Simulación Desacoplada de Eventos Discretos en Videojuegos

Inmaculada García y Ramón Mollá
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera S/N, 46022 Valencia
e-mail: ingarcia@dsic.upv.es

Resumen

Las aplicaciones gráficas en tiempo real y en concreto los videojuegos, utilizan un paradigma de simulación continuo donde se acoplan las fases de visualización y simulación. Este paradigma puede ser ineficiente (repartiendo inadecuadamente la potencia de cálculo de la máquina) o producir simulaciones incorrectas (orden incorrecto en la ejecución de eventos). El uso de un paradigma de simulación discreto y desacoplado, evita simulaciones incorrectas, además de mejorar la calidad y la eficiencia de la simulación. GDESK es un núcleo de simulación discreto desacoplado que puede integrarse en cualquier motor de videojuegos o aplicación gráfica en tiempo real. Una vez GDESK se integra dentro del videojuego, el videojuego puede verse como un conjunto de objetos comunicándose mediante paso de mensajes, modelados mediante eventos discretos. GDESK gestiona el proceso de intercambio de mensajes, sincronización de eventos, generación, envío,...

Palabras clave: simulación, eventos discretos, videojuegos.

1. Introducción

Los videojuegos tradicionales, como Doom v1.1 [11], Quake v2.3 [14] o Fly3D [22, 23] siguen un paradigma de simulación continuo acoplado. Los núcleos de videojuegos (como 3D GameStudio [4], Crystal Space [5] o Genesis 3D [10]), o bien siguen el mismo esquema de simulación o son únicamente núcleos de visualización. El bucle principal de estas aplicaciones tiene tres fases principales [12]:

1. **Comprobar si hay eventos de usuario**.
2. **Simular**: supone evolucionar todos los elementos del sistema un incremento de tiempo no definido pero igual para todos los objetos del videojuego. Simular

supone recorrer el grafo de escena preguntando a cada objeto si tiene algo pendiente de simular. Algunos juegos aprovechan el primer recorrido del grafo de escena para ir seleccionando los objetos que deben visualizarse, incluyéndolos en alguna clase de estructura de datos. De este modo, la visualización requiere únicamente recorrer esta estructura de datos secundaria, disminuyendo el coste del proceso de visualización. Este muestreo produce en el sistema un esquema de simulación continua [2]. El ciclo de simulación o de muestreo es el intervalo de tiempo entre dos simulaciones consecutivas por lo que este ciclo puede ser diferente en cada iteración del bucle principal.

3. **Visualizar** la escena actual. Siempre debe visualizarse la escena en cada paso de simulación (acoplo de las fases de simulación y visualización).

1.1. Inconvenientes de la Simulación Continua

El muestreo implica acceder a todos los objetos del grafo de escena, independientemente de que tengan eventos pendientes o no. La prioridad de los objetos depende de su situación en el grafo de escena. Por tanto, los eventos no se ejecutan ordenados temporalmente. Se ejecutan en el orden en que los objetos están situados en el grafo de escena (posibilidad de simulaciones incorrectas). La frecuencia de muestreo es común a todos los objetos del sistema, no se adapta a las necesidades individuales de cada objeto.

1.2. Inconvenientes del Acople de las Fases de Simulación y Visualización

Para cada ciclo de simulación debe visualizarse la escena, independientemente de que vaya a ser mostrada en pantalla o no. La frecuencia de muestreo de los objetos es altamente dependiente de la potencia de cálculo de la máquina y de los costes temporales de ambas fases. Todos los eventos se sincronizan con el periodo de muestreo, por lo que, el sistema no es sensible a tiempos inferiores al periodo de muestreo.

2. Propuesta de Mejora

Para evitar los inconvenientes la simulación continua acoplada, se propone cambiar el paradigma de simulación a un esquema discreto desacoplado. Para ello se elige un simulador de eventos discretos y se integra en un núcleo de videojuegos. Con esto se consigue discretizar el núcleo de videojuegos, permitiendo desacoplar las fases de simulación y visualización.

2.1. Simulación de Eventos Discretos

Los simuladores de eventos discretos aparecen en la década de los 50, como una forma de analizar problemas basados en la teoría de colas [7]. Un simulador de eventos discretos permite modelar comportamientos discretos, continuos e híbridos [2]. Los simuladores de eventos discretos se implementan de tres formas diferentes: lenguajes de programación (GPSS [21] o SIMSCRIPT II.5 [3]), librerías de lenguajes de programación de propósito general (DESK [8] o QNAP [13]) o toolkits (Arena [18] o AutoMod [17]).

