas Domésticos

En 1971 apareció *Odyssey*, el primer sistema de videojuegos de televisión casero comercial. Pero no fue hasta 1977 cuando apareció la primera consola (Atari 2600). La consola fue durante mucho tiempo la reina de los sistemas caseros de videojuegos. Durante los 80 los ordenadores personales (Commodore, CPC Amstrad, Spectrum y Amiga) fueron ganando terreno a las consolas, pero los ordenadores no estaban al alcance de cualquier bolsillo.

En la segunda mitad de los años 80 los ordenadores compatibles PC empezaban a instalarse en los hogares, si bien se reservaban a un público iniciado en el mundo de la informática, con escaso interés en los videojuegos. Una de las razones para su escasa difusión era el elevado precio de estas máquinas y así como su escasa potencia de cálculo. En esta época, los videojuegos para este tipo de ordenador se limitaban a las reproducciones de algunos juegos de mesa, en especial el ajedrez. Progresivamente, se efectuaron las primeras conversiones de los juegos clásicos de consola o arcade. Poco a poco, aumentó el número de títulos disponibles en formato PC, si bien las innovaciones técnicas resultaron escasas. El número de juegos disponible para ordenador personal aumentó en la medida en que descendían los precios de los primeros compatibles con el IBM-PC.

Con la instauración del estándar AT (procesador 80286) a finales de los 80, se inició un salto cualitativo en la producción de videojuegos. A partir de este momento, el videojuego para PC comenzó su despegue en una carrera frenética. En los 90 los ordenadores personales todavía no podían competir con las consolas, por lo que los videojuegos para PC iban dirigidos a un público adulto. Fue la era dorada de las aventuras conversacionales, la niñez de las aventuras gráficas.

Los juegos para PC eran cada vez más complejos, lo que requería espacio para almacenarlos. La introducción del CD-ROM vino a aliviar algo este problema, ya que no se tenía que instalar todo el juego en disco. El siguiente gran paso se produjo con la aparición de los procesadores i80386 y i80486. Los videojuegos del género de las aventuras gráficas adquirieron un papel preponderante.

La revolución llegó al mundo de los videojuegos para ordenador personal con *Doom* (John Carmack, Id Software). Crea un estilo copiado hasta la saciedad en años posteriores. Es un juego 2D, aunque el uso de perspectivas y sprites hacen creer al jugador que se encuentra en un entorno 3D. Su calidad es muy superior a otros juegos de la época. Podía presentar diferentes niveles, aunque no superpuestos y contaba con capacidades multijugador (hasta 8 jugadores en red local). Id Software hace públicas las especificaciones para realizar nuevos niveles para el juego y en poco tiempo surge en Internet un colectivo de programadores y diseñadores aficionados que inundan la red de nuevos niveles y modificaciones del juego original. Fue uno de los baluartes de Internet, en una época en que era casi terreno privado de los

Año	**Videojuego**	**Hito**
1961	*Space War*	Primer juegos con gráficos vectoriales (mainframe)
1977	*Akalabeth*	Programado en BASIC (Apple II)
1979	*MUD (Multi-User Dungeon)*	Primer juego multi-usuario (mainframe). Chat rudimentario (número limitado de comandos)
	Temple of Apshai	Primer juego de rol
1980	*Mystery House*	Primer juego con gráficos (Apple II) y primera aventura gráfica
	Ultima	Primer juego de rol
1982	*Invasion Orion*	Juego para Comodore PC C-64
1983	*King's Quest*	Encargado por IBM para demostrar la capacidad gráfica de sus PC
1986	*The Two Guys from Andromeda*	Clásico
	Space Quest	Clásico
1987	*Leisure Suit Larry*	Introdujo al público adulto en el mundo de los videojuegos (introduce la tecla "jefe")
1988	*Tetris*	Versiones en consolas, arcades y computadoras. Incluso hoy en día siguen saliendo nuevas versiones.
1990	*Monkey Island*	Edad dorada de las aventuras gráficas
	Indiana Jones	Usan ratón y teclado para seleccionar los diálogos
	Maniac Mansión	
	Broken Sword	
	Simon the Sorcerer	
	Dune	Juegos de estrategia
	Centurion 50 A.D.	
1992	*Street Fighter*	Los juegos de lucha para arcade amplían sus plataformas
	Wolfenstein 3D	Primer juego de acción en primera persona
1993	*Doom*	Juego de acción 2D que simula 3D
	Quake	Primer juego con entorno real 3D
1995	*Warcraft*	Estrategia en tiempo real
1997	*Ultima Online*	Mayor juego en red hasta el momento
	Quake 2	Soporte de fábrica de aceleración 3D. Multijugador
1998	*Unreal*	Trata de emular a *Quake 2*
2000	*Euro Fighter 2000*	Simulador de vuelo
	X-Wing	
...	*All-Star Baseball 2004*	Juegos recientes [Gamezone]
	The Lost Age	
	Pro Bass Fishing 2003	
	PC Casino Inc.	
	James Bond 007: NightFire	

Tabla 2.2: Historia de los videojuegos domésticos [Videogames] [Videopatia]

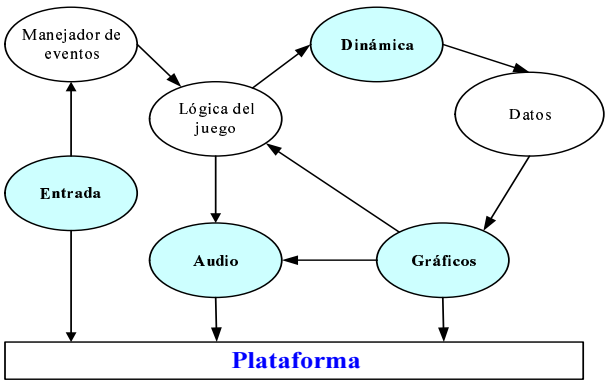

Figura 2.1: Núcleo de videojuegos [Bishop:1998]

militares y ciertas universidades. Fue el primer motor usado por otras compañías para realizar otros títulos.

Quake (Id Software) vuelve a revolucionar el mundo de los videojuegos por ser el primer motor 3D y su versión *Quake 2* por permitir hasta 200 jugadores en red.

En la actualidad el ordenador personal gana terreno a las consolas, pues su uso se ha generalizado, pero las consolas siguen teniendo gran impacto, sobre todo en un público juvenil. Los videojuegos son sistemas en continuo cambio, debido a que los requerimientos del usuario son cada vez mayores, lo que exige cambios en el hardware y por ello en el software [Bishop:1998]. El futuro del videojuego depende directamente del avance tecnológico.

La tabla 2.2 muestra los hitos más importantes en los videojuegos para sistemas caseros.

2.1.2. Estado Actual

Los videojuegos inicialmente se creaban escribiendo el código completo del videojuego. Pero, la complejidad de los videojuegos ha crecido tanto que ya no es posible utilizar esta técnica y se han creado núcleos de videojuegos de código modular [Lewis:2002]. Habitualmente los núcleos de videojuegos se diseñaban para un juego específico, pero se hacían tan generales que podían utilizarse para videojuegos de la misma familia. Un núcleo de videojuego debe contener lo elementos mostrados en la figura 2.1.

Actualmente, por núcleo de videojuego se entiende el conjunto de módulos que componen el videojuego exceptuando los que no especifican directamente el comportamiento del juego (lógica) o el entorno (datos) [Lewis:2002].

El término motor de videojuegos aúna tres tipos de código diferente:

- **Motores de visualización**: rutinas gráficas que permiten crear la parte gráfica del videojuego de una forma rápida y estandarizada.
- **Motores de videojuegos**.
- **Videojuegos completos**: el código de un videojuego se modifica para crear otro juego diferente cambiando su comportamiento. Los videojuegos comerciales suelen contener editores de personajes y escenarios y ficheros de definición de estos, de forma que crear un juego a partir de otro es relativamente sencillo.

2.1.2.1. Motores de Visualización

3D GameStudio

3D GameStudio [Conitec] es un kit de desarrollo comercial de juegos de ordenador. Consta de un motor 3D, un motor 2D, un editor de niveles y modelos, un compilador de scripts y librerías de modelos, texturas,... Manipula con igual rendimiento escenas de interior y de exterior. Tiene un motor de iluminación que soporta sombras verdaderas y fuentes de luz en movimiento. El principal objetivo que esta aplicación persigue es que el desarrollador del juego no necesite ser un programador experimentado. Se reduce al máximo el esfuerzo de desarrollo a costa de perder flexibilidad en el diseño del juego. La última versión (v5) es de marzo de 2002. Ofrece tres posibilidades para crear un juego:

1. Juegos diseñados a base únicamente de controles, para usuarios con pocos conocimientos de programación.
2. Juegos o efectos diseñados con algo de programación utilizando C-scripts.
3. Juegos o efectos programados en C++ o Delphi, para programadores con experiencia.

Incorpora un núcleo de videojuegos, llamado A5. Pero lo que aquí se entiende por núcleo es un sistema de desarrollo que se encarga de generar efectos 3D y controlar la inteligencia artificial del juego.

Crystal Space

Cristal Space [CrystalSpace] es un kit de desarrollo gratuito de juegos 3D libre y portable escrito en C++. Soporta seis grados de libertad, luces de colores, mipmapping, portales, espejos, transparencias, superficies reflectivas, sprites 3D (basados en frames o animaciones de esqueleto), texturas procedurales, radiosidad, sistemas de partículas, halos, niebla volumétrica, lenguaje de script (Python y otros), soporte para visualización a 8-bits, 16-bits y 32-bits, Direct3D, OpenGL, Glide, y visualización por software, soporte para fuentes, transformaciones jerárquicas,... Actualmente

Crystal Space puede ejecutarse sobre GNU/Linux, Windows, Windows NT, OS/2, BeOS, NextStep, OpenStep, MacOS/X Server, DOS, y Macintosh entre otros. *Crystal Space* es un gran proyecto para el desarrollo de software abierto. Hay alrededor de 600 personas subscritas a sus listas de correo.

Es un paquete orientado a eventos. La cola de eventos gestiona los eventos del sistema y envía eventos a quien proceda. El gestor de eventos se encarga de lanzar la visualización (evento *cscmdProcess*) y gestiona los eventos de usuario. Aunque existe cierto paso de mensajes en la aplicación, se limita a mensajes de visualización y eventos de usuario.

Genesis3D

Genesis3D [Genesis3D] es un motor de código libre para la visualización de escenas tridimensionales en tiempo real y que permite construir aplicaciones gráficas 3D de altas prestaciones. Ha sido diseñado principalmente para la visualización de escenas de interior logrando una elevada tasa de fotogramas por segundo siempre y cuando estén compuestas por una cantidad moderada de polígonos. También puede ser utilizado para escenas de exterior si el diseño de las escenas se realiza tomando ciertas precauciones. Sus principales características son la detección rápida de colisiones, iluminación precalculada y chequeo de la visibilidad. Su principal inconveniente es la visualización de escenarios exteriores sin imponer algún tipo de restricción o límite al tamaño de la escena. La versión actual es la v1.1 (noviembre de 1999), se va enriqueciendo mediante las contribuciones o comentarios que realizan los usuarios.

No incluye rutinas de gestión de eventos.

The Nebula Device

The Nebula Device [RadonLabs] es un nuevo motor de juegos gratuito de calidad profesional. Sus creadores, son el equipo que desarrolló *Urban Assault* (publicado por Microsoft en 1998). Utilizado también para desarrollar *The Nomads* (Xbox). *Nebula* es un motor de arquitectura moderna. Desarrollado en C++ y orientado a objetos, sus clases (DLLs) se cargan de forma independiente en tiempo de ejecución. El motor puede ejecutarse en Linux, Windows 9X, Windows NT. Permite intercambiar sin interrupción la visualización con OpenGL y Direct3D. *Nebula* utiliza como lenguaje de script el estándar tcl/tk. Los principales objetivos de *Nebula* son los siguientes:

- Independencia de la plataforma.
- Gestión de la base de datos del juego: jerarquías 3D, texturas, materiales, luces, sonidos, animaciones, estados y sus relaciones.
- Proporcionar herramientas básicas de trabajo en equipo para el desarrollo del juego.

El servidor de entrada gestiona los eventos de entrada de usuario utilizando una lista de eventos. Se utiliza un mecanismo de paso de mensajes para el tratamiento de los jugadores en red.

OGRE

OGRE (Object-Oriented Graphics Rendering Engine) [Ogre] es un motor escrito en C++ flexible, orientado a escenas, y diseñado para hacer más simple e intuitiva a los desarrolladores la producción de juegos utilizando hardware de aceleración 3D. La librería de clases permite abstraer los detalles asociados a las librerías de bajo nivel (OpenGL o Direct3D), proporcionando una interfaz basada en objetos.

Shark3D

Shark3D [Shark3D] es un kit de desarrollo de alto nivel para aplicaciones 3D en tiempo real. Está orientado a juegos, aplicaciones de realidad virtual y visualización. Optimizado para aplicaciones multiusuario a través de una red de computadores o internet. Distribuido. Contiene un servidor de múltiples escenas.

Incluye gestión de eventos de usuario y paso de mensajes para comunicación en red y como comunicación entre los diferentes elementos de su arquitectura.

The Torque Game Engine

The Torque Game Engine (*TGE*) [Garagegames] es el motor comercial desarrollado por DINAMIX para su juego *Tribes 2*. Enfocado a la simulación de misiones militares, incluye utilidades para la creación de terrenos, superficies acuáticas, interiores estilo portal y sistemas de partículas. También incluye soporte multiplataforma (Windows, Mac OS y Linux), soporte para red, creación de interfaces de usuario y lenguaje de script estilo C++. Permite importar objetos desde 3D Studio MAX y dispone de librerías matemáticas, de detección de colisiones, de física de vehículos y una base de datos espacial.

Incluye gestión de eventos de usuario y paso de mensajes para comunicación en red.

CDX Game Development Kit

CDX Game Development Kit [Cdx] es un kit de desarrollo de videojuegos de código libre. Consta de un conjunto de clases de C++ para crear juegos para Windows y utilizando DirectX. Permite la creación de juegos sencillos.

Contiene rutinas de gestión de los eventos de entrada.

Artist

Artist (Animation Package for Real-Time Simulation) [Artist] es un paquete de animación de bajo coste para el desarrollo de aplicaciones 3D complejas e interactivas

en tiempo real: juegos, simulación y realidad virtual.

Artist permite generar gráficos con velocidad optimizada y aspectos de comportamiento de los objetos en un juego, simulaciones o aplicaciones de realidad virtual. Consigue enlazar ambos aspectos de forma eficiente y sencilla, generando el código necesario para ejecutar la simulación del comportamiento. Puede generar bases de datos gráficas optimizadas para mejorar la velocidad de visualización que permite maximizar el uso del hardware gráfico disponible. Estas estructuras jerárquicas permitirán mantener una frecuencia de cuadro uniforme, sobre los 30 fps, consiguiendo la continuidad visual necesaria en el juego.

Incluye las siguientes herramientas:

- Modelador de objetos geométricos de propósito general.
- Base de datos jerárquica gráfica.
- Conversores de fichero desde los formatos gráficos usuales.
- Interfaz 3D para añadir objetos interactivos en las escenas de los juegos.
- Editor de jerarquía gráfica para la configuración de los parámetros de la gestión de la base de datos y la incorporación de descripciones de comportamiento a los objetos geométricos de la base de datos.
- Descripción de alto nivel orientada a objeto del comportamiento de los objetos y las características del juego.

OpenGL Performer

OpenGL Performer [Sgi] [Sgiwp] es una herramienta de programación comercial de aplicaciones 3D y simulación en tiempo real orientada a objetos. Simplifica el desarrollo de aplicaciones complejas realizadas para simulación visual, realidad virtual, entretenimiento interactivo o diseñado asistido por ordenador. Mejora el rendimiento del sistema, permitiendo un uso óptimo de sus capacidades.

- Construido a partir de la librería gráfica *OpenGL*.
- Puede compilarse en C y C++ estándar.
- Disponible para Windows y Linux.
- Biblioteca de visualización *libpr*. Es una librería de bajo nivel con funciones de visualización de gran velocidad.
- Entorno de simulación visual en tiempo real *libpf*, que permite multiproceso de altas prestaciones del grafo de escena y del sistema de visualización.

- Biblioteca de definición de geometría y apariencia de objetos tridimensionales *libpfdu*.
- Biblioteca para importar ficheros de bases de datos *libpfdb*.
- Biblioteca para crear interfaces de usuario *libpfui*.
- Biblioteca de utilidades *libpfutil* (configuración multiproceso, soporte a multicanal, texturas, herramientas de interfaz, obtención y gestión de eventos de entrada).

Una aplicación desarrollada con *OpenGL Performer* puede ejecutarse en computadores con varios procesadores. *OpenGL Performer* maneja los pipelines para generar una imagen. Permite fácilmente escalar a múltiples procesadores y múltiples pipes gráficos.

La arquitectura de *OpenGL Performer* consta de las siguientes partes:

1. *APP*: proceso de simulación. Incluye lectura de la entrada de los dispositivos de control, simulación de modelos dinámicos, actualización de la base de datos visual e interacción con otras bibliotecas o estaciones de simulación.
2. *CULL*: recorre la base de datos visual y determina que porción de la escena es visible (culling), selecciona el nivel de detalle (LOD) de cada modelo, clasifica los objetos y optimiza su gestión y genera una lista de objetos a visualizar.
3. *DRAW*: recorre la lista de los objetos a visualizar y emite los comandos al pipe de geometría para crear la imagen para el dispositivo de salida.

El usuario tiene control total sobre estas tareas, incluyendo la posibilidad de combinar múltiples tareas en un único proceso o dividirlas entre varios procesos o procesadores. Incluso *OpenGL Performer* puede automatizar la división en procesos dinámicamente.

OpenGL Performer permite garantizar una frecuencia de cuadro fija [Perguide], incluyendo funciones para fijar esta frecuencia y gestionar problemas como tiempos de visualización mayores que el periodo de refresco.

El bucle de simulación, incluido en el proceso APP, repite indefinidamente las siguientes etapas [Pergsguide]:

1. Actualiza la escena.
2. Actualiza la cámara.
3. Dibuja la escena.

OpenGL Performer desacopla las fases de simulación y visualización. La simulación sigue realizándose muestreando los objetos.

Open Scene Graph

Open Scene Graph [Osg] es un toolkit gráfico de alto nivel y portable para el desarrollo de aplicaciones gráficas de alto rendimiento tales como simuladores de vuelo, juegos, realidad virtual o visualización científica. Está orientado a objetos y construido a partir de la librería gráfica *OpenGL*, esto libera al desarrollador de implementar y optimizar llamadas gráficas de bajo nivel, y provee muchas utilidades adicionales para un rápido desarrollo de aplicaciones gráficas.

El corazón del grafo de escena ha sido diseñado para tener mínimas dependencias de una plataforma específica, requiriendo poco más que C++ estándar y *OpenGL*. Esto ha permitido al grafo de escena ser rápidamente portado a un gran número de plataformas (originalmente desarrollado en IRIX, portado a Linux, Windows, FreeBSD, Mac OSX, Solares, HP-UX e incluso PlayStation2).

Todo el código de *Open Scene Graph* esta publicado bajo la Open Scene Graph Public License (permite a proyectos de código abierto y cerrado utilizarla, modificarla y distribuirla libremente). *Open Scene Graph* soporta view frustum culling, occlusion culling, small feature culling, nodos con nivel de detalle (LOD), clasificación de estado, vertex arrays y listas de dibujado como parte del corazón del grafo de escena.

Open Scene Graph es uno de los grafos de escena disponibles de mayor rendimiento. Este rendimiento iguala a otros grafos de escena como *OpenGL Performer* o *Vega Scene Graph*. *Open Scene Graph* opta por soluciones muy parecidas a *OpenGL Performer*. Por contra, no soporta multiproceso, característica que soporta *OpenGL Performer*.

2.1.2.2. Gestión de Eventos en Motores de Visualización

Algunos de estos motores incluyen gestión de eventos de usuario, proporcionando rutinas para gestionar alguna estructura de datos donde se inserten estos eventos. Pero los eventos se limitan a eventos de usuario. *Crystal Space* incluye un evento especial de visualización.

Hay motores que implementan paso de mensajes, pero como forma de comunicación en red o como parte de su arquitectura.

En [Lucia:2003] se muestra una comparativa de algunos de los motores gráficos y videojuegos incluidos en este estudio.

2.1.2.3. Motores de Videojuegos

Fly3D

Fly3D [Fly3D] es un motor de juegos gratuito que acompaña al libro *3D Games* de Alan Watt y Fabio Policarpo [Watt:2001]. El motor se encuentra actualmente en la segunda versión v2.0 (disponible en el segundo volumen del libro [Watt:2003]). Es un núcleo de videojuegos de reciente creación, altamente estructurado y comentado y orientado a objetos. Ambas versiones están desarrolladas en C++ y todo el código específico del juego se encuentra desarrollado con DLLs plugins (se incluye una herramienta para crear plugins). Contiene una amplia gama de herramientas y utilidades que facilitan el proceso de desarrollo del juego.

El motor permite crear videojuegos y aplicaciones gráficas en tiempo real de forma fácil y sencilla. Para ello se crean nuevos plugins que se enlazan con la aplicación, sin necesidad de recompilar el núcleo y permitiendo un desarrollo modular. Se pueden implementar múltiples aplicaciones sin necesidad de replicar el núcleo.

La implementación del comportamiento visual y dinámico está también modularizada, debiendo implementar para cada objeto una función específica de visualización y otra de comportamiento (simulación). Para conseguir que los objetos del juego se comporten de forma uniforme, heredan de un objeto base de *Fly3D*. El objeto base contiene una serie de funciones virtuales que los objetos del juego deben redefinir.

Es un motor de código libre y los videojuegos creados usando *Fly3D v.1* pueden incluso comercializarse.

Es representativo de la tecnología actual y, por lo tanto, va a ser utilizado como marco de desarrollo en el que verificar y probar la presente tesis.

2.1.2.4. Videojuegos Completos

La información existente sobre videojuegos es muy abundante en la red. Gran parte de esta información hace referencia a trucos de juego, manuales, modificaciones de código, creación de ficheros de configuración, evaluaciones, módulos de desarrollo concretos e incluso, a veces, el código fuente. No obstante, la información sobre la implementación de videojuegos es mínima, en algunos casos se limita a algún fichero cabecera, algún API o cierta información sobre las primitivas utilizadas. En cualquier caso, esta información, además de ser escasa, se suele limitar a la parte gráfica del juego. Sólo en contadas ocasiones se dispone del código del juego para poder estudiar como se gestionan los eventos del sistema.

La mayor parte de la investigación en el campo de los videojuegos se lleva a cabo en las compañías desarrolladoras de juegos. La investigación que se ha llevado

a cabo en las universidades y centros de investigación se centra en aspectos como visualización o en la aplicación de técnicas de inteligencia artificial para la gestión del comportamiento de los personajes, siendo el modelo de simulación algo implícito, a lo que no se ha prestado mucha importancia.

Existen numerosos juegos con código abierto. La mayor parte de ellos corresponde a pequeños juegos creados por aficionados y que no pueden considerarse como motores de videojuegos. Muy pocos son los videojuegos complejos (habitualmente comerciales) con código publicado. Dentro de estos se han elegido como base del estudio los videojuegos *Doom* y *Quake* por las siguientes razones:

- Son juegos de código libre. Ciertas versiones de *Doom* y *Quake* tienen el código liberado, disponible para su redistribución y modificación bajo licencia GNU [Gnu].
- Su código está completamente liberado. En otros videojuegos de código libre, ciertas partes del código son bibliotecas de código cerrado, pueden usarse pero no modificarse (como *Serious Sam* [Croteam] o *Unreal Tournament* [Epic]).
- *Doom* y *Quake* marcaron un hito en la historia de los videojuegos y son ampliamente conocidos y utilizados.
- Despiertan interés de una amplia comunidad de programadores.
- Son juegos robustos y ampliamente probados.
- Siguen el paradigma convencional de videojuegos.
- El primer videojuego importante cuyo motor se utilizó para la creación de otros videojuegos fue *Doom*. El autor de *Doom* mejoró e incorporó características 3D a este motor, creando *Quake*, que a su vez sirvió también para la creación de nuevos videojuegos.

Además, se ha incluido en el estudio el videojuego *Unreal Tournament*, porque a pesar de no tener el código liberado, despierta un gran interés en la comunidad científica.

Doom

Doom es un juego de laberintos en primera persona creado por John Carmack en 1993. Es un juego 2D, aunque simula cierta profundidad. La única versión del juego liberada es la v1.1 para Linux [Idsw].

Utiliza como entidades básicas las estructuras *things*, que definen elementos del juego como monstruos, armas, llaves o posiciones de inicio de un jugador. El tipo de datos *thing* incluye únicamente información sobre la parte física de la entidad,

como posición de la entidad o tipo. Las entidades no incluyen información sobre los eventos del sistema.

Para gestionar los eventos se utiliza una lista doblemente enlazada con un único puntero a cabeza. Cada evento contiene tres funciones que definen las acciones a realizar cuando suceda el evento. Los eventos se ejecutan comenzando por la cabeza de la lista hasta alcanzar la cola, ejecutando para cada evento las acciones asociadas. Los eventos no tienen tiempos asociados, se ejecutan todos y en el orden en que aparecen en la lista. La lista de eventos se recorre desde la cabeza hasta la cola. Todos los eventos son evolucionados obligatoriamente a la máxima velocidad que el sistema pueda obtener. El sistema analiza desde el último estado, cual es el nuevo estado en el que se debe encontrar el sistema en función del tiempo transcurrido. Este esquema de simulación obliga a que un objeto nunca pueda programar eventos situados a varios intervalos de muestreo. Por ejemplo, en *Doom* no existe la bomba de relojería. Para que una granada estalle, ésta debe ser disparada y estallará cuando colisione con el suelo, una pared u otro personaje.

La gestión de eventos discretos debe realizarse de forma explícita por el programador al margen del núcleo del videojuego y cada objeto debe gestionarla de forma independiente.

La ordenación de los eventos en la lista no es cronológica sino que dependerá del orden en que se gestione el sistema. La inserción de nuevos eventos se realiza en cabeza de la lista, por ello, los nuevos eventos se tratarán siempre en el siguiente ciclo, independientemente de que algún evento corresponda al ciclo actual. Cualquier evento que se produzca con una frecuencia superior a la frecuencia de muestreo, es sencillamente obviado.

Quake

Quake [Abrash] es un juego de arcade de laberintos en primera persona creado por John Carmack en 1996. Es el primer juego realmente 3D. La versión objeto de este estudio ha sido la v2.3 [Quake].

El elemento básico de *Quake* para la gestión del comportamiento de los personajes y objetos es la entidad. Las entidades son partes independientes del entorno virtual, como monstruos, jugadores, objetos o posiciones en el espacio. Una entidad especial es *world*, que define el entorno estático por el que se mueven los personajes. En la entidad confluyen una serie de propiedades físicas y otras propiedades que determinan sus reglas de comportamiento. Al comportamiento de las entidades se le denomina dentro del juego inteligencia artificial, aunque no utilicen técnicas propiamente de inteligencia artificial. La inteligencia artificial utiliza tres propiedades de la entidad para gestionar los eventos:

- Tiempo en el que debe suceder el siguiente evento relacionado con la entidad,

momento en el que cambiará de algún modo su comportamiento.

- Función que define el comportamiento justo antes de modificar físicamente la entidad (como moverla o hacer que dispare).

- Función que define el comportamiento de la entidad después de modificar su parte física.

La descripción de la escena está constituida por un vector de entidades. El vector se recorre en orden, comenzando por la entidad *world* (primera posición del vector), y se comprueba si cada una de las entidades tiene que modificar su comportamiento en el instante actual, comprobando si el tiempo asociado al evento vencerá en el próximo intervalo de tiempo. Este intervalo lo predefine el programador.

Unreal Tournament

Unreal Tournament [Gerstmann:1999] [Unrt] es un motor gráfico comercial desarrollado por la empresa *Epic Games* [Epic], altamente modular y orientado a objetos. Es multijugador, basado en arquitectura cliente/servidor. Nace en 1998 tratando de emular a *Quake2*.

Disponible para las plataformas Linux, Windows, Macintosh, Playstation 2 y Xbox. Para desarrollar con este motor se debe adquirir una licencia que da acceso a todo el código fuente, a las herramientas y juegos. Cierto código y las librerías del núcleo están disponibles gratuitamente. Para trabajos y desarrollos que hagan uso del motor gráfico sin tratar modificar el núcleo (como son los desarrollos y proyectos de inteligencia artificial e interfaces) el motor es una herramienta posible, por lo que se utiliza ampliamente en investigación (Gamebots [Gamebots] [Kaminka:2002] o CaveUT [Caveut] [Jacobson:2002]).

Utiliza el sistema DSG (Dynamic Scene Graph Technology) que es una extensión natural del renderizado en portales, interpolación de meshes, radiosidad, árboles BSP, LOD, superficies curvas y superficies reflectantes. Incorpora luces multicolores, dinámicas, lightmaps, raytracing y enveloped lighting. Soporta el formato DXF. Incorpora texture mapping, mapas de sombras, mapas de niebla, textura detallada para definir objetos muy detallados, texturas procedurales, texturas en tiempo real de ondas, 12 niveles de mipmapping, animación de texturas, texturas procedurales, dinámicas y multitextura. De entre otros detalles destacamos: detección de colisiones cilíndrica, superficies curvas con LOD, mapas de entorno, inteligencia artificial avanzada, sistema físico adaptable, sprites 3D o sonido digital 3D.

Define un lenguaje de script: UnrealScript, que es un lenguaje semicompilado para acceder a la lógica del juego y usar el potencial del motor gráfico. Permite trabajar con una interfaz de alto nivel para controlar los objetos en un juego.

El desarrollo de juegos y herramientas que hacen uso del editor de mapas UnrealED.

La orientación a objetos de *Unreal* permite añadir nuevos objetos y funcionalidades una vez terminado el desarrollo.

La maquina virtual de *Unreal* esta compuesta de: servidor, cliente, motor de visualización y motor de soporte de código. El servidor controla el juego y las interacciones entre los jugadores y los actores. Cada actor en el mapa puede estar bajo control de un jugador o el control de un script. El script define completamente como el actor se mueve e interacciona con otros actores.

El bucle de actualización sigue los siguientes pasos [Epic]:

1. El servidor comunica el estado del juego a los clientes. El estado del juego es el conjunto de estados de sus elementos.
2. Cada cliente envía al servidor la petición de movimiento. El servidor le contesta con el nuevo estado del juego. El cliente comienza el proceso de visualización de su escena.
3. El cliente realiza la operación de actualización (*Tick()*) del estado del juego, tomando en consideración el tiempo desde la última actualización (última llamada a la función *Tick()*).

Para gestionar el tiempo, *Unreal* divide cada segundo del juego en ticks [Lucia:2003]. El estado del videojuego se actualiza con cada tick. Un tick es la menor unidad de tiempo con la cual el actor puede ser actualizado. Normalmente la frecuencia de ticks suele estar entre 10 y 100 veces por segundo, aunque este valor depende de la potencia de la CPU. Las funciones que necesitan mas de un tick para su ejecución se llaman funciones latentes (como *Sleep*, *FisnishAnim* o *MoveTo*). Mientras un actor está ejecutando una función latente, la ejecución del actor se paraliza hasta que termina de ejecutarse. Sin embargo, otros actores o la maquina virtual puede seguir ejecutándose. Cada actor tiene su propio hilo de ejecución. Pero, estos hilos son virtuales. *Unreal* simula la utilización de hilos. Los scripts se ejecutan en paralelo.

En la operación de actualización de los actores, se les informa de los eventos pendientes, ejecutando el código del script asociado. Todo el código asociado a la actualización de los actores se diseña para que el tiempo que ha pasado desde la última actualización pueda ser variable (igual que en Fly3D, pero diferente a *Doom* y *Quake*, quienes actualizan el sistema suponiendo que el intervalo desde la última actualización es fijo).

La clase *Actor* es la clase padre de todos los objetos de un juego en *Unreal*. La clase *Actor* contiene todas las funcionalidades necesarias para que el actor se mueva,

interacciones con otros actores, afecte al entorno y se relacione con los objetos del juego. *Pawn* es la clase padre de todas las criaturas y jugadores en *Unreal*, las cuales son capaces de tener un control a alto nivel definido por su inteligencia artificial o por el control del jugador.

2.1.2.5. Gestión de Eventos en Videojuegos y Motores de Videojuegos

Las partes de los videojuegos que interesan para la presente tesis son:

- Bucle principal o bucle de actualización, para comprobar si sigue un esquema acoplado o desacoplado de las fases de visualización y simulación.
- Gestión de eventos del sistema [Alexander:2003][Treglia:2002], para comprobar si se sigue un esquema de simulación continuo (se recorren todos los objetos del sistema preguntando a cada objeto si debe actualizarse) o discreto.

La figura 2.2 incluida en el apartado 2.4 del presente capítulo, muestra el bucle principal de *Doom*, *Quake* y *Fly3D*.

El código de *Quake* es más legible y está más organizado que el código de *Doom*, si bien ambos juegos están bastante desestructurados y escasamente documentados. Debido a que la comunidad de usuarios de estos videojuegos está fundamentalmente interesada en la personalización de los escenarios y personajes del juego, no existe documentación referente a la gestión de eventos. Éste hecho ha dificultado enormemente su estudio. Tanto en *Doom* como *Quake* existe una gestión de eventos de usuario (como indicaciones de movimientos, menús o disparos) completamente diferenciada de la gestión del comportamiento. Incluso en *Quake* existen mensajes de consola con un tratamiento también diferenciado. En cambio, el código de *Fly3D* es un ejemplo de claridad y estructuración.

La obtención de los eventos de usuario en *Doom*, *Quake* y *Fly3D* se realiza mediante técnicas de polling, priorizándose los eventos generados por el usuario frente a los generados por el resto del mundo. Esto es debido a que los eventos generados por el usuario se resuelven antes de pasar a analizar el resto del videojuego.

Se define el **ciclo de simulación** como el intervalo de tiempo transcurrido en cada recorrido del bucle principal del programa. Este intervalo contiene la lectura de los eventos de usuario, gestión del comportamiento de los personajes, emisión de sonidos y visualización de la escena. El tiempo empleado por un ciclo de simulación corresponde al periodo de muestreo de un simulador continuo. Cada cálculo de la evolución del mundo requiere forzosamente una visualización completa de todo el mundo. Por tanto, siguen un esquema de simulación continuo (algoritmo 1 del apartado 2.4) en el que la frecuencia de muestreo depende de la potencia de cálculo disponible y de la complejidad del sistema a simular.

Los eventos se gestionan recorriendo el grafo de escena (*Doom* y *Fly3D*) o la lista de eventos (*Quake*) (en cualquier caso, recorriendo el descriptor de escena). Los principales inconvenientes de esta forma de gestionar los eventos son los siguientes:

1. Debe recorrerse todo el universo, preguntando a cada objeto si tiene un evento asociado que deba cumplirse en el instante actual. Se recorren todas los objetos, independientemente de que estén activos o no, o tengan o no eventos pendientes. Esto supone ralentizar el proceso comprobando objetos innecesariamente.

2. No hay ninguna ordenación de eventos según el tiempo. Todos los eventos se ejecutan en el orden en que los objetos están situados en el grafo de escena o el vector de entidades, lo que implica:

   - El sistema no es sensible a tiempos menores que el periodo de muestreo.
   - El periodo de muestreo no es seleccionable ni configurable explícitamente.
   - Los eventos se pueden ejecutar en orden incorrecto ya que se ejecutan en función de su ordenación en el grafo o vector y no en el tiempo en el cual acontecen. El orden depende de la posición del evento en la lista y no del instante de tiempo en que dicho evento debía haber ocurrido.
   - Los eventos son artificialmente sincronizados coincidiendo con el periodo de muestreo.

3. Si existe realimentación de eventos dentro del mismo ciclo, ésta no puede resolverse hasta el siguiente ciclo.

Las conclusiones obtenidas para *Unreal Tournament* son muy similares, pero no es posible acceder a la parte del código que interesa al presente estudio, lo que dificulta las conclusiones.

Unreal Tournament tiene un bucle de actualización muy parecido al bucle principal de los videojuegos anteriores. En cada pasada del bucle se visualiza y se actualiza la escena para el siguiente movimiento. La operación de simulación de un objeto toma en cuenta el tiempo desde la última actualización para calcular el nuevo estado del objeto, al igual que *Fly3D*. En el momento de la simulación se le indican al objeto los eventos que tiene pendientes.

2.2. Visualización y Simulación en las Aplicaciones Gráficas de Realidad Virtual

Las primeras aplicaciones gráficas en tiempo real trabajaban con un bucle principal que acoplaba la fase de visualización con la fase de simulación. Es decir, por

cada evolución del mundo se realizaba una visualización de este. El acoplo puede producirse en dos niveles:

- **Nivel de objeto**: se aprovecha el recorrido del grafo de escena para simular y visualizar al mismo tiempo cada uno de los objetos.
- **Nivel de sistema**: primero se simula, bien recorriendo el grafo de escena o alguna estructura auxiliar, y después se visualiza (o viceversa). Por cada simulación se realiza una visualización, pero son procesos separados.

Si se desacoplan estas fases, las escenas se visualizan más rápidamente, incluso cuando se hacen más complejas [Shaw:1992]. El desacoplo incrementa el rendimiento del sistema [Darken:1995]. El desacoplo es una técnica ampliamente utilizada en el campo de la realidad virtual.

El campo de la realidad virtual tiene multitud de similitudes con el campo de los videojuegos, de forma que la amplia investigación en el campo de la realidad virtual puede ser considerada a la hora de crear núcleos de videojuegos. Algunas de estas similitudes son: interacción con el usuario, necesidad de tiempo real, algoritmos de visualización, detección de colisiones o necesidad de dotar de comportamiento continuo las entidades basadas en técnicas de inteligencia artificial.

Entre otros sistemas de realidad virtual, algunos de los que desacoplan las fases de visualización y simulación son:

- ***Cognitive Coprocessor Architecture*** [Robertson:1989]: es un modelo de arquitectura para interfaces de aplicaciones de realidad virtual basado en el modelo de interacción del sistema de tres agentes [Sheridan:1984]. El mecanismo básico de control del sistema contiene una cola de tareas y una cola de visualización, independientes la una de la otra, que se tratan de forma separada. En el bucle de procesado se gestionan los eventos de usuario de la cola de tareas hasta que está vacía y se visualizan los objetos de la cola de visualización.
- ***Interfaz de diálogo*** [Lewis:1991]: describe una arquitectura software para mundos virtuales basada en múltiples procesos comunicándose a través de una interfaz basada en eventos. El sistema se descompone en tres partes: simulación, mapeado y visualización. Cada una de ellas corresponde a un proceso independiente. Cada sistema se comunica con los restantes mediante paso de mensajes asíncrono. También se comunica de esta forma con los dispositivos de E/S. El paso de mensajes está coordinado por un núcleo central.
- ***MR Toolkit*** [Shaw:1992]: es un conjunto de herramientas para desarrollar aplicaciones de realidad virtual. Está basado en el Modelo de Simulación De-

sacoplada, consistente en descomponer el sistema en partes. Una parte se encarga de la computación no gráfica, de actualizar los datos de la aplicación en series de pasos en tiempo discreto. Cuando los datos son consistentes, se los envía a la parte encargada de gestionar el modelo geométrico. Otra parte se encarga de visualizar, obteniendo los datos de la parte de gestión del modelo geométrico y de la parte encargada de la interacción. Este modelo consta de dos bucles ejecutándose asíncronamente, lo que permite soportar simulación continua y discreta. Cada parte se ejecuta en procesadores independientes.

- ***Alice & Diver*** [Pausch:1994] utiliza el desacoplo para el prototipado de entornos virtuales: *Alice* ejecuta la simulación y *Diver* [Gossweiler:1994] proporciona una base de datos gráfica, gestiona las funciones gráficas de bajo nivel y gestiona los dispositivos de E/S. Ambos sistemas se distribuyen.

- ***VB2 Virtual Builder II*** [Gobbetti:1995]: es una arquitectura para la construcción de aplicaciones interactivas 3D. El sistema se compone de un grupo de procesos interconectados. Cada proceso está continuamente ejecutándose y comunicándose con otros procesos a través de mensajes asíncronos. Un proceso central se encarga de gestionar el sistema completo y de evolucionar el sistema como respuesta de los eventos del sistema y las E/S.

- ***Bridge*** [Darken:1995]: es una arquitectura para la construcción de mundos virtuales. El sistema se descompone en cuatro partes interconectadas: simulación, aplicación, visualización y gestión de diálogo. Esta descomposición incrementa las prestaciones del sistema, manteniendo una frecuencia de cuadro elevada.

El desacoplo de las fases de visualización y simulación permite incrementar la precisión y velocidad del sistema, además de la independencia de velocidad de los procesos del sistema [Agus:2002].

La evolución natural del desacoplo fue distribuir los procesos del sistema en una red de computadores o usar paralelismo [Pausch:1994]. Sin embargo, esta distribución no es posible en los videojuegos creados para ejecutarse sobre un único procesador. Los videojuegos en red no son juegos distribuidos, permiten múltiples usuarios pero no distribuyen los procesos en red. En [Macedonia:1997] [Smed:2001] se muestra una clasificación de aplicaciones de realidad virtual. Algunas aplicaciones de realidad virtual distribuida son *Vega* [Vega], *Dive* [Frecon:1998] o *Massive* [Greenhalgh:1998].

2.3. Simulación de Eventos Discretos

2.3.1. Conceptos Básicos de la Simulación de Eventos Discretos

La Teoría de Sistemas [Bertallanfy:1968] define un **sistema** como cualquier entidad real o artificial, que puede estar compuesta por entidades más simples con relaciones espaciales o temporales. La interacción entre las entidades del sistema define sus características y se traduce en la evolución del sistema hacia un objetivo concreto.

Un **evento** o **suceso** es el cambio de estado de una entidad del sistema, una ocurrencia instantánea que altera el estado del sistema [Fishman:1978]. El estado de una entidad lo define el valor de sus atributos. El estado del sistema lo define el conjunto de todos los estados de todos los objetos y entidades que forman el sistema.

Existen diversas formas de clasificar los sistemas. Una posible clasificación atiende a la forma en la que evoluciona el tiempo dentro del sistema cuando se produce un suceso [Law:1982]:

- **Sistema continuo**: el sistema bajo estudio se considera como un flujo continuo de información, el tiempo de simulación evoluciona de forma continua. Ejemplos de estos sistemas son los encontrados en la naturaleza.
- **Sistema discreto**: el tiempo de simulación evoluciona de forma discreta (se incrementa a intervalos de tiempo). La información del sistema únicamente se procesa en estos intervalos de tiempo. Un caso particular de los sistemas discretos son los **sistemas de eventos discretos**, en estos sistemas el tiempo evoluciona con la ocurrencia de un suceso del sistema, el tiempo pasa a ser el tiempo de ocurrencia del suceso. Ejemplos de estos sistemas son líneas de producción, redes de ordenadores, sistemas de control de tráfico,...
- **Sistema híbrido**: son sistemas donde algunas entidades se comportan de forma continua y otras entidades de forma discreta [Pritsker:1974].

Hay tres métodos básicos para adquirir información de un sistema, con el objeto de conocer su comportamiento o modificarlo: experimentar, analizar y simular.

- **Experimentar** consiste en estudiar el sistema real en funcionamiento y tomar medidas directamente. Es el método más preciso, pero no siempre es posible llevarlo a cabo, hay experimentos que son demasiado peligrosos, demasiado caros o simplemente no es posible disponer del sistema real todavía.
- **Analizar** es utilizar modelos basados en el razonamiento para calcular directamente los tiempos del sistema, lo que supone una simplificación del sistema real. Supone realizar asunciones que no siempre se corresponden con la

realidad. Cualquier variación del modelo resulta complicada de realizar. Este método es el que consume mayor cantidad de tiempo de desarrollo.

- **Simular** es imitar el comportamiento dinámico de un sistema con el fin de llegar a conclusiones aplicables al mundo real [Banks:2001]. Se define la secuencia de actividades que tienen lugar en la dinámica del sistema [Forrester:1970]. La simulación permite variar el comportamiento del sistema y estudiarlo.

Habitualmente no es posible estudiar un sistema directamente, ni mucho menos modificarlo para observar su comportamiento. Por ello, un sistema se representa como un modelo. Un **modelo** es una representación abstracta de un sistema [Naylor:1975]. La simulación es un método experimental, pero en vez de experimentar con el sistema real, se experimenta con el modelo de simulación que se crea a tal efecto. Habitualmente, el modelo se construye mediante un computador, pero el término simulación va más allá: un modelo puede ser una maqueta o cualquier representación de un sistema. La simulación no es la forma perfecta de obtener información de un sistema, tiene ventajas e inconvenientes [Coos:1992].

La simulación de eventos discretos se utiliza para modelos de sistemas donde los cambios de estado pueden representarse mediante la ocurrencia de una serie de sucesos en *quantos* de tiempo discretos [Fishman:1978] [Schriber:1999]. Los cambios en el sistema ocurren en el momento del evento, por tanto el tiempo del sistema avanza con la ocurrencia de eventos.

Los modelos de sistemas de eventos discretos se pueden elaborar siguiendo tres enfoques diferentes [Fishman:1978]:

1. **Enfoque de programación temporal de eventos**: un evento es el cambio de estado de una entidad del sistema. Mediante este enfoque, el modelo se describe como la secuencia de pasos que suceden a la ocurrencia de un evento.

2. **Enfoque de examen de la actividades**: un proceso es una secuencia de eventos ordenada temporalmente. Se consideran que actividades deben iniciarse o finalizarse con la ocurrencia de un evento.

3. **Enfoque de interacción de procesos**: una actividad es una serie de operaciones que transforma el estado de una entidad. Este enfoque indica el progreso de una entidad a través del sistema, desde el evento que provoca su llegada hasta que abandona el sistema.

2.3.2. Lenguajes de Simulación

Para que un lenguaje de programación sea útil en la simulación debe ser capaz de:

- Definir las clases de entidades que hay en un sistema y las características de cada entidad.
- Ajustar el número de entidades según varíen las condiciones del sistema.
- Relacionar las entidades entre sí y con el ambiente común.
- Proporcionar un método de control de la simulación.
- Gestionar los eventos que se producen en el sistema, ejecutándolos en el instante de tiempo especificado como de ocurrencia del evento.
- Disponer de mecanismos para la generación de números aleatorios.
- Permitir el análisis estadístico de los resultados de la simulación.
- Proporcionar un método para gestionar el tiempo de simulación del sistema (reloj del sistema).

Estas características no son exclusivas de los lenguajes de programación de simulación, sino que se encuentran en otros lenguajes. Cualquier lenguaje de programación puede utilizarse para simular, pero los lenguajes específicos de simulación ofrecen facilidades para implementar el modelo fácilmente y obtener resultados, además de permitir una programación mínima.

Los lenguajes de simulación se pueden clasificar en 4 categorías [Corbacho:1997]:

1. **Biblioteca de un lenguaje de programación generalista**: permiten incluir en la simulación todas las facilidades de un lenguaje de programación genérico. Si el lenguaje es orientado a objetos, permite utilizar dicha orientación a objetos en la creación del modelo. Se utiliza un entorno de desarrollo familiar al usuario. Los modelos heredan las características de portabilidad del lenguaje genérico. Suelen conllevar tiempos de definición del modelo largos, con fases de depuración y mantenimiento costosas.
2. **Lenguajes de simulación de propósito general**: estos simuladores se crean definiendo un lenguaje propio para la simulación. Permiten crear los modelos fácilmente si se domina el lenguaje, pero exigen que el usuario aprenda un nuevo lenguaje. El modelo no puede integrarse en una aplicación genérica y, además, suelen tener formatos de E/S muy estrictos. Incluyen facilidades para detección de errores en la definición del modelo. Suelen incluir entornos de desarrollo con editores y compiladores.
3. **Lenguajes generalistas de especificación de modelos**: son lenguajes propios que permiten definir un modelo de forma descriptiva, por lo que se facilita

la implementación del modelo, pero suelen tener tiempos de simulación elevados. Después de describir el modelo se puede simular. Algunos lenguajes de este tipo incorporan facilidades para crear el modelo gráficamente, permitiendo definir el modelo de forma fácil e intuitiva.

4. **Lenguajes de simulación específicos**: con el trascurso de los años se ha tendido a especializar los simuladores para una tarea concreta o un campo específico, intentando facilitar al máximo la creación del modelo, permitiendo incluso animar la simulación (a costa de perder velocidad). Estos simuladores añaden la facilidad de utilizar una nomenclatura muy similar a la utilizada en el sistema real, lo que facilita la tarea de creación del modelo.

2.3.3. Simulación de Eventos Discretos

La construcción de modelos de simulación se inició en el Renacimiento, pero, el uso actual de la palabra simulación viene de 1940 [Smith:2000], cuándo los científicos Von Newman y Ullam, que trabajaban en el proyecto de Monte Carlo durante la segunda guerra mundial, resolvieron problemas relacionados con reacciones nucleares, cuya solución experimental sería muy costosa y cuyo análisis matemático resultaría demasiado complejo. La simulación de eventos discretos apareció algo más tarde, en la década de los 50, como método de análisis para estudiar problemas, habitualmente basados en la Teoría de Colas [Fishman:1978] [Jain:1991]. El impulso real a la simulación de sistemas fue la utilización de computadoras, que permitió llevar a cabo simulaciones complejas, imposibles de llevar a cabo analíticamente.

La historia de la simulación se puede desglosar en los siguientes periodos (entre 1955 y 1987 [Nance:1993] [Nance:1995], de 1988 a la actualidad [Banks:2001]):

1955-1960 Periodo de búsqueda.

1961-1965 Periodo de advenimiento.

1966-1970 Periodo formativo.

1971-1978 Periodo de expansión.

1979-1986 Periodo de consolidación y regeneración.

1987-Actualidad Periodo de entornos integrados.

Durante estos periodos hay una característica básica, la intensa competición comercial que se genera entre los diversos simuladores [Nance:1995].

Lenguaje	Orientación	Fecha Creación
GPSS	Procesos	1961
SIMSCRIPT	Eventos	1963
GASP	Eventos	1961
SIMULA	Procesos	1961
CSL	Actividades	1961
SIMPAC	Actividades	1963
OPS-3	Actividades	1964

Tabla 2.3: LPSED 1961-1965

2.3.3.1. Periodo de Búsqueda (1955-1960)

En los primeros tiempos de la simulación de eventos discretos se utilizaban lenguajes de propósito general, como *FORTRAN* o *ALGOL60*, para realizar simulaciones. Utilizar lenguajes sin rutinas específicas de simulación es laborioso y complejo, por lo que durante este periodo se realizó un gran esfuerzo en clarificar conceptos y sentar las bases de la representación de modelos con el objeto de crear lenguajes de simulación.

El primer lenguaje de programación de simulación de eventos discretos (LPSED) fue ***GPS*** (General Simulation Program) [Tocher:1960]. Es un lenguaje orientado a actividades y constituye la base para posteriores LPSED.

2.3.3.2. Periodo de Advenimiento (1961-1965)

En esta etapa nacen los precursores de los lenguajes que han sentado las bases de la simulación y que, algunos, se usan en la actualidad (alguna de sus versiones posteriores) y otros se han usado hasta hace pocos años. La tabla 2.3 muestra los principales LPSED de esta etapa y su orientación. La tabla 2.4 muestra como evolucionan estos lenguajes en etapas posteriores.

GPSS (General Purpose Simulation System) [Wexelblat:1981] se creó para la simulación de comunicaciones y sistemas de computadores. Pero su facilidad de uso hizo que se extendiese rápidamente a otros campos. Su semántica es apropiada para modelar problemas de colas. Permite modelar el sistema como una serie de cambios de estado que ocurren instantáneamente. Utiliza métodos de computación numérica para mantener un seguimiento de los elementos del sistema en el tiempo. Consiste en un conjunto de bloques (como generadores, colas, servidores o selectores) y las conexiones entre ellos.

SIMSCRIPT [Markowitz:1963] [Markowitz:1979] está basado en *FORTRAN*, pero las versiones posteriores los fueron desligando. El objetivo de su creación fue disminuir los tiempos de desarrollo e implementación del modelo. En un intento de

61-65	**66-70**	**71-78**	**79-86**
GPSS *GPSS III*	*GPSS/360* [Reitman:1970] *GPSS* V [Gpssv:1970] *GPSS/NORDEN* [Reitman:1970]	*GPSS 1100* [Gpss1100a:1971] *NGPSS* [Ngpss:1971] *GPSS V/6000* [Gpssv6000:1975] *GPSS/H* [Gpss1100b:1971]	*GPSS/PC* [Cox:1984] *GPSS/85* [Henriksen:1985] *PL/I GPSS* [Rubin:1981] *GPSS/FORTRAN* [Schmidt:1980]
SIMSCRIPT	*SIMSCRIPT II* [Delfose:1976] *SIMSCRIPT II Plus* [Nance:1995]	*SIMSCRIPT II.5* [Russell:1983] *C-SIMSCRIPT* [Nance:1995]	*SIMFACTORY II.5* [Goble:1991] *COMNET II.5* [Garrison:1991] *NETWORK II.5* [Garrison:1991]
GASP	*GASP II* [Pritsker:1969] *GERTS* [Klein:1970]	*GASP IV* [Pritsker:1974] *SLAM* [Pritsker:1979] *GASP PL/I* [Pritsker:1975] *SAINT* [Nance:1995]	*SIMAN* [Pedgen:1990] *SLAM II* [Pritsker:1995] *INSIGHT* [Roberts:1983]
SIMULA	*SIMULA67* [Nygaard:1981]	*DEMOS* [Nance:1995] *SIMPL/I* [SimplI:1972] *PASCAL* [Davies:1989]	*SIMPL/I X* [Nance:1995] *SIMPAS* [Bryant:1980] *PASSIM* [Uyeno:1980] *INTERACTIVE* [Lakshmanan:1983]
CSL	*ECSL* [Clementson:1966]	*EDSIM* [Nance:1995]	-
SIMPAC	-	-	-
OPS-3	*OPS-4* [Jones:1967]	-	-

Tabla 2.4: Evolución de los LPSED 1961-1986

que pudiesen utilizarlo usuarios no expertos en programación, para cada elemento del sistema se utilizaban una serie de formularios y rutinas que definían aspectos concretos.

GASP [Kiviat:1963] (General Activity Simulation Program) es un conjunto de rutinas de *FORTRAN* (aunque las primeras versiones se basaban en *ALGOL*). Basado en diagramas de bloques. Nació para llenar el hueco entre programadores e ingenieros.

SIMULA [Wexelblat:1981] se basa en ALGOL60. Considerado difícil de aprender, con una interfaz complicada de utilizar pero muy innovador y flexible. La comunidad científica lo descubrió sobre 1970. Intenta modelar los objetos del mundo real de una manera natural, utilizando conceptos como tipos abstractos de datos o herencia.

CSL [Buxton:1966] (Control and Simulation Language) nació para solucionar problemas complejos de toma de decisiones de control industrial (creado por *Esso Petroleum Company*). En cuanto al tiempo de ejecución se consideró un programa muy eficiente. Está muy influenciado por *FORTRAN*, pero sigue las bases conceptuales del primer LPSED, *GPS*. La simulación toma la forma de un diagrama de entidades.

OPS-3 [Greenberger:1965] fue un LPSED puntero en su época. Incluye características como multipropósito o modularidad.

Los simuladores de este periodo están implementados como lenguajes propios. Aprender un lenguaje suponía un alto coste para la simulación. Además, se debían migrar los programas a cada hardware [Nance:1993]. Por ello comenzó el interés por las bibliotecas de simulación de lenguajes de propósito general.

Durante este periodo se realizaron multitud de comparativas de lenguajes según diversos aspectos [Reitman:1967] [Teichroew:1966] [Young:1963] [Krasnow:1964].

2.3.3.3. Periodo formativo (1966-1970)

En este periodo se consolidan los conceptos de la simulación de eventos discretos, se revisan y redefinen los conceptos para clarificar la representación de los sistemas. Los lenguajes se hacen más maduros. La rápida evolución del hardware forzó a algunos lenguajes a crear nuevas versiones.

Algunos hitos importantes:

- ***SIMULA67***: [Nygaard:1981] añade los concepto de clases de objetos y herencia.
- ***SIMSCRIPT II***: [Delfose:1976] establece los conceptos de entidad, atributo y conjunto.

- ***OPS-4***: [Jones:1967] permite la interrupción de la simulación y la redefinición de atributos.

2.3.3.4. Periodo de Expansión (1971-1978)

Durante este periodo hay una gran expansión de lenguajes basados en los ya existentes. Se crean versiones para distintas máquinas y sistemas operativos. Siguen evolucionando y mejorando.

2.3.3.5. Periodo de consolidación y regeneración (1979-1986)

Durante este periodo los lenguajes extienden su implementación con el objeto de distribuirse o paralelizarse, mientras que las características básicas del lenguaje no varían.

Hitos importantes:

- Aparece el lenguaje de propósito general ***PASCAL*** [Davies:1989] y se crean diversos simuladores basados en él: *SIMPAS* [Bryant:1980], *PASSIM* [Uyeno:1980] e *INTERACTIVE* [Lakshmanan:1983].
- ***SLAM*** (Simulation Languaje for Alternative Modeling) [Pritsker:1979] cambia la estrategia de compilado de sus predecesores, es un preprocesador de *FORTRAN* (sus predecesores eran paquetes).
- Comienza la aparición de simuladores especializados:
  - ***SIMAN*** (Simulation Analisys) [Pedgen:1990] se crea para procesos de fabricación. Supone la aparición de los paquetes especializados. Modelado similar a *GASP IV*. Se convirtió en el principal LPSED ejecutable en los PC de IBM.
  - ***INSIGHT*** [Roberts:1983] se crea para modelar sistemas médicos. Utiliza un diagrama de transacciones para el modelado. La representación gráfica del sistema se debe traducir manualmente a sentencias. Incluye un preprocesador de *FORTRAN*.

2.3.3.6. Periodo de entornos integrados (1987-Actualidad)

Durante este periodo hay un gran avance en los lenguajes de simulación para ordenadores personales y nacen multitud de entornos de simulación con interfaces gráficas de usuario.

Los paquetes de simulación poseen las siguientes **características**, si bien no todos los paquetes las poseen todas:

- Datos de la simulación:
  - Análisis estadístico de los datos de salida.
  - Herramientas de optimización basadas en los resultados del análisis de los datos.
  - Permiten automatizar la obtención de datos de E/S.
  - Permiten exportar los datos de salida a otras aplicaciones.
- Creación de modelos:
  - Las interfaces de usuario tienden a mejorarse, simplificando la creación de modelos.
  - Definición de escenarios.
  - Creación gráfica de modelos.
  - Intentan simplificar la creación de modelos usando diagramas de flujo o de bloques.
  - Utilizan técnicas de *relleno de huecos* permitiendo al usuario programar sin necesidad de conocer la sintaxis del lenguaje.
- Especialización:
  - Son simuladores especializados u orientados a campos concretos, o a un conjunto de ellos.
  - El lenguaje utilizado está más próximo al área de conocimiento, lo que simplifica su utilización por no-programadores.
- Animaciones 2D o 3D. La animación puede realizarse durante la simulación o posteriormente. Algunos permiten almacenar la animación en un fichero de vídeo.
- Control de la ejecución, como la posibilidad de variar las propiedades de los objetos durante la simulación.
- Simuladores comerciales, con un coste elevado.

Los lenguajes de simulación difieren en el grado en que es posible su aplicación a tipos particulares de sistemas y hasta el grado en que pueden suministrar procedimientos de simulación mas o menos automáticos. El lenguaje de simulación más adecuado para un estudio en particular depende de la naturaleza del sistema y de la habilidad para programar que tenga el individuo que realiza el estudio. Como regla general, se requiere un mayor entendimiento de los procedimientos de programación para obtener un incremento en la flexibilidad de un programa de simulación. Del mismo modo, cualquier reducción en el tiempo de programación que se logre mediante

la utilización de lenguajes de simulación, está generalmente asociada a incrementos en el tiempo de cómputo.

La decisión de la utilización de un lenguaje de simulación u otro depende de una serie de consideraciones, tales como:

- Disponibilidad de la computadora: confirmar que el compilador del lenguaje sea compatible con la computadora.
- Disponibilidad de programadores con conocimientos de lenguajes determinados.
- Documentación y diagnóstico de errores: forma en que el simulador trata las inconsistencias y errores lógicos.
- Eficiencia: coste temporal del análisis, programación, mantenimiento y simulación.

Seleccionar un software de simulación es una tarea complicada, considerando que la oferta del mercado es muy amplia [Banks:2001]. Durante este periodo se crean guías de software de simulación [BuyersGuide:1999] [Swain:1999] (habitualmente sólo tratan software comercial).

2.3.3.7. Lenguajes de Simulación de Eventos Discretos y Bibliotecas

GPSS es el LPSED más utilizado a lo largo de la historia [Stahl:2001]. Multitud de versiones siguen utilizándose, entre ellas destaca ***GPSS/H*** [Crain:1999]. Basado en la interacción de procesos, orientado a colas y altamente estructurado. El sistema se describe como un diagrama de bloques. Incluye un animador 2D que puede ejecutarse durante la simulación o después de obtener resultados. Es una herramienta potente de simulación, la ejecución es más rápida que en versiones anteriores. Es un simulador comercial [Wolverine]. Ciertas versiones antiguas están liberalizadas.

CSIM [Schwetman:1987] es una biblioteca de C++ [Stroustrup:2001] que facilita el uso de C++ en la simulación. Ampliamente utilizado tanto en la industria como en docencia. Su utilización principal es el modelado de sistemas de ordenadores y comunicaciones. Tiene versiones liberadas. La versión más completa, ***CSIM18*** [Schwetman:1996], no está liberada [Mesquite].

QNAP2 [Qnap] [Qnap:1990]: es un lenguaje de modelado y simulación basado en el paradigma de redes de colas. Es un lenguaje orientado a objetos. Permite diseñar modelos de alto nivel. Contiene diversos métodos analíticos para el análisis de datos de salida.

Otros LPSED que siguen utilizándose en la actualidad son:

- ***SIMSCRIPT II.5*** [Caci].
- ***SIMPACK*** [Simpack] es una biblioteca de C de código libre para simulación continua, discreta e híbrida. La versión para C++ es *SIMPACK++*. Es un simulador para docencia e investigación.
- ***PASION32*** [Raczynski] es un traductor que genera código en *PASCAL*. Es un simulador comercial.
- ***SIMPLE++***, actualmente denominado *eM-Plant* [eM-Plant] y orientado a la industria del automóvil.

2.3.3.8. Paquetes de Simulación de Eventos Discretos y Bibliotecas

Arena [Sadowski:1999] es un paquete de simulación comercial [Arena]. Permite simular sistemas continuos o discretos. Está especializado en el modelado de procesos comerciales. El modelo se describe gráficamente y permite realizar la descripción del modelo en otras aplicaciones. El núcleo de simulación de Arena es *SIMAN*. *Arena Product Family* incluye una gama de herramientas para facilitar el proceso de simulación.

AutoMod [Rohrer:1999] es un paquete de simulación comercial [AutoMod]. La familia de productos *AutoMod* incluye el paquete de simulación *AutoMod*, el paquete de experimentación y análisis *AutoStat* y el paquete *AutoView* para construir videos con las animaciones 3D de la simulación. Se centra en producción y sistemas de tratamiento de materiales. Un modelo está compuesto de sistemas y los sistemas se componen de procesos. El modelo se crea definiendo los sistemas, su lógica y flujo de control.

QUEST (Queuing Event Simulation Tool) [Donald:1998] es un paquete de simulación comercial [Quest]. Es un paquete de simulación de eventos discreto basado en objetos. Permite crear el modelo de forma gráfica. Permite modificar los parámetros del modelo en tiempo real. Esta orientado a procesos de producción. Permite construir el modelo gráficamente. Permite procesado batch.

Extend [Krahl:1999] es un paquete de simulación comercial [ImagineThat]. Es una aplicación modular: permite añadir una serie de módulos para adaptarlo al área en la que se desee realizar la simulación. El módulo básico de simulación está diseñado para simular sistemas continuos, lo que lo hace apropiado para diseños de ingeniería, análisis científico,... La adición de módulos al paquete básico permite simulación discreta, continua e híbrida. Está orientado a procesos. Está especializado en simulación de procesos industriales. Los modelos se construyen conectando bloques, cuyas características se han definido previamente. Los conjuntos de modelos forman jerarquías, lo que permite una creación modular del sistema.

MicroSaint [Bloechle:1999] es un paquete de simulación comercial [MicroSaint]. Es un simulador de eventos discreto que permite simular cualquier proceso que pueda representarse como un diagrama de flujo de tareas. La simulación se anima en formato icónico. Se usa en campos como producción, medicina o aplicaciones militares.

ProModel [Price:1999] es un paquete de simulación comercial [ProModel] diseñado para la industria de producción. Es fácil de usar y permite obtener resultados rápidamente. Permite animar el modelo. Es posible incluir rutinas en PASCAL o C en el modelo. El modelo se construye definiendo el esquema y los elementos dinámicos de la simulación.

WITNESS [Mehta:1999] es un paquete de simulación comercial [Lanner]. Contiene elementos para simulación continua y discreta. Los modelos se basan en elementos de plantillas, que pueden combinarse formando módulos reutilizables. Permite la animación 2D del modelo. Especializado en procesos de fabricación discreta.

SIMPROCESS 3.2 es un simulador comercial de negocios [Caci] basado en *SIMSCRIPT II.5*. Está orientado a procesos y combina mapeado de procesos con simulación de eventos discretos y coste basado en actividades. Es fácil de usar, pues el modelo se construye gráficamente. Diseñado para organizaciones que necesitan analizar una variedad de escenarios de operación. Utiliza tecnología *Java* y *XML*.

Modline [Simulog] permite obtener indicadores de rendimiento de una gran variedad de aplicaciones, como sistemas de información distribuidos, redes de comunicaciones, redes de transporte o sistemas de logística y producción. Integra el núcleo de simulación *QNAP2*, por lo que contempla multitud de posibilidades de análisis de resultados.

La evolución posterior de la simulación de eventos discretos ha seguido dos líneas diferentes: paralelizar o distribuir el simulador y adaptar los simuladores a Internet:

- **Simulación paralela** [Alois:1994] [Overeinder:1991]: *ParaSol* [Purdue], POSE [Ppl], *PROSIT* [Ferrante:1994], *SPADES* [Spades], *COST* [Yucesan:2002],...
- **Simulación distribuida** [Misra:1986]: *Yaddes* [Yaddes] [Preiss:1989], *Maisie* [Bagrodia:1994], *SimKit* [Gomes:1995], *CS-DEVS* [Wangerin:2002],...
- **Simulación basada en web**: se analiza en el apartado 2.3.4.

2.3.4. Simulación Basada en Web

Los simuladores se han ido adaptando a las nuevas tecnologías, creando versiones de los simuladores tradicionales adaptadas a Internet. La simulación basada en web

Característica web	**Simulación basada en web**	**Simulación tradicional**
Estandarizada	Si	No
Independiente de la plataforma	Si	No
Interoperatividad	Casi nunca	No
Facilidad de navegación	Varía	Varía
Facilidad de uso	No	No

Tabla 2.5: Simulación basada en web vs. tradicional [Kuljis:2003]

no es un campo nuevo dentro de la simulación, sino la necesidad de adaptar el campo de la simulación a las nuevas tecnologías [Fishwick:1996]. No responde a una auténtica necesidad de la industria de utilizar herramientas de simulación en web [Kuljis:2003]. La simulación basada en web no ha aportado ninguna novedad al campo de la simulación [Kuljis:2003], pero la ha hecho accesible a gran cantidad de usuarios. La tabla 2.5 muestra como se ha adaptado la simulación en web a ciertas características propias de internet. Gran parte de los simuladores basados en web son versiones de simuladores existentes.

La simulación basada en web está permanentemente disponible desde cualquier navegador, lo que facilita el acceso de los usuarios desde diferentes ubicaciones, lo cual es una ventaja respecto a la simulación tradicional. Algunos simuladores disponen de bibliotecas y repositorios de modelos propios o aportados por los usuarios, lo que permite crear modelos más complejos de forma sencilla, basándose en modelos previamente creados, o aprender a modelar sistemas mediante ejemplos. No es necesario que el simulador esté instalado en los ordenadores locales, lo que facilita su posibilidad de utilización y permite que esté siempre actualizado. Una ventaja importante de la simulación en web es que permite construir simulaciones distribuidas (Distributed Interactive Simulation [Little:2003]), donde varios usuario interaccionan. Existe un gran número de simuladores de eventos discretos basados en web [Kuljis:2000], como *WebGPSS*, *SIMJAVA*, *JSIM*, *SILK* o *SML*.

WebGPSS [Stahl:2002] [Webgpss] es una herramienta de simulación en web que incluye un tutorial interactivo para aprender las bases de la simulación. Está basado en *micro-GPSS* [Stahl:1996], versión simplificada de *GPSS*. Es un simulador de eventos discretos muy sencillo de usar. Es un simulador utilizado en el campo de la docencia. Permite definir el modelo como un diagrama de flujo.

SIMJAVA [Howell:1998] es un simulador de eventos discreto basado en *SIM++* [Sim] [Fishwick:1995] de código abierto [Simjava]. Permite representar los objetos de la animación como iconos animados. Una simulación es un conjunto de entidades ejecutándose independientemente y comunicándose mediante eventos.

JSIM [Miller:1997] [Miller:2000] es un simulador basado en web escrito en *Java*

de código abierto [Jsim] [Jsimg]. Permite definir el modelo de forma gráfica, conectando nodos de diferente tipo. Incluye animaciones durante la ejecución de la simulación. Permite definir el modelo de forma modular y reutilizar componentes.

SILK [Healy:1998] es un simulador generalista escrito en *Java* como una biblioteca. Los modelos se escriben directamente en *Java* utilizando una biblioteca de clases. Permite definir los modelos de forma gráfica y modular, aunque también de forma textual.

SML [Kilgore:2001] [Kilgore2:2001] es una biblioteca de simulación de código libre que permite construir modelos de forma modular y reutilizar componentes.

Como en el resto de los simuladores, los simuladores basados en web también se han especializado, algunos ejemplos:

- Cirugía [Nis].
- Crecimiento de plantas: *PHENAPP* [Ars].
- Planetarios [Planetarium].
- Producción: *WaterSim* [Weisman:2000] (semiconductores), *scSimulator* [Dcra] (cadenas de producción) o *iRise Application Simulato*r [Devx] (análisis económico).

Todos estos simuladores están escritos en *Java*. Unos definen un nuevo lenguaje de programación y otros son bibliotecas de *Java* para la simulación. *Java* es un lenguaje elegante y potente, con características apropiadas para su ejecución en web [Kilgore:1998] [McNab:1996] [Haggar:2000] [Dibble:2002]:

- *Java* es un lenguaje orientado a objetos, lo que permite crear programas flexibles, modulares y reusables.
- Permite crear simuladores fácilmente accesibles en web, mediante un hiperenlace. Permite escribir código de simulación sin instalar el simulador. Las simulación se crean simplemente accediendo a una página web mediante un navegador.
- Soporte a la distribución de procesos.
- Independencia de la plataforma.
- Programación multihilo.

2.4. Conclusiones

La simulación de eventos discretos es de especial importancia en la actualidad por el elevado impacto que representa en áreas tan importantes como procesos de producción, ingeniería de la construcción, aplicaciones militares, logística, aplicaciones de transporte y distribución, simulación de procesos económicos, sistemas humanos o evaluación y configuración de sistemas de computadoras [Coos:1992]. Un intento de unir el campo de la simulación al campo de las aplicaciones gráficas apuntaba a la utilización de simuladores como animadores de aplicaciones gráficas, de forma que la salida de la simulación fuese una secuencia gráfica [Lee:1999].

Las conclusiones del estudio de los motores de videojuegos y de los núcleos de simulación son:

1. La mayor parte de los motores de videojuegos son o bien motores de visualización o videojuegos completos, no son motores específicos de simulación adaptados a los requerimientos de videojuegos.

2. No hay normalización en la creación de videojuegos, cada videojuego se programa explícitamente desde cero. Han surgido infinidad de motores de visualización que han permitido cierta estandarización en la parte gráfica. Entre ellos destaca *OpenGL Performer*.

3. La simulación y la visualización están fuertemente acoplados: la asociación de la visualización a la simulación obliga a que, para poder realizar un nuevo ciclo de simulación, se deba realizar forzosamente una visualización, con independencia de que ésta vaya a ser representada o no. Como excepción, *OpenGL Performer* desacopla estas fases.

4. Las aplicaciones de realidad virtual tienen un componente de simulación muy fuerte. Las exigencias de velocidad de interacción con el usuario son elevadas. Por ello, la carga de visualización suele minimizarse (escenas sencillas). La carga de visualización es también muy elevada, aunque se empleen técnicas para disminuir esa carga, como texturas pequeñas, baja poligonalización o pocas luces. La complejidad de estos sistemas es tan elevada, que se tiende a distribuir o paralelizar la aplicación para poder abordarla.

