Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXVII, págs. 3023-3039 (artículo completo)
Alberto Cardona, Mario Storti, Carlos Zuppa. (Eds.)
San Luis, Argentina, 10-13 Noviembre 2008

INCORPORACIÓN DE COMPORTAMIENTO FÍSICO EN MOTORES GRÁFICOS

Cristian García Bauza[a,b], Marcos Lazo[a,b], Marcelo Vénere[a,c]

[a] *PLADEMA, Universidad Nacional del Centro, 7000 Tandil, Argentina, ({crgarcia,venerem}@exa.unicen.edu.ar), http://www.plademа.net*
[b] *Comisión de Investigaciones Científicas de la Provincia de Buenos Aires*
[c] *CNEA*

Palabras Clave: Realidad Virtual, Simulación Computacional, Motores Físicos, Computación Gráfica.

Resumen: Actualmente, podemos encontrar en el mercado diferentes opciones de plataformas de software, llamadas motores físicos, encargadas de simular el comportamiento de un conjunto de objetos en un mundo tridimensional. Por ahora estos motores incluyen colisiones elasto-plásticas y fricción con el medio.
En este trabajo se describe la implementación de una capa de abstracción que permite comunicar fácilmente un motor gráfico con distintos motores físicos existentes, minimizando el esfuerzo de actualización y modificación de código. Se probó este esquema con dos de los motores físicos más difundidos en la actualidad (Newton Game Dynamics y Open Dynamics Engine) y se realizó un análisis de desempeño sobre escenarios suficientemente complejos como los que se requieren en las aplicaciones reales (terrenos descriptos con cientos de miles de polígonos y colocando miles de objetos en la escena).

1 INTRODUCCIÓN

Gracias a los últimos avances en hardware, hoy existen placas gráficas capaces de renderizar cientos de millones de polígonos por segundo, y también placas dedicadas a la simulación de modelos físicos en un mundo virtual (por ejemplo, el producto de nVIDIA® denominado PhysX).

Uno de los principales desafíos, es lograr realismo e inmersión a un costo computacional aceptable. Además de la calidad de la visualización de la escena; la mayoría de las aplicaciones virtuales actuales incluyen simulación de distintos fenómenos físicos, ya sea implementado en hardware o en bibliotecas de software. Estas simulaciones han evolucionado al punto donde la mayoría de los objetos en el mundo virtual pueden ser manipulados, reaccionando de forma realista de acuerdo a la fuerza aplicada, fricción y elasticidad (Göransson, 2005; Hansson, 2007).

El agregado de comportamiento físico en aplicaciones de escenarios virtuales permite al usuario desempeñarse en un ambiente similar al mundo real, el cual ya conoce, facilitándole el conocimiento de las reglas necesarias para dominarlo. Campos, tales como medicina, educación, construcción y entretenimiento, pueden beneficiarse de estas aplicaciones. Un ejemplo claro de esto, son los videojuegos o simuladores.

Para cumplir este objetivo, se han desarrollado en los últimos años varios motores físicos implementados en forma de bibliotecas (ODE, NGD, TPE, BPL[1]); los cuales permiten agregar comportamiento físico a escenas tridimensionales.

La secuencia de pasos a seguir para utilizar uno de estos motores físicos en una plataforma de creación de entornos 3D, se puede definir de forma genérica como:

1. **Inicializar Motor gráfico.**
   a. Crear entorno 3D.
   b. Crear objeto 3D (al cual se le aplicará comportamiento físico).
2. **Inicializar Motor físico.**
   a. Crear entorno de comportamiento físico.
   b. Crear objeto físico.
   c. Aplicar estado inicial al entorno y al objeto (gravedad, fricción, etc.).
   d. Establecer correspondencia entre primitiva 3D y primitiva física.
   e. Aplicar estado inicial al objeto.
3. **Repetir:**
   a. Invocar paso de actualización en Motor físico.
   b. Obtener transformaciones de objeto físico.
   c. Aplicar transformaciones a objeto 3D.
   d. Dibujar entorno 3D y objeto 3D.

Si bien el flujo central de esta secuencia se mantiene en algunos de los motores físicos, no existe un estándar de funciones, ni tampoco de parámetros entre todos ellos. Ocurre también que cada uno de estos motores necesita una configuración distinta de inicialización y muchas veces la funcionalidad implementada difiere entre uno y otro. Esto acarrea principalmente problemas para los usuarios que quieren probar rendimiento y/o precisión de cada uno de los motores físicos o utilizar distintas funcionalidades de cada uno en una misma aplicación.

Para solucionar dichos problemas, el presente trabajo se centra principalmente en crear una capa de abstracción o PAL (del inglés, *Physics Abstraction Layer*) que sea utilizada por un

[1] Ver Información adicional en Sección Referencias

motor gráfico, la cual brinde una comunicación fluida con diversos motores físicos maximizando la reutilización de código y permita cambiar el uso entre ellos. Como plataforma de pruebas de la PAL propuesta se utiliza el motor gráfico Impromptu (D'Amato et al. 2004).

En las siguientes secciones desarrollamos los puntos principales de la PAL propuesta, primeramente se explica el esquema de funcionalidad soportado y el alcance de la capa de abstracción, posteriormente se desarrollan los mecanismos estudiados para la comunicación y sincronización entre el motor gráfico y el motor físico.

Finalmente explicamos la arquitectura de la solución propuesta y su implementación; presentando también una serie de pruebas comparativas de desempeño y rendimiento entre los motores físicos estudiados mediante el uso de la PAL.

2 CAPA DE ABSTRACCION PROPUESTA

2.1 Soporte de funcionalidad

Si bien existen esfuerzos en esta área (Boeing, 2005; Fischer et al., 2005), estas propuestas se alejan de la idea de facilitar el intercambio de motores físicos de una manera transparente y cómoda para el usuario como la propuesta de Fischer et al., 2005, o proveen un conjunto acotado de funcionalidad como Boeing, 2005.

Estas últimas, proveen únicamente la funcionalidad común de los motores físicos que soporte la PAL; es decir, se utiliza un esquema de intersección de funcionalidades; no permitiendo al usuario utilizar algunas de las características de aquellos motores físicos que tengan mayor funcionalidad (Figura 1).

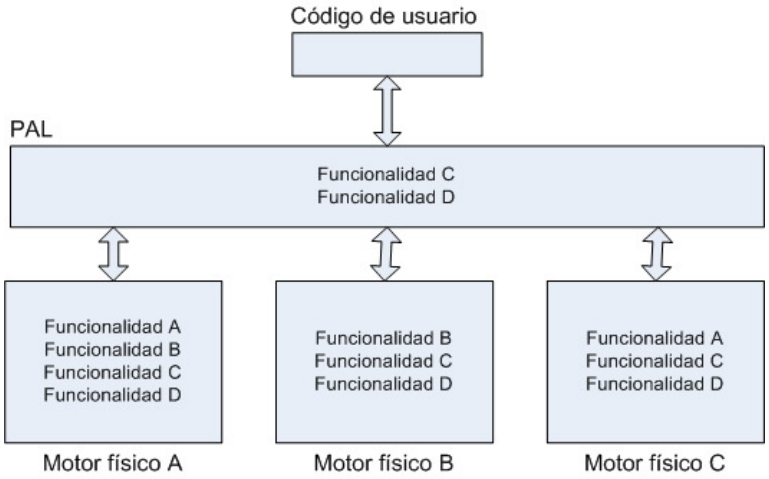

Figura 1: Capa de abstracción utilizando esquema de intersección de funcionalidades.