2.2. Desacople de las Fases de Simulación y Visualización

Un paradigma de simulación desacoplado independiza las fases de simulación y visualización. El objetivo del desacople es evitar visualizaciones innecesarias y poder adaptar el número de visualizaciones a la tasa de refresco de pantalla. De esta forma se libera potencia de cálculo [16] que puede usarse para mejorar otras partes de la aplicación o para ejecutar la aplicación en máquinas con una potencia de cálculo menor. El desacoplo de las fases de simulación y visualización no sólo incrementa las prestaciones del sistema [6], la precisión de la simulación y la velocidad, sino también la independencia de otros procesos del sistema [1]. La técnica de desacoplo se utiliza en aplicaciones de realidad virtual de tiempo real. La utilidad fundamental del desacoplo es distribuir el sistema en una red o usar paralelismo. Algunas aplicaciones de realidad virtual que utilizan un modelo desacoplado son: MR toolkit [19], Alice & Diver [15] o Bridge [6].

2.3. Selección de Herramientas: Fly3D y GDESK

Se ha elegido el núcleo de aplicaciones gráficas Fly3D para realizar la integración con el simulador de eventos discretos por las siguientes razones: código libre y licencia freeware, implementado en C++, bien estructurado y modularizado. Orientado a objetos (OO), suficientemente documentado, las funciones de simulación, tanto del núcleo como de las aplicaciones están claramente separadas, núcleo y aplicaciones completamente separados.

Se ha elegido el simulador de eventos discreto DESK para la integración por las siguientes razones: implementado como una librería de C++, altamente estructurado y OO, código abierto. El simulador DESK no es completamente apropiado para la integración en una aplicación gráfica. Deben cambiarse mecanismos tales como la gestión de tiempos. La versión de DESK adaptada a un núcleo de aplicaciones gráficas es GDESK [9].

2.4. Objetivos

El objetivo es mostrar los resultados de la integración de GDESK en el núcleo de aplicaciones gráficas Fly3D para cambiar su paradigma de simulación continuo y acoplado a un paradigma discreto desacoplado.

3. Núcleo de Aplicaciones Gráficas en Tiempo Real Fly3D

Fly3D SDK es un motor de videojuegos y de desarrollo de aplicaciones 3D en tiempo real. Incluye un potente motor 3D de simulación OO que se encarga de tareas como visualización en tiempo real, captura de eventos de entrada o simulación física de objetos 3D. Fly3D está orientado a plugins. Los plugins están completamente separados del núcleo de Fly3D. Todas las aplicaciones tiene un front-end común con la interfaz de usuario y el bucle principal del videojuego.

Una aplicación Fly3D es un conjunto de objetos que heredan del objeto básico de Fly3D. Cada objeto redefine un conjunto de funciones virtuales que se ponen en ejecución en determinados momentos del videojuego. Entre estas funciones están la función de simulación del objeto y la de visualización. El programador define el comportamiento del objeto usando estas funciones. El proceso de simulación recorre todos los objetos del videojuego ejecutando la función de simulación de cada objeto tomando en consideración el tiempo desde la última simulación. La visualización sigue un proceso similar.

4. Núcleo de Simulación de Eventos Discretos GDESK

GDESK es un simulador de eventos discretos adaptado a aplicaciones gráficas en tiempo real [9]. Una vez integrado en Fly3D, gestiona los eventos del videojuego. GDESK controla la comunicación de los objetos de la aplicación mediante paso de mensajes. GDESK es una librería de funciones que se invocan desde el núcleo de videojuegos. Los procesos de simulación y visualización de Fly3D se sustituye por el proceso de simulación de GDESK. La estructura básica de GDESK es el dispatcher. Sus funciones son:

- Llevar a cabo el proceso de envío de mensajes.
- Mantener los mensajes ordenados hasta que se cumpla el tiempo del mensaje.
- Enviar el mensaje al objeto destino en el tiempo indicado por el objeto emisor.

Todos los objetos de un videojuego deben heredar del objeto base de GDESK. En este objeto base se definen las funciones necesarias para el proceso de recepción y envío de mensajes.