5. Todos los videojuegos analizados siguen el mismo esquema de simulación continua. La figura 2.2 muestra el esquema de funcionamiento de *Doom*, *Quake* y *Fly3D*. El bucle principal de estos videojuegos (algoritmo 1) sigue los siguientes pasos:

   a) Gestión y tratamiento de eventos de usuario.

   b) Gestión de eventos de objetos del sistema.

c) Visualización.

El bucle de actualización de *Unreal Tournament* es muy similar.

Se prioriza la gestión de eventos de usuario a la del resto de los objetos de la aplicación gráfica. La obtención de eventos de usuario se realiza mediante técnicas de polling.

Esquema de simulación que sigue cada uno de estos juegos:

a) *Doom*: acoplado continuo. Se simula y visualiza recorriendo el grafo de escena. Por tanto, el acoplo se realiza a nivel de objeto.

b) *Quake*, *Fly3D* y *Unreal Tournament*: acoplados continuos a nivel de sistema. Se simula y después se visualiza. El acoplo es a nivel de sistema.

Algoritmo 1 Bucle principal de los videojuegos [Pausch:1995]

```
while true do
  get information from input devices
  compute a tick of the simulation
  update graphics to the user
end while
```

Se ha simulado en laboratorio el comportamiento de los modelos acoplado y desacoplado. Sea:

T_S tiempo de simulación en un paso del videojuego.

T_R tiempo en visualizar un cuadro o fotograma.

ν_C frecuencia de cuadro (fotogramas por segundo).

SRR frecuencia de refresco de la pantalla (fotogramas por segundo).

N_S número de simulaciones.

N_R número de visualizaciones.

La conclusión obtenida para un sistema continuo acoplado se muestra en las ecuaciones siguientes.

Si

$$SRR \geq \nu_C \tag{2.1}$$

entonces

$$T_S + T_R \leq \frac{1}{SRR} \Rightarrow \frac{1}{T_S + T_R} = \nu_C \tag{2.2}$$

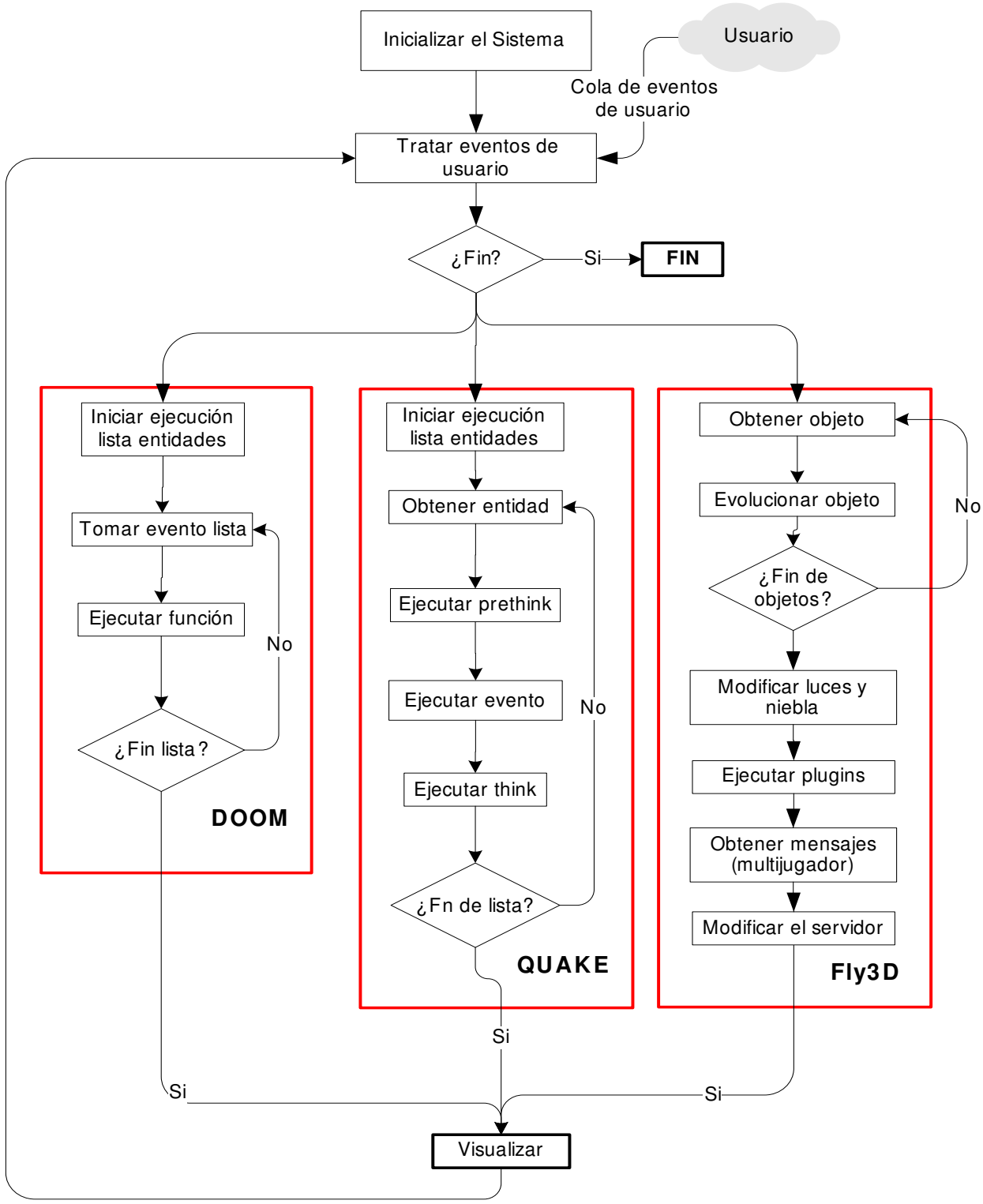

Figura 2.2: Bucle principal de *Doom*, *Quake* y *Fly3D*

Si

$$SRR < \nu_C \tag{2.3}$$

entonces

$$T_S + T_R > \frac{1}{SRR} \Rightarrow \frac{1}{T_S + T_R} = \nu_C \tag{2.4}$$

y

$$N_S = N_R \tag{2.5}$$

La ecuación 2.2 permite un rendimiento máximo, ya que se alcanza SRR. Las aplicaciones gráficas se comportan mejor en 2.2 que en 2.4. Usando simulación discreta se pretende reducir T_S, incrementando la probabilidad de operar bajo las condiciones de la ecuación 2.2.

2.5. Crítica

2.5.1. Esquema de Simulación Continua

Cada evolución del mundo siempre requiere una visualización completa (acoplo). Todos los eventos se evolucionan obligatoriamente a la máxima velocidad que puede proporcionar el hardware del ordenador, siguiendo un esquema continuo.

El sistema no es sensible a tiempos inferiores al periodo de muestreo. Los eventos se sincronizan artificialmente con el periodo de muestreo, no se ejecutan en el momento exacto en que deben ocurrir.

La simulación se produce recorriendo el grafo de escena por lo que los objetos de la aplicación gráfica tienen prioridades en función de su situación caprichosa dentro del grafo de escena. Esta prioridad es ficticia e intrínseca al propio modelo de simulación. Los eventos se ejecutan en el orden en que se accede a los objetos en el grafo de escena. No están ordenados por tiempo de ocurrencia.

Se recorren todas los objetos, independientemente de que estén activos o no, o tengan o no eventos pendientes. Esto supone ralentizar el proceso de simulación comprobando objetos innecesariamente. Es muy ineficiente tener que recorrer todo el grafo de escena cuando muchos de los objetos no generarán nunca ningún evento.

Posibilidad de simulaciones erróneas:

- **Ejecución desordenada de eventos**: supóngase la situación de la figura 2.3. La piedra y el proyectil se dirigen hacia la pared. En el instante t_0, la piedra S se encuentra en la posición S_{P0} y el proyectil M en M_{P0}. De acuerdo con sus velocidades, el proyectil impactará con el muro en el instante t_2 y la piedra en el instante t_3. Sea O es la posición de un objeto en el grafo de escena, la tabla 2.6 muestra las diferentes posibilidades de simulación correcta o incorrecta

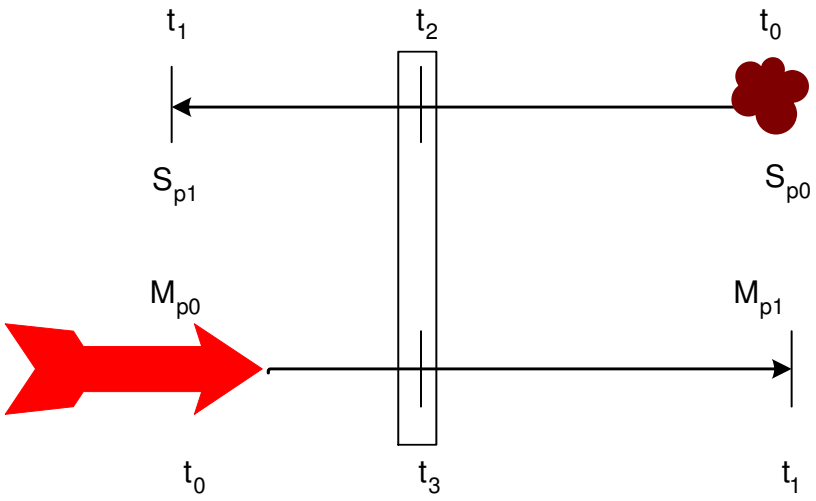

Figura 2.3: Ejemplo de orden de ejecución de eventos

Instante de impacto	**Orden en el grafo de escena**	**Acción**	**Correcto**
$t_2 < t_3$	$O_S < O_M$	La piedra rebota El proyectil destruye la pared	Si
$t_2 < t_3$	$O_S > O_M$	El proyectil destruye la pared La piedra rebota	No
$t_2 > t_3$	$O_S < O_M$	La piedra atraviesa la pared El misil destruye la pared	No
$t_2 > t_3$	$O_S > O_M$	El misil destruye la pared La piedra atraviesa la pared	Si

Tabla 2.6: Ejemplo de orden de ejecución de eventos

dependiendo de los instantes de tiempo en que ambos objetos colisionan con la pared y la posición de ambos en el grafo de escena.

- **Ejecución de eventos *cancelados***: en el mismo ciclo de simulación, un evento puede decidir si otros eventos deben ejecutarse o no. Si el evento que cancela los anteriores está situado posteriormente en el grafo de escena, el evento a cancelar se ejecuta. Por ejemplo, un misil debe destruir una pared en el ciclo de simulación actual. En el mismo ciclo, el jugador presiona un botón que desactiva el misil. Cuando se simula el misil, su estado es *destruir la pared*. Usando simulación continua, la pared se destruye, pues el nuevo estado del misil no se comprueba hasta el siguiente ciclo de simulación.

Todos los objetos del sistema son muestreados y animados a la misma velocidad, con independencia de su comportamiento. La frecuencia de muestreo es la misma para todos los objetos, independientemente de sus requisitos. Si el comportamiento de los objetos no cumple el teorema de Nyquist-Shannon, no se simularán adecuadamente, produciendo pérdida de eventos o colisiones no detectadas. Por ejemplo:

- Un objeto con comportamiento lento es sobremuestreado, desaprovechando potencia de cálculo.
- Un objeto con un comportamiento rápido es submuestreado (aliasing).

La frecuencia de muestreo es variable y no está acotada, puesto que depende de tópicos que varían durante el juego, como potencia disponible del ordenador, complejidad del mundo, otras tareas activas en el sistema, carga de red, de simulación o de visualización. Por tanto, la frecuencia de muestreo es variable y no está predefinida.

Si la potencia de cálculo es elevada, se calculan más imágenes de las que puede mostrar el periférico gráfico y que, por lo tanto, nunca son visualizadas. Es decir, se malgasta potencia de cálculo.

2.5.2. Acoplo de las Fases de Renderizado y Simulación

Una solución para mejorar los sistemas de simulación continuos fuertemente acoplados es utilizar **desacoplo**.

El modelo acoplado supone hacer una visualización para cada simulación. Con baja potencia de cálculo este esquema, no sólo es correcto, sino que es el más eficiente. Se aprovecha el recorrido del grafo de escena para visualizar la escena. El esquema acoplado es eficiente si la carga de simulación es grande respecto a la potencia de cálculo. Pero si la potencia de cálculo es elevada se visualiza innecesariamente.

Las tarjetas gráficas actuales suelen soportar frecuencias de refresco de pantalla superiores o iguales a 75Hz. Frecuencias superiores son redundantes ya que a partir de 72Hz, no se aprecia efecto de parpadeo. Pruebas realizadas sobre *Quake 3 v1.17* en equipos actuales empleando tarjetas de última generación obtienen tasas de refresco comprendidas entre los 130 y los 250 fps [TomHw].

Este comportamiento es ineficiente por cuanto pierde tiempo realizando visualizaciones que nunca serán volcadas a pantalla. En caso de llegar a generar 250 fps, se perdería el 70 % de la potencia de visualización, ya que por cada fotograma que es enviado a pantalla, se calculan 2.33 fotogramas que nunca son visualizados. La asociación de visualización y simulación obliga a que para poder realizar un nuevo ciclo de simulación, se debe realizar forzosamente una visualización, con independencia de que ésta vaya a ser representada o no.

Una mejora a ese esquema de funcionamiento consistiría en separar la fase de simulación de la fase de visualización, de forma que la frecuencia de refresco elegida se convirtiera en la frecuencia de muestreo exacta del sistema (se visualiza una vez por refresco de pantalla). El sistema sólo calcula el número de escenas como veces se refresca la pantalla. De esta forma, la potencia de cálculo liberada redundaría en una mayor cantidad de ciclos de simulación con lo que determinados comportamientos se calcularían de forma más precisa [Reynolds:2000].

Desacoplar el sistema no es la solución si el esquema de simulación sigue siendo continuo, ya que:

- Si se simula a la máxima velocidad, se está utilizando potencia de cálculo en simular estados intermedios, cuando sólo el último de dichos estados es el que interesa representar en pantalla (desajuste del periodo de muestreo).
- El grafo de escena se accede dos veces, una vez para simular y otra vez para simular.

Una práctica habitual en los primeros videojuegos, con una complejidad de escena baja, era simular y visualizar simultáneamente, recorriendo el grafo de escena sólo una vez [Bishop:1998]. Cuando la potencia de cálculo es escasa y/o el sistema a simular es muy complejo, la frecuencia de cuadro es inferior a la frecuencia de refresco de pantalla. Por ello, la política de preguntar a cada objeto si tiene algo que realizar aprovechando que se recorre el grafo de escena para realizar la visualización es una buena técnica ya que ahorra un recorrido del grafo de escena. La situación es distinta actualmente, pues las tasas de visualización son superiores a la frecuencia de refresco de la pantalla [TomHw].

Por otro lado, es muy ineficiente tener que recorrer todo el grafo de escena cuando muchos de los objetos no generarán nunca ningún evento. Sería más conveniente disponer de un gestor de eventos, de forma que sólo se recorran aquellos objetos que generen los eventos, evitando al resto de objetos. A pesar de todo, un modelo continuo desacoplado tiene ventajas respecto a un modelo continuo acoplado.

Ante los nuevos periféricos para interacción hombre-máquina y aplicaciones de simulación complejas (medicina virtual, periféricos con realimentación táctil,...), el esquema de simulación continua acoplada es insuficiente.

2.5.3. Motores de Videojuegos

El término motor de videojuegos se refiere muchas veces a motor de visualización. Los motores de videojuegos han mejorado la creación de videojuegos. Han supuesto un esfuerzo de estandarización, además de constituir algunos de estos juegos un trabajo más que notable. Pero, los motores de videojuegos, siguen utilizando un

esquema de simulación continua. Algunos motores permiten desacoplar las fases de simulación y visualización.

2.6. Propuesta de Mejora

La presente tesis propone cambiar el paradigma de simulación de las aplicaciones gráficas en tiempo real. El cambio propuesto supone:

- Desacoplar las fases de visualización y simulación del sistema.
- Utilizar un esquema de simulación discreta en lugar del actual sistema continuo. En este esquema los objetos del videojuego generan eventos cuando desean evolucionar en el sistema. Un sistema de simulación discreto permite soportar, además de simulación discreta, simulación continua e híbrida.

Con el cambio de paradigma, los eventos suceden en el instante en que deben suceder. Se ejecutan ordenados en el tiempo. Se dispone de un gestor de eventos, de forma que sólo se atienden aquellos objetos que generan eventos, evitando el resto de objetos. Los objetos se atienden según el momento de ocurrencia del evento. Todos los objetos tienen la misma prioridad, incluyendo el usuario. No se pierden eventos, porque no hay submuestreo. En este caso puede suceder que el sistema sea tan complejo que se ralentice para poder simular el sistema adecuadamente.

No existe una frecuencia de muestreo fija y común a todos los elementos del sistema: cada objeto tiene su propia frecuencia de muestreo (determinada por el programador). Los objetos con comportamientos rápidos se muestrean a la máxima frecuencia. Los objetos lentos se muestrean sólo cuando sea necesario, no sobrecargando el sistema con muestreos innecesarios. La frecuencia de muestreo es menos dependiente de la potencia de cálculo. Si la potencia de cálculo es insuficiente:

- El sistema va más lento pero no cambia la proporción de muestreo de los objetos. La simulación sigue siendo correcta.
- Si el sistema tiene mucha potencia de cálculo, en un esquema continuo se sobremuestrea y se desaprovecha la potencia de cálculo. En cambio, con un esquema de simulación discreta la simulación sólo consume la potencia de cálculo estrictamente necesaria.
- Independientemente de la potencia de cálculo, con un esquema de simulación discreta se simula lo necesario. Los objetos no son sobremuestreados ni submuestreados.

Se ha elegido *Fly3D* como motor de videojuegos donde realizar las mejoras propuestas en el esquema de simulación. La elección de *Fly3D* está justificada por las siguientes razones:

- Es un motor de videojuegos altamente estructurado y modularizado. La implementación de la simulación y visualización se realiza en funciones específicas de cada objeto de la aplicación.
- Código muy documentado. Se han publicados dos libros dedicados íntegramente a su código [Watt:2001] [Watt:2003].
- Está orientado a plugins, por lo que se pueden crear distintas aplicaciones gráficas o videojuegos sin necesidad de tener diferentes copias del núcleo para cada aplicación gráfica.
- Las fases de visualización y simulación están acopladas a nivel de sistema, lo que evita tener que separarlas previamente al cambio del esquema acoplado de simulación por el desacoplado.
- Dispone de múltiples herramientas que facilitan la tarea de creación de videojuegos.

En el apéndice D se estudia el motor *Fly3D* con detalle, haciendo especial hincapié en la arquitectura de éste. Se estudia el bucle principal de Fly3D y los procesos de visualización y simulación, elementos que serán modificados con el cambio de paradigma de simulación.

El trabajo a desarrollar en la tesis sigue las siguientes etapas:

1. Crear el simulador de eventos discretos DESK desde cero (apartado 3.1 del capítulo 3). Este simulador debe tener una prestaciones lo más óptimas posibles y comparables con otros simuladores de eventos discretos. La decisión de crear un simulador propio se explica en el apartado 3.1.1 del capítulo 3.
2. Modificar el simulador de eventos discretos con el objeto de crear el simulador basado en web JDESK (apartado 3.2 del capítulo 3). Se abre una rama de investigación no desarrollada totalmente en esta tesis con el objeto de distribuir el simulador.
3. Adaptar DESK a un núcleo de aplicaciones gráficas en tiempo real, obteniendo GDESK (apartado 3.3 del capítulo 3).
4. Integrar GDESK en Fly3D, obteniendo el núcleo de videojuegos discreto DFly3D (capítulo 4).

5. Evaluar las prestaciones de DESK, comparándolo con SMPL (apartado 5.1.1 del capítulo 5).

6. Evaluar las prestaciones de JDESK, comparándolo con DESK (apartado 5.1.2 del capítulo 5).

7. Comprobar los resultados del cambio de paradigma de simulación, comparando DFLy3D y Fly3D (apartado 5.2 del capítulo 5).

Desacoplar simulación y visualización supone que la frecuencia de cuadro se adapta a la frecuencia de refresco de pantalla y a la frecuencia que el ojo humano es capaz de apreciar. El instante en que se lanza la visualización debe ser lo más cercano posible al instante de refresco de pantalla. Para ello se debe conocer el tiempo necesario para visualizar la escena. Como este tiempo no es posible conocerlo, debe predecirse.

Existe gran cantidad de información sobre predicción de tiempo de visualización. Mucha de ella se refiere únicamente a la técnica de trazado de rayos, no utilizada en videojuegos por su elevado coste temporal ([Aronov:2002] [Cleary:1988] [MacDonald:1990] [Subramanian:1991] [Whang:1995] [Reinhard:1996] [Reinhard:1998]). Otros trabajos más recientes, como [Wimmer:2003] tratan el tema de la predicción del tiempo de visualización en tiempo real. Una línea de investigación futura es la inclusión de estos trabajos y otros similares al núcleo de simulación desarrollado.

Capítulo 3

Núcleo de Simulación de Eventos Discretos

En este capítulo se presentan las tres versiones del simulador de eventos discreto realizadas durante la tesis:

1. DESK: simulador generalista de eventos discreto.
2. JDESK: simulador de eventos discreto basado en web.
3. GDESK: simulador de eventos discreto como núcleo de aplicaciones gráficas en tiempo real.

Se estudia la estructura y funcionalidad de cada uno de los simuladores y se especifican las aportaciones que han supuesto cada uno.

3.1. DESK: Simulador de Eventos Discretos

3.1.1. Motivación

El primer paso en el desarrollo de la tesis es crear un simulador de eventos discreto que pueda servir como base para la creación del núcleo de simulación de aplicaciones gráficas en tiempo real. Se ha creado previamente un simulador de eventos discretos completo, con el objetivo de poder comparar sus prestaciones con las de otros simuladores (velocidad y flexibilidad). De esta forma se parte de un simulador con las mejores prestaciones posibles. Este simulador se especializa posteriormente para crear el núcleo de simulación de aplicaciones gráficas. Este apartado se centra en la creación del simulador de eventos discretos DESK.

Las **conclusiones** obtenidas del estudio realizado sobre los simuladores de eventos discretos son:

- Los simuladores de eventos discretos que pueden ser hipotéticamente apropiados para los objetivos de esta tesis por su flexibilidad y velocidad (la integración del simulador en una aplicación gráfica) son lenguajes de código cerrado.
- Los simuladores de eventos discretos de código abierto suelen ser antiguos, se han dejado de mantener o están fuera de la corriente actual de la informática, por lo que su empleo conlleva migrarlos a entornos de desarrollo actuales. Los simuladores actuales suelen estar especializados en la simulación de un tipo de sistema concreto o en un campo determinado.
- Hay tres formas de programación de simuladores de eventos discretos: como bibliotecas, como lenguajes propios o como interfaces gráficas para la creación del modelo con acceso a un motor de simulación automático.

Para una aplicación gráfica en tiempo real, los requerimientos del núcleo de simulación son:

1. Debe ser una biblioteca e implementado en un lenguaje de programación generalista y utilizado habitualmente en la implementación de aplicaciones gráficas.
2. El código debe ser abierto.

Justificación:

1. **Biblioteca de un lenguaje generalista** (tabla 3.1): no puede ser una lenguaje de programación específico porque estos tienen una semántica insuficiente para los requisitos de una aplicación gráfica. Los simuladores implementados como interfaces gráficas son claramente inadecuados. El lenguaje debe ser el mismo que el utilizado en la mayoría de los núcleos de aplicaciones gráficas. Debe soportar: estructuras dinámicas (como heaps o grafos de escena), Opengl, DirectX, nuevas tecnologías de ingeniería del software,... y debe ser ampliamente utilizado por la comunidad científica.
2. **Código abierto**: es necesario conocer como está implementado el simulador para saber si lo está de la forma más eficiente posible. El código debe modificarse, pues la aplicación gráfica tiene unos requisitos que no se adaptan a los simuladores de eventos. Se deben eliminar ciertas características para incrementar su velocidad y añadir nuevos elementos.

Por estas razones se ha impuesto la necesidad de crear un núcleo de simulación de eventos discretos propio desde cero: **DESK** (**D**iscrete **E**vents **S**imulation **K**ernel) [Garcia:1997] [Garcia:2000].

Tipo	Ventajas	Inconvenientes
Biblioteca	Uso del compilador y linkador del lenguaje. Flexibilidad de E/S de datos. Integración de otras librerías y funciones en la simulación. No es necesario aprender un nuevo lenguaje. Portabilidad.	Difícil verificación y detección de errores. Tiempos de simulación menores. Tiempo de desarrollo largo.
Lenguaje Propio	Verificación y detección de errores más sencilla. Menor tiempo de desarrollo si se domina el lenguaje. Sentencias específicas para trabajar con la simulación de un sistema.	Obligatoriedad de conocer el lenguaje específico. Limitado a las posibilidades del lenguaje. La nomenclatura de los elementos se adapta a la simulación.

Tabla 3.1: Biblioteca vs. lenguaje propio

3.1.2. Antecedentes

La creación de DESK se basó en dos simuladores radicalmente diferentes en cuanto a sus prestación y forma de definir el modelo: SMPL y QNAP. Estos simuladores se utilizan principalmente en el campo de la docencia. DESK aúna las ventajas de ambos, minimizando los inconvenientes. La tabla 3.2 muestra una comparativa de estos simuladores.

SMPL (Simple Portable Simulation Library) [MacDougal:1980] [MacDougal:1987] [Smpl] es una biblioteca de C [Schildt:2000], lo que permite utilizar todo el potencial de este lenguaje en la implementación de los modelos. Destaca por su velocidad de simulación, pero su gran inconveniente es que la definición del modelo es complicada y trabajosa, pues supone definir el modelo como la secuencia de todos los posibles eventos del sistema. Deben definirse los algoritmos de planificación, el control de tiempos,... Modificar y depurar el modelo es complicado. No permite modelar cualquier sistema: existe un valor límite del número de clientes y de Estaciones de Servicio (ES) en el modelo. SMPL permite modelar cualquier comportamiento del sistema, pero con un coste de implementación muy elevado

QNAP [Veran:1984] [Potier:1984] [Qnap:1990] [Qnap] es un simulador implementado como un lenguaje propio. Definir un modelo en este simulador es sencillo y rápido, pues únicamente deben definirse las ES del modelo, sus propiedades y su

Característica	**SMPL**	**QNAP**
Lenguaje de implementación	Extensión de C (biblioteca)	Lenguaje propio
Definición del modelo	Definición de eventos y relaciones	Definición de estaciones de servicio y su interconexión
Creación y modificación del modelo	Difícil	Fácil
Limitaciones del modelo	Número de clientes, estaciones de servicio,...	Sin limitaciones
Utilidad	Implementación final del modelo	Depuración y validación del modelo
Velocidad	Rápido	4 o 6 veces mas lento que SMPL
Prototipado del modelo	Lento	Rápido
Depuración del modelo	Fácil	Difícil

Tabla 3.2: SMPL vs. QNAP

interconexión. Modificar y depurar el modelo es rápido y sencillo. Los algoritmos de planificación y el control de tiempos los realiza el simulador automáticamente. No impone limitaciones en el modelo a simular. Por contra, el tiempo de simulación es entre 4 y 6 veces mayor que SMPL.

Comparando ambos simuladores, se puede destacar una característica básica, la diferencia de velocidad. SMPL es bastante más rápido que QNAP. El gran inconveniente de SMPL es su dificultad de uso y su limitación en las funciones que permite realizar. Implementar modelos en SMPL supone definir todos y cada uno de los eventos que van a tener lugar en el sistema, llevar un control de tiempos,... Por tanto, se debe obtener un simulador que manteniendo la forma de definición del modelo de QNAP ofrezca toda la potencia y velocidad de SMPL.

3.1.3. Objetivos

El objetivo primordial de DESK es incrementar la velocidad y flexibilidad de las simulaciones, para ello:

1. El simulador debe ser lo más flexible posible, no imponiendo ninguna restricción al sistema a simular. La única limitación impuesta será la de memoria, debida al hardware del ordenador en el que se realice la simulación. Para conseguir este objetivo se utiliza gestión dinámica de memoria, mejorada con la

utilización de *pools* de elementos.

2. La definición del modelo se realiza definiendo las ES que componen el modelo del sistema y sus características. A partir de esta información, el simulador calcula los eventos necesarios, sin que el usuario conozca estos eventos.

3. Los parámetros del sistema deben poder variar dinámicamente, para obtener resultados de la simulación con distintas configuraciones del mismo sistema.

4. El simulador debe permitir utilizar diversos algoritmos de encaminamiento y políticas de planificación. No debe limitarse a los algoritmos estándar, sino que debe permitir comportamientos caprichosos definidos por el usuario.

5. La topología del sistema puede variar dinámicamente, permitiendo modificar o añadir nuevos elementos.

6. Nueva concepción de eventos: se considera un evento como el cambio de comportamiento o estado de un cliente. Un evento no necesita ser una entidad propia ni disponer de una estructura de datos que lo soporte.

7. Control automático de eventos. Los eventos son transparentes al usuario.

8. Estos objetivos deben conseguirse minimizando el tiempo de ejecución.

3.1.4. Lenguaje de Implementación

QNAP está implementado como un lenguaje propio, SMPL como una biblioteca de C. Ambas formas de implementación del modelo tienen ventajas e inconvenientes (tabla 3.1).

DESK se ha implementado como una extensión de C++ [Stroustrup:2001], pues ciertas ventajas, como la portabilidad o la inclusión de funciones del propio lenguaje, se han considerado importantes. Además de ser el lenguaje de programación más utilizado en la actualidad [Hernandez:2002], posee características de programación muy avanzadas [Schildt:2001] como portabilidad, orientación a objetos, soporte a polimorfismo, genericidad o abstracción de datos.

Estas características le hacen apropiado, no sólo para la implementación de DESK, sino para que posteriormente el usuario implemente los modelos de simulación.

3.1.5. Entidades Básicas

Un modelo en DESK está compuesto de dos entidades básicas: ES y clientes [Banks:2001]. Estas son las estructuras con las que el programador construye los modelos de simulación. Otras estructuras del simulador son el pool de clientes y el gestor de eventos. Estas estructuras son internas y no accesibles al usuario.

3.1.5.1. Clientes

Los clientes son elementos pasivos que viajan a través del sistema, cambiando su estado y el de las ES que atraviesan. Las características de los clientes pueden variar dinámicamente. Contienen información que puede ser modificada por las ES. Los clientes se pueden generar automáticamente dentro del sistema o bien el programador puede crearlos explícitamente. Los eventos del sistema dejan de tener una estructura de datos que los soporte en DESK para pasar a ser el cambio de estado de un cliente dentro del sistema.

Los clientes tienen asociados los siguientes atributos: identificador, prioridad y clase. Además, el cliente contiene información de soporte a las estructuras del simulador:

- ES donde se encuentra recibiendo servicio.
- Tiempo de servicio que demanda el cliente de la ES donde se encuentra.
- Información para contabilidad.
- Relación padre-hijo.
- Información de evento. Un evento no tiene una entidad física que lo soporte. Para modelar un evento se usa el propio cliente. La información necesaria para un evento es el tiempo en el que se va a producir el próximo evento asociado al cliente.
- Enlaces a estructuras de datos dinámicas: los clientes siempre residen en alguna estructura del simulador. A partir del cliente se conoce directamente si está recibiendo servicio en alguna ES y en cual, o si está esperando a ser servido.

3.1.5.2. Estaciones de Servicio

Una ES modela una parte del sistema real que proporciona un determinado servicio a los clientes. Toda la información del modelo la contienen las ES. Las ES son las estructuras dinámicas del simulador (junto con el gestor de eventos). A partir de la información que contienen las ES se extraen los resultados de la simulación. Las ES pueden cambiar sus características y funcionalidad dinámicamente.

Los parámetros que definen una ES son: nombre, tipo (servidor, fuente y recurso/semáforo), número de servidores o recursos, tiempo de servicio a clientes, política de planificación o servicio de clientes y topología del sistema (encaminamiento de los clientes a la salida de la ES).

Las funciones principales de la ES son: contabilidad, control de los clientes dentro de la ES (esperando o recibiendo servicio), gestión de recursos y gestión de prioridades y clases de clientes.

Una ES está compuesta por un número determinado de servidores y las estructuras de datos de gestión de clientes en servicio y esperando a ser servidos.

Un modelo se define mediante las ES que lo componen, sus propiedades y cómo se conectan unas con otras (topología del sistema).

DESK define tres tipos de ES:

- **Fuentes**: tienen como objetivo principal generar clientes dinámicamente. La cadencia con la que la fuente genera clientes y las propiedades de estos, los marca el programador al definir la fuente. Una vez los clientes se crean, dejan la fuente para encaminarse a otra ES.
- **Servidores**: están compuestos por las estructuras de gestión de clientes y un conjunto de servidores. Cuando un cliente entra en una ES, si no hay un servidor libre, espera hasta que algún servidor queda libre. Si el cliente entra en un servidor recibe servicio durante un tiempo determinado.
- **Recursos**: una ES tipo recurso en DESK aúna los conceptos de recurso y semáforo. Si a la llegada del cliente, el número de recursos demandados no está disponible, el cliente espera hasta que se liberen recursos suficientes. Si los recursos demandados por el cliente se le pueden asignar, el cliente continúa fluyendo libremente por el sistema. El cliente puede liberar los recursos en cualquier momento.

3.1.6. Estructuras de Datos Dinámicas

Uno de los objetivos básicos del simulador DESK es no imponer restricciones al sistema a simular. Se consigue mediante:

1. Utilización de estructuras dinámicas para la gestión del simulador.
2. Utilización de funciones para definir el comportamiento del sistema.

Las estructuras de datos del simulador son de dos tipos:

- Estructuras FIFO: gestor de ES del sistema y pool de clientes.
- Estructuras ordenadas según diferentes criterios: gestor de eventos del sistema (ordenado por tiempo de ocurrencia del suceso) y estructuras dinámicas incluidas en cada ES :

Estructura	**Inserción ordenada**	**Inserción no ordenada**	**Eliminar cabeza**	**Eliminar (puntero)**
Lista simplemente enlazada	O(N)	O(1)	O(1)	O(N)
Lista doblemente enlazada	O(N)	O(1)	O(1)	O(1)
Heap	O(log N)	O(log N)	O(log N)	O(log N)

Tabla 3.3: Costes de estructuras dinámicas ordenadas [Sedgewick:1998] [Kingston:1990]

- Clientes que actualmente están recibiendo servicio en la ES (FIFO).
- Clientes que están esperando a recibir servicio en la ES (ordenada según criterio definido por el usuario).
- Clientes que tienen reservados recursos pertenecientes a la ES (FIFO).

Las operaciones a realizar sobre las estructuras de datos del simulador son:

- Estructuras dinámicas ordenadas por diferentes criterios: inserción ordenada, extracción de la cabeza y extracción de un elemento cualquiera (conocido y señalado por un puntero).
- Estructuras dinámicas FIFO: inserción y extracción de la cabeza.

Según estas operaciones y sus costes (tabla 3.3), las estructuras de datos elegidas son:

- **Estructuras dinámicas ordenadas por diferentes criterios**: los costes temporales menores los proporciona el heap (tabla 3.3). La implementación habitual de un heap se realiza sobre un vector, pero esto supone imponer restricciones en el número de clientes del sistema, lo que impide alcanzar uno de los objetivos del simulador (no imponer restricciones en el sistema). Si bien es cierto que la utilización de C++ permite reasignar memoria dinámicamente a vectores, no imponiendo restricciones, el proceso es muy costoso temporalmente. Como solución de compromiso, se decidió utilizar una estructura dinámica y utilizar un pool de elementos. Se ha elegido como estructura de almacenamiento de datos una lista o cola doblemente enlazada (cola básica del simulador), con un puntero a cabeza y a cola, tanto para estructuras ordenadas como no ordenadas.

- **Estructuras dinámicas FIFO**: puede utilizarse una lista simplemente enlazada o doblemente enlazada, pues el coste de las operaciones en ambos casos es constante. Estas estructuras de datos se utilizan en el momento de inicializar el simulador, por lo que la sobrecarga por el hecho de utilizar una lista simplemente enlazada o doblemente enlazada es mínima. Se ha elegido una lista doblemente enlazada por homogeneidad.

Para evitar llamadas innecesarias al gestor de memoria dinámica, para crear y destruir los nodos de las listas, los propios elementos del sistema (clientes y ES) serán los nodos de las listas (contienen los enlaces a las listas donde pueden estar insertados).

3.1.6.1. Gestor de Estaciones de Servicio

El gestor de ES sirve para inicializar las ES del sistema y extraer resultados de la simulación. Las única operación que se realiza sobre esta estructura es la inserción no ordenada: se insertan las ES siguiendo el orden en que el usuario las crea. Durante la simulación pueden crearse ES dinámicamente, que se irán insertando en esta lista.

Las ES únicamente pueden estar incluidas en esta lista, por lo que contienen los punteros de enlace a la lista para evitar crear y destruir nodos innecesariamente.

3.1.6.2. Gestión de Clientes en las Estaciones de Servicio

Las ES contienen estructuras para gestionar los clientes que solicitan su servicio, para ello almacenan los clientes que actualmente: están recibiendo servicio, están esperando a recibir servicio y los que tienen reservados recursos pertenecientes a la ES.

El usuario define la política de servicio o planificación de clientes para cada ES del sistema. Mientras los clientes lleguen a la ES y encuentren servidores libres, la política de servicio no se tienen en cuenta. La política de servicio afecta al orden en que los clientes se insertan en la cola de espera. La extracción de clientes siempre se realiza siguiendo el mismo orden, por lo que la política de servicio la define el orden en que se insertan los clientes. Ciertas políticas de servicio se definen en función de la prioridad asociada a los clientes.

El algoritmo de inserción en cola lo define el usuario, pero el de extracción es fijo. Si la política de planificación no utiliza prioridades, todos los clientes se insertan en la misma cola. Independientemente de las prioridades de los clientes, los algoritmos de servicio pueden definirse prioritarios o no prioritarios. Para definir prioridades son necesarias ambas cosas: que se definan los clientes con prioridades y que el algoritmo de inserción en cola de espera contemple prioridades.

Esta estructura se usa en las colas de espera y servicio de las ES:

- **Cola de espera**: incluye los clientes que están esperando a ser atendidos por la ES. Cuando un cliente entra en la ES y los servidores están ocupados o no hay recursos disponibles para asignarle, se queda esperando a ser atendido en la cola de espera. También se insertan en esta cola los clientes que habiendo estado recibiendo servicio han sido expulsados de ejecución por la llegada de un cliente más prioritario (política de servicio expulsiva). Cuando un servidor de la ES queda libre se extrae de esta cola el siguiente cliente a servirse.

- **Cola de servicio**: en las políticas de servicio expulsivas es necesario mantener información sobre los clientes que están recibiendo servicio en cada uno de los servidores de la ES, para poder saber si son mas o menos prioritarios que un cliente recién llegado, y en el caso de ser más prioritario, sacar de ejecución el cliente que actualmente recibe servicio.

Cola de Clientes en Recursos

Un caso particular de cola del simulador es la cola de clientes con recursos reservados. Esta cola mantiene constancia de los clientes que tienen asignados recursos. Cuando un cliente reserva recursos de una ES sigue fluyendo por el sistema, puede reservar tantos recursos de distintas ES tipo recurso como sea necesario, por lo que un mismo cliente puede estar en varias colas de recursos, por ello no se pueden incluir en el cliente los punteros a las colas de recursos de las ES. Un cliente sólo puede estar en cola de espera de un único recurso.

Cada vez que a un cliente se le asignan recursos de una ES se crea dinámicamente un nodo que se inserta en la cola de recursos de la ES. Después, el cliente sale de la ES y continúa fluyendo libremente por el sistema. Cuando un cliente libera todos los recursos que tenía asignados en la ES se elimina el nodo de la cola y se destruye.

Esta cola sirve para realizar funciones de contabilidad de recursos y para saber si un cliente que libera un recurso lo tenía realmente asignado. Esta cola se utiliza en las ES tipo recurso.

La cola de clientes en recursos es un caso especial de cola. Es simplemente una cola básica pero cuyos nodos no son elementos del sistema, sino nodos creados a tal efecto. La utilización de recursos supone ralentizar el sistema, pero se ofrece la posibilidad de poder utilizarlos.

3.1.6.3. Gestor de Eventos del Sistema

El gestor de eventos es la estructura principal del simulador. Modela la dinámica del sistema, junto con las ES. Es la estructura de gestión de la simulación. La estructura elegida para gestionar los eventos del sistema es una lista de eventos doblemente enlazada. Las operaciones que se realizarán sobre esta estructura son:

- Insertar un evento ordenadamente (según el tiempo en que debe suceder el evento).
- Extraer el evento de menor tiempo.
- Extraer un evento cualquiera, de cualquier posición. Este elemento se conoce (está apuntado por un puntero). Sirve para tratar preempciones.

Los eventos en DESK dejan de tener una estructura de datos que los soporte, para pasar a considerarse el cambio de estado o comportamiento de un cliente dentro del sistema. Por ello, un evento no es más que un cliente con cierta información. El único evento posible en el sistema es que un cliente que actualmente está recibiendo servicio, deje de hacerlo porque se ha cumplido su tiempo de servicio. La única información necesaria para un evento será la ES donde se encuentra actualmente el cliente (de la cual quiere salir) y el tiempo en el que debe producirse el evento.

La cola de eventos realmente contiene clientes.

Los clientes siempre residen en alguna cola del simulador. Un cliente puede encontrarse en una de las siguientes situaciones:

- Esperando a recibir servicio en una ES: insertado en la cola de espera de la ES correspondiente.
- Recibiendo servicio en una ES. En este caso el cliente estará insertado en dos colas diferentes: la cola de servicio de la ES y la cola de eventos del sistema.

El número máximo de colas en las que un cliente puede estar insertado simultáneamente es dos. Para mejorar la gestión de las colas del simulador y agilizar el proceso de inserción o eliminación de clientes, un cliente contiene los punteros a la cola del gestor de eventos y a la ES donde está insertado, de modo que dado un cliente se conoce directamente su posición en la cola o colas en que se encuentra y si está recibiendo servicio en alguna ES y en cual, o si está esperando.

El principal objetivo de la cola de eventos es sacar de ejecución en el momento indicado a los clientes que están recibiendo servicio en las ES.

Funciones de la cola de eventos:

- Extraer el cliente (evento) de cabeza y llamar a la ES donde dicho cliente está recibiendo servicio para indicarle que su tiempo ha expirado.
- Insertar ordenadamente un cliente cuando una ES lo solicita. La inserción se realiza considerando el tiempo en que debe finalizar el servicio del cliente en la ES. El único evento posible es la salida de un cliente de la ES donde se encuentra.

Los eventos se ordenan en función del instante de tiempo en el que van a ocurrir, de forma que en la cabeza de la cola se encuentra el primer evento que debe producirse (tiempo menor).

3.1.6.4. Pool de Clientes

Cuando un cliente entra al sistema, se llama al Gestor de Memoria Dinámica (GMD) para reservar la memoria necesaria para el objeto. Cuando deja el sistema, se llama de nuevo al GMD para que libere la memoria del cliente. Esta situación no sería un problema si no se tratase de minimizar el tiempo de simulación

Para evitar pérdidas de tiempo innecesarias, DESK reutiliza los clientes que salen del sistema para crear los nuevos clientes. Los clientes que abandonan el sistema, en vez de destruirse (llamando al GMD), se insertan en una estructura del simulador denominada pool de clientes.

El pool de clientes es una cola básica de clientes no ordenada, por lo que los costes de inserción y extracción son constantes.

Cada vez que se necesita un nuevo cliente en el sistema, se comprueba si hay un cliente disponible en el pool y si lo hay, se extrae del pool y se introduce en el sistema. Si no lo hay, se llama al GMD para que lo cree dinámicamente. El pool de clientes es la estructura del simulador encargada de almacenar los clientes que ya no están dentro del sistema (figura 3.2). Los clientes se crean durante la simulación pero no se destruyen.

La figura 3.1 muestra la evolución de los clientes en el sistema para un ejemplo de simulación. Inicialmente todos los clientes del sistema se crean mediante llamadas al GMD, pues no hay clientes disponibles en el pool. Cuando desciende el número de clientes en el sistema el número de clientes generados permanece constante, pues los clientes que no están en el sistema están en el pool. Mientras el número de clientes en el sistema no alcance un nuevo máximo no se realizan nuevas llamadas al GMD. Los clientes en el sistema varían constantemente, pero los clientes generados sólo varían cuando se sobrepasa el último máximo de clientes generados. Por tanto, el número de clientes en el sistema permanece constante durante la mayor parte de la simulación . Con este método se reduce considerablemente el numero de llamadas al GMD, lo que reduce el tiempo de simulación. El coste de las operaciones sobre el pool es constante.

Sea:

C_M número de clientes en el modelo. Valor instantáneo C_{M_i}.

C_S número de clientes en el simulador (clientes en el modelo y en el pool). Valor instantáneo C_{S_i}.

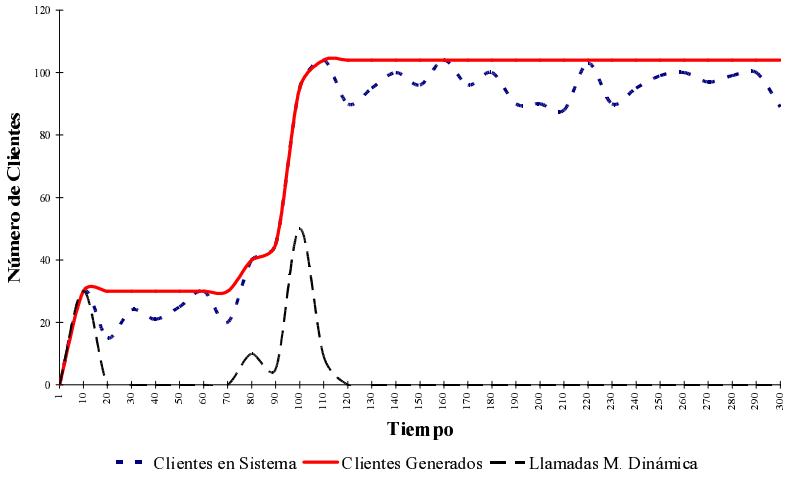

Figura 3.1: Llamadas al gestor de memoria dinámica en DESK

C_L número de clientes en el pool. Valor instantáneo C_{L_i}.

C_T número de clientes generados por el GMD.

T_N unidades de tiempo necesarias para que el GMD realice una operación *new*.

T_R unidades de tiempo necesarias para que el GMD realice una operación *release*.

N número de entradas y salidas de clientes del sistema durante la simulación.

N_I número de clientes que entran en el sistema.

N_O número de clientes que salen del sistema.

L carga del programa debido a la creación y destrucción de clientes.

En el inicio de la simulación todos los clientes se crean dinámicamente, cuando C_M decrece, C_S permanece constante. Si C_S no alcanza un máximo local no se llama al GMD (ecuación 3.1).

$$C_T = C_S = C_M + C_L \tag{3.1}$$

C_M y C_L pueden variar durante la simulación, pero C_T y C_S permanecen constantes mientras la cantidad de clientes no alcanza un nuevo máximo. Cuando se produce un nuevo máximo local (ecuaciones 3.2, 3.3):

$$C_T = C_S = C_M \tag{3.2}$$

$$C_L = 0 \tag{3.3}$$

L en un simulador convencional es (ecuación 3.4):

$$L = \sum_{i=1}^{N_I} T_{N_i} + \sum_{i=1}^{N_O} T_{R_i} \tag{3.4}$$

En DESK es (ecuación 3.5):

$$L = \sum_{i=1}^{C_{T_{max}}} N_i \tag{3.5}$$

El número de llamadas al GMD se reduce considerablemente, por lo que decrece el coste de la simulación, aunque la cantidad de memoria utilizada aumente (coste totalmente asumible con la tecnología actual).

Cuando se trabaja con sistemas con restricciones de memoria se puede limitar la cantidad de memoria utilizada por el pool, con la consiguiente penalización del coste de simulación.

3.1.7. Modelado de Sistemas

El modelo de un sistema en DESK se crea, básicamente, definiendo las ES que componen el modelo. El comportamiento del sistema es el conjunto de comportamientos de todas las ES del sistema.

El comportamiento de la ES se define mediante una serie de parámetros. Estos parámetros varían dependiendo del tipo de ES. En la tabla 3.4 se muestran los parámetros de la ES, su tipo y cuales de ellos están presentes en la declaración de cada ES. Entre estos parámetros hay:

- **Constantes**: el valor del parámetro debe obligatoriamente seleccionarse entre una serie de valores constantes.
- **Variables**: permite cualquier valor del tipo correcto.
- **Funciones**: se debe pasar como parámetro una función previamente definida. DESK contiene una biblioteca de funciones y patrones de funciones. El comportamiento de las ES se define en su mayor parte por funciones. El usuario puede implementar las funciones o bien utilizar funciones de biblioteca. Para

Parámetros de la ES	Servidor	Recurso	Fuente	Tipo de Parámetro
Nombre	✓	✓	✓	String (variable)
Número de servidores o recursos	✓	✓		Valor (variable)
Política de servicio preemptiva	✓			Constante
Tiempo de servicio demandado por los clientes	✓	✓	✓	Función
Algoritmo de planificación (inserción en cola de espera de la ES)	✓	✓		Función
Algoritmo de planificación (reinserción en cola de espera de la ES)	✓			Función
Número de recursos demandados por el cliente		✓		Función
Algoritmo de encaminamiento de clientes a la salida de la ES	✓	✓	✓	Función
Función de usuario	✓	✓		Función

Tabla 3.4: Parámetros de definición de las ES en DESK

usuarios inexpertos o modelos con comportamientos típicos, es recomendable usar funciones de biblioteca. Cuando el sistema tiene un comportamiento inusual se debe implementar este comportamiento mediante funciones.

Las funciones de comportamiento permiten variar dinámicamente el sistema: su topología, su estructura o sus características. También son estas funciones las que permiten hacer simulaciones en tiempo real, incluir multimedia, alarmas,...

Las funciones de comportamiento de las ES son:

- **Función de tiempo de servicio**: obtiene el tiempo que un cliente determinado debe recibir servicio en la ES actual o la cadencia de creación de clientes en las fuentes. El tiempo de servicio asignado podría depender de su estado, de valores aleatorios o de distribuciones temporales (las principales funciones de distribución temporal están implementadas en el simulador como funciones de biblioteca: uniforme, exponencial, erlang, hiperexponencial y normal). Se ejecuta cuando un cliente entra en el servidor o cuando hay que crear un nuevo cliente en una fuente.

- **Función de planificación de servicio de clientes**: permite insertar en la cola de espera de la ES un cliente que no puede servirse porque todos los servidores están ocupados. Las principales políticas de servicio están implementadas en el simulador como funciones de biblioteca: LIFO, FIFO, según prioridades y las versiones preemptivas de estas políticas.

- **Función de planificación de servicio de clientes (reinserción)**: (sólo para ES definidas como preemptivas) con la llegada de un cliente más prioritario, el cliente que está recibiendo servicio actualmente deja de recibir servicio y se reinserta en la cola de espera. El hecho de diferenciar entre inserción y reinserción permite dotar de una mayor flexibilidad al simulador.

- **Función de encaminamiento**: define la ES donde transitará el cliente cuando abandone la ES actual. El conjunto de las funciones de encaminamiento define la topología del sistema. Usar funciones para definir la topología del sistema permite variarla dinámicamente o que dependa de cualquier condición. También podría utilizarse con otros propósitos, como modificar los atributos de los clientes conforme atraviesan las ES. Si se están utilizando recursos, puede utilizarse esta función para liberarlos

- **Función de número de recursos**: permite obtener el número de recursos que solicita un cliente cuando llega a una ES tipo recurso.

- **Función de usuario**: no es necesaria para el funcionamiento del simulador, pero, se ha incluido en DESK, pues dota al simulador de una gran flexibilidad. Se ejecuta cuando un cliente entra en servicio o se le asigna un recurso. Puede utilizarse para hacer trazas, para simulaciones en tiempo real o crear hijos, en general, para cualquier comportamiento atípico.

El coste de las funciones incluidas en el repositorio de funciones de comportamiento es constante. Si el usuario implementa sus propias funciones de comportamiento, pueden tener un coste no constante, lo que ralentiza el proceso de simulación.

3.1.8. Dinámica del Sistema

La dinámica del sistema (figura 3.2) la modelan las ES que componen el modelo y la cola de eventos. La cola de eventos está compuesta por clientes (eventos del sistema). En DESK sólo hay un posible evento: un cliente debe abandonar la ES donde está recibiendo servicio porque ha finalizado su tiempo de servicio. La cola de eventos está ordenada por el tiempo de finalización de servicio de los clientes. La simulación comienza porque un cliente entra en una ES (figura 3.3 a). Si la ES tiene un servidor libre, comienza a recibir servicio (figura 3.3 b). Se genera un evento de salida del cliente al cabo del tiempo de servicio del cliente (figura 3.3 c).

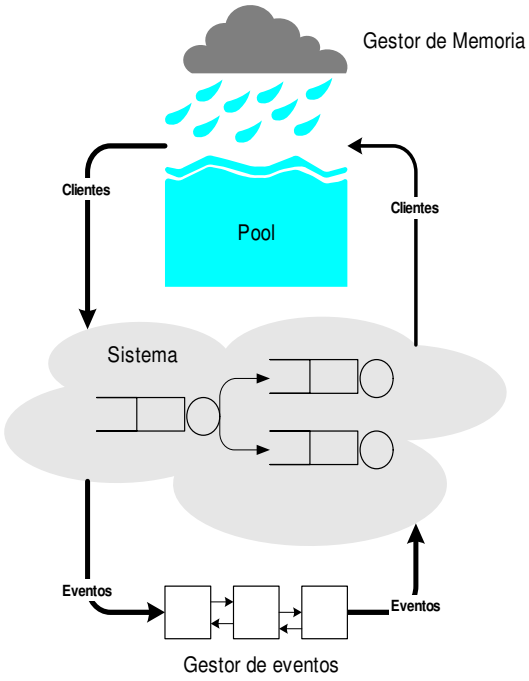

Figura 3.2: Dinámica del sistema en DESK

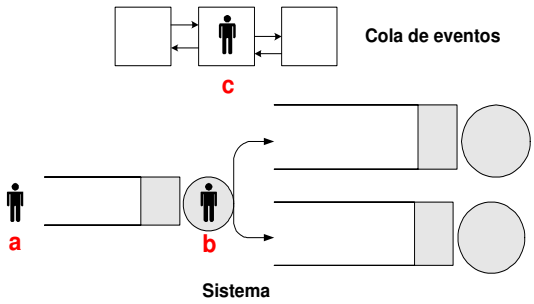

Figura 3.3: El cliente entra en la ES en DESK

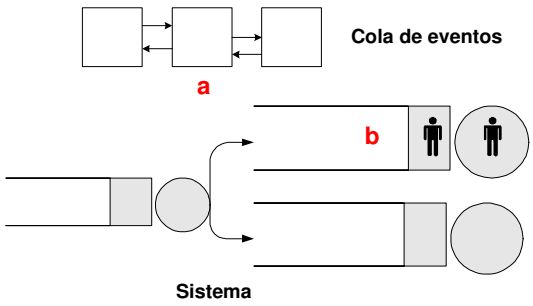

Figura 3.4: El cliente sale de la ES en DESK

Hasta este momento la simulación ha estado controlada por la ES donde ha entrado el cliente y ahora el control pasa a la cola de eventos. La cola de eventos ejecuta el siguiente evento, eliminándolo de la cola (figura 3.4 a). La cola de eventos invoca a la ES correspondiente para que libere al cliente. El cliente deja la ES y entra en otra ES o sale del sistema. El control de la simulación pasa a la ES destino. En la ES destino vuelve a comenzar el proceso. Si a la llegada de un cliente a la ES, no hay servidores libres, se inserta en cola de espera y no se genera ningún evento (figura 3.4 b). El control vuelve a la cola de eventos.

Cuando no hay eventos en la cola de espera la simulación termina, aunque ha podido terminar con anterioridad por cualquier otra causa: ha finalizado el tiempo de simulación definido por el usuario o explícitamente el usuario ha decidido finalizar, dependiendo del estado del sistema o de algún evento en concreto. El núcleo de simulación recorre la cola de eventos, extrayendo el evento de cabeza (evento con tiempo menor) e invocando a la ES correspondiente. El control de la simulación pasa alternativamente de las ES a la cola de eventos.

El principal objetivo de la cola de eventos es sacar de ejecución en el momento indicado a los clientes que están recibiendo servicio en las ES. Las ES sólo actúan como consecuencia de un evento o de una llamada desde otra ES. Si la cola de eventos se quedase vacía terminaría la simulación, pues supondría que el sistema ya no evoluciona a ningún estado. Cuando se produce un evento, un cliente deja de servirse en una ES y puede entrar en otra ES o salir del sistema, lo que controla la ES en la que termina su ejecución. También es esta ES la que, al tener un servidor libre, pone en ejecución un cliente. Como consecuencia de la entrada de un cliente en la ES destino, el control de la simulación lo toma esta ES. Si la ES tiene un servidor libre pone el cliente en ejecución y crea el evento de salida del cliente. En este punto el control de la simulación pasa al gestor de eventos.

La ES tiene dos funciones que modelan su dinámica (figura 3.5):

- **Función de entrada**: (figura 3.6) se ejecuta cuando un cliente abandona una

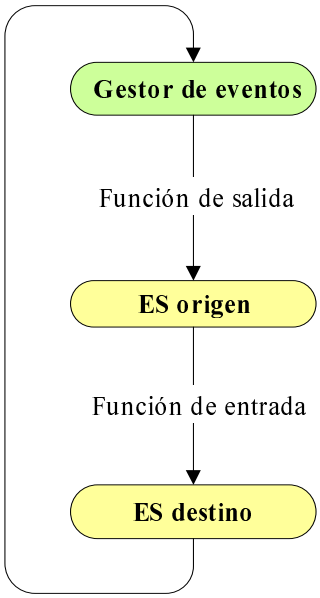

Figura 3.5: Control de la simulación en DESK

ES y entra en otra ES. La ES origen invoca la función de entrada de la ES destino.

- **Función de salida**: (figura 3.7) se ejecuta cuando un cliente termina de servirse en una ES. Esta función la invoca el gestor de eventos.

3.1.9. Cambio Dinámico del Modelo de Simulación

DESK permite simular sistemas que cambian sus características, topología o estructura dinámicamente, sin necesidad de detener la simulación ni redefinir el modelo. Este cambio dinámico del sistema puede producirse por cualquier condición en el sistema, en cualquier ES. Por ejemplo, si se detecta que la longitud media de la cola de espera de una ES supera un cierto valor, se puede crear otra ES y redirigir determinado número de clientes a la nueva ES. DESK permite detectar situaciones críticas y actuar dependiendo de esta situación.

El cambio dinámico del modelo del sistema puede llevarse a cabo fácilmente y de forma modular.

Permite crear o destruir ES dinámicamente, cambiar la topología del sistema, cambiar el comportamiento de las ES (políticas de servicio, tiempos de servicio, número de servidores,...), crear o destruir clientes dinámicamente o cambiar el comportamiento de los clientes.

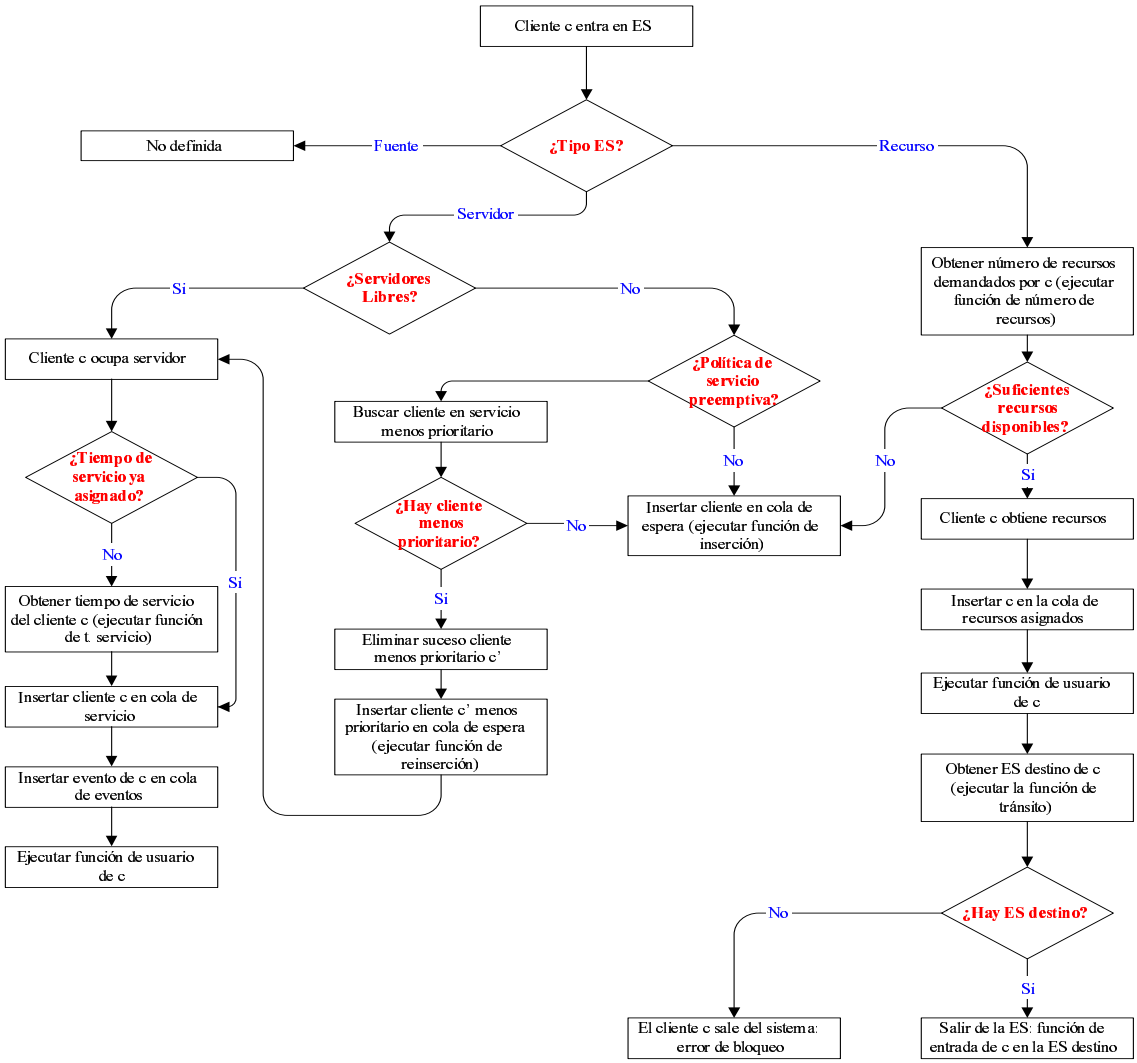

Figura 3.6: Función de entrada de la ES en DESK

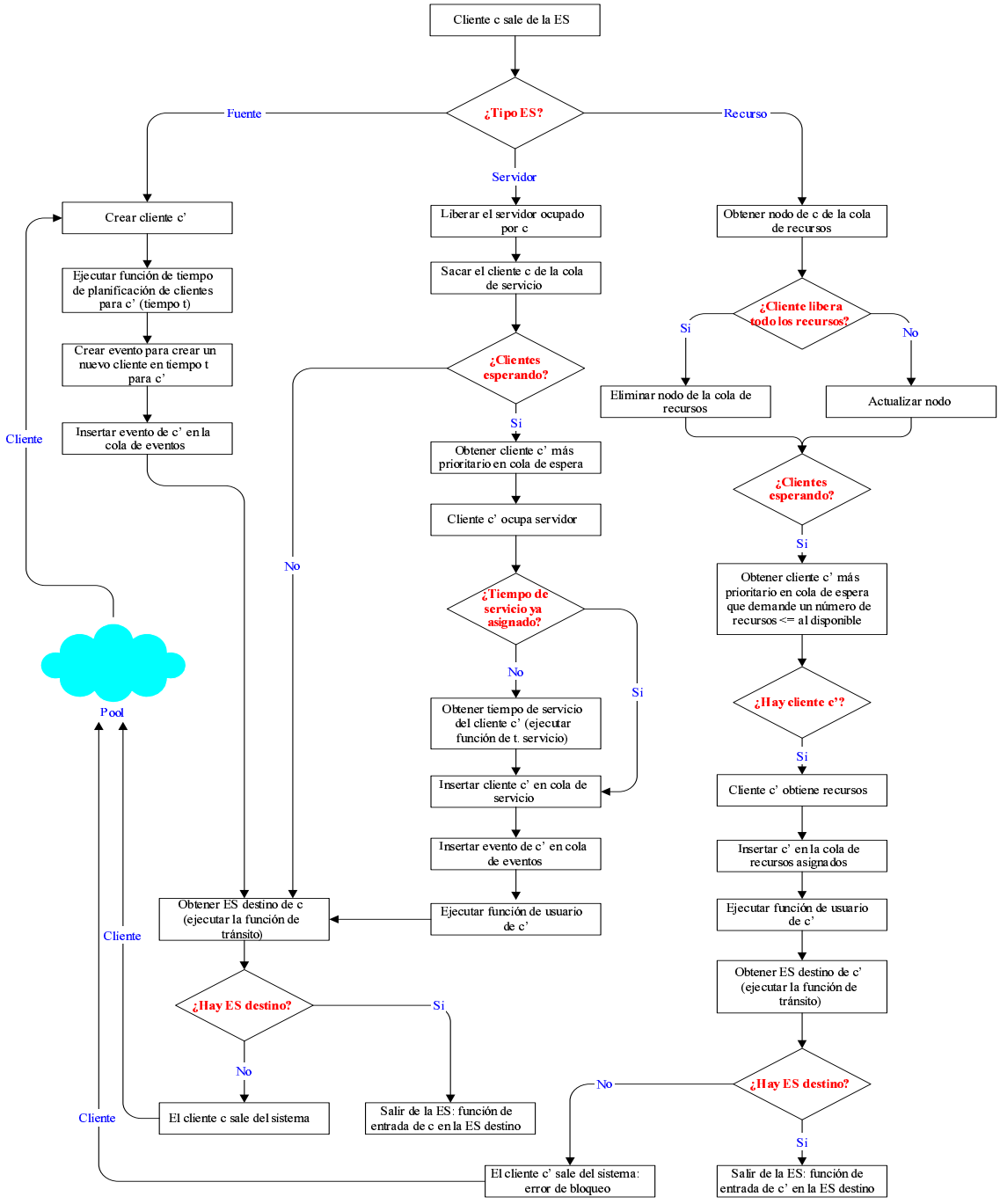

Figura 3.7: Función de salida de la ES en DESK

Hay dos características de DESK que le permiten variar el modelo dinámicamente:

- **DESK define el modelo utilizando funciones de comportamiento**. Esto le da al sistema la flexibilidad suficiente para variar cualquier característica dinámicamente: topología, comportamiento, tiempos de servicio o planificación. La mayor parte del comportamiento del sistema se define mediante funciones de comportamiento. La topología del sistema se define mediante las funciones de encaminamiento de clientes a la salida de la ES. Variando estas funciones, se varía dinámicamente la topología del sistema
- **DESK está implementado como una biblioteca de C++**. Por lo que la flexibilidad y modularidad de C++ puede utilizarse en la simulación.

DESK ofrece diferentes formas de modificar el modelo del sistema:

1. **Muestreo**: se muestrea continuamente el sistema hasta detectar la situación crítica que obliga a cambiar el sistema. El muestreo puede llevarse a cabo en cualquier función de comportamiento de la ES. Incluso pueden estar implicadas varias funciones. Una ES toma el control de la simulación cuando un cliente entra en la ES o abandona la ES. Las funciones de comportamiento implicadas en la entrada de un cliente en la ES son diferentes a las implicadas en la salida del cliente de la ES. Dependiendo de cual sea la situación que se desea muestrear, las funciones de comportamiento involucradas deben ser diferentes. El muestreo debe realizarse en la función de comportamiento adecuada de la ES adecuada, dependiendo de las necesidades del muestreo. Utilizar muestreo sobrecarga el sistema, pues se comprueba la condición crítica continuamente. La sobrecarga depende de la condición testeada y de la función donde se lleve a cabo. Por ejemplo, si la condición crítica es que la longitud media de la cola de espera de una ES alcance un determinado valor, el muestreo debe realizarse en la función de inserción de clientes en la cola de espera.
2. **Cambio de las funciones de comportamiento**: consiste en sustituir dinámicamente una o más funciones de comportamiento por otras. Debe muestrearse la condición crítica hasta que se cumpla. Cuando se detecta la condición crítica, la función de comportamiento (o funciones) que realiza el muestreo puede cambiarse por otra versión de la función adaptada a la nueva situación que modele el nuevo comportamiento. Esta nueva función ya no incluye el muestreo (a no ser que el comportamiento deba variar dinámicamente otra vez, con lo que el muestreo corresponderá a la nueva situación a controlar). Las funciones de comportamiento asignadas a la ES al inicio de la simulación pueden reasignarse en cualquier momento. DESK incluye funciones de biblioteca que permite reasignar funciones de comportamiento a la ES.

3. **ES de control**: dependiendo del cambio dinámico del sistema que deba llevarse a cabo, puede ser aconsejable utilizar una ES que se dedique a gestionar este cambio. La ES de control detecta la situación crítica y modifica el sistema convenientemente. Una ES de control puede ser una ES dedicada específicamente al control o su papel puede ser asumido por una ES del modelo. Usar una ES de control puede disminuir la sobrecarga del sistema llevada a cabo por el muestreo de la condición crítica. La ES de control toma el control de la simulación cuando un cliente entra o sale de ella. Esta técnica combina las dos anteriores.

En el apéndice A se muestran ejemplos de modelos de sistemas que cambian dinámicamente utilizando los tres métodos.

3.1.10. Aritmética

DESK utiliza dos tipos de aritmética: coma flotante y coma fija [Molla:2001]. La selección del tipo de aritmética se realiza mediante una sentencia de preprocesador en tiempo de compilación.

La utilización de coma fija es completamente transparente al usuario (la presentación de resultados y la definición del modelo es independiente de la aritmética utilizada).

3.1.10.1. Representación del Tiempo del Sistema

Coma Flotante

La representación de tiempos en el simulador se ha realizado con 64 bits.

Coma Fija

Los cuatro tipos básicos de representación en coma fija [Marven:1994] con los que se puede realizar un proceso de simulación se muestran en la tabla 3.5 (la unidad de tiempo utilizada es el segundo).

La representación en coma fija utiliza 32 bits para representar tanto la parte entera como la decimal. Según el formato elegido para la representación de coma fija estos 32 bits se utilizan para el exponente o para la mantisa. Se ha elegido para representar el tiempo el formato Q23, que utiliza 24 bits para la parte entera y 8 bits para la parte decimal, por ser la que proporciona el mayor valor representable, a costa de perder precisión en la parte decimal. La parte entera tiene signo.

Existe una considerable pérdida de representación numérica entre ambos formatos, lo que limita las posibilidades de la simulación en coma fija respecto a la coma flotante. Una representación numérica basada en aritmética en coma flotante, tiene como ventaja fundamental, el amplio rango de valores que puede tomar,

	Parte Entera (bits)	Parte Decimal (bits)	Rango Máximo	Resolución
Q0	0	32	1s	233ps
Q7	8	24	4m 16s	60ns
Q15	16	16	18h 12m 16s	15us
Q23	24	8	194d 4h 20m 16s	3.9ms
Q31	32	0	138a 30d 6h 28m 16s	1s

Tabla 3.5: Representación de datos en coma fija

basado principalmente en la posibilidad de modificar el escalado (exponente) con el que se afecta a la mantisa. Es normal disponer de ES con distribuciones temporales del orden del milisegundo y periodos de simulación del orden de minutos o incluso horas.

Por contra, la representación en coma fija permite trabajar con los operadores aritméticos enteros, facilitando su uso en equipos que no disponen de unidades aritmético-lógicas de coma flotante, como dispositivos portátiles PDA, móviles o consolas de videojuegos. Así como paralelizar cálculos en las CPU actuales, poniendo en funcionamiento tanto los operadores aritméticos como los reales simultáneamente.

3.1.10.2. Generación de Números Aleatorios

El generador de números aleatorios es distinto en coma fija y en coma flotante. En ambos casos se ha utilizado un grupo de semillas comunes.

Coma Flotante

Se han utilizado las rutinas de SMPL (rand.c), implementadas íntegramente en C.

Coma Fija

Las rutinas de generación de números aleatorios son independientes e implementadas en parte en ensamblador. Según el método utilizado para la obtención de los números aleatorios varía la precisión y la velocidad del simulador. En este caso prima la velocidad sobre la precisión por lo que se ha utilizado la representación más veloz pero menos precisa, basada en una tabla de logaritmos neperianos de 16K.

La notación interna de la generación de números aleatorios es Q15 (16 bits para el exponente y 16 bits para la mantisa), pero dentro de DESK se realiza una conversión a formato Q23, para adecuarlo a la representación temporal.