En nuestra propuesta, (Figura 2) se utiliza un esquema de unión de funcionalidades, no sólo proveyendo acceso a la funcionalidad común de los motores físicos, sino también brindando acceso a funcionalidad específica a través del uso de decoradores de diseño (conocidos como *wrappers)*. Esto permite que la PAL soporte el conjunto completo de funcionalidades de los motores físicos soportados (por ejemplo la Funcionalidad E mostrada en la Figura 2), y que el usuario pueda agregar funcionalidad faltante en un motor particular (Funcionalidad B).

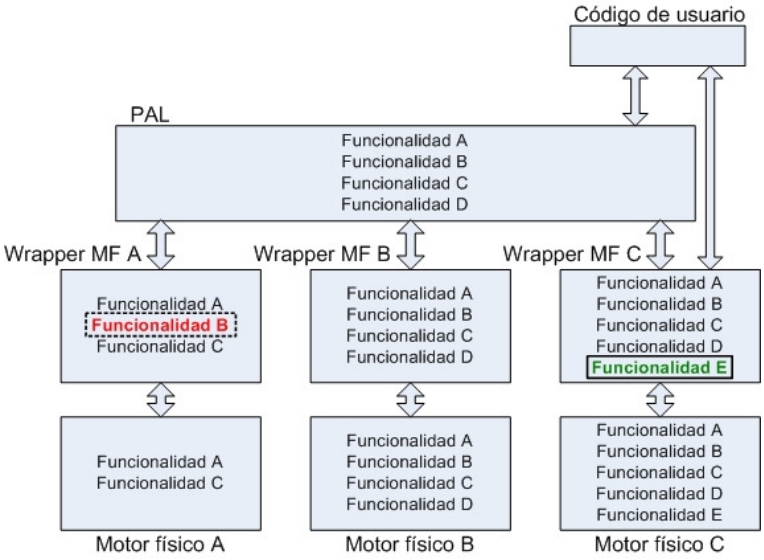

Figura 2: Propuesta de capa de abstracción utilizando esquema de unión de funcionalidades.

Se provee, además, un mecanismo para obtener listado de capacidades o *capabilities*, utilizando el mismo esquema de los controladores de placas gráficas para implementar las extensiones ARB (ARB). Brindando este esquema, el usuario puede consultar si una funcionalidad específica es soportada por el motor físico seleccionado, antes de utilizar dicha funcionalidad.

Si observamos la Figura 2, la PAL provee la capa de abstracción para la funcionalidad D; aunque si el motor físico seleccionado es el MF A, dicha funcionalidad no estará disponible; mediante el mecanismo implementado del listado de capacidades, el usuario tiene la certeza de la disponibilidad de cada función, haciendo que su aplicación sea robusta y no falle en tiempo de ejecución.

2.2 Alcance de Integración

Tanto las propuestas de Boeing, 2005 como Fischer et al., 2005 dejan a criterio del usuario la utilización o integración de una biblioteca gráfica, y no proveen casos de ejemplo o pruebas de uso de la capa de abstracción con motores gráficos propios o de terceros; siendo este un problema no trivial, puesto que comúnmente la interacción entre las dos partes, es crucial. Estas propuestas presentan también la falencia que no son provistas en formato de biblioteca de enlace dinámico de manera que puedan ser utilizadas en cualquier lenguaje de programación.

Nuestro trabajo, incluye pruebas de uso de la capa de abstracción propuesta con un motor gráfico concreto. Asimismo, la implementación se estructuró como una biblioteca de enlace dinámico de manera que la PAL, pueda ser utilizada por cualquier motor gráfico y lenguaje de programación.

2.3 Comunicación entre un motor físico y un motor gráfico

Existen dos puntos importantes a tener en cuenta al momento de comunicar un motor gráfico y un motor físico, uno es la actualización de los objetos del mundo virtual 3D; el otro es mantener la sincronía entre ambos motores.

El motor gráfico es el encargado de crear y visualizar los objetos 3D en pantalla, el motor físico es el encargado de calcular colisiones y el comportamiento del objeto dentro del mundo virtual recreado.

Para representar fielmente lo que ocurre en el entorno físico, el motor gráfico recibe los datos calculados desde el motor físico y los aplica en los objetos 3D utilizando matrices de transformación (Figura 3).

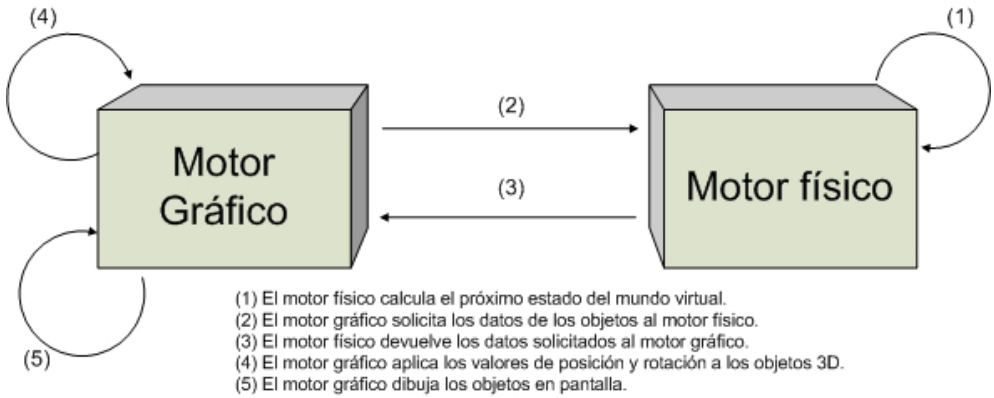

Figura 3: Comunicación de datos entre un motor gráfico y un motor físico.

Una dificultad que surge en esta interacción es que los motores físicos utilizan paso de tiempo discreto para el cálculo y los motores gráficos pueden utilizar tiempo variable para su actualización[2], con lo cual se debe implementar un mecanismo de sincronización entre ambos.

Mientras menor sea el paso de tiempo de actualización en los motores físicos, mayor será la precisión de la simulación, aumentando el tiempo de cálculo y disminuyendo el desempeño.

En este trabajo implementamos y probamos dos enfoques para mantener la sincronía entre los motores. El primero, basado en la utilización de tiempo variable de actualización en el motor gráfico, consiste en la utilización de un acumulador de tiempo. En cada ciclo de la aplicación se mide el tiempo que transcurrió desde el ciclo anterior y se suma al acumulador; mientras que el valor del acumulador sea mayor o igual al paso de tiempo prefijado del motor físico, se resta este último al acumulador y se realiza la actualización del motor físico (Figura 4).

```
delta := 0.01; //paso del motor físico
timeElapsedAnt:=0;
accumulated_time := 0.0; //acumulador
while (not(exit)) do
begin
  timeActual := clock.GetElapsedSeconds;//obtener tiempo actual
  timeElapsed :=  timeActual - timeElapsedAnt;
  timeElapsedAnt := timeActual;
  accumulated_time := accumulated_time + timeElapsed;
  while(accumulated_time > delta ) do
  begin
     world.update(delta);
     accumulated_time := accumulated_time - delta;
  end;
```

Figura 4: Algoritmo implementado que mantiene la sincronía entre los motores para tiempo de actualización variable

Por ejemplo, si el paso de tiempo del motor físico es $T = 0,01$ segundos y el tiempo

[2] También conocido como tasa de refresco de cuadros por segundo (FPS) fija o variable (Framerate fijo o variable)

transcurrido desde la ultima actualización gráfica es $X3 = 0,045$ segundos; se necesitaran realizar 4 actualizaciones del motor físico para consumir el tiempo acumulado, debido a que $X3 > T$. En cambio, si el tiempo transcurrido es $X1 = 0,0052$ segundos ($X1 < T$) no se realizará la actualización física y se sumará $X1$ al tiempo acumulado, para ser consumido en el futuro (Figura 5).