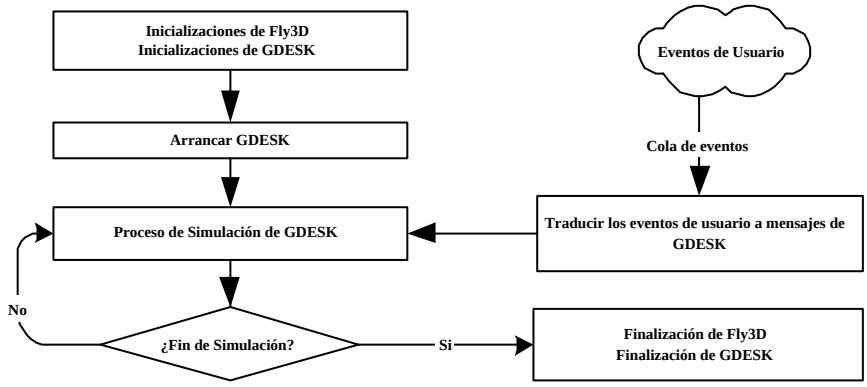

Figura 1: Bucle Principal de DFly3D

La estructura de datos del mensaje contiene dos tipos de información:

- **Información interna de GDESK**: contiene los parámetros necesarios para el control del proceso de envío y recepción del mensajes: objeto origen, objeto destino y tiempo de recepción del mensaje.
- **Información del videojuego**: parte de la estructura de datos del mensaje es accesible al programador para que defina que parámetros necesita el videojuego. De esta manera se da completa libertad al programador para definir el mecanismo de paso de mensajes sin sobrecargar el mensaje con estructuras de datos innecesarias. Estos parámetros los gestiona el programador y son completamente transparentes para GDESK.

5. DFly3D: Fly3D Discreto

DFly3D es el núcleo de aplicaciones gráficas en tiempo real discreto y desacoplado que se obtiene integrando el núcleo de simulación GDESK en Fly3D. 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 renderizado 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 lanza cada visualización.

5.1. Bucle Principal de DFly3D

El bucle principal de DFly3D (figura 1) 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). El proceso de simulación devuelve el control al bucle principal cuando no tiene mensajes pendientes de procesar y sólo por el periodo de tiempo que va a estar inactivo. El bucle principal puede gestionar este tiempo como considere oportuno.

5.2. Objetos de DFly3D

Hay dos tipos de objetos en DFly3D:

1. **Objetos del sistema**: son los objetos que realizan tareas del núcleo de Fly3D. Se crean para aislar el código correspondiente a ciertas tareas para que puedan ser objetos de GDESK y puedan comunicarse mediante paso de mensajes. Heredan del objeto básico de GDESK. No son accesibles para el programador. Cada objeto controla una tarea específica del núcleo de DFly3D manteniendo la misma funcionalidad de los procesos de Fly3D. Uno de los objetos del sistema se encarga de controlar el proceso de visualización.

2. **Objetos del videojuego**: son los objetos creados por el programador en un videojuego. Por ejemplo, personajes, paredes o proyectiles. Son los mismos objetos creados en Fly3D (heredan del objeto base de Fly3D) pero su comportamiento se modela mediante paso de mensajes (función de simulación del objeto). Estos objetos aúnan las características de los objetos básicos de Fly3D con las características de los objetos básicos de GDESK. Estos objetos los crea el programador.

GDESK trata todos los mensajes de la misma forma, independientemente del tipo de objeto del que procedan.

5.2.1. Simulación de Objetos del Videojuego en DFly3D

La simulación del sistema se modela:

- En Fly3D mediante muestreo de los objetos. Los objetos de Fly3D heredan de un objeto base que define, entre otras, una función virtual para la simulación del objeto. En cada pasada del bucle principal se recorren todos los objetos invocando la función de simulación de cada objeto.

- En DFLy3D mediante paso de mensajes. La simulación de cada objeto se define en la función de recepción de mensajes. El objeto sólo actúa como consecuencia de la llegada de un mensaje, por tanto los posibles comportamientos del objeto son las posibles respuestas a los diferentes tipos de mensajes.