3.1.11. Pasos en la Creación del Modelo

Para definir un modelo en DESK se deben seguir los siguientes pasos:

1. Inicializar el simulador.
2. Definir las ES del modelo:
   a) Implementar las funciones de comportamiento de cada ES o usar funciones predefinidas o implementadas para otras ES.
   b) Determinar las características de la ES.
3. Introducir clientes en el sistema. Para sistemas cerrados o para sistemas abiertos con comportamientos determinados, es necesario comenzar la simulación con clientes en el sistema. La creación de un cliente supone insertarlo en una ES.
4. Comenzar la simulación. Se puede indicar la forma de finalización de la simulación. La forma más usual de finalizar la simulación es indicar el tiempo máximo de la simulación. También, permite comenzar a obtener medidas de la simulación en un instante determinado para eliminar estados transitorios.
5. Obtener resultados de la simulación.

3.1.12. Funciones de Biblioteca

DESK proporciona al usuario una biblioteca de funciones que permiten: control de la simulación (simulación, inicialización, finalización, inicio de contabilidad y gestión de errores), creación de ES de cada uno de los tipos, gestión de hijos, gestión de recursos, reasignación dinámica de funciones de comportamiento y contabilidad.

3.1.13. Conclusiones

DESK es un núcleo de simulación de eventos discretos generalista orientado a objetos implementado como una biblioteca de C++. Es un simulador versátil y con tiempos de simulación bajos. Permite simular sistemas con cualquier complejidad y tamaño. Permite definir cualquier modelo fácilmente siguiendo una metodología top-down.

DESK permite definir un modelo de sistema describiendo los elementos que componen el sistema (ES) y su interconexión (topología del sistema). El estilo de definición del modelo es similar al utilizado por QNAP. El programador sólo se centra en la descripción del modelo, no en los eventos del sistema. Permite variar el modelo fácil y rápidamente, por lo que DESK puede ser utilizado como prototipador.

Como ventaja respecto a QNAP en cuanto a la definición del modelo, incluye la no imposición de restricciones en el sistema. Esta no imposición de restricciones se refiere tanto al número de elementos en el sistema como al comportamiento del sistema:

- Número de ES y clientes ilimitado (depende de la memoria del sistema).
- Comportamiento del sistema. El sistema incluye los comportamientos típicos predefinidos, pero posibilita cualquier comportamiento.
- El control de la simulación puede realizarse en cualquier elemento del sistema y dependiendo de cualquier condición.
- Incluye características avanzadas del modelo: recursos, relación padre-hijo o semáforos.
- Permite cambiar el sistema dinámicamente, tanto su topología, comportamiento o estructura.

El modelo definido en QNAP está limitado en cuanto a comportamiento y número de elementos.

En DESK el comportamiento de cada ES puede seleccionarse entre una serie de comportamiento predefinidos (como en QNAP) o puede definirse cualquier comportamiento que el usuario desee, por caprichoso que este sea. Por tanto, además de permitir definir modelos fácil y rápidamente como QNAP, ofrece la posibilidad de definir cualquier comportamiento del sistema.

El coste temporal de la simulación de los modelos estudiados es inferior al coste temporal de simulación de los correspondientes modelos utilizando SMPL (apartado 5.1.1 del capítulo 5). SMPL limita severamente el número de clientes en el sistema. Permite definir comportamientos atípicos pero con un coste de programación, depuración y modificación muy alto, pues debe programarse cada posible evento del sistema. El coste de depurar o modificar un modelo en DESK es muy inferior al coste en SMPL.

DESK aúna las ventajas de SMPL y QNAP. Es más rápido que SMPL y permite definir el modelo al estilo de QNAP. Por ello puede utilizarse como prototipador y para obtener resultados finales de la simulación. Permite definir cualquier modelo, con cualquier comportamiento. DESK es un núcleo orientado a objetos, por lo que incluye todas las ventajas de un sistema modular (como reutilizar elementos del sistema fácilmente o crear subsistemas definidos por separado). Además, DESK incluye funcionalidad no presente en QNAP ni SMPL.

DESK permite incluir fácilmente dentro de la simulación elementos externos, como multimedia, alarmas o realizar la simulación en tiempo real.

El comportamiento, la estructura y la topología del sistema del sistema pueden variar dinámicamente, sin necesidad de detener la simulación y reconfigurar el modelo. Permite, por tanto modelar sistemas que cambian dinámicamente, de una manera natural. El modelo de simulación puede definirse para detectar cualquier situación crítica y entonces, cambiar el comportamiento de las ES o la topología del sistema, crear o destruir clientes, crear o destruir ES, cambiar políticas de servicio, algoritmos de planificación, encaminamiento de clientes en el sistema o extraer resultados. El modelo del sistema puede variar dinámicamente dependiendo de cualquier condición. El sistema puede reconfigurarse dinámicamente tantas veces como sea necesario. Otros simuladores, como SMPL, permiten variar el sistema dinámicamente, pero con coste de implementación excesivo.

Por tanto DESK puede modelar sistemas que otros simuladores no pueden con un coste de implementación y simulación bajo. DESK llega donde otros simuladores no llegan.

3.2. JDESK: Simulador de Eventos Discretos Basado en Web

La evolución de DESK ha seguido los pasos de otros simuladores de eventos discretos, adaptándose a las nuevas tecnologías. Se ha desarrollado una versión de DESK en Java que permite su utilización vía web. Incluye un asistente para facilitar el proceso de creación del modelo.

JDESK (**J**ava **D**iscrete **E**vents **S**imulation **K**ernel) [Garcia:2003] [Garciab:2003] es un simulador generalista de eventos discreto basado en web que permite modelar y simular un modelo implementado en código Java.

Es un simulador basado en web de acceso libre [Jdesk], por lo que puede utilizarse desde cualquier navegador. El modelo de simulación obtenido puede ejecutarse de forma local en cualquier plataforma. JDESK es un simulador versátil y con costes temporales de simulación bajos.

JDESK incluye un asistente para ayudar a los usuarios a definir modelos con comportamientos típicos, de forma fácil y rápida. El asistente guía al usuario durante todo el proceso de creación del modelo, permitiéndole definirlo mediante controles. Incluye también bibliotecas de componentes del modelo y ejemplos de modelos completos.

Además de las características de DESK, JDESK posee ciertas características especiales de la simulación vía web:

- El modelo se implementa en Java: el usuario no debe aprender un lenguaje específico para utilizar el simulador. Permite integrar elementos externos a

la simulación. Las funciones del simulador pueden estar mezcladas con otras funciones escritas en Java. Permite utilizar objetos implementados para otros propósitos. Es multiplataforma: una vez creado el modelo de simulación usando JDESK, puede ejecutarse en diferentes plataformas sin necesidad de volver a acceder al simulador, compilando el código en cada plataforma.

- JDESK facilita la definición, depuración y modificación de modelos, mediante ventanas de edición y controles.

- Creación modular del modelo de simulación. Permite reutilizar componentes de otros modelos o del propio modelo, debido a la utilización de objetos de Java. Soporta simulación distribuida, permitiendo crear equipos de trabajo sobre un mismo modelo, por la utilización de Java como lenguaje de implementación del modelo. Permite crear un repositorio de ejemplos y componentes de modelos, mediante bibliotecas escritas en Java y compartir el conocimiento. Los modelos escritos por un usuario pueden ser utilizados por otros usuarios como parte del sistema simulado.

JDESK y DESK tienen ciertas características comunes:

- Creación rápida y fácil del modelo de simulación. Los modelos de simulación se implementan definiendo los componentes del modelo y su interconexión. El modelo del sistema es una descripción de los elementos que lo componen.

- Fácil depuración y modificación del modelo. JDESK puede usarse como prototipador.

- Posibilidad de simular cualquier modelo: no impone restricciones al modelo a simular (número de clientes o ES) y permite definir cualquier comportamiento del sistema, por caprichoso que sea.

- Permite incluir elementos externos dentro de la simulación, como multimedia o alarmas y simulación en tiempo real.

- Las características del sistema pueden variar dinámicamente, incluso su topología. Permite crear o destruir dinámicamente elementos del modelo de simulación.

- Permite la utilización de características avanzadas de modelos como semáforos, recursos o creación de hijos.

- El control de la simulación puede realizarse en cualquier parte del sistema, dependiendo de cualquier condición o evento.

El simulador incluye una serie de comportamientos predefinidos de sistemas, de forma que cualquier sistema convencional puede modelarse fácil y rápidamente. También ofrece al usuario la posibilidad de definir comportamientos más sofisticados; por ejemplo, políticas de planificación no convencionales que incluso varíen dependiendo de las características del cliente actual o del estado del sistema. En el apéndice B se indica la forma de utilizar JDESK.

3.2.1. Conclusiones

JDESK tiene exactamente la misma funcionalidad de DESK. Aporta, respecto a DESK, todas las ventajas de la simulación en web respecto a la simulación convencional. Aunque, como desventaja, los costes de simulación de JDESK son superiores a los de DESK (apartado 5.1.2 del capítulo 5).

3.3. GDESK: Simulador de Eventos Discretos como Núcleo de Aplicaciones Gráficas

GDESK (**G**ame **D**iscrete **E**vent **S**imulation **K**ernel) [Garcia:2002] [Garcia:2003] [Garcia:2004] es un núcleo de simulación de aplicaciones gráficas en tiempo real, que, una vez integrado en una aplicación gráfica, gestiona los eventos del sistema. GDESK no puede funcionar de forma aislada, no tiene sentido si no se integra dentro de una aplicación.

GDESK controla la comunicación de los objetos (figura 3.8). Esta comunicación se realiza mediante paso de mensajes. Las funciones principales de GDESK son:

- Proporcionar a la aplicación gráfica las estructuras de datos y funciones necesarias para modelar el mecanismo de paso de mensajes.
- Gestionar la comunicación de los objetos mediante paso de mensajes:
  - Gestionar el proceso de envío de mensajes.
  - Mantener los mensajes enviados y todavía no recibidos por el objeto destino ordenados por tiempo de ocurrencia.
  - Enviar los mensajes a los objetos destino en el instante de tiempo indicado.
  - Garantizar que los mensajes son enviados a los objetos receptores en el tiempo indicado y ordenados por tiempo.

GDESK se limita a gestionar los mensajes que se envían y reciben en el sistema. No realiza ninguna tarea que los mensajes lleven asociada. El objeto receptor

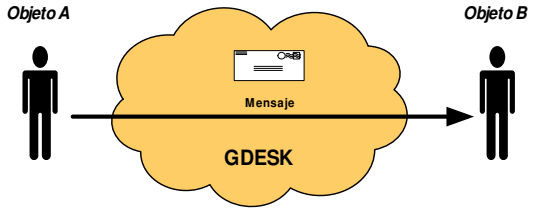

Figura 3.8: Ámbito de GDESK

será quien ejecute la función que el propio objeto tenga asociada al mensaje. El objeto, en función del tipo de mensaje, procedente de un objeto emisor determinado y con unos parámetros determinados, seleccionará el proceso de respuesta al mensaje. Todo este proceso pertenece a la aplicación gráfica donde se integre GDESK y no al propio GDESK.

En este caso, GDESK se utiliza para gestionar los eventos de una aplicación gráfica en tiempo real, pero puede usarse en otro tipo de aplicaciones que necesiten comunicar sus elementos mediante paso de mensajes. Si los mensajes llevan asociado un tiempo de ocurrencia, la aplicación donde se integre trabajará en tiempo real. Si los mensajes tienen asociado tiempo 0, se reciben en el mismo instante en que se envían. En este caso la aplicación no considera el tiempo real.

3.3.1. Objetivos

El objetivo de GDESK es crear un simulador de eventos discreto (partiendo de DESK) que pueda servir como núcleo de simulación de aplicaciones gráficas en tiempo real. Este núcleo debe:

1. Gestionar los eventos del sistema de manera discreta.

2. Ser autocontenido, de manera que sea fácilmente integrable en otros motores gráficos. Debe poder adaptarse a diferentes aplicaciones gráficas, por lo que, parte de su funcionalidad debe poder delegarse en la aplicación gráfica.

3. Permitir una gestión de eventos transparente al usuario. El programador debe interferir lo menos posible en el funcionamiento del núcleo.

3.3.2. Eventos del Sistema

El concepto de evento del sistema en GDESK corresponde a la definición de evento de la simulación de sistemas [Fishman:1978] (ver apartado 2.3.1 del capítulo 2).

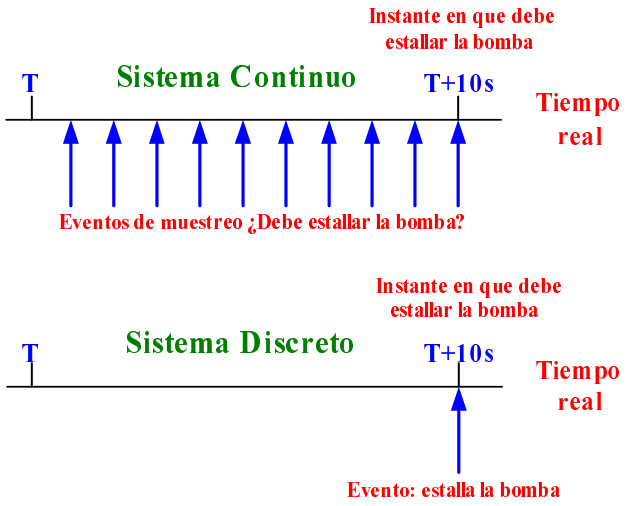

Figura 3.9: Eventos en un sistema continuo y discreto

Uno de los objetivos de GDESK es gestionar los eventos del sistema de manera discreta. Un ejemplo de diferencia entre el comportamiento continuo y discreto puede ser una bomba con temporizador (figura 3.9). En el instante T se programa una bomba para que estalle en 10 segundos. Un sistema continuo muestrea el sistema. En cada muestreo pregunta si se ha alcanzado el tiempo en que la bomba debe estallar. En un sistema discreto se genera un evento programado para dentro de 10 segundos que hará estallar la bomba. Se elimina el muestreo durante ese intervalo de tiempo.

Con la integración de GDESK se considera la aplicación gráfica como un sistema discreto que evoluciona con la ocurrencia de eventos. La concepción de la aplicación gráfica como un sistema discreto permite modelar comportamientos discretos, continuos e híbridos. Un comportamiento continuo corresponde a un comportamiento discreto donde los eventos se producen a intervalos de tiempo fijos. En la aplicación pueden coexistir comportamientos de todo tipo, en diferentes objetos o en el mismo objeto.

Un sistema discreto permite que un comportamiento continuo se muestree a intervalos variables, según las necesidades del objeto. La figura 5.33 del capítulo 5 muestra un ejemplo de muestreo de un objeto donde, según el comportamiento varíe más o menos rápidamente, se generan eventos con una cadencia diferente.

En GDESK cada evento se modela mediante un mensaje. Con la introducción de GDESK el sistema debe funcionar como un sistema discreto utilizando para ello el mecanismo de paso de mensajes.

DESK	GDESK
ES	Objeto
Cliente o Evento	Mensaje
Pool de clientes	Gestor de memoria o Pool
Cola de eventos	Gestor de mensajes o Dispatcher
Funciones de comportamiento	Funciones de respuesta a mensajes

Tabla 3.6: Correspondencia entre los elementos de DESK y GDESK

3.3.3. Cambios en DESK para la Creación de GDESK

DESK es un simulador de eventos discretos y, por tanto, no puede usarse directamente como núcleo de aplicaciones gráficas en tiempo real (exige ciertos ajustes). Los cambios que se han llevado a cabo son:

1. **Nomenclatura**: la nomenclatura típica de simulación de eventos discretos se ha adaptado a una nomenclatura más cercana a las aplicaciones gráficas. La tabla 3.6 muestra los cambios llevados a cabo.

2. **Gestión de tiempos**: en un simulador de eventos discretos el tiempo del simulador viene determinado únicamente por el tiempo de ocurrencia de los eventos. El reloj de simulación avanza cada vez que se produce un evento en el sistema. Los eventos se ejecutan consecutivamente, sin esperas entre ellos. Este tiempo no tiene relación alguna con el tiempo real, pues lo que interesa es el comportamiento del sistema para extraer conclusiones. En una aplicación gráfica en tiempo real, los eventos deben ocurrir en el instante de tiempo real en el que se ha indicado que ocurran. Por ello, el reloj de GDESK debe avanzar con la ocurrencia de eventos, al igual que en DESK, pero únicamente si el tiempo del sistema ha alcanzado el tiempo del evento. Si el siguiente evento de GDESK debe suceder en el instante de tiempo T_{evento} y el reloj del sistema se encuentra en el instante de tiempo T_{real} y $T_{evento} > T_{real}$, el sistema debe quedar inactivo durante $T_{evento} - T_{real}$ unidades de tiempo. Transcurrido ese tiempo se ejecuta el evento pendiente y el reloj del simulador se actualiza con el tiempo del evento ejecutado $T_{reloj} = T_{evento}$.

3. **Objetos** (ES en DESK): la aplicación gráfica pasa a ser un conjunto de objetos que intercambian mensajes (eventos). Todos los objetos de la aplicación heredan de la clase base de GDESK, permitiendo que:

   - Los objetos contengan los métodos de gestión de mensajes entrantes y salientes.
   - GDESK pueda tratar todos los objetos de forma uniforme.

Un objeto, cuando recibe un mensaje realiza una serie de tareas que el programador define como consecuencia de la llegada de un mensaje específico. Un mensaje llega a un objeto y aquí termina su cometido, por lo que se elimina, enviándolo al pool de mensajes. El mensaje pasa directamente al pool de mensajes, no transita a otro objeto.

Desaparecen ciertas estructuras de datos que anteriormente eran necesarias para la gestión de las ES:

- Los objetos no contienen ninguna clase de estructura que contenga mensajes, por lo que las colas de clientes dejan de existir.
- Los objetos no dan servicio a los mensajes, por lo que los objetos no contienen servidores.
- En DESK se definen tres tipos de ES: fuentes, servidores y recursos. Esta diferenciación ya no existe en GDESK. Para el núcleo de simulación sólo hay un tipo de objeto. La aplicación gráfica donde se integra GDESK puede definir tantos tipos de objetos como considere necesario, pero para GDESK todos se tratan de igual manera.
- Funciones de comportamiento de las ES: las ES tienen asociado en DESK funciones de comportamiento, que en GDESK dejan de existir:
  - Función de tiempo de servicio: los mensajes no reciben ningún servicio del objeto.
  - Función de inserción en cola de espera y función de reinserción en cola de espera: la cola de espera deja de existir.
  - Función de tránsito: un mensaje cuando abandona un objeto va directamente al pool, no transita a otro objeto.
  - Función de número de recursos: no hay recursos.
  - Función de usuario: se ejecuta cuando el cliente comienza a recibir servicio en una ES. Ahora los mensajes no reciben servicio, por lo que deja de tener sentido esta función.

  El comportamiento de los objetos se define en la aplicación gráfica. Debe existir un comportamiento del objeto mensaje al que sea sensible el objeto.

4. **Mensajes** (eventos en DESK): un mensaje modela la comunicación entre dos objetos o el comportamiento de un objeto. Un mensaje siempre lo genera un objeto y tiene otro objeto como destino. Los mensajes en DESK contienen una serie de atributos que sirven para definir el comportamiento de los objetos implicados en la interacción. En GDESK parte de los atributos del mensaje son propios del simulador y parte los define el programador de la aplicación. Los mensajes dejan de tener el significado que un cliente (evento) tenía en DESK: la salida de un cliente de la ES donde estaba recibiendo servicio. Ahora un mensaje es una cierta información que recibe un objeto en un instante

determinado. El objeto no retiene el mensaje. Los mensajes en GDESK, al igual que los clientes en DESK, sirven para decidir que elemento del sistema está activo en un momento determinado.

5. **Fin de la simulación**: DESK permite varias formas de finalizar la simulación de un sistema: simular hasta que se alcanza un tiempo máximo, simular hasta que se cumpla una determinada condición que establece el programador o simular hasta que no se producen eventos en el sistema. Ninguna de estas formas de finalización de la simulación es apropiada para una aplicación gráfica en tiempo real. La finalización de la simulación en GDESK debe producirse explícitamente mediante un evento de usuario o por una condición de error. Si la cola de mensajes pendientes está vacía, la aplicación gráfica está pausado o realizando un determinado proceso.

GDESK mantiene toda la potencia y funcionalidad de DESK, aunque ciertos elementos hayan desaparecido por dejar de tener sentido en aplicaciones gráficas en tiempo real. Los aspectos que se han modificado afectan a su operatividad, pero mantienen la misma flexibilidad y potencia de DESK. Ciertos elementos se eliminan de GDESK porque se complementan con otros elementos de la aplicación gráfica donde se integra. La tabla 3.7 muestra un resumen de los aspectos que se han cambiado y mantenido. La declaración de los objetos de la aplicación y la definición de su comportamiento corresponde a la definición de ES y su comportamiento en DESK.

3.3.4. Lenguaje de Implementación

Fly3D está implementado utilizando C++, al igual que DESK, por lo que no es necesario cambiar de lenguaje para la implementación de GDESK.

3.3.5. Entidades Básicas

Las entidades básicas con las que trabaja el núcleo de simulación son los mensajes. Los mensajes sirven para comunicar objetos de la aplicación gráfica donde se integra GDESK. Los objetos se definen y gestionan en la aplicación gráfica. GDESK sólo envía y recibe mensajes de los objetos. Aunque los objetos no pertenecen a GDESK deben incluir las funciones de recepción y envío de mensajes de GDESK para poder comunicarse mediante paso de mensajes.

3.3.5.1. Mensajes

Los mensajes son elementos pasivos que sirven para comunicar los objetos de la aplicación gráfica. Pertenecen a la clase *GDeskMessage* (figura 3.10). Un objeto sólo puede actuar y modificar su comportamiento cuando le llega un mensaje que así se

	DESK	**GDESK**	**Funcionalidad**
Flexibilidad			Alta y similar
Potencia			Alta y similar
Estructura			Similar
Ejecución	Autoejecutable	Contenido en una aplicación gráfica	
Elementos del sistema	ES. Pertenecientes y controladas por DESK	Objetos. Pertenecientes y controlados por la aplicación gráfica	Definidos por el usuario
Tipos de elementos	Servidores, fuentes y recursos	Definidos por el usuario con una parte común de recepción y envío de mensajes	
Parámetros del evento/mensaje	Fijos	Parte fijos y parte definidos por el programador de la aplicación donde se integre	
Dispatcher	Gestión de eventos	Gestión de mensajes	Pequeños cambios. Igual funcionalidad
Pool	Eventos	Mensajes	Igual funcionalidad
Definición de comportamiento	Funciones de comportamiento	Funciones de recepción de mensajes	Similar
Tiempo	T. de simulación	T. de simulación real	
Comunicación de elementos del sistema	Eventos	Mensajes	Similar
Fin de la simulación	Definida mediante una condición u evento	Evento de usuario	
Control de la ejecución	DESK	GDESK y la aplicación donde se integre	
Monitorización	Funciones de contabilidad	Funciones de consumo de tiempo, número de ejecuciones, de los procesos del sistema	Modificadas

Tabla 3.7: Comparativa de funcionalidad de DESK y GDESK

Figura 3.10: Mensaje de GDESK

lo indica. El comportamiento del objeto es el conjunto de respuestas a los posibles mensajes a los que es sensible (corresponde a las antiguas funciones de comportamiento de DESK). El mecanismo de paso de mensajes modela el comportamiento de cada objeto y del sistema en general. Cualquier proceso que involucre a los objetos debe ser modelado mediante paso de mensajes.

Un mensaje contiene dos tipos de parámetros:

1. **Parámetros de gestión del simulador o parámetros de GDESK**: los utiliza y gestiona el simulador, pero el usuario puede darles valor y consultarlos. Contiene la información necesaria para comunicar el objeto fuente con el objeto destino. Son:

   - Objeto origen del mensaje.
   - Objeto destino del mensaje.
   - Tiempo de ocurrencia del evento asociado al mensaje. Es el intervalo de tiempo, transcurrido el cual, se desea que el mensaje sea recibido por el objeto destino.
   - Enlace del mensaje al pool.

   El objeto destino y el tiempo de ocurrencia los determina el programador de la aplicación gráfica al enviar un mensaje. Estos parámetros no son directamente accesibles por el programador (debe utilizar una función para modificarlos), para evitar que el programador pueda modificar incorrectamente datos necesarios para el simulador. Estos son los únicos datos del mensaje que el simulador

utiliza. La declaración de estos atributos pertenece al mensaje base del simulador y no es modificable por el programador. Estos parámetros los gestiona GDESK.

2. **Parámetros de la aplicación gráfica**: cada aplicación puede necesitar una colección diferente de parámetros, por lo que se deja libertad al usuario para que defina parte de los parámetros del mensaje con la finalidad de dotar al núcleo de simulación de la máxima flexibilidad posible. El programador puede añadir tantos atributos al mensaje como sean necesarios para dotarlo de contenido. Estos parámetros los define, declara y gestiona el programador y son comunes a todos los objetos de la aplicación. Son completamente transparentes al núcleo de simulación. Contienen la información necesaria para modelar el comportamiento de los objetos y la interacción entre éstos. La respuesta de un objeto a un mensaje puede ser diferente en función de determinados valores de estos parámetros. Por ejemplo, una piedra choca contra una pared. La piedra envía un mensaje a la pared indicándole su tamaño, velocidad, peso, tipo de material y forma. Cuando la pared recibe el mensaje, puede destruirse, deformarse o permanecer inalterable dependiendo de los parámetros del mensaje. El comportamiento del objeto receptor depende de quien es el objeto emisor del mensaje, del tipo de mensaje (parámetros de la aplicación) y del estado actual del objeto.

El objeto origen y destino de un mensaje puede ser el mismo. Esta es una forma de modelar comportamientos continuos del simulador (modela el comportamiento del objeto). Si el objeto receptor es un objeto diferente, el mensaje sirve para comunicar los dos objetos.

3.3.5.2. Objetos

Los objetos básicos del simulador son los objetos básicos de la aplicación gráfica. Simulador y aplicación comparten objetos. Para que un objeto pueda comunicarse con el resto de objetos debe incluir las funciones de recepción y envío de mensajes del simulador, por tanto, debe heredar de la clase base de los objetos del simulador *GDESK_CEntity* (figura 3.11). Esta clase base proporciona a los objetos métodos para la recepción y el envío de mensajes. El resto del objeto lo define y controla el programador.

Las funciones de envío y recepción de mensajes del objeto son:

- *GDESK_SendMessage(objeto_receptor, tiempo)*: envía un mensaje que debe recibir el objeto receptor cuando transcurra el tiempo indicado como parámetro. Si el mensaje debe producirse en el instante actual el tiempo del mensaje debe ser cero. Esta función la invoca el programador dentro de las funciones de

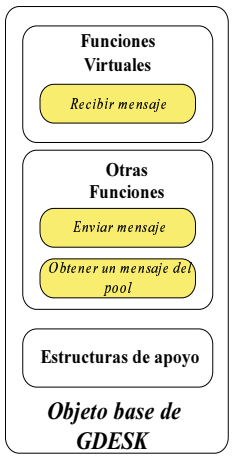

Figura 3.11: Objeto base de GDESK

comportamiento del objeto cuando desea que el objeto envíe un mensaje.

- *GDESK_ReceiveMessage(mensaje)*: función virtual que se ejecuta cuando el objeto recibe un mensaje. El contenido de esta función depende exclusivamente del programador del juego. Un objeto toma el control de la simulación cuando recibe un mensaje. Esta función define el comportamiento del objeto o invoca a las funciones de comportamiento.

3.3.6. Funciones de Comportamiento

Las funciones de comportamiento definen en DESK el comportamiento, la topología y la estructura del sistema. Se definen y asocian a cada ES. Estas funciones dejan de existir en GDESK. Las ES de DESK son en GDESK los objetos de la aplicación gráfica y por tanto están definidos por el programador de la aplicación gráfica, no están incluidos en el núcleo de GDESK. GDESK trabaja con los mensajes que los objetos se envían. En DESK las funciones de comportamiento las ejecuta el propio DESK.

En GDESK el proceso se ha adaptado a la nueva situación: la coexistencia durante la ejecución de GDESK y de la aplicación gráfica. Mediante el mecanismo de paso de mensajes GDESK envía los datos a los objetos para que pongan en ejecución su función de comportamiento adecuada (aunque el término función de comportamiento ya no tiene sentido en GDESK, es realmente el proceso de respuesta del objeto al mensaje). Durante la ejecución, el control de la ejecución pasa constantemente de GDESK a un objeto del sistema determinado. El objeto del sistema define su respuesta a un tipo de mensaje determinado, con unos parámetros determinados. Esta respuesta de los objetos corresponde a las funciones de comportamiento de GDESK.

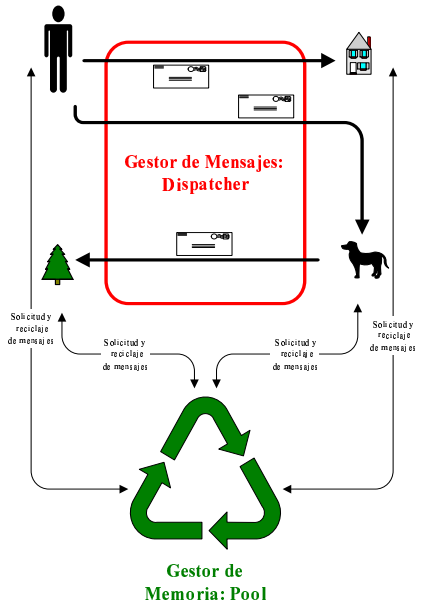

Figura 3.12: Arquitectura de GDESK

3.3.7. Arquitectura

Para gestionar el proceso de paso de mensajes, GDESK utiliza dos estructuras principales: el gestor de mensajes y el gestor de memoria (figura 3.12). Estas estructuras trabajan con mensajes. GDESK trata con objetos para enviarles y recibir de ellos mensajes, pero no gestiona objetos. Los objetos son entidades de la aplicación gráfica manejadas por la propia aplicación.

3.3.7.1. Gestor de Mensajes: Dispatcher

El gestor de mensajes es la estructura de datos principal del simulador y se encarga de controlar el envío y recepción de mensajes. Todos los mensajes del sistema los gestiona el dispatcher. Cuando un objeto envía un mensaje a otro objeto (o a si mismo), no lo hace directamente, sino que es el dispatcher quien controla el proceso. Sus funciones principales son:

- Capturar los mensajes enviados por los objetos (de un objeto a otro o de un objeto a si mismo). Los objetos envían y reciben mensajes como si el dispatcher no existiese. El dispatcher es transparente a los objetos. Un objeto envía un mensaje a otro objeto utilizando la función de envío de mensajes, pero realmente el mensaje es capturado por el dispatcher. Los objetos reciben los mensajes como si los hubiesen enviado directamente los objetos emisores.

La función de envío de mensaje de los objetos, realmente envía el mensaje al dispatcher.

- Sincronizar los mensajes de los objetos. El tiempo del mensaje es el tiempo que debe transcurrir desde que se envía el mensaje hasta que es recibido por el objeto destino. Este tiempo se convierte a un tiempo global del simulador.
- Mantener los mensajes cuyo tiempo todavía no ha vencido ordenados por tiempo de ocurrencia.
- Enviar los mensajes cuyo tiempo ha vencido a los objetos destino, invocando la función de recepción de mensaje del objeto destino. La función de recepción de mensaje del objeto la implementa el programador, por lo que el programador decide el comportamiento del objeto como respuesta al mensaje.

El dispatcher contiene dos elementos básicos:

- **Reloj de la simulación**: contiene el tiempo actual de simulación del sistema. Se actualiza cada vez que el dispatcher envía un mensaje al objeto destino.
- **Estructura de almacenamiento de mensajes**: contiene los mensajes que los objetos han enviado y todavía no ha vencido su tiempo de recepción ordenados ascendentemente por el tiempo asociado al mensaje. La estructura elegida para almacenar los mensajes es un heap por las siguientes razones:
  - Las operaciones a realizar con el heap son únicamente inserción ordenada y eliminación de cabeza. El coste de estas operaciones es O(log N) (tabla 3.3).
  - No se deben realizar operaciones de eliminación intermedia (motivo por el cual se eligió utilizar como estructura de gestión de eventos en DESK una lista doblemente enlazada).

  El heap se implementa sobre un vector (figura 3.13). Se ha optado por esta representación porque los costes amortizados de ordenación son menores que utilizando una representación de nodos enlazados. Los elementos del vector son estructuras con dos campos: tiempo absoluto del mensaje y puntero al mensaje almacenado.

  El gestor de mensajes se redimensiona cuando el número de mensajes esperando a ser enviados alcanza un determinado porcentaje de su capacidad. Este proceso supone incrementar el tamaño del vector en un cierto número de elementos. El valor límite para lanzar la redimensión y el número de elementos que se añaden al vector son configurables por el usuario.

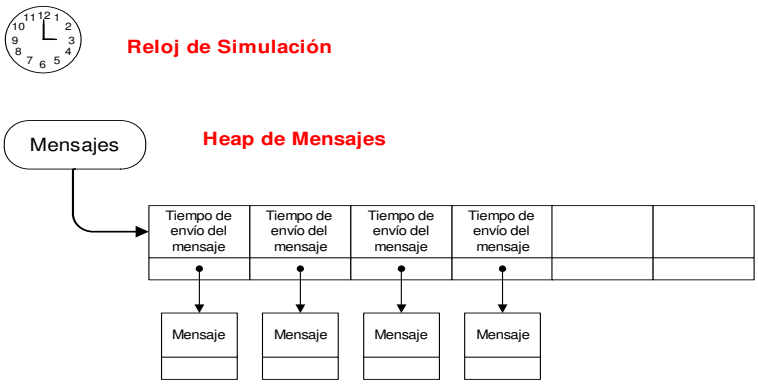

Figura 3.13: Dispatcher de GDESK

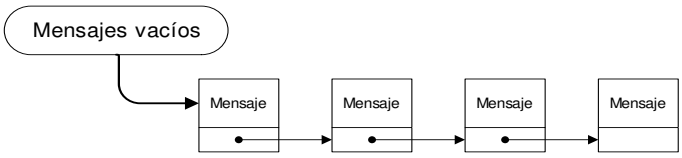

Figura 3.14: Pool de GDESK

3.3.7.2. Gestor de Memoria: Pool

El gestor de memoria o pool (figura 3.14) se encarga de almacenar los mensajes que no están actualmente en el sistema. El gestor de memoria tiene como objetivo minimizar las llamadas al GMD, evitando crear y destruir mensajes dinámicamente si no es imprescindible. La funcionalidad y estructura es casi la misma que en DESK (apartado 3.1.6.4). Las diferencias entre ambas estructuras son:

- El pool de GDESK se inicializa con un número determinado de mensajes. Este número corresponde con el número de mensajes que inicialmente puede gestionar el dispatcher. Este valor es configurable por el programador de la aplicación gráfica. El pool de DESK está inicialmente vacío.

- Cuando el gestor de mensajes se redimensiona, el pool también se redimensiona. Se crean dinámicamente tantos mensajes como nuevos elementos se han añadido al gestor de mensajes. De esta forma existe siempre un mensaje en el pool si hay una posición libre del vector del dispatcher. El número de mensajes totales en el simulador será el número de mensajes actualmente en el pool mas el número de mensajes circulando por el sistema. Este número total de mensajes corresponde con el número de mensajes que puede gestionar el dispatcher.

El pool utiliza una lista LIFO simplemente enlazada para almacenar los mensajes. El propio mensaje contiene el puntero de enlace en esta lista, para evitar crear y destruir nodos de la lista dinámicamente. Los objetos se comunican directamente con el pool para solicitar mensajes vacíos y para deshacerse de los mensajes recibidos una vez procesados (figura 3.12).

3.3.7.3. Dinámica del Núcleo de Simulación

La figura 3.15 muestra el proceso llevado a cabo por el dispatcher en la dinámica del sistema global. GDESK toma el control de la simulación en dos ocasiones:

1. La aplicación gráfica invoca la función de simulación de GDESK (habitualmente desde el bucle principal de la aplicación).
2. Un objeto de la aplicación gráfica envía un mensaje a otro objeto.

Cuando se invoca la función de simulación de GDESK (*GDESK_Simulate*) el control pasa al dispatcher. El proceso llevado a cabo por el dispatcher cuando se invoca la función de simulación es:

1. El dispatcher envía todos los mensajes cuyo tiempo es menor o igual al tiempo de simulación. Para ello invoca el proceso de recepción de mensajes del objeto destino.
2. El reloj de simulación se actualiza con cada mensaje procesado.
3. Si el tiempo del mensaje de cabeza T_{cabeza} todavía no se ha cumplido significa que no hay mensajes pendientes de enviar. Y no los habrá durante $T_{inactivo} = T_{simulacion} - T_{cabeza}$ unidades de tiempo.

Cuando un objeto de la aplicación gráfica envía un mensaje a otro objeto el control de la simulación pasa al dispatcher de GDESK. Los pasos seguidos son:

1. El dispatcher captura el mensaje enviado antes de llegar a su destinatario.
2. Calcula el tiempo absoluto del mensaje. El tiempo del mensaje indica el intervalo de tiempo que debe tardar el mensaje en ser recibido por su destinatario. Este tiempo debe convertirse a un tiempo de simulación absoluto, añadiéndole el tiempo de simulación de GDESK.
3. El mensaje se inserta en el heap ordenado por el tiempo absoluto del mensaje.
4. El dispatcher comprueba si tiene mensajes pendientes de enviar y los envía (siguiendo el mismo proceso que cuando se comienza la simulación porque se ha invocado la función de simulación de GDESK).

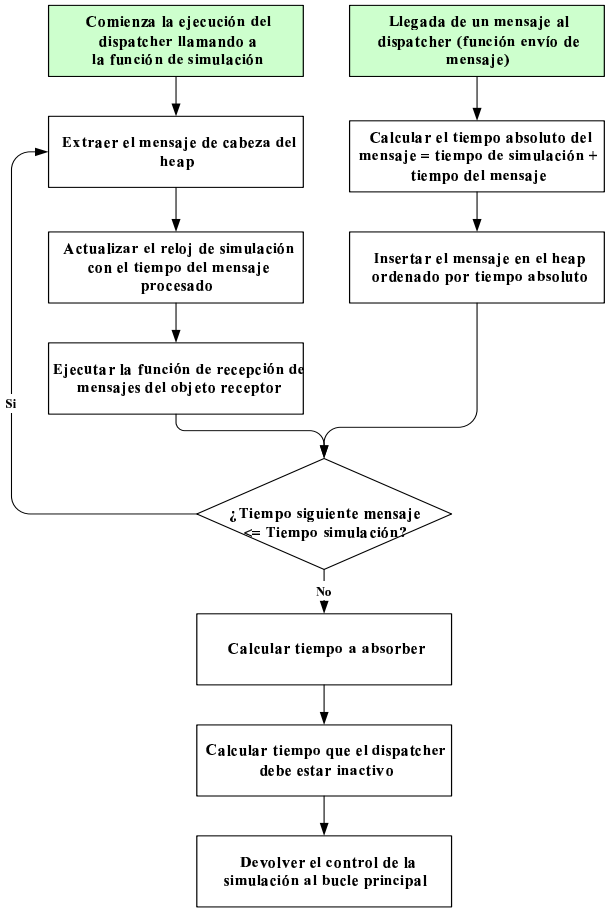

Figura 3.15: Proceso del dispatcher en GDESK

El dispatcher finaliza su ejecución, por lo que el control de la simulación pasa al bucle principal de la aplicación gráfica, pero únicamente durante $T_{inactivo}$ unidades de tiempo. Este tiempo inactivo debe recalcularse para absorber los tiempos de proceso, como se verá posteriormente. El control de este tiempo depende exclusivamente de la aplicación, no lo trata GDESK. La aplicación podría liberarlo para otras aplicaciones o quedarse en un bucle de espera. Transcurrido este tiempo, el control debe volver al dispatcher para que pueda seguir el proceso de envío de mensajes. Cada vez que comienza a ejecutarse la simulación de GDESK es porque hay un mensaje pendiente.

GDESK debe integrarse en una aplicación gráfica para poder funcionar. Por ello la dinámica del sistema la comparte con la aplicación gráfica en la que está integrado. En el apartado 4.5 del capítulo 4 se explica detalladamente la dinámica del sistema de GDESK integrado en Fly3D.

El mecanismo de paso de mensajes tiene sentido integrado en una aplicación gráfica, por lo que también se explica en el capítulo 4 (apartado 4.4).

3.3.7.4. Tiempo de Simulación

GDESK trata de ejecutar correctamente el comportamiento definido para los objetos a costa de que el tiempo de simulación pueda diferir del tiempo real. Si el tiempo de simulación es superior al tiempo real, significa que el sistema está colapsado (no es capaz de simular manteniendo la calidad de servicio definida en tiempo real). Por ello, el reloj de simulación no se ajusta al reloj del sistema.

Absorción de Tiempo

Cuando el dispatcher envía un mensaje a su destinatario se ejecuta todo el proceso asociado al mensaje, como proceso de visualización, simulación, detección de colisiones o envío de mensajes respuesta. Estos procesos pueden consumir un tiempo variable, dependiendo del tipo de proceso, la potencia de cálculo de la máquina o el número de objetos implicados.

El dispatcher cuando no tiene mensajes pendientes de procesar retorna el control al programa principal por un tiempo limitado, hasta que se cumpla el tiempo del primer mensaje pendiente. El dispatcher calcula el tiempo que quiere estar inactivo y comunica este tiempo al programa principal, quien lo usa como considere más apropiado. Para calcular el tiempo de inactividad debe tenerse en cuenta el tiempo que el dispatcher ha estado ocupado ejecutando mensajes y sus procesos correspondientes.

Sea:

$T_{inactivo}$ tiempo que debe estar inactivo el dispatcher hasta que deba procesar el siguiente evento.

$T_{simulacion}$ tiempo actual del reloj de simulación.

N número de mensajes procesados consecutivamente por el dispatcher la última vez que tomó el control de la ejecución.

$T_{proceso}$ tiempo consumido en el procesamiento de los N mensajes y sus procesos asociados.

$T_{proceso_i}$ tiempo de procesamiento consumido por el mensaje i.

$T_{mensaje\_pendiente}$ tiempo asociado al primer mensaje pendiente que el dispatcher debe ejecutar cuando vuelva a tomar el control de la simulación.

El tiempo de proceso durante una ejecución del dispatcher viene dado por la ecuación 3.6.

$$T_{proceso} = \sum_{i=1}^{N} T_{proceso_i} \tag{3.6}$$

El tiempo $T_{proceso}$ debe ser absorbido por el tiempo inactivo (ecuación 3.7). Este tiempo no puede ser negativo.

$$T_{inactivo} = max(T_{mensaje\_pendiente} - T_{simulacion} - T_{proceso}, 0) \tag{3.7}$$

Si se cumple la ecuación 3.8 el sistema no tiene potencia de cálculo suficiente para simular y visualizar el sistema en tiempo real (sistema colapsado).

$$T_{mensaje\_pendiente} - T_{simulacion} - T_{proceso} \leq 0 \tag{3.8}$$

Los sistemas continuos siguen el tiempo real, aun a costa de perder calidad en la simulación, simular incorrectamente o no detectar colisiones.

GDESK prioriza la correcta simulación del sistema y el mantenimiento de la calidad de servicio de los objetos sobre la ejecución del sistema en tiempo real. El sistema consume el tiempo necesario para simular los objetos adecuadamente. Por ello, el sistema se ralentiza para poder ejecutar todos los mensajes con la cadencia indicada. En este caso el tiempo de simulación es mayor que el tiempo real del sistema. Se ejecutan todos los eventos, independientemente de que el sistema esté o no colapsado.

Si el usuario desea que en su aplicación gráfica utilizando GDESK se priorice la ejecución en tiempo real sobre la correcta simulación del sistema, debe definir comportamientos de objetos adaptables a la carga del sistema. Un comportamiento adaptable supondría, por ejemplo disminuir la carga de simulación o visualización de los objetos o disminuir la frecuencia de cuadro.

3.3.8. Monitorización del Sistema

GDESK permite monitorizar el sistema. Las antiguas funciones de contabilidad de DESK se adaptan a las necesidades de un núcleo de simulación de aplicaciones gráficas en tiempo real. Estas funciones se utilizan para obtener los resultados de la monitorización del sistema.

La monitorización del sistema se lleva a cabo en dos niveles:

1. **Monitorización de GDESK**. Monitorización de los procesos asociados únicamente a GDESK.
2. **Monitorización de la aplicación gráfica donde se integra GDESK**. Esta monitorización se define para la aplicación gráfica donde se integra y se explica detalladamente en el capítulo 4.

La monitorización de GDESK es independiente de la monitorización de la aplicación gráfica, pues es independiente de la aplicación donde se integra. En cambio la monitorización de la aplicación gráfica depende exclusivamente de la aplicación gráfica aunque utilice mecanismos de GDESK para llevarse a cabo.

3.3.8.1. Monitorización de GDESK

La monitorización de GDESK permite comprobar la sobrecarga de gestión por la utilización de GDESK y en que procesos se consume esta sobrecarga. En el apartado 5.2.2.7 del capítulo 5 se muestra cual es el porcentaje de sobrecarga de gestión de eventos en las pruebas realizadas.

Se selecciona mediante una sentencia de preprocesador. Permite obtener un fichero (*THR_GDeskProfile.txt*) con los resultados de la monitorización al finalizar la ejecución de la simulación. En este fichero se incluye una serie de información temporal de las funciones del núcleo de GDESK. Esta monitorización permite conocer el tiempo consumido en cada uno de los procesos de GDESK. Para cada proceso relevante de GDESK se muestra:

- Número de ejecuciones.
- Número de ciclos de reloj consumidos.
- Segundos consumidos.

En este fichero, además, se incluyen resultados del proceso global de simulación:

- Tiempo total consumido en el proceso de simulación.

- Tiempo total consumido en el proceso de visualización.
- Tiempo real consumido por la aplicación gráfica en la ejecución.

3.3.9. Aritmética

DESK incluye la posibilidad de seleccionar el tipo de aritmética (coma flotante o coma fija). Por simplicidad se ha considerado únicamente aritmética en coma flotante en GDESK. Una posible línea futura de investigación es la integración de coma fija en el núcleo de simulación.

3.3.10. Conclusiones

GDESK es la adaptación del simulador de eventos discreto DESK a núcleo de aplicaciones gráficas en tiempo real. GDESK es un núcleo monolítico e independiente de la aplicación en la que se integra. Pero no puede funcionar si no se integra en una aplicación gráfica.

Aunque algunos de los elementos de DESK (completamente innecesarios para gestionar los eventos de una aplicación gráfica) han desaparecido en GDESK, pues dejan de tener sentido, los elementos básicos y la potencia del núcleo se han mantenido. Los principios básicos de ambos núcleos son los mismos: flexibilidad, posibilidad de simular cualquier sistema y la no imposición de restricciones al sistema. Las estructuras básicas de GDESK son la mismas que DESK, pero adaptadas a la nueva situación.

La principal diferencia entre DESK y GDESK es la gestión de tiempos. En GDESK el tiempo de simulación corresponde, mientras el sistema no esté colapsado, al tiempo real del sistema.

GDESK gestiona los eventos del sistema haciendo que los objetos de la aplicación gráfica donde se integra se comuniquen mediante paso de mensajes. GDESK discretiza la aplicación gráfica donde se integra. La simulación discreta permite modelar comportamientos discretos, continuos e híbridos. Una consecuencia de la discretización del sistema es el desacoplo de las fases de visualización y simulación. Para que GDESK funcione correctamente la aplicación donde se integra debe utilizar el mecanismo de paso de mensajes para modelar el comportamiento de los objetos de la aplicación. Si GDESK se integra en la aplicación, pero ésta no lo utiliza correctamente, el comportamiento puede seguir siendo continuo o no desacoplar el sistema.

GDESK controla el proceso de envío y recepción de mensajes. Los mensajes tiene asociado un tiempo que indica el instante en que deben ser recibidos por el objeto receptor. GDESK captura los mensajes enviados y los almacena hasta que se cumple el tiempo del mensaje. En ese instante envía el mensaje al objeto receptor.

GDESK sólo trata con mensajes, no realiza los procesos que el objeto receptor tenga asociados al mensaje.

En el capítulo 4 se muestra la integración de GDESK en una aplicación gráfica en tiempo real concreta: el núcleo de videojuegos Fly3D, para obtener el núcleo de videojuegos DFLy3D, discreto desacoplado. Los resultados de esta integración se muestran en el capítulo 5.

Capítulo 4

DFly3D: Fly3D Discreto

GDESK por si sólo no tiene sentido, debe integrarse dentro de una aplicación gráfica para comprobar su funcionalidad. El presente capítulo muestra la integración de GDESK en una aplicación gráfica de tiempo real: el motor de videojuegos Fly3D, cómo se modifica la funcionalidad de Fly3D y se complementa con GDESK. Las razones por las que se ha elegido Fly3D para integrar GDESK se muestran en el apartado 2.6 del capítulo 2.

La aplicación gráfica seleccionada debe modificarse en dos sentidos:

1. La aplicación debe orientarse a eventos, modelándolos mediante paso de mensajes. El proceso visualización debe orientarse a eventos también.
2. Integrar el núcleo de simulación GDESK, que gestiona los mensajes (eventos) del sistema.

DFly3D (**D**iscrete **Fly3D**) [Garciab:2004] [Garciac:2004] [Garciad:2004]

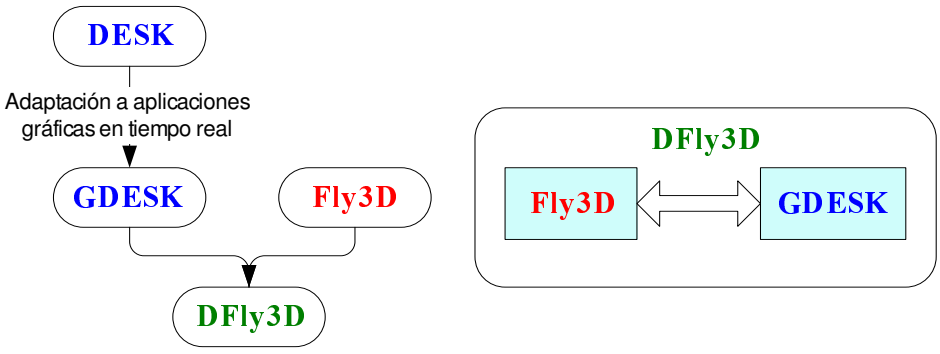

Figura 4.1: Integración de GDESK en Fly3D

[Garciae:2004] es un núcleo de aplicaciones gráficas en tiempo real discreto y desacoplado. Se obtiene integrando el núcleo de simulación GDESK en Fly3D (figura 4.1).

Una aplicación creada usando DFLy3D es un conjunto de objetos comunicándose mediante paso de mensajes.

Integrar GDESK en Fly3D no supone cambiar procesos del videojuego como proceso de visualización o método de detección de colisiones, ni estructuras como el grafo de escena. Simplemente se modifica la simulación del videojuego y el instante de tiempo en que se realiza cada visualización.

Una vez integrado en Fly3D, GDESK gestiona el paso de mensajes del sistema. Todos los objetos que se comuniquen mediante paso de mensajes tendrán un comportamiento orientado a eventos. La orientación a eventos permite discretizar el sistema. Simular el comportamiento de un objeto de forma discreta permite modelar cualquier comportamiento del objeto, continuo, discreto e híbrido (ver apartado 3.3.2 del capítulo 3). Uno de los principales procesos que deben discretizarse es el proceso de visualización, pues permite desacoplar las fases de simulación y visualización. GDESK por si sólo, ni discretiza ni desacopla el sistema.

La integración conlleva modificar Fly3D para que el sistema funcione mediante paso de mensajes (discretizar el sistema), incluido el proceso de visualización (desacoplo).

DFly3D se deja preparado para funcionar de modo continuo o discreto (coexisten Fly3D y DFly3D). La selección de un modo u otro se realiza mediante sentencias de preprocesador.

4.1. Objetos

Antes de comenzar con la integración de GDESK en Fly3D, se debe introducir el elemento básico de la aplicación gráfica DFly3D, resultado de la integración: el objeto de DFly3D. Una aplicación de DFly3D es un conjunto de objetos intercambiando mensajes (figura 4.2). Todos los elementos de DFly3D deben ser objetos. Además, estos objetos deben contener las funciones para comunicarse con otros objetos del sistema. Estas funciones les permiten enviar y recibir mensajes, y por tanto que la gestión de sus eventos la realice GDESK. Todos los objetos deben heredar del objeto base de GDESK.

A pesar de que todos los objetos de DFly3D deben contener las funciones de comunicación, no tienen porque ser todos exactamente del mismo tipo. El programador puede definir tantos tipos de objetos como sea necesario, pero deben estar agrupados en dos categorías (figura 4.3):

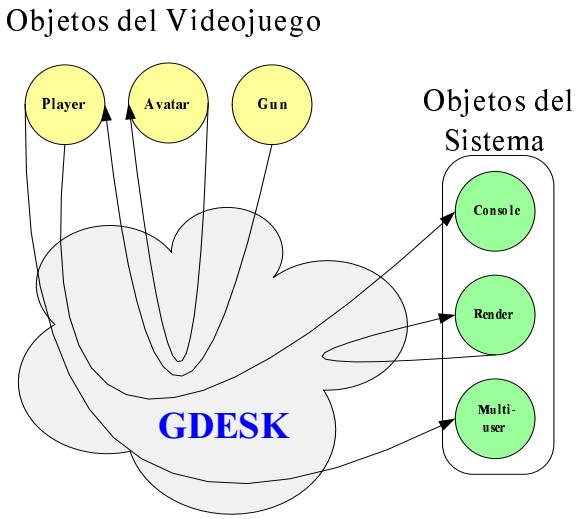

Figura 4.2: Objetos en DFLy3D

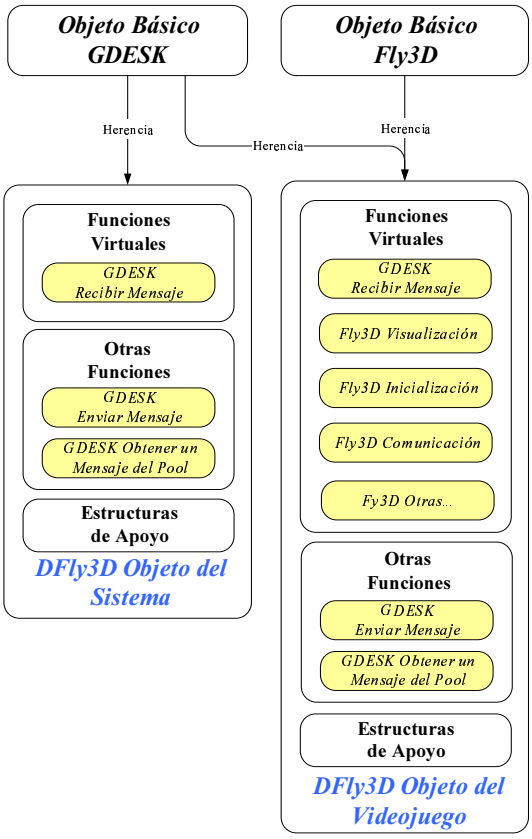

Figura 4.3: Tipos de objetos en DFLy3D

1. **Objetos del sistema**: son los objetos que realizan tareas del núcleo de Fly3D (módulo *flyEngine*). Se crean durante el proceso de integración de GDESK en Fly3D para aislar el código correspondiente a ciertas tareas, para que puedan ser objetos de GDESK y puedan comunicarse mediante paso de mensajes. No se definen nuevos objetos del sistema una vez se ha terminado el proceso de integración. Heredan del objeto base de GDESK. No son accesibles para el programador de las aplicaciones creadas usando DFly3D.

2. **Objetos del videojuego**: son los objetos creados para una aplicación o videojuego (plugin) de DFly3D. Son independientes del núcleo de DFLy3D. Por ejemplo, personajes, paredes, armas o proyectiles. Son los mismos objetos creados en Fly3D (heredan del objeto base de Fly3D) pero su comportamiento se modela mediante paso de mensajes (heredan del objeto base de GDESK). La diferencia fundamental radica en la función de simulación del objeto (*step*). Estos objetos aúnan las características de los objetos básicos de Fly3D con las características de los objetos básicos de GDESK. Estos objetos los crea y define el programador de la aplicación gráfica.

Ambos objetos generan mensajes. GDESK trata todos los mensajes de la misma forma, independientemente del tipo de objeto que los genera. No se establecen prioridades implícitas de mensajes. La prioridad viene determinada por el tiempo de ocurrencia del mensaje (los mensajes se ejecutan ordenados por el tiempo del mensaje).

4.2. Integración de GDESK en Fly3D

En este apartado se van a detallar los cambios que se han llevado a cabo en Fly3D para integrar GDESK.

Fly3D diferencia claramente el núcleo de la aplicación gráfica de los videojuegos creados usando el núcleo. Por tanto, la integración de GDESK en Fly3D debe realizarse en dos niveles (figura 4.4):

1. Cambiar el núcleo de Fly3D para integrar GDESK. El núcleo de Fly3D está altamente modularizado. Esto permite que los cambios a realizar impliquen únicamente a los siguientes módulos:

   - Módulo *flyEngine* de Fly3D. Este módulo contiene, entre otros, los procesos de simulación y visualización del sistema (estos procesos invocan las funciones de simulación y visualización de cada objeto del videojuego).
   - Front-end del videojuego. Este modulo incluye el bucle principal de la aplicación (que debe ser modificado).

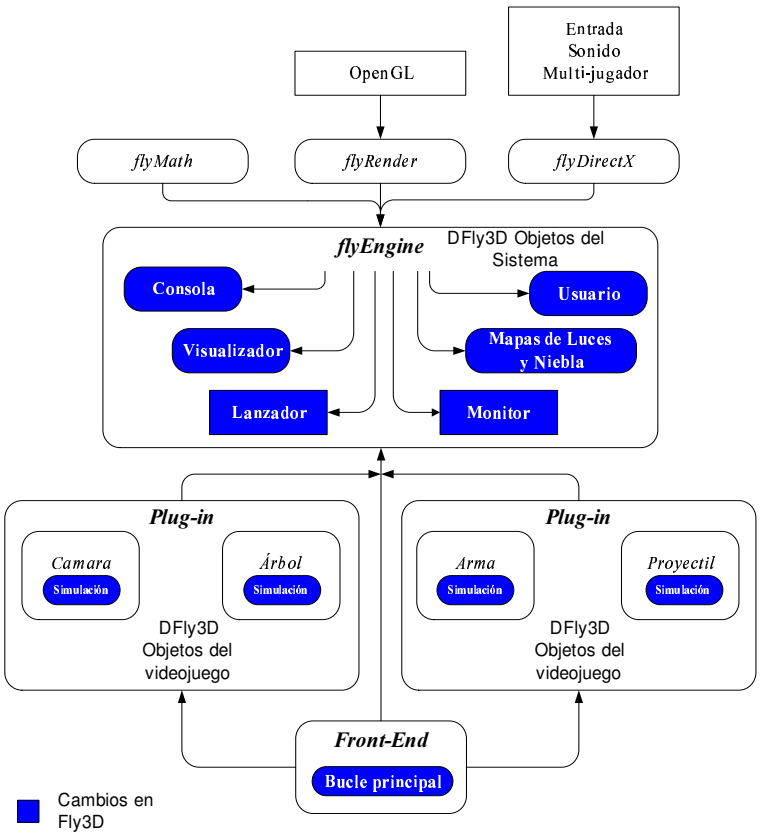

Figura 4.4: Cambios en Fly3D para integrar GDESK

2. Modificar las aplicaciones creadas usando Fly3D que se van a utilizar en las pruebas de integración. Supone modificar cada uno de los objetos de las aplicaciones (plugins) para que la simulación esté controlada por GDESK.

4.2.1. Cambios en el Núcleo de Fly3D: Objetos del Sistema

El módulo *flyEngine* de Fly3D contiene funciones llamadas directamente por el videojuego (plugins de la figura D.1 del apéndice D) o por el bucle principal de la aplicación (front-end de la figura D.2 del apéndice D). Durante la integración, estas tareas que son invocadas por las aplicaciones o por el núcleo del videojuego deben aislarse en objetos. La mayor parte del código del módulo *flyEngine* no se modifica, únicamente una pequeña parte.

Los objetos del sistema en DFly3D son (figura 4.4):

- *Consola.*
- *Visualizador.*
- *Mapas de luces y nieblas.*
- *Usuario.*
- *Lanzador.*
- *Monitor.*

Cada objeto controla una tarea específica del núcleo de DFly3D manteniendo la misma funcionalidad de los procesos de Fly3D. Los objetos del sistema usan estructuras comunes del núcleo. El objetivo de crear estos objetos no es modularizar el código, sino crear unidades dinámicas que soporten paso de mensajes, para lo que deben convertirse en objetos base de GDESK.

El comportamiento y la interacción de estos objetos se modela mediante paso de mensajes. Por ejemplo, cada vez que el objeto *consola* debe ejecutarse (como consecuencia de que el usuario ha cambiado a modo consola), se envía un mensaje al objeto *consola* desde el objeto *usuario*. La *consola*, como consecuencia del mensaje, toma el control de la ejecución y actúa en consecuencia.

Originalmente, las tareas que realizan estos objetos se muestreaban en cada iteración del bucle principal en Fly3D. En DFly3D cada objeto decide cuando y como actuar. Por ejemplo, la *consola* sólo actúa cuando el usuario elige el modo consola o cuando debe ejecutarse un comando de consola. El principal objetivo de la creación de estos objetos es evitar el muestreo de componentes que produce un bucle principal de videojuegos tradicional continuo acoplado (algoritmo 1 del capítulo 2).

Cada antigua llamada a funciones de Fly3D, ahora incluida método de un objeto del sistema, debe sustituirse por el envío de un mensaje al objeto correspondiente.

Cada objeto sólo actúa cuando debe hacerlo. No es necesario preguntar a la *consola* (o a cualquier otro objeto del sistema) si debe activarse o tiene tareas pendientes. Cada objeto del sistema define su comportamiento como respuesta a la llegada de un mensaje. Su respuesta puede depender de: el objeto emisor del mensaje, los parámetros del mensaje o el estado del propio objeto.

Los objetos *consola*, *mapas de luces* y *usuario* tienen exactamente la misma funcionalidad que en Fly3D, lo único que ha cambiado es la forma en la que se arranca su comportamiento. El objeto *visualizador*, además de realizar la misma tarea de visualización de Fly3D, añade como funcionalidad extra la posibilidad de elegir el momento de generar una visualización de la escena (desacoplo).

Además de los objetos que realizan tareas de Fly3D, DFly3D incluye dos objetos del sistema que realizan tareas propias de DFly3D y que, por tanto, no existe su funcionalidad en Fly3D: los objetos *lanzador* y *monitor*. Estos objetos son los únicos que arrancan su comportamiento mediante un mensaje que no proviene de ningún objeto, sino de la rutina de inicialización del sistema.

El objeto *lanzador* se crea para arrancar el comportamiento de los objetos de DFly3D:

- Objetos del sistema: envía un mensaje a cada uno de los objetos del sistema cuyo comportamiento debe iniciarse al arrancar el videojuego (como el objeto *visualizador*).
- Objetos del videojuego. Los plugins de los videojuegos creados usando DFly3D deben contener un objeto *lanzador* propio (objeto del videojuego definido por el programador de la aplicación gráfica que utiliza DFly3D). El objeto *lanzador* del sistema envía un mensaje a cada uno de los objetos *lanzador* de los plugins cargados. El programador del videojuego define en los objetos *lanzador* de los plugins, que objetos del plugin arrancan su comportamiento inicialmente (enviándoles un mensaje con el tiempo apropiado). De esta forma, puede arrancarse únicamente el comportamiento de los objetos del grafo de escena que lo requieran (los objetos pueden visualizarse pero no muestrearse si no tienen ninguna tarea que realizar). Sólo se arrancan los objetos que el programador decide y de la forma que decide. Creando los objetos *lanzador* de los plugins se conserva la modularidad del núcleo de DFly3D.

El objeto *monitor* realiza las funciones de monitorización del sistema. El comportamiento de este objeto sólo se arranca si se ha definido que se monitorice el sistema (mediante las apropiadas sentencias de preprocesador). El proceso de monitorización se verá en el apartado 4.7.

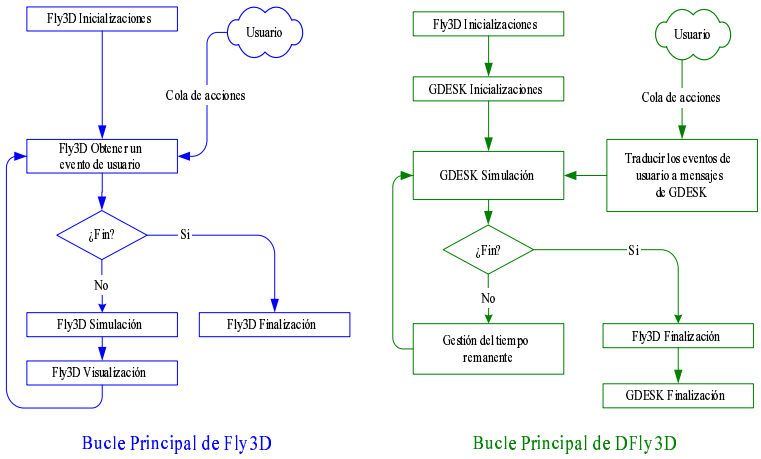

Figura 4.5: Bucle principal de Fly3D y de DFly3D

En el apéndice E.1 se muestra la integración del objeto *consola*.

4.2.2. Cambios en el Front-End de Fly3D: Bucle Principal

El bucle principal de una aplicación de Fly3D está incluido en el front-end de la aplicación. Este bucle principal debe discretizarse y desacoplarse. El bucle principal de Fly3D invoca procesos del módulo de *flyEngine*, como visualización y simulación de los objetos.

El bucle principal de DFly3D (figura 4.5) sustituye los procesos de simulación y visualización, invocados en cada pasada del bucle por el proceso de simulación de GDESK (la visualización y el resto de procesos del sistema se integran en el proceso de simulación). Este bucle principal, básicamente, llama a la función de simulación de GDESK.

El proceso de simulación de GDESK devuelve el control al bucle principal cuando está inactivo y sólo por el periodo de tiempo que va a estar inactivo. El bucle principal puede gestionar este tiempo como considere oportuno. Transcurrido ese tiempo, el control debe volver a GDESK.

Para más información sobre la gestión de tiempos ver el apartado 3.3.7.4 del capítulo 3.

4.2.3. Cambios en el Videojuego de Fly3D: Objetos del Videojuego

Las aplicaciones creadas usando DFly3D y Fly3D son muy similares. Los objetos y los procesos generales son similares. Se diferencian únicamente en una pequeña parte de la implementación del objeto, el proceso de simulación. La simulación del

comportamiento del objeto y la interacción con otros objetos se modela:

- En Fly3D mediante muestreo del objeto. Los objetos de Fly3D heredan de un objeto base que define, entre otras, una función virtual para la simulación (*step*). La simulación del objeto se implementa en dicha función, a la que se le pasa como parámetro el tiempo desde la última simulación. En cada pasada del bucle principal se recorren todos los objetos del grafo de escena invocando la función *step* de cada objeto.
- En DFLy3D mediante paso de mensajes. Los objetos de DFly3D no necesitan implementar la función *step* pues el proceso de simulación de cada objeto se invoca desde la función de recepción de mensajes. Cuando un objeto recibe un mensaje invoca el comportamiento correspondiente (dependiendo del tipo de mensaje, el objeto emisor o los parámetros del mensaje recibido por el objeto).

En el apéndice E.2 se muestra la integración del objeto *bola* de un videojuego creado usando DFLy3D. El objeto se crea primero para un videojuego de Fly3D y se modifica para DFly3D.

4.3. Simulación de Objetos

Una aplicación creada usando DFly3D es un conjunto de objetos intercambiando mensajes (figura 4.2). El programador define el comportamiento de cada objeto y el comportamiento del sistema usando el mecanismo de paso de mensajes. Los mensajes tienen dos utilidades para los objetos:

1. **Comunicar objetos**: el objeto A necesita comunicarse con el objeto B porque ha colisionado con él o porque desea modificar su comportamiento. A genera un mensaje dirigido a B. El objeto B puede cambiar su estado o actuar de determinada manera. El objeto B también puede generar mensajes a otros objetos.
2. **Modelar el comportamiento de un objeto**: los comportamientos continuos o discretos de los objetos se modelan mediante mensajes al propio objeto en intervalos de tiempo fijos o variables. Un objeto sólo actúa como consecuencia de la llegada de un mensaje. Cuando un objeto A quiere cambiar su comportamiento al cabo de un tiempo T (por ejemplo modificar su posición o estado), genera un mensaje dirigido a si mismo. El tiempo del mensaje será T. Como consecuencia de la llegada de un mensaje de si mismo, el objeto A modifica su estado (su estado actual puede depender de los parámetros del mensaje).

Cuando un objeto envía un mensaje a otro objeto define el tiempo del mensaje:

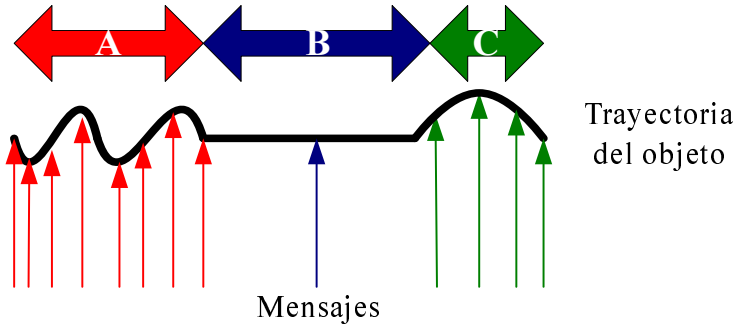

Figura 4.6: Ejemplo de tasa de generación de mensajes según el comportamiento del objeto en DFly3D

- Si el tiempo del mensaje T es 0 significa que el mensaje lo recibe el objeto receptor en el mismo instante que el objeto fuente lo envía.

- Si el tiempo del mensaje T es mayor que 0, el objeto receptor recibe el mensaje transcurridas T unidades de tiempo. Si la potencia de cálculo de la máquina no es suficiente para simular el sistema correctamente, el tiempo que realmente transcurre hasta la recepción del mensaje es mayor que T.

La figura 5.33 muestra un objeto con una trayectoria cambiante en función del tiempo. El objeto debe muestrearse con una frecuencia diferente en cada intervalo (A, B o C) para cumplir el teorema de Niquist-Shannon. El tiempo de cada mensaje debe adaptarse dinámicamente a las necesidades de muestreo del objeto. El número de mensajes generados en una interacción depende de la forma en que el programador ha modelado o definido el comportamiento del objeto y del sistema en general.

El comportamiento del objeto se define en su función de recepción de mensajes (función virtual de la clase base de GDESK). Esta función debe permitir discriminar el comportamiento del objeto en función del mensaje. Esta función define a que mensajes es sensible el objeto y su respuesta para cada tipo de mensaje.

La figura 4.7 muestra un ejemplo de comportamiento de un objeto ante la llegada de un mensaje. Todo este comportamiento debe invocarse desde la función de recepción de mensajes. Esta función puede producir un cambio de estado o comportamiento y/o el envío de nuevos mensajes. El objeto sólo actúa como consecuencia de la llegada de un mensaje.

Un sistema continuo en DFly3D es un sistema discreto donde los objetos se envían mensajes a intervalos de tiempo constantes e iguales para todos los objetos del videojuego.

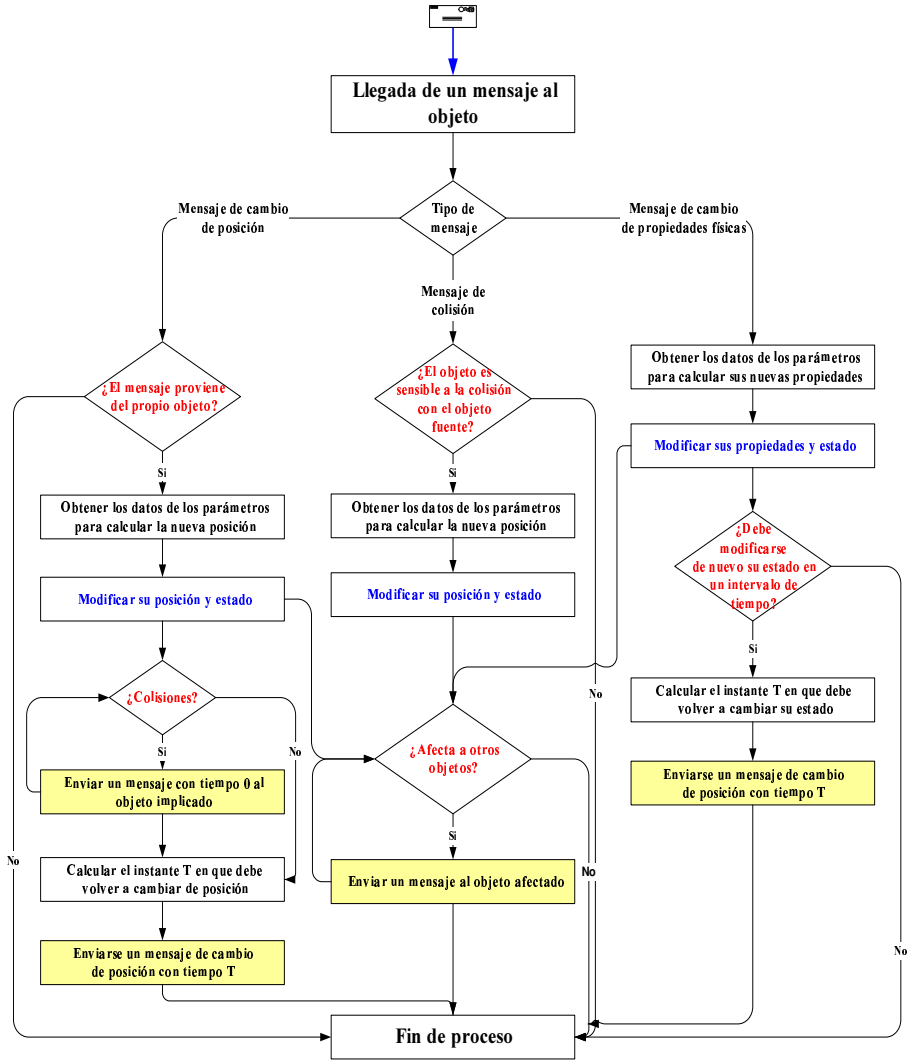

Figura 4.7: Ejemplo de simulación de un objeto en DFly3D

4.3.1. Relojes de Simulación

Los sistemas continuos utilizan un reloj de simulación que define los pasos en el proceso de simulación (frecuencia de muestreo igual a frecuencia de cuadro). En cada paso se evoluciona todo el sistema y se visualiza (sistema continuo acoplado). La duración del paso depende de la carga del sistema y de la potencia de cálculo de la máquina. Por tanto el reloj de un sistema continuo tiene la función de sincronizar los objetos.