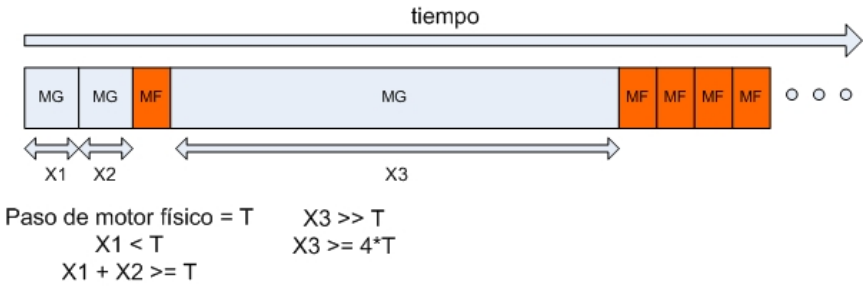

Figura 5: Esquema de interacción utilizando tiempo de actualización variable en motor gráfico (un hilo de ejecución).

Este enfoque presenta dos problemas; dado que la suma de tiempos contenida en el acumulador no necesariamente será múltiplo del paso de tiempo del motor físico utilizado, pequeños restos de tiempo quedarán en el acumulador para ser consumidos en el futuro, o en el peor de los casos, nunca serán consumidos; produciendo un desfasaje entre el tiempo transcurrido en la simulación y en la realidad. Por otra parte, en algunas ocasiones se actualizará el motor gráfico innecesariamente, cuando la posición u orientación de los objetos no cambia debido a que no se actualiza el motor físico. Como se muestra en la Figura 5, la segunda actualización del motor gráfico es innecesaria, ya que desde la primera actualización, no cambió el estado de los objetos.

El segundo enfoque se basa en el uso de tiempo de actualización fijo en el motor gráfico. Si establecemos, por ejemplo, al tiempo entre actualizaciones del motor gráfico igual a 1/20, y el paso de tiempo del motor físico igual a 1/60; por cada actualización (ciclo de renderizado) del motor gráfico, realizaremos tres actualizaciones del motor físico para mantener la sincronía (Figura 6). Es importante que el paso del motor gráfico sea un múltiplo del paso del motor físico. Por lo tanto, si el tiempo entre actualizaciones del motor gráfico es menor, se deberán realizar menos actualizaciones del motor físico, si el tiempo es mayor, entonces se deberán realizar más actualizaciones.

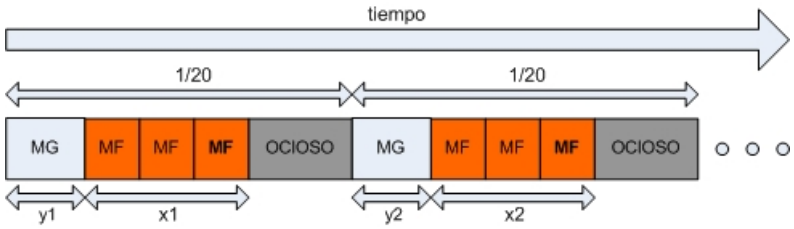

Figura 6: Esquema de interacción utilizando tiempo de actualización fijo en motor gráfico (un hilo de ejecución).

Con este enfoque no se producen desfasajes entre el tiempo simulado y el tiempo real como sucede con el enfoque anterior, ya que las actualizaciones de la vista se realizarán a intervalos de longitud conocida.

El uso de actualización de tiempo fijo, puede implementarse de dos maneras; asumiendo que la actualización del motor físico se realizará en el tiempo ocioso del motor gráfico (un

único hilo de ejecución) o utilizando procesamiento multihilo.

En el primer caso, la simulación será fluida siempre y cuando el tiempo de ciclo de renderizado del motor gráfico, sumado al tiempo de cálculo del motor físico sea menor al paso de tiempo del motor gráfico ($x_i + y_i < 1/20$).

En el caso del uso de procesamiento multi-hilo, se implementa la actualización del motor físico en otro hilo de ejecución distinto al del manejo del motor gráfico (Figura 7). De esta manera, la simulación será fluida siempre y cuando el tiempo correspondiente a la actualización del motor físico (x_i) sea menor al paso de tiempo del motor gráfico ($1/20$).

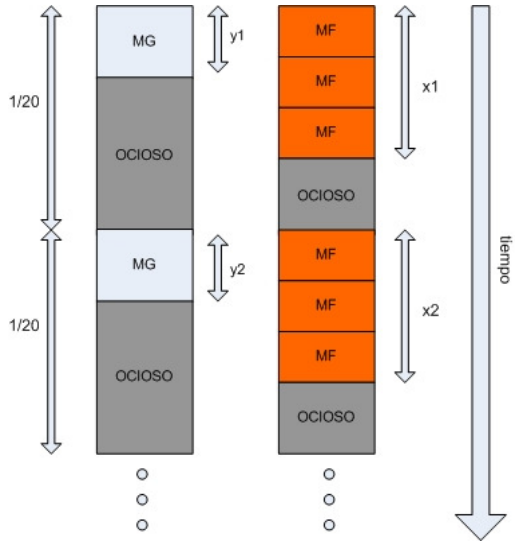

Figura 7: Esquema de interacción utilizando tiempo de actualización fijo en motor gráfico (dos hilos de ejecución).

En este trabajo se utiliza el enfoque de actualización de tiempo fijo en el motor gráfico, dado las ventajas ya expuestas del método. Como trabajo futuro se tiene previsto actualizar dicho esquema de un hilo de ejecución a un esquema multihilo (actualización del motor físico en un hilo de ejecución distinto al del manejo del motor gráfico), aprovechando, posiblemente, las características multi-hilo de algunos de los motores físicos recientes.

2.4 Arquitectura de clases y mecanismos implementados

Cuando se implementan aplicaciones que utilizan motores físicos, los desarrolladores se encuentran con el problema que cada motor presenta una interfaz distinta en sus métodos y en la mayoría de los casos se debe reescribir el código que se ha programado si se desea intercambiar un motor por otro.

Por esto, es necesario proveer una arquitectura de clases que brinde una interfaz uniforme de comunicación con el conjunto de motores físicos a utilizar, de manera que no sólo se permite reutilizar código escrito previamente (independiente al motor físico que se utilice) sino que se gana extensibilidad y adaptabilidad en el sistema.

En el presente trabajo, la capa de abstracción propuesta se implementó utilizando un conjunto de patrones de diseño (Gamma et al. 1997), principalmente los patrones *Abstract Factory*, *Template Method* y *Singleton*.

El patrón de diseño *Abstract Factory* intenta solucionar los problemas que se presentan al crear diferentes familias de objetos (en este caso, una familia de objetos por cada motor físico que se quiera utilizar). Como ya dijimos, cada motor físico tiene sus parámetros y

características, con lo cual, el código de ejecución escrito para utilizar un motor particular (por ejemplo *Newton Game Dynamics*), no servirá para utilizar otro como ODE, con lo cual es necesario contar con un mecanismo que permita reutilizar el código escrito. Dicho mecanismo es provisto por el patrón de diseño *Abstract Factory*, mediante la definición de interfaces genéricas para la creación y manipulación de los distintos objetos de los motores físicos.