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:

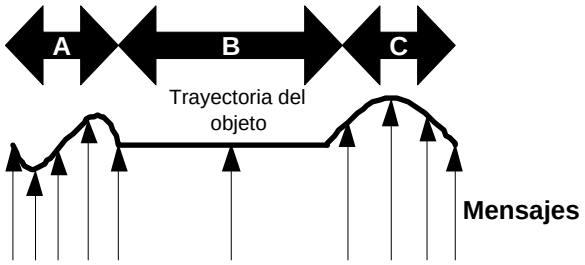

Figura 2: Ejemplo de tasa de generación de mensajes según el comportamiento del objeto

1. **Comunicar objetos**: el objeto A necesita comunicarse con el objeto B, por lo que genera un mensaje dirigido a B. El objeto B puede cambiar su estado o actuar de determinada manera o 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. 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 del mensaje, 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. Si el tiempo del mensaje es 0 significa que el mensaje lo recibe el objeto destino en el mismo instante que el objeto fuente lo envía.Si el tiempo del mensaje es mayor que 0, el objeto receptor recibe el mensaje transcurridas las unidades de tiempo definidas por el tiempo del mensaje.

La figura 2 muestra un objeto cuya trayectoria debe muestrearse de forma diferente en cada intervalo (A, B o C). El tiempo de cada mensaje debe adaptarse dinámicamente a las necesidades del objeto. El número de mensajes generados en una interacción depende de la forma en que el programador modela el comportamiento del objeto y del sistema en general.

El comportamiento del objeto lo modela el programador en su función de recepción de mensajes. Esta función debe permitir discriminar el comportamiento del objeto en función del tipo de mensaje que le puedan enviar, de los parámetros del mensaje o del objeto receptor. Esta función define a que mensajes es sensible el objeto y su respuesta para cada tipo de mensaje.

5.3. El Objeto Visualizador

DFLy3D permite la independencia del proceso de simulación del proceso de visualización (desacopla el sistema). El objeto *visualizador* 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 el número de escenas por segundo generadas 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 constante o variable durante la ejecución del videojuego.

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

1. Generar únicamente tantas visualizaciones como refrescos de pantalla. Si el número de visualizaciones es mayor se desperdicia potencia de cálculo. Evitando calcular visualizaciones que nunca se mostrarán por pantalla.
2. Determinar el instante exacto de cada visualización para permitir generar una escena antes del próximo refresco de pantalla. Esto supone que en cada refresco de pantalla se debe mostrar siempre la última escena calculada.

6. Resultados

Sea:

T_G tiempo total consumido en la ejecución del videojuego.

T_V tiempo total de visualización de las escenas.

T_S tiempo total de simulación del videojuego.

T_L tiempo total no consumido por el videojuego.

SRR tasa de refresco de pantalla (número de escenas generadas por unidad de tiempo).

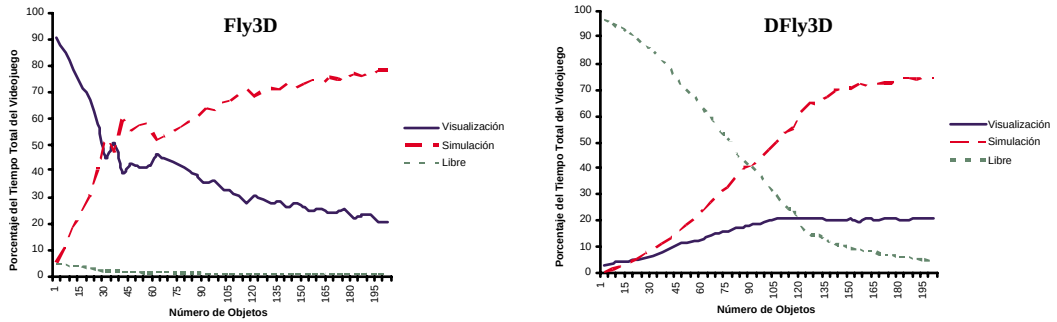

Figura 3: Tiempos de Fly3D y DFly3D

6.1. Tiempos de Visualización y Simulación

Tanto en Fly3D como en DFly3D se cumple la ecuación 1:

$$T_G = T_V + T_S + T_L \tag{1}$$

La carga del sistema depende del número de objetos en el sistema, tanto la carga de simulación como la de visualización se incrementa con el número de objetos.

Fly3D sigue un esquema de simulación continuo acoplado. El sistema está continuamente simulando y visualizando (figura 3). Un incremento en la carga de simulación supone un decremento del tiempo destinado a la visualización debido a la ecuación 1. Si SRR es baja, el videojuego no se visualiza adecuadamente. Un incremento en la carga de visualización supone un decremento del tiempo de simulación debido a la ecuación 1. Si el muestreo de cada objeto no es suficiente para cumplir el teorema de Niquist-Shannon, algún objeto puede estar muestreándose insuficientemente. Esto puede producir pérdida de eventos, colisiones no detectadas, comportamientos incorrectos,...