Los sistemas discretos no evolucionan todo el sistema siguiendo los pasos definidos por un reloj global. Los objetos actúan en respuesta a mensajes enviados por otros objetos o por ellos mismos. Si un objeto no debe actualizarse, no genera mensajes. Por ejemplo, una bola en movimiento debe muestrearse cumpliendo el teorema de Niquist-Shannon. Si la frecuencia de muestreo de la bola es FM, debe enviarse un mensaje cada $1/FM$ unidades de tiempo. Cada objeto puede tener una frecuencia de muestreo diferente e independiente de la frecuencia de cuadro.

La frecuencia de muestreo de cada objeto puede ser constante o variable durante la ejecución del videojuego. Depende del comportamiento del objeto y de las interacciones con el resto de objetos del sistema. Tener frecuencias de muestreo diferentes no supone tener un reloj para cada objeto. El dispatcher sincroniza los tiempos de los mensajes de cada objeto con el reloj de simulación. El reloj de simulación se actualiza cada vez que el dispatcher procesa un mensaje pendiente. Cada objeto i debe actualizarse al cabo de T_i unidades de tiempo (valor que depende de su comportamiento instantáneo). El dispatcher utiliza el reloj de simulación para convertir el tiempo T_i en un tiempo global (sincroniza los mensajes).

4.4. Mecanismo de Paso de Mensajes

La percepción que tiene cualquier objeto sobre el mecanismo de paso de mensajes es diferente a la que tiene GDESK de este mismo proceso. La figura 4.8 muestra el proceso de paso de mensajes desde el punto de vista de un objeto. El objeto A envía un mensaje al objeto B. Los pasos seguidos en este proceso son:

1. A desea interaccionar con B:
   - *a*) A obtiene un mensaje del pool (figura 4.8 paso 1).
   - *b*) A rellena los parámetros del mensaje.
   - *c*) A invoca la función de envío del mensaje de GDESK (figura 4.8 paso 2). En la llamada a la función se pasan como parámetros: el objeto receptor (B) y el tiempo de recepción ($T_{mensaje}$).

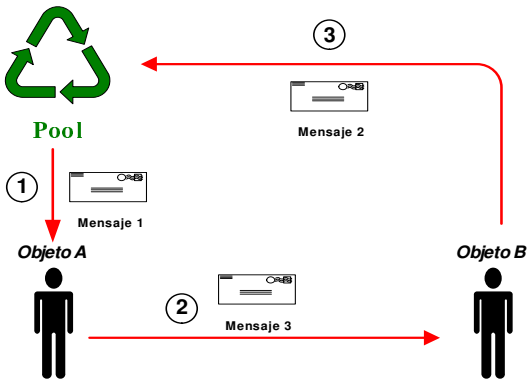

Figura 4.8: Mecanismo de paso de mensajes en DFly3D desde el punto de vista de un objeto

2. B recibe el mensaje transcurridas $T_{mensaje}$ unidades de tiempo. B no percibe que haya transcurrido un tiempo desde el envío del mensaje. Para B el mensaje acaba de ser enviado.

   a) B obtiene los parámetros del mensaje que considera necesarios.

   b) B actúa como consecuencia del mensaje y cambia su estado.

   c) B desecha el mensaje (figura 4.8 paso 3) enviándolo al pool. El mensaje se almacena en el pool.

Pero el mecanismo de paso de mensajes es un proceso algo más complejo pues intervienen estructuras de GDESK. La figura 4.9 muestra el proceso completo desde el punto de vista de GDESK:

1. A desea interaccionar con B (solicita el mensaje al pool, rellena los parámetros e invoca a la función de envío de mensaje).

2. El dispatcher captura el mensaje (figura 4.9 paso 1):

   a) Calcula el tiempo absoluto del mensaje $T_{absoluto}$ usando el reloj de simulación $T_{simulacion}$ y el tiempo de mensaje $T_{mensaje}$: $T_{absoluto} = T_{mensaje} + T_{simulacion}$

   b) Almacena el mensaje (ordenado por $T_{absoluto}$) en el heap (figura 4.9 paso 2).

   c) El dispatcher continúa la simulación del sistema, procesando los mensajes cuyo tiempo es menor o igual que el tiempo de simulación ($T_{absoluto} \leq T_{simulacion}$).

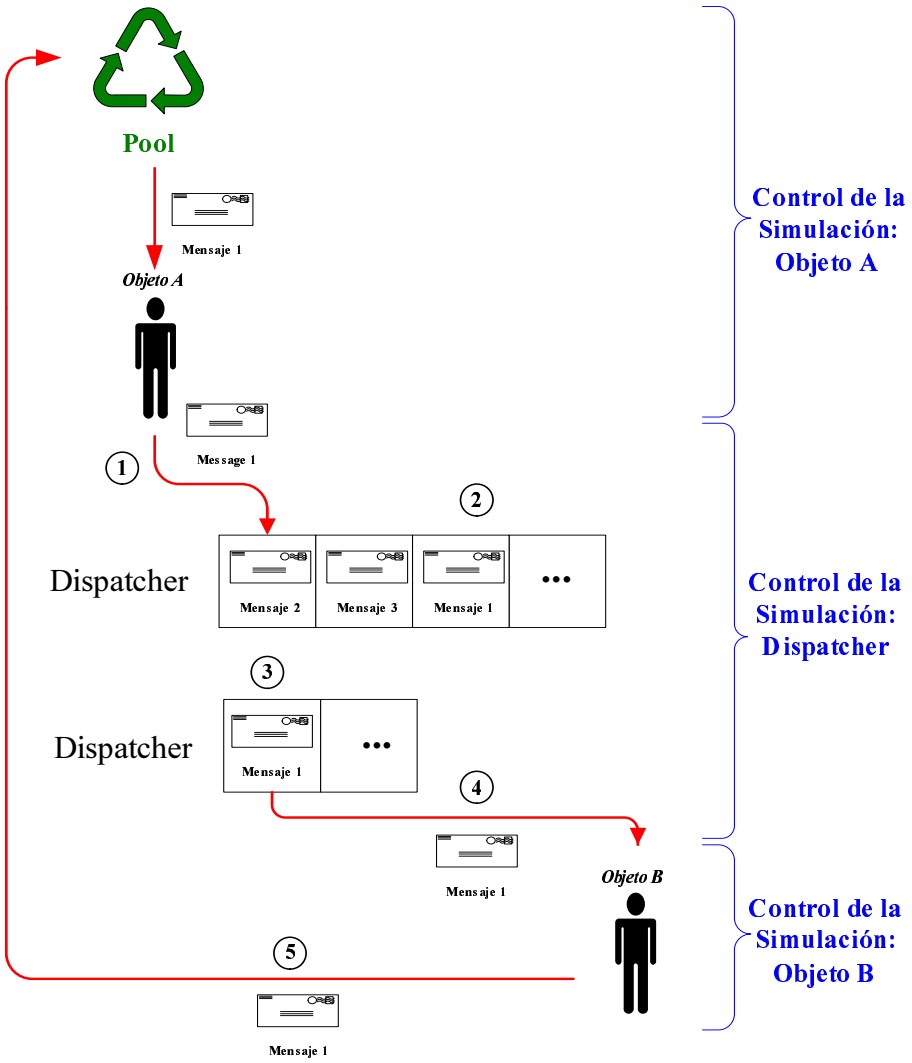

Figura 4.9: Mecanismo de paso de mensajes de DFly3D desde el punto de vista de GDESK

d) Cuando se alcanza el tiempo del mensaje $T_{absoluto}$ enviado por A, el mensaje se extrae del heap (figura 4.9 paso 3).

e) El dispatcher envía el mensaje a B, invocando la función de recepción de mensajes de B (figura 4.9 paso 4).

3. B recibe el mensaje.

   a) B no percibe que el mensaje proviene del dispatcher. Considera que el mensaje proviene de B.

   b) B actúa como consecuencia de la llegada del mensaje. Cambia su estado y podría enviar mensajes a otros objetos o a si mismo.

   c) B libera el mensaje (pool).

La figura 4.10 muestra el ciclo de vida de un mensaje.

4.5. Dinámica del Sistema

La dinámica de la simulación la comparten los objetos (independientemente de su tipo) y el gestor de mensajes o dispatcher (figura 4.11). Los mensajes son elementos pasivos de soporte al proceso de comunicación de los objetos.

El dispatcher toma el control de la simulación en dos situaciones:

1. **Función de simulación**: se ejecuta la función de simulación de GDESK desde el bucle principal de la aplicación. El dispatcher toma el control de la simulación. Cuando se arranca la simulación desde el bucle principal es porque hay, al menos, un mensaje pendiente de procesar. Por lo tanto, pasa a procesar el mensaje de cabeza del heap. Continúa el procesamiento de mensajes mientras haya mensajes pendientes de procesar.

2. **Envío de mensajes**: un objeto ejecuta la función de envío de mensaje a otro objeto. Cualquier objeto puede enviar y recibir mensajes de cualquier otro objeto, independientemente de su tipo (objeto del sistema u objeto del videojuego). Todos los mensajes que intercambian los objetos pasan previamente por el dispatcher, quien los almacena hasta que se alcanza el tiempo asociado al mensaje. Este proceso es transparente al objeto emisor y receptor. El objeto emisor sabe que ha enviado un mensaje a otro objeto, y el objeto receptor únicamente conoce que ha recibido un mensaje de otro objeto.

   El dispatcher almacena los mensajes de los objetos ordenadamente y los envía a los objetos receptores cuando el reloj del sistema alcanza el tiempo de ocurrencia del mensaje (tiempo en el que debe enviarse el mensaje). Cada vez que se procesa un mensaje se actualiza el tiempo de simulación con el tiempo del

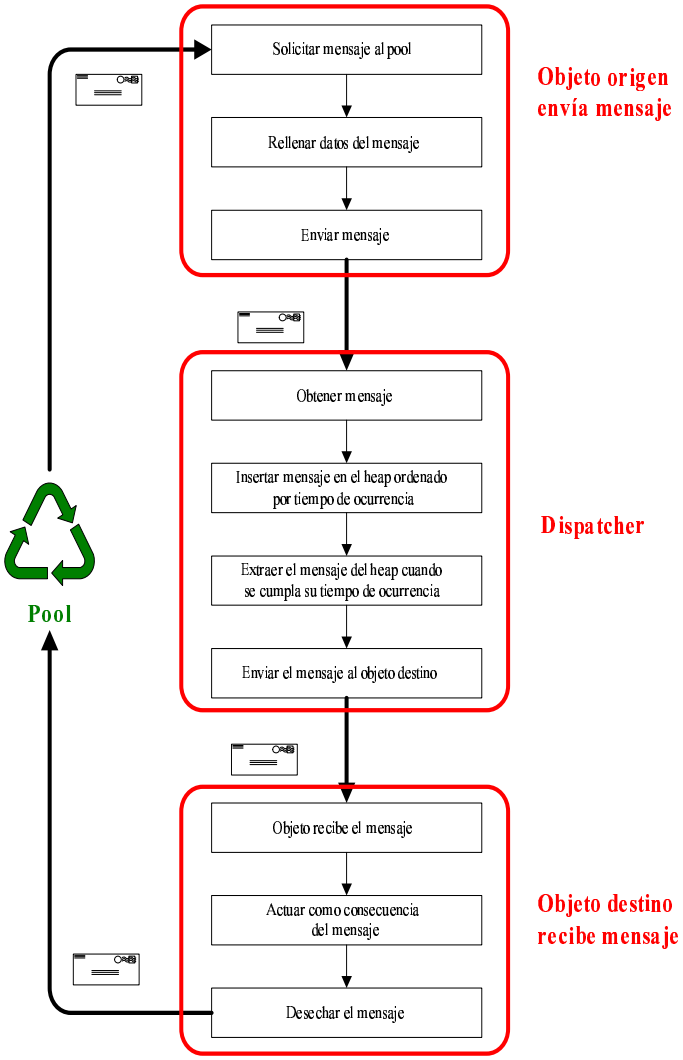

Figura 4.10: Ciclo de vida de un mensaje en DFly3D

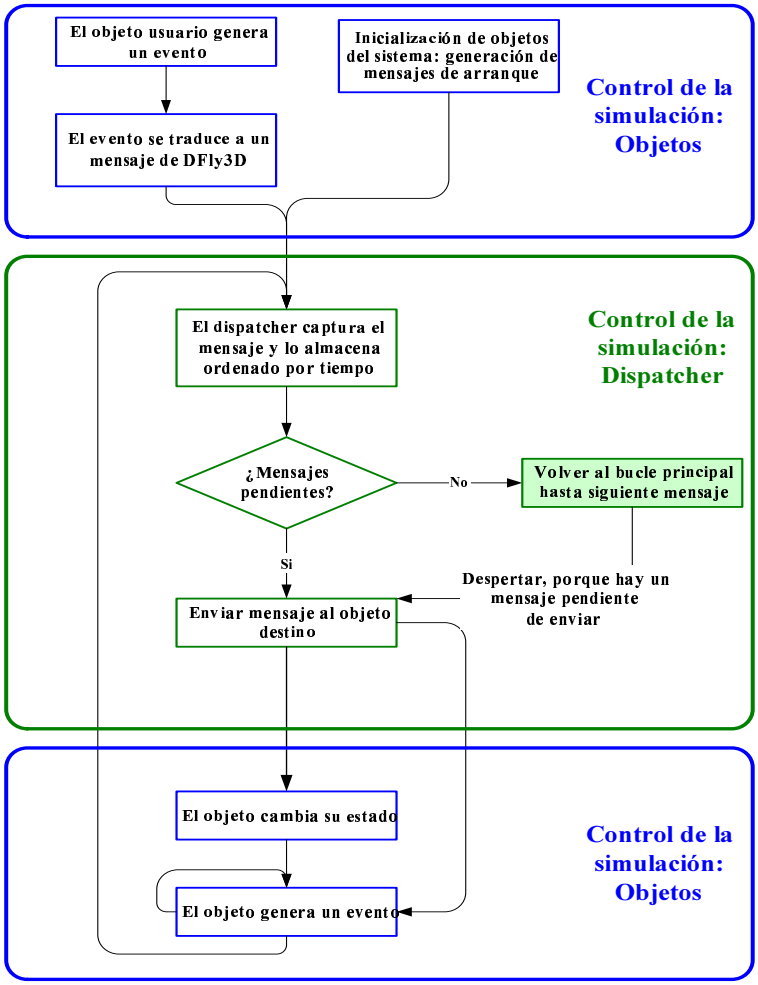

Figura 4.11: Dinámica del sistema en DFly3D

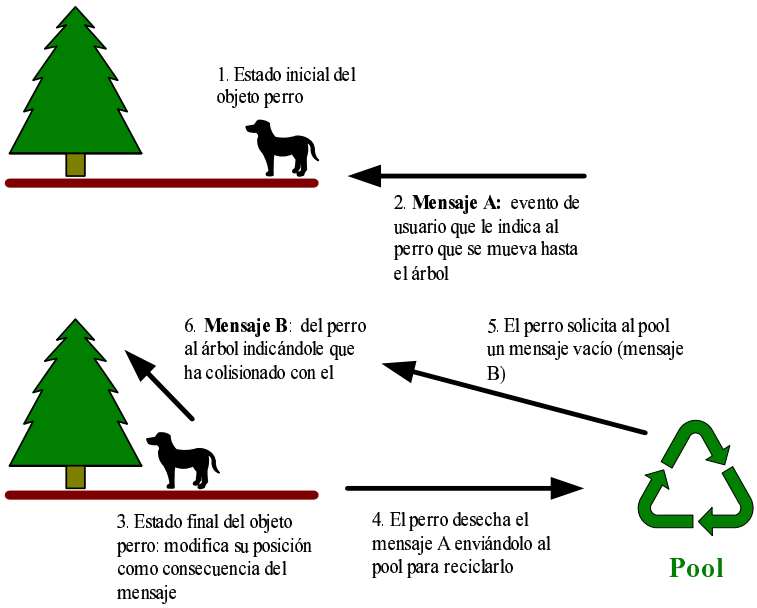

Figura 4.12: Comunicación mediante mensajes en DFLy3D

mensaje. El dispatcher ejecuta la función de recepción de mensajes del objeto destino. Esta es una función virtual que el objeto debe programar para definir su comportamiento. El objeto receptor pasa a tener el control de la simulación. Puede modificar su comportamiento dinámico o visual y enviar mensajes a otros objetos (si el cambio de comportamiento afecta a otros objetos) o a si mismo. El programador define los parámetros del mensaje. La función de recepción debería actuar consecuentemente a estos parámetros.

Cuando el objeto envía los mensajes o cuando finaliza el proceso que ha arrancado la recepción del mensaje, el control vuelve nuevamente al dispatcher. El dispatcher continúa con el procesamiento de mensajes mientras haya mensajes pendientes de enviar (ha vencido su tiempo de mensaje). Si no hay mensajes pendientes, el dispatcher finaliza su ejecución y el control vuelve al bucle principal. El control de la ejecución se devuelve únicamente durante el tiempo que resta para que venza el tiempo del siguiente mensaje. Cuando se cumpla el tiempo, el dispatcher volverá a tomar el control de la simulación. Si antes de que deba enviar el mensaje de cabeza llegase otro evento de usuario, tomaría el control de la simulación (insertando el mensaje en la posición correspondiente).

La figura 4.12 muestra un ejemplo de proceso de envío y recepción de mensajes. En el paso 1, con la llegada del mensaje, el control de la simulación pasa al objeto del videojuego *perro*, quien mantendrá el control de la simulación hasta el paso 6. Con el envío del mensaje, el control pasa al dispatcher.

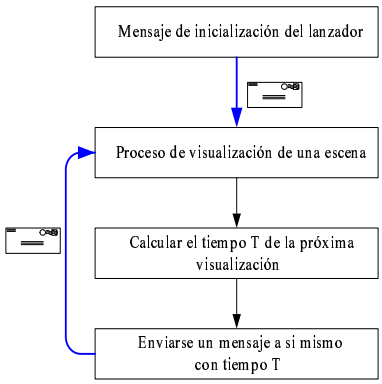

Figura 4.13: Visualización mediante paso de mensajes en DFly3D

4.6. Proceso de Visualización

DFLy3D permite la independencia del proceso de simulación del proceso de visualización (desacopla el sistema).

El proceso de visualización lo controla el objeto del sistema *visualizador*. Este objeto tiene las mismas características que el resto de objetos del sistema:

- Se comunica con el resto de los objetos mediante paso de mensajes.
- Modela su comportamiento mediante paso de mensajes.

El objeto *visualizador* (figura 4.13) arranca el proceso de visualización cuando recibe un mensaje. Cuando se inicializa el sistema, el objeto *lanzador* envía un mensaje al *visualizador* para que comience el proceso de visualización de la escena actual. Una vez se ha arrancado el comportamiento del *visualizador*, tiene un comportamiento autónomo. Cuando el objeto *visualizador* recibe un mensaje:

1. Visualiza la escena n.
2. Calcula el instante T_{n+1} de la visualización de la siguiente escena $n+1$.
3. Genera un mensaje, dirigido a si mismo que debe recibir transcurridas T_{n+1} unidades de tiempo.

El propio objeto *visualizador* decide el instante de la siguiente visualización. Por tanto la frecuencia de cuadro depende del número de mensajes generados por el objeto *visualizador*. Esta tasa de visualización puede ser fijada dependiendo de las necesidades del videojuego y las necesidades de dispositivo de visualización. Puede ser:

- **Tasa de visualización constante**: el tiempo de todos los mensajes del objeto *visualizador* es el mismo durante toda la ejecución del videojuego. El cálculo de cada escena se realiza a intervalos constantes.
- **Tasa de visualización adaptable**: el tiempo de cada mensaje del objeto *visualizador* puede ser variable. El programador define el mecanismo para determinar el tiempo de la siguiente visualización. Este mecanismo puede basarse en información del resto del sistema, como la carga del sistema. Este proceso lo define completamente el programador, por lo que la frecuencia de cuadro puede adaptarse dinámicamente a las condiciones del videojuego.

Sea:

SRR tasa de refresco de pantalla.

N_{RE} número de visualizaciones (escenas generadas por la aplicación, frecuencia de cuadro).

N_{SR} número de refrescos de pantalla (frecuencia de refresco).

T_{RS_n} tiempo de comienzo del proceso de cálculo de la visualización de la escena n.

T_{R_n} tiempo utilizado en visualizar la escena n.

T_{S_n} tiempo dedicado a la simulación entre dos visualizaciones consecutivas.

T_{SR_n} tiempo en el que se inicia el refresco de pantalla n.

T_{RE_n} tiempo de finalización del proceso de cálculo de la visualización de la escena n.

A pesar de que el proceso de visualización lo define y controla completamente el programador, los objetivos del objeto *visualizador* deben ser:

1. La frecuencia de cuadro debe adaptarse a la frecuencia de refresco de la pantalla. La relación entre las dos frecuencias depende del sistema concreto. Por ejemplo:
   - En un videojuego cuya salida gráfica es un televisor, en el sistema NTCS la frecuencia de refresco es de unos 30 fps y en el sistema PAL de unos 25 fps. En este caso, esta frecuencia está muy cercana a la frecuencia mínima para que el ojo humano visualice la escena correctamente, por lo que, se debe cumplir la ecuación 4.1:

$$N_{RE} \geq N_{SR} \tag{4.1}$$

- En un videojuego para PC con una tarjeta gráfica XGA, con frecuencias de refresco de 85 fps, la frecuencia de cuadro no es necesario que alcance la frecuencia de refresco (puede ser suficiente 50 o 30 fps). En este caso se cumple la ecuación 4.2. Si el número de visualizaciones es mayor se desperdicia potencia de cálculo.

$$N_{RE} \leq N_{SR} \tag{4.2}$$

2. El instante exacto de cada visualización debe permitir visualizar una escena antes del próximo refresco de pantalla (ecuación 4.3). Esto supone que en cada refresco de pantalla se debe mostrar siempre la última escena calculada. La ecuación 4.4 da el valor de la tasa de refresco de pantalla.

$$T_{RS_n} + T_{R_n} \leq T_{SR_n} \tag{4.3}$$

$$SRR = \frac{1}{T_{SR_{n+1}} - T_{SR_n}} \tag{4.4}$$

3. Evitar calcular visualizaciones que nunca se mostrarán por pantalla.

Desacoplar el sistema permite evitar visualizaciones innecesarias, lo que es especialmente útil en sistemas con baja potencia de cálculo (ecuación 4.5). El objeto visualizador puede decidir generar un mensaje de visualización en un intervalo de refresco si sabe que hay posibilidades de que esa escena se muestre en pantalla. También puede decidir no calcular una escena en este intervalo de refresco, sino pasar al siguiente ciclo directamente y continuar simulando.

$$T_{S_n} + T_{R_n} > T_{SR_{n+1}} - T_{SR_n} \tag{4.5}$$

El objeto *visualizador* puede comunicarse con otros objetos del sistema para, por ejemplo, adaptar el comportamiento de otros objetos a las necesidades del sistema.

Dependiendo de la complejidad de la escena, de la carga del sistema, del cambio entre escenas, se puede alcanzar la ecuación 4.6. La ecuación 4.7 da el tiempo de visualización de la escena n.

$$T_{R_n} \simeq T_{R_{n+1}} \Leftrightarrow T_{R_n} = T_{R_{n+1}} + k \tag{4.6}$$

$$T_{R_n} = T_{RE_n} - T_{RS_n} \tag{4.7}$$

De acuerdo con la ecuación 4.3:

$$T_{SR_n} \leq T_{RS_{n+1}} \leq T_{SR_{n+1}} - T_{R_{n+1}} \tag{4.8}$$

Una vez se ha visualizado la escena n, el objeto de *visualizador* se envía un mensaje a si mismo para realizar la visualización de la siguiente escena $(n+1)$. Este mensaje necesita un tiempo de mensaje, $T_{RS_{n+1}}$. El tiempo en el que se comienza la visualización de la escena $n+1$ debe ser posterior al refresco de pantalla n (ecuación 4.8).

Para ajustar el estado actual de simulación al que realmente se visualiza (ecuación 4.9):

$$T_{RS_{n+1}} = T_{SR_{n+1}} - T_{R_n} - \delta \tag{4.9}$$

Siendo δ una estimación de la cantidad de tiempo $T_{R_{n+1}}$ que variará en función de las escenas visualizadas anteriormente. δ es una estimación de k (ecuación 4.6). No es posible determinar exactamente el tiempo necesario para visualizar la escena actual, por lo que debe hacerse una estimación. La estimación del tiempo de visualización puede basarse en la historia previa, el mayor tiempo de visualización posible en el videojuego, un valor obtenido usando alguna clase de herramienta de predicción,... Trabajos como [Wimmer:2003] tratan el tema de la predicción del tiempo de visualización en tiempo real.

En un sistema adaptativo se pueden utilizar técnicas de control de procesos [Ogata:2003] de lazo abierto y lazo cerrado para calcular δ. En este sistema se utiliza localidad espacial y tiempo real para predecir δ, para determinar el tiempo disponible para realizar la simulación del sistema.

La aproximación más simple de δ puede ser (ecuación 4.10):

$$\delta = T_{SR_{n+1}} - T_{SR_n} - T_{R_n} \tag{4.10}$$

Usando esta aproximación, el tiempo de comienzo de la visualización de la escena $n+1$ es el tiempo de refresco n (ecuación 4.10).

$$T_{RS_{n+1}} = T_{SR_n} \tag{4.11}$$

Pero, esta situación supone que la escena mostrada en el refresco $n+1$ no muestra el estado real de la escena $n+1$. El valor ideal de δ debe permitir la ecuación 4.12.

$$T_{RE_{n+1}} \simeq T_{SR_{n+1}} \Leftrightarrow T_{RE_{n+1}} = T_{SR_{n+1}} + \alpha, \alpha \to 0 \tag{4.12}$$

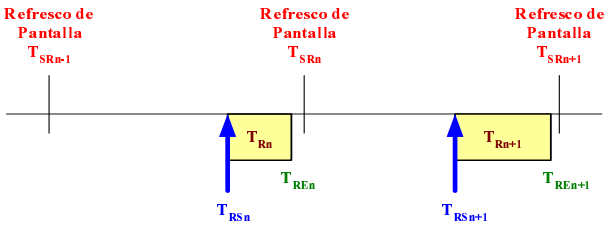

Figura 4.14: Mensajes de visualización en DFLy3D

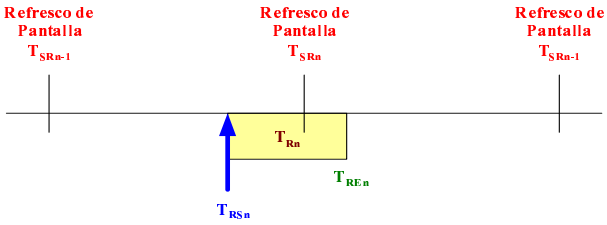

Figura 4.15: Fallo en la estimación del tiempo del visualización en DFly3D

T_{RS} se basa en un valor estimado. Por tanto, la aproximación depende de la precisión de δ. Hay dos posibles situaciones:

- El tiempo de visualización estimado es mayor o igual que el tiempo real de visualización (ecuación 4.13, figura 4.14). La escena actual se muestra correctamente por pantalla.

$$T_{RS_{n+1}} + T_{R_n} + \delta < T_{SR_{n+1}} \tag{4.13}$$

- El tiempo de visualización estimado es menor que el tiempo real de visualización (ecuación 4.14, figura 4.15). La escena actual no se muestra por pantalla, por lo que se produce una visualización fallida.

$$T_{RS_{n+1}} + T_{R_n} + \delta \geq T_{SR_{n+1}} \tag{4.14}$$

Varias situaciones pueden producir una visualización fallida:

- Una predicción incorrecta del tiempo de visualización, producida, por ejemplo, por un cambio radical de escena a visualizar.
- Una carga de simulación alta o una carga de visualización alta (ecuación 4.5). En esta situación, el sistema no es capaz de ejecutarse correctamente. Debe cambiarse el diseño del videojuego, el número de escenas generadas por unidad de tiempo,...

El objeto *visualizador* necesita información sobre el resultado del proceso de visualización: si se ha podido mostrar la escena en el refresco actual. El objeto *visualizador* debe recopilar información del proceso de visualización para ajustar su comportamiento. El valor de δ estimado debe actualizarse para evitar visualizaciones fallidas. Además, debe asegurar que el siguiente refresco no producirá una visualización fallida. El mecanismo para estimar δ debe modelarse siguiendo un autómata de estados, tomando en consideración la información del estado actual y de los estados anteriores. Por otro lado, este mecanismo puede obtener la diferencia media entre el tiempo estimado y el tiempo real, incrementando la futura estimación en la diferencia de las medias.

Por tanto, los sistemas discretos desacoplados permiten evitar visualizaciones innecesarias en sistemas con baja potencia de cálculo (donde se cumple la ecuación 4.5). El objeto *visualizador* puede decidir generar mensajes en un intervalo de refresco si considera que existe la posibilidad de mostrar la escena en el próximo refresco o pasar al siguiente.

4.7. Monitorización del Sistema

La monitorización [Loukides:1992] [Lilja:2000] permite comprobar si hay potencia de cálculo suficiente para ejecutar la aplicación gráfica manteniendo la calidad de servicio. En el caso de que sea insuficiente, la monitorización debe indicar cual es el proceso o procesos que hacen disminuir la calidad de servicio.

"Un monitor es una herramienta usada para observar la actividad de un sistema. El monitor observa las prestaciones del sistema, recoge información estadística, analiza los datos y muestra los resultados" [Jain:1991]. Monitorizar el sistema es una herramienta útil en videojuegos para detectar ineficiencias, distribuciones de carga o descompensaciones con el objetivo de ajustar el comportamiento del videojuego. Uno de los objetivos de la tesis es poder definir una calidad de servicio para cada uno de los elementos de la aplicación gráfica. Tener una herramienta de monitorización del sistema permite comprobar si el objetivo se cumple.

Son numerosos los estudios sobre monitorizado y depuración de aplicaciones en tiempo real [Nahrstedt:1996] [Mueller:1997] [Albertsson:2001]. Sin embargo, estos estudios se refieren a detección de cuellos de botella, utilización de CPU,... Pero no existen estudios específicos sobre monitorización en videojuegos y/o en aplicaciones gráficas en tiempo real. Tampoco los videojuegos estudiados disponen de herramientas de monitorización.

Una aplicación gráfica en tiempo real debe ser capaz de distribuir su potencia de cálculo de forma no uniforme entre todos los objetos del sistema en función de su carga propia, de forma que satisfaga la calidad de servicio definida para cada uno de los objetos y procesos del sistema. Si todas las calidades de servicio se cumplen,

el videojuego se ejecuta apropiadamente.

El objetivo de la inclusión de la monitorización del sistema en la tesis no ha sido definir la monitorización más óptima posible de un videojuego, sino mostrar las ventajas de la discretización del sistema en el proceso de monitorización. Por ello, se han incluido diferentes tipos de monitorización, mostrando ejemplos de su uso. La monitorización es un proceso discreto, de forma que cada aspecto de la monitorización tiene su propia frecuencia de muestreo independiente. Los resultados de la tesis (capítulo 5) se han obtenido utilizando las herramientas de monitorización definidas en el presente capítulo.

4.7.1. Balanceo de Carga

El monitorizado de videojuegos puede ser necesario en dos fases diferentes:

1. Fase de producción del videojuego.
2. Fase de ejecución.

4.7.1.1. Monitorización de la Fase de Producción del Videojuego

Durante el proceso de producción del videojuego son necesarios ajustes en la programación del videojuego. Estos ajustes pueden implicar al diseño visual de los personajes u objetos (como número de polígonos o texturas) o a su simulación (como inteligencia artificial o detección de colisiones). La monitorización del sistema permite conocer si hay potencia de cálculo suficiente para simular y visualizar las escenas convenientemente. La carga del videojuego se ajusta según los resultados de la monitorización. Un resultado especialmente útil de esta monitorización es el coste temporal de la visualización y de la simulación de cada objeto del videojuego, para tomar posteriormente las decisiones de diseño oportunas, así como de la aplicación en su totalidad, tiempos muertos o porcentaje de uso de la CPU.

4.7.1.2. Monitorización de la Fase de Ejecución

Una vez el videojuego está finalizado y el usuario final lo está ejecutando, la monitorización tiene dos utilidades:

- Ajustar la configuración del videojuego antes de su ejecución.
- Ajustar la carga del videojuego dinámicamente durante su ejecución.

Ajustar la Configuración del Videojuego antes de su Ejecución

La monitorización para llevar a cabo estos ajustes suele ser una mera comprobación del hw de la máquina y/o del sistema operativo donde se ejecuta el videojuego.

Permite definir ciertos elementos de la interfaz entre el sistema operativo y el videojuego, adaptando el videojuego a la máquina en concreto donde se va a ejecutar o a los requerimientos del usuario. Depende de la configuración del sistema. Esta parametrización la realiza el usuario, no es decidible por el programador.

Estos ajustes existen en prácticamente todos los videojuegos del mercado, en concreto, están incluidos en el núcleo original Fly3D. Al ser un proceso externo a la ejecución del videojuego, no se ha discretizado y por tanto no se ha incluido en DFly3D. Se puede seguir utilizando la herramienta de configuración original de Fly3D.

En concreto, la utilidad de configuración de Fly3D, le permite al usuario definir parámetros como la resolución de pantalla, el modo de visualización, la ubicación de plugins y ficheros de datos o la velocidad del ratón y joystick.

Ajustar la Carga del Videojuego Dinámicamente durante su Ejecución

Estos ajustes corresponden a la monitorización dinámica del sistema o en tiempo real. El videojuego puede estar programado para que se adapte dinámicamente a la carga del sistema o las necesidades puntuales de carga de los objetos (no corresponde a una serie de parámetros definidos por el usuario, sino que es el programador del videojuego quien lo determina).

Esta monitorización está incluida en el núcleo de DFLy3D, pero es el programador del videojuego quien debe utilizarla para adaptar dinámicamente el videojuego, pues involucra a los objetos del videojuego (completamente definidos por el programador). Es, por tanto, un ajuste personalizado.

Un ejemplo de utilización de ajuste dinámico podría ser: un videojuego puede contener un personaje que requiere una carga de visualización muy alta. Cuando este personaje está en la escena actual, podría programarse el videojuego para que el resto de los objetos reduzcan temporalmente su carga de visualización, su carga de simulación, su frecuencia de muestreo o el empleo de técnicas de multirresolución, entre otros aspectos.

El programador puede utilizar este ajuste dinámico en todos los objetos, de forma que si la potencia de cálculo es insuficiente, cada uno de los objetos del videojuego se adapten a la situación, El programador puede definir una forma diferente de adaptación para cada objeto, por ejemplo, variando la frecuencia de cuadro, la frecuencia de muestreo del objeto, variando la carga de visualización o simulación: multirresolución, comportamiento menos predictivo (nivel de podado del árbol de búsqueda),... De esta forma cada objeto define una nueva calidad de servicio degradada, pero acorde a su comportamiento.

4.7.2. Tipos de Monitorización de la Aplicación Gráfica

DFly3D incluye la posibilidad de monitorizar el sistema. La monitorización está controlada por un objeto del sistema específico: el objeto *monitor*.

El proceso de monitorización, a pesar de estar controlado por GDESK, se define durante el proceso de integración, pues depende del núcleo de aplicaciones gráficas donde se integra. En el apartado 3.3.8 del capítulo 3 se habla de dos tipos de monitorización: monitorización del propio GDESK y monitorización de la aplicación gráfica donde se integra (en este caso, la monitorización de DFLy3D). En el apartado actual se trata únicamente la monitorización de la aplicación gráfica donde se integra GDESK. En adelante sólo se hará referencia a esta monitorización.

Se definen dos tipos de monitorización de la aplicación gráfica en DFLy3D:

1. Monitorización en línea.
2. Monitorización fuera de línea.

La monitorización de DFly3D, al igual que la de GDESK, se selecciona mediante sentencias de preprocesador. Permite seleccionar de forma independiente la monitorización en tiempo real y la monitorización mediante históricos o trazas de la simulación. Cada uno de estos tipos de monitorización permite obtener diferentes tipos de resultados, de forma que se monitorizan aspectos diferentes de la aplicación. Se puede seleccionar que categorías de resultados se desea obtener en cada tipo de monitorización. De esta forma se evita sobrecargar el sistema con el coste de monitorización cuando no se va a utilizar.

4.7.2.1. Monitorización en Línea

El objeto *monitor* permite monitorizar el sistema en tiempo real de dos formas diferentes (ambos tipos de monitorización se seleccionan de forma independiente):

1. Mediante parámetros del sistema.
2. Mediante mensajes de control.

Monitorización en Línea Mediante Parámetros del Sistema

El *monitor* contiene una serie de parámetros con información del sistema que pueden ser consultados por los objetos de DFly3D directamente (tanto objetos del sistema como objetos del videojuego). Estos parámetros los actualiza el objeto monitor con una frecuencia definida inicialmente por el programador del videojuego pero que puede ser ajustada por el usuario. Para cada parámetro del *monitor* se define

una holgura, de forma que sólo se consideran cambios en el parámetro en intervalos superiores a la holgura. Así se evita que pequeñas variaciones del sistema puedan afectar a los objetos.

Los objetos pueden consultar estos parámetros directamente y utilizarlos para ajustar su comportamiento.

Los parámetros que calcula el objeto monitor son:

- *Frecuencia de cuadro instantánea* (actual).
- *Frecuencia de cuadro media*. Se define un periodo de inicialización de medidas, de forma que la frecuencia de cuadro media sólo considera el último intervalo (para evitar transitorios).
- *Porcentaje de colapso del sistema*. Cuando el sistema se colapsa, se ralentiza. Esta medida se obtiene midiendo el desfase de tiempos definidos para los mensajes y los tiempos reales de ejecución.
- *Tiempo libre en el sistema*. Si el sistema no está colapsado, se libera tiempo. Esta medida indica el tiempo medio liberado desde la última inicialización de medidas. Este valor pueden utilizarlo los objetos para conocer si el sistema está cercano al colapso (y actuar consecuentemente para que el colapso no se produzca).

Un ejemplo de uso de los parámetros del monitor puede ser: un determinado objeto del videojuego consulta si el sistema está colapsado y si lo está, cambia su comportamiento para contribuir a que el sistema no se colapse. El objeto puede definir una frecuencia de consulta de los parámetros del monitor diferente de su frecuencia de muestreo de comportamiento, de forma que se sobrecargue el sistema lo menos posible.

Monitorización en Línea Mediante Mensajes de Control

El objeto monitor puede enviar mensajes a los objetos (broadcast) cuando se cumplen determinadas condiciones del sistema (por ejemplo, cuando la frecuencia de cuadro desciende en un determinado porcentaje sobre la frecuencia de cuadro definida). Los mensajes se envían a todos los objetos, sean objetos del sistema o del videojuego.

Cada objeto sensible a los mensajes de control del monitor debe tener definida su respuesta al mensaje en la función de recepción del mensaje. Recuérdese que si el objeto no tiene asociado un comportamiento al mensaje, éste es, simplemente, obviado.

Un ejemplo de funcionamiento podría ser el siguiente: cuando el objeto monitor detecta que el sistema se está ralentizando más de un 20 % (el sistema está colapsado y el factor de colapso indica una ralentización de esta magnitud), envía un mensaje a todos los objetos del sistema indicándoselo. La respuesta a este mensaje puede ser diferente en cada objeto. Un objeto podría decidir, por ejemplo, bajar su frecuencia de refresco en un porcentaje, otro podría cambiar de método de detección de colisiones a un método menos preciso pero con un coste interior.

El objeto monitor puede comunicarse con los objetos del sistema y el resto de objetos con este. Actualmente se han definido, a modo de ejemplo, dos posibilidades de comunicación de los objetos del sistema o del videojuego con el objeto *monitor*:

- *Inicialización de parámetros*: cuando el *monitor*recibe un mensaje de inicialización, provenga de quien provenga, inicializa las medidas tomas hasta el momento para calcular los parámetros del sistema.
- *Modificación de la holgura o frecuencia de las medidas*: los parámetros del mensaje le indican al monitor que holgura o frecuencia modificar y como modificarla. Este mensaje puede provenir de cualquier objeto.

Un ejemplo de utilización podría ser: un determinado objeto utiliza muy frecuentemente el parámetro que indica la frecuencia de cuadro media y le interesa que esa medida se calcule tomando en cuenta un mayor intervalo de historia previa. Para ello, le envía un mensaje al monitor para que baje la frecuencia de inicialización de las medidas.

4.7.2.2. Monitorización Fuera de Línea

Esta monitorización produce como salida una serie de históricos o trazas de la simulación. Esta monitorización es útil durante el proceso de producción del videojuego.

DFly3D permite monitorizar:

- Detección de colisiones.
- Tiempos del sistema.
- Evolución de la frecuencia de cuadro durante la ejecución.
- Traza completa del sistema.

La monitorización incluye los objetos del sistema y los objetos del videojuego (creados por el usuario). Para ello se incluyen funciones de monitorización que el

usuario puede invocar, de forma que los resultados de la monitorización incluyan aspectos generales o concretos de su videojuego. Pueden añadirse fácilmente nuevos elementos a monitorizar siguiendo el mismo proceso definido para los elementos existentes.

Cada una de estas monitorizaciones se seleccionan por separado.

Detección de Colisiones

La monitorización de la detección de colisiones muestra cada una de las llamadas a la detección de colisiones de DFly3D (proceso de Fly3D). Se muestra el objeto que arranca la detección de colisiones y su posición en el momento de solicitar la detección de colisiones.

La monitorización de la detección de colisiones produce como resultado la información sobre las solicitudes de detección de colisiones y el éxito o fracaso de éstas.

Se incluye una utilidad que permite comparar dos ficheros de detección de colisiones. Esta utilidad tiene como objetivo comparar los ficheros de colisiones de diferentes ejecuciones del programa (o la ejecución del sistema continuo y discreto). Ha sido especialmente útil durante la integración de GDESK para comprobar el funcionamiento del sistema discreto respecto al continuo. Muestra:

1. Valor del número de llamadas a la detección de colisiones en cada ejecución.
2. Valor del número de colisiones detectadas en cada ejecución.
3. Comparación de colisiones de ambas ejecuciones, indicando las que son similares y las que difieren. Se indica la referencia a la posición de las colisiones en el fichero que contiene las llamadas a la detección de colisiones, por si es necesario comprobar la historia previa.

Un ejemplo de utilización de esta monitorización podría ser para comprobar si se están detectando las colisiones que realmente deberían ocurrir. Se puede obtener estas medidas con diferentes frecuencias de muestreo de los objetos hasta llegar al muestreo mínimo que permita detectar las colisiones adecuadamente.

Tiempos del Sistema

La monitorización de tiempos del sistema es la que puede ofrecer una información más interesante, pues muestra cuellos de botella del sistema o posibles colapsos.

Se generan dos ficheros, el primero contiene tiempos del núcleo de GDESK (ver apartado 3.3.8 del capítulo 3). El segundo contiene los tiempos de DFLy3D. Contiene información sobre los procesos de simulación de cada objeto, visualización del sistema, otros procesos como gestión de los mapas de luces, intercambio de buffers,...

La monitorización muestra el consumo de tiempo de diferentes procesos del sistema. Para estos procesos se muestra:

- Número de ejecuciones totales.
- Ciclos de reloj consumidos.
- Segundos consumidos.
- Segundos consumidos en cada ejecución.

Los procesos de los que se obtiene esta información son:

- Función de simulación de cada objeto del videojuego.
- Función de visualización de cada objeto del videojuego.
- Proceso global de visualización de todos los objetos.
- Tiempo consumido por los objetos en la detección de colisiones.
- Tiempo consumido por el intercambio de buffers de pantalla.
- Tiempo global que el sistema está siendo controlado por GDESK.
- Tiempo consumido por la gestión de mapas de luces y niebla.

Además, se muestran estos valores para el total de la aplicación.

Uno de los datos más importantes que muestra este fichero es el tiempo real consumido por la simulación del sistema. Si el tiempo real supera el tiempo de simulación, el sistema está colapsado en mayor o menor grado. El resto de tiempos pueden dar información sobre cual es la fuente del colapso.

Un ejemplo de uso de esta monitorización puede ser: el programador del videojuego obtiene los tiempos del sistema y comprueba que el sistema se colapsa en un porcentaje inaceptable para la calidad de servicio esperada. Por ello consulta cual o cuales son los tiempos de visualización y simulación de cada uno de los objetos, buscando las razones del colapso. Una vez detectados los motivos, cambia el diseño o el comportamiento de uno o más objetos.

Frecuencia de Cuadro

Se genera un fichero diferente para el sistema continuo y para el sistema discreto. Estos ficheros muestran la evolución de la frecuencia de cuadro del sistema, con los porcentajes de cada frecuencia de cuadro generada durante la ejecución. Permiten comprobar si el sistema tiene unas condiciones aceptables de visualización.

Podría utilizarse, por ejemplo, para comprobar la dispersión de las frecuencias de cuadro instantáneas del sistema, así puede verse si la frecuencia de cuadro media ha sido más o menos constante durante la ejecución.

Traza del Sistema

Se incluye una traza general que muestra la evolución completa del sistema. Muestra todos los eventos generados por el sistema, incluyendo origen, destino del mensaje y tiempo del mensaje. Sigue todo el proceso de inserción del mensaje en el heap (cuando el mensaje está insertado muestra el árbol del heap). Continúa con el proceso del dispatcher, enviando los mensajes del heap a los objetos receptores.

Se muestra el tiempo consumido en la ejecución de cada aspecto de cada objeto y para cada evento. Muestra también como se realiza la absorción de tiempos para cada uno de los eventos del sistema.

La traza incluye todos los procesos que se invocan durante la ejecución, por lo que mediante la traza puede conocerse costes de un proceso de visualización en concreto, o de un proceso de simulación de un objeto concreto, del mecanismo de intercambio de buffers,...

Esta traza se selecciona con una sentencia de preprocesador. Es un proceso con un coste temporal muy elevado por los que la simulación se ralentiza mucho, pero ofrece una perspectiva detallada de lo que ocurre en el sistema.

4.8. Conclusiones

La integración de GDESK en Fly3D permite obtener DFly3D, un motor de aplicaciones gráficas en tiempo real discreto desacoplado.

Una aplicación en DFly3D es un conjunto de objetos que se comunican mediante paso de mensajes. El mecanismo de paso de mensajes modela tanto comportamientos continuos como comportamientos discretos de los objetos. Este mecanismo tiene dos utilidades: modelar la interacción entre objetos y modelar el comportamiento de los objetos. El proceso de envío y recepción de mensajes lo controla GDESK.

Cada objeto tiene asociada una función de recepción de mensajes. En esta función se define la respuesta del objeto a un tipo de mensaje determinado (con unos parámetros determinados, o de un objeto emisor determinado). En esta función se define también a que mensajes es sensible un objeto. Como respuesta a un mensaje, el objeto puede generar nuevos mensajes. Este proceso lo controla GDESK.

Cada mensaje tiene asociado un tiempo. Este tiempo indica el instante en que el mensaje debe ser recibido por el objeto destino. GDESK mantiene los mensajes ordenados por tiempo de ocurrencia hasta que se alcanza su tiempo.

El proceso de visualización requiere de un objeto específico que lo controle y se modela mediante el mecanismo de paso de mensajes. Por tanto por cada mensaje que recibe el objeto visualizador se genera una escena. De este modo se controla la frecuencia de cuadro del sistema. La visualización es un proceso más del sistema, con la misma prioridad que el resto de procesos. La frecuencia de cuadro la define el usuario (mientras el sistema no esté colapsado). La frecuencia de cuadro no tiene porque ser constante durante la ejecución de la aplicación, puede adaptarse dinámicamente a los requerimientos de la aplicación. Otros motores de visualización como *OpenGL Performer* permiten definir también la frecuencia de cuadro (ver apartado 2.1.2.1 del capítulo 2).

GDESK dota a DFly3D con la posibilidad de monitorizar el sistema de dos formas diferentes:

- Monitorización fuera de línea: se generan una serie históricos con los resultados de la monitorización del sistema, útil en el proceso de producción.
- Monitorización en línea: define mecanismos para que los objetos obtengan información del sistema y puedan ajustar dinámicamente su comportamiento.

La monitorización es un proceso discreto que define su propia calidad de servicio y, por tanto, el muestreo óptimo para cada uno de sus procesos, evitando sobrecargar el sistema innecesariamente.

El apartado 5.2 del capítulo 5 muestra los resultados de la integración de GDESK en Fly3D. Para ello se compara el sistema continuo acoplado Fly3D con el sistema discreto desacoplado DFly3D.

Capítulo 5

Resultados

En este capítulo se muestran los resultados del:

1. Simulador de eventos discreto DESK. De los dos simuladores utilizados como modelo en el proceso de creación de GDESK, QNAP se utiliza como referente a la hora de establecer la forma de definir el modelo de simulación y SMPL como referente de simulador veloz. Se comparan los resultados temporales obtenidos con el simulador SMPL y con JDESK (versión web de DESK).

2. Núcleo de aplicaciones gráficas DFLy3D. DFLy3D se crea partiendo del núcleo de aplicaciones gráficas Fly3D e integrando el simulador de eventos discreto GDESK para gestionar los eventos de la aplicación gráfica. Se comparan los resultados del núcleo de aplicaciones gráficas original continuo acoplado Fly3D y el núcleo discreto desacoplado DFly3D. Estos resultados muestran en qué aspectos mejora la simulación discreta al núcleo de aplicaciones gráficas.

5.1. Simulador de Eventos Discreto

Las pruebas se han realizado en un ordenador personal Pentium 4 (2 GHz) con 512Mb de Ram. Los compiladores utilizados han sido: Visual C++ v6.0 (DESK y SMPL) y BlueJ v1.3.0 (JDESK). Ambos compiladores ejecutándose sobre el sistema operativo Windows XP. No se muestran los resultados obtenidos con otras configuraciones de máquina pues no es el objetivo de la tesis comprobar el porcentaje exacto de mejora de DESK respecto a SMPL o JDESK, ni como depende esta mejora de la arquitectura de la máquina.

5.1.1. Comparativa de Resultados de DESK y SMPL

DESK se creó basándose en dos simuladores radicalmente diferentes: SMPL y QNAP, con el objetivo de permitir definir el modelo a simular de una forma similar a QNAP y ser rápido como SMPL. QNAP permite crear el modelo de simulación definiendo las ES que componen el modelo, sus propiedades y la topología del modelo. DESK permite definir el modelo de manera similar. Incluso la topología del modelo se define como una propiedad más de las ES. En cuanto al coste de simulación, en este apartado se va a demostrar que el objetivo de DESK de tener un coste de simulación bajo, como SMPL, no sólo se ha alcanzado, sino que se ha conseguido un coste de simulación menor para los modelos simulados. No se ha comparado el tiempo de simulación de DESK con QNAP, pues es un simulador lento. QNAP se ha considerado en esta tesis como referente a seguir para la definición del modelo del sistema a simular en DESK.

En este apartado se muestran una serie de sistemas modelados en DESK y en SMPL, con el objeto de poder comparar sus costes de simulación. DESK permite obtener resultados del modelo utilizando dos tipos de aritmética: en coma fija o en coma flotante. Para cada ejemplo se obtiene el coste temporal de la simulación, variando el tiempo de simulación o el número de clientes en el sistema, tanto en DESK como SMPL.

Para cada ejemplo se compara:

- SMPL vs DESK-Flotante: coste temporal de simulación del modelo en SMPL respecto a DESK en coma flotante.
- SMPL vs DESK-Fija: coste temporal de simulación del modelo en SMPL respecto a DESK en coma fija.
- DESK-Fija vs DESK-Flotante: coste temporal de simulación del modelo en DESK en coma fija respecto a coma flotante.

El rango de valores representables en coma fija no es tan amplio como el rango en coma flotante (para la representación de datos elegida de coma fija). Por ello, los resultados de los ejemplos en DESK-Fija no se han podido obtener para todos los valores de tiempo de simulación, debido a esta limitación en la representación numérica. En las tablas y las figuras se muestran los valores obtenidos para coma fija hasta el rango representable. El código fuente de estos ejemplos se encuentra en [Garcia:1997].

5.1.1.1. Ejemplo 1: Sistema Abierto M/M/1

El sistema de la figura 5.1 muestra un sistema muy simple con una fuente que genera clientes para la única CPU del sistema (M/M/1). La fuente genera clientes de

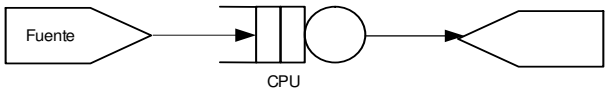

Figura 5.1: Sistema abierto M/M/1

Tiempo de simulación	SMPL vs DESK-Flotante	SMPL vs DESK-Fija	DESK-Fija vs DESK-Flotante
1.000.000	0 %	0 %	0 %
5.000.000	25 %	25 %	0 %
10.000.000	22,2 %	-	-
50.000.000	8,33 %	-	-
100.000.000	10,64 %	-	-

Tabla 5.1: Incremento de los costes de simulación de DESK respecto de SMPL del ejemplo 5.1 sin esperas

forma exponencial. La CPU tiene una política de inserción en cola de espera FIFO no expulsiva y los tiempos de servicio de los clientes en la CPU son exponenciales. Se varía la configuración de este sistema inicial para observar como los cambios influyen en el coste temporal de la simulación.

Sistema Inicial

En el sistema inicial, la tasa de servicio de los clientes en la CPU y la tasa de creación de clientes en la fuente son iguales, por lo que la longitud media de la cola de espera de la CPU es pequeña. Los costes temporales de la simulación de DESK (tabla 5.1 y figura 5.2) disminuyen respecto a los costes de SMPL (alrededor de un 10 %). Los tiempos de DESK usando coma fija y coma flotante son muy similares. La falta de precisión en la medida no permite apreciar la diferencia entre ambas implementaciones.

Sistema Abierto con Aumento en la Longitud Media de la Cola de Espera

Se aumenta el tiempo de servicio de los clientes en la CPU por lo que se producen esperas de clientes. La diferencia del coste temporal de la simulación entre SMPL y DESK aumenta (tabla 5.2 y figura 5.3), pasando a ser de un 150 %. Los tiempos de coma fija siguen siendo similares a los de coma flotante.

Sistema Cerrado

Se convierte el sistema en cerrado, eliminado la fuente. Los clientes no se crean ni se destruyen. Cuando un cliente sale de la CPU vuelve a pedirle servicio. Se simula un comportamiento parecido al sistema abierto pero sin fuente. La diferencia entre

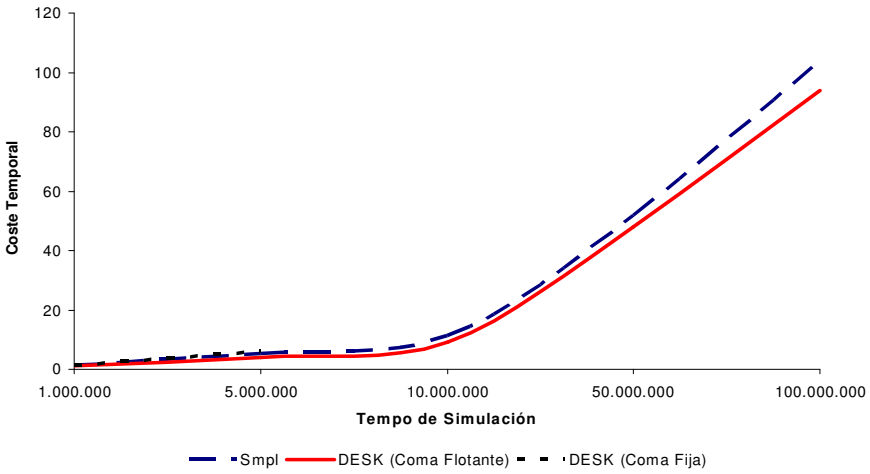

Figura 5.2: Costes de simulación de DESK y SMPL del ejemplo 5.1 sin esperas

Tiempo de simulación	**SMPL vs DESK-Flotante**	**SMPL vs DESK-Fija**	**DESK-Fija vs DESK-Flotante**
1.000.000	300 %	300 %	0 %
5.000.000	300 %	300 %	0 %
10.000.000	250 %	-	-
50.000.000	128 %	-	-
100.000.000	145 %	-	-

Tabla 5.2: Incremento de los costes de simulación de DESK respecto de SMPL del ejemplo 5.1 con esperas

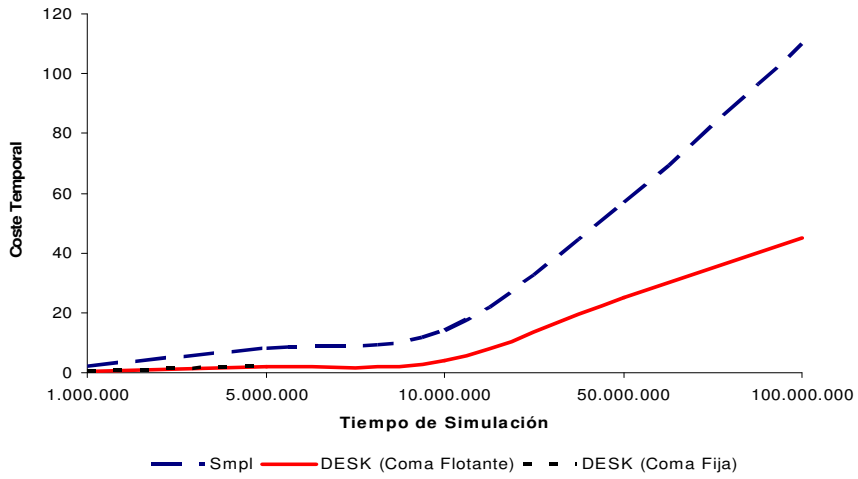

Figura 5.3: Costes de simulación de DESK y SMPL del ejemplo 5.1 con esperas

Tiempo de simulación	**SMPL vs DESK-Flotante**	**SMPL vs DESK-Fija**	**DESK-Fija vs DESK-Flotante**
1.000.000	0 %	0 %	0 %
5.000.000	134 %	75 %	34 %
10.000.000	100 %	-	-
50.000.000	77 %	-	-
100.000.000	83 %	-	-

Tabla 5.3: Incremento de los costes de simulación de DESK respecto de SMPL del ejemplo 5.1 cerrando el sistema

los costes de simulación entre SMPL y DESK aumentan respecto al sistema inicial (tabla 5.3 y figura 5.4), pasando a ser para DESK en coma flotante alrededor de un 80 % más rápido. La diferencia entre coma flotante y fija aumenta ligeramente. Los costes en SMPL son parecidos al sistema abierto, porque la creación de clientes es simplemente incrementar un contador.

Sistema con Recursos

Se añade un recurso al sistema abierto inicial. Un cliente, para poder recibir servicio en la CPU debe previamente obtener un recurso. El coste temporal de la simulación aumenta tanto en DESK como en SMPL (tabla 5.4 y figura 5.5), pero el incremento en DESK es mayor. El porcentaje de mejora de DESK en coma flotante respecto a SMPL pasa a ser un 12 %.

Sistema con Clientes Prioritarios

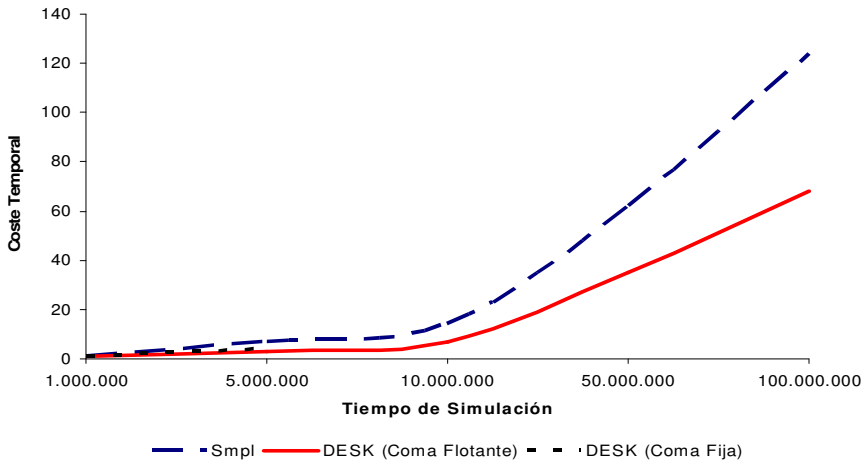

Figura 5.4: Costes de simulación de DESK y SMPL del ejemplo 5.1 cerrando el sistema

Tiempo de simulación	**SMPL vs DESK-Flotante**	**SMPL vs DESK-Fija**	**DESK-Fija vs DESK-Flotante**
1.000.000	100 %	0 %	100 %
5.000.000	13 %	13 %	0 %
10.000.000	14 %	-	-
50.000.000	10 %	-	-
100.000.000	12 %	-	-

Tabla 5.4: Incremento de los costes de simulación de DESK respecto de SMPL del ejemplo 5.1 con recursos

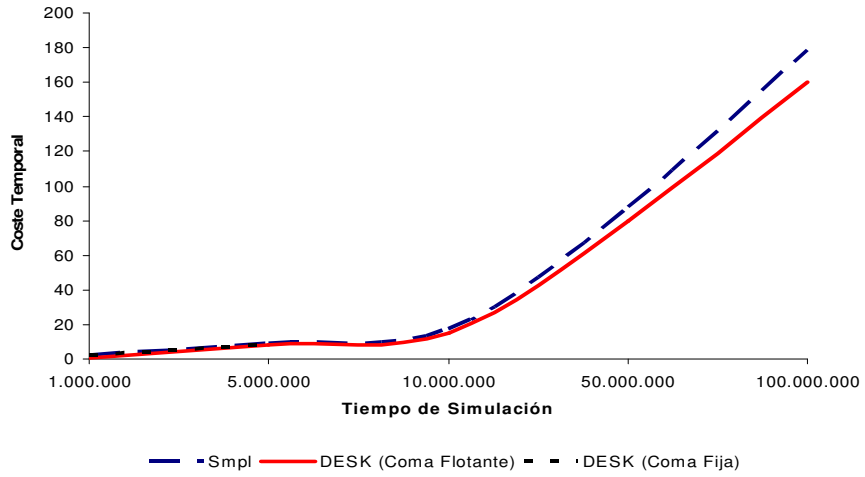

Figura 5.5: Costes de simulación de DESK y SMPL del ejemplo 5.1 con recursos

Tiempo de simulación	**SMPL vs DESK-Flotante**	**SMPL vs DESK-Fija**	**DESK-Fija vs DESK-Flotante**
1.000.000	0 %	0 %	0 %
5.000.000	25 %	0 %	25 %
10.000.000	57 %	-	-
50.000.000	44 %	-	-
100.000.000	43 %	-	-

Tabla 5.5: Incremento de los costes de simulación de DESK respecto de SMPL del ejemplo 5.1 con prioridades

Se modifica el sistema para utilizar clientes con prioridades. Se parte de un sistema cerrado con 20 clientes, la mitad de los cuales tienen prioridad 1 y la otra mitad prioridad 2. Los costes temporales de la simulación en DESK se incrementan ligeramente (tabla 5.5 y figura 5.6) debido al tiempo extra que supone la gestión de colas con prioridades. Los costes en SMPL también son parecidos, aumentando menos incluso que en DESK, por lo que la mejora de costes entre SMPL y DESK se reduce respecto al sistema cerrado, siendo en coma flotante de alrededor de un 44 %.

En todos las variaciones del sistema los costes temporales de DESK en coma flotante son menores que en SMPL, siendo considerable la diferencia de costes en algún caso. Los costes en DESK utilizando coma fija o coma flotante son muy similares. LA precisión de las medidas no permite apreciar la diferencia adecuadamente en algunos de los resultados.

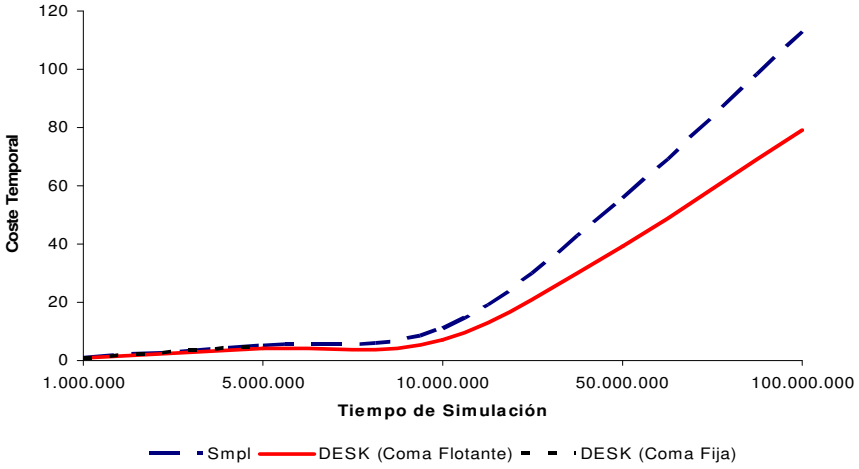

Figura 5.6: Costes de simulación de DESK y SMPL del ejemplo 5.1 con prioridades

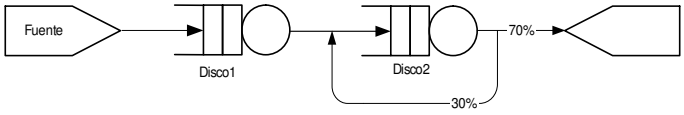

Figura 5.7: Sistema abierto con dos discos

5.1.1.2. Ejemplo 2: Sistema Abierto con dos Discos

La figura 5.7 muestra un sistema con dos discos y una fuente. Ambos discos sirven a los clientes durante un tiempo determinado por una distribución exponencial. La política de servicio de clientes en los discos es FIFO no expulsiva. La fuente genera clientes con una tasa exponencial. Los clientes, a la salida del primer disco, tienen una probabilidad del 30 % de ir al disco 2 y una probabilidad 70 % de salir del sistema.

El coste de la simulación en DESK en coma flotante es alrededor de un 15 % menor que en SMPL y un 46 % menor que en coma fija. A medida que el sistema se hace más complejo, la diferencia entre coma fija y flotante aumenta.

5.1.1.3. Ejemplo 3: Sistema Cerrado

La figura 5.9 muestra un típico sistema multiprocesador cerrado. El sistema contiene 10 terminales (hay un cliente por terminal). Desde los terminales se accede a dos CPU. A la salida de cualquiera de ellas, los clientes acceden a una tercera CPU (obteniendo previamente el recurso *BUS*). El sistema contiene, por tanto, 5 ES:

Tiempo de simulación	**SMPL vs DESK-Flotante**	**SMPL vs DESK-Fija**	**DESK-Fija vs DESK-Flotante**
1.000.000	0 %	0 %	0 %
5.000.000	8 %	-26 %	46 %
10.000.000	16 %	-	-
50.000.000	14 %	-	-
100.000.000	14 %	-	-

Tabla 5.6: Incremento de los costes de simulación de DESK respecto de SMPL del ejemplo 5.7

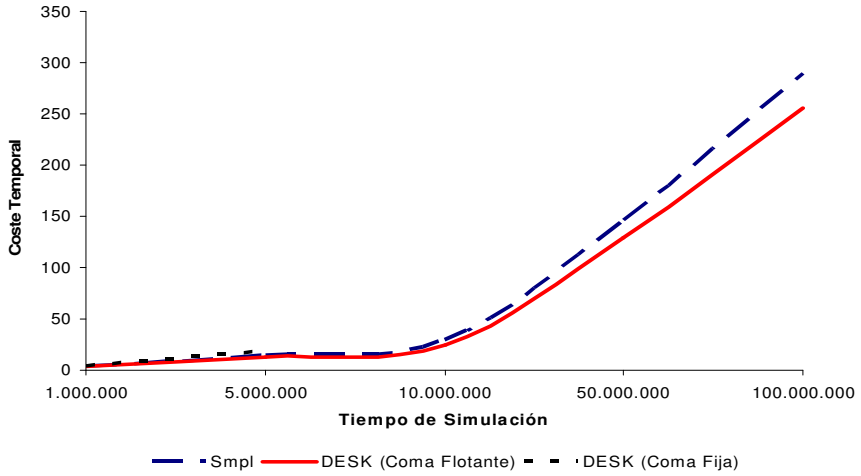

Figura 5.8: Costes de simulación de DESK y SMPL del ejemplo 5.7

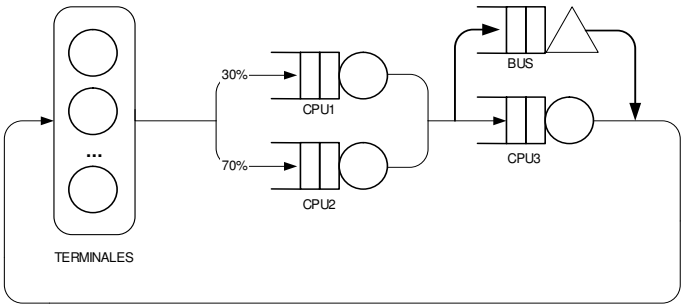

Figura 5.9: Sistema cerrado

Número de Clientes	SMPL vs DESK-Flotante	SMPL vs DESK-Fija	DESK-Fija vs DESK-Flotante
1	650 %	400 %	50 %
5	730 %	728 %	0 %
10	825 %	957 %	-13 %
100	1.277 %	1.671 %	-22 %
500	3.511 %	4.542 %	-22 %
1.000	6.755 %	8.714 %	-22 %
2.000	13.611 %	15.325 %	-11 %

Tabla 5.7: Incremento de los costes de simulación de DESK respecto de SMPL del ejemplo 5.9

procesadores, 3 CPU y el recurso *BUS*. Los clientes, cuando salen de *CPU3* liberan el recurso *BUS* y vuelven a los procesadores, iniciando de nuevo el recorrido. Todas las CPU tienen las siguientes características: un único servidor, política de inserción en cola de espera FIFO no expulsiva, tiempo de servicio de distribución exponencial y función de usuario no definida. Todos los clientes tienen la misma prioridad y clase. Cuando un cliente sale de los procesadores tiene un 30 % de posibilidades de ir a la *CPU1* y un 70 % a la *CPU2*.

Se ha realizado la misma simulación variando el número de clientes. El cuello de botella es el recurso *BUS*. Cuando aumenta el número de clientes, la longitud media de la cola de espera del recurso *BUS* crece considerablemente. El coste temporal de la DESK mejora con respecto a SMPL, sobre todo cuando el número de clientes aumenta (crece la longitud media de las colas de espera). A medida que aumenta el número de clientes, aumenta el coste de la simulación en DESK, pero este aumento es mucho mayor en SMPL. Para 5 clientes en el sistema DESK es 7 veces más rápido que SMPL, con 2.000 clientes es 136 veces más rápido. La tabla 5.7 y la figura 5.10 muestran la variación de los costes temporales de simulación. Los costes en DESK en coma fija empeoran respecto a los de coma flotante, en un 22 %.

El modelo del sistema en DESK, JDESK y SMPL se encuentra en el apéndice C. En este ejemplo se puede observar como se definen las ES y la sencillez de modificación del modelo a simular en DESK y JDESK.