Para utilizar el patrón, la implementación contiene los siguientes elementos:

- La definición de interfaces para la familia de productos genéricos (ej: *Box*, *Sphere*, *Cylinder*, etc.).
- Implementación de las interfaces de los productos para cada una de las distintas familias concretas (ej: *NewtonBox*, *NewtonSphere*, *NewtonCylinder* y *ODEBox*, *ODESphere*, *ODECylinder*, etc.).
- La definición de los métodos de creación de los productos genéricos en la interfaz de la fábrica (ej: *CreateBox*, *CreateSphere*, *CreateCylinder*, etc.) cuyo tipo de retorno serán las interfaces genéricas.
- Implementación de una fábrica para cada una de las familias concretas (ej: *NewtonPhysicsFactory*, *ODEPhysicsFactory*).

En nuestra implementación, este patrón provee una interfaz con métodos para la creación de objetos del mundo físico que el usuario utilizará para añadir a la simulación. El patrón de diseño está materializado a través de una clase abstracta (*TPhysicsFactory*) junto a las clases concretas correspondientes a cada uno de los motores físicos, y la jerarquía de clases que proveerán la interfaz genérica para manipular los objetos.

Para hacer uso de la fábrica, se instancia con un motor físico en particular a través del método *selectEngine('nombre_motor_fisico')*. Luego de la llamada a este método, la fábrica se encuentra en condiciones de crear objetos de la familia seleccionada (un motor físico en particular).

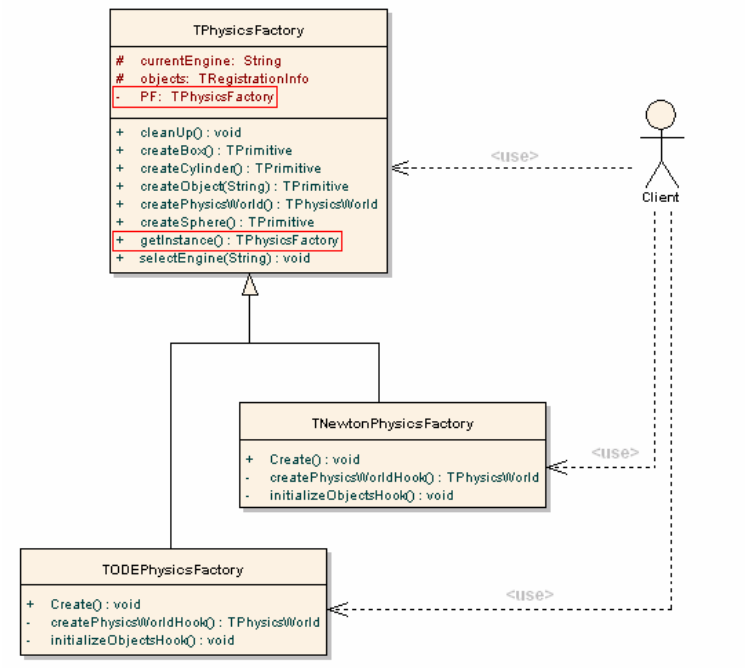

Figura 8: Esquema simplificado de la implementación del patrón *Abstract Factory* en conjunto con el patrón *Singleton*.

Es prioritario que sólo exista una única fábrica en ejecución, de modo que no haya usos inconsistentes de los objetos de la simulación. Para esto se añade a la solución propuesta por el patrón *Abstract Factory* el uso del patrón de diseño *Singleton* el cual está diseñado para restringir la creación de objetos pertenecientes a una clase a un único objeto, garantizando que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella. (Figura 8).

Suponiendo que el código relacionado a cada motor físico sea el mismo, este estará implementando en las clases genéricas *Box*, *Sphere*, *Cylinder*, etc. (o si tienen comportamiento común, en una clase madre, llamada *Primitive* por ejemplo); pero como vimos anteriormente, cada motor físico provee su propia interfaz para manipular objetos, por lo que es necesario proveer un mecanismo capaz de abstraer el código común en las clases genéricas e implementar el código propio de cada motor en las clases concretas (*NewtonBox* y *ODEBox*, por ejemplo). El patrón de diseño *Template Method* define una estructura algorítmica en la súper clase, delegando la implementación a las subclases, útil cuando es necesario realizar un algoritmo que sea común para muchas clases, presentando pequeñas variaciones entre una y otras (para nuestro caso, las variaciones serán el código propio de cada motor físico). (Figura 9).

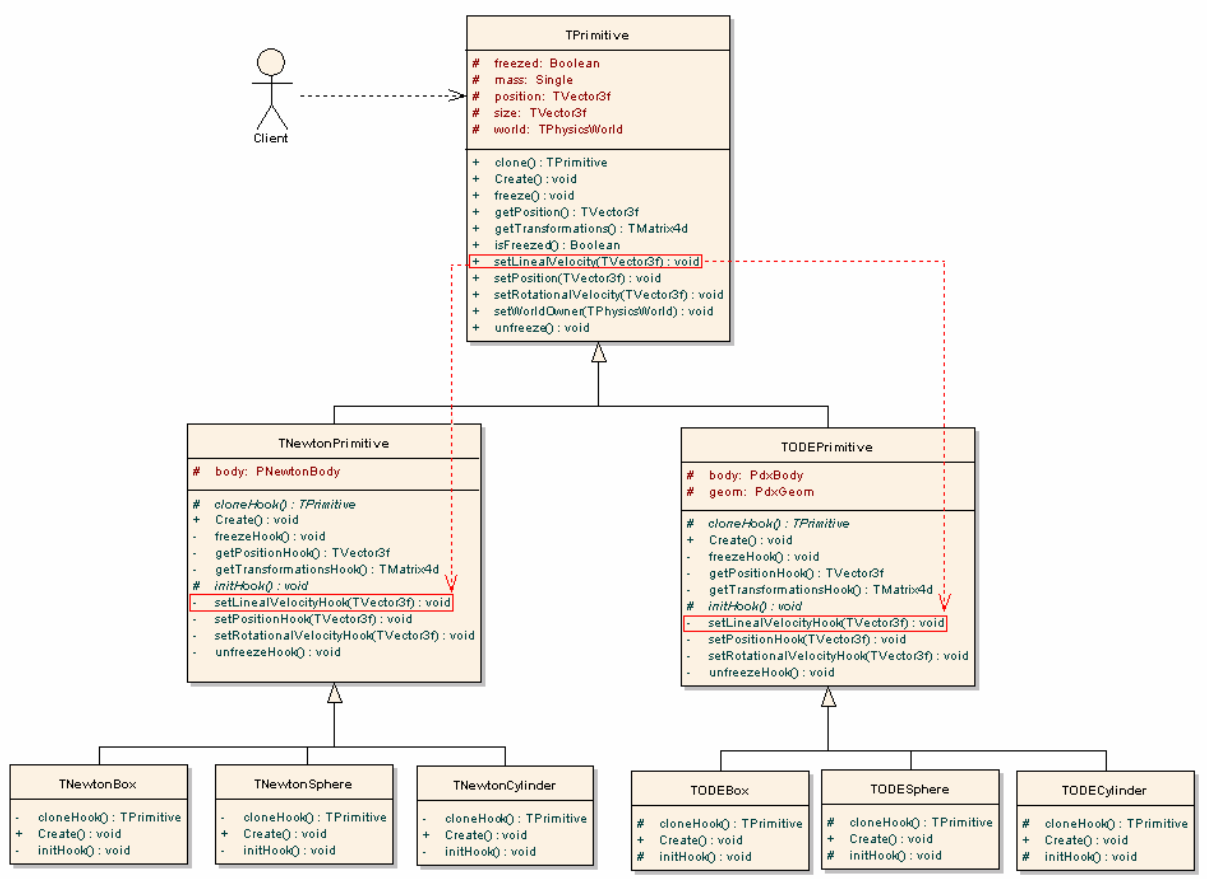

Figura 9: Diseño simplificado de la jerarquía de clases relacionada a las primitivas