DFly3D permite la independencia de los procesos de simulación y visualización además de introducir un sistema discreto de simulación. El bucle principal de la aplicación no está continuamente simulando y visualizando, sino que libera tiempo para otras tareas. Sólo se libera tiempo si el sistema no está colapsado. El tiempo del videojuego no lo comparten las fases de simulación y visualización, figura 3), por lo que se libera tiempo (si la potencia de cálculo y la complejidad del videojuego lo permiten), es decir $T_L > 0$.

Mientras el sistema no está colapsado, el tiempo de simulación depende únicamente de la carga de simulación y el tiempo de visualización de la complejidad de la escena. La tasa de visualizaciones y simulaciones es constante durante la ejecución del videojuego y aumenta linealmente con el número de objetos mientras el sistema

no se colapsa. DFLy3D libera tiempo que puede ser usado para mejorar aspectos del videojuego o liberado para otras aplicaciones.

Mientras el sistema no esté colapsado se cumple la ecuación y los tiempos de simulación y visualización son independientes. Incrementar el tiempo de simulación supone decrementar el tiempo libre e incrementar el tiempo de visualización supone decrementar el tiempo libre, también.

Si el sistema está colapsado el tiempo libre es 0. En este caso, el sistema no es capaz de simular o visualizar el número apropiado de veces para garantizar la calidad del videojuego.

6.2. Tasa de Refresco de Pantalla

La tasa de refresco de pantalla de Fly3D depende de la complejidad de las fases de simulación y visualización. El sistema simula y visualiza a la máxima velocidad.

DFly3D permite definir cual es el SRR óptimo que debe generar el videojuego. El SRR definido se mantiene mientras el sistema no esté colapsado. Las pruebas realizadas fijan el SRR a 25fps. El tiempo de visualización de Fly3D decrece cuando el número de objetos aumenta porque aumenta el tiempo de simulación. El SRR 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 videojuego).

El tiempo de visualización en DFly3D es siempre menor que el de Fly3D, por lo que Fly3D genera visualizaciones innecesarias.

6.3. Tiempo Libre

DFly3D evita visualizaciones innecesarias, liberando la potencia de cálculo para otras aplicaciones o para mejorar otras partes del videojuego, como inteligencia artificial, precisión de la detección de colisiones o incrementar el realismo del videojuego.

La figura 4 muestra la diferencia entre el tiempo liberalizado por DFly3D y Fly3D. El sistema discreto siempre libera más tiempo que el sistema continuo. La diferencia siempre es positiva, hasta que se colapsa el sistema, instante en que la diferencia es cercana a cero. El tiempo libre con DFLy3D para un objeto es casi el 97 % del tiempo asignado a la aplicación (figura 3). Esto es debido a las siguientes razones:

- Los procesos de visualización y simulación son independientes en DFly3D.
- La tasa de refresco de pantalla la define el programador en DFly3D. Puede ser constante durante la ejecución o variar para ajustarse al comportamiento deseado del sistema.
- 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.