5.1.1.4. Conclusiones

Comparativa de SMPL y DESK en Coma Flotante

DESK en coma flotante es más rápido que SMPL para los ejemplos probados, aunque el porcentaje depende mucho del modelo simulado. Ciertos elementos de la simulación ralentizan DESK respecto de SMPL, otros elementos hacen la simulación más rápida. Dependiendo de la combinación de aspectos que ralentizan la simulación

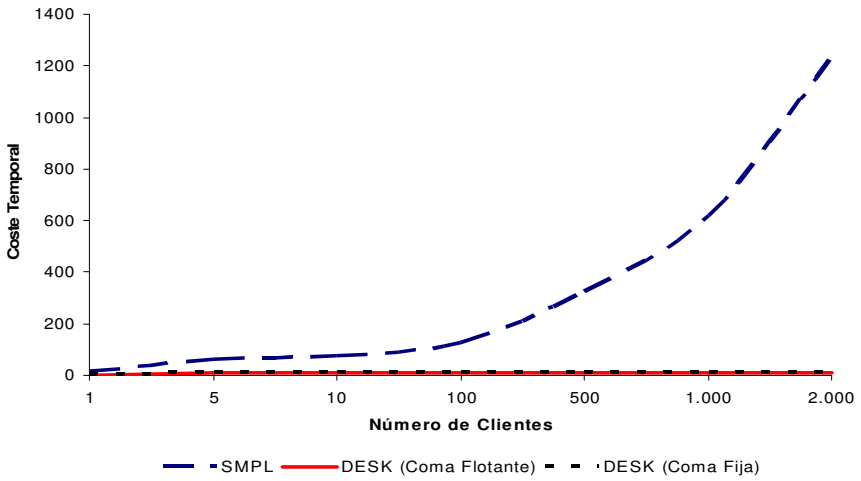

Figura 5.10: Costes de simulación de DESK y SMPL del ejemplo 5.9

y que la agilizan, varía el coste temporal de la simulación.

Cuando los costes de simulación de DESK son mayores que los de SMPL, nunca lo son en un porcentaje muy elevado. En cambio determinadas situaciones producen que los costes de SMPL sean cientos de veces mayores que los de DESK. En general, utilizar siempre DESK en vez de SMPL es más ventajoso. DESK tiene además otras ventajas que ya se han descrito anteriormente, como permitir llegar a simulaciones donde SMPL no llega, posibilidad de usarlo como prototipador,...

Algunos de los aspectos del sistema que hacen diferir el coste de simulación entre DESK y SMPL son:

- **Carga del Sistema**: cuando aumenta el número de clientes en el sistema o aumenta la longitud media de las colas de espera, los costes temporales de la simulación de DESK mejoran con respecto a SMPL. El aumento del coste de SMPL conforme aumenta el número de clientes es lineal, en cambio el coste de DESK se mantiene prácticamente constante. Esta diferencia se debe a que gestión de colas de espera en DESK es independiente del número de clientes (sólo la cola de eventos depende del número de clientes en el sistema).

  El coste de simulación depende del sistema a simular, de tópicos como el número de colas de espera, la longitud media de cada una, los algoritmos de inserción en cola de espera, el número de ES o la utilización de recursos.

  Además, el número de clientes en SMPL está limitado (2048 clientes simultáneamente en el sistema), mientras que en DESK la limitación está en la memoria disponible (la numeración de clientes admite hasta 4.294.967.295

valores diferentes, pero no hay límite de número de clientes). En este aspecto, como en otros, DESK llega donde SMPL no llega e incluso con mejores costes temporales.

- **Sistemas Abiertos o Cerrados**: la utilización de fuentes en DESK produce un aumento del coste de la simulación, respecto al sistema cerrado. Esto no significa que un modelo, por el hecho de tener una fuente sea más lento en DESK que en SMPL, únicamente que la diferencia entre los costes de ambos simuladores se acorta. En SMPL la creación de clientes puede ser tan sencilla como incrementar el contador de clientes. Esto siempre será más rápido que la gestión de creación de clientes de DESK. En el mejor de los casos existirán clientes en el pool y se tomará uno, pero ya se añade al simulador la desventaja de la inserción y eliminación de clientes de la cola del pool. Cuando SMPL utiliza clientes con clases o prioridades, el coste de simulación aumenta, pues se ha añadido un elemento más de gestión. En SMPL es necesario usar una estructura de datos externa siempre que los clientes usen prioridades, clases o vectores de ES. En DESK no es necesario añadir nada nuevo. DESK gestiona cualquier característica del cliente sin aumentar el coste de la simulación.

- **Prioridades**: en DESK el uso de prioridades ralentiza ligeramente el sistema. Para extraer un cliente de la cola de espera se busca desde la cola de mayor prioridad a la de menor prioridad. Cuando no hay prioridades, los clientes se insertan en la cola de prioridad mayor, por lo que los clientes siempre se encuentran en la primera cola consultada. Cuando se usan prioridades, los clientes se distribuyen por el resto de las colas, según sus prioridades.

- **Recursos**: la utilización de recursos supone ralentizar el sistema en DESK. Una ES tipo recurso mantiene una cola con los clientes que tienen asignados sus recursos. Un cliente puede mantener recursos reservados de cualquier ES tipo recurso, por lo que los punteros a las colas de recursos de cada ES no están en el propio cliente (como ocurre con el resto de colas del simulador). Los nodos de las colas de recursos se deben crear y destruir dinámicamente. Todo este proceso supone un coste adicional en la simulación. En SMPL no hay un tratamiento especial de los recursos, son ES convencionales, no necesitan estructuras adicionales. DESK permite que los clientes reserven más de un recurso de la ES, en SMPL únicamente se puede reservar un recurso. Esto supone añadir funcionalidad a DESK, pero a costa de aumentar el coste temporal de la simulación.

Comparativa de DESK en Coma Fija y Coma Flotante

La utilización de aritmética en coma fija no ha supuesto ninguna mejora en los costes de simulación, al contrario, los costes aumentan. Para sistemas sencillos los costes utilizando ambas aritméticas son muy similares. La diferencia aumenta

con la complejidad del sistema y con el número de clientes. La principal causa de esta situación es la arquitectura del procesador utilizado y el compilador, que optimiza los cálculos en coma flotante. La utilización de coma fija supone en este caso ralentizar la simulación con conversiones de tipos innecesarias y con implementación de operaciones ya optimizadas por el compilador y soportadas por el hardware nativo de la máquina.

El uso de coma fija impone restricciones en el sistema a simular, pues el máximo valor temporal representable es 2^{23}, mientras que en coma flotante es 2^{63}. Otra ventaja de usar coma flotante es que el tamaño del exponente y de la mantisa se adapta al dato a representar por lo que puede obtener una mayor precisión en la representación cuando el tamaño de la mantisa lo permite.

La generación de números aleatorios en coma fija puede emplear distintos mecanismos, la forma de generación que mayor precisión en la representación de datos produce es también la más lenta. El principal objetivo del simulador es minimizar costes temporales, por lo que se debe emplear la forma de calculo más rápida y menos precisa. Los resultados de la simulación en coma fija son menos precisos que los obtenidos mediante coma flotante, no solo porque cada representación de tiempo es menos precisa, sino porque el error en la representación de tiempos se va acumulando con cada evento (el reloj del simulador es el tiempo en el que se producen los eventos, a este valor se le añade el valor temporal en el que se debe producir el siguiente evento, con su error correspondiente).

Sin embargo, se incluye aritmética en coma fija para poder realizar simulaciones por software en ausencia de operadores en coma flotante, como PDAs. Las ventajas de la representación de datos en coma fija se indican en el capítulo 3.

5.1.2. Comparativa de Resultados de DESK y JDESK

Se han implementado los ejemplos anteriores en JDESK y se ha comparado el coste temporal de simulación de DESK y JDESK. A continuación se muestran los resultados obtenidos. La tablas 5.8, 5.9 y 5.10 muestran los costes temporales de simulación de los ejemplos de las figuras 5.1, 5.7 y 5.9 respectivamente.

JDESK proporciona unos costes temporales de simulación mucho mayores que DESK. Según modelos, oscila entre costes 15 veces mayores a 48 veces mayores, siendo el valor más habitual tener en JDESK un coste 20 veces superior a DESK. Sin embargo, estos costes pueden ser muy dependientes de los compiladores utilizados. No se aprecian diferencias significativas según la longitud de las colas de espera, recursos,... La tabla 5.10 muestra que los costes de JDESK para ese sistema en concreto, cuando aumenta el número de clientes difieren menos de los costes de DESK que en otros sistemas.

A pesar de que los costes de JDESK empeoran considerablemente respecto a los

Tiempo de simulación	Sistema Inicial	Aumento c. espera	Sistema Cerrado	Recursos	Prior.
1.000.000	2.298 %	758 %	1.860 %	3.983 %	2.141 %
2.000.000	2.131 %	1.654 %	1.784 %	2.664 %	2.134 %
3.000.000	2.181 %	1.878 %	3.185 %	2.368 %	3.369 %
4.000.000	2.120 %	4.102 %	2.480 %	2.219 %	2.970 %
5.000.000	2.918 %	4.873 %	3.105 %	2.462 %	2.796 %

Tabla 5.8: Incremento de los costes de simulación de JDESK respecto de DESK del ejemplo 5.1

Tiempo de simulación	JDESK vs DESK
1.000.000	1.947 %
2.000.000	2.279 %
3.000.000	2.118 %
4.000.000	2.220 %
5.000.000	2.164 %

Tabla 5.9: Incremento de los costes de simulación de JDESK respecto de DESK del ejemplo 5.7

Número de Clientes	JDESK vs DESK
1	672 %
5	615 %
10	663 %
100	636 %
500	655 %
1.000	679 %
2.000	675 %

Tabla 5.10: Incremento de los costes de simulación de JDESK respecto de DESK del ejemplo 5.9

costes de DESK, las ventajas de la simulación en web (apartado 2.3.4 del capítulo 2) justifica la utilización del simulador a pesar del empeoramiento de los costes.

5.1.3. Conclusiones

DESK es un simulador generalista de sucesos discreto implementado como una librería de C++. Los costes de simulación permiten utilizarlo como prototipador y la precisión de los resultados obtenidos permiten utilizarlo para obtener resultados finales del modelo a simular.

El modelo se define describiendo el sistema, sus componentes y su topología. La definición del modelo se realiza de forma modular.

Como ya se ha mencionado, es un simulador rápido y preciso, pero además es flexible y potente. Permite definir modelos con cualquier número de componentes y con cualquier comportamiento, por atípico que sea. No impone restricciones al modelo a simular. Permite incluir fácilmente elementos externos dentro de la simulación y realizar simulación en tiempo real. El modelo puede variarse dinámicamente dependiendo de cualquier condición del sistema, sin necesidad de redefinir el modelo. Puede cambiarse su topología, crear y destruir ES, crear y destruir clientes, cambiar el comportamiento de una o más ES,...

El simulador incluye bibliotecas de componentes que permiten definir modelos con comportamientos típicos rápida y fácilmente.

Todo esto permite realizarlo con unos costes de simulación muy inferiores a los de SMPL. El porcentaje de mejora de los costes de DESK respecto a los costes de SMPL depende mucho del sistema en concreto: del número de ES, de la longitud media de las colas de espera, de la utilización de recursos, de la utilización de clientes con prioridades y del número de clientes en el sistema. En general, los costes de simulación de DESK son muy inferiores a los de SMPL cuando crece la longitud de las colas del simulador (cuando hay esperas de clientes en las ES o cuando el número de clientes en el sistema crece). A medida que aumenta el número de clientes en el sistema, el coste de simulación de SMPL se dispara, mientras que el de DESK permanece casi constante (tabla 5.7).

Se proporciona una versión web del simulador (JDESK), que a pesar de tener mayores costes de simulación permite tener el simulador disponible desde cualquier navegador y tener permanentemente actualizado el código del simulador. Incluye un asistente para definir el modelo fácilmente.

5.2. Simulador de Eventos Discreto como Núcleo de Aplicaciones Gráficas

En este apartado se realiza un estudio de las mejoras obtenidas por la utilización de un sistema discreto desacoplado frente al sistema continuo acoplado clásico.

En primer lugar se realiza un estudio teórico (apartado 5.2.1), para posteriormente contrastar las conclusiones de este estudio con los resultados obtenidos usando como sistema continuo Fly3D y como sistema discreto DFly3D (apartado 5.2.2).

5.2.1. Comparativa Teórica de los Modelos Discreto y Continuo

Estos resultados teóricos se han obtenido al comparar el videojuego continuo acoplado Fly3D con el videojuego discreto desacoplado DFly3D, creado durante la realización de la tesis. Sin embargo estos resultados teóricos son extrapolables a otras aplicaciones gráficas en tiempo real.

Previamente se define la nomenclatura utilizada en este apartado y se definen dos conceptos generales usados en todos los apartados de esta comparativa: colapso del sistema y de calidad de servicio.

5.2.1.1. Nomenclatura

T_G tiempo total consumido en la ejecución del sistema.

T_R tiempo total consumido en la visualización de las escenas.

T_S tiempo total consumido en la simulación.

T_F tiempo total no consumido por el sistema.

$T_{SISTEMA}$ tiempo total de simulación consumido en la ejecución de un sistema, incluyendo todas las fases: simulación y visualización.

T_{REAL} tiempo real total consumido en la ejecución de un sistema, incluyendo todas las fases: simulación y visualización.

N_R número de visualizaciones totales.

N_S número de simulaciones totales.

O conjunto de todos los objetos de un sistema.

N_O número de objetos en el sistema.

O_i colección de aspectos del objeto i.

FF frecuencia de cuadro (número de fps generados por la aplicación).

SRR frecuencia de refresco de pantalla (número de fps generados por el dispositivo físico).

SSF frecuencia de muestreo del sistema.

T periodo de muestreo del sistema.

OSF_i frecuencia de muestreo del objeto i que modela su comportamiento.

$OSF_{i_{min}}$ mínima OSF_i para simular el objeto i adecuadamente.

$OSF_{i_{max}}$ máxima OSF_i para simular el objeto i si hay potencia de cálculo suficiente.

$OSF_{i,j}$ frecuencia de muestreo del aspecto j del objeto i.

$OSF_{i,j_{min}}$ valor mínimo de $OSF_{i,j}$ para simular el aspecto j del objeto i apropiadamente.

OSF_{i,j_t} valor de $OSF_{i,j}$ en el instante t.

$OSFr_i$ frecuencia de muestreo del objeto i que modela su comportamiento, respecto del tiempo real del sistema.

$OSFs_i$ frecuencia de muestreo del objeto i que modela su comportamiento, respecto del tiempo de simulación del sistema.

SO sobremuestreo de pantalla. Es el SRR que no se llega a visualizar en pantalla debido al sobremuestreo.

CF_S factor de colapso global del sistema.

$CF_{i,j}$ factor de colapso del objeto i aspecto j.

QoS_{i,j_t} QoS del aspecto j del objeto i en el instante de tiempo t.

5.2.1.2. Definiciones

Sistema Colapsado

Se dice que un sistema está *colapsado* cuando no es capaz de simular y visualizar en un ciclo de refresco de pantalla. Es decir, no es capaz de mostrar el número de escenas suficientes o no es capaz de simular todos los objetos adecuadamente. Un sistema colapsado tiene un comportamiento incorrecto, pues no cumple con todas las QoS definidas para cada objeto. Un sistema puede colapsarse debido a:

- Alta complejidad del sistema (simulación o visualización).

- Número de objetos en el sistema.
- Potencia de cálculo insuficiente de la máquina.

La definición de sistema colapsado se trata en mayor profundidad en el apartado 5.2.1.4.

Calidad de Servicio (QoS)

El criterio de QoS en aplicaciones gráficas en tiempo real suele restringirse a ciertos parámetros gráficos, pero realmente incluye otras características, como simulación o física. Los tres aspectos fundamentales de la QoS en aplicaciones gráficas son:

1. Estimulación Sensorial de Usuario (SUS): la interfaz con el usuario tradicional (HCI) incluye:
   - Aspectos gráficos: geometría compleja de objetos, grandes texturas, frecuencia de cuadro, anti-aliasing, motion blur, accumulation buffers, resolución de pantalla, 2D, 3D, iluminación o mapas de luces.
   - Sonido: sonido surround, calidad de sonido, midi o wavetables.

   Pero otros aspectos de la HCI empiezan a incluirse en las aplicaciones gráficas en tiempo real, como visión estéreo, realimentación táctil o reconocimiento visual del usuario.
2. Física: cinemática inversa, detección de colisiones, inercia, dinámica del sistema o fuerzas.
3. Simulación: predicción de comportamientos, respuesta compleja del avatar o inteligencia artificial.

Estos aspectos no tienen la misma relevancia en todas las aplicaciones gráficas. A modo de ejemplo, la tabla 5.11 muestra algunos ejemplos de familias de videojuegos (como ejemplo de aplicación gráfica en tiempo real) y el nivel de relevancia (bajo, medio o alto) de cada uno de los aspectos de la QoS.

La QoS depende del objeto de la aplicación para la que se define:

- Objeto *visualizador*: la QoS depende de aspectos generales del proceso de visualización como FF, anti-aliasing o resolución de pantalla. Estos aspectos están incluidos en la categoría de HCI de la QoS de la aplicación gráfica.
- Objetos de la aplicación gráfica: la QoS depende de tópicos como frecuencia de muestreo, detección de colisiones, inteligencia artificial, texturas, geometría

Familia	**SUS**	**Simulación**	**Física**	**Ejemplo**
Estrategia	Bajo	Alto	Bajo	Chess, The Ancient Art of War, Warcraft
Arcade	Alto	Bajo	Alto	Quake, Doom, Call of Duty, Halo
Simuladores de vuelo	Alto	Alto	Alto	Microsoft Fly, Simulator, Comanche, ATP, Lock On
Otros simuladores	Bajo-Medio	Alto	Bajo	Simcity, The Sims
Aventuras gráficas	Medio	Medio	Bajo	Broken Sword, The Westerner
Rol	Bajo-Medio	Medio	Bajo	Knights of the Old Republic, Neverwinter Night

Tabla 5.11: QoS en familias de videojuegos

de objetos, iluminación y mapas de luces. Estos tópicos pertenecen a las tres categorías anteriores de la QoS.

El programador de la aplicación gráfica puede definir parámetros de la QoS como: geometría de objetos, número de polígonos, tamaño del objeto, profundidad de la textura de color o número de texturas. En muchos casos se le permite al usuario definir algunas características de visualización como anti-aliasing o resolución de pantalla. Esta es una forma de ajustar la QoS del objeto visualizador a la potencia de cálculo de la máquina donde se ejecuta la aplicación.

Diferentes aplicaciones gráficas en tiempo real tienen diferentes necesidades de SUS, física y simulación. La situación ideal es tener un paradigma de simulación que permita que:

- Diferentes objetos pueden tener comportamientos continuos o discretos.
- El mismo objeto pueda tener un comportamiento continuo o discreto, instantáneamente o durante la ejecución del sistema.
- El sistema debe distribuir la potencia de cálculo de la máquina de acuerdo con las necesidades de cada objeto. El sistema debe permitir adaptar el comportamiento de los objetos dinámicamente a:
  - La familia y tipo del aplicación gráfica en tiempo real.
  - La carga del sistema.

- La evolución del sistema durante su ejecución.

5.2.1.3. Distribución de Tiempos

En este apartado se muestra la diferencia de la distribución de tiempos entre los procesos de la aplicación gráfica en el sistema continuo y discreto. En ambos sistemas se cumple la ecuación 5.1:

$$T_G = T_R + T_S + T_F \tag{5.1}$$

La carga del sistema depende del número de objetos, pues tanto la carga de simulación como la de visualización se incrementa con este número (ecuaciones 5.2 y 5.3).

$$N_O \uparrow \Rightarrow T_R \uparrow \tag{5.2}$$

$$N_O \uparrow \Rightarrow T_S \uparrow \tag{5.3}$$

En un **esquema de simulación continuo acoplado** el sistema está continuamente simulando y visualizando (ecuaciones 5.4 y 5.5). El sistema usa casi el 100 % del tiempo de la aplicación en estos dos procesos. En cada pasada del bucle principal se simula y se visualiza. No hay tiempo libre ni recursos libres.

$$T_G \simeq T_R + T_S \tag{5.4}$$

$$N_S = N_R = SSF \tag{5.5}$$

Los tiempos de simulación y visualización son mutuamente dependientes. El tiempo de la aplicación lo comparten estos dos procesos. Un incremento en la carga de simulación supone un decremento del tiempo destinado a la visualización (ecuación 5.6), lo que supone un decremento en frecuencia de cuadro. Si la frecuencia de cuadro es lo suficientemente baja, el sistema no se visualiza adecuadamente.

$$T_S \uparrow \Rightarrow T_R \downarrow \Rightarrow N_R \downarrow \Rightarrow FF \downarrow \tag{5.6}$$

Si el número de visualizaciones disminuye, también disminuye el número de simulaciones (ecuación 5.7). La frecuencia de muestreo del sistema disminuye.

$$N_R \downarrow \Rightarrow N_S \downarrow \Rightarrow SSF \downarrow \tag{5.7}$$

Igualmente, un incremento en la carga de visualización supone un decremento del tiempo de simulación (ecuación 5.8). Si el muestreo de cada objeto no es suficiente para cumplir el teorema de Niquist-Shannon o la QoS definida para cada objeto, algún objeto puede estar muestreándose insuficientemente. Esto puede producir comportamientos incorrectos como pérdida de eventos o colisiones no detectadas. Si el número de simulaciones baja, también baja el número de visualizaciones y por tanto la frecuencia de cuadro (ecuación 5.9).

$$T_R \uparrow \Rightarrow T_S \downarrow \Rightarrow N_S \downarrow \tag{5.8}$$

$$N_S \downarrow \Rightarrow N_R \downarrow \Rightarrow SSF \downarrow \Rightarrow FF \downarrow \tag{5.9}$$

El **esquema de simulación discreto desacoplado** permite la independencia de los procesos de simulación y visualización, además de introducir un sistema discreto de simulación. Si la potencia de cálculo es suficiente, el tiempo de simulación depende únicamente del número de objetos del sistema y aumenta linealmente con este número. Igual ocurre con el tiempo de visualización. No hay una dependencia entre el tiempo de simulación y el tiempo de visualización. El sistema sólo utiliza la potencia de cálculo estrictamente necesaria para cumplir con la QoS de cada objeto.

Los procesos de simulación y visualización son independientes, por lo que la aplicación no usa el 100 % del tiempo simulando y visualizando (ecuación 5.10). El tiempo del sistema depende únicamente de la carga de simulación y de la complejidad de la escena. El sistema discreto libera tiempo que puede ser usado para mejorar aspectos del sistema o liberado para otras aplicaciones. El sistema sólo usa el 100 % del tiempo de la aplicación si el sistema está colapsado.

$$T_G > T_R + T_S \tag{5.10}$$

La tasa de simulación es independiente de la tasa de visualización (ecuación 5.11).

$$N_S \neq N_R \tag{5.11}$$

Las tasas de visualización y simulación son independientes y constantes durante la ejecución mientras haya potencia de cálculo suficiente. El tiempo destinado a la visualización y el destinado a la simulación aumenta linealmente con el número de objetos. Incrementar el tiempo de simulación supone decrementar el tiempo libre (ecuación 5.12) e incrementar el tiempo de visualización supone decrementar el tiempo libre (ecuación 5.13).

$$T_S \uparrow \Rightarrow T_F \downarrow \tag{5.12}$$

Característica	**Sistema Continuo**	**Sistema Discreto**
Tiempo del sistema	$T_G \simeq T_R + T_S$	$T_G = T_R + T_S + T_F$
$T_S \uparrow$	$T_R \downarrow, FF \downarrow$	$T_F \downarrow, FF \simeq$
$T_R \uparrow$	$T_S \downarrow, FF \downarrow$	$T_F \downarrow, FF \simeq$
$N_O \uparrow$	Reparto del aumento entre T_S y T_R	Aumento lineal de T_S y T_R y decremento proporcional de T_F
N_R, N_S	$N_R = N_S$	$N_R \neq N_S$

Tabla 5.12: Distribución de tiempos

$$T_R \uparrow \Rightarrow T_F \downarrow \tag{5.13}$$

La tabla 5.12 muestra un resumen de la distribución de tiempos en ambos sistemas.

Tiempo Libre

El tiempo de la aplicación en un **sistema continuo** lo comparten los procesos de simulación y visualización. El sistema usa casi el 100 % del tiempo de la aplicación en estos procesos. No se libera tiempo ni recursos (ecuación 5.14).

$$T_F = 0 \tag{5.14}$$

En un **sistema discreto** los procesos de simulación y visualización son independientes. Las tasas de simulación y visualización son independientes y constantes durante la ejecución. Por tanto sólo se consume el tiempo y los recursos estrictamente necesarios para simular y visualizar con la QoS definida. Evita visualizaciones y simulaciones innecesarias, liberando potencia de cálculo, que puede destinarse a otras aplicaciones o a mejorar otras partes del sistema, como inteligencia artificial, precisión de la detección de colisiones o incrementar el realismo.

Si el sistema está colapsado el tiempo libre es casi 0 (ecuaciones 5.15 y 5.16). En este caso, el sistema no es capaz de simular o visualizar el número apropiado de veces para garantizar la calidad de la aplicación gráfica. En este caso se comporta igual que el sistema continuo en cuanto a la distribución de tiempos.

$$T_F \simeq 0 \tag{5.15}$$

$$T_G = T_R + T_S \tag{5.16}$$

Un sistema discreto usa únicamente la potencia de cálculo necesaria para simular el sistema manteniendo la QoS definida. Los recursos consumidos por un sistema

Característica	**Sistema Continuo**	**Sistema Discreto**
T_F sistema no colapsado	$T_F = 0$	$T_F = T_G - T_V - T_R$
T_F Sistema colapsado	$T_F = 0$	$T_F = 0$

Tabla 5.13: Tiempo libre

discreto son siempre menores que los consumidos por un sistema continuo siempre que el sistema no esté colapsado.

La tabla 5.13 muestra un resumen de las conclusiones obtenidas sobre el tiempo liberado en ambos sistemas en condiciones de colapso y no colapso del sistema.

Distribución de la Potencia de Cálculo

El **sistema continuo** constantemente simula y visualiza a la máxima velocidad. Por tanto, la potencia de cálculo del sistema se reparte por igual entre todos los objetos del sistema. No hay un reparto de potencia de cálculo en función de las necesidades del objeto. La potencia de cálculo no se distribuye adecuadamente entre los objetos. Todos los objetos se muestrean a la frecuencia más alta que el sistema puede alcanzar.

Si la potencia de cálculo es elevada en relación con la carga del sistema, se desperdicia potencia de cálculo calculando visualizaciones innecesarias que nunca llegarán a mostrarse por pantalla. El número de visualizaciones innecesarias se muestra en la ecuación 5.17.

$$SO = FF - SRR \tag{5.17}$$

En el **sistema discreto** se define la frecuencia con la que cada objeto debe ser muestreado y se mantiene mientras el sistema no esté colapsado. Por tanto, cada objeto consume el tiempo necesario para mantener su QoS. El reparto del tiempo del sistema se realiza en función de las necesidades de cada objeto. Se puede definir un rango de frecuencias (ecuación 5.18), de forma que la frecuencia de muestreo del objeto esté incluida en el rango de frecuencias y pueda variar en el rango dependiendo de la carga del sistema o las necesidades del objeto.

$$\forall i \in O : OSF_i \in [OSF_{i_{min}}..OSF_{i_{max}}] \tag{5.18}$$

En concreto, el sistema discreto define la frecuencia de cuadro que desea obtener ($OSF_{visualizador}$), por lo que el valor de sobremuestreo (si lo hay) del objeto *visualizador* lo controla el programador.

Característica	**Sistema Continuo**	**Sistema Discreto**
Reparto de potencia de calculo entre los objetos	No uniforme No adecuada	Depende de las necesidades del objeto Adecuada

Tabla 5.14: Potencia de Cálculo

El sistema utiliza sólo el tiempo necesario para simular y visualizar con la QoS definida. La potencia remanente puede dedicarse a incrementar la QoS de otros objetos. Si el sistema no está colapsado, el tiempo ahorrado puede usarse para adaptar la OSF_i, tratando de mantener la $OSF_{i_{max}}$ o, al menos, alcanzar $OSF_{i_{min}}$ (ecuación 5.19).

Si la potencia de cálculo es suficiente para simular todos los objetos adecuadamente, no hay ni submuestreo ni sobremuestreo (si el programador ha definido el comportamiento de los objetos adecuadamente). La potencia liberada por el muestreo adecuado de un objeto pueden usarla el resto de los objetos para alcanzar su QoS. Un objeto con un muestreo complementario al mostrado en la figura 5.33 puede usar la potencia liberada para incrementar su QoS.

$$\forall i \in O: \quad OSF_{i_{max}} > OSF_i > OSF_{i_{min}} \tag{5.19}$$

La tabla 5.14 muestra un resumen de las conclusiones obtenidas sobre la potencia de cálculo en ambos sistemas.

5.2.1.4. Colapso del Sistema

Un sistema está colapsado cuando la potencia de cálculo es insuficiente para simular y estimular sensorialmente cumpliendo las condiciones mínimas de QoS. Sin embargo, es diferente la definición de colapso del sistema discreto y del sistema continuo, porque en el sistema discreto no existe SSF.

Un **sistema continuo** está colapsado si se cumple la ecuación 5.20 (no está colapsado si se cumple la ecuación 5.21). Cada objeto o cada aspecto del objeto tiene una frecuencia ideal de muestreo $OSF_{i,j_{min}}$. Esta frecuencia no puede definirla el programador de la aplicación. Dependiendo de la frecuencia de muestreo del sistema se cumple o no el muestreo ideal del objeto.

$$\exists i \in O, \exists j \in O_i: \quad OSF_{i,j_{min}} > SSF \tag{5.20}$$

$$\forall i \in O, \forall j \in O_i: \quad OSF_{i,j_{min}} \leq SSF \tag{5.21}$$

En un **sistema discreto** colapsado el tiempo de simulación difiere del tiempo real del sistema, en un intento de simular el sistema adecuadamente (este aspecto se explica con detalle posteriormente, en este mismo apartado). Por tanto, se definen dos posibles OSF de los objetos:

1. OSF_i **de simulación** ($OSFs_i$): frecuencia de muestreo del objeto i respecto al tiempo de simulación del sistema. Esta frecuencia la define el programador. El programador puede definir un rango de frecuencias aceptable para el objeto, es decir una frecuencia mínima que garantice la correcta simulación del objeto y una frecuencia máxima que sería aconsejable alcanzar si la carga del sistema lo permite. La frecuencia del objeto puede adaptarse dinámicamente entre estas frecuencias (apartado 5.2.2.6).

2. OSF_i **real** ($OSFr_i$): frecuencia de muestreo del objeto i respecto al tiempo real del sistema. Es la frecuencia efectiva del objeto.

Mientras en sistema discreto no está colapsado, ambas frecuencias coinciden (ecuación 5.22). Por lo que, en sistemas no colapsados no es necesario diferenciar la una de la otra. Por ello, en los resultados y mientras no se hable de colapso del sistema, sólo se habla de OSF_i.

El sistema discreto colapsado desfasa el tiempo real del tiempo de simulación (si bien hay mecanismos que evitan o alivian este desfase), en un intento de primar la correcta simulación del sistema sobre la ejecución en tiempo real. Cuando el sistema se colapsa se mantiene la $OSFs$ de los objetos (ecuación 5.22). En el sistema colapsado, la simulación se ralentiza y por tanto, ambas OSF difieren. La frecuencia real desciende respecto a la frecuencia de simulación (ecuación 5.23).

$$\forall i \in O : OSFr_i = OSFs_i = OSF_i \tag{5.22}$$

$$\exists i \in O : OSFr_i < OSFs_i \tag{5.23}$$

El sistema discreto permite definir cual es la frecuencia de muestreo de simulación $OSFs_{i,j}$ de cada objeto. Esta definición es extensible a cada aspecto del objeto (la ecuación 5.24 muestra un sistema colapsado y la ecuación 5.25 un sistema no colapsado).

$$\exists i \in O, \exists j \in O_i : \quad OSFr_{i,j} < OSFs_{i,j} \tag{5.24}$$

$$\forall i \in O, \forall j \in O_i : \quad OSFr_{i,j} = OSFs_{i,j} \tag{5.25}$$

Factor de Colapso

El colapso del sistema puede medirse por el **Factor de Colapso** CF_S. Cada aspecto j del objeto i tiene su propio factor de colapso $CF_{i,j}$:

- **Sistema continuo**: si la frecuencia de muestreo mínima para garantizar la correcta simulación de cada aspecto de cada objeto es mayor que la frecuencia de muestreo del sistema, el sistema está colapsado (ecuación 5.26) y el factor de colapso es la diferencia de muestreos. Si el sistema no está colapsado, este factor es 0 (ecuación 5.27).

$$\begin{gathered}\exists i \in O, \exists j \in O_i : \\ OSF_{i,j_{min}} > SSF \Longrightarrow CF_{i,j} = OSF_{i,j_{min}} - SSF\end{gathered} \tag{5.26}$$

$$\begin{gathered}\forall i \in O, \forall j \in O_i : \\ OSF_{i,j_{min}} \leq SSF \Longrightarrow CF_i = 0\end{gathered} \tag{5.27}$$

- **Sistema discreto**: si la frecuencia de muestreo $OSFs_{i,j}$ de cada aspecto de cada objeto es mayor que la frecuencia de muestreo real del objeto $OSFr_{i,j}$, el sistema está colapsado (ecuación 5.28) y el factor de colapso es la diferencia de muestreos. Si el sistema no está colapsado, este factor es 0 (ecuación 5.29).

$$\begin{gathered}\exists i \in O, \exists j \in O_i : \\ OSFs_{i,j} > OSFr_{i,j} \Longrightarrow CF_{i,j} = OSFs_{i,j} - OSFr_{i,j}\end{gathered} \tag{5.28}$$

$$\begin{gathered}\forall i \in O, \forall j \in O_i : \\ OSFs_{i,j} = OSFr_{i,j} \Longrightarrow CF_{i,j} = 0\end{gathered} \tag{5.29}$$

El factor de colapso global del sistema viene determinado por los factores de colapso de cada aspecto de cada objeto (ecuación 5.30).

$$CF_S = \sum_{i}^{i \in OS} \sum_{j}^{j \in O_i} \mid CF_{i,j} \mid \tag{5.30}$$

Cualquier sistema está colapsado cuando hay al menos un aspecto j de un objeto i que no mantiene su $OSF_{i,j}$. Si se degrada la $OSF_{i,j}$ de un aspecto de un objeto, todas las frecuencias de muestreo de los aspectos de los objetos se degradan también.

El comportamiento incorrecto del sistema aumenta a medida que aumenta el factor de colapso.

Si el sistema está colapsado, ni el sistema continuo ni el discreto son capaces de ejecutar la aplicación convenientemente, aunque por razones diferentes:

- **Sistema continuo**: si el sistema está colapsado no se mantiene la QoS del sistema. Ante un sistema continuo colapsado la única posibilidad es volver a la fase de producción del sistema y variar su programación, variando el coste de simulación y/o visualización.

- **Sistema discreto**: no es capaz de simular siguiendo el tiempo real, pero la simulación es correcta. Ante un sistema discreto colapsado, se pueden utilizar políticas de adaptación dinámica del sistema en tiempo de ejecución (apartado 5.2.2.6), sin necesidad de volver a la fase de producción (utilizando para ello el monitor del sistema).

El momento exacto del colapso depende del sistema utilizado.

Comportamiento del Sistema Colapsado

La forma en la que se comporta el sistema colapsado depende mucho del factor de colapso y de los objetos que se ven implicados en el colapso. En un sistema colapsado, puede haber objetos afectados por el colapso y otros no afectados (ecuaciones 5.20 y 5.24). El efecto del colapso puede ser más o menos importante en el resultado de la ejecución de la aplicación.

En condiciones de colapso, el **sistema continuo** submuestrea los objetos, produciendo que sus movimientos puedan ser caóticos y que la detección de colisiones falle en muchos casos. Cuanto mayor es la carga del sistema, mayor es el colapso y por tanto, más aspectos de los objetos no cumplen el teorema de Nyquist-Shannon o la QoS definida.

Ante un sistema colapsado continuo se puede variar la programación del sistema disminuyendo la carga de simulación y/o visualización. Pero siempre modificando la programación del sistema.

En condiciones de colapso, el **sistema discreto** produce una simulación correcta pero ralentiza el proceso de simulación, pues no es capaz de ejecutar todos los eventos en tiempo real. Prevalece la correcta ejecución de la simulación frente al tiempo real. Produce una salida correcta pues ejecuta la simulación completa, aunque la evolución del sistema es más lenta. Cuanto mayor es la carga del sistema, más colapsado está y más lenta es la simulación.

El sistema discreto se ralentiza, pues no hay potencia de cálculo suficiente para simular el sistema en tiempo real. El número de simulaciones respecto al tiempo de simulación permanece constante a pesar de que la carga del sistema aumenta. No hay simulaciones incorrectas. La QoS de cada objeto se mantiene.

Tiempo Real y Tiempo de Simulación

El **sistema continuo** sigue el tiempo real durante la simulación, aunque para

ello deba disminuir la QoS del sistema. El tiempo del sistema es exactamente el tiempo real (ecuación 5.31). Si el sistema está colapsado, el intento de seguir el tiempo real produce simulaciones incorrectas. Prevalece la ejecución en tiempo real sobre la corrección de la simulación.

$$T_{SISTEMA} = T_{REAL} \tag{5.31}$$

El **sistema discreto** tiene su propio reloj de simulación que avanza siguiendo los eventos de simulación. En el sistema discreto prevalece la correcta simulación del sistema sobre la ejecución en tiempo real. Todos los eventos se ejecutan, consumiendo el tiempo definido por el programador. Se pueden producir dos situaciones diferentes:

- **Sistema no colapsado**: los eventos se ejecutan en el instante exacto en que deben ocurrir (siguiendo el tiempo real, ecuación 5.31). Cuando el tiempo de simulación sobrepasa el tiempo real, el simulador se detiene hasta que se alcanza el tiempo real (se sincroniza con el tiempo real). Esta situación se produce cuando el siguiente mensaje esperando a ejecutarse en el gestor de mensajes debe ocurrir en un tiempo posterior al tiempo real. El tiempo de espera hasta la ocurrencia del siguiente mensaje puede liberarse para otras aplicaciones o se puede reprogramar el sistema para adaptarse dinámicamente utilizando estos tiempos muertos. El tiempo del sistema sigue el tiempo real.
- **Sistema colapsado**: la potencia de cálculo no es suficiente para simular y visualizar con la QoS definida para cada objeto. Por ello, los eventos se ejecutan igual que si hubiese potencia suficiente, pero haciendo que el tiempo de simulación y el tiempo real se desincronicen. El tiempo de simulación siempre está por detrás del tiempo real porque la potencia de cálculo no es suficiente para simular y visualizar en tiempo real. Si el sistema está colapsado, prima la QoS de cada objeto, del objeto *visualizador* y del sistema en general sobre el seguimiento del tiempo real del sistema. En este caso, el sistema se ralentiza para poder ejecutar todos los eventos convenientemente. El tiempo real supera al tiempo de simulación (ecuación 5.32).

  $$T_{SISTEMA} < T_{REAL} \tag{5.32}$$

  Cuanto más colapsado está el sistema mayor es la diferencia entre el tiempo real y el tiempo del sistema (ecuación 5.33).

  $$CF_S \uparrow \;\Rightarrow\; T_{REAL} \uparrow,\; T_{SISTEMA} \simeq \;\Rightarrow\; (T_{REAL} - T_{SISTEMA}) \uparrow \tag{5.33}$$

  El tiempo del sistema de un sistema discreto siempre es el mismo independientemente de la carga del sistema, independientemente de si el sistema está o no

colapsado. Lo determina el conjunto de las QoS de todo el sistema, independientemente del tiempo real que dure la ejecución.

El sistema discreto colapsado no es capaz de ejecutar la simulación en tiempo real, por lo que prevalece la correcta ejecución del sistema sobre el tiempo real. En cambio en el sistema continuo, prevalece la ejecución en tiempo real sobre la corrección de la simulación.

Efecto del Colapso en la Degradación del Sistema

El colapso del sistema produce la degradación del sistema. La degradación del sistema es diferente en un sistema discreto o continuo.

En el **sistema continuo** cada aspecto del objeto tienen un diferente porcentaje de degradación. La frecuencia de muestreo del sistema decrece, por lo tanto, la frecuencia de muestreo de cada objeto decrece también. La nueva frecuencia de muestreo del sistema, o bien, degrada cada objeto en un porcentaje diferente, o bien, puede ser suficiente para mantener la QoS de cada objeto. Si el sistema está colapsado, el sistema continuo puede tener comportamientos incorrectos, como la perdida de eventos o colisiones no detectadas.

En el **sistema discreto** la degradación del sistema es uniforme. Todos las frecuencias de muestreo de los aspectos de los objetos se degradan (ralentizan) en un porcentaje determinado (ecuación 5.34). El sistema discreto produce una **Degradación Elegante del Sistema** (SSD). Si el sistema está colapsado y la frecuencia de muestreo de cada objeto se selecciona correctamente, el sistema se ralentiza, pero la simulación sigue siendo correcta.

$$\begin{gathered} \forall t_1, t_2 \in T_S, \forall i_1, i_2 \in O, \forall j_1 \in O_{i_1}, \forall j_2 \in O_{i_2} : \\ OSF_{i_1,j_1{}_{t_1}} = OSF_{i_1,j_1{}_{t_2}} \wedge OSF_{i_2,j_2{}_{t_1}} = OSF_{i_2,j_2{}_{t_2}} \Longrightarrow \\ CF_{i_1,j_1} = CF_{i_2,j_2} \end{gathered} \tag{5.34}$$

La tabla 5.15 muestra un resumen de las conclusiones obtenidas sobre el colapso del sistema.

5.2.1.5. Calidad de Servicio (QoS)

Muestreo de Objetos

En la mayor parte de los sistemas, cada objeto tiene unas necesidades de muestreo independientes del resto de los objetos. Incluso diferentes aspectos de un mismo objeto pueden tener necesidades de muestreo diferentes (parte gráfica, simulación o física).

El esquema de simulación ideal debe permitir que:

Característica	Sistema Continuo	Sistema Discreto
T_{REAL} sistema no colapsado	$T_{SISTEMA} = T_{REAL}$	$T_{SISTEMA} = T_{REAL}$
T_{REAL} sistema colapsado	$T_{SISTEMA} = T_{REAL}$	$T_{SISTEMA} < T_{REAL}$
$CF_S \uparrow$	$T_{SISTEMA} \simeq$ $T_{REAL} \simeq$	$T_{SISTEMA} \simeq$ $T_{REAL} \uparrow$
Degradación del sistema ante el colapso	No Uniforme Mal funcionamiento	Uniforme Ralentización

Tabla 5.15: Colapso del sistema

- Diferentes aspectos del objeto pueden tener diferentes $OSF_{i,j_{min}}$. La geometría de un objeto puede ser necesario muestrearla con una frecuencia diferente que la detección de colisiones (a causa de la alta velocidad del objeto). Por ejemplo, una bola se distorsiona cada unidad de tiempo, pero su velocidad obliga a cambiar su posición y detectar colisiones cada 0.05 unidades de tiempo. Sus cambios físicos deben muestrearse con una frecuencia diferente a su cambio de posición y detección de colisiones.

- Durante la ejecución del sistema, la frecuencia de muestreo de un objeto puede cambiar. Por ejemplo, cuando una bola cae afectada por la gravedad necesita aumentar $OSF_{bola_{min}}$ a medida que aumenta su velocidad. Si la bola se detiene $OSF_{bola_{min}}$ puede reducirse casi a 0.

- El programador debe poder definir cada $OSF_{i,j_{min}}$.

Los **sistemas continuos** evolucionan el sistema en periodos de T unidades de tiempo, siguiendo la ecuación 5.35.

$$\forall i \in O, \forall j \in O_i : \quad SSF = \frac{1}{T} = OSF_{i,j} \tag{5.35}$$

La frecuencia de muestreo del sistema continuo SSF es común a todo el sistema, incluido el proceso de visualización (ecuación 5.36). Todos los objetos se muestrean a la misma frecuencia y todos los aspectos del objeto se muestrean a la misma frecuencia, también. Esta frecuencia de muestreo depende de la potencia de cálculo de la máquina y de la carga del sistema. Si la carga del sistema es constante, la frecuencia de muestreo es la misma durante toda la ejecución.

$$\forall i \in O : \quad SSF = OSF_i = FF \tag{5.36}$$

Si un objeto necesita muestrearse con una frecuencia $OSF_{i_{min}}$, el objeto i puede submuestrearse, por lo que el comportamiento del objeto sería erróneo (colisiones no detectadas o trayectorias del objeto por pantalla incorrectas, ecuación 5.37).

$$\exists i \in O: \quad OSF_i = SSF, \quad OSF_i > OSF_{i_{min}} \tag{5.37}$$

Es difícil que la frecuencia de muestreo del sistema sea la frecuencia de muestreo ideal para cada uno de los objetos del sistema. Por tanto, si la QoS del objeto requiere una frecuencia de muestreo mayor, el objeto se submuestrea. Si el objeto requiere una frecuencia de muestreo menor, el objeto se sobremuestrea.

El programador no tiene ningún control sobre la frecuencia de muestreo del objeto ni puede adaptarla dinámicamente. La frecuencia de muestreo varía dinámicamente en función del coste temporal de la fase de simulación actual y la fase de visualización actual.

El **sistema discreto** permite definir diferentes frecuencias de muestreo (QoS) para cada objeto del sistema (ecuación 5.38). No existe la frecuencia de muestreo del sistema SSF. Sólo el sistema discreto es capaz de definir una frecuencia de muestreo independiente para cada aspecto. Se adapta el muestreo a la QoS de los objetos. El paradigma discreto desacoplado permite al sistema alcanzar los objetivos de QoS de cada objeto. Cada objeto tiene su propia OSF_i. Incluso puede definirse una frecuencia de muestreo $OSF_{i,j}$ para cada aspecto j del objeto i. Las características del objeto que dependen del valor de OSF_i son, por ejemplo, posición, detección de colisiones o inteligencia artificial.

$$\exists i, j \in O: \quad OSF_i \neq OSF_j \tag{5.38}$$

Un simulador discreto soporta simulación continua e híbrida:

- *Simulación continua*: se generan eventos cada cierto intervalo de tiempo. Cuando un objeto recibe un mensaje, genera otro mensaje. Si el aspecto j del objeto i define su frecuencia mínima de muestreo a $OSF_{i,j_{min}}$, el siguiente mensaje se generará con tiempo $1/OSF_{i,j_{min}}$ unidades de tiempo (para suceder transcurrido dicho tiempo). Ese mensaje sólo afecta al aspecto j del objeto i.
- *Simulación discreta*: se genera un evento (evento programado) que debe suceder en un instante determinado o transcurrido un tiempo determinado.
- *Simulación híbrida*: mezcla de ambas simulaciones.

El número de muestreos definidos para el sistema discreto se mantiene constante, por lo que, suponiendo que la frecuencia de muestreo que ha definido el programador

Característica	**Sistema Continuo**	**Sistema Discreto**
Muestreo de objetos	$\forall i, j \in O : OSF_i = OSF_j$	$\exists i, j \in O : OSF_i \neq OSF_j$
Muestreo del sistema	$\forall i \in O : OSF_i = SSF$	$\exists i \in O : OSF_i \neq OSF_j$
OSF_i	Variable	Constante o adaptable
OSF_i definida por el programador	No	Si
OSF_i adaptable dinámicamente	No	Si
OSF_i dependiente de la carga del sistema	Si	No
OSF_i sistema no colapsado	$\forall i \in O : OSF_i = SSF \neq OSF_{i_{min}}$	$\forall i \in O : OSFs_i = OSFr_i = OSF_i$ $OSF_i \in [OSF_{i_{min}}..OSF_{i_{max}}]$
OSF_i sistema colapsado	$\exists i \in O : OSF_i = SSF \neq OSF_{i_{min}}$	$\exists i \in O : OSFs_i > OSFr_i$ $OSFs_i = OSF_{i_{min}}$

Tabla 5.16: Muestreo de objetos

para el objeto es la óptima, el sistema discreto mantiene la frecuencia óptima de muestreo del objeto en cualquier situación.

La frecuencia de muestreo de los objetos puede cambiar dinámicamente para ajustar su muestreo al comportamiento ideal de los objetos. La OSF_i de un objeto puede definirse de forma que varíe dinámicamente dependiendo de las necesidades del objeto o la carga del sistema. El programador siempre tiene el control sobre la frecuencia de muestreo del objeto, sea constante o adaptable.

La tabla 5.16 muestra un resumen de las conclusiones obtenidas sobre el muestreo de objetos en ambos sistemas en condiciones de colapso y no colapso del sistema.

Frecuencia de Cuadro

La QoS del objeto *visualizador* determina la frecuencia de cuadro del sistema. La situación ideal es que el programador pueda definir la frecuencia de cuadro que genera el sistema. Esta FF debe mantenerse durante toda la ejecución y debe ser independiente de la carga del sistema mientras el sistema no esté colapsado.

Para que el sistema se visualice con una calidad aceptable, se debe cumplir que la frecuencia de muestreo del objeto *visualizador* (frecuencia de cuadro) supere la QoS mínima definida para este objeto.

El **sistema continuo** tiene una SSF común a todo el sistema (todos los objetos y el proceso de visualización).

Por tanto, el muestreo del objeto visualizador sigue las mismas consideraciones que el muestreo de un objeto cualquiera. Si la frecuencia de cuadro es inferior a la QoS mínima del objeto visualizador (ecuación 5.39), las escenas no se visualizan adecuadamente. Si es superior (ecuación 5.40), se generan visualizaciones innecesarias, desperdiciando potencia de cálculo.

$$FF < OSF_{visualizador_{min}} \Rightarrow submuestreo \tag{5.39}$$

$$FF > OSF_{visualizador_{min}} \Rightarrow sobremuestreo \tag{5.40}$$

La FF depende del coste temporal de las fases de simulación y visualización. El sistema constantemente simula y visualiza a la máxima velocidad. La FF del sistema continuo es altamente dependiente de la potencia de cálculo de la máquina y el programador no tiene ningún control sobre este valor (no puede fijarse en función de las necesidades del sistema). Por tanto, el sistema continuo no permite definir la QoS del objeto visualizador.

El **sistema discreto** permite definir cual es la FF óptima que debe generar el sistema. La FF definida se mantiene mientras el sistema no esté colapsado.

En el sistema discreto no existe SSF porque cada objeto i tiene su propia OSF_i (ecuación 5.41). El valor de FF es configurable pues corresponde con la frecuencia de muestreo fijada por el programador para el objeto *visualizador* (ecuación 5.42). Además, esta frecuencia de muestreo puede ser ajustada por el sistema, si así se programa.

$$\exists i, j \in O, \quad i \neq j : \quad OSF_i \neq OSF_j, \quad OSF_i \neq FF \tag{5.41}$$

$$OSF_{visualizador} = FF \tag{5.42}$$

El sistema discreto permite alcanzar la QoS de la visualización. De este modo, si la $QoS_{visualizador}$ se define adecuadamente se evitan visualizaciones innecesarias, liberando la potencia de cálculo, permitiendo, al mismo tiempo, una visualización del sistema adecuada.

Frecuencia de Cuadro Máxima

El **sistema discreto** puede alcanzar un FF mayor que el **sistema continuo** aunque el sistema empiece a colapsarse, pues la carga se reparte de forma óptima entre los objetos. Las pruebas realizadas muestran que la tasa de escenas generadas por unidad de tiempo que el sistema discreto es capaz de generar es siempre un poco

Característica	Sistema Continuo	Sistema Discreto
SSF	Común a todo el sistema	No existe
FF	$FF = SSF$	$FF = OSF_{visualizador}$
OSF_i	$\forall i \in O : OSF_i = FF$	$\exists i \in O : OSF_i \neq FF$

Tabla 5.17: Número de escenas generadas

mayor que la tasa generada por el sistema continuo. Esto se debe a las siguientes razones:

- El OSF_i de un objeto en un sistema discreto suele ser menor que el SSF en un sistema continuo cuando la carga del sistema es alta (ecuación 5.43).

$$\forall i \in O : \quad SSF_{continuo} > OSF_i \tag{5.43}$$

- El valor de OSF_i en un sistema discreto no depende de FF.
- La sobrecarga producida por la gestión de mensajes en un sistema discreto no es significativa.

La tabla 5.17 muestra un resumen de las conclusiones obtenidas sobre el número de escenas generadas en ambos sistemas.

Detección de Colisiones

En el sistema continuo el muestreo de los objetos depende de la carga del sistema y es igual para todos los objetos del sistema. En el sistema discreto se puede fijar la frecuencia muestreo de cada objeto, independientemente del resto de objetos del sistema y de la carga del sistema. En ambos sistemas, el orden de la detección de colisiones varía dependiendo del muestreo de los objetos:

- **Muestreo inferior al óptimo** (teorema de Niquist-Shannon): tanto en el sistema continuo como en el discreto no se garantiza que se detecten las colisiones que deberían ocurrir.
- **Muestreo superior o igual al óptimo**: la detección de colisiones es altamente dependiente del número de objetos, de la velocidad de cada uno de ellos y del muestreo de cada objeto.

  El **sistema continuo** no permite variar la frecuencia de muestreo de los objetos sin variar la carga del sistema. Por tanto, no se puede observar como se comporta variando esta frecuencia. Una mayor carga de simulación supone que se simula un número de veces menor. Por tanto, el sistema discreto puede no detectar colisiones dependiendo de la carga del sistema.

En el **sistema discreto**, la simulación es la misma, independientemente de la carga. El sistema se ralentiza para simular correctamente, pero la simulación permanece inalterable. Por tanto el número de colisiones detectadas es contante e independiente de la carga.

5.2.1.6. Eventos Programados

Un evento programado es aquel que debe suceder en un instante de tiempo determinado (o transcurrido un intervalo de tiempo determinado).

La **simulación continua** no permite eventos programados. Si se desea un evento programado debe simularse mediante el muestreo de una determinada condición crítica que arranca el proceso del evento programado, con el coste temporal que supone.

La **simulación discreta** permite programar eventos que sucederán en un instante de tiempo determinado sin necesidad de muestreo y, por tanto, con un coste de simulación mínimo.

Por ejemplo, supóngase una bomba de relojería que debe estallar dentro de T segundos.

- *Sistema continuo*: en cada simulación del objeto se debe comprobar si la bomba debe estallar. La sobrecarga producida por el muestreo depende mucho del coste de la comprobación de la condición de disparo.
- *Sistema discreto*: se programa un evento que sucederá dentro de T segundos. No se debe muestrear si el evento ha sucedido o no. Cuando el objeto bomba reciba el mensaje transcurridas T unidades de tiempo, ejecutará la función asociada a ese evento.

5.2.1.7. Adaptación Dinámica del Sistema

Adaptar el sistema dinámicamente supone que el propio sistema es capaz de redefinir sus criterios de QoS para adaptarse a las condiciones de colapso del sistema. Para ello, el sistema debe detectar el instante en que el factor de colapso del sistema alcanza un determinado valor crítico y ser capaz de tomar las decisiones apropiadas. Por ejemplo, si el sistema se está colapsando, los objetos pueden variar su QoS_i a la $QoS_{i_{min}}$ que le permita tener un comportamiento todavía aceptable pero sin sobrecargar el sistema.

El sistema puede estar programado para prevenir condiciones futuras de colapso y adaptarse dinámicamente. Por ejemplo, entra en ejecución un determinado objeto con una geometría muy compleja y por tanto con un coste de visualización muy

elevado. Durante el tiempo en que este objeto está escena todos los objetos reducen su QoS_i a la $QoS_{i_{min}}$.

El programador define hasta que punto los objetos deben estar dispuestos a degradarse (valor de $QoS_{i_{min}}$) para evitar el colapso del sistema.

Tipos de Adaptación Dinámica del Sistema

La adaptación dinámica del sistema puede implicar adaptar procesos diferentes:

- Adaptar la complejidad de la simulación del objeto (u objetos).
- Adaptar el número de muestreos del objeto (u objetos).
- Adaptar la frecuencia de cuadro.
- Adaptación combinada.

En los siguientes apartados se estudia cada tipo de adaptación.

Detección de la Condición Crítica

Uno de los requisitos para que un sistema pueda adaptarse dinámicamente es que sea capaz de definir y detectar alguna condición crítica que le permita llevar a cabo la adaptación. Una posible condición crítica es el factor de colapso del sistema.

En el **sistema continuo** estudiado en la tesis no se incluye ningún método de monitorización ni se permite definir si el sistema está más o menos colapsado. La única posibilidad es utilizar como factor de colapso la frecuencia de cuadro del sistema. Sin embargo, no quiere decir que ningún sistema continuo incluya métodos de detección del colapso y definición de condiciones críticas.

El **sistema discreto** permite detectar fácilmente el colapso del sistema y su factor. El monitor del sistema define diferentes posibilidades de definición de una condición crítica.

Si un mecanismo para definir el colapso o condiciones críticas no es posible una adaptación del sistema, es un requisito indispensable aunque no suficiente.

Adaptación de la Complejidad de la Simulación del Objeto

Adaptar la complejidad de la simulación del sistema supone que si el sistema está colapsado o es necesaria la adaptación, el sistema es capaz de cambiar su simulación. Un ejemplo de esta adaptación puede ser un objeto que cuando detecta que se colapsa el sistema cambia su algoritmo de detección de colisiones por otro menos preciso pero con un coste temporal menor.

Sólo es posible adaptar la simulación del objeto (en cualquier sistema) si el objeto tiene partes de la simulación que puede adaptar (partes de la simulación que

se pueden calcular o no, o cambiar por otras con un coste de diferente). Por ejemplo, pueden coexistir varios algoritmos de detección de colisiones de diferente precisión (de diferente coste temporal de ejecución). Si el sistema está colapsado se puede prescindir de la precisión en la detección de colisiones. O puede hacer un cálculo más o menos preciso de su posición.

Igual que el sistema mejora dinámicamente, también se degrada dinámicamente. Si en algún momento el tiempo entre muestreos vuelve a ser menor que el tiempo óptimo mínimo entre muestreos el sistema vuelve a simularse correctamente.

Esta adaptación del sistema se puede hacer tanto en un sistema continuo como discreto. Ambos sistemas deben definir una condición crítica para variar la simulación.

- **Sistema continuo**: se muestrea la condición crítica en cada simulación del objeto. Dependiendo de que se cumpla o no la condición se realiza el proceso de adaptación o no. Por tanto, la adaptación se realiza por muestreo.

  Se sobrecarga el sistema con el muestreo. Además se dificulta mucho la programación de la aplicación, sobre todo si la adaptación se realiza en varias posibles situaciones y que añade un coste adicional al sistema.

  Si con la adaptación se disminuye el tiempo de simulación, el tiempo ganado se reparte con el proceso de visualización (aumenta N_S y FF), por lo que no se puede controlar cuanto tiempo se mejora el sistema.

- **Sistema discreto**: el frecuencia de muestreo del objeto permanece constante pero se varía la simulación del objeto. Esta adaptación puede hacerse usando muestreo, como en el sistema continuo. O utilizar la generación de eventos para mejorar el proceso. Se comprueba la condición crítica, e el instante en que se cumple se genera un evento para modificar la simulación del objeto. A partir de este momento ya no se comprueba la condición crítica.

  Se puede mejorar el sistema si la comprobación inicial de la condición crítica se realiza con una frecuencia de muestreo diferente a la frecuencia de muestreo de la simulación de objeto.

  También se puede utilizar el mecanismo de eventos programados si se conoce que el sistema debe adaptarse en un momento determinado (proceso que debe hacerse obligatoriamente con muestreo en un sistema continuo).

  Utilizar eventos permite adaptar la simulación sobrecargando el sistema lo mínimo posible.

Adaptación del Número de Muestreos del Objeto

Adaptar el número de muestreos del objeto supone que el sistema es capaz de decidir en un instante determinado aumentar o disminuir la frecuencia de muestreo

de uno o más objetos. Por ejemplo, un objeto que modela una nave espacial se muestrea con una frecuencia OSF_1. En un instante determinado, atraviesa un campo de asteriodes, lo que implica que las posibilidades de colisión son mucho mayores, por lo que durante este intervalo, la frecuencia de muestreo aumenta, siendo OSF_2, de forma que $OSF_1 < OSF_2$. Transcurrido este intervalo recupera su frecuencia de muestreo original OSF_1.

En un **sistema continuo** la frecuencia de muestreo de cada objeto corresponde con la frecuencia de muestreo del sistema. Es constante durante toda la ejecución, depende de la carga del sistema y sólo varía si la carga del sistema varía. En cualquier caso, no es configurable por el programador y las variaciones tampoco las controla éste.

Para adaptar el muestreo de los objetos se puede utilizar una condición crítica, de forma se ejecute o no la simulación del objeto dependiendo de si se cumple esta condición. Permite reducir el número de simulaciones pero no aumentarla y además con un coste extra de programación y simulación.

Por tanto, se utiliza muestreo de determinada condición crítica, sobrecargando el sistema y no se permite aumentar el número de simulaciones de forma directa. Otra dificultad de el sistema continuo es como detectar que un sistema está colapsado.

No evita el colapso del sistema en la misma medida que el sistema ni consigue los mismos efectos de adaptación, pues la reducción en la carga de simulación se reparte con la carga de visualización y aumenta la frecuencia de muestreo del sistema SSF.

Otro inconveniente es que el sistema se sigue muestreando igual, por tanto el porcentaje de adaptación del sistema es proporcional a SSF. El sistema no es sensible a tiempos inferiores al periodo de muestreo. La adaptación afecta a todos los objetos del sistema, por lo que la mejora de la adaptación se reparte entre todos.

En un sistema continuo es difícil predecir como se comportará el sistema adaptado.

En el **sistema discreto** se define la frecuencia de cuadro y el muestreo de cada objeto independientemente del resto y es el programador quien la define según la QoS de cada objeto. El sistema puede adaptarse dinámicamente variando la frecuencia de muestreo de todos los objetos o parte de ellos de forma sencilla y con un coste bajo de programación y temporal de ejecución.

La frecuencia de cada objeto puede variar dinámicamente para adaptarse al comportamiento del objeto. El programador puede definir como cambia dinámicamente y bajo que condiciones.

En el instante en que se detecta o cumple la condición crítica, se varía la frecuencia de muestreo y no se vuelve a comprobar la condición crítica (a no ser que

sea necesario variarla nuevamente). Así se consigue que disminuya el tiempo de simulación de los objetos y que aumente la frecuencia de cuadro efectiva del sistema, al disminuir algo el colapso del sistema.

Una adaptación real del número de muestreos sin variar la carga del sistema sólo es posible en el sistema discreto. El sistema continuo lo único que hace es tratar de que la ejecución del sistema sea parecida a la ejecución si se adaptase el número de muestreos. Además, en el sistema continuo sólo es aplicable a ciertos casos concretos y por tanto no alcanza toda la funcionalidad de sistema continuo.

Adaptación de la Frecuencia de Cuadro

Este tipo de adaptación es un caso particular de la adaptación del muestreo de los objetos, ya que el objeto visualizador es un objeto como cualquier otro del sistema.

En el **sistema continuo** la FF no es configurable por el programador. Si varía durante la ejecución del sistema se debe a cambios en la carga de visualización y/o simulación no controlados por el programador. Por tanto no es posible adaptar dinámicamente la frecuencia de cuadro. La única posibilidad sería disminuir la carga de simulación o la propia carga de visualización, lo que degradaría el sistema.

El **sistema discreto** permite un proceso de visualización adaptativo. El sistema controla el proceso de visualización usando el objeto *visualizador*. Este objeto puede tomar decisiones para adaptar su comportamiento a la carga del sistema, evitando visualizaciones innecesarias que nunca se mostrarán por pantalla. La FF puede variar para adaptarse a las necesidades del sistema dinámicamente, permitiendo aplicaciones gráficas más complejas, evitando al máximo colapsar el sistema. Esto permite una cierta independencia de la aplicación de la potencia de cálculo de la máquina. La misma aplicación puede ejecutarse en máquinas con potencias de cálculo diferentes manteniendo la misma calidad en todos los sistemas. Este proceso adaptativo debe definirlo el programador como comportamiento del objeto *visualizador*, no está incluido explícitamente en el núcleo de simulación.

Esta adaptación dinámica del sistema sólo es posible en el sistema discreto, pues el sistema continuo no permite definir la frecuencia de cuadro del sistema y, por tanto, no permite redefinirla. El sistema discreto puede detectar una determinada condición crítica y variar su frecuencia de cuadro.

Adaptación Combinada

Por adaptación combinada se entiende la combinación de adaptaciones de varios objetos diferentes, o bien de objetos y la frecuencia de cuadro. Por ejemplo, si la frecuencia de cuadro baja de un determinado valor umbral, se disminuye el número de muestreos del objeto para intentar que la frecuencia de cuadro suba.

Cualquiera de estas adaptaciones dinámicas se pueden combinar para conseguir el efecto deseado.

En el **sistema continuo** sólo pueden llevarse a cabo ciertas adaptaciones y siempre con un coste de programación elevado y una sobrecarga en el coste de ejecución.

En cambio, el **sistema discreto** permite cualquier adaptación de una forma natural, sencilla y con un coste de programación y ejecución bajo. Puede definirse para adaptar la potencia de cálculo al factor de colapso del sistema CF_S o redefinir la QoS de los objetos del sistema para adaptar su comportamiento a la carga real del sistema.

En el sistema discreto unos objetos pueden comunicarse con otros (o con objetos del sistema) para indicarles que se ha cumplido una condición crítica y deben adaptarse. Esto evita tener que comprobar la condición crítica en cada uno de los objetos del sistema, evitando la sobrecarga de la comprobación. Incluso, unos objetos pueden pedirles a otros que disminuyan su frecuencia de muestreo para que ellos puedan simularse correctamente.

Por ejemplo, el objeto *visualizador* genera 50 fps y el objeto *bola* se muestrea 20 veces por segundo. Si la potencia de cálculo es baja y la bola necesita muestrearse durante un intervalo de tiempo 30 veces por segundo, el objeto *visualizador* puede disminuir el número de escenas generadas a 25 fps y liberar potencia de cálculo. La situación opuesta también se puede producir. Si el objeto *visualizador* detecta que hay potencia de cálculo suficiente para aumentar el número de escenas generadas, puede aumentar su frecuencia de muestreo. El objeto *bola* también puede hacerlo.

El sistema discreto llega donde el sistema continuo no llega.

5.2.1.8. Monitorización del Sistema

La monitorización de un sistema no es una herramienta exclusiva de un sistema discreto. Pueden existir sistemas continuos que incluyan monitorización. No es el caso de Fly3D. Sin embargo, la utilización de un sistema discreto facilita las tareas de monitorización. El sistema discreto DFly3D incluye un monitor del sistema con toda esta funcionalidad.

El sistema discreto puede definir un factor de degradación que le permita al sistema adaptarse. El sistema discreto es capaz de degradarse o mejorarse uniformemente utilizando la monitorización del sistema. La monitorización del factor de degradación puede hacerse:

- **Para cada objeto**: el objeto recoge información que indique si el objetivo de calidad de servicio se ha cumplido o no. Si no se ha cumplido redefine su

objetivo como sea necesario o su comportamiento. El objeto puede recoger información del resto del sistema, también. Las decisiones del objeto pueden implicar a otros objetos. Por ejemplo, el programador decide que la simulación de la *bola* es prioritaria al número de escenas generadas. Si la *bola* detecta que no está siendo simulada correctamente, envía un mensaje al objeto *visualizador* para que libere potencia de cálculo, disminuyendo el número de escenas generadas (el *visualizador* puede alcanzar su objetivo de calidad de servicio).

- **Para cada aspecto del objeto**.
- **Para el sistema completo**. Hay dos posibilidades:
  - Crear un objeto de monitorización (MO). El MO recoge información del sistema. Si la calidad de servicio de algún aspecto de algún objeto debe modificarse, el objeto MO envía un mensaje al objeto implicado. El mensaje contiene la información necesaria para permitir al objeto cambiar su objetivo de calidad de servicio.
  - Cualquier objeto monitoriza el sistema. No es necesario crear un MO específico. Su funcionalidad puede ser asumida por cualquier objeto.

La monitorización para el sistema discreto propuesto se trata con detalle en el apartado 4.7 del capítulo 4. En dicho apartado se detalla como la simulación discreta permite monitorizar el sistema.

5.2.2. Comparativa de Fly3D y DFly3D

Fly3D es el núcleo de aplicaciones gráficas original, continuo y acoplado. DFly3D es la versión discreta desacoplada de este mismo núcleo. Ambos sistemas tienen la misma funcionalidad, los procesos son los mismos. La única diferencia es como se gestionan los eventos del sistema, como y cuando se arranca cada uno de ellos (entre estos procesos se incluye el proceso de visualización). Se van a comparar ambos sistemas para comprobar si el nuevo esquema de simulación mejora al anterior esquema.

Cada uno de los apartados de esta comparativa demuestran los resultados obtenidos en la comparativa teórica, por tanto hay una correspondencia de apartados entre los resultados teóricos y prácticos.

5.2.2.1. Condiciones de Prueba

Los resultados se han obtenido creando un videojuego para FLy3D y posteriormente discretizándolo para DFLy3D. El videojuego consiste en una serie de bolas cambiando su posición y colisionando.

Ambas versiones del videojuego tienen la misma funcionalidad. La implementación de ambas es también muy similar. La única diferencia es la implementación de la función de simulación:

- **Fly3D**: la simulación del objeto se define en la función *step*, que se muestrea en cada iteración de bucle principal de la aplicación. La bola cambia su posición considerando el intervalo de tiempo desde la última simulación.
- **DFly3D**: no hay muestreo. La simulación se define mediante el mecanismo de paso de mensajes (la funcionalidad del proceso de simulación corresponde a la función de recepción de mensajes). Cada vez que el objeto bola recibe un mensaje actualiza su estado convenientemente y define el siguiente instante en que debe actualizarse, enviándose un mensaje con dicho tiempo. Los mensaje se envían con una tasa de muestreo fija de forma que las colisiones pueden detectarse correctamente y la trayectoria del objeto es correcta.

El videojuego permite variar el número de objetos (bolas). El incremento en el número de objetos en el sistema produce un incremento tanto en la carga de simulación como en la carga de visualización.

Ambos sistemas han sido probados en un ordenador personal Pentium 4 (2 GHz) con 512Mb de Ram, con una tarjeta gráfica Nvidia GeForce 4 MX440.

Los recursos consumidos por ambos sistemas son similares, pero la carga de CPU es diferente dependiendo del esquema de simulación utilizado.

5.2.2.2. Distribución de Tiempos

En este apartado se muestran los resultados de la comparación de la distribución de tiempos de Fly3D (sistema continuo) y DFly3D (sistema discreto), para comprobar que se cumplen los resultados teóricos obtenidos.

Fly3D sigue un esquema de simulación continuo acoplado (figura 5.11), el sistema está continuamente simulando y visualizando, por lo que usa casi el 100 % del tiempo de la aplicación en estos dos procesos. Un incremento en la carga de simulación supone un decremento del tiempo destinado a la visualización, lo que supone un decremento en la frecuencia de cuadro. Un incremento en la carga de visualización supone un decremento del tiempo de simulación. Si el muestreo de cada objeto no es suficiente para cumplir el teorema de Niquist-Shannon, algún objeto puede estar muestreándose insuficientemente. Si la frecuencia de cuadro no es suficiente, el sistema puede no visualizarse correctamente.

DFly3D permite la independencia de los procesos de simulación y visualización (figura 5.12). Si el sistema no está colapsado, el tiempo de simulación depende únicamente del número de objetos del sistema y aumenta linealmente con este valor.