3 PRUEBAS Y RESULTADOS

Las pruebas del presente trabajo, se dividen en dos grandes grupos. El primer grupo se centra en la PAL y su funcionalidad y el segundo grupo de pruebas se centra en los motores físicos y su desempeño.

La Figura 10 muestra un código de ejemplo para utilizar la capa de abstracción. Es un caso

básico en el que se inicializa la fábrica de objetos con el motor físico *Newton Game Dynamics* (NGD), creando un mundo virtual de la simulación física y un conjunto de objetos a colisionar (un par de esferas). El mismo código puede ser reutilizado por el usuario en distintos motores físicos, únicamente cambiando la primera línea de código. Este mismo concepto se repite en casos de prueba más complejos, demostrando que la PAL implementada presenta facilidad de uso y brinda la posibilidad de utilizar los distintos motores físicos con un mínimo esfuerzo por parte del programador. A futuro, está previsto desarrollar mas pruebas de cantidad de líneas de código ahorradas, cambios realizados por cambio de motor, etc.

```
//Se elige el motor físico a utilizar
TPhysicsFactory.getInstance().selectEngine('Newton');

//Se obtiene el ambiente físico (TPhysicsWorld)
world := TPhysicsFactory.getInstance().createPhysicsWorld();
world.init(initVector(-50,-50,-50), initVector(50,150,50),
           initVector(0,-9.81,0));

//Se crea una esfera
sphere1 := TPhysicsFactory.getInstance().createSphere();
sphere1.init(position1.x, position1.y, position1.z, radio1, masa1,
             world, MatrizIdentidad());

 //Se crea otra esfera
sphere2 := TPhysicsFactory.getInstance().createSphere();
sphere2.init(position2.x, position2.y, position2.z, radio2, masa2,
             world, MatrizIdentidad());
```

Figura 10: Código ejemplo para la creación de una escena sencilla utilizando la capa de abstracción física.

Sobre los motores físicos se pueden realizar dos grupos de pruebas; los que comparan el valor simulado o calculado con el valor exacto; y los que miden su desempeño y funcionalidad.

Algunos trabajos en el área, como los descriptos en Hecker et al. (2000b, a), Boeing et al. (2007), exponen un análisis de las características principales de algunos de los motores existentes; realizando principalmente pruebas de ejecución en escenas sencillas.

En este trabajo realizamos ambas pruebas centrándonos en la utilización de los motores físicos Open Dynamics Engine (ODE) y Newton Game Dynamics (NGD), aprovechando la facilidad de uso que brinda la PAL para intercambiar dos motores en una aplicación.

Los motores físicos pueden ser configurados para obtener mayor o menor grado de precisión, seleccionando el método a utilizar para resolver el movimiento y colisiones entre objetos, modelo de fricción, paso de tiempo de actualización, etc. Para estas pruebas se configuraron para obtener la mayor precisión posible.

Las pruebas fueron realizadas sobre una PC de escritorio actual: Pentium D 2.8 GHz, 1 GB RAM, NVIDIA Geforce 7600 GS. A continuación se presenta una de las varias pruebas implementadas utilizando la capa de abstracción física.

3.1 Pruebas de comportamiento: esfera rebotando sobre un plano fijo

Se realizaron una cantidad considerable de pruebas en ambos motores para distintos casos con solución conocida, de forma de poder evaluar su comportamiento y precisión: caída libre de un cuerpo, deslizamiento con fricción, choques simples, etc. Por razones de espacio solo incluimos aquí los resultados para el caso de una esfera cayendo y rebotando sobre un plano fijo.

En esta prueba comparamos el resultado exacto contra los valores obtenidos por los

motores físicos para el caso de una esfera de radio igual a 1 metro y masa igual a 10 Kg. que cae sobre un plano estático desde 100 metros de altura, actuando sobre ella una fuerza de gravedad de 9,81 m/s^2 con diferentes coeficientes de restitución.

Para el caso de una esfera con valor de restitución igual a 0,5 se obtuvo el comportamiento que se muestra a continuación.

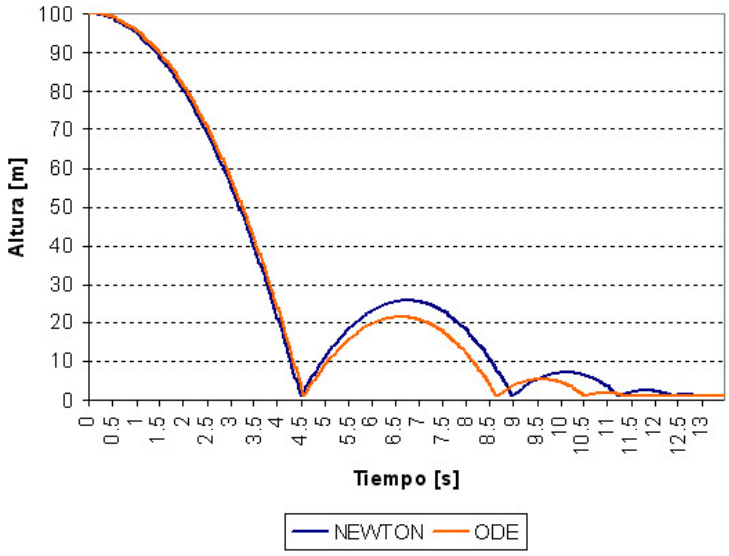

Figura 11: Altura del objeto a través del tiempo, para Newton Game Dynamics y Open Dynamics Engine.

En la Figura 11 se puede apreciar una diferencia significativa en la altura obtenida por la esfera luego de cada rebote, cuando se utiliza Newton u ODE. Esta diferencia está asociada a la forma en que los motores físicos aplican el coeficiente de restitución en las colisiones.

Para discernir cual de los motores físicos es más exacto, se compara la altura obtenida luego de cada rebote con el valor exacto utilizando (1), la cual es aplicable siempre y cuando la colisión se realice entre un cuerpo en caída libre y otro estático.

$$Cr = \sqrt{\left(h/H\right)} \tag{1}$$

Donde Cr es el coeficiente de restitución, h es la altura obtenida despues del rebote, y H es la altura desde la cual se arroja el objeto.

En la Tabla 1 se muestra el valor de altura resultante luego de cada rebote para ambos motores físicos y el caso exacto.

Rebote	Exacto	Newton	ODE
1	25	24,77	20,54
2	6,25	6,1678	4,2637
3	1,5625	1,5306	0,8744
4	0,3906	0,3798	0,182
5	0,0977	0,0914	0,0382
6	0,0244	0,0238	0,0108

Tabla 1: Altura obtenida en cada rebote para NGD, ODE, y el caso exacto medida en metros.