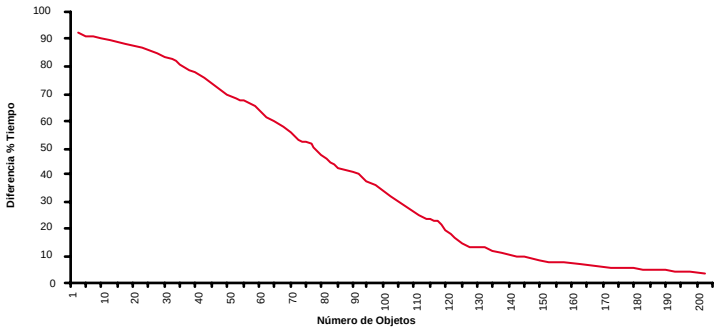

Figura 4: Diferencia entre el tiempo libre discreto y continuo

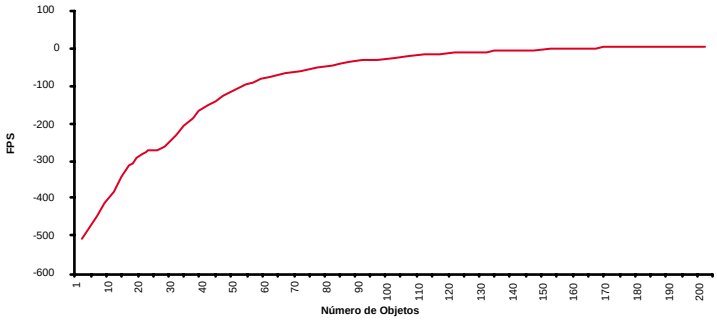

Figura 5: Diferencia entre el número de escenas visualizadas innecesariamente por segundo entre el sistema discreto y continuo

El sistema utiliza el 100 % de la CPU únicamente si el sistema está colapsado (la carga del sistema es mayor que la potencia de cálculo).

6.4. Número de Escenas Generadas

La figura 5 muestra la diferencia entre los fps generados por el sistema discreto y por el sistema continuo, manteniendo la misma calidad de imagen. DFly3D fija la SRR y la mantiene mientras el sistema no se colapsa. El sistema continuo, sin embargo, visualiza innecesariamente desperdiciando potencia de cálculo.

DFly3D permite definir el número de muestreos de un objeto y redefinir este valor dinámicamente para adaptarse al comportamiento del objeto o las necesidades del sistema (figura 6). El número de muestreos definidos para el sistema discreto se

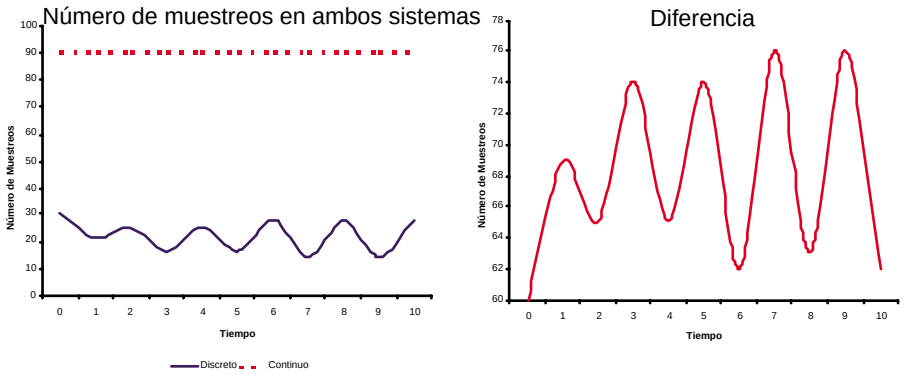

Figura 6: Muestreo de objetos en los sistemas discreto y continuo

mantiene si el sistema no está colapsado, por lo que, suponiendo que la frecuencia de muestreo que ha definido el programador para el objeto es la óptima, el sistema discreto mantiene la frecuencia óptima de muestreo del objeto. En cambio, en Fly3D, la frecuencia de muestreo es siempre la misma (si no hay variaciones en la carga del sistema durante la ejecución). La figura 6 muestra el número de muestreos innecesarios que realiza el sistema continuo.

La tabla 1 muestra una comparativa resumen de ambos sistemas.

7. Conclusiones

El esquema tradicional de simulación continua acoplada de los videojuegos puede ser mejorado utilizando un simulador de eventos discretos. De esta forma se consigue un videojuego discreto desacoplado. Se consiguen simulaciones más precisas, evitando situaciones de error y un ahorro en la potencia de cálculo consumida por el videojuego, al evitar muestreos innecesarios.

El nuevo paradigma de simulación permite que cada objeto que muestre un comportamiento continuo defina su propia frecuencia de muestreo independientemente del resto del sistema. La visualización del sistema tiene su propia frecuencia de muestreo. Las frecuencias de muestreo las determina el programador del videojuego y pueden adaptarse dinámicamente a las necesidades del videojuego.

El nuevo sistema independiza el videojuego de la potencia de cálculo de la máquina (mientras no se colapse el sistema), por lo que un mismo videojuego puede ejecutarse en máquinas con potencia de cálculo diferente con la misma calidad de simulación o mejorar la calidad de simulación en función de la potencia de cálculo.