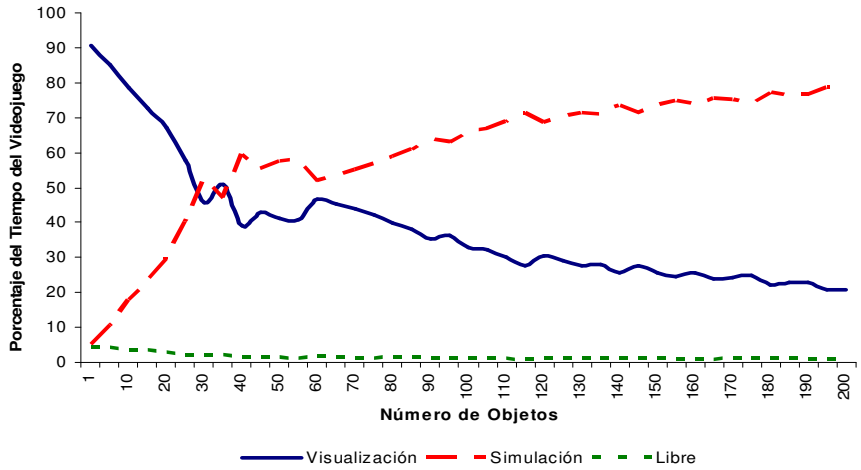

Figura 5.11: Tiempos de FLy3D aumentando el número de objetos en el sistema

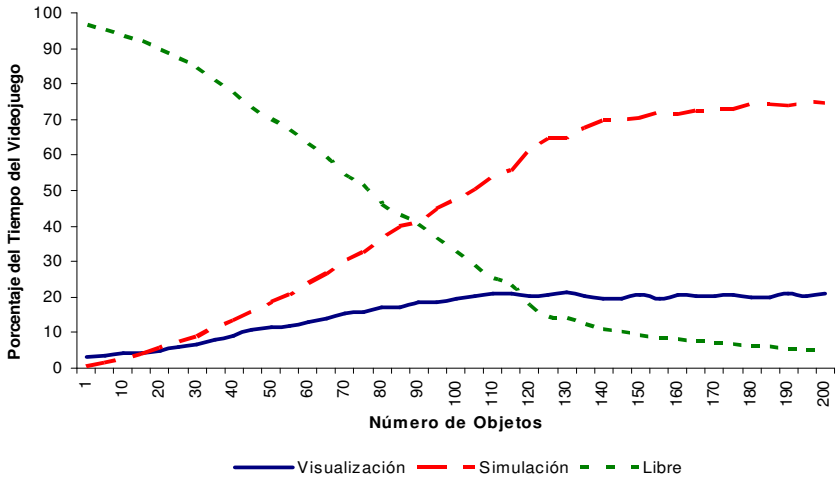

Figura 5.12: Tiempos de DFLy3D aumentando el número de objetos en el sistema

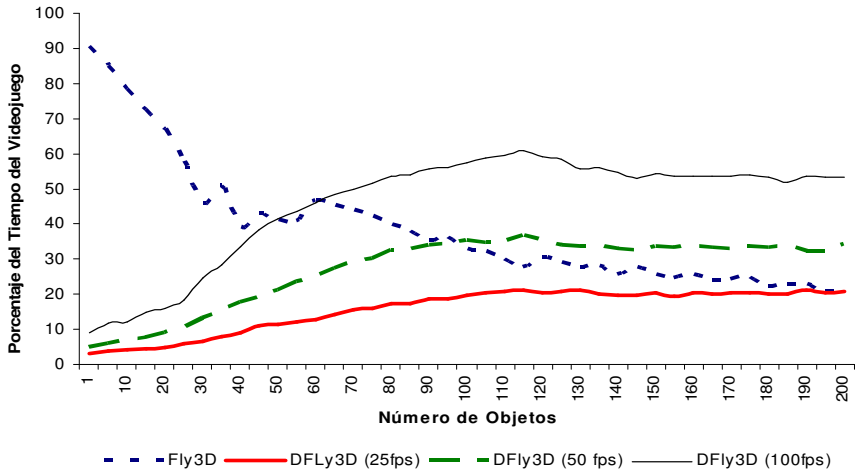

Figura 5.13: Tiempo de visualización de Fly3D y DFly3D

Igual ocurre con el tiempo de visualización. Los procesos de simulación y visualización son independientes, por lo que consumen únicamente el tiempo necesario para mantener la QoS definida por el programador. El resto del tiempo se libera.

DFLy3D utiliza la potencia de cálculo estrictamente necesaria para cumplir con la QoS de cada objeto. Si el sistema está colapsado el tiempo libre es casi 0, el sistema no es capaz de simular o visualizar el número apropiado de veces para garantizar la calidad del videojuego. En este caso se comporta igual que Fly3D en cuanto a la distribución de tiempos.

Las figuras 5.13 y 5.14 muestran los tiempos de visualización y simulación de ambos sistemas cuando aumenta el número de objetos. Para DFLy3D se muestran los valores correspondientes a diferentes ejecuciones variando la frecuencia de cuadro (FF).

El tiempo de visualización de Fly3D, en vez de aumentar con el número de objetos, disminuye, pues el aumento de simulación debe compartir el tiempo de la aplicación con el tiempo de visualización. Los tiempos de simulación y visualización de DFly3D aumentan linealmente con el número de objetos, hasta que empieza a colapsarse el sistema. El tiempo de simulación siempre es más alto en Fly3D que en DFly3D, pues en DFly3D se fija una tasa de muestreo de cada objeto y se mantiene constante.

Tiempo Libre

Fly3D usa casi el 100 % del tiempo de la aplicación en los procesos de simulación y visualización. No libera tiempo. En DFLy3D los procesos de simulación y visuali-

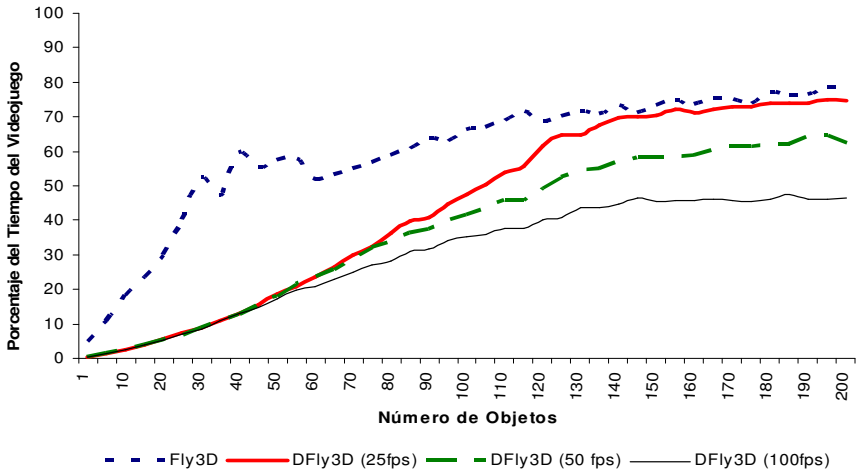

Figura 5.14: Tiempo de simulación de Fly3D y DFly3D

zación son independientes (figura 5.12). El sistema utiliza el 100 % del tiempo que le asigna la CPU en los procesos de visualización y simulación únicamente si el sistema está colapsado.

La figura 5.15 muestra el tiempo liberado por Fly3D y DFly3D. Para DFly3D se muestran los tiempos para diferentes FF. El tiempo libre con DFLy3D para un objeto es casi el 97 % del tiempo asignado a la aplicación. Esto es debido a las siguientes razones:

- Los procesos de visualización y simulación son independientes en DFly3D.
- La frecuencia de cuadro FF la define el programador en DFly3D. Puede ser constante durante la ejecución o variar para ajustarse al comportamiento deseado del sistema (comportamiento definido por el programador).
- La tasa de simulación de cada objeto la determina el programador para cada objeto. Puede ser constante durante la ejecución del videojuego o puede ser variable para adaptarse a las necesidades de muestreo del objeto.

La figura 5.16 muestra la diferencia entre el tiempo liberado por DFly3D y Fly3D. La diferencia siempre es positiva, hasta que se colapsa el sistema, instante en que la diferencia es cercana a cero.

El tiempo liberado por DFLy3D puede usarse para mejorar aspectos del videojuego (o para adaptar el sistema dinámicamente) o liberarlo para otras aplicaciones.

Potencia de Cálculo de la Máquina

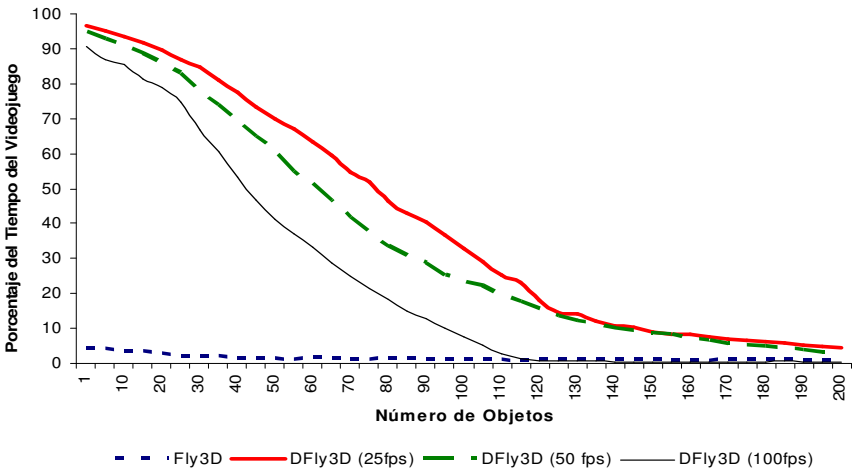

Figura 5.15: Tiempo liberado por Fly3D y DFly3D

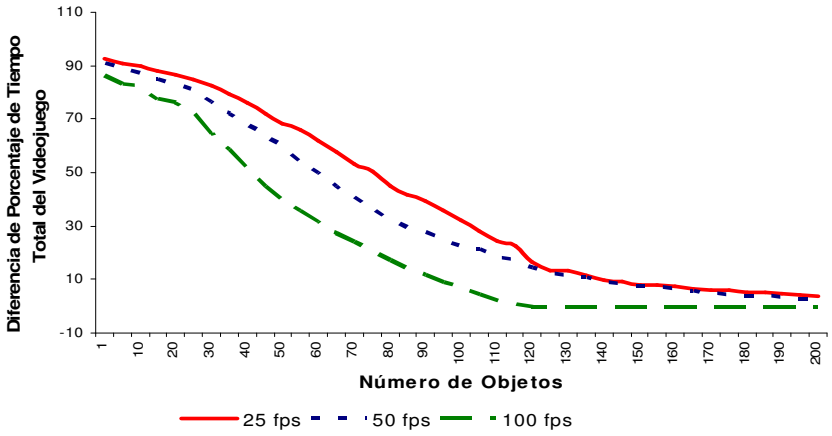

Figura 5.16: Diferencia de tiempo liberado por Fly3D y DFly3D

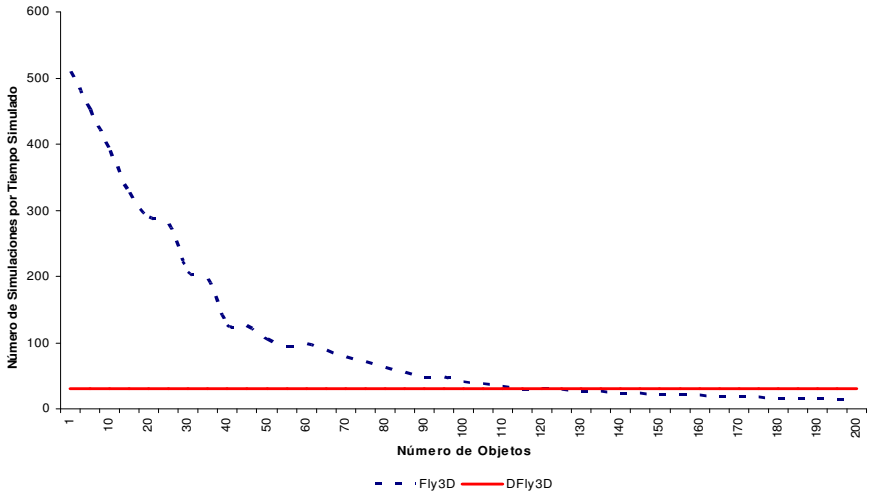

Figura 5.17: Número de simulaciones del objeto por unidad de tiempo simulado en Fly3D y DFly3D

Fly3D desperdicia potencia de cálculo calculando visualizaciones innecesarias que nunca llegarán a mostrarse por pantalla (figura 5.11) o simulando un número innecesario de veces. La potencia de cálculo no se distribuye adecuadamente entre los objetos. Todos los objetos se muestrean a la frecuencia más alta que el sistema puede alcanzar.

En cambio, en DFLy3D, se simula y visualiza el número de veces indicado por el programador (figura 5.12). Por tanto el reparto de la carga del sistema se realiza en función de las necesidades de cada objeto en concreto.

5.2.2.3. Colapso del Sistema

Las pruebas desarrolladas usan la correcta trayectoria del movimiento de los objetos como criterio de QoS del objeto (número de muestreos de los objetos).

Comportamiento del Sistema Colapsado

Si el sistema está colapsado, Fly3D submuestrea los objetos, produciendo comportamientos incorrectos. A mayor carga del sistema, mayor es el colapso y por tanto, más aspectos de los objetos no cumplen el teorema de Nyquist-Shannon.

DFly3D produce una salida correcta, aunque la evolución del sistema es más lenta, pues no es capaz de ejecutar todos los eventos en tiempo real. Cuanto mayor es la carga del sistema, más lenta es la simulación.

Las figuras 5.17 y 5.18 muestran el porcentaje de tiempo de simulación total

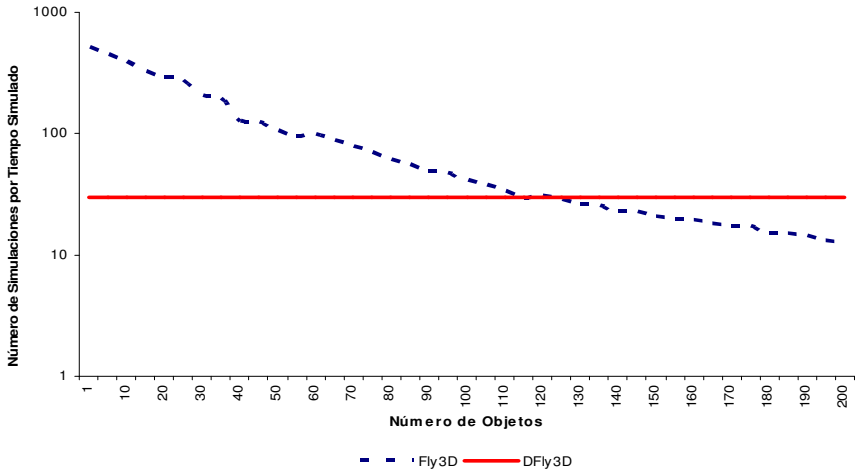

Figura 5.18: Número de simulaciones del objeto por unidad de tiempo simulado en Fly3D y DFly3D (escala logarítmica)

usado por Fly3D y DFly3D a medida que el número de objetos en el sistema aumenta. En DFLy3D el número de simulaciones permanece constante a medida que aumenta la carga del sistema, sin embargo en Fly3D decrece. El punto de colapso es el punto en que se cruzan ambas líneas, alrededor de 105 objetos. Es el momento en que Fly3D no es capaz de mantener el número de muestreos óptimo de los objetos.

Las figuras 5.19 y 5.20 muestran el número de simulaciones de ambos sistemas por unidad de tiempo real y las figuras 5.17 y 5.18 por unidad de tiempo simulado. El número de simulaciones en Fly3D es el mismo independientemente de que se considere tiempo real o tiempo simulado, pues ambos tiempos son iguales (la simulación sigue el tiempo real aunque el sistema se colapse, disminuyendo la QoS). El número de simulaciones en DFLy3D difiere según el tiempo sea real o de simulación (figura 5.21). El número de simulaciones respecto al tiempo de simulación permanece constante, en cambio, disminuyen el número de simulaciones respecto al tiempo real, porque el sistema se colapsa.

DFly3D se ralentiza ($T_{SISTEMA} < T_{REAL}$), pues no hay potencia de cálculo suficiente para simular el sistema en tiempo real. El número de simulaciones permanece constante a pesar de que la carga del sistema aumenta. No hay simulaciones incorrectas. La QoS de cada objeto se mantiene. En estas situaciones, el sistema puede adaptarse dinámicamente, relajando las condiciones de QoS. De este modo se libera tiempo de CPU que puede hacer sincronizarse el sistema con el tiempo real. En el apartado 5.2.2.6 se tratan estas circunstancias con mayor profundidad.

Fly3D es eficiente cuando la potencia de cálculo de la máquina es insuficiente

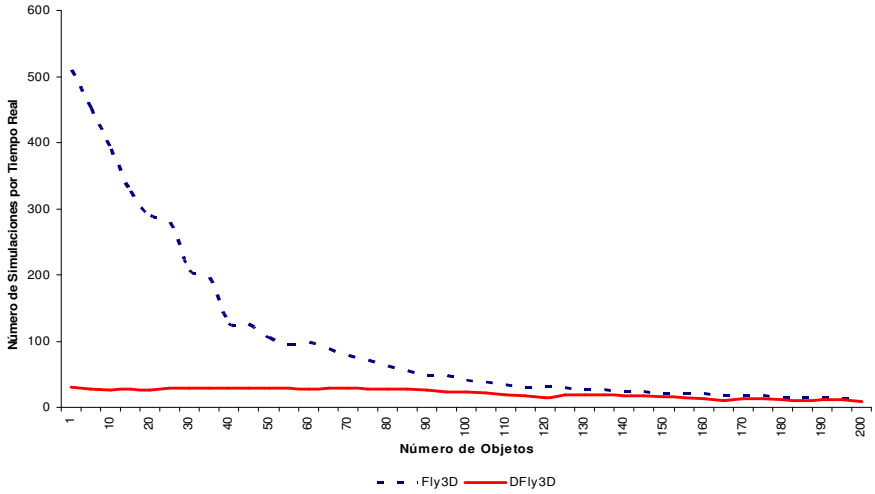

Figura 5.19: Número de simulaciones del objeto por unidad de tiempo real en Fly3D y DFly3D

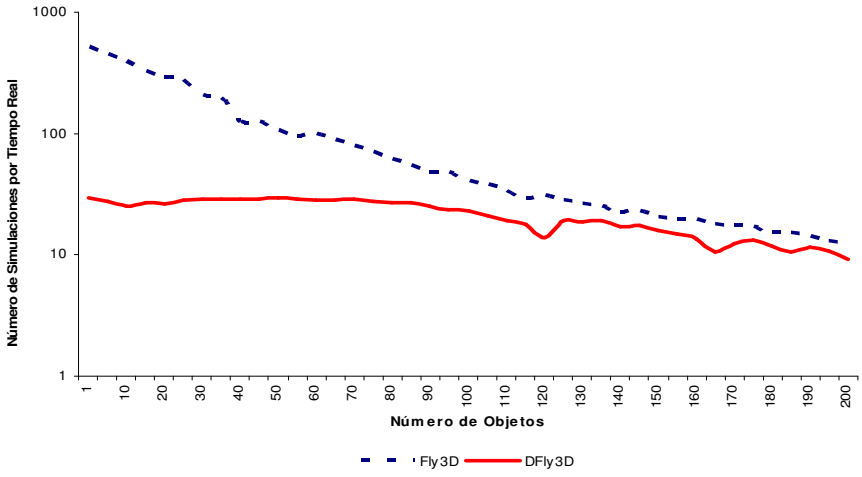

Figura 5.20: Número de simulaciones del objeto por unidad de tiempo real en Fly3D y DFly3D (escala logarítmica)

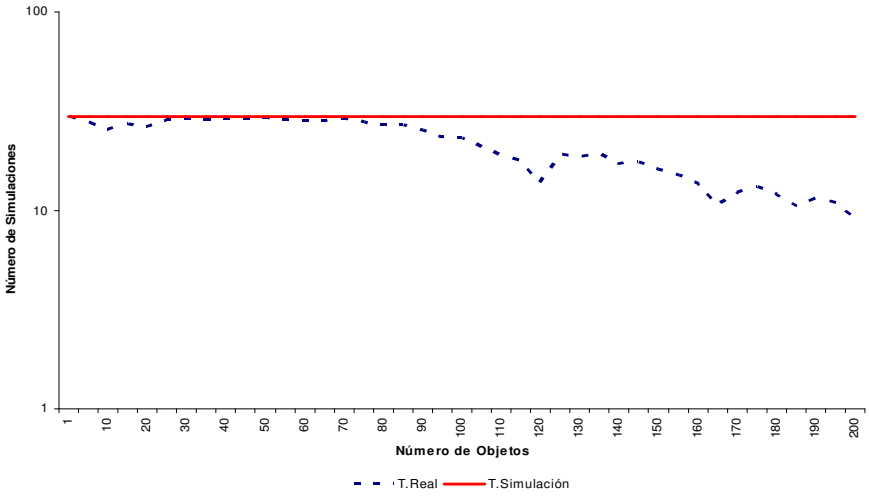

Figura 5.21: Número de simulaciones del objeto por unidad de tiempo real y por unidad de tiempo simulado en DFly3D (escala logarítmica)

o la complejidad del videojuego es tan alta que el sistema está colapsado. En estas situaciones, un sistema continuo acoplado simula y visualiza a la máxima velocidad, aunque puede simular de forma incorrecta (no es capaz de mantener la QoS del sistema).

La figura 5.22 muestra la simulación de un sistema en DFLy3D con dos tipos de objetos, con comportamientos diferentes y, por tanto, con diferentes frecuencias de muestreo. La mitad de los objetos del sistema son de tipo *1* y la otra mitad de tipo *2*. La figura muestra como aumenta el colapso de los objetos de ambos tipos. Ambas líneas tienen una pendiente similar, por tanto el colapso afecta igual a los dos tipos de objetos. En la gráfica se muestran valores negativos, pero esto sólo es debido al error cometido en la toma de medidas, que hace que pequeños valores tengan un efecto desproporcionado.

Tiempo Real y Tiempo de Simulación

Fly3D sigue el tiempo real para realizar la simulación, aunque para ello deba disminuir la QoS del sistema. En Fly3D prevalece la ejecución en tiempo real sobre la corrección de la simulación.

DFly3D prioriza la correcta simulación sobre la ejecución en tiempo real, por lo que a medida que el sistema se colapsa, el tiempo real y el tiempo del sistema comienzan a diferir. La figura 5.23 muestra la diferencia entre el tiempo real y el tiempo de simulación en DFLy3D. Cuando el sistema no está colapsado el tiempo de simulación sigue el tiempo real. En el instante de colapso, ambos tiempos comienzan

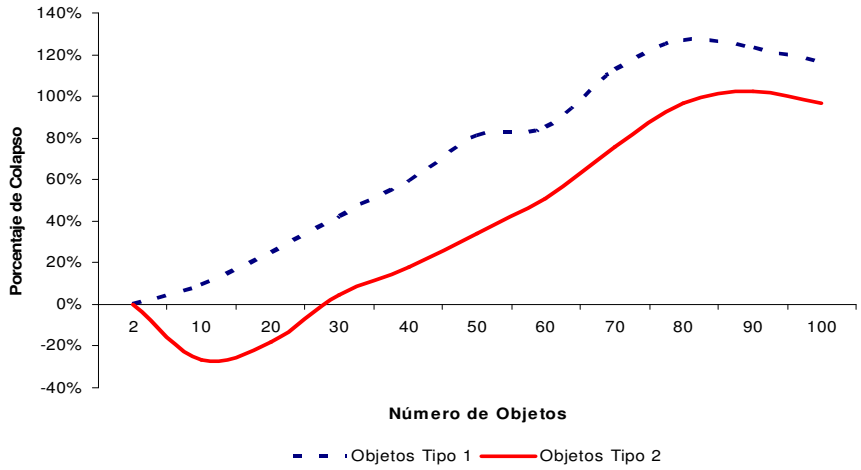

Figura 5.22: Colapso del sistema en DFly3D

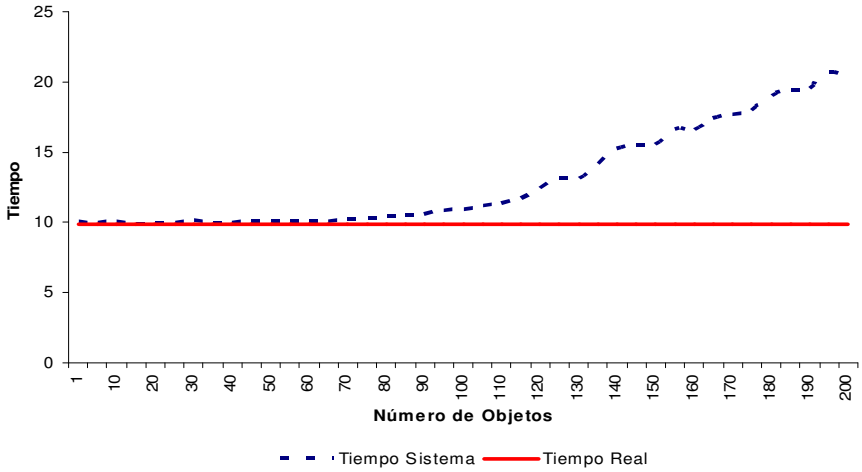

Figura 5.23: Relación entre el tiempo del sistema y el tiempo real en DFly3D

a diferir cada vez más. Cuanto más colapsado está el sistema mayor es la diferencia entre el tiempo real y el tiempo del sistema. Todos los eventos se ejecutan y consumen el tiempo definido por el programador.

El tiempo del sistema (tiempo de simulación) en DFLy3D siempre es el mismo independientemente de la carga del sistema, independientemente de si el sistema está o no colapsado.

5.2.2.4. Calidad de Servicio (QoS)

Los aspectos de la QoS considerados en las pruebas han sido:

1. Objeto *bola*:
   - La precisión en la detección de colisiones.
   - La precisión en la trayectoria de los objetos.
2. Objeto *visualizador*: visualización correcta sin parpadeo.

Muestreo de Objetos

Fly3D evoluciona el sistema en periodos de T unidades de tiempo, siendo la frecuencia de muestreo igual para todos los objetos e igual a la frecuencia de cuadro. La frecuencia de muestreo del sistema varía dinámicamente en función del coste temporal de la fase de simulación actual y la fase de visualización actual.

DFly3D permite que cada objeto tenga una frecuencia de muestreo diferente (incluso diferentes aspectos del mismo objeto pueden tener frecuencias diferentes). La frecuencia de cuadro se define independientemente del resto de las frecuencias de muestreo de los objetos.

La figura 5.14 muestra como varía el tiempo de simulación en Fly3D y DFly3D con el aumento del número de objetos. El tiempo de simulación en DFly3D es proporcional al número de objetos, pues el número de simulaciones es constante. En cambio, no hay una relación directa entre el número de simulaciones en Fly3D y el número de objetos, pues aumenta la carga del sistema disminuyendo SSF.

Frecuencia de Cuadro

La frecuencia de cuadro del sistema es la frecuencia de muestreo del objeto *visualizador*.

En Fly3D la frecuencia de cuadro es igual a la frecuencia de muestreo del sistema y no es configurable por el programador. La FF depende del coste temporal de las fases de simulación y visualización. El sistema constantemente simula y visualiza a la máxima velocidad. Fly3D no permite definir la QoS del objeto *visualizador*.

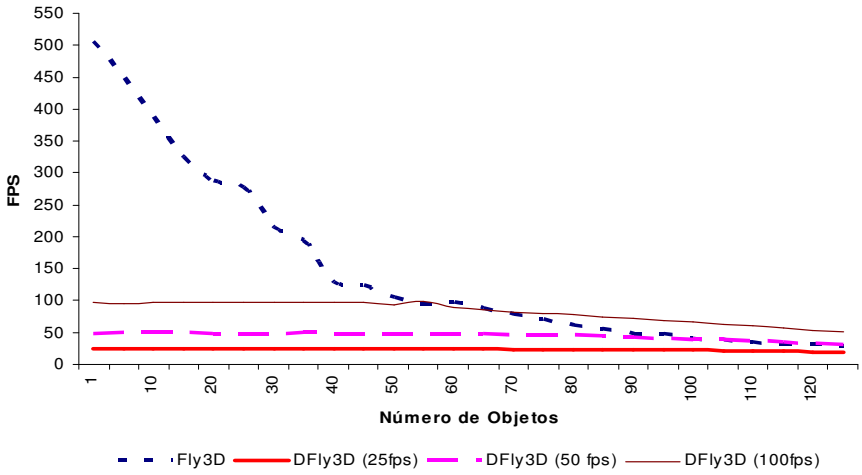

Figura 5.24: Número de escenas generadas en Fly3D y DFly3D

DFly3D permite definir cual es la FF óptima que debe generar el videojuego. La FF definida se mantiene mientras el sistema no esté colapsado.

La figura 5.24 muestra el número de escenas generadas por ambos sistemas (para DFLy3D se muestran los fps generados en diferentes ejecuciones con diferentes valores de FF). El programador define cual es la FF óptima del sistema en DFly3D. Si la FF óptima es 25 fps, el tiempo de visualización de DFly3D es siempre menor que en Fly3D, por lo que durante toda la ejecución de Fly3D se generan visualizaciones innecesarias (para la QoS definida). Con 50 fps o 100 fps, hasta un número de objetos determinado, Fly3D genera visualizaciones innecesarias y a partir de ese número de objetos no es capaz de mantener la QoS de visualización.

La figura 5.25 muestra la diferencia entre los fps generados por Fly3D y DFly3D, manteniendo la misma calidad de imagen. DFly3D fija la FF y la mantiene mientras el sistema no se colapsa. Fly3D, sin embargo, visualiza innecesariamente desperdiciando potencia de cálculo y no es capaz de asegurar la QoS.

DFly3D evita visualizaciones innecesarias, liberando la potencia de cálculo.

La figura 5.26 muestra el número máximo de escenas que puede visualizar cada sistema. DFly3D puede alcanzar una FF mayor que Fly3D aunque el sistema empiece a colapsarse. Las pruebas realizadas muestran que la FF que DFly3D es capaz de generar es siempre un poco mayor que la FF por Fly3D. Esto se debe a una mejor distribución de la potencia de cálculo de la máquina.

Cuando se alcanza la tasa máxima de escenas generadas ambos sistemas usan el 100 % del tiempo de la CPU (DFly3D se comporta como Fly3D y por tanto no

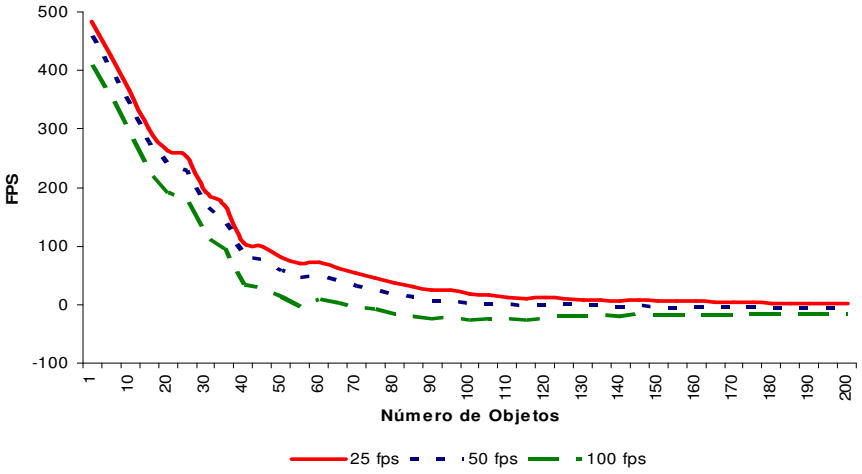

Figura 5.25: Diferencia en el número de escenas generadas en Fly3D y DFly3D

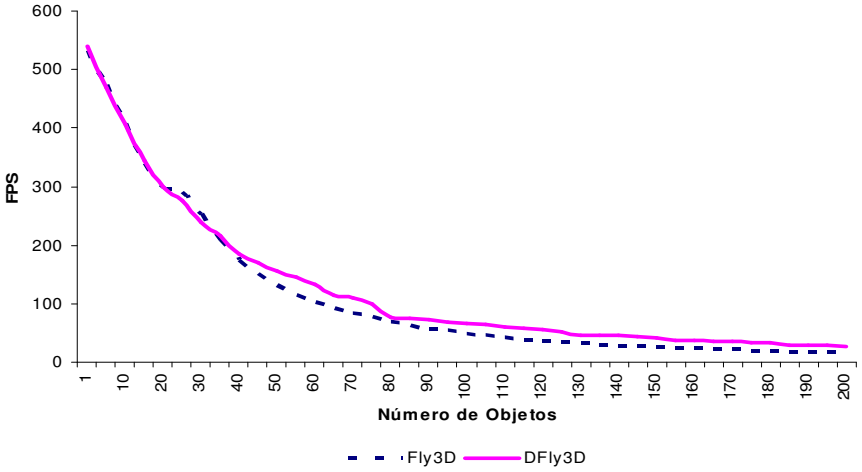

Figura 5.26: Número máximo de escenas generadas en Fly3D y DFly3D

libera tiempo).

Detección de Colisiones

DFly3D utiliza los procesos de Fly3D, sólo se modifica la gestión de eventos del sistema. Por tanto, procesos como visualización o detección de colisiones son los originales de Fly3D. No se han variado estos procesos pues no era objetivo de la tesis comprobar si un proceso u otro mejoraba el rendimiento del sistema. Pero la utilización de estos procesos ha supuesto ciertas limitaciones para la tesis.

Fly3D utiliza un algoritmo de detección de colisiones no determinista. Cada vez que un objeto cambia de posición, se comprueba si durante este movimiento se ha producido alguna colisión. La comprobación se realiza con los objetos en el instante actual, lo que supone que ciertos objetos habrán evolucionado su posición, pero el resto de los objetos no se han evolucionado a su nueva posición. En Fly3D, dependiendo de su situación en el grafo de escena respecto al objeto actual, unos objetos habrán evolucionado y otros no. Por tanto, el algoritmo de detección de colisiones considera unos objetos evolucionados y otros no.

El proceso de detección de colisiones de Fly3D se arranca cuando un objeto cambia su posición y sigue los siguientes pasos:

1. Dada la posición P_0 original del objeto, calcular la posición destino del objeto P_1 si no hubiesen colisiones.
2. Invocar la detección de colisiones. Este proceso calcula como máximo 3 rebotes en la colisión del objeto, para cada rebote i:
   - Calcular la dirección y longitud del movimiento.
   - Si la longitud del movimiento es inferior a un cierto umbral, se termina el proceso sin detectar colisiones.
   - Comprobar si hay colisión:
     - Crear una caja de movimiento que contenga todo el movimiento del objeto, desde la última vez que se detectaron colisiones (desde la posición P_0 o desde la posición resultante del rebote anterior P_{2_i}).
     - Comprobar si hay colisiones con esta caja de movimiento.
   - Si hay colisión, calcular la nueva posición P_{2_i}. Este proceso devuelve la nueva posición P_2, si se han detectado colisiones.
3. Modificar la posición del objeto a P_1 o P_2, según corresponda: si se han detectado colisiones: $P = P_2$, si no se han detectado $P = P_1$.

El algoritmo de detección de colisiones sólo arranca la detección de colisiones si el objeto se ha desplazado más de un cierto umbral. Por ello cuando el mues-

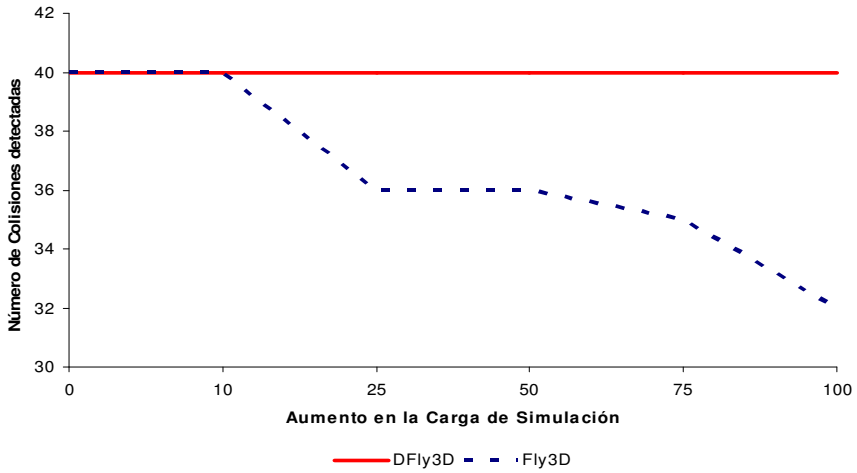

Figura 5.27: Número de colisiones detectadas a medida que aumenta la carga de simulación en Fly3D y DFly3D

treo del objeto aumenta considerablemente, se detectan menos colisiones, pues el desplazamiento del objeto es demasiado pequeño.

Se ha observado que cuando se varía la frecuencia de muestreo de los objetos en DFly3D, el número de colisiones detectadas y la posición de estas colisiones varía.

Cuando el número de objetos en el sistema que colisionan es pequeño y la velocidad de éstos es baja, las colisiones detectadas en DFly3D son las mismas y se detectan, prácticamente, en la misma posición, independientemente de la frecuencia de muestreo (a no ser que la frecuencia de muestreo sea muy elevada). Cuando el número de objetos crece o la velocidad de los objetos aumenta, el efecto no determinista del algoritmo de colisiones empleado produce que las colisiones difieran a medida que se varía la frecuencia de muestreo de los objetos.

Si Fly3D dispusiese de un algoritmo determinista de detección de colisiones, en DFly3D se podría ajustar la frecuencia de muestreo para que la detección de colisiones fuese la óptima (para detectar todas las colisiones con un comportamiento visual adecuado y sin sobrecargar el sistema con muestreos innecesarios). Además, esta frecuencia de muestreo óptima puede definirse para cada objeto. Sin embargo, este aspecto no ha podido probarse en DFly3D, por lo que esta hipótesis queda pendiente como futura línea de investigación de la tesis (cambiando el algoritmo de detección de colisiones o integrando GDESK en una aplicación gráfica en tiempo real con un algoritmo de detección de colisiones determinista).

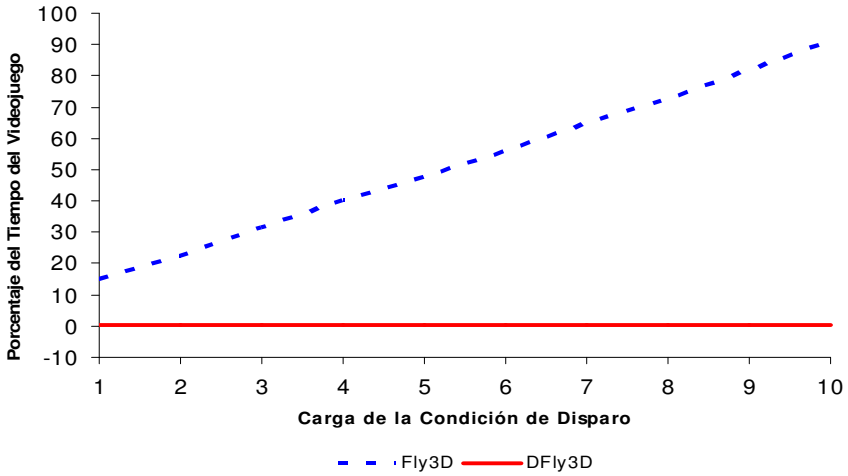

Figura 5.28: Incremento de coste por muestreo en FLy3D respecto a la utilización de eventos programados en DFly3D

La figura 5.27 muestra el número de colisiones detectadas en DFLy3D y Fly3D a medida que aumenta la carga de simulación del sistema (permaneciendo constante el número de objetos). En DFly3D, la simulación es la misma, independientemente de la carga. El sistema se ralentiza para simular correctamente, pero la simulación permanece inalterable. Por tanto, el número de colisiones detectadas es contante e independiente de la carga de simulación. En cambio, en Fly3D una mayor carga de simulación supone que se simula un número de veces menor. Por tanto, Fly3D puede no detectar colisiones dependiendo de la carga del sistema, degradando la QoS del sistema.

5.2.2.5. Eventos Programados

La figura 5.28 muestra el coste temporal de la programación en Fly3D y DFly3D de un evento programado (una bomba de relojería que debe estallar dentro de T segundos).

Fly3D define los eventos programados mediante muestreo, por lo que es necesaria una condición de disparo (en este caso indica si se ha alcanzado el tiempo en que debe estallar la bomba). Las pruebas en Fly3D se han realizado con la condición de disparo original y posteriormente se ha aumentado la carga de la comprobación de la condición de disparo para ver como aumenta la carga del sistema.

En DFly3D únicamente se programa un evento que sucederá en T segundos. Este proceso tiene un coste inapreciable. No hay muestreo de la condición de disparo.

5.2.2.6. Adaptación Dinámica del Sistema

La adaptación (degradación o mejora) dinámica del sistema puede llevarse a cabo utilizando las herramientas de monitorización de DFLy3D.

Adaptación de la Complejidad de la Simulación del Objeto

Utilización de Muestreo en Fly3D y DFly3D

En las pruebas realizadas, los resultados obtenidos son muy similares para ambos sistemas si se utiliza muestreo. Si la carga de simulación es alta y el sistema está colapsado, al reducir la carga mediante adaptación dinámica un 60 % aproximadamente, el sistema consigue pasar de una frecuencia de cuadro de 19fps a 25fps. Variando el número de objetos del sistema los porcentajes de mejora obtenidos han sido similares. No varían los resultados en ambos sistemas porque la mejora no afecta al número de muestreos del objeto. Ambos sistemas dedican el 100 % del tiempo a simular y visualizar. La diferencia entre los resultados de ambos sistemas radica en la QoS de la simulación. DFly3D realiza el mismo número de muestreos del objeto, a costa de aumentar el tiempo real de la simulación. Fly3D degrada el resultado de la simulación.

Utilización de Muestreo en Fly3D y Eventos Programados en DFly3D

Si se usan eventos para modelar la adaptación en DFly3D, los resultados son muy diferentes, pues las conclusiones son muy similares a los obtenidos en la utilización de eventos programados (apartado 5.2.2.5). La mejora de DFly3D respecto a Fly3D es notable. Estos resultados son muy dependientes del ejemplo en concreto.

Adaptación del Número de Muestreos del Objeto

Variación en el Número de Muestreos del Objeto

La frecuencia de muestreo de un objeto en DFly3D puede definirse de forma que varíe dinámicamente. La figura 5.29 muestra un sistema donde el número de muestreos de un objeto varía dinámicamente en DFly3D. Esta situación no es posible definirla usando Fly3D, por lo que el muestreo del objeto es constante durante la ejecución. En DFly3D, el programador siempre tiene el control sobre la frecuencia de muestreo del objeto, sea constante o variable.

La figura 5.30 muestra la diferencia en el número de muestreos de Fly3D y DFly3D. DFly3D cambia la frecuencia de muestreo del objeto para adaptarse a la QoS cambiante del objeto. Por tanto DFly3D muestrea el objeto de forma óptima. La figura 5.30 muestra el número de muestreos innecesarios que realiza Fly3D para la situación mostrada en la figura 5.29. Fly3D genera muestreos innecesarios, por lo que se consume potencia de cálculo de la máquina innecesariamente que DFLy3D libera.

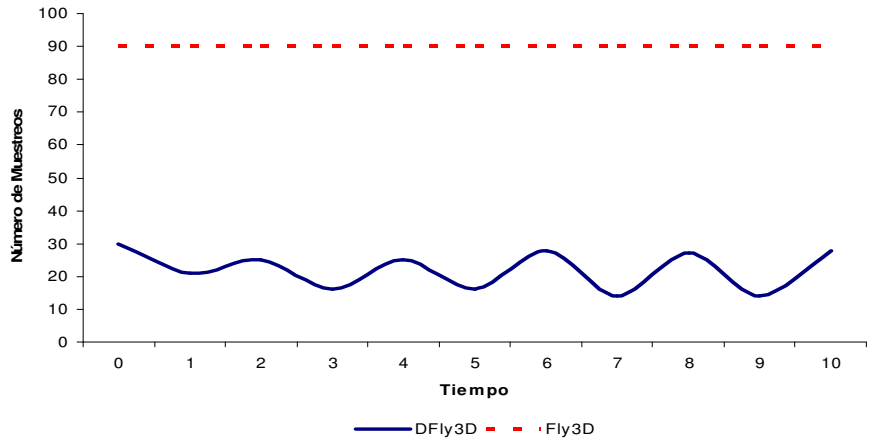

Figura 5.29: Número de muestreos de un objeto en Fly3D y DFly3D

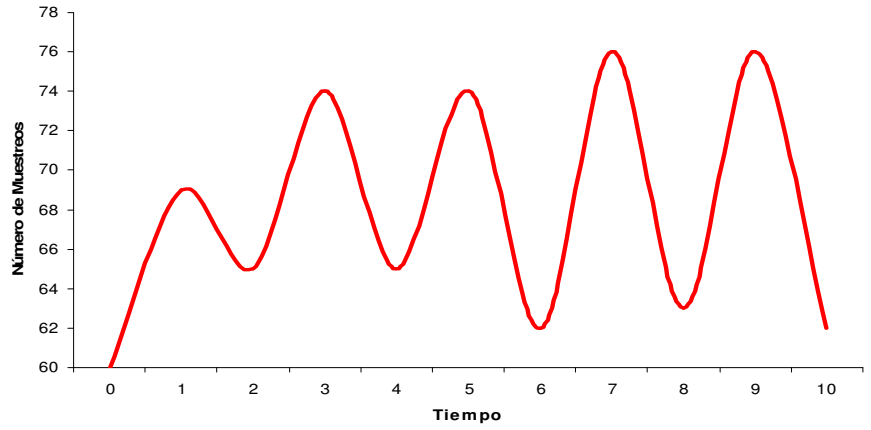

Figura 5.30: Diferencia en el número de muestreos de un objeto en Fly3D y DFly3D

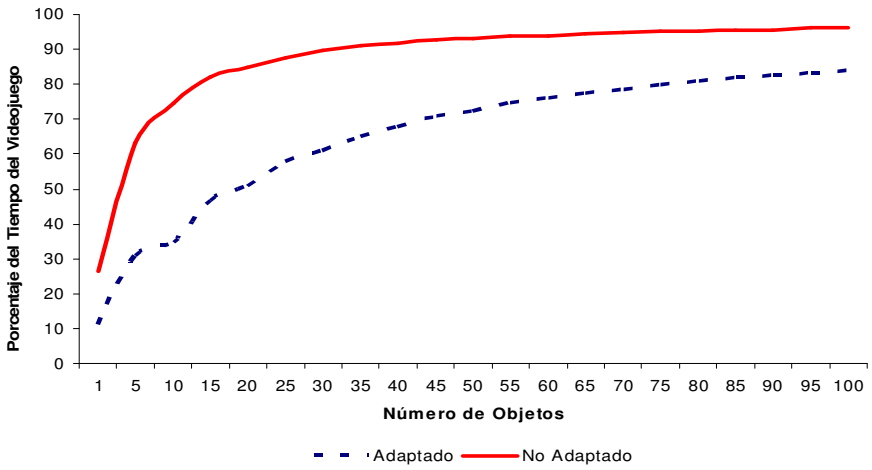

Figura 5.31: Evolución del tiempo de simulación de los objetos en un sistema adaptado y no adaptado en DFly3D

Ejemplo de Adaptación Dinámica del Muestreo con Costes

Supóngase el ejemplo de un sistema con dos tipos de objetos. Los objetos tipo A tienen una frecuencia de muestreo de 30 veces por segundo, mientras que los objetos tipo B, al tener una velocidad mayor, necesitan una frecuencia de muestreo de 60 veces por segundo. La carga de los objetos de tipo B es 10 veces mayor que la carga de los objetos de tipo A. Si el sistema está colapsado se puede reducir su muestreo hasta 10 muestreos por segundo. En el sistema adaptado los objetos tipo A se muestrearán correctamente, mientras que los objetos tipo B se submuestrearán cuando el sistema esté colapsado. Como condición crítica se ha utilizado que el tiempo de simulación aumente un 20 % sobre el tiempo real. Mientras se cumpla esta condición, el sistema adoptará la nueva frecuencia de muestreo de los objetos tipo B. Cuando esta condición deje de cumplirse, el sistema volverá a su situación inicial.

Fly3D no es capaz de simular esta situación adecuadamente, por lo que lo que se va a mostrar es como la adaptación en DFly3D permite aliviar el colapso del sistema.

En DFly3D se consigue que disminuya el tiempo de simulación de los objetos (figura 5.31) y que aumente la frecuencia de cuadro efectiva del sistema (figura 5.32), al disminuir algo el colapso del sistema. Estas gráficas muestran el comportamiento de un caso concreto, podría haberse optado por disminuir el muestreo de todos los objetos o disminuir el muestreo de los objetos de tipo A que son más lentos.

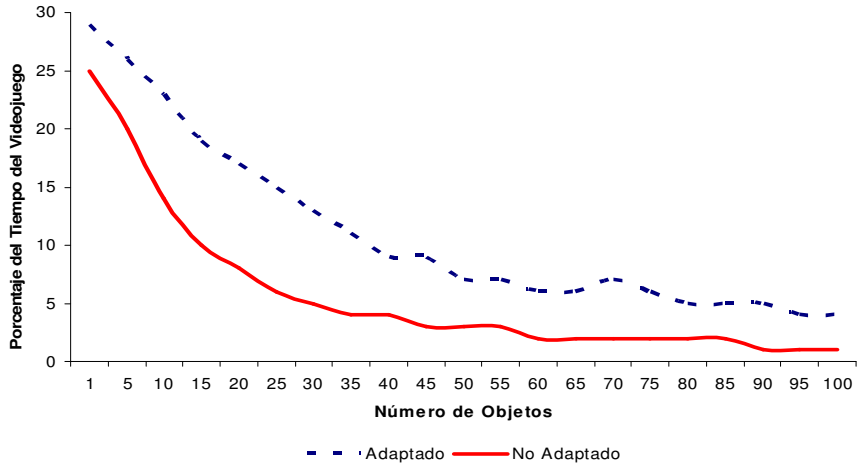

Figura 5.32: Evolución de la frecuencia de cuadro en un sistema adaptado y no adaptado en DFly3D

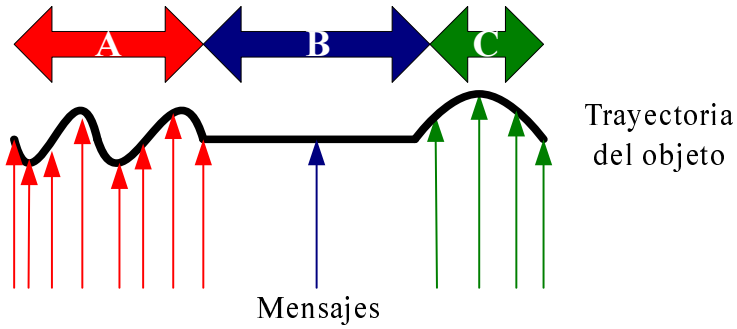

Figura 5.33: Ejemplo de muestreo adaptativo de un objeto en DFly3D

Sobremuestreo o Submuestreo del Comportamiento del Objeto

Otro ejemplo de muestreo adaptativo de un objeto es el siguiente. La figura 5.33 muestra un objeto cuyo comportamiento varía en el tiempo y necesita diferentes frecuencias de muestreo en cada intervalo (A, B y C). Durante los intervalos A y C, el objeto necesita un frecuencia de muestreo mayor que durante el intervalo B.

La frecuencia de muestreo de Fly3D es común a todo el sistema (figura 5.34). Todos los objetos se muestrean a la misma frecuencia y todos los aspectos del objeto se muestrean a la misma frecuencia, también. Por tanto, si la QoS del objeto requiere una frecuencia de muestreo mayor, el objeto se submuestrea. Si el objeto requiere una frecuencia de muestreo menor, el objeto se sobremuestrea. DFly3D adapta el muestreo a la QoS de los objetos, permitiendo definir una frecuencia de

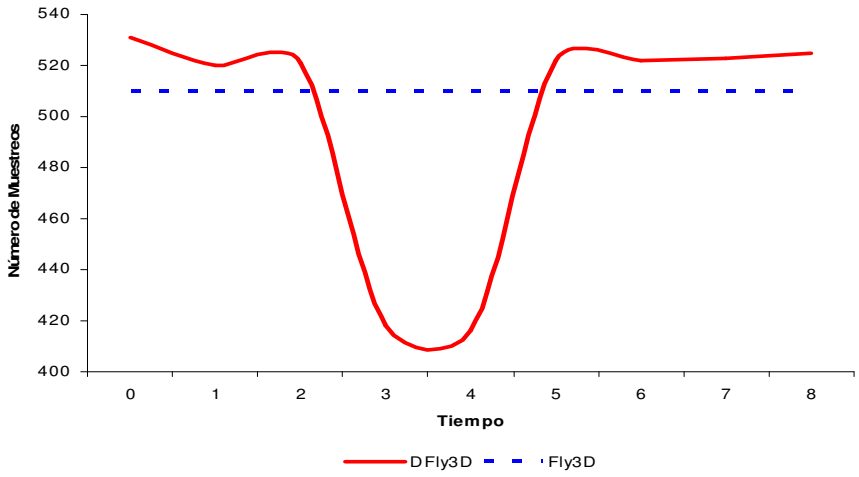

Figura 5.34: Número de muestreos del objeto (figura 5.33) en Fly3D y DFly3D

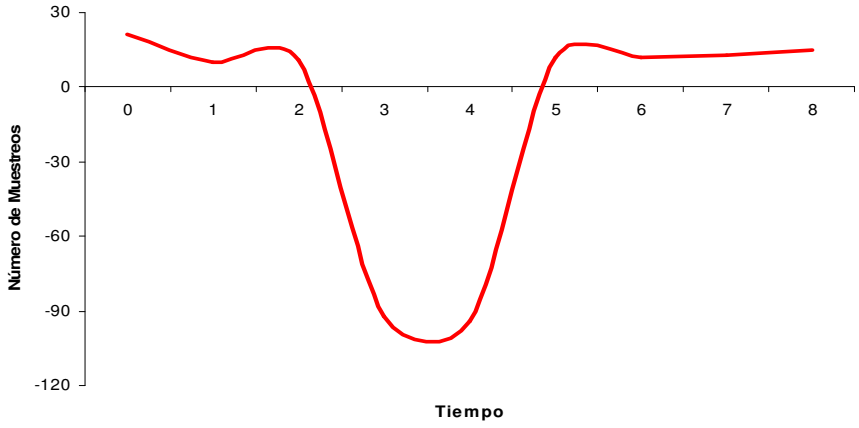

Figura 5.35: Diferencia en el número de muestreos del objeto (figura 5.33) en Fly3D y DFly3D

muestreo diferente para cada objeto. La frecuencia de muestreo de DFly3D puede variar dinámicamente para adaptarse al comportamiento del objeto (figura 5.34).

La figura 5.34 muestra el número de muestreos durante la ejecución en ambos sistemas. El número de muestreos en Fly3D permanece constante durante toda la ejecución independientemente del comportamiento del objeto. Sin embargo, el número de muestreos de DFly3D se adapta a cada uno de los intervalos con diferente comportamiento del objeto. DFly3D define para el objeto la frecuencia de muestreo óptima en cada intervalo.

La figura 5.35 muestra la diferencia en el número de muestreos de ambos sistemas durante la ejecución del videojuego:

- Los valores positivos corresponden al intervalo de tiempo B, cuando Fly3D esta submuestreando el objeto.
- Los valores negativos corresponden al intervalo de tiempo A y C, cuando Fly3D esta sobremuestreando el objeto.

Adaptación de la Frecuencia de Cuadro

Los resultados obtenidos para la adaptación de la frecuencia de muestreo son extrapolables a la adaptación de la frecuencia de cuadro.

En Fly3D no se puede adaptar la frecuencia de cuadro, pero DFly3D si lo permite. Basta definir las condiciones o los instantes de tiempo en que la frecuencia de cuadro debe modificarse y generar en ese momento eventos programados al objeto *visualizador*. El objeto visualizador también puede utilizar las herramientas de monitorización para adaptar su comportamiento, en función, por ejemplo, del factor de colapso del sistema o de la historia previa de la frecuencia de cuadro.

Adaptación Combinada

Consiste en combinar las técnicas anteriores, con resultados extrapolables y muy dependientes del ejemplo en concreto.

5.2.2.7. Sobrecarga por la Gestión de Eventos Discretos en DFLy3D

El tiempo consumido por los procesos de simulación y visualización de DFly3D incluye el tiempo consumido en simular y visualizar y el tiempo consumido en el procesamiento de los mensajes de los objetos. La sobrecarga de cálculo producida por la gestión de eventos de DFly3D es mínima (figura 5.36), por lo que la potencia de cálculo de ambos sistemas es similar.

Si el sistema está colapsado, la sobrecarga en la gestión de eventos, aunque mínima, produce que Fly3D sea más óptimo que DFly3D.

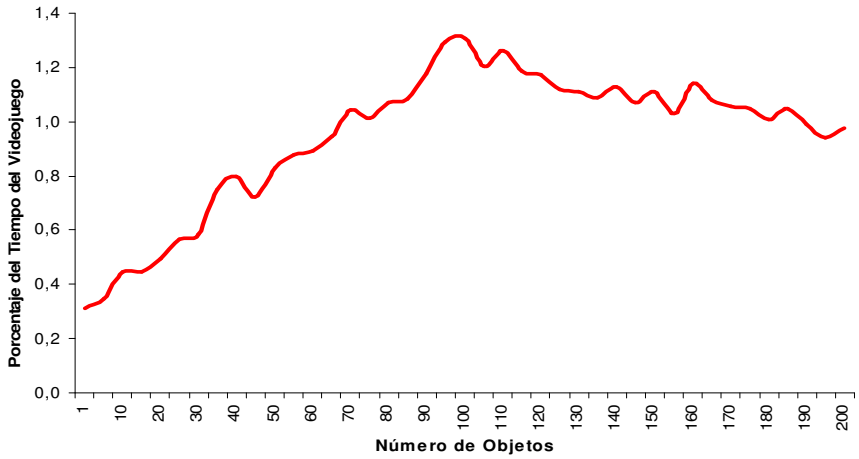

Figura 5.36: Sobrecarga de tiempo por la gestión de eventos en DFly3D

5.2.2.8. Monitorización del Sistema

La monitorización del sistema es una herramienta que se incluye en DFly3D que no existía en Fly3D. Se apoya en el núcleo de simulación GDESK para llevar a cabo la monitorización.

La monitorización sirve, entre otras cosas, para implementar los procesos que se han visto en apartados anteriores, como adaptación dinámica del sistema.

Sin embargo, la monitorización consume recursos del sistema. Los recursos consumidos dependen en gran medida de la monitorización utilizada. Se definen dos tipos de monitorización en DFLy3D (apartado 4.7 del capítulo 4):

1. Monitorización en línea.
2. Monitorización fuera de línea.

Todos los tipos de monitorización y sus subcategorías se seleccionan mediante sentencias de preprocesador, de forma que el sistema sólo se sobrecarga si explícitamente el programador desea monitorizar el sistema y con el coste de la monitorización elegida.

Monitorización en Línea

La monitorización en línea permite adaptar el sistema dinámicamente. Este tipo de monitorización puede llevarse a cabo mediante parámetros del sistema o mediante mensajes de control

El monitor contiene una serie de parámetros que actualiza cada cierto intervalo de tiempo definido por el programador del videojuego. Algunos de estos parámetros son: frecuencia de cuadro media e instantánea, tiempo libre del sistema o factor de colapso. Los objetos del videojuego tienen la posibilidad de usarlos o no.

Otra posibilidad de monitorización en línea es mediante mensajes de control. Si se selecciona este tipo de monitorización, el monitor enviará mensajes a los objetos si se supera un cierto valor crítico de los parámetros de control. Cada objeto decide si es sensible o no a cada posible mensaje.

El coste de la actualización de los parámetros del monitor para que los objetos del sistema los puedan consultar es inapreciable. El programador define la frecuencia con la que se actualizan los parámetros del monitor y la frecuencia con la que se inicializan las estructuras intermedias de cálculo de valores. Si la actualización es muy frecuente, el coste de la monitorización aumenta. Pero en condiciones normales no debe suponer un incremento considerable.

El coste de la monitorización mediante mensajes de control también suele tener un coste inapreciable, salvo en el caso de que los mensajes de control sean muy frecuentes.

Monitorización Fuera de Línea

DFly3D permite monitorizar fuera de línea de los siguientes procesos:

- Detección de colisiones.
- Tiempos del sistema.
- Evolución de la frecuencia de cuadro durante la ejecución.
- Traza completa del sistema.

Los tres primeros procesos producen una sobrecarga inapreciable, pues contiene resultados que en algunos casos se calculan durante la ejecución, independientemente de la monitorización.

El proceso más costoso, pero que más información aporta es la traza del sistema. La figura 5.37 muestra la evolución del coste temporal de simulación de un sistema cuando se utiliza la monitorización mediante traza a medida que aumenta el número de objetos en el sistema y por tanto mayor número de eventos hay que incluir en la traza. Los tiempos llegan a ser hasta 45 veces mayores.

5.2.3. Conclusiones

Sea:

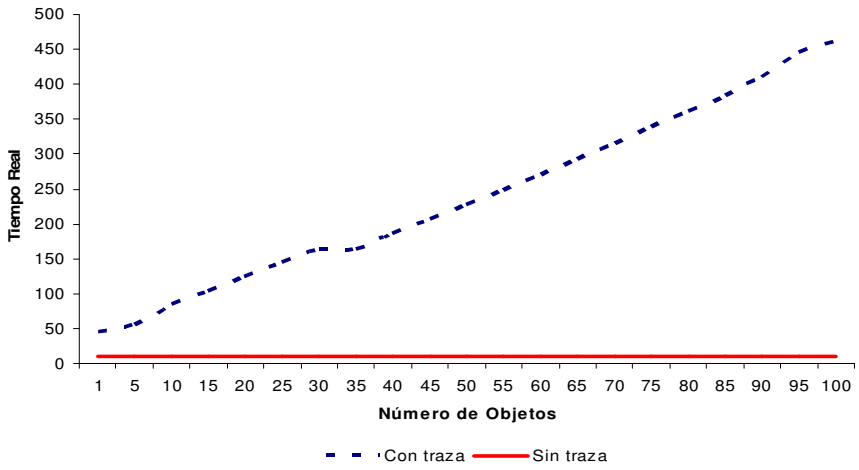

Figura 5.37: Sobrecarga de tiempo por la realización de una monitorización del sistema mediante traza en DFly3D

SC sistema colapsado.

SNC sistema no colapsado.

La tabla 5.18 muestra una comparativa resumen de ambos sistemas.

En este capítulo se muestran los resultados de la comparación entre el sistema continuo acoplado Fly3D y el sistema discreto desacoplado DFly3D (obtenido de la integración de GDESK en Fly3D). Las conclusiones obtenidas de estos resultados son extrapolables a los sistemas continuos acoplados y discretos desacoplados.

El sistema continuo constantemente simula y visualiza a la máxima velocidad que la potencia de cálculo es capaz de proporcionar. Consume todos los recursos disponibles. Sin embarco, el sistema discreto consume únicamente el tiempo que el programador ha destinado al proceso de visualización y a la simulación de cada objeto. El resto del tiempo lo libera. El sistema discreto es capaz de ejecutar la misma simulación con un consumo menor de recursos.

El sistema continuo muestrea todos los objetos del sistema con la misma frecuencia, independientemente de la QoS de cada objeto. Esta frecuencia es igual a la frecuencia de cuadro. La frecuencia de muestreo del sistema puede tener un valor tal que haya objetos que se sobremuestreen y otros que se submuestreen. No es posible definir una frecuencia diferente para cada objeto. El sistema discreto permite definir una frecuencia de muestreo diferente para cada objeto, incluso una frecuencia diferente para cada aspecto de cada objeto. El sistema discreto es capaz de adaptarse a las necesidades de QoS de cada objeto y de cada aspecto de cada objeto. Los objetos

Característica	**Sistema Continuo**	**Sistema Discreto**
Simulación	Continua Acoplado	Discreta Desacoplado
T_G SNC	$T_G \simeq T_R + T_S$	$T_G = T_R + T_S + T_F$
T_G SC	$T_G = T_R + T_S$	$T_G = T_R + T_S$
T_F SNC	$T_F = 0$	$T_F = T_G - T_V - T_R$
T_F SC	$T_F = 0$	$T_F = 0$
$T_S \uparrow$	$T_R \downarrow, FF \downarrow$	$T_F \downarrow, FF \simeq$
$T_R \uparrow$	$T_S \downarrow, FF \downarrow$	$T_F \downarrow, FF \simeq$
$N_O \uparrow$	Reparto del aumento entre T_S y T_R	Aumento lineal de T_S y T_R y decremento proporcional de T_F
N_R, N_S	$N_R = N_S$	$N_R \neq N_S$
Reparto de potencia de calculo entre los objetos	No uniforme No adecuada	Depende de las necesidades del objeto Adecuada
T_{REAL} SNC	$T_{SISTEMA} = T_{REAL}$	$T_{SISTEMA} = T_{REAL}$
T_{REAL} SC	$T_{SISTEMA} = T_{REAL}$	$T_{SISTEMA} < T_{REAL}$
$CF_S \uparrow$	$T_{SISTEMA} \simeq$ $T_{REAL} \simeq$	$T_{SISTEMA} \simeq$ $T_{REAL} \uparrow$
Muestreo del sistema SSF	$\forall i \in O : OSF_i = SSF$ Común a todo el sistema	$\exists i \in O : OSF_i \neq OSF_j$ No existe
FF	$FF = SSF$ $OSF_i = FF$	$FF = OSF_{visualizador}$ $OSF_i \neq FF$
Muestreo de objetos	$\forall i, j \in O : OSF_i = OSF_j$	$\exists i, j \in O : OSF_i \neq OSF_j$
OSF_i	Variable	Constante o adaptable
OSF_i definida por el programador	No	Si
OSF_i adaptable dinámicamente	No	Si
OSF_i dependiente de la carga del sistema	Si	No
OSF_i SNC	$\forall i \in O : OSF_i = SSF \neq OSF_{i_{min}}$	$\forall i \in O : OSFs_i = OSFr_i = OSF_i$ $OSF_i \in [OSF_{i_{min}}..OSF_{i_{max}}]$
OSF_i SC	$\forall i \in O : OSF_i = SSF \neq OSF_{i_{min}}$	$\forall i \in O : OSF_i = OSF_{i_{min}}$

Tabla 5.18: Comparativa del sistema continuo y discreto

siempre se muestrean a la frecuencia óptima. Por tanto, el sistema discreto permite cumplir la QoS definida para cada objeto.

El sistema continuo no tiene en cuenta las necesidades de cada objeto a la hora de repartir la potencia de cálculo del sistema, pues la frecuencia de muestreo es común a todo el sistema. El sistema discreto permite optimizar el reparto de la potencia de cálculo, al permitir al programador repartir la potencia de cálculo en función de la QoS de cada objeto. La QoS de los objetos debe tener en cuenta la correcta simulación de los objetos y otros aspectos relacionados con la estimulación áptica para dispositivos HCI.

En el sistema continuo todos los procesos están acoplados. En cada iteración del bucle principal de la aplicación se simulan todos los objetos recorriendo el grafo de escena y después se visualiza la escena actual. En el sistema discreto cada objeto es independiente del resto e independiente del proceso de visualización. En el sistema continuo todos los objetos están desacoplados. El comportamiento de cada objeto está desligado del resto.

La frecuencia de cuadro depende de la carga del sistema en el sistema continuo y está integrada con el resto de procesos (igual frecuencia de muestreo). En el sistema discreto, el proceso de visualización está desacoplado del proceso de simulación del resto de objetos (es independiente) y tiene una frecuencia diferente.

Cuando el sistema no es capaz de simular y visualizar correctamente en un ciclo de refresco de pantalla se dice que el sistema está colapsado. El colapso del sistema tiene efectos diferentes en el sistema continuo y en el discreto. El sistema continuo colapsado tiene un funcionamiento incorrecto (colisiones no detectadas, trayectorias incorrectas,...). El comportamiento incorrecto se acentúa cuando aumenta el factor de colapso. El sistema discreto en cambio simula el sistema correctamente independientemente del colapso del sistema. Para conseguir una simulación correcta sacrifica la ejecución del sistema en tiempo real. El sistema discreto colapsado consume un tiempo real en ejecutar la simulación mayor que el tiempo de simulación del sistema. La diferencia entre estos dos tiempos será mayor cuanto mayor sea el factor de colapso. Ante el colapso del sistema, el sistema discreto degrada la QoS de los objetos, pero el sistema discreto la mantiene.

El sistema discreto es capaz de tomar información del proceso de monitorización del sistema y adaptarse dinámicamente. La QoS de cada objeto puede redefinirse dinámicamente para adaptarse a las condiciones del sistema. Si el sistema está colapsado, uno o más objetos pueden reducir su QoS, para evitar el colapso. Si el sistema libera tiempo (hay tiempo remanente en su ejecución), los objetos pueden readaptar su QoS, para aprovechar este tiempo libre en mejorar su inteligencia artificial, detección de colisiones,... En el sistema discreto los objetos definen un rango de QoS por si el sistema debe adaptarse ($[QoS_{min}..QoS_{max}]$). La adaptación del sistema puede

implicar que algunos objetos se degraden en beneficio de otros. La gestión de eventos discreta mediante paso de mensajes hace que este proceso no sólo sea posible llevarlo a cabo, sino que se realice de una forma sencilla de programar y con una sobrecarga del sistema muy baja (aunque muy dependiente del uso que el programador de la aplicación haga de ella).

El sistema discreto incluye un monitor del sistema que permite monitorizar las aplicaciones en línea y fuera de línea. El monitor del sistema es capaz de comunicarse con el resto de objetos del sistema para que adapten su comportamiento si se produce alguna condición crítica definida por el programador. También pone a disposición de los objetos información crítica del comportamiento del sistema por si los propios objetos desean usarla para definir adecuadamente su QoS y adaptarla dinámicamente. La monitorización fuera de línea produce una serie de información e históricos sobre la ejecución del sistema.

El sistema discreto permite:

- Desacoplar todos los objetos del sistema, no sólo la fase de simulación de la fase de visualización, sino todas las simulaciones de cada objeto, incluso todas las simulaciones de cada aspecto de cada objeto.
- Un reparto óptimo de los recursos del sistema entre los objetos.
- Una utilización más racional de la potencia de cálculo por parte de cada objeto del sistema, consumiendo únicamente la potencia de cálculo necesaria.
- Definir la QoS de cada objeto, independientemente del resto. Permite definir el rango de QoS que puede llegar a asumir el objeto si hay colapso en el sistema o si otros objetos necesitan alcanzar su QoS máxima.
- Adaptar dinámicamente la QoS de cada uno de los objetos del sistema.
- Ejecutar la simulación del sistema correctamente independientemente del colapso del sistema.
- Monitorizar el sistema de formas diferentes, dependiendo de las necesidades de cada fase de producción de la aplicación gráfica en tiempo real. El objeto monitor provee al resto de los objetos del sistema con información que el resto de objetos pueden utilizar. Se comunica con el resto de los objetos para que adapten su comportamiento si es necesario.

Capítulo 6

Conclusiones y Trabajos Futuros

6.1. Conclusiones

Las aplicaciones gráficas en tiempo real tradicionales, como los videojuegos, suelen seguir un esquema de simulación continua. Habitualmente este esquema de simulación obliga a un acople de las fases de visualización y simulación, aunque algunas aplicaciones permiten desacoplar estas fases. Este esquema de simulación se ha mostrado claramente ineficiente debido fundamentalmente a las siguientes razones:

1. El sistema puede presentar comportamientos incorrectos, como ejecución de eventos desordenados, prioridades de objetos no definidas por el programador o eventos no detectados.

2. La aplicación gráfica es altamente dependiente de la potencia de cálculo de la máquina. El reparto de la potencia de cálculo entre los objetos de la aplicación no se realiza en función de las necesidades de cada objeto.

3. Todos los objetos se muestrean a la misma frecuencia, independientemente de sus necesidades o criterios de QoS. Si la QoS de un objeto necesita un muestreo inferior a la frecuencia de muestreo del sistema, el objeto tendrá un comportamiento correcto, pero se generarán muestreos innecesarios. Si la QoS del objeto es superior a la frecuencia de muestreo, el objeto no se simula correctamente.

4. En las aplicaciones en las que se acoplan las fases de simulación y visualización, la frecuencia de cuadro es igual a frecuencia de muestreo del sistema. El programador no tiene control sobre la frecuencia de cuadro que debe generar la aplicación. No puede definirla ni adaptarla al dispositivo de salida. La frecuencia de cuadro es muy dependiente de la potencia de cálculo de la máquina.

5. El paradigma continuo asume que los objetos del sistema tienen un comportamiento continuo. Cualquier comportamiento discreto o híbrido debe ser modelado mediante simulación continua. Es complicado y poco intuitivo definir comportamientos discretos y aumenta el coste temporal de la simulación.

6. Si la potencia de cálculo de la máquina es elevada para las necesidades de la aplicación gráfica se desperdicia potencia de cálculo. El sistema está continuamente simulando y visualizando a la máxima velocidad. Se generan escenas que nunca se visualizan y simulaciones innecesarias.

7. El sistema no es capaz de manejar las situaciones de colapso del sistema y adaptarse dinámicamente a estas, redefiniendo la QoS de cada objeto separadamente.

La tesis propone el cambio de paradigma de simulación de las aplicaciones gráficas en tiempo real. El esquema de simulación propuesto es discreto desacoplado. Este esquema de simulación se obtiene sustituyendo el gestor de eventos continuo del sistema por un simulador de eventos discreto (adaptado). La nueva aplicación tendrá un comportamiento discreto, como consecuencia del cual desacopla las fases de simulación y visualización. Para alcanzar este objetivo se han seguido los siguientes pasos:

1. Se ha creado un simulador de eventos discreto con unas prestaciones óptimas (DESK). Este simulador se adapta posteriormente a la simulación en web (siguiendo la tendencia de otros simuladores), creando JDESK.

2. Se ha adaptado el simulador de eventos para funcionar como núcleo de aplicaciones gráficas en tiempo real (GDESK).

3. Se ha integrado el núcleo de simulación en una aplicación gráfica. La aplicación seleccionada es el núcleo de videojuegos Fly3D. El resultado de la integración es el núcleo de videojuegos DFly3D.

DESK es un núcleo de simulación de eventos discretos generalista orientado a objetos implementado como una biblioteca de C++. Es un simulador versátil y con tiempos de simulación bajos. Permite simular sistemas con cualquier complejidad y tamaño. Permite definir cualquier modelo fácilmente de forma modular. Los modelos se definen describiendo los elementos que componen el sistema y su interconexión (topología del sistema). Permite variar el modelo fácil y rápidamente, por lo que puede utilizarse como prototipador. Es un simulador muy preciso, por lo que puede utilizarse para obtener resultados finales. El coste temporal de la simulación de los modelos estudiados es bajo.