Se puede apreciar que Newton se acerca en mayor medida que ODE al valor correcto, aunque ambos motores obtienen alturas menores luego de cada rebote, produciendo un

desfasaje en el desplazamiento con respecto al caso exacto, claramente visible en la Figura 11. Este desfasaje se debe al error introducido por las colisiones, debido a que la integración se realiza a intervalos discretos y por esto, es natural que se produzcan penetraciones entre objetos que intervienen en una colisión. Mientras mayor sea el paso de tiempo del motor físico, mayor será la penetración producida.

Si se produce una penetración entre dos cuerpos, es porque la colisión fue detectada más tarde, entonces la velocidad resultante de los cuerpos luego de la colisión no será la correcta y la altura máxima obtenida será menor que el valor exacto. Para dar solución a esta problemática los motores físicos utilizan una heurística para aproximar al valor exacto, mediante un factor de corrección el cual está asociado al material de los objetos.

3.2 Pruebas de rendimiento: Esfuerzo del motor físico

En esta prueba se obtiene una medición preliminar del tiempo que insume el motor físico para realizar la actualización del estado a medida que aumenta el número de objetos que colisionan en la escena. Los objetos involucrados son cubos de 1 metro de lado y masa igual a 10 Kg. que son arrojados sobre un plano estático desde 20 metros de altura. La gravedad asociada es 9,81 m/s^2 y el paso de tiempo utilizado para actualizar el motor físico es 0,01 segundos. La prueba consiste en una simulación que durará 10 segundos; donde se arrojan N cajas desde cierta altura, en una formación de grilla de X * X cajas (Figura 12). Dado que la duración de la simulación es 10 segundos, y el paso de tiempo utilizado es 0,01 segundos; entonces será necesario realizar 1000 actualizaciones del motor físico.

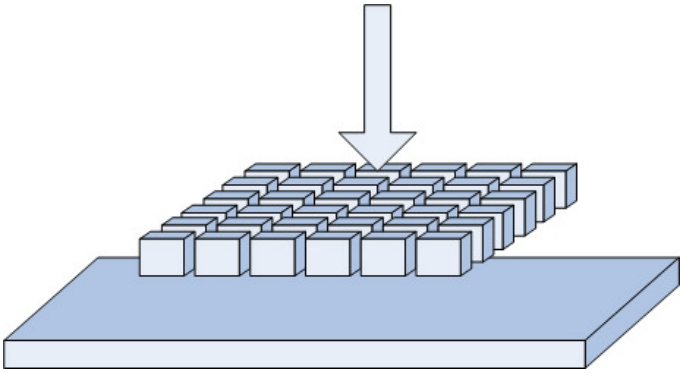

Figura 12: Esquema de la prueba de esfuerzo realizada

El esfuerzo del motor físico, para N objetos, será la sumatoria de los tiempos medidos para cada actualización del motor físico (x_i) (Figura 13). Con esta medida, estimamos el porcentaje de tiempo de la simulación que el motor físico utiliza para realizar la actualización.

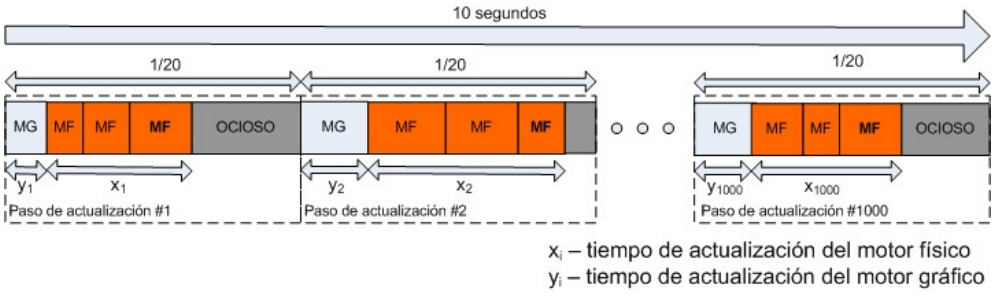

Figura 13: Esquema de tiempos para el caso de prueba de esfuerzo del motor físico.

En el gráfico de la Figura 14 se muestran los valores correspondientes al esfuerzo computacional de ambos motores físicos, a medida que se incrementa el número de objetos en la escena. De los motores analizados, Newton es el que presenta mejor desempeño, permitiendo mayor cantidad de objetos en escena.

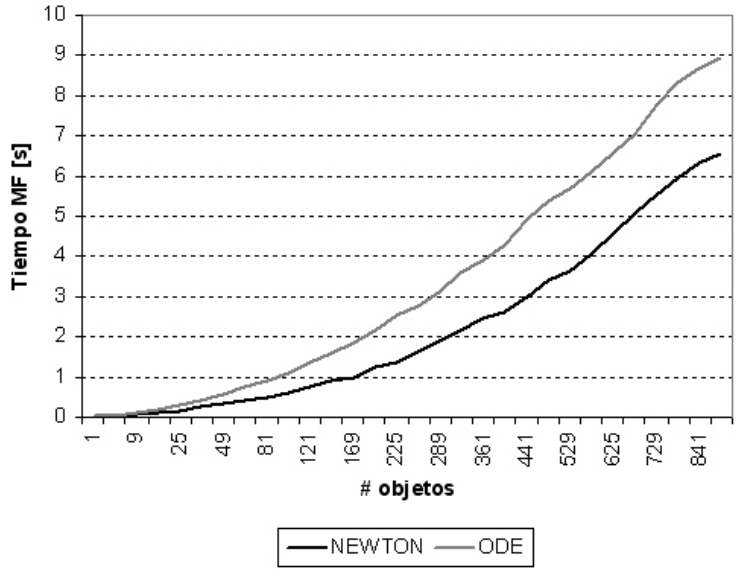

Figura 14: Esfuerzo de los motores físicos (medido en segundos) para realizar la actualización del estado.

3.3 Mejoras de rendimiento

Muchos motores físicos, como el caso de los motores analizados, ofrecen la posibilidad de desactivar los objetos que se encuentren en reposo para que no sean considerados en la actualización (auto-desactivación), la PAL implementada permite utilizar esta mejora. A continuación se presentan los resultados obtenidos hacienda uso de la auto-desactivación comparándolos con los resultados de no utilizarla. NEWTON+ y ODE+ hacen referencia a los motores físicos utilizando la auto-desactivación.