Característica	**Fly3D**	**DFLy3D**
Simulación	Continua Acoplado	Discreta Desacoplado
Tiempo del videojuego	$T_G \simeq T_V + T_S$	$T_G = T_V + T_S + T_L$
Tiempo libre	$T_L \simeq 0$	$T_L = T_G - T_S - T_V$
$T_S \uparrow$	$T_V \downarrow, SRR \downarrow$	$T_L \downarrow, SRR \simeq$
$T_V \uparrow$	$T_S \downarrow, SRR \uparrow$	$T_L \downarrow, SRR \simeq$
Sistema colapsado	$T_G = T_V + T_S$	$T_G = T_V + T_S$
Aumento del número de objetos	Reparto del aumento entre T_S y T_V	Aumento lineal de T_S y T_V
SRR	Dependiente de la carga del sistema. Variable.	Definida por el programador. Constante o adaptable.
Tasa de muestreo de objetos	Igual para todos los objetos. Variable según carga del sistema.	Definida por el programador para cada objeto. Constante o adaptable.

Cuadro 1: Comparativa de Fly3D y DFly3D

8. Agradecimientos

Este trabajo ha sido financiado por el Ministerio de Ciencia y Tecnología (MCYT TIC2002-04166-C03-01).

Referencias

[1] Agus, M., Giachetti, A., Gobbetti, E., Zanetti, G., .ª Multiprocessor Decoupled System for the Simulation of Temporal Bone Surgery", Computing and Visualization in Science, 2002.

[2] Banks, J., Carson II, J.S., Nelson, B.B., Nicol, D.M., Discrete-Event System Simulation, Prentice Hall International Series in Industrial and Systems Engineering, 2001.

[3] CACI Products Company, http://www.caciasl.com/.

[4] Conitec Datasystems. http://www.conitec.net/.

[5] Crystal Space. http://crystal.sourceforge.net/drupal/.

[6] Darken, R., Tonnesen, C., Passarella, K., "The Bridge Between Developers and Virtual Environments: a Robust Virtual Environment System Architecture", Proceedings of SPIE 1995.

[7] Fishman, G.S., Principles of discrete event simulation, John Wiley & Sons, 1978.

[8] García, I., Mollá, R., Ramos, E., Fernández, M., "D.E.S.K. Discrete Events Simulation Kernel", ECCOMAS, 2000.

[9] García, I., Mollá, R., Barella, T., "GDESK: Game Discrete Event Simulation Kernel", Journal of WSCG 12(1), pp. 121-128, 2004.

[10] Genesis3D. http://www.genesis3d.com/.

[11] Idsoftware Page. www.idsoftware.com/archives/doomarc.html.

[12] Pausch, R., Burnette, T., Capehart, A.C., Conway, M., Cosgrove, D., DeLine, R., Durbin, J., Gossweiler, R., Koga, S., White, J., A Brief Architectural Overview of Alice, a Rapid Prototyping System for Virtual Environments, IEEE Computer Graphics and Applications, 1995.

[13] "QNAP: Queuing Network Analysis Package". 1st Int. Conf. on the Numerical Solutions of Markov chains, 1990.

[14] Quake Developers Page. www.gamers.org/dEngine/quake/.

[15] Randy, P., Conway, M., DeLine, R., Gossweiler, R., Maile, S., Ashton, J., Stoakley, R., .Alice & DIVER: A Software Architecture for the Rapid Prototyping of Virtual Environments", Course notes for SIGGRAPH'94 course, Programming Virtual Worlds, 1994.

[16] Reynolds, C., "Interaction with Groups of Autonomous Characters", Game Developers Conference Proceedings, 2000.

[17] Rohrer, M., .AutoMod Product Suite Tutorial (AutoMod, Simulator, AutoStat) by AutoSimulations", Winter Simulation Conference, 1999.

[18] Sadowski, D., Bapat, V., "The Arena Product Family: Enterprise Modeling Solutions", Winter Simulation Conference, 1999.

[19] Shaw, C., Liang, J., Green, M., Sun, Y., "The Decoupled Simulation Model for Virtual Reality Systems", CHI Proceedings 1992.

[20] Schwetman, H., ÇSIM: A C-based Process Oriented Simulation Language", Winter Simulation Conference, 1986.

[21] Ståhl, I., "GPSS - 40 Years of Development", Winter Simulation Conference, 2001.

[22] Watt, A., Policarpo, F., 3D Computer Games Technology: Real-Time Rendering and Software, Addison-Welsey. 2001.

[23] Watt, A., Policarpo, F., 3D Computer Games, Addison-Welsey. 2003.