En DESK el comportamiento de cada elemento del modelo puede seleccionarse entre una serie de comportamiento predefinidos o puede definirse cualquier comportamiento que el usuario desee, por caprichoso que este sea. Por tanto, ofrece la posibilidad de definir cualquier comportamiento del sistema. El comportamiento, la estructura y la topología del sistema del sistema pueden variar dinámicamente, sin necesidad de detener la simulación y reconfigurar el modelo. Permite, por tanto, modelar sistemas que cambian dinámicamente, de una manera natural.

DESK permite incluir fácilmente dentro de la simulación elementos externos, como multimedia, alarmas o realizar la simulación en tiempo real.

DESK puede modelar sistemas que otros simuladores no pueden con un coste de implementación y simulación bajo. DESK llega donde otros simuladores no llegan.

La evolución de DESK ha seguido los pasos de otros simuladores de eventos discretos, adaptándose a las nuevas tecnologías. Se ha desarrollado una versión de DESK en Java que permite su utilización vía web. Incluye un asistente para facilitar el proceso de creación del modelo. JDESK tiene exactamente la misma funcionalidad de DESK. Aporta, respecto a DESK, todas las ventajas de la simulación en web respecto a la simulación convencional. Aunque, como desventaja, los costes de simulación de JDESK son superiores a los de DESK.

GDESK es la adaptación del simulador de eventos discreto DESK a núcleo de aplicaciones gráficas en tiempo real. GDESK es un núcleo independiente de la aplicación en la que se integra. Pero no puede funcionar si no se integra en una aplicación gráfica. GDESK conserva toda la potencia y funcionalidad de DESK. Es un núcleo flexible, permite simular cualquier sistema y no impone restricciones. Las estructuras básicas de GDESK son la mismas que DESK, pero adaptadas a la nueva situación. La principal diferencia entre DESK y GDESK es la gestión de tiempos. En GDESK el tiempo de simulación corresponde, mientras el sistema no esté colapsado, al tiempo real del sistema.

GDESK discretiza la aplicación gráfica donde se integra. La simulación discreta permite modelar comportamientos discretos, continuos e híbridos. Una consecuencia de la discretización del sistema es el desacoplo de las fases de visualización y simulación. La aplicación donde se integra GDESK debe utilizar el mecanismo de paso de mensajes para modelar el comportamiento de los objetos.

GDESK gestiona los eventos del sistema haciendo que los objetos de la aplicación gráfica donde se integra se comuniquen mediante paso de mensajes. GDESK controla el proceso de envío y recepción de mensajes. Los mensajes tiene asociado un tiempo que indica el instante en que deben ser recibidos por el objeto receptor. GDESK captura los mensajes enviados y los almacena hasta que se cumple el tiempo del mensaje. En ese instante envía el mensaje al objeto receptor. GDESK sólo trata con mensajes, no realiza los procesos que el objeto receptor tenga asociados al mensaje.

GDESK se integra en una aplicación gráfica: el núcleo de aplicaciones gráficas Fly3D, obteniéndose DFly3D, un motor de aplicaciones gráficas en tiempo real discreto desacoplado.

Una aplicación en DFly3D es un conjunto de objetos que se comunican mediante paso de mensajes. Son objetos, tanto los elementos del sistema (como la consola o el objeto visualizador), como los objetos creados en la programación de la aplicación (como el avatar, pistolas o muros). El simulador GDESK controla el proceso de envío y recepción de mensajes, controlando que los mensajes sean recibidos por el objeto destino en el instante fijado por el objeto receptor y que los mensajes se ejecuten ordenados por tiempo de ocurrencia. No se muestrean los objetos, sino que los objetos sólo actúan como consecuencia de la llegada de un mensaje. El mecanismo de paso de mensajes modela tanto comportamientos continuos como comportamientos discretos de los objetos. Este mecanismo tiene dos utilidades:

- **Comunicar objetos**: cuando un objeto desea comunicarse con otro objeto, por ejemplo para que objeto modifique su comportamiento, le envía un mensaje con los parámetros adecuados.
- **Modelar el comportamiento de un objeto**:
  - **Comportamiento continuo**: el comportamiento continuo del objeto debe actualizarse cada cierto intervalo de tiempo. Para ello, el objeto se envía un mensaje a si mismo para modificar su comportamiento transcurridas T unidades de tiempo.
  - **Comportamiento discreto**: los mensajes permiten también modelar comportamientos discretos. Si se desea programar un evento que suceda en T unidades de tiempo. Un objeto envía un mensaje a otro objeto o a si mismo con tiempo T. Como consecuencia del mensaje, el objeto lleva a cabo un determinado proceso (a diferencia del comportamiento continuo, el objeto no se envía ningún mensaje a si mismo para continuar con el proceso).
  - **Comportamiento híbrido**: sería una combinación de los dos procesos anteriores.

El comportamiento del objeto se define en la función de recepción de mensajes. Cada objeto tiene asociada una función de recepción de mensajes. En esta función se define la respuesta del objeto a un tipo de mensaje determinado (con unos parámetros determinados, o de un objeto emisor determinado). En esta función se define también a que mensajes es sensible un objeto. Como respuesta a un mensaje, el objeto puede generar nuevos mensajes.

El proceso de visualización requiere de un objeto específico que lo controle y su comportamiento se modela mediante el mecanismo de paso de mensajes también.

Por tanto, por cada mensaje que recibe el objeto visualizador se genera una escena. De este modo se controla la frecuencia de cuadro del sistema. La visualización es un proceso más del sistema, con la misma prioridad que el resto de procesos (los eventos o mensajes se ejecutan ordenados por tiempo de ocurrencia). La frecuencia de cuadro la define el usuario (mientras el sistema no esté colapsado). La frecuencia de cuadro no tiene porqué ser constante durante la ejecución de la aplicación, puede adaptarse dinámicamente a los requerimientos de la aplicación.

GDESK dota a DFly3D con la posibilidad de monitorizar el sistema de dos formas diferentes:

- **Monitorización fuera de línea**: se generan una serie históricos con los resultados de la monitorización del sistema, útil en el proceso de producción de la aplicación gráfica.
- **Monitorización en línea**: define mecanismos para que los objetos obtengan información del sistema y puedan ajustar dinámicamente su comportamiento.

La monitorización no es proceso exclusivo de los sistemas discretos, pero el mecanismo de paso de mensajes y la posibilidad de definir diferentes frecuencias de muestreo para cada aspecto de cada objeto facilitan mucho este proceso. La monitorización es un proceso discreto que define su propia QoS y, por tanto, el muestreo óptimo para cada uno de sus procesos, evitando sobrecargar el sistema innecesariamente. Fly3D no incluye procesos de monitorización.

Se dispone de un sistema continuo acoplado (Fly3D) y el mismo sistema discreto desacoplado (DFLy3D). Los procesos como mecanismo de visualización, detección de colisiones,... son comunes a ambos proceso. Por ello, se pueden comparar los dos sistemas, pues únicamente difieren en el mecanismo de simulación. Se han obtenido resultados numéricos de la comparación de ambos sistemas, pero las conclusiones obtenidas de estos resultados son extrapolables a los sistemas continuos acoplados y discretos desacoplados en general.

Resultado de la comparación:

1. **Distribución de tiempo**: el sistema continuo constantemente simula y visualiza a la máxima velocidad que la potencia de cálculo es capaz de proporcionar. Consume todos los recursos disponibles. Sin embarco, el sistema discreto consume únicamente el tiempo que el programador ha destinado al proceso de visualización y a la simulación de cada objeto. El resto del tiempo lo libera. El sistema discreto es capaz de ejecutar la misma simulación con un consumo menor de recursos.
2. **QoS de los Objetos**: el sistema continuo muestrea todos los objetos del sistema con la misma frecuencia, independientemente de la QoS de cada objeto.

Esta frecuencia es igual a la frecuencia de cuadro. La frecuencia de muestreo del sistema puede tener un valor tal que haya objetos que se sobremuestreen y otros que se submuestreen. No es posible definir una frecuencia diferente para cada objeto. El sistema discreto permite definir una frecuencia de muestreo diferente para cada objeto, incluso una frecuencia diferente para cada aspecto de cada objeto. Es capaz de adaptarse a las necesidades de QoS de cada objeto y de cada aspecto de cada objeto. Los objetos siempre se muestrean a la frecuencia óptima. Por tanto, el sistema discreto permite cumplir la QoS definida para cada objeto.

3. **Reparto de la Potencia de Cálculo**: el sistema continuo no tiene en cuenta las necesidades de cada objeto a la hora de repartir la potencia de cálculo, pues la frecuencia de muestreo es común a todo el sistema. El sistema discreto optimiza el reparto de la potencia de cálculo, al permitir al programador definir la QoS de cada objeto.

4. **Acople de Procesos**:

   - En el sistema continuo todos los procesos están acoplados. En cada iteración del bucle principal de la aplicación se simulan todos los objetos recorriendo el grafo de escena y después se visualiza la escena actual. En el sistema discreto cada objeto es independiente del resto e independiente del proceso de visualización. Todos los objetos están desacoplados. El comportamiento de cada objeto está desligado del resto.
   - La frecuencia de cuadro depende de la carga del sistema en el sistema continuo y está integrada con el resto de procesos (igual frecuencia de muestreo). En el sistema discreto, el proceso de visualización está desacoplado del proceso de simulación del resto de objetos (es independiente) y tiene una frecuencia diferente.

5. **Colapso del Sistema**: cuando el sistema no es capaz de simular y visualizar correctamente en un ciclo de refresco de pantalla se dice que el sistema está colapsado. El colapso del sistema tiene efectos diferentes en el sistema continuo y en el discreto. El sistema continuo colapsado tiene un funcionamiento incorrecto (colisiones no detectadas, trayectorias incorrectas,...). El comportamiento incorrecto se acentúa cuando aumenta el factor de colapso. El sistema discreto en cambio simula el sistema correctamente independientemente del colapso del sistema. Para conseguir una simulación correcta sacrifica la ejecución del sistema en tiempo real. El sistema discreto colapsado consume un tiempo real en ejecutar la simulación mayor que el tiempo de simulación del sistema. La diferencia entre estos dos tiempos será mayor cuanto mayor sea el factor de colapso. Ante el colapso del sistema, el sistema discreto degrada la QoS de los objetos, pero el sistema discreto la mantiene.

6. **Monitorización del Sistema**: el sistema discreto incluye un monitor del sistema que permite monitorizar las aplicaciones en línea y fuera de línea. El monitor del sistema es capaz de comunicarse con el resto de objetos del sistema para que adapten su comportamiento si se produce alguna condición crítica definida por el programador. También pone a disposición de los objetos información crítica del comportamiento del sistema por si los propios objetos desean usarla para definir adecuadamente su QoS y adaptarla dinámicamente. La monitorización fuera de línea produce una serie de información e históricos sobre la ejecución del sistema.

7. **Adaptación Dinámica del Sistema**: el sistema discreto es capaz de tomar información del proceso de monitorización y adaptarse dinámicamente. La QoS de cada objeto puede redefinirse dinámicamente para adaptarse a las condiciones del sistema. Si el sistema está colapsado, uno o más objetos pueden reducir su QoS, para evitar el colapso. Si el sistema libera tiempo (hay tiempo remanente en su ejecución), los objetos pueden readaptar su QoS, para aprovechar este tiempo libre en mejorar su inteligencia artificial, detección de colisiones,... En el sistema discreto los objetos definen un rango de QoS por si el sistema debe adaptarse ($[QoS_{min}..QoS_{max}]$). La adaptación del sistema puede implicar que algunos objetos se degraden en beneficio de otros. La gestión de eventos discreta mediante paso de mensajes hace que este proceso no sólo sea posible llevarlo a cabo, sino que se realice de una forma sencilla de programar y con una sobrecarga del sistema muy baja (aunque muy dependiente del uso que el programador de la aplicación haga de ella).

El sistema discreto permite:

- Desacoplar el comportamiento de todos los objetos del sistema, no sólo la fase de simulación de la fase de visualización, sino todas las simulaciones de cada objeto, incluso todas las simulaciones de cada aspecto de cada objeto.
- Un reparto óptimo de los recursos del sistema entre los objetos.
- No se establecen prioridades de objetos. La prioridad de un objeto durante la simulación depende de como se modela el comportamiento del objeto. Los eventos se ejecutan ordenados en el tiempo y no hay prioridades implícitas entre ellos debidas a su situación en el grafo de escena.
- El sistema es sensible a tiempos menores que la frecuencia de muestreo de un sistema continuo. Los eventos se ejecutan ordenados por tiempo. Los eventos ocurren en el instante exacto para el que están planificados. No se sincronizan artificialmente con el periodo de muestreo del sistema, como ocurre en los sistemas continuos.

- Una utilización más racional de la potencia de cálculo por parte de cada objeto del sistema, consumiendo únicamente la potencia necesaria. Sólo los objetos que cambian su estado producen eventos y consumen potencia de cálculo. Los objetos no se sobremuestrean o submuestrean. Diferentes aspectos del objeto pueden tener diferentes frecuencias de muestreo. Estas frecuencias son gestionadas directamente por el programador.

- Definir la QoS de cada objeto, independientemente del resto. Permite definir el rango de QoS que puede llegar a asumir el objeto si hay colapso en el sistema o si otros objetos necesitan alcanzar su QoS máxima.

- Adaptar dinámicamente la QoS de cada uno de los objetos del sistema. La QoS de un objeto puede ser constante durante la ejecución o puede variar dinámicamente para adaptarse a las necesidades del objeto o del sistema en general. El objeto *visualizador* define también sus criterios de QoS y puede adaptarse dinámicamente. Por tanto, la frecuencia de cuadro es configurable por el programador.

- Ejecutar la simulación del sistema correctamente independientemente del colapso del sistema. El paradigma discreto permite la independencia del número de escenas generadas y la carga del sistema.

- Monitorizar el sistema de formas diferentes, dependiendo de las necesidades de cada fase de producción de la aplicación gráfica. El objeto *monitor* provee al resto de los objetos del sistema con información que pueden utilizar. Puede comunicarse con ellos para que adapten su comportamiento si es necesario.

- El esquema de simulación discreta soporta simulación continua e híbrida de forma natural. El esquema de simulación continua soporta simulación discreta pero a costa de una programación compleja y un coste temporal adicional.

- El sistema discreto sólo trata de simular siguiendo el tiempo real si el sistema no está colapsado. Si está colapsado, no es capaz de simular en tiempo real. Por lo que, en vez de degradar el sistema, simula más lentamente. El tiempo real y el tiempo del sistema se desfasan. Prima la QoS del sistema sobre la velocidad de simulación.

El sistema discreto aprovecha el tiempo de forma más eficiente que el sistema continuo, utilizando únicamente la potencia de cálculo necesaria para simular y visualizar con la frecuencia definida por el programador para cada objeto. La potencia no consumida se libera para utilizarse en otras aplicaciones o para mejorar aspectos del sistema. Si el sistema se colapsa no libera tiempo. Al aprovecha mejor la potencia de cálculo, permite ejecutar la aplicación en máquinas con menos potencia de cálculo

y distribuirla mejor entre los objetos del sistema. El sistema no depende, mientras no se colapse, de la potencia de cálculo de la máquina.

El nuevo paradigma puede utilizarse en cualquier tipo de aplicación gráfica y se puede compatibilizar con otras líneas de investigación, como las dedicadas a disminuir los costes de la detección de colisiones de objetos o de la generación de escenas. Incluso los resultados pueden extrapolarse a otras aplicaciones gráficas independientemente de que sean en tiempo real o no, pues si los eventos del sistema se definen con tiempo 0, se ejecutan sin sincronizarse con el tiempo real.

6.2. Líneas Futuras de Investigación

6.2.1. Sistema Adaptable

El sistema discreto deja libertad al programador para definir el comportamiento de los objetos para que se adapte a aspectos como: potencia de cálculo de la máquina, factor de colapso o necesidades de potencia de cálculo puntuales del propio objeto o de otros. La posibilidad de adaptar el sistema y ciertos mecanismos para hacerlo se incluyen en el sistema discreto creado (DFly3D), pero no se ha profundizado en la forma más óptima de adaptar el sistema.

Una posible línea de investigación es comprobar los métodos más óptimos para llevar a cabo esta adaptación, sobre todo lo que se refiere al objeto visualizador. El sistema puede adaptarse también al dispositivo de salida dinámicamente.

6.2.2. Instante de generación de los Eventos de Visualización: Predicción del Coste de Simulación

El objeto visualizador decide el instante exacto de generación del evento de visualización. Los objetivos de este objeto deben ser:

- Generar sólo una visualización por refresco de pantalla (evitando generar escenas que nunca se visualizarán por pantalla).
- El momento exacto de la generación de la visualización debe permitir generar la escena antes del siguiente refresco de pantalla

Para conseguir estos objetivos es necesario predecir el coste de generación de la siguiente escena. Existen estudios de predicción de visualización como [Wimmer:2003] que pueden integrarse en DFly3D.

6.2.3. Núcleo de Aplicaciones Gráficas en Tiempo Real

DFly3D es el núcleo de aplicaciones gráficas Fly3D modificado para que soporte eventos discretos y desacoplado. Los procesos de simulación y visualización son los definidos por Fly3D. El objetivo de utilizar Fly3D es poder comparar el mismo sistema con los dos paradigmas de simulación. Pero, una vez obtenidos los resultados y probadas las ventajas de la utilización de GDESK, el siguiente paso es crear un núcleo de aplicaciones gráficas en tiempo real discreto desacoplado completo. Para ello se puede utilizar librerías gráficas como SimGear [Simgear] u OSG [Osg].

6.2.4. Mundos Autónomos

La independencia de las fases de simulación y visualización permite anular la fase de visualización hasta que, por ejemplo, se produzca un determinado evento en el sistema. Por ello, una posible aplicación de este sistema es la evolución de mundos sin visualización, que en un momento determinado comienzan a visualizarse o que se obtiene su estado. En Fly3D no se anima el mundo si no hay cámara, aquí puede simularse sin visualizaciones y el mundo evoluciona de forma autónoma.

6.3. Publicaciones

Las publicaciones ha las que ha dado lugar la tesis son:

- I. García, R. Mollá, *Simulación Desacoplada de Eventos Discretos en Videojuegos*, Proceedings de GEIG'2004.
- I. García, R. Mollá, *Making Discrete Games*, Computational Science and its Applications - ICCSA'2004 LNCS v.3045.
- I. García, R. Mollá, E. Camahort, *Introducing Discrete Simulation into Games*, European Research Consortium for Informatics and Mathematics. ERCIM News, Abril v.57. 2004.
- I. García, R. Mollá, P. Morillo, *From Continuous to Discrete Games*, Proceedings de 2004 Computer Graphics International (CGI'04), IEEE Computer Society Press. 2004.
- I. García, R. Mollá, A. Barella, *GDESK: Game Discrete Event Simulation Kernel*, Journal of WSCG. Roberto Scopigno, Vaclav Skala (eds). 2004.
- I. García, R. Mollá, J. Cabanillas, *JDESK Web-Based Discrete Event Simulation Kernel*, Menu Conference'2003.
- I. García, R. Mollá, *JDESK: Simulador de Eventos Discreto Basado en Web*, Jenui'2003.

- I. García, R. Mollá, *Discrete Events Simulation as a Computer Game Kernel*, Grupo Portugués de Computación Gráfica. Virtual Journal. 2002.
- I. García, R. Mollá, E. Ramos, M. Fernández, *D.E.S.K.: Discrete Events Simulation Kernel*, ECCOMAS'2000.
- I. García, R. Mollá, *Decoupled Discrete Event Simulation Kernel for Videogames*, Computer Graphics Forum, pendiente de aceptación (proceso de segunda revisión).

Apéndices

Apéndice A

Modelado de Sistemas que Cambian Dinámicamente Utilizando DESK

En este apéndice se muestran tres ejemplos de sistemas modelados usando DESK que cambian su topología y/o comportamiento dinámicamente.

A.1. Sistema de Computadoras

Este ejemplo muestra como crear ES dinámicamente en DESK y como modificar el resto del sistema para integrar la nueva ES. El ejemplo muestra dos formas diferentes de implementar el cambio en la topología del sistema: usando muestreo y redefiniendo funciones de comportamiento.

El sistema de computadoras contiene una fuente de clientes y dos CPU. Los clientes generados por la fuente transitan a la *CPU1* y después de recibir servicio en ésta, transitan a la *CPU2*. Cuando abandonan la *CPU2*, los clientes tiene un 45 % de posibilidades de dejar el sistema y un 55 % de volver a recibir servicio en la *CPU1*. Los clientes requieren un tiempo de servicio de la *CPU2* con una distribución exponencial con una tasa de 1.2 unidades de tiempo. La fuente crea clientes con una tasa de 1 unidad de tiempo. Por tanto, la longitud media de la cola de espera de la *CPU2* es alta. La *CPU2* es el cuello de botella del sistema.

Para solucionar el problema, el sistema cambia su topología dinámicamente cuando se detecta la situación crítica. Una forma de reducir la longitud media de la cola de espera de la *CPU2* es tener una tercera CPU (*CPU3*). Si la longitud media de la cola de espera de la *CPU2* alcanza un determinado valor (condición crítica), se crea dinámicamente la *CPU3* y se redireccionan el 90 % de los clientes a la *CPU3*. Como

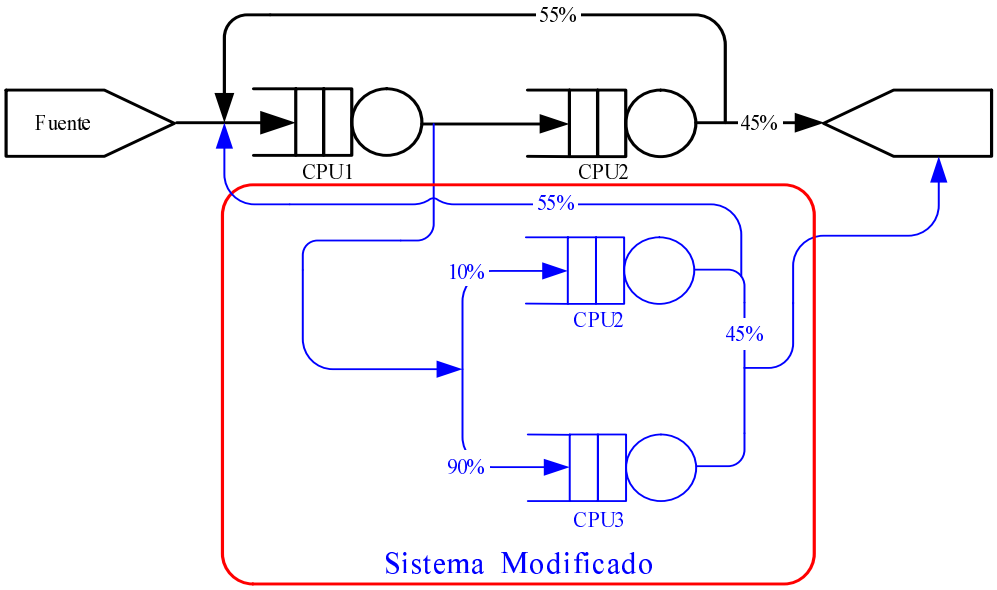

Figura A.1: Sistema de computadoras (muestreo)

consecuencia de la creación de la *CPU3*, se modifica el tránsito de clientes desde la *CPU1*. Mientras la condición crítica no se produce el sistema tiene la topología inicial. A partir del momento en que se cumple la condición crítica el sistema cambia su topología y ya no retorna a su topología inicial.

A.1.1. Implementación por Muestreo

Las funciones de comportamiento implicadas en el cambio de topología son:

- Función de encaminamiento de la *CPU1*. En esta función se muestrea continuamente la condición crítica. Dependiendo de esta condición se decide cual es la ES destino. Si el cambio de topología debe realizarse, la ES destino es la *CPU2* o la *CPU3*, dependiendo de un valor aleatorio.
- Función de usuario de la *CPU2*. Esta función se usa en el ejemplo para comprobar si se ha producido el cuello de botella (condición crítica). Se ha elegido esta función para testear la condición de cambio del sistema porque se ejecuta cada vez que un cliente accede a la *CPU2*. Esta función siempre testea la condición critica, independientemente de que el cambio de topología se haya realizado o no. Si la topología debe cambiar, crea dinámicamente la *CPU3*.

La figura A.1 muestra el sistema inicial y modificado. El siguiente código muestra la implementación del sistema.

```
// Funciones de tiempo de servicio
DK_T_system_time t_source (DK_CLIENT *c) {return (expntl(1));}
DK_T_system_time t_cpu1 (DK_CLIENT *c) {return (expntl(1.2));}
DK_T_system_time t_cpu2 (DK_CLIENT *c) {return (expntl(1.5));}
DK_T_system_time t_cpu3 (DK_CLIENT *c) {return (expntl(0.75));}

// Funciones de encaminamiento
DK_STATION *trans_source (DK_CLIENT *c)
    { return cpu1;}
DK_STATION *trans_cpu1 (DK_CLIENT *c)
    { if (new_topology)
        if (random(1,100)<=10) return cpu2;
        else return cpu3;
      else return cpu2;}
DK_STATION *trans_cpu2 (DK_CLIENT *c)
    { if (random(1,100)<=55)
        return cpu1;
      else return NULL;}
DK_STATION *trans_cpu3 (DK_CLIENT *c)
    { return NULL;}

// Funciones de usuario
void user_cpu2 (DK_STATION *es, DK_CLIENT *c)
    {if (!new_topology && DK_Average_Queue_Length (cpu2)>10)
        {new_topology=true;
         cpu3 = DK_Server_Station ("Cpu 3", 1, DK_NO_EXPULSIVE,
                t_cpu3, SS_Fifo, NULL, trans_cpu3, NULL) }}

void main (void)
{
DK_Init();
source = DK_Source_Station ("Source of Clients", t_source, trans_source);
cpu1   = DK_Service_Station ("Cpu1", 1, DK_NO_EXPULSIVE, t_cpu1,
                             DK_Fifo, NULL, trans_cpu1, NULL);
cpu2   = DK_Service_Station ("Cpu2", 1, DK_NO_EXPULSIVE, t_cpu2,
                             DK_Fifo, NULL, trans_cpu2, user_cpu2);
if (!DK_Simul ("Example 1", 0, 1000000)) DK_Error();
else DK_Results (); }
```

A.1.2. Implementación por Cambio de Funciones de Comportamiento

Esta implementación usa redefinición de funciones de comportamiento y una ES de control. Los pasos seguidos son:

1. La función de usuario de la *CPU2* testea continuamente la condición crítica.
2. Cuando se detecta la condición crítica, se crea un cliente y se envía (transita) a la ES de control.
3. Cuando el cliente llega a la ES de control:
   a) Crea la *CPU3* dinámicamente.
   b) Cambia la función de encaminamiento de la *CPU1*. La nueva función de encaminamiento distribuye los clientes entre la *CPU2* y la *CPU3*. Esta nueva función de encaminamiento no necesita muestrear si se ha producido el cambio topológico o no.
   c) Cambia la función de usuario de la *CPU2* (función que muestrea la situación crítica) cambiándola por una función nula. De esta forma se evita que se compruebe la condición crítica una vez se ha alcanzado.

La figura A.2 muestra el sistema inicial y final. El siguiente código muestra la implementación de este ejemplo.

```
// Funciones de tiempo de servicio
DK_T_system_time t_source (DK_CLIENT *c) {return (expntl(1));}
DK_T_system_time t_cpu1 (DK_CLIENT *c) {return (expntl(1.2));}
DK_T_system_time t_cpu2 (DK_CLIENT *c) {return (expntl(1.5));}
DK_T_system_time t_cpu3 (DK_CLIENT *c) {return (expntl(0.75));}
DK_T_system_time t_control (DK_CLIENT *c){return (DK_Cero());}

// Funciones de encaminamiento
DK_STATION *trans_source (DK_CLIENT *c){return cpu1;}
DK_STATION *trans_cpu1_1 (DK_CLIENT *c){return cpu2;}
DK_STATION *trans_cpu1_2 (DK_CLIENT *c)
    {if (random(1,100)<=10)
        return cpu2;
     else return cpu3;}
DK_STATION *trans_cpu2 (DK_CLIENT *c)
    {if (random(1,100)<=55)
        return cpu1;
```

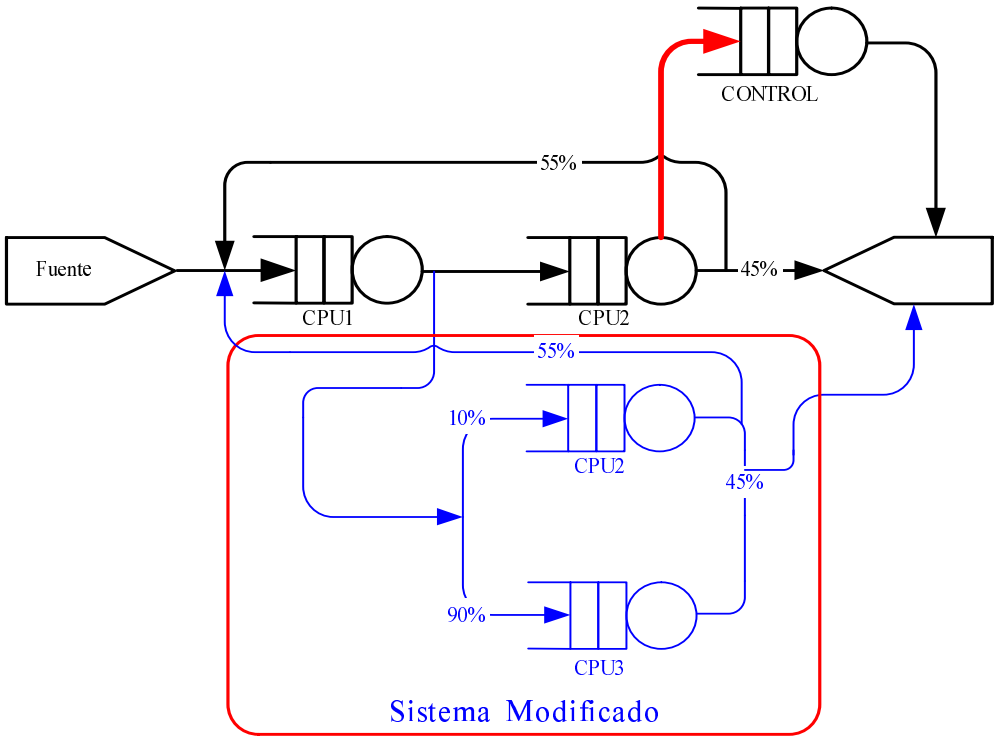

Figura A.2: Sistema de computadoras (cambio de las funciones de comportamiento)

```
     else return NULL;}
DK_STATION *trans_cpu3 (DK_CLIENT *c){return NULL;}

// Funciones de usuario
void user_control (DK_STATION *es, DK_CLIENT *c)
    {// Crea cpu3
    cpu3 = DK_Server_Station("Cpu 3", 1, DK_NO_EXPULSIVE, t_cpu3,
            DK_Fifo, NULL, trans_cpu3, NULL);
    // Cambia las funciones de encaminamiento y de usuario
    DK_ChangeF_Routing(cpu1,trans_cpu1_2);
    DK_ChangeF_User(cpu2,NULL);}
void user_cpu2 (DK_STATION *es, DK_CLIENT *c)
    {if (DK_Average_Queue_Length(cpu2)>10)
        DK_Insert_Client(0,0,0,control);}

void main (void)
{
DK_Init();

source = DK_Source_Station("Source of Clients", t_source,
trans_source);
cpu1   = DK_Service_Station("Cpu 1", 1, DK_NO_EXPULSIVE, t_cpu1,
                            DK_Fifo, NULL, trans_cpu1_1, NULL);
cpu2   = DK_Service_Station("Cpu 2", 1, DK_NO_EXPULSIVE, t_cpu2,
                            DK_Fifo, NULL, trans_cpu2, user_cpu2);
control= DK_Service_Station("Control Service Station", 1,
                            DK_NO_EXPULSIVE, t_control, DK_Fifo, NULL,
                            DK_Transition_Out, user_control);
if (!DK_Simul("Example 1", 0, 1000000)) DK_Error();
else DK_Results(); }
```

Cambiar las funciones de comportamiento para variar el sistema dinámicamente reduce el coste temporal de la simulación entre un 10 % y un 5 % respecto a la opción con muestreo. La variación depende mucho del número de ES implicadas en el cambio dinámico y el número de condiciones a muestrear.

A.2. Cajas de Cobro en un Supermercado

Este ejemplo muestra el uso de vectores de punteros de ES como una forma de automatizar el proceso de cambio dinámico del sistema.

En un supermercado (figura A.3) hay cinco cajas de cobro. Cada caja corresponde

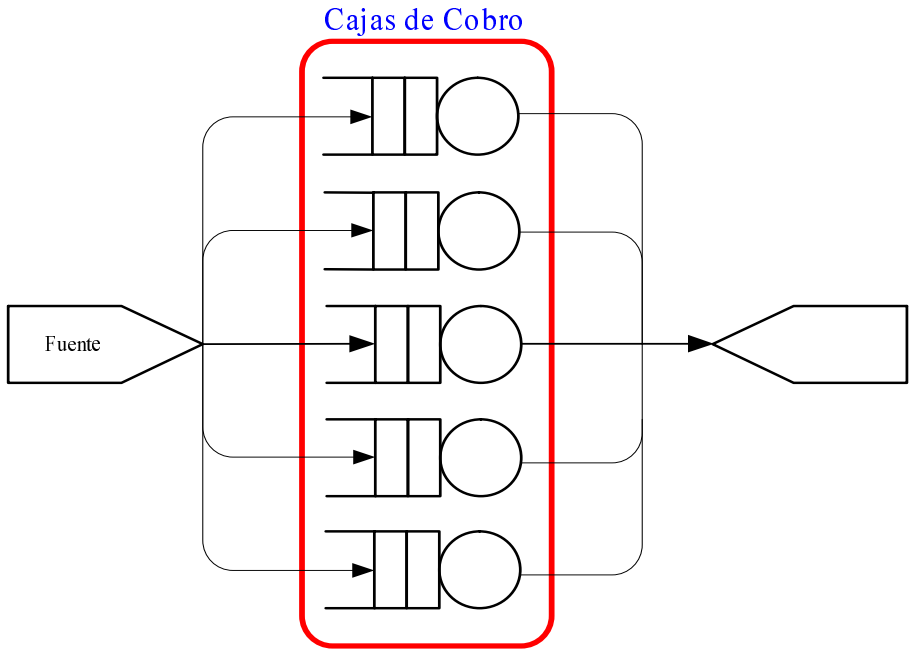

Figura A.3: Cajas de cobro en un supermercado

a una posición de un vector de ES. Todas las cajas tienen el mismo comportamiento. Inicialmente sólo hay una caja abierta (sólo hay una ES en el sistema). Cuando una caja alcanza un número determinado de clientes esperando para pagar, se abre otra caja (se crea una ES dinámicamente).

La fuente se usa como ES para controlar la simulación. En este caso la función de ES de control la asume una ES existente. Controla la creación de clientes. Cada vez que se crea un cliente, la última ES creada se testea para comprobar si ha alcanzado la condición crítica.

El siguiente código muestra la implementación de este ejemplo.

```
// Funciones de tiempo de servicio
DK_T_system_time t_source (DK_CLIENT *c)
    {if (DK_Average_Queue_Length(cashier[opened])>10)
        cashier[++opened] = DK_Server_Station("Cashier", 1,
                            DK_NO_EXPULSIVE, t_cashier, DK_Fifo,
                            NULL, trans_cashier, NULL);
     return (expntl(1));}
DK_T_system_time t_cashier (DK_CLIENT *c)
    {return (expntl(4.2));}
```

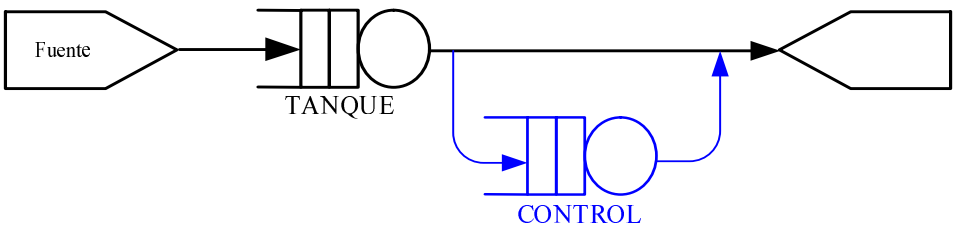

Figura A.4: Tanque de agua

```
// Funciones de encaminamiento
DK_STATION *trans_source (DK_CLIENT *c)
    {int rn, dest=0;
    float increment, limit;
    increment = 100/(float)(opened+1);
    limit = increment;
    rn = random(1,100);
    while (limit<=100)
        {if (rn<=limit) return cashier[dest++];
        limit+=increment;}
    return NULL;}
DK_STATION *trans_cashier (DK_CLIENT *c) {return NULL;}


void main (void)
{
DK_Init();
source = DK_Source_Station("Source of Clients", t_source,
         trans_source);
cashier[opened] = DK_Service_Station("Cashier", 1,
                  DK_NO_EXPULSIVE,t_cashier, DK_Fifo, NULL,
                  trans_cashier, NULL);
if (|DK_Simul("Supermarket", 0, 1000000.0)) DK_Error();
else DK_Results();}
```

A.3. Tanque de Agua

Este ejemplo muestra el uso de una ES de control para cambiar la topología del sistema (figura A.4).

Se tiene un tanque de agua alimentado por un grifo (fuente de clientes). Cada

cliente modela un número de litros de agua determinados. Cuando el volumen de agua que espera entrar en el tanque alcanza un valor tope (condición crítica), el tanque se vacía durante 100 segundos. Después de este tiempo, el agua vuelve a entrar en el tanque normalmente. Cuando el tanque detecta la condición crítica, envía un cliente a la ES de control. Cuando el cliente llega a la ES de control:

1. Cambia la función de tiempo de servicio de la fuente. El ratio de creación de clientes será ahora de 100 segundos. Por tanto no se crean clientes durante el tiempo en que el tanque se está vaciando.
2. Cambia la función de tránsito del tanque. Todos los clientes dejan el sistema después de salir del tanque. Este cambio evita muestrear la condición crítica hasta que el sistema vuelva a comportarse con normalidad.
3. El tiempo de servicio de los clientes en la ES de control es de 100 segundos. De esta forma, cuando el cliente de esta ES finaliza su servicio se indica que el sistema debe volver a la situación inicial.

Cuando transcurren 100 segundos, el sistema vuelve a la normalidad. Cuando el cliente que está recibiendo servicio en la ES de control finaliza su tiempo de servicio (transcurridos 100 segundos) y abandona la ES de control:

1. Cambia la función de tiempo de servicio de la fuente a la función inicial para que genere clientes con el ratio inicial.
2. Cambia la función de encaminamiento del tanque para que vuelva a testear la condición inicial.
3. El cliente deja el sistema.

```
// Funciones de tiempo de servicio
DK_T_system_time t_source (DK_CLIENT *c) {return (expntl(1));}
DK_T_system_time t_tank (DK_CLIENT *c)   {return (expntl(1));}
DK_T_system_time t_control (DK_CLIENT *c){return (expntl(100));}

// Funciones de encaminamiento
DK_STATION *trans_source (DK_CLIENT *c)
    {return tank;}
DK_STATION *trans_tank (DK_CLIENT *c)
    {if (DK_Average_Queue_Length(tank)>10)
        return control;
     else return NULL;}
DK_STATION *trans_control (DK_CLIENT *c)
```

```
    {DK_ChangeF_Service_Time(source,t_control);
     DK_ChangeF_Routing(tank,trans_tank);
     return NULL;}
// Funciones de usuario
void user_control (DK_STATION *es, DK_CLIENT *c)
    {DK_ChangeF_Service_Time(source,t_source);
     DK_ChangeF_Routing(tank,DK_Transition_Out);}

void main (void)
{
DK_Init();
source  = DK_Source_Station("Source of Clients", t_source,
          trans_source);
tank    = DK_Service_Station("Water Tank", 1, DK_NO_EXPULSIVE,
          t_tank, DK_Fifo, NULL, trans_tank, NULL);
control = DK_Service_Station("Control", 1, DK_NO_EXPULSIVE,
          t_control, DK_Fifo, NULL, trans_control, user_control);
if (!DK_Simul("Water Tank", 0, 1000000.0)) DK_Error(); else
DK_Results();}
```

Apéndice B

Utilización de JDESK

JDESK es un simulador basado en texto, por lo que el modelo simulado es código escrito en Java con funciones de biblioteca específicas del simulador. El código de simulación debe contener las funciones de comportamiento de las ES y la definición del modelo.

La página principal del simulador [Jdesk] contiene una ventana de edición, una ventana de mensajes y un conjunto de controles. En el editor de texto se puede escribir el código del modelo de simulación. Permite editar modelos creados anteriormente, usar ejemplos de modelos completos del repositorio de JDESK, funciones de comportamiento predefinidas o copiar y pegar código de otras aplicaciones. El código debe tener el formato de un programa principal en Java.

Para usuarios inexpertos es aconsejable usar el asistente (figuras B.1 y B.2). El asistente es un Applet [Cowell:2000] de Java que permite crear modelos de forma rápida y sencilla mediante controles, escribiendo únicamente algunas funciones de comportamiento. El asistente guía al usuario en el proceso de definición del modelo. Permite construir el código Java del modelo de simulación a partir de las especificaciones del usuario. El código producido por el asistente puede ser editado.

El asistente permite:

- Definir los parámetros generales de la simulación, como tiempos de inicio y final de la simulación.
- Añadir funciones al modelo. El asistente permite ver ejemplos de funciones de la biblioteca de JDESK. Si el usuario elige escribir directamente la función, aparece una ventana de edición con un esqueleto de la función (en forma de método de la clase correspondiente), de forma que el usuario sólo debe escribir el código estrictamente necesario. Esta función se podrá asignar posteriormente a una ES.

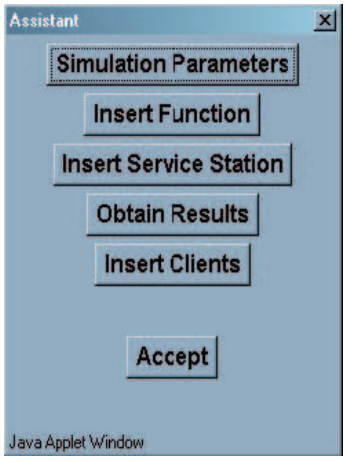

Figura B.1: Asistente de JDESK

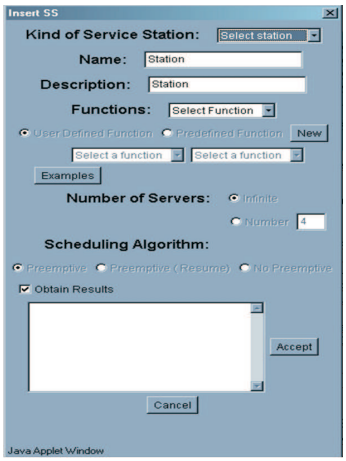

Figura B.2: Ventana de inserción de estaciones de servicio en JDESK

- La inserción de una ES en el modelo simulado utilizando el asistente se realiza utilizando los controles de la ventana de inserción de ES (figura B.2). El asistente permite seleccionar el tipo de ES y según el tipo seleccionado habilita únicamente los controles correspondientes. Permite definir las funciones de comportamiento de la ES: seleccionándolas entre las previamente definidas, para ésta o para otras ES, o entre las funciones de biblioteca o bien escribir la función directamente. Si se escribe la función muestra un esqueleto de la clase correspondiente.

- Insertar clientes: permite crear clientes con unas determinadas características e insertarlos en una ES previamente definida.

- Obtener resultados: permite indicar que resultados de la simulación se desea obtener y en que formato.

El código creado mediante el asistente aparece en el editor de texto y puede ser modificado para ajustar o modificar su comportamiento.

Una vez se ha definido el modelo, se compila utilizando para ello un control de JDESK. Si la compilación produce errores se mostrarán los mensajes correspondientes en la ventana de mensajes. Si la compilación tiene éxito, el fichero conteniendo el modelo de simulación queda disponible para que el usuario lo ejecute de forma local. El fichero de la simulación se compila de forma local para adaptarlo a la plataforma correspondiente.

El compilador de JDESK se ejecuta en un servidor y se puede acceder a este mediante la página HTML [Musciano:2002] de JDESK. Por ello, un usuario que desee utilizar JDESK debe tener instalado un navegador de internet y una máquina virtual de Java (Java Virtual Machine [Lindholm:1996] [Venners:1999]). El navegador debe permitir la ejecución de Applets de Java.

Los pasos a seguir para crear un modelo con JDESK son:

1. Acceder a la página de JDESK utilizando un navegador. En el momento del acceso a la página de JDESK, si el ordenador local desde el que se accede no tiene una máquina virtual Java instalada, se descarga automáticamente.

2. Escribir el modelo de simulación en la ventana de edición o utilizar el asistente. El Applet de Java para la edición se ejecuta de forma local, por lo que no es necesario mantener la conexión a internet durante este proceso.

3. Compilar el modelo de simulación, utilizando el control de construcción del modelo. El resultado de la compilación se muestra en la ventana de mensajes. Durante este proceso es necesario estar conectado a internet. Si el modelo de

simulación no es correcto se mostrarán los mensajes de error correspondientes. El usuario debe, en este caso, modificar el modelo y volver a compilarlo.

4. Obtener los ficheros de simulación.

5. Compilar la simulación de forma local y ejecutarla.

6. Obtener resultados de la simulación.

El algoritmo C.3 (apéndice C) muestra el ejemplo 5.9 implementado en Java para JDESK.

Apéndice C

Algoritmos Ejemplo del Simulador de Eventos Discreto

El presente capítulo muestra la implementación del sistema de la figura en DESK, SMPL y JDESK, con el objetivo de comparar las diferentes formas de implementación de cada modelo en los tres sistemas.

C.1. Implementación del sistema cerrado en DESK

```
#define tcpu1 2
#define tcpu2 1
#define tcpu3 1
#define tproc 1
#define tmax  1000000
#define nt    10

// Declarar variables de ES del sistema
```

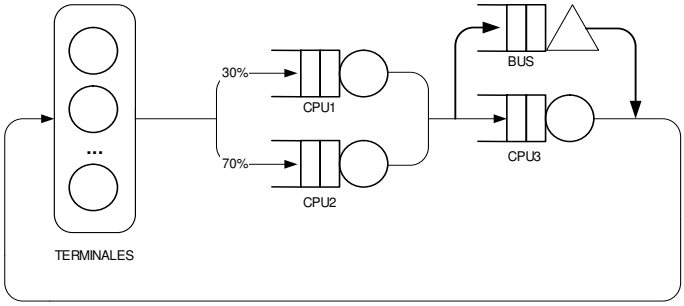

Figura C.1: Sistema Cerrado

```
DK_STATION *processors, *cpu1, *cpu2, *cpu3, *bus;

// Funciones de tiempo de servicio
DK_T_system_time t_processors (DK_CLIENT *c)
    {return (expntl(tproc));}
DK_T_system_time t_cpu1 (DK_CLIENT *c)
    {return (expntl(tcpu1));}
DK_T_system_time t_cpu2 (DK_CLIENT *c)
    {return (expntl(tcpu2));}
DK_T_system_time t_cpu3 (DK_CLIENT *c)
    {return (expntl(tcpu3));}

// Funciones de transito (topologia)
DK_STATION *trans_cpu (DK_CLIENT *c)
    {
    return bus;
    }
DK_STATION  *trans_cpu3 (DK_CLIENT *c)
    {
    if(!DK_Free_Resource(c,bus,1))
        {DK_Error(); exit(0);}
    return processors;
    }
DK_STATION  *trans_bus (DK_CLIENT *c)
    {
    return cpu3;
    }
DK_STATION *trans_processors (DK_CLIENT *c)
    {
    if (random(1,100)<=30) return cpu1;
    else return cpu2;
    }

// Funcion de asignacion de recursos
DK_T_servers  resource (CLIENT *c)
    {
    return 1;
    }

void main (void)
{
int i;
```

```
// Inicializar estructuras del simulador
DK_Init();

// Crear las ES
processors = DK_Service_Station("Processors", DK_INFINITE,
             DK_NO_EXPULSIVE, t_processors, DK_Prior,
             NULL, trans_processors, NULL);
cpu1       = DK_Service_Station("Cpu1", 1, DK_NO_EXPULSIVE,
             t_cpu1, DK_Fifo, NULL, trans_cpu, NULL);
cpu2       = DK_Service_Station("Cpu2", 1, DK_NO_EXPULSIVE,
             t_cpu2, DK_Fifo, NULL, trans_cpu, NULL);
bus        = DK_Resource_Station("Bus", 4,  resource, DK_Fifo,
             trans\_bus, NULL);
cpu3       = DK_Service_Station("Cpu3", 1, DK_NO_EXPULSIVE ,
             t_cpu3, DK_Fifo, NULL, trans_cpu3, NULL);

// Crear clientes
for (i=0; i<nt; i++)
    DK_Insert_Client (i+1, DK_NO_PRIOR, DK_NO_CLASS, processors);

// Simular y obtener resultados
if (!DK_Simul("CPUs and a Bus", 0, tmax))
    DK_Error();
else DK_Results();
}
```

C.2. Implementación del sistema cerrado en SMPL

```
#define tcpu1 2.0
#define tcpu2 1.0
#define tcpu3 1.0
#define tproc 1.0
#define tmax  1000000.0
#define nt    2000

void main()
{
int custom=1, num=2, event, cpu1, cpu2, cpu3,bus, i;

smpl(1,"Clientes");
```

```
cpu1 = facility($"cpu1"$,1);
cpu2 = facility($"cpu2"$,1);
bus  = facility($"bus"$,4);
cpu3 = facility($"cpu3"$,1);

for (i=1; i<=nt; i++) schedule(1,0.0,i);

while (time()<tmax)
    {
    cause(&event,&custom);
    switch(event)
        {
        case 1:  // Inicializar
            if (random(1,100)<=30) schedule(2,0.0,i);
            else  schedule(3,0.0,i);
            break;
        case 2: // Pedir acceso a cpu1
            if (request(cpu1,custom,0)==0)
            schedule(4,expntl(tcpu1),custom);
            break;
        case 3: // Pedir acceso a cpu2
            if (request(cpu2,custom,0)==0)
            schedule(5,expntl(tcpu2),custom);
            break;
        case 4: // Salir de cpu1
            release (cpu1,custom);
            schedule(6,0.0,custom);
            break;
        case 5: // Salir de cpu2
            release (cpu2,custom);
            schedule(6,0.0,custom);
            break;
        case 6: // Entrar a bus
            if (request(bus,custom,0)==0)
            schedule(7,0.0,custom);
            break;
        case 7: // Entrar a cpu3
            if (request(cpu3,custom,0)==0)
            schedule(8,expntl(tcpu3),custom);
            break;
        case 8: // Salir de cpu3
```

```
            release(bus,custom);
            release(cpu3,custom);
            schedule(9,0.0,custom);
            break;
        case 9: // Entrar a procesadores
            schedule(1,expntl(tproc),custom);
            break;
        }
    }
report();
}
```

C.3. Implementación del sistema cerrado en JDESK

```
import JDESK.*;

public class Central_Server extends Global
{
JDESK_STATION processors, cpu1, cpu2, cpu3, bus;
JDESK_Interface f;

public Central_Server()
    {
    processors = new JDESK_STATION();

    cpu1 = new JDESK_STATION();
    cpu2 = new JDESK_STATION();
    cpu3 = new JDESK_STATION();
    bus = new JDESK_STATION();
    f = new JDESK_Interface();
    }

// Funciones de tiempo de servicio
class t_processors implements JDESK_T_ptrfuncserv
    {
    rand r;
    public t_processors () {r=new rand();}
    public double ptrfuncserv (JDESK_CLIENT c)
        {return (r.expntl(1));}
    }
class t_cpu1 implements JDESK_T_ptrfuncserv
```

```
    {
    rand r;
    public t_cpu1 () {r=new rand();}
    public double ptrfuncserv (JDESK_CLIENT c)
        {return (r.expntl(2));}
    }
class t_cpu2 implements JDESK_T_ptrfuncserv
    {
    rand r;
    public t_cpu2 () {r=new rand();}
    public double ptrfuncserv (JDESK_CLIENT c)
        {return (r.expntl(1));}
    }
class t_cpu3 implements JDESK_T_ptrfuncserv
    {
    rand r;
    public t_cpu3 () {r=new rand();}
    public double ptrfuncserv (JDESK_CLIENT c)
        {return (r.expntl(1));}
    }

// Funciones de transito (topologia)
class trans_cpu implements JDESK_T_ptrfunctran
    {
    public trans_cpu(){}
    public JDESK_STATION ptrfunctran (JDESK_CLIENT c)
        {return bus;}
    }

class trans_cpu3 implements JDESK_T_ptrfunctran
    {
    rand r;
    public trans_cpu3 () {r=new rand();}
    public JDESK_STATION ptrfunctran (JDESK_CLIENT c)
        {
        if (!f.JDESK_Free_Resource (c,bus,new JDESK_T_servers(1)))
            { f.JDESK_Error(); System.exit(0); }
        return processors;
        }
    }
class trans_bus implements JDESK_T_ptrfunctran
    {
```

```
    public trans\_bus(){}
    public JDESK_STATION ptrfunctran (JDESK_CLIENT c)
        {return cpu3;}
    }
class trans_processors implements JDESK_T_ptrfunctran
    {
    rand r;
    public trans_processors (){r=new rand();}
    public JDESK_STATION ptrfunctran (JDESK_CLIENT c)
        { if (r.random(1,100)<=30) return cpu1;
          else return cpu2;
        }
    }

// Funcion de asignacion de recursos
class recurso implements JDESK_T_ptrfuncrec
    {
    public recurso(){}
    public long ptrfuncrec (JDESK_CLIENT c)
        {return 1;}
    }

void simul()
{
int i;

JDESK_T_schedule plan = new JDESK_T_schedule ();

f.JDESK_Init();

processors = f.JDESK_Service_Station("Processors", JDESK_INFINITE,
             plan.JDESK_NO_PREEMT, new t_processors(), new JDK_Prior(),
             null, new trans_processors(), null);
cpu1       = f.JDESK_Service_Station("Cpu1", 1,
             plan.JDESK_NO_PREEMT, new t_cpu1(), new JDESK_Fifo(), null,
             new trans_cpu(), null);
cpu2       = f.JDESK_Service_Station("Cpu2", 1,
             plan.JDESK_NO_PREEMT, new t_cpu2(), new JDESK_Fifo(), null,
             new trans_cpu(), null);
bus        = f.JDESK_Resource_Station("Bus", 4, new recurso(),
             new JDK_Fifo(), new trans_bus(), null);
cpu3       = f.JDESK_Service_Station("Cpu3", 1,
```

```
            plan.JDESK_NO_PREEMT,
            new t_cpu3(), new JDESK_Fifo(), null, new trans_cpu3(),
            null);

for (i=0; i<10; i++)
    f.JDESK_Insert_Client (new JDESK_T_client_number(i+1),
    new JDESK_T_client_prior(JDK_NO_PRIOR),
    new JDESK_T_client_class(JDESK_NO_CLASS), processors);

if (!f.JDESK_Simulation("Central Server M/M/1", new
    JDESK_T_system_time(0), new JDESK_T_system_time(25)))
    f.JDESK_Error();
else
    {
    f.JDESK_Results_Table (processors, true);
    f.JDESK_Results_Table (cpu1, false);
    f.JDESK_Results_Table (cpu2, false);
    f.JDESK_Results_Table (cpu3, false);
    f.JDESK_Results_Table (bus, false);
    }
}

public static void main(String[] args)
    {
    Central_Server rp=new Central_Server();
    p.simul();
    }
}
```

Apéndice D

Estudio de Fly3D

D.1. Introducción

Fly3D SDK es un motor de videojuegos y de desarrollo de aplicaciones 3D en tiempo real. Creado por Alan Watt y Fabio Policarpo para Paralelo Computação (Brasil) [Paralelo]. Incluye un potente motor 3D de simulación orientado a objetos que se encarga de tareas como visualización en tiempo real, captura de eventos de entrada (ratón, teclado y joystick), simulación física de objetos 3D,...

La versión utilizada en la presente tesis es la v2.0. La versión v1.2 es de código libre y está permitida su utilización en la creación de aplicaciones (incluso comerciales). La versión v2.0 se adquiere con el libro [Watt:2003].

El motor está completamente implementado en C++. Soporta OpenGL 1.1 como interfaz gráfica de visualización 3D, DirectInput para interfaz de usuario, DirectSound para sonido 3D y DirectPlay para multijugador. Los requerimientos de Fly3D son: Windows 9x/2000/XP/NT, tarjeta gráfica 3D que soporte OpenGL, DirectX SDK, Visual C++ 6.0 o superior para la creación de plugins y 3DStudio Max 3.x o superior para modelado de escenas.

D.2. Características

Fly3D está orientado a la creación de juegos similares a Quake (FPS) pero permite la creación de otros tipos de videojuegos. Tanta es la orientación a Quake, que incluye un conversor de mapas de Quake 3 (por lo que es posible utilizar cualquier editor de niveles de Quake 3 para crear mapas de Fly3D). También incluye un visualizador de grandes terrenos, lo que permite no limitar el tipo de juego creado.

Fly3D está completamente orientado a plugins (DLL), es decir, el código de una aplicación es mayoritariamente un conjunto de DLL. Para crear un nuevo juego es

necesario crear nuevos plugins. Fly3D contiene un asistente de Visual C++ para la creación de plugins.

Fly3D difiere de otros motores de videojuegos en la aproximación utilizada para la creación de videojuegos. Otros motores esperan que el programador cree el núcleo del juego, mientras que Fly3D ya proporciona este núcleo y permite incorporar elementos del juego usando nuevos plugins. El programador no modifica el núcleo, sino que incorpora nuevos elementos que se ejecutarán iterando un bucle principal común a todas las aplicaciones. El programador se despreocupa de ciertas tareas de la creación del videojuego, como código para cargar niveles, localizar entidades o cargar sus valores por defecto. Esto puede verse como una restricción, pero ahorra mucho tiempo en la creación de videojuegos. Es posible crear un juego sin necesidad de teclear una sola línea de código, utilizando los plugins del motor. Permite definir niveles y entidades utilizando el editor.

Fly3D está basado en tecnología BSP (Binary Space Partitioning). La tecnología BSP se desarrolló para juegos en primera persona, pero es una tecnología tan generalista y versátil que admite multitud de aplicaciones. Los diferentes tipos de aplicaciones utilizan diferente particionado del BSP, pero lo realiza el motor de forma completamente transparente al usuario (esta tarea la realiza el plugin que convierte el modelo en un BSP). Esta tecnología permite combinar tipos de escenas diferentes en los juegos, como entornos cerrados, estrategia 3D o grandes superficies. Mezcla técnicas BSP con jerarquías de objetos por niveles.

D.2.1. Herramientas

Junto al SDK, Fly3D también ofrece diversas herramientas y utilidades:

- ***flyFrontend***: es la interfaz principal del motor (permite que el juego se ejecute, controlando el bucle principal de la aplicación). Consta de una ventana de visualización, donde se realiza la simulación y un menú para abrir archivos *.fly*. Contiene un menú que permite seleccionar el modo de pantalla, elegir entre modo jugador o multijugador o reproducir un archivo de tipo demo previamente salvado. La secuencia de operaciones que realiza este front-end es:

  - Inicializar el videojuego: ventana principal, motor 3D, visualizador y otros componentes importantes del núcleo.
  - Cargar un nivel: geometría estática, plugins, modelos,... desde un archivo *.fly*.
  - Iterar el bucle principal de la aplicación.
  - Cerrar el motor y liberar recursos.

- ***flyEditor***: es la principal herramienta de edición de Fly3D, consiste en una estructura de árbol, donde se muestran y clasifican todas las entidades de la escena, una lista donde los parámetros se muestran y editan y una ventana de visualización donde se puede observar el juego o la aplicación ejecutándose. Permite el cambio de cualquier parámetro y la inmediata visualización de los cambios en la ventana de visualización. También se pueden cargar plugins adicionales a usar en el juego, añadir, modificar o eliminar objetos, y salvar archivos *.fly* con la configuración.

- ***flyServer***: consola servidora para multijugador. Se llama desde *flyFrontend*, a través de la opción del multijugador del menú, para crear un servidor para la escena deseada. De esta manera, los jugadores se conectan al servidor usando el comando de consola *connect* o desde el menú de *flyFrontend*.

- ***flyShader***: permite editar efectos de sombras. Permite cargar la lista completa de sombras de una escena, editar efectos y comprobar el resultado en la ventana de visualización.

- ***fly3d.ocx***: control ActiveX para Fly3D. Con él, la totalidad del motor esta disponible para aplicaciones web, tal como juegos 3D para web, navegación 3D, tiendas virtuales 3D o presentaciones 3D. Únicamente se necesita crear un archivo *.html* con el control Fly3d.ocx dentro de éste, usar los comandos del control para cargar archivos *.fly* y arrancar la simulación 3D.

- Editor de scripts para archivos *.fly*.

- Plugins para 3DStudio Max 3.x y 4.x para importar o exportar a formatos *.f3d* y *.k3g* con cualquier número de animaciones y cualquier número de keys. Permite crear modelos de juegos y niveles.

- Conversor de niveles para otros formatos del juego. Convierte geometría y mapas de textura desde otros formatos de juegos a Fly3D. En concreto, el plugin V26 es un conversor de niveles de Quake 3 que permite utilizar un editor compatible con Quake3 para la creación de niveles. Es también un editor de propósito general para construir el juego completo a partir de las partes creadas anteriormente.

- Asistente para Visual C++ de creación de plugins. Permite crear plugins fácilmente utilizando esqueletos de clases y controles.

D.2.2. Desarrollo de Videojuegos

Para desarrollar un nuevo juego se deben seguir los siguientes pasos:

- Modelado de la escena: construir el entorno o nivel usando una herramienta de modelado o un editor. Habitualmente se utiliza la herramienta de modelado 3DStudio Max para crear la geometría estática de la escena y las mallas de objetos. BSP, PVS (Potentially Visible Set) y mapas de luces se generan desde el archivo *.max* de 3DStudio Max

- Creación de los objetos dinámicos de la escena: para cada objeto se define su comportamiento (visual y dinámico) y la interactividad con los otros objetos. Los objetos deben heredar del objeto base del motor y deben aglutinarse en plugins. Permite editar plugins ya existentes para definir nuevos juegos con el mínimo esfuerzo. Los plugins se usan para crear las clases que describen los objetos del videojuego.

- Incorporar los plugins al motor de la aplicación.

- Crear la interfaz con el usuario (front-end).

D.2.3. Ventajas e Inconvenientes de Fly3D como Herramienta de Creación de Videojuegos

Ventajas

- Contiene excelentes herramientas y utilidades. No se emplea tiempo en reinventar la rueda, las herramientas que ofrece Fly3D ahorran tiempo de desarrollo.

- Código estructurado y modularizado. Buena organización para trabajo en grupo.

- Código fuente del motor disponible.

- Licencia freeware.

Inconvenientes

- Dependiente del sistema.

- Sólo se puede trabajar en máquinas con tarjetas gráficas 3D.

- El diseño de niveles se realiza con 3DStudio Max, lo cual implica un conocimiento mínimo de esta herramienta.

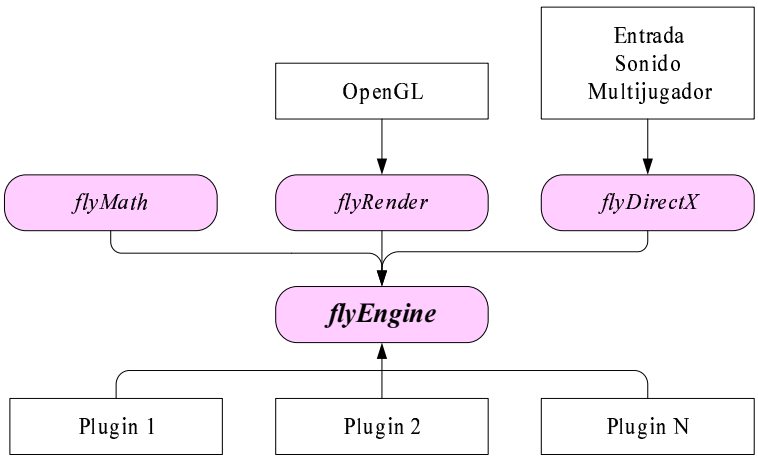

Figura D.1: Arquitectura de Fly3D

D.3. Arquitectura

Los objetivos de diseño de Fly3D han sido:

- Conseguir una generación de imágenes eficiente y de calidad.
- Crear una plataforma de desarrollo de aplicaciones gráficas en tiempo real que permita desarrollar aplicaciones fácilmente.
- Posibilitar la ampliación del motor con nuevas funcionalidades.

Para adaptarse a los objetivos de diseño, la arquitectura de Fly3D:

1. Separa el motor de las posibles aplicaciones creadas. El motor se convierte en una herramienta usada por el desarrollador del videojuego, por lo que no necesita conocer detalles sobre su implementación.

   a) Los videojuegos creados con el motor son un conjunto de plugins, completamente diferenciados del motor.

   b) Los plugins está separados del front-end del videojuego creado, permitiendo una mayor modularidad.

2. Explota al máximo la herencia de C++. El motor trata con objetos de tipo *flyBspObject*. Todos los objetos del juego heredan de esta clase base.

D.3.1. Módulos

Fly3D está divido en cuatro módulos (figura D.1): la biblioteca de DirectX, la librería de visualización, la librería matemática y la librería del motor. Cada uno de estos módulos son plugins. Los tres primeros módulos se enlazan al módulo del motor, que los integra en una potente herramienta de visualización y simulación.

- **flyDirectX**: interfaz entre la API de DirectX y Fly3D. Es el único módulo que tiene acceso a las estructuras de datos y operaciones ofrecidas por DirectX. Implementa clases que gestionan: entrada, sonido e interfaz multijugador.
- **flyRender**: módulo de visualización del motor. Se encarga de todas las operaciones y estructuras de datos relacionadas con la tarea de visualización, usando OpenGL como API gráfica. El núcleo es independiente del componente de visualización.
- **flyMath**: módulo matemático. Incluye clases que implementan todas las operaciones matemáticas necesarias en la simulación.
- **flyEngine**: módulo principal del motor. Representa un interfaz entre los plugins y el resto de los módulos de Fly3D. Gestiona la carga y salvado de escenas estáticas, los plugins, los objetos dinámicos y realiza la simulación. Implementa varias clases y métodos que manejan la actualización del estado del sistema por cada frame y coordina los distintos plugins, dándoles a cada uno de ellos la oportunidad de añadir su funcionalidad a la simulación. También permite cargar la escena, inicializar los plugins y objetos y arrancar la simulación.

D.3.2. Elementos de una Aplicación Gráfica: Front-Ends y Plugins

Añadir nuevas funcionalidades a Fly3D implica definir dos tipos de elementos: plugins y front-ends.

El motor debe enlazarse con los plugins y el front-end de la aplicación creada (figura D.2). Se puede acceder desde los plugins y el front-end a los componentes del motor directamente.

D.3.2.1. Front-Ends

El **front-end** es la interfaz con el usuario de la aplicación creada con Fly3D. Se ejecuta sobre el sistema operativo utilizando elementos del motor de simulación, plugins y utilidades. El front-end es el responsable de inicializar la aplicación, llamar a los plugins necesarios, abrir una escena y controlar el bucle principal de la aplicación.

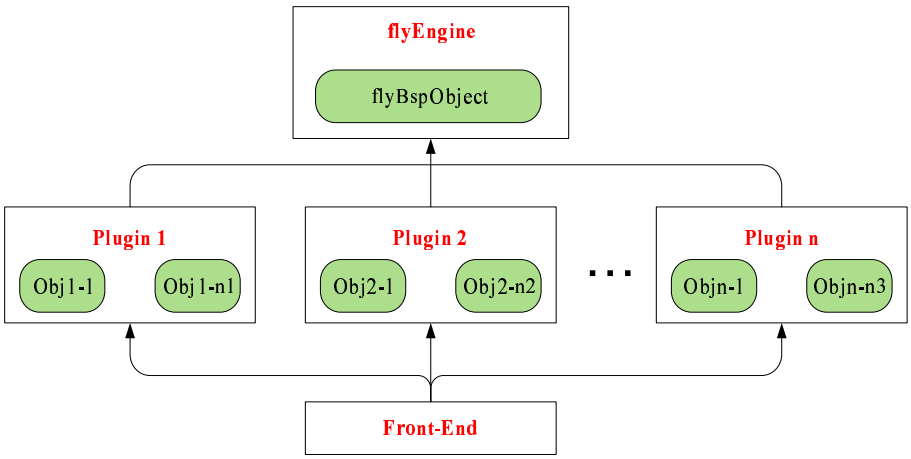

Figura D.2: Comunicación entre los elementos de una aplicación en Fly3D

Una aplicación tiene, como mínimo, un front-end que es la interfaz de la aplicación con el usuario. Pero una misma aplicación puede tener varias subaplicaciones incluidas, cada una con su front-end.

D.3.2.2. Plugins

Los **plugins** son ficheros DLL (Dynamic Link Libraries [Klein:1993]) que implementan comportamientos nuevos o específicos del juego. Los plugins se enlazan con la aplicación en tiempo de ejecución. Aglutinan objetos definidos por el programador del juego. Para cada objeto se define su comportamiento visual y dinámico, lo que supone definir el comportamiento global del juego.

Los objetos del juego heredan del objeto base de *flyEngine flyBspObject*, común para todo el motor. Las clases heredadas redefinen los métodos virtuales de *flyBspObject* para definir o ajustar el comportamiento del objeto. Un plugin puede definir cualquier número de clases de objetos y en cada clase pueden incluirse propiedades adicionales del objeto.

Esta forma de organizar los objetos permite definir fácilmente objetos y asignarles comportamientos. Con esta nueva aproximación, es mucho más simple añadir una nueva funcionalidad o característica al software sin tener que recompilar todo de nuevo, simplemente se crea un plugin y se enlaza al motor. El plugin es capaz de usar todas las clases, métodos y variables del motor.

Los plugins puede definirlos el usuario o utilizar los ya existentes. Un plugin no pertenece en exclusiva a una aplicación en concreto. El front-end no tiene conocimiento de la naturaleza de los plugins que utiliza. La figura D.2 muestra la integración de plugins y front-ends en Fly3D. Cuando se ejecuta un front-end se

cargan únicamente los plugins que utiliza.

D.3.3. Objetos Base *flyBspObject*

La clase *flyBspObject* define el objeto base de todos los objetos de las aplicaciones creadas con Fly3D. Incluye una serie de funciones virtuales que deben redefinir las clases heredadas (entre ellas hay dos especialmente importantes, pues son las funciones virtuales que usan los objetos heredados para definir simulación y visualización). Estas funciones se pueden dividir en categorías:

1. **Funciones de visualización**: estas funciones sirven para visualizar el objeto, para realizar la detección de colisiones,... Son:
   - *get_mesh*: retorna la malla que representa un objeto (si tiene una representación de malla). Esta función dibuja, obtiene la intersección de rayos y realiza la detección de colisiones. La invoca *collide* de *flyBoundBox* para detectar colisiones.
   - *draw_shadow*: dibuja la sombra de un objeto. La invoca *draw_bsp* durante el proceso de visualización. Es parte del proceso de visualización.
   - *ray_intersect y ray_intersect_test*: definidas para objetos que requieren rutinas especiales de intersección de rayos. Se invoca durante el proceso de visualización mediante una llamada directa a función.
   - *draw*: dibuja el objeto. Durante el proceso de visualización se invoca esta función para cada uno de los objetos de la escena, para que ellos mismos se dibujen.
2. **Funciones de comunicación**:
   - *message*: envía y procesa los mensajes entre objetos (de tipo *msg*). Fly3D permite cierta comunicación entre objetos mediante paso de mensajes. Los mensajes definidos por Fly3D tienen una funcionalidad muy limitada (iluminación dinámica o mensajes del cliente). El envío y la recepción de mensajes la modela el usuario cuando define los objetos. Los tipos de mensajes que define Fly3D son (aunque el usuario puede definir sus propios tipos de mensajes):
     - *FLY_OBJMESSAGE_CHANGEPARAM*: cambio de parámetros.
     - *FLY_OBJMESSAGE_ILLUM*: buscar luces y auto-iluminación.
     - *FLY_OBJMESSAGE_STATICILLUM*: buscar luces estáticas y auto-iluminación.
     - *FLY_OBJMESSAGE_DYNILLUM*: buscar luces dinámicas y auto-iluminación.
     - *FLY_OBJMESSAGE_DAMAGE*: splash damage.

- *FLY_OBJMESSAGE_DAMAGECONTACT*: contact damage.
- *FLY_OBJMESSAGE_LIGHT*: iluminar, generar una fuente de luz en la posición actual.
- *FLY_OBJMESSAGE_TRIGGER*: generar un disparador.