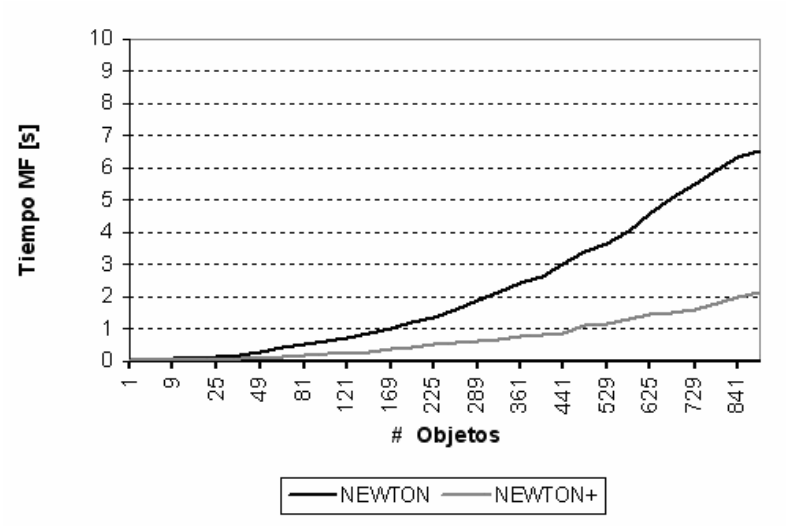

Figura 15: Comparación Newton y Newton+ (auto-desactivación)

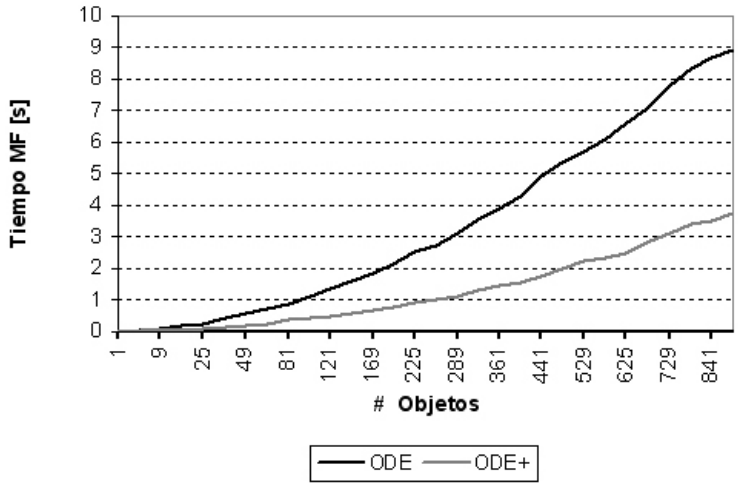

Figura 16: Comparación ODE y ODE+ (auto-desactivación)

Como se muestra en la Figura 15 y Figura 16, utilizando la capacidad de auto-desactivación, los tiempos empleados por los motores físicos disminuyen notablemente, dejando gran parte del tiempo de simulación para el motor gráfico permitiendo gráficos más complejos, o bien mayor cantidad de objetos en escena.

Es claro que esta característica es bien utilizada sólo en casos en los cuales la simulación posea un grupo de objetos en reposo, o que existan grandes posibilidades de que a lo largo de la simulación cambien de estado activo a reposo, y viceversa. La auto-desactivación es una característica importante de los motores físicos y la PAL implementada, ya que permite obtener más ciclos de CPU libres.

Además de la técnica anterior, se implementó una versión modificada de clonación de modelos 3D (Dudash, 2004; Zelsnack, 2004) en la que se mantiene en memoria una única copia del modelo 3D y cada clon posee su propia matriz de transformaciones, con lo cual se reserva recursos para otros propósitos. Además de la clonación se establecieron límites al mundo virtual. Cada objeto que se encuentre fuera de los límites es eliminado de la biblioteca física y del motor gráfico, liberando recursos posibilitando la creación de otros objetos (Figura 17).

4 CONCLUSIONES

La capa de abstracción provee una interfaz independiente de las bibliotecas físicas, sin agregar sobrecarga de tiempo, reduciendo significativamente el esfuerzo del programador para portar sus aplicaciones a otros motores físicos. Esto es importante, dado que cada motor físico brinda a las simulaciones velocidad de cálculo, estabilidad o precisión.

Un motor físico es una solución aproximada para representar una escena del mundo real y ninguno se adapta a todas las necesidades de los usuarios; por ejemplo, para aplicaciones de juegos se buscará conseguir mayor velocidad de cálculo, mientras que para herramientas de carácter científico primará la precisión. Es muy factible que el programador quiera seleccionar el motor que mejor se adapte a sus necesidades; realizando pruebas de rendimiento y cota de error a un conjunto de motores físicos.

En la simulación existen un gran número de parámetros que el usuario debe ajustar para lograr el efecto deseado, entre ellos coeficientes de rozamiento y elasticidad asociados a materiales, precisión del método de cálculo o modelo de fricción a utilizar, paso de tiempo del motor físico, etc. Es importante y es responsabilidad del usuario utilizar parámetros

adecuados en la simulación para lograr el efecto deseado.

Para el caso particular de los motores analizados, los resultados obtenidos son próximos a la realidad, siendo Newton el que mejores resultados obtuvo en las pruebas comparativas con la realidad y los test de esfuerzo. Ambos motores son adecuados para juegos o simulaciones que no requieran un grado de precisión importante.

En trabajos futuros se prevé extender la PAL para brindar soporte a mayor cantidad de motores físicos, realizando también pruebas LOC (*lines of code*) o cantidad de líneas de código modificadas y ahorradas entre distintos motores como así también de *overhead* de tiempo agregado por la PAL en diversos casos de prueba. Particularmente se está trabajando también en la extensión de la PAL mediante código de usuario para soportar modelos de simulación de aguas superficiales mediante Lattice Boltzmann.

Utilizando la PAL implementada, logramos realizar simulaciones con hasta 4000 objetos en movimiento en una PC de escritorio. A continuación se muestran algunas capturas obtenidas de las simulaciones.

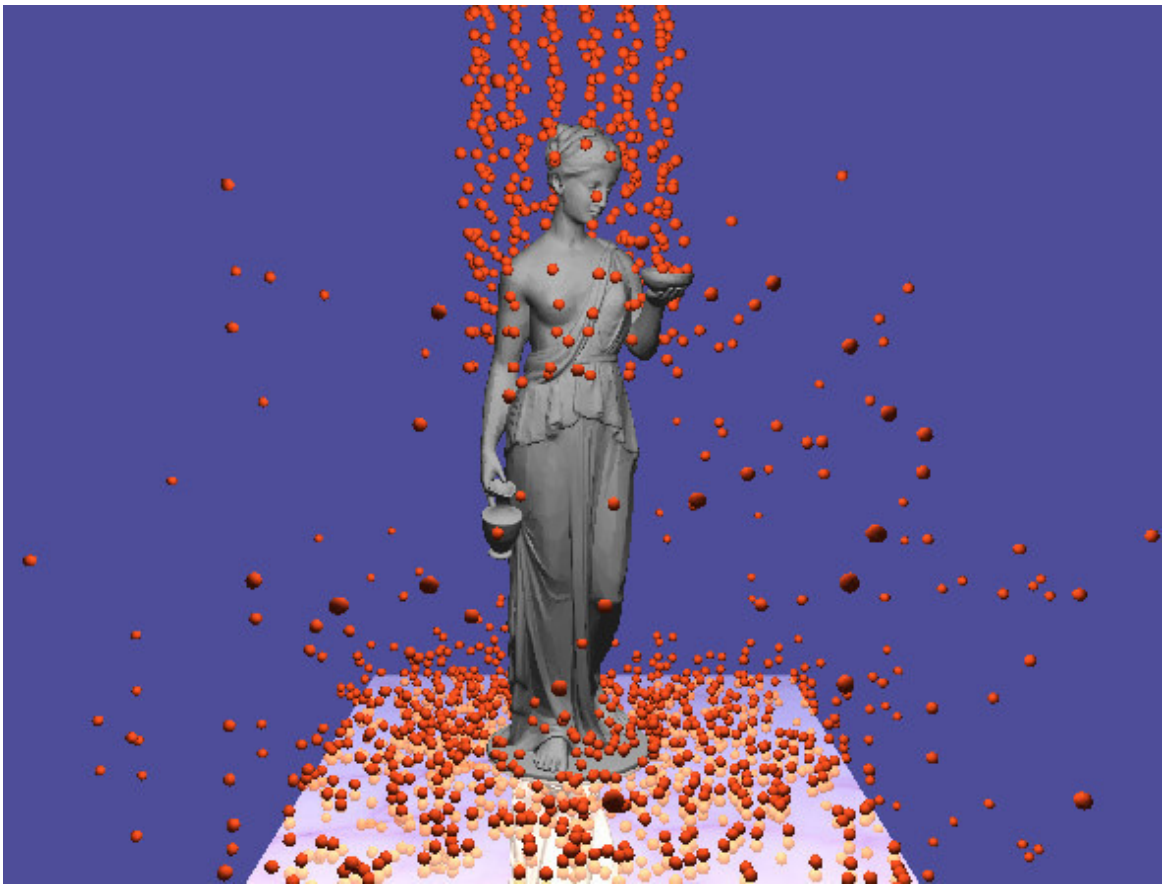

Figura 17: Escena de 4000 esferas (320.000 polígonos) con movimiento dinámico. Para el modelo estático de la estatua se utilizó una geometría para visualizar (42.000 polígonos) y otra aproximada para simular comportamiento (5800 polígonos).

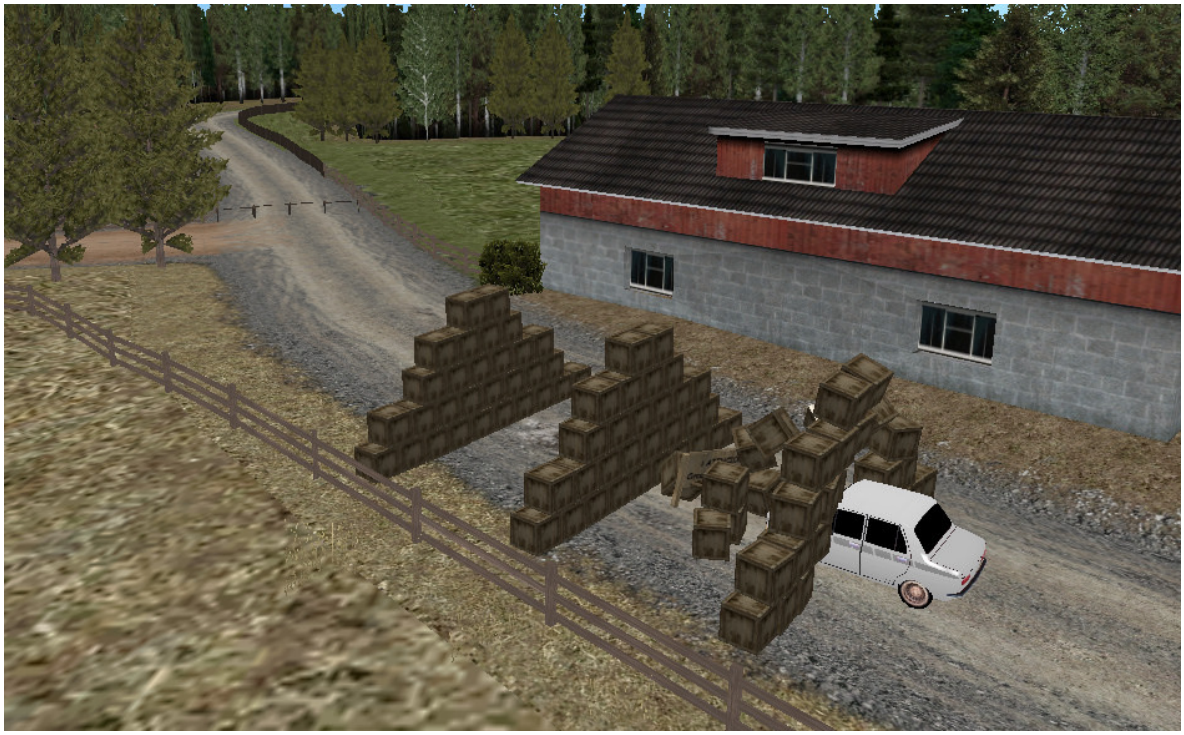

Figura 18: Escena de 90 cajas (1500 polígonos) con movimiento dinámico en una escena compleja (8500 polígonos).

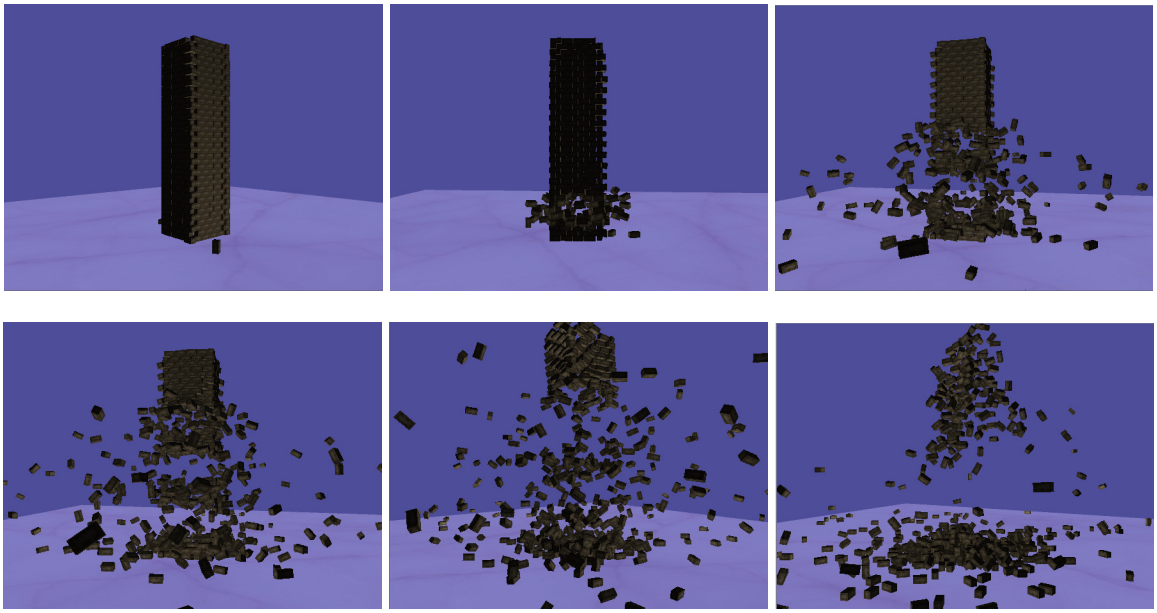

Figura 19: Escena de 900 cajas (15.000 polígonos). La secuencia muestra distintos momentos de una simulación de explosión interna en la pila de cajas.

5 REFERENCIAS

Boeing A. PAL, Physics Abstraction Layer Home Page. [Online] Disponible en: http://www.adrianboeing.com/pal/. 2005

Boeing A., Bräunl T., Evaluation of real-time physics simulation systems, *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, I:281-288, 2007.

BPL, Bullet Physics Library. [Online]. Disponible en: http://www.bulletphysics.com
D'Amato J.P, García Bauza C.. Simulación de Escenarios Tridimensionales Dinámicos. Tesis de grado de Ingeniería de Sistemas, Universidad Nacional del Centro. 2004.
Dudash Bryan. Mesh Instancing. NVIDIA *Technical report*, 2004. [Online] Disponible en: http://http.download.nvidia.com/developer/SDK