Además de los objetos, otros elementos del sistema utilizan la función *message* para comunicarse con los objetos, como, *update_instances* de *CFlyEditorView* quien envía mensajes a los objetos de tipo *FLY_OBJMESSAGE_CHANGEPARAM*

3. **Funciones de simulación**:

   - *step*: realiza la simulación del objeto. Tiene como parámetro el tiempo transcurrido desde la última escena. El objeto debe actualizarse utilizando este tiempo. Si durante este intervalo, el objeto ha cambiado su posición debe actualizarse el BSP.
   - *post_step*: proceso extra que se realiza después de ejecutar la simulación de todos los objetos (función *step* de *flyEngine*). Para que se ejecute esta función para un objeto, debe estar incluido en una lista de objetos especial (*poststeps*). Cualquier objeto puede incluirse en esta lista ejecutando la función *add_post_step*. El proceso de modificación es similar al anterior.

4. **Varias**:

   - *clone*: retorna una nueva instancia del objeto con el mismo estado que el objeto original. Se utiliza para duplicar objetos. Se invoca:
     - Cuando se quiere crear un nuevo objeto de un tipo determinado, primero se clona y después se activa el objeto clonado. La invocan los objetos definidos por el usuario. La creación de los objetos puede ser:
       - Consecuencia del propio proceso de simulación del objeto, por ejemplo en la función *step* del objeto.
       - Consecuencia de la interacción con otros, por ejemplo, cuando se crea un *powerup* como consecuencia de un mensaje de la función *mp_message* de tipo *FLYMP_MSG_POWERUPCREATE*.
     - Cuando se une al juego un nuevo jugador por red. En la función *mp_message* de ciertos objetos, cuando reciben un mensaje de tipo *FLY_MP_MSG_JOIN*.
     - En la función *command_exec* de la consola (*flyConsole.cpp*), como consecuencia de un comando *activate*.
     - En la función *command* del servidor (*flyServer.cpp*), como consecuencia de un comando *activate*.
     - En la función *OnActivate* de *flyEditorDoc.cpp*.

- *get_param_desc y get_custom_param_desc*: retorna la descripción de los parámetros del objeto. Lo invocan
  - *command_exec* de *flyConsole*
  - *delete_references* de *flyDllGroup*
  - *save_fly_file* de *flyEngine*
  - *get_obj_param* de *flyEngine*
  - *set_obj_param* de *flyEngine*
  - *command* de *server_flyEngine*
  - *OnEditCopy* de *CLeftView* (*flyEditor*)
  - *OnEditPaste* de *CLeftView* (*flyEditor*)
  - *set_param_list* de *CFlyEditorView*
  - *update_param_list* de *CFlyEditorView*
- *init*: inicializa el objeto antes de activarlo. La función de inicialización se invoca en las siguientes situaciones:
  - Cuando se cargan los datos de juego. La función *load_data* de *flyEngine* invoca a la función de inicialización de cada objeto definido en el fichero. La función *load_data* se invoca desde *open_fly_file*, que a su vez se invoca en el bucle principal de la aplicación.
  - Cuando se crea dinámicamente un objeto y se incluye en el juego (misiles que explotan generando partículas, o powerups). Se invoca la función *activate* de *flyEngine*. Esta función después de inicializar el objeto lo incluye en el grafo de escena.
  - En la función *mp_message* de ciertos objetos, cuando reciben un mensaje de tipo *FLYMP_MSG_RESTARTGAME* indicando que se inicialice el juego.

D.4. Bucle Principal

El bucle principal de Fly3D (algoritmo 2) sigue el esquema continuo típico de videojuegos de acoplo de visualización y simulación. En este bucle se realizan tres procesos principales: obtención de eventos de usuario, simulación y visualización. Este bucle principal se utiliza en casi todas las aplicaciones creadas con Fly3D (ciertos front-end no necesitan visualizado, como el front-end servidor, encargado del control de red, pero son casos particulares). Las aplicaciones gráficas en tiempo real que el usuario pueda crear contendrán este bucle completo.

La simulación la controla el módulo *flyEngine* y se lleva a cabo invocando a la función *step* del núcleo. La visualización la controla el módulo *flyRender* y se arranca mediante la función *draw_frame*.

Algoritmo 2 Bucle principal de Fly3D

```
while true do
  Obtener y procesar mensajes de usuario
  {Simular}
  flyengine → step()
  {Visualizar}
  rend → draw_frame()
end while
```

D.4.1. Proceso de Simulación

Cada objeto del juego define en su función *step* (función virtual de *flyBspObject*) su propio proceso de simulación. El mecanismo de simulación se encarga de que se llame a la función *step* de cada uno de los objetos de la aplicación.

El núcleo posee las siguientes funciones para el proceso de simulación:

- *step*: simula la escena desde el último frame. Calcula el intervalo de tiempo desde la última simulación y llama a *step(dt)*.
- *step(dt)*: actualiza la escena para el intervalo *dt*, invocando para ello las siguientes funciones:
  - *step_init(dt)*: actualización inicial de la escena, mapas de luces, niebla y caras.
  - *step_objects(dt)*: actualiza los objetos de la escena.
  - *step_end(dt)*: actualización final de la escena, después de la simulación de los objetos (mapas de luces, niebla y caras).

La simulación comienza cuando se invoca la función *step* del núcleo (algoritmo 3). Sus tareas principales son:

- Comprobar si debe lanzarse la simulación debido al proceso de un nuevo jugador.
- Calcular el intervalo de tiempo desde la última simulación.
- Comprobar si el intervalo de tiempo está entre los límites válidos.
- Lanzar la simulación en ese intervalo de tiempo, invocando la función *step(dt)* del núcleo.

La función *step(dt)* del núcleo (algoritmo 4) se encarga de:

Algoritmo 3 Función *step()* del núcleo de Fly3D

```
if dt_multijugador ≠ 0 then
  {Simulación por multijugador}
  dt = dt_multijugador
  T_n = T_{n-1} + dt
  T_{n-1} = T_n
  step(dt)
  return dt
end if
{Calcular el tiempo transcurrido desde el último frame}
Obtener el tiempo actual T_n
dt = T_n − T_{n-1}
{Si el intervalo de tiempo dt es muy elevado o muy bajo, no se simula}
if dt > 0 then
  T_{n-1} = T_n
  if dt < 1000 then
    {Simular todo el sistema durante el intervalo dt ms}
    step(dt)
    return dt
  end if
end if
return 0
```

- Procesar los comandos de la consola.
- Procesar mapas de luces y niebla.
- Simular los objetos activos (algoritmo 5). Todos los objetos activos de la aplicación están incluidos en la lista de objetos activos. Se invoca la función de simulación *step* de cada objeto de la lista de objetos activos. La cámara y el jugador se simulan antes que el resto de elementos de la lista.
- Envía un mensaje a todos los plugin cargados para que actualicen su estado. Aquí se realiza la simulación a nivel de plugin, lo que puede afectar a los objetos del plugin. Realiza tareas de simulación que no dependen de un objeto concreto, que pueden afectar a todos ellos, como por ejemplo obtener eventos de usuario y modificar los objetos del plugin convenientemente. Si no existe un código de simulación común al plugin, no es necesario implementar la respuesta a este mensaje en el plugin.
- Si la aplicación es multijugador, procesa los mensajes multijugador.
- El último elemento a simular es la consola, por lo que la visualización de la consola se superpone al resto (si está activa). Ejecuta la función *step* de la consola.

El proceso completo de simulación se muestra en la figura D.3.

La simulación se produce únicamente si el tiempo desde la anterior simulación (dt) es menor que 1000ms. Si dt es mayor que este valor el salto en la simulación será demasiado grande y en estas condiciones no se puede ejecutar el juego, por lo que se toma la decisión de no simular y detener el juego, borrando el intervalo de tiempo.

D.4.2. Proceso de Visualización

Cada objeto define una función *draw* (función virtual de *flyBspObject*) que permite dibujar el objeto, con llamadas a OpenGL.

La visualización comienza después del proceso de simulación. Se invoca desde el bucle principal de la aplicación (algoritmo 2) mediante la función *draw_frame* del núcleo. La visualización sólo se realiza si se ha simulado.

El proceso de visualización (figura D.4) es el siguiente:

- El núcleo envía un mensaje FLY_MESSAGE_DRAWSCENE a todos los plugins de la aplicación indicando que deben visualizarse.

Algoritmo 4 Función *step(dt)* del núcleo de Fly3D

```
Procesar los comandos de consola
Vaciar la lista de objetos para los que se ejecuta post_step (poststeps)
{Restaurar mapas de luz, de niebla y de caras}
step_init(dt)
{Simular cada objeto activo}
step_objects(dt)
{Volver a modificar mapas de luz, de niebla y de caras después de la simulación de objetos}
step_end(dt)
Procesar los mensajes multijugador
Actualizar los clientes multijugador
{Enviar un mensaje a todos los plugins indicando que se ha realizado un paso de simulación}
dll.send_message(FLY_MESSAGE_MPUPDATE, 0, 0)
{Llamar a la función post_step de todos los objetos}
for all objeto ∈ lista_objetos_post_step do
  objeto → post_step()
end for
{Simular la consola, si está activa}
if consola_activa then
  consola → step(dt)
end if
```

Algoritmo 5 Función *step_objects(dt)* del núcleo de Fly3D

```
jugador → step(dt)
camara → step(dt)
{Procesar objetos dinámicos}
for all objeto_activo ∈ lista_objetos_activos do
  if vida_objeto < 0 then
    Destruir objeto
    Eliminar objeto del BSP y de la lista de objetos activos
  else
    {Simular el objeto}
    objeto → step(dt)
    Reposicionar el objeto en el BSP, si es necesario
  end if
end for
```

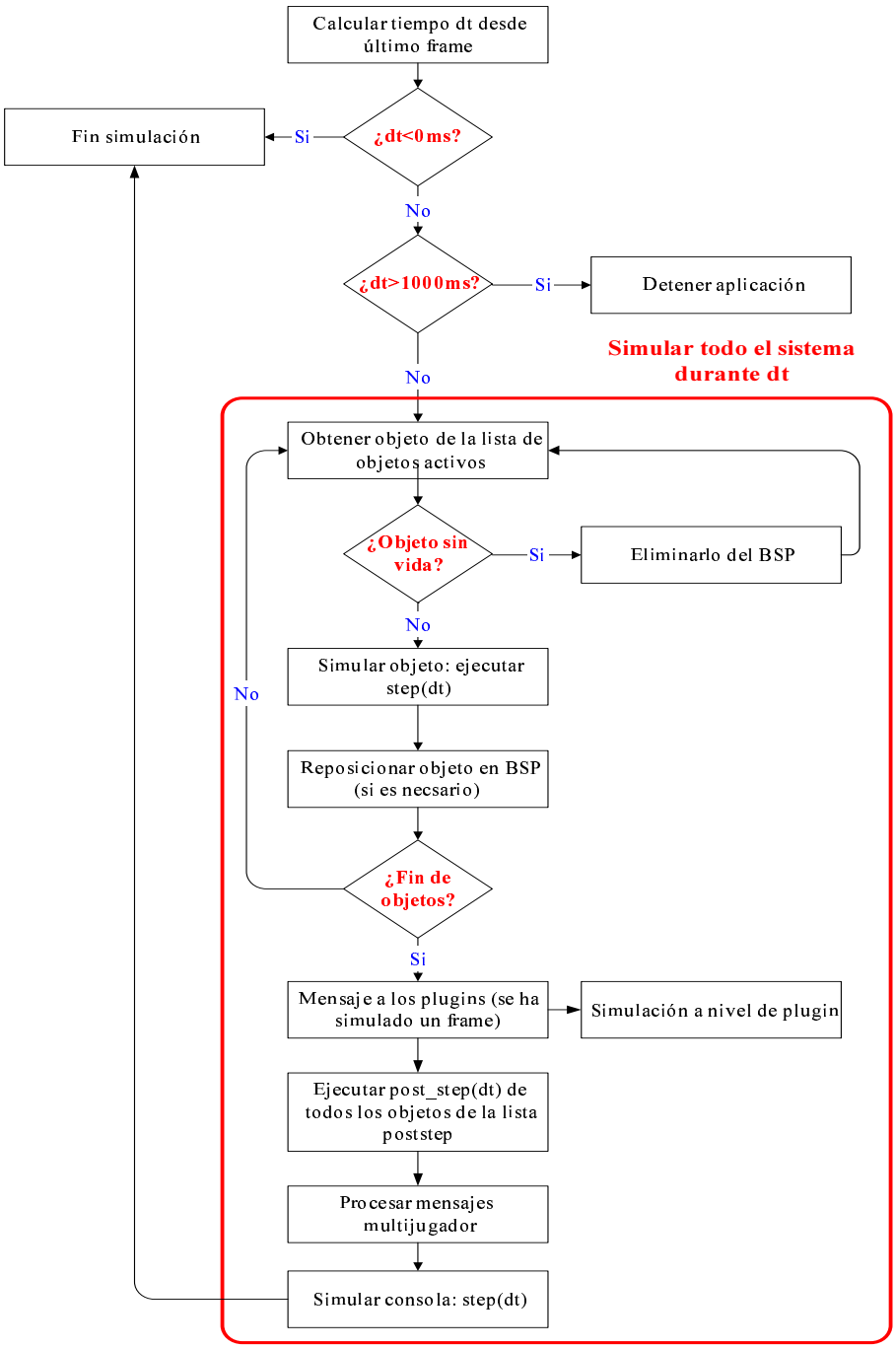

Figura D.3: Proceso de simulación de Fly3D

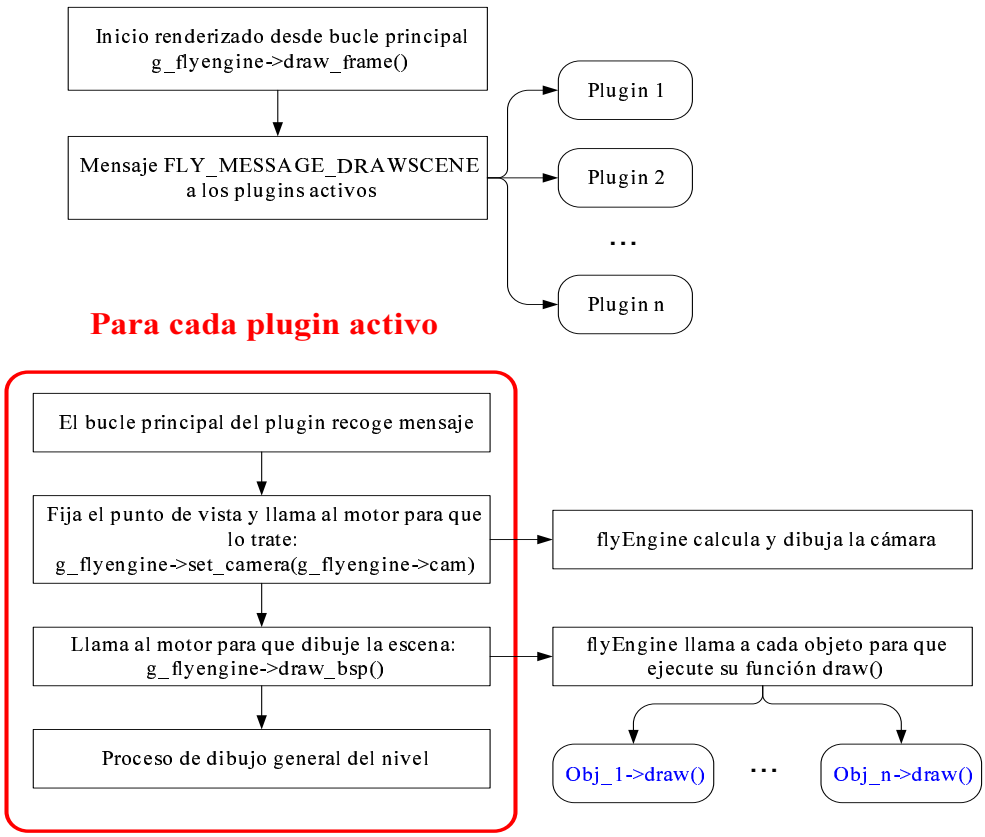

Figura D.4: Proceso de visualización en Fly3D

- Cada plugin fija el punto de vista de la cámara y llama al núcleo para que lo trate.
- Llama al núcleo para que dibuje la escena.
- El núcleo invoca, uno a uno los objetos visibles, para que se dibujen, ejecutando su función *draw*. Por tanto, la visualización se produce dentro de los objetos.

D.4.3. Comunicación entre Objetos y Plugins

Los plugins pueden comunicarse con los objetos que contienen o con otros plugins de la aplicación mediante dos tipos de mensajes:

- Mensajes de objetos: se procesan mediante la función virtual de flyEngine *send_bsp_message*. Envía un mensaje a todos los objetos derivados de *flyBspObject* especificados en su esfera de influencia. Las esferas de influencia se

usan para múltiples operaciones del motor. Permiten una selección rápida de los objetos de la esfera, sea cual sea su posición en el BSP.

- Mensajes de plugins: se procesan mediante la función de flydll *send_message*. Esta función envía el mensaje a todos los plugins de la escena actual, llamando a la función *fly_message* de flydll.

Apéndice E

Integración de GDESK en Fly3D

En este apartado se muestran dos ejemplos representativos del cambio de Fly3D a DFly3D:

- Creación de un objeto del sistema (objeto *consola*).
- Cambio del objeto *bola* continuo al objeto *bola* discreto.

E.1. Objeto del Sistema *Consola*

E.1.1. La Consola en Fly3D

La consola (tipo *flyConsole* de Fly3D) es un elemento del núcleo de Fly3D presente en todas las aplicaciones creadas con el motor. La consola es un atributo de *flyEngine* (*con*). Se encarga de procesar los comandos de usuario que se introducen vía teclado. La consola consta de una línea de comandos donde se introducen en modo texto los comandos a ejecutar y se muestra información sobre su procesamiento. En modo consola, además de los comandos de texto, pueden usarse ciertas teclas de desplazamiento (*home*, *end*, *page up* y *page down*) para deslizarse por la pantalla de consola.

La consola no es un objeto de Fly3D:

- No deriva del objeto base del sistema *flyBspObject*, pero contiene funciones de visualización y simulación.
- No está incluida en el grafo de escena.
- La visualización y la simulación de la consola no se realizan junto con el resto de los objetos del sistema, sino que se tratan de forma particular. De esta forma la consola se sobrepone al resto de elementos del sistema.

  - **Simulación**: al finalizar la simulación del sistema (función *step* de *flyEngine*) se simula la consola (función *step* de la consola). La función de simulación *step* de la consola actualiza la posición actual en la pantalla de consola.
  - **Visualización**: la función de visualización *draw* de la consola se invoca en la misma función que el resto de objetos del sistema (en la función *draw_frame* de *flyEngine*, que es la función general de visualización del sistema). La función *draw* de la consola dibuja la pantalla y el texto mostrado en ella.
- No contiene la función *message*, por lo que no es sensible a mensajes de otros objetos o de *flyEngine*.

El objetivo del estudio previo de la consola es identificar que partes de la consola y del sistema en general es necesario modificar.

Para integrar la consola en DFly3D se debe identificar como interactúa la consola con el resto de objetos del sistema:

1. Identificar que elementos externos utiliza la consola en su funcionamiento: utiliza *console_command* de *flyEngine*, de tipo *flyString*. Contiene los comandos que deben ejecutarse en modo consola.
2. Identificar quien y cuando modifica los elementos externos que la consola utiliza. La cadena de comandos *console_command* de *flyEngine* la rellenan, o bien el usuario o bien el sistema con sus propios mensajes.
   - Comandos de usuario: la función de simulación *step* de *menucamera* rellena *console_command* a partir de los comandos que introduce el usuario por teclado cuando Fly3D está en modo consola. Cuando detecta que se ha pulsado la tecla *enter* rellena la cadena y la deja lista para su proceso. Si la cadena de comandos ya contenía datos, se borran. Los comandos quedan preparados para la siguiente simulación, momento en el que se ejecutan.
   - Comandos del sistema:
     - Función *check_multiplayer* de *flyEngine*.
     - Función *WndProc* de *flyFrontend*.
     - Función *play_demo* de *flyEngine*.
     - Función *step* de *trigger*.
     - Función *menu* de *gamemenu*.
     - Función *on_fly_files_change* de *gamemenu*.
     - Función *auto_demo_playback* de *menu*.

- Función *fly_message* de *multimedia*.

3. Identificar que procesos debería realizar la consola pero realizan otros objetos: la función de simulación del núcleo de Fly3D (*step* de *flyEngine*) se encarga de ejecutar los comandos de consola. Comprueba si hay comandos pendientes de proceso. Si los hay, invoca a la función de ejecución de comandos *command_exec* de la consola para cada uno de los comandos pendientes y vacía *console_command* cuando ha finalizado la ejecución.

4. Identificar quien invoca funciones de la consola:

   - Función *key_down*: esta función permite mostrar u ocultar la consola y procesar los eventos de usuario correspondientes a teclas de navegación. La invoca el bucle principal de la aplicación (*WinMain*) cuando detecta un evento de usuario correspondiente a la pulsación de una tecla. Si se pulsa *enter* se ejecutan los comandos introducidos anteriormente en *cmd_line* (mediante la función *key_char*).
   - Función *key_char*: procesa un carácter de teclado introduciéndolo en la cadena de comandos *cmd_line*. La invoca el bucle principal de la aplicación (*WinMain*) cuando detecta un evento de usuario correspondiente a la pulsación de una tecla de tipo carácter.
   - Función *command_exec*: ejecuta directamente un comando pasado como argumento. Se ejecuta en dos ocasiones. Una, invocada por *WndCommand* para ejecutar el menú de la aplicación. Otra, en la función de simulación de *flyEngine*, para ejecutar los comandos de usuario de Fly3D en modo consola.
   - Función *add_string*: permite añadir mensajes al buffer de pantalla en modo comando. Este buffer es el que se muestra en pantalla en la próxima visualización de la consola. Permite mostrar mensajes de Fly3D, las opciones de la aplicación que se activan o desactivan utilizando los menús o mediante teclado,... Se invoca desde *console_printf* de *flyEngine*. Esta función se invoca cada vez que alguna aplicación o el núcleo quiere mostrar algún mensaje informativo en la línea de comandos.
   - Función *draw*: función de visualización de la consola. Se invoca desde *draw_frame* de *flyEngine* (proceso de visualización general).
   - Función *step*: simula la consola. Se ejecuta en la función general de simulación (*step* de *flyEngine*) para simular la consola. Esta simulación actualiza las líneas y posiciones en la pantalla de modo texto de la consola.
   - Otras funciones usan atributos de la consola, los consultan, pero no los modifican:

- *flyServer* usa *linecount* (líneas en pantalla) y *buf* (buffer de escritura de comandos por pantalla).
- *flyFrontend* usa en *WndProc* el atributo *mode* (modo de consola).
- *flyEngine* usa en *step* y *draw_frame* el atributo *mode*.

5. Determinar que parámetros debería tener un mensaje para poder comunicar la consola con el resto de elementos del sistema:
   - Deben existir diferentes tipos de mensajes, pues son varias las situaciones en las que la consola se pone en funcionamiento, por lo que un campo del mensaje debe ser el tipo de mensaje.
   - Las funciones *key_down* y *key_char* tienen como parámetro un valor entero correspondiente a la tecla pulsada.
   - La función *command_exec* tiene como parámetro el comando a ejecutar.
   - La función *add_string* tiene como parámetro la cadena a añadir al buffer de la consola.
   - La función *draw* no utiliza parámetros.
   - La función *step* tiene como parámetro el diferencial de tiempo de la simulación, pero este diferencial se calcula para cada simulación en concreto, por lo que realmente no será un parámetro.

E.1.2. El Objeto *Consola* en DFLy3D

Los objetivos de la creación del objeto *consola* son:

1. La consola debe contener todo el código que le concierne.
2. La consola se debe comunicarse con los otros objetos mediante mensajes de GDESK.
3. La consola únicamente debe actuar si tiene comandos que procesar (no se debe muestrear). La consola sólo actuará cuando reciba un mensaje.

Para conseguir estos objetivos se deben realizar ciertos cambios tanto en la consola como en el núcleo de Fly3D:

1. Cualquier objeto que se integre en DFly3D debe ser descendiente del tipo base de GDESK (*GDESK_CEntity*). En la declaración de la clase consola (*flyConsole*) se debe añadir la herencia pública del objeto base de GDESK.

```
class FLY_ENGINE_API flyConsole : public GDESK_CEntity
```

2. Añadir en el mensaje de la aplicación los atributos necesarios para contener los parámetros que necesita la consola (fichero *GDeskMessage.h*).

   - *MessageType*: tipo de mensaje.
   - *Key*: tecla pulsada por el usuario.
   - *Command*: comando que debe ejecutarse en modo consola.

3. Añadir en la consola (*flyConsole.h*) los atributos necesarios para los nuevos procesos incluidos: variables de tiempo para el cálculo del diferencial de tiempo de la simulación (este proceso se conserva para intentar que el objeto consola tenga la misma funcionalidad en DFLy3D que tenía en Fly3D):

```
int cur_time;         //current time in ms
float cur_time_float; //current time in floating point
int start_time,       //global start time
    frame_rate,       //current frame rate
    frame_count;      //total number of frames
int T0,               //last frame time
    T1;               //current frame time
```

4. La consola es un objeto de GDESK por lo que debe redefinir la función virtual de la clase base *GDESK_ReceiveMessage*. Esta función se ejecuta cuando la consola recibe un mensaje de otro objeto, de él mismo o del sistema. Lleva como parámetro el mensaje recibido. La función *GDESK_ReceiveMessage* selecciona el código a ejecutar dependiendo del tipo de mensaje (definido mediante los parámetros del mensaje). Las posibles acciones a realizar son:

   - Simular (antigua *step* y procesos globales).
   - Procesar una tecla (llamando a *key_down*).
   - Procesar un carácter (llamando a *key_char*).
   - Procesar un comando directamente (llamando a *command_exec*).
   - Añadir un comando al buffer (llamando a *add_string*).

```
void flyConsole::GDESK_ReceiveMessage(GDESK_MESSAGE *pMsg)
{
switch ( pMsg->MessageType ) {
case GDESK_CONSOLE_MSG_KEYDOWN:
    key_down(pMsg->Key); break;
case GDESK_CONSOLE_MSG_KEYCHAR:
    key_char(pMsg->Key); break;
```

```
case GDESK_CONSOLE_MSG_SIMULATION:
    console_simulation(); break;
case GDESK_CONSOLE_MSG_EXEC:
    command_exec(pMsg->Command); break;
case GDESK_CONSOLE_MSG_ADDSTRING:
    add_string(pMsg->Command); break;
default: break;
}}
```

5. **Simulación**: los objetos de Fly3D se simulan mediante la ejecución de la función *step* de cada uno de los objetos. Esta función *step* de los objetos se invoca en la función *step* del núcleo. En concreto la función *step* de la consola se ejecuta al finalizar el proceso de simulación del núcleo para darle prioridad sobre el resto de los objetos del sistema. La función de simulación de la *consola* debe aunar todas la funcionalidad que anteriormente estaba distribuida por el núcleo:

   - Procesar los comandos de consola. La consola comienza a funcionar cuando recibe un mensaje indicando que hay comandos pendientes de procesar. Este código lo realizaba *step* de *flyEngine*. Se ha decidido mantener la función *step* de la consola, en lugar de integrar su código en la nueva función de simulación de la consola (*console_simulation*), para intentar minimizar los cambios respecto a la versión original.
   - Calcular el tiempo *dt* desde la última simulación. La función *step* tiene como parámetro el diferencial de tiempo para el que se va a realizar la simulación. Este diferencial lo calcula la función de simulación de *flyEngine* y es el mismo diferencial para todos los objetos del sistema. Ahora ese cálculo del diferencial debe integrarse en la simulación.
   - Simular: ejecutar la función *step* de la consola (sólo si la consola está activa).

```
void flyConsole::console_simulation() {
// Procesar los comandos de consola
if (g_flyengine->console_command[0])
{   int p;
    flyString str=g_flyengine->console_command;
    flyString str2;
    do  {
        p=str.find(';');
        if (p==-1) p=str.length();
        command_exec(str.left(p));
```

```
            if (str.length()-p-1>0){
                str2.copy(str,p+1,str.length()-p-1);
                str=str2;}
            else str="";
        } while(str.length());
    }
    g_flyengine->console_command="";
    // Calculo de dt
    int dt;
    cur_time=timeGetTime()-start_time;
    cur_time_float=cur_time/1000.0f;
    dt=cur_time-T0;
    if (dt>0){
        T0=cur_time;
        if (dt<1000){
            step(dt);
            frame_count++;
            if (cur_time-T1>500) {
                frame_rate=frame_count*1000/(cur_time-T1);
                T1=cur_time;
                frame_count=0;}}}
    // Funcion de simulacion de la consola en Fly3D
    if (g_flyengine->con.mode) step(dt);}
```

6. **Visualización**: la visualización de la consola se realiza de forma separada y posteriormente al resto de los objetos del sistema, en la función de visualización general *draw_frame*. La visualización sólo se modifica como proceso global, es decir se cambia cuando se visualizan todos los objetos, pero el proceso es el utilizado en Fly3D. Por ello, no debe hacerse ningún cambio específico para la visualización de la consola.

7. Eliminar código innecesario o que se ha trasladado a otras funciones:

   - La función *step* de la consola se invoca dentro de la propia consola.
   - Modificar la función de simulación general (*step* de *flyEngine*). En esta función:
     - Eliminar el proceso de la línea de comandos.
     - Eliminar la llamada a la función de simulación (función *step*) de la consola (si está activa).

8. Sustituir las llamadas directas a funciones de la consola por envío de mensajes (tabla E.1). Pasos:

- Obtener un mensaje de GDESK:

```
GDESK_MESSAGE *pMsg;
pMsg=g_flyengine->con.GDESK_GetMessageToFill();
```

- El mensaje *pMsg* debe rellenarse convenientemente según la función invocada, para que el mensaje contenga los parámetros de la anterior llamada a función. También debe indicarse el tipo del mensaje enviado para que la función *GDESK_ReceiveMessage* de la consola sea capaz de discriminar el tipo de mensaje recibido. Los mensajes se envían con tiempo cero, pues el proceso debe ser inmediato.
- Llamar a la función de envío de mensajes de la consola:

```
g_flyengine->con.GDESK_SendMessage
                (pMsg, g_flyengine->con, 0);
```

9. Decidir en que momento debe comenzar a funcionar la consola y enviar mensajes a la consola en esos puntos para que procese los comandos (hasta ahora esperaba al siguiente ciclo de simulación). Cada vez que se desea procesar un comando de consola, debe enviarse un mensaje a esta. En las siguientes funciones se llena *console_command* con mensajes de Fly3D. En todas ellas se debe procesar el comando enviando un mensaje a la consola.

   - Función *WndProc* de *flyFrontend*.
   - Función *step* de *menucamera*.
   - Función *step* de *trigger*.
   - Función *fly_message* de *multimedia*.
   - Función *auto_demo_playback* de *menu*.
   - Función *menu* de *gamemenu*.
   - Función *on_fly_files_change* de *gamemenu*.
   - Función *check_multiplayer* de *flyEngine*.
   - Función *play_demo* de *flyEngine*.

   En este caso el comando a ejecutar por la consola se encuentra en el atributo *console_command* de flyEngine, por lo que no debe copiarse el comando en el mensaje. Los pasos a seguir para enviar el mensaje son los mismos que en el caso de llamada directa. La diferencia es que el único valor que contiene el mensaje es el tipo de mensaje, que indica que debe procesarse los comandos contenidos en *console_command*.

```
pMsg->MessageType = GDESK_CONSOLE_MSG_SIMULATION;
```

Función / Módulo	**Llamada directa**	**Parámetros**
console_printf / flyEngine	con.add_string(ach)	pMsg→MessageType= GDESK_CONSOLE_MSG_ADDSTRING strcpy(pMsg→Command,ach)
WinMain / flyFrontend	g_flyengine→con. key_down(msg.wParam)	pMsg→MessageType= GDESK_CONSOLE_MSG_KEYDOWN pMsg→Key = msg.wParam
WinMain / flyFrontend	g_flyengine→con. key_char(msg.wParam)	pMsg→MessageType= GDESK_CONSOLE_MSG_KEYCHAR pMsg→Key = msg.wParam
WndCommand / flyFrontend	g_flyengine→con. command_exec("map menu.fly")	pMsg→MessageType= GDESK_CONSOLE_MSG_EXEC strcpy(pMsg→Command, "map menu.fly")

Tabla E.1: Cambio de llamadas a consola por mensajes en DFly3D

E.2. Objeto del Videojuego *Bola*

E.2.1. El Objeto *Bola* en Fly3D

Todos los objetos de Fly3D derivan de la clase base de los objetos de *flyEngine* (*flyBspObject*). Todos los objetos que el usuario define en sus aplicaciones están incluidos en el grafo de escena. Los cambios a realizar únicamente implican a este objeto.

El hecho de derivar del objeto base de Fly3D obliga a los objetos a redefinir una serie de funciones virtuales (apéndice D) que *flyEngine* invoca en los momentos apropiados. Estas funciones se pueden clasificar en categorías:

1. **Funciones de visualización**: estas funciones sirven para visualizar el objeto, para realizar la detección de colisiones,... Entre estas funciones cabe destacar la función *draw* que es la que permite visualizar el objeto.

2. **Funciones de comunicación**: es la función *message*, que envía y procesa los mensajes entre objetos (de tipo *msg*). Fly3D permite cierta comunicación entre objetos mediante paso de mensajes. Los mensajes definidos por Fly3D tienen una funcionalidad muy limitada, básicamente referidos a iluminación. El envío y la recepción de mensajes la modela el usuario cuando define los objetos.

3. **Funciones de simulación**:

   - *step*: realiza la simulación del objeto. Tiene como parámetro el tiempo transcurrido desde el último frame.
   - *post_step*: proceso extra que se realiza después de ejecutar la simulación de todos los objetos (función *step* de *flyEngine*).

4. **Varias**: son funciones de creación de una nueva instancia de objeto (*clone*) y de inicialización de un objeto (*init*).

Los cambios a realizar en las funciones son:

1. **Funciones de visualización**: no se va a modificar. No se modelan mediante mensajes, pues el proceso de visualización de los objetos no se va a modificar, se mantiene el mismo proceso de Fly3D. Sólo se modifica el instante en que se arranca la visualización, lo que no tiene nada que ver con el objeto *bola*.

2. **Funciones de comunicación**: estos mensajes deben integrarse con los mensajes de DFLy3D. Se sustituye cada uno de estos mensajes por su correspondiente en DFLy3D. El programador decide a que mensajes es sensible el objeto. En el caso del objeto *bola*, no es sensible a ninguno de estos mensajes.

3. **Funciones de simulación**: Estas funciones son las que se modifican más profundamente con la integración de Fly3D. La forma en que Fly3D simula supone que el objeto, dado un intervalo de tiempo definido por el resto de procesos del sistema, evoluciona para adaptarse a la nueva situación en ese intervalo. Todo el proceso de simulación debe modelarse mediante mensajes. El comportamiento de un objeto debe modelarse como una sucesión de mensajes a otros objetos o a él mismo. Por ello, esta función de simulación ya no se invoca desde el proceso general de simulación, sino que se ejecuta cada vez que le llegue un mensaje del tipo apropiado.

   - *step*: se debe adaptar la función para que no simule en el intervalo que se le ha pasado, sino que decida en que momento tiene que volver a enviarse un mensaje a si mismo para seguir funcionando.
   - *post_step*: la simulación ya no se divide en anterior y posterior, por lo que se debe aunar todo el proceso de simulación en una misma función que se invoque tantas veces como sea necesario. Por tanto, esta función ya no se utiliza.

4. **Varias**: no se modifican.

E.2.2. Simulación del Objeto *Bola* en DFLy3D

La única diferencia entre el objeto *bola* en Fly3D y DFly3D es la simulación del objeto. En Fly3D se define la función *step* y en DFLy3D la función *GDESK_ReceiveMessage*. Se ha optado por mantener la simulación del objeto bola en DFLy3D lo más parecida posible a DFLy3D. Por ello en la función *GDESK_ReceiveMessage* invoca a la función de simulación continua *step*, pero con el valor de tiempo del mensaje. Se mantiene la misma función, pero sólo se invoca cuando llega un mensaje. Los mensajes se envían con una cadencia constante que define el programador.

```
#define FPS_BOLA 30
#define TIEMPO_BOLA 1000/(float)FPS_BOLA

void Bola::GDESK_ReceiveMessage(GDESK_MESSAGE *pMsg) {
    switch ( pMsg->MessageType )
    {
    case GDESK_BOLA_MSG_SIMULATION:
        // Simulacion continua
        step((int)pMsg->Tiempo);
        g_flyengine->add_to_bsp(this);
        // Mensaje de nueva simulaci\'{o}n
        GDESK_MESSAGE *pMsg;
        pMsg = GDESK_GetMessageToFill();
        pMsg->m_eMessageType = GDESK_BOLA_MSG_SIMULATION;
        pMsg->m_Tiempo = TIEMPO_BOLA;
        GDESK_SendMessage(pMsg, *this, Tiempo_Proximo_Evento);
        break;
    case GDESK_BOLA_MSG_MSG:
        // Es una llamada a la funcion message de Fly3D
        message(pMsg->vector,pMsg->rad,pMsg->msg, pMsg->param,
                pMsg->data);
        break;
    }
}
```

E.3. Integración de GDESK en Aplicaciones Gráficas Continuas en Tiempo Real

Los cambios a realizar en una aplicación gráfica en tiempo real para integrar GDESK son altamente dependientes de la aplicación en concreto. Aspectos como

la separación del núcleo de las aplicaciones creadas o la modularización del sistema son muy importantes.

El primer paso para integrar GDESK en una aplicación gráfica es modularizar la propia aplicación. Deben crearse objetos que contengan la funcionalidad de diferentes aspectos del núcleo. Deben crearse tantos objetos del sistema como elementos del sistema se ejecutan durante el bucle principal de la aplicación. Es necesario, como mínimo un objeto visualizador que controle el proceso de renderizado y un objeto lanzador que arranque el comportamiento del resto de objetos. Pueden crearse cualquier número de objetos del sistema.

Los videojuegos creados usando la aplicación gráfica también deben modularizarse. Debe haber un objeto por cada componente del videojuego con un comportamiento dinámico en el sistema o con un comportamiento estático si es susceptible de interaccionar con otros elementos del videojuego.

Todos los objetos del sistema deben ser capaces de heredar del objeto base de GDESK para poder utilizar las funciones de recepción y envío de mensajes.

Los objetos deben definir su comportamiento en la función de recepción de mensajes. Es decir, cualquier comportamiento del objeto siempre debe arrancarse como consecuencia de la llegada de un mensaje. En ese momento el objeto puede modificar su estado y comunicarse con otros objetos.

Los mensajes dirigidos a los objetos pueden tener tiempo cero (procesos que se ejecutan instantáneamente) o pueden tener un tiempo asociado a dicho mensaje (en este caso el mensaje lo recibe el objeto receptor transcurrido este tiempo). Esto debe tenerse en cuenta a la hora de definir el comportamiento de los objetos.

Una vez se ha modularizado el sistema, el siguiente paso es modificar el bucle principal de la aplicación para que la simulación la controle GDESK (figura 4.5 del capítulo 4). Se eliminan las funciones de simulación y visualización del bucle principal y se sustituyen por la función de simulación de GDESK. El siguiente código muestra básicamente el bucle principal de GDESK.

```
while (1)
    {
    while (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE) == TRUE)
        {
        // Gestion de eventos de usuario
        }
    // Simulacion
    Intervalo_Siguiente_Mensaje = GDESK_gMsgDispatcher->Simulate();
    // Gestion del tiempo inactivo de GDESK
    GDESK_Delay(Intervalo_Siguiente_Mensaje);
```

```
}
```

La función de gestión del tiempo que el sistema está inactivo debe definirla el programador del núcleo de videojuegos para, por ejemplo, ceder el tiempo a otras aplicaciones. GDESK sólo incluye la posibilidad de realizar un bucle de espera durante este tiempo.

La integración de GDESK supone un cambio importante en la forma en la que el sistema se simula, sobre todo a la hora de implementar los objetos del videojuego. Los objetos no actúan como respuesta a un muestreo continuo, sino que son independientes y fijan su propia frecuencia de muestreo.

Referencias

[Abrash] M. Abrash. Ramblings in Realtime. http://www.bluesnews.com/abrash/.

[Adventure] The Adventure Collective. http://www.adventurecollective.com/.

[Agus:2002] M. Agus, A. Giachetti, E. Gobbetti, G. Zanetti. A multiprocessor decoupled system for the simulation of temporal bone surgery. *Computing and Visualization in Science*, 5(1), 2002.

[Albertsson:2001] L. Albertsson. Simulation-based debugging of soft real-time applications. In *Real-Time Application Symposium, IEEE Computer Society.* IEEE Computer Society Press, May 2001.

[Alexander:2003] T. Alexander, editor. *Massively multiplayer game development.* Charles River Media, Inc., 2003.

[Alois:1994] F. Alois, T. Satish. Parallel and distributed simulation of discrete event systems. Technical Report TCS-TR-3336, University of Maryland, 1994.

[Arena] Rockwell Software, Inc. http://www.arenasimulation.com/.

[Aronov:2002] B. Aronov, H. Bronnimann, A.Y. Chang, Y.J. Chiang. Cost prediction for ray shooting. In *ACM Symp. on Computational Geometry (SoCG '02)*, pages 293–302, 2002.

[Ars] Agricultural Research Service. http://www.nal.usda.gov/ttic/ tektran/data/000009/83/0000098344.html/.

[Artist] Proyecto ARTIST. http://robotica.uv.es/grupos/artec/Spanish/ proyectartist.html/.

[AutoMod] Brooks Automation. http://www.automod.com/.

[Baer:1999] R. Baer. How video games invaded the home tv set. http://www.ralphbaer.com/how_video_games.htm/.

[Bagrodia:1994] R. Bagrodia. Maisie: A language for the design of efficient discrete-event simulations. *IEEE Transactions on Software Emgineering*, 1994.

[Banks:2001] J. Banks, J.S. Carson II, B.B. Nelson, D.M. Nicol. *Discrete-Event System Simulation.* Prentice Hall International Series in Industrial and Systems Engineering, 2001.

[Bertallanfy:1968] L. von Bertallanfy. *General System Theory. Foundations, Development, Applications.* Braziller, New York, 1968.

[Bishop:1998] L. Bishop, D. Eberly, T. Whitted. Designing a PC game engine. *IEEE C&GA*, 18(1), 1998.

[Bloechle:1999] W.K. Bloechle, K.R. Laughery. Simulation interoperability using MicroSaint simulation software. *Proceedings of the Winter Simulation Conference*, pages 286–288, 1999.

[Bryant:1980] R.M. Bryant. SIMPAS – a simulation language based on PASCAL. *Proceedings of the 1980 Winter Simulation Conference.*

[Buxton:1966] J.N. Buxton. Writing simulations in CSL. *The Computer Journal*, 1966.

[BuyersGuide:1999] Simulation software buyer's guide. *Industrial Engineering*, 1999.

[CStory] Computer Story. http://www.comsto.org/.

[Caci] CACI Products Company. http://www.caciasl.com/.

[Caveut] CaveUT. http://planetjeff.net/ut/CaveUT_1.2.html/.

[Cdx] CDX Library. http://www.cdxlib.com/.

[Cfx] CFXweb Demo and Game Development. http://www.cfxweb.net/.

[Cgri] Computer Games Research Institute, Shizuoka University, Hamamatsu, Japan. http://www.cs.inf.shizuoka.ac.jp/ iida/CGRI/CGRI.html/.

[Cleary:1988] J.G. Cleary, G. Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. In *The Visual Computer*, volume 4, pages 65–83, 1988.

[Clementson:1966] A.T. Clementson. Extended control and simulation language. *The Computer Journal*, 9(3):215–220, 1966.

[Conitec] Conitec Datasystems. http://www.conitec.net/.

[Coos:1992] R. Coos. *Simulación: un enfoque práctico.* Limusa, Mexico, 1992.

[Corbacho:1997] M.I. Corbacho. Ampliación de una herramienta de simulación distribuida bajo el paradigma de DEVS. Master's thesis, Facultad de Informática, Universidad Politécnica de Valencia, 1998.

[Cowell:2000] J. Cowell. *HTML and XHTML: the definitive guide*. Springer, 2000.

[Cox:1984] S. Cox. The user interface of GPSS/PC. In *Proceedings of the 16th conference on Winter simulation*, pages 544–551, 1984.

[Crain:1999] R.C. Crain, J.O. Henriksen. Simulation using GPSS/H. *Proceedings of the Winter Simulation Conference*, pages 182–187, 1999.

[Croteam] Croteam. http://www.croteam.com/.

[CrystalSpace] Crystal Space. http://crystal.sourceforge.net/drupal/.

[Darken:1995] R. Darken, C. Tonnesen, K. Passarella. The bridge between developers and virtual environments: a robust virtual environment system architecture. *SPIE*, 1995.

[Davies:1989] R.M. Davies, R.M. O'Keefe. *Simulation modelling with PASCAL*. Prentice-Hall, 1989.

[Dcra] Decisioncraft Analytics. http://www.decisioncraft.com/.

[Delfose:1976] C.M. Delfose. Continuous simulation and combined simulation in SIMSCRIPT II.5. *CACI*, 1976.

[Devx] DevX.com. http://www.devx.com/DevX/HTML/11610/.

[Dibble:2002] P.C. Dibble. *Real-time Java: platform programming*. Sun Microsystems, 2002.

[Donald:1998] D.L. Donald. Tutorial of ergonomic and process modeling using QUEST and IGRIP. *Proceedings of the Winter Simulation Conference*, pages 297–302, 1998.

[DoomW] Doom World. http://doomworld.com/.

[DotEaters] The Dot Eaters, Videogame History 101. http://www.emuunlim.com/doteaters.

[Epic] Epic Games. http://www.epicgames.com/.

[Ferrante:1994] P. Ferrante, P. Mussi, G. Siegel, L. Mallet. Object oriented simulation: Highlights on the PROSIT parallel discrete event simulator. *Western Simulation Conference*, 1994.

[Fishman:1978] G.S. Fishman. *Conceptos y Métodos en la Simulación Digital de Eventos Discretos*. Limusa, Mexico, 1978.

[Fishwick:1995] P. Fishwick. Sim++ user manual. *Universidad de Florida*, 1995.

[Fishwick:1996] P.A. Fishwick. Web-based simulation: Some personal observations. *28th Winter Simulation Conference*, 1996.

[Fly3D] FLY3D Page. http://www.fly3d.com.br/.

[Flyer] The Arcade Flyer Archive. http://www.arcadeflyers.com/.

[Forrester:1970] J.W. Forrester. *World Dynamics*. Wright-Allen Press,Cambridge, 1970.

[Frecon:1998] E. Frecon, M. Stenius. DIVE: A scaleable network architecture for distributed virtual environments. *Distributed Systems Engineering*, 5(3):91–100, 1998.

[Gamasutra] Gamasutra. http://www.gamasutra.com/.

[Gamebots] Gamebots. http://gamebots.sourceforge.net/.

[Gamedev] Gamedev.net. http://www.gamedev.net/.

[Gameprogrammer] Gameprogrammer.com. http://www.gameprogrammer. com/.

[Gametutorials] Game Tutorials. http://www.gametutorials.com/.

[Gamezone] Gamezone On-Line. http://www.gamezone.com/.

[Garagegames] Garagegames. http://www.garagegames.com/.

[Garcia:1997] I. García. Simulador generalista de sucesos discretos orientado a eventos. Master's thesis, Facultad de Informática, Universidad Politécnica de Valencia, 1997.

[Garcia:2000] I. García, R. Mollá, E. Ramos, M. Fernández. D.E.S.K.: Discrete events simulation kernel. *ECCOMAS*, 2000.

[Garcia:2002] I. García, R. Mollá. Discrete events simulation as a computer game kernel. *Grupo Portugues de Computacao Grafica. Virtual Journal junio 2002, http://virtual.inesc.pt*, 2002.

[Garcia:2003] I. García, R. Mollá, J. Cabanillas. JDESK web-based discrete event simulation kernel. *Menu Conference*, 2003.

[Garcia:2004] I. García, R. Mollá, A. Barella. GDESK: Game discrete event simulation kernel. *WSCG*, 2004.

[Garciab:2003] I. García, R. Mollá. JDESK: Simulador de eventos discreto basado en web. *Jenui*, 2003.

[Garciab:2004] I. García, R. Mollá. Making discrete games. *Computational Science and its Applications - ICCSA'2004 LNCS*, 3045:877–885, 2004.

[Garciac:2004] I. García, R. Mollá, P. Morillo. From continuous to discrete games. In IEEE Computer Society Press, editor, *2004 Computer Graphics International (CGI'04)*, pages 626–630, 2004.

[Garciad:2004] I. García, R. Mollá. Simulación desacoplada de eventos discretos en videojuegos. In *CEIG*, 2004.

[Garciae:2004] I. García, R. Mollá, E. Camahort. Introducing discrete simulation into games. *European Research Consortium for Informatics and Mathematics. ERCIM News, Abril*, 57:45–46, 2004.

[Garrison:1991] W.J. Garrison. NETWORK II.5, LANNET II.5 and COMNET II.5. In *Proceedings of the 23rd conference on Winter simulation*, pages 72–76. IEEE Computer Society, 1991.

[Gdconf] Game Developers Conference. http://www.gdconf.com/.

[Gdmag] Game Developers Magazine. http://www.gdmag.com/homepage. htm/.

[Genesis3D] Genesis3D. http://www.genesis3d.com/.

[Gerstmann:1999] J.Gerstmann. Unreal tournament: Action game of the year. *GameSpot*, 1999.

[Gnu] The GNU Proyect. http://www.gnu.org/.

[Gobbetti:1995] E. Gobbetti, J.F. Balaguer. An integrated environment to visually construct 3D animations. *SIGGRAPH Conference Proceedings, Annual Conference Series. ACM SIGGRAPH, Addison-Wesley*, 1995.

[Goble:1991] J. Goble. Introduction to SIMFACTORY II.5. In *Proceedings of the 23rd conference on Winter simulation*, pages 77–80. IEEE Computer Society, 1991.

[Gomes:1995] F. Gomes, J. Cleary, A. Covington, S. Franks, B. Unger, Z. Ziao. SimKit: a high performance logical process simulation class library in C++. *Proceedings of the 27th Winter simulation Conference*, 1995.

[Gossweiler:1994] R. Gossweiler, C. Long, S. Koga, R. Pausch. DIVER: A distributed virtual environment research platform. *IEEE Symposium on Research Frontiers in Virtual Reality*, 1994.

[Gpss1100a:1971] UNIVAC 1100. UNIVAC 1100 series general purpose simulator (GPSS 1100) programmer reference. Technical Report UP-7883, 1971.

[Gpss1100b:1971] UNIVAC 1100. UNIVAC 1100 general purpose systems simulator H reference manual. Technical Report UP-4129, 1971.

[Gpssv6000:1975] GPSS V/6000 general information manual. Technical Report 84003900, Conaol Data Corporation, 1975.

[Gpssv:1970] *General Purpose Simulation System V, User's Manual*. IBM Publications, 1970.

[Greenberger:1965] M. Greenberger, M.M. Jones, H. Morris, D.N. Ness. *On-line computation and simulation: The OPS-3 system*. MIT Press, 1965.

[Greenhalgh:1998] C. Greenhalgh. Awareness-based communication management in the MASSIVE systems. *Distributed Systems Engineering*, 5(3):129–137, 1998.

[Haggar:2000] P. Haggar. *Practical Java : programming language guide*. Addison-Wesley, 2000.

[Healy:1998] K.J. Healy, R.A. Kilgore. Introduction to Silk and Java-based simulation. *30th Winter Simulation Conference*, 1998.

[Henriksen:1985] J.O. Henriksen. The development of GPSS/85. In *Proceedings of the 18th annual symposium on Simulation*, pages 61–77. IEEE Computer Society Press, 1985.

[Hernandez:2002] E. Hernández, J. Hernández, M.C. Juan. *C++ Estándar*. Paraninfo, 2002.

[Howell:1998] F. Howell, R. McNab. Simjava: a discrete event simulation package for java with applications in computer systems modelling. *First International Conference on Web-based Modelling and Simulation*, 1998.

[Idevgames] iDevGames. http://www.idevgames.com/.

[Idsw] Idsoftware. www.idsoftware.com/archives/doomarc.html/.

[ImagineThat] Imagine That, Inc. http://www.imaginethatinc.com/.

[Jacobson:2002] J. Jacobson, Z. Hwang. Unreal tournament for immersive interactive theater. *Communications of the ACM*, 45:39–45, 2002.

[Jain:1991] R. Jain. *The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation, and modeling*. John Wiley and Sons, Inc., New York, 1991.

[Jdesk] Página Principal de JDESK (SIG, UPV). http://www.sig.upv.es/proyectos/simulacion/JDESK.htm/.

[Jones:1967] M.M. Jones. *Incremental Simulation an a Time-Shared Computer*. PhD thesis, Massachusetts Institute of Technology, 1967.

[Jsim] JSIM. http://chief.cs.uga.edu/ jam/jsim/.

[Jsimg] User Guide to JSIM. http://chief.cs.uga.edu/ jam/home/theses/ huang_thesis/userguide/userguide.html/.

[Kaminka:2002] G.A. Kaminka, M.M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A.N. Marshall, A. Scholer, S. Tejada. Gamebots: A flexible test bed for multiagent team research. *Communications of the ACM*, 45:43–45, 2002.

[Kilgore2:2001] R.A. Kilgore. Open-source SML and Silk for java-based object-oriented simulation. *Winter Simulation Conference*, 2001.

[Kilgore:1998] R.A. Kilgore, K.J. Healy, G.B. Kleindorfer. The future of Java-based simulation. *30th conference on Winter simulation*, 1998.

[Kilgore:2001] R.A. Kilgore. Open-source simulation modelling language (SML). *Winter Simulation Conference*, 2001.

[Kingston:1990] J.H. Kingston. *Algorithms and Data Structures*. Addison-Wesley, 1990.

[Kiviat:1963] P.J. Kiviat. GASP – A general activity simulation program. *Project No. 90.17-019(2), Applied Research Lab., United States Steel Corp., Monroeville, PA*, 1963.

[Klein:1970] K.I. Klein. *Modeling Stochastic Systems with GERTS*. Unpublished undergraduate thesis, Lehigh University, 1970.

[Klein:1993] M. Klein. *Guía de programación en Windows DLL y gestión de memoria*. Anaya Multimedia, D.L., 1993.

[Krahl:1999] D. Krahl. Modeling with Extend^{TM}. *Proceedings of the Winter Simulation Conference*, pages 188–195, 1999.

[Krasnow:1964] H.S. Krasnow, R.A. Merikallio. The past, present and future of general simulation languages. *Management Science*, 11(2):236–267, 1964.

[Kuljis:2000] J. Kuljis, R.J. Paul. A review of web based simulations: Whither we wander? *Winter Simulation Conference*, 2000.

[Kuljis:2003] J. Kuljis, R.J. Paul. Web-based discrete event simulation models: current status and posible futures. *Simulation and Gaming*, 34(1):39–53, 2000.

[Lakshmanan:1983] R. Lakshmanan. *Design and implementation of a PASCAL based interactive network simulation language for microcomputers.* PhD thesis, Ockland University, Rochester, Michigan, 1983.

[Lanner] The Lanner Group. http://www.lanner.com/.

[Law:1982] A.M. Law, W.D. Kelton. *Simulation Modeling and Analysis.* McGraw-Hill Series in Industrial Engineering and Management Science, 1982.

[Lee:1999] G.S. Lee. Towards an integration of computer simulation with computer graphics. *Proceedings of the Western Computer Graphics Symposium*, 1999.

[Lewis:1991] J.B. Lewis, L. Koved, D.T. Ling. Dialogue structures for virtual worlds. *CHI*, 1991.

[Lewis:2002] M. Lewis, J. Jacobson. Game engines in scientific research. *Comunications of the ACM*, 45(1):27–31, 2002.

[Lilja:2000] D.J. Lilja. *Measuring computer performance. A practitioner's guide.* Cambridge University Press, 2000.

[Lindholm:1996] T. Lindholm, F. Yellin. *The JavaTM Virtual Machine Specification.* Addison-Wesley, 1996.

[Little:2003] R. Little. Architectures for distributed interactive simulation. *Software Engineering Institute, Carnegie Mellon University (Pittsburgh, Pennsylvania) http://www.sei.cmu.edu/publications/articles/arch-dist-int-sim.html/*, 2003.

[Loukides:1992] M. Loukides. *System performance tuning.* O'Reilly & Associates, Inc., 1992.

[Lucia:2003] R.A. Lucia. Estudio y desarrollo de un sistema sensor para simulaciones 3d multipersonaje sobre el motor gráfico del unrealtournament. Master's thesis, Escuela Técnica Superior de Ingeniería, Universitat de València, 2003.

[MacDonald:1990] J.D. MacDonald, K.S. Booth. Heuristics for ray tracing using space subdivision. In *The Visual Computer*, volume 6, page 153166, 1990.

[MacDougal:1980] M.H. MacDougal. *SMPL - A Simple Portable Simulation Language.* Amdahl, 1980.

[MacDougal:1987] M.H. MacDougall. *Simulating Computer Systems - Technique and Tools*. MIT Press, Cambridge, 1987.

[Macedonia:1997] M.R. Macedonia. A taxonomy for networked virtual environments! *IEEE Multimedia*, 4(1):48–56, 1997.

[Markowitz:1963] H.M. Markowitz. *SIMSCRIPT: A Simulation Programming Language*. The RAND Corporation, Prentice Hall, Inc., 1963.

[Markowitz:1979] H.M. Markowitz. SIMSCRIPT: past, present and some thoughts about the future. *Current issues in Computer Simulation*, 1979.

[Marven:1994] C. Marven, G. Ewers. *A Simple Aproach to Digital Signal Processing*. Texas Instruments, 1994.

[McNab:1996] R. McNab, F.W. Howell. Using java for discrete event simulation. *UK Computer and Telecommunications Performance Engineering Workshop (UKPEW)*, 1996.

[Mehta:1999] A. Mehta, I. Rawles. Business solutions using WITNESS. *Proceedings of the Winter Simulation Conference*, pages 230–233, 1999.

[Mesquite] Mesquite Software, Inc. http://www.mesquite.com/.

[MicroSaint] Micro Analysis & Design. http://www.maad.com/.

[Miller:1997] J.A. Miller, R.S. Nair, Z. Zhang, H. Zhao. A java-based simulation and animation environment. *30th Annual Simulation Symposium (SS '97)*, 1997.

[Miller:2000] J.A. Miller, A.F. Seila, X. Xiang. The JSIM web-based simulation environment. *Future Generation Computer Systems*, 17(2):119–133, 2000.

[Misra:1986] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys (CSUR)*, 18(1):39–65.

[Molla:2001] R. Mollá. *Aplicaciones de la Aritmética en coma fija a la representación de primitivas gráficas de bajo nivel*. PhD thesis, Universidad Politécnica de Valencia, 2001.

[Mueller:1997] F. Mueller, D.B. Whalley. On debugging real-time applications. Dept. of Computer Science. Florida State University. http://cis.famu.edu/ harmon/papers/lcts9402/, 1997.

[Musciano:2002] C. Musciano, B. Kennedy. *Essential Java 2 fast: how to develop applicatlions and applets with Java 2*. O'Reilly, 2002.

[Nahrstedt:1996] K. Nahrstedt, L. Qiao. A tuning system for distributed multimedia applications. Technical Report UIUCDCS-R-96-1958, University of Illinois at Urbana-Champaign Computer Science Department, July 1996.

[Nance:1993] R.E. Nance. Simulation programming languages: an abridged history. *Winter Simulation Conference Proceedings*, 1995.

[Nance:1995] R.E. Nance. A history of discrete event simulation programming languages. *ACM SIGPLAN Notices*, 28(3):149–175, 1993.

[Naylor:1975] T.H. Naylor, J.L. Balintfy, D.S. Burdick, K. Chu. *Técnicas de Simulación en computadoras*. Limusa, Mexico, 1975.

[Ngpss:1971] User's guide to NGPSS. Technical Report Norden Report 4339 R 0003, Norden Division of United Aircraft Corporation, 1971.

[Nis] Neurosurgery Image Server, Universidad de Manchester. http://synaptic.mvc.mcc.ac.uk/simulators.html.

[Nygaard:1981] K. Nygaard, O. Dahl. The deveolpment of the SIMULA languages. *History of Programming Languages, Academic Press*, pages 439–491, 1981.

[Ogata:2003] K. Ogata. *Ingeniería de control moderna*. Pearson Educación, 4 edition, 2003.

[Ogre] The Ogre Team. http://ogre.sourceforge.net/.

[Osg] Open Scene Graph. http://openscenegraph.sourceforge.net/.

[Overeinder:1991] B.J. Overeinder, L.O. Hertzberger, P.M.A. Sloot. Parallel discrete event simulation. *The Third Workshop Computersystems, Faculty of Electrical Engineering, Eindhoven University*, pages 19–30, 1991.

[Paralelo] Paralelo Computação. http://www.paralelo.com.br/en/index.php.

[Pausch:1994] R. Pausch, C. Conway, R. DeLine, R. Gossweiler, S. Miale. Alice & DIVER: A software architecture for the rapid prototyping of virtual environments. *Course notes for SIGGRAPH course, Programming Virtual Worlds*, 1994.

[Pausch:1995] R. Pausch, T. Burnette, A.C. Capehart, M. Conway, D. Cosgrove, R. DeLine, J. Durbin, R. Gossweiler, S. Koga, J. White. A brief architectural overview of Alice, a rapid prototyping system for virtual environments. *IEEE Computer Graphics and Applications*, 1995.

[Pedgen:1990] C.D. Pegden, R.E. Shannon, R.P. Sadowski. *Introduction to Simulation using SIMAN*. McGraw Hill, 1990.

[Pergsguide] OpenGL Performer Getting Started Guide. http://www.cineca.it/manuali/Performer/GetStarted/ html/index.html.

[Perguide] OpenGL Performer Programmer's Guide. http://www.cineca.it/manuali/Performer/ProgGuide24/html/.

[Planetarium] Planetarium Software. http://www.seds.org/billa/ astrosoftware.html/.

[PongStory] Pong Story, The site of the first videogame. http://www.pongstory.com/.

[Potier:1984] D. Potier. New usersíntroduction to QNAP2. Technical report, 1984.

[Ppl] Parallel Programming Laboratory, Illinois University. http://charm.cs.uiuc.edu/ppl_research/pose/.

[Preiss:1989] B.R. Preiss. The Yaddes distributed event simulation specification language and execution environments. *SCS Multiconference on Distributed Simulation*, 1989.

[Price:1999] R.N. Price, C.D. Pegden. Simulation modeling and optimization using ProModel. *Proceedings of the Winter Simulation Conference*, pages 208–214, 1999.

[Pritsker:1969] A. Pritsker, B. Alan, P.J. Kiviat. *Simulation with GASP II: A FORTRAM based simulation language.* Prentice Hall, 1969.

[Pritsker:1974] A.A.B. Pritsker. *The GASP IV Simulation Language.* John Wiley and Sons, Inc., New York, 1974.

[Pritsker:1975] A. Pristker, B. Alan, R.E. Young. *Simulation with GASP_PL/I.* John Wiley, 1975.

[Pritsker:1979] A. Pristker, B. Alan, C.D. Pegden. *Introduction to Simulation and SLAM.* John Wiley, 1979.

[Pritsker:1995] A. Pritsker. *Introduction to Simulation and SLAM II.* John Wiley, 1995.

[ProModel] ProModel Corporation. http://www.promodel.com/.

[Purdue] Purdue University. http://www.cs.purdue.edu/research/PaCS/ parasol.html/.

[Qnap] Simulog Technologies. http://www.simulog.fr/formation/ qnam.htm.

[Qnap:1990] QNAP: queuing network analysis package. *1st Int. Conf. on the numerical Solutions of Markov chains*, pages 684–686, 1990.

[Quake] Quake Developers Page. www.gamers.org/dEngine/quake/.

[Quest] Delmia. http://www.delmia.com/.

[Raczynski] Raczynski. http://www.raczynski.com/pn/pn.htm/.

[RadonLabs] Radon Labs. http://www.radonlabs.de/.

[Reinhard:1996] E. Reinhard, A.J.F. Kok, F.W. Jansen. Cost prediction in ray tracing. In *Rendering Techniques*, pages 41–50. Springer-Verlag, 1996.

[Reinhard:1998] E. Reinhard, A.J.F. Kok, A. Chalmers. Cost distribution prediction for parallel ray tracing. In *Proceedings of the Second Eurographics Workshop on Parallel Graphics and Visualisation*, pages 77–90. Eurographics, 1998.

[Reitman:1967] J. Reitman. The user of simulation languages - the forgotten man. *ACM NAtional Meeting*, 1967.

[Reitman:1970] J. Reitman, D. Ingerman, J. Katzke, J. Shapiro, K. Simon, B. Smith. A complete interactive simulation environment GPSS/360-Norden. *Proceedings of the Fourth Annual Conference on Applications of Simulation*, 1970.

[Reynolds:2000] C. Reynolds. Interaction with groups of autonomous characters. *Game Developers Conference Proceedings*, 2000.

[Roberts:1983] S.D. Roberts. Simulation modeling and analysis with INSIGHT. *Proceedings of the Winter Simulation Conference*, pages 7–16, 1983.

[Robertson:1989] G.G. Robertson, S.K. Card, J.D. Mackinlay. The cognitive coprocessor architecture for interactive user interface. *UIST*, 1989.

[Rohrer:1999] M. Rohrer. AutoMod product suite tutorial (AutoMod, Simulator, AutoStat) by AutoSimulations. *Proceedings of the Winter Simulation Conference*, pages 220–226, 1999.

[Rubin:1981] J. Rubin. Imbedding GPSS in a general purpose programming language. In *Proceedings of the 13th conference on Winter simulation*, pages 33–40, 1981.

[Russell:1983] E.C. Russell. Building simulation models with SIMSCRIPT II.5. Technical report, CACI Inc., 1983.

[Sadowski:1999] D. Sadowski, V. Bapat. The Arena product family: enterprise modeling solutions. *Proceedings of the Winter Simulation Conference*, pages 159–166, 1999.

[Schildt:2000] H. Schildt. *C Manual de referencia*. Osborne/McGraw-Hill, 2000.

[Schildt:2001] H. Schildt. *C++ guía de autoenseñanza*. Osborne/McGraw-Hill, 2001.

[Schmidt:1980] B. Schmidt. *GPSS-FORTRAN: Wiley Series in Computing*. John Wiley & Sons, Inc., 1980.

[Schriber:1999] T.J. Schriber, D.T. Brunner. Inside discrete-event simulation software: how it works and why it matters. *Proceedings of the 1999 Winter Simulation Conference*, 1999.

[Schwetman:1987] H. Schwetman. CSIM: A C-based process oriented simulation language. *Proceedings of the Winter Simulation Conference*, pages 387–396, 1987.

[Schwetman:1996] H. Schwetman. CSIM18 – the simulation engine. *Proceedings of the Winter Simulation Conference*, pages 387–396, 1996.

[Sedgewick:1998] R. Sedgewick. *Algorithms in C*, volume 1-4. Addison-Wesley, 1998.

[Sgi] SGI. http://www.sgi.com/software/performer/.

[Sgiwp] SGI White Paper. OpenGL Performer Real-Time 3D Rendering for High-Performance and Interactive Graphics Applications.

[Shark3D] Shark3D TM. http://www.shark3d.com/.

[Shaw:1992] C. Shaw, J. Liang, M. Green, Y. Sun. The decoupled simulation model for virtual reality systems. *CHI*, 1992.

[Sheridan:1984] T.B. Sheridan. Supervisory control of remote manipulators, vehicles and dynamic processes. *Advances in Man-Machine Systems Research, JAI Press*, 1, 1984.

[Sim] Sim++. http://www.simplusplus.com/introduction.html/.

[Simgear] SimGear - Simulator Construction Tools. http://www.simgear.org/.

[Simjava] Institute for Computing Systems Architecture. Division of Informatics, University of Edinburgh. http://www.dcs.ed.ac.uk/home/hase/simjava/.

[Simpack] SIMPACK. http://www.simpack.de/websitep.html/.

[SimplI:1972] IBM Corporation. SIMPL/I (simulation language based on PL/I): Program reference manual. Technical Report SH19-5060-0, IBM Corporation Data Processing Division, 1972.

[Simulog] Simulog. http://www.simulog.fr/.

[Smed:2001] J. Smed, T. Kaukoranta, H. Hakonen. Aspects of networking in multiplayer computer games. In *Proceedings of International Conference on Application and Development of Computer Games in the 21st Century*, 2001.

[Smith:2000] R.D. Smith. Simulation article. *Encyclopedia of Computer Science http://www.modelbenders.com/encyclopedia/*, 2000.

[Smpl] EESA, Universidad de Gent, Bélgica. http://www.autoctrl.rug.ac.be/ftp/smpl/.

[Sourceforge] Sourceforge.net Open Source Software Development Website. http://sourceforge.net/.

[Spades] Spades. http://spades-sim.sourceforge.net/.

[Stahl:1996] I. Ståhl. Simulation made simple using micro-GPSS. *SSE*, 1996.

[Stahl:2001] I. Ståhl. GPSS - 40 years of development. *Proceedings of the Winter Simulation Conference*, 2001.

[Stahl:2002] I. Ståhl. A Web-based distance learning system in business administration experiences from an Inter-Nordic course. *SSE/EFI Working Paper Series in Business Administration No 2002:4*, 2002.

[Stroustrup:2001] B. Stroustrup. *El Lenguaje de Programación C++*. Addison Wesley, 2001.

[Subramanian:1991] K.R. Subramanian, D.S. Fussell. Automatic termination criteria for ray tracing hierarchies. In *Graphics Interface*, page 9310, 1991.

[Swain:1999] J.J. Swain. Simulation software survey. *OR/MS Today*, 26(3), 1999.

[Teichroew:1966] D. Teichroew, J.F. Lubin. Computer simulation - discussion of the technique and comparison of languages. *Comunications of the ACM*, 9(10), 1966.

[Tocher:1960] K.D. Tocher, D.G. Owen. The automatic programming of simulations. *Proceedings of the Second International Conference on Operational Research*, 1960.

[TomHw] Tom's Hardware Guide. www6.tomshardware.com/graphic/02q1/020304/geforce4-09.html/.

[Treglia:2002] D. Treglia, editor. *Game programming gems 3*. Charles River Media, Inc., 2002.

[Unrt] Unreal Tournament Page. http://www.unrealtournament.com/.

[Uyeno:1980] D.H. Uyeno, W. Vaessen. PASSIM a discrete-event simulation package for PASCAL. *Simulation*, 35(6):183–190, 1980.

[Vega] Multigen Paradigm. http://www.multigen-paradigm.com/products/runtime/vega/index.shtml.

[Venners:1999] B. Venners. *Inside the Java virtual machine*. McGraw-Hill, 1999.

[Veran:1984] M. Veran, D. Potier. QNAP2: A portable environment for queuing systems modelling. Rapports de recherche 314, INRIA (Institut National de Recherche en Informatique et en Automatique), Rocquencourt, France, 1984.

[Videogames] Videogames.org. http://www.videogames.org.

[Videopatia] Videopatia. The Electronic Conservance Inc. http://www.videotopia.com/.

[Wangerin:2002] D. Wangerin, C. DeCoro, L. Campos, H. Coyote, I. Scherson. A modular client-server discrete event simulator for networked computers. *35th Annual Simulation Symposium*, 2002.

[Watt:2001] A. Watt, F. Policarpo. *3D Computer Games Technology: Real-Time Rendering and Software*, volume I. Addison-Welsey, 2001.

[Watt:2003] A. Watt, F. Policarpo. *3D Computer Games*, volume II. Addison-Wesley Publishing, 2003.

[Webgpss] WebGPSS. http://webgpss.hk-r.se/ENG/background.asp.

[Weisman:2000] E. Weisman. WaterSim: A computer simulator and integrated educational module for semiconductor manufacturing systems. Technical Report U.G. 2000-5, The Institute for System Research, Universidad de Maryland, 2000.

[Wexelblat:1981] R.L. Wexelblat. *History of Programming Languages*. ACM Monograph Series, 1981.

[Whang:1995] K.Y. Whang, J.W. Song, J.W. Chang, J.Y. Kim, W.S. Cho, C.M. Park, I.Y. Song.

[Wimmer:2003] M. Wimmer, P. Wonka. Rendering time estimation for real-time rendering. In *Proceedings of the 14th Eurographics workshop on Rendering*, pages 118–129. Eurographics Association, 2003.

[Wolverine] Wolverine Software Corporation. http://www.wolverinesoftware.com/.

[Yaddes] Yaddes. http://www.brpreiss.com/page75.html/.

[Young:1963] K. Young. A user's experience with three simulation languages (GPSS, SIMSCRIPT and SIMPAC). Technical Report TM-1755/000/00, System Development Corporation, 1963.

[Yucesan:2002] E. Yücesan, C.H. Chen, J.L. Snowdon, J.M. Charnes. COST: A component-oriented discrete event simulator. *Winter Simulation Conference*, 2002.

[eM-Plant] eM-Plant. http://www.eM-Plant.de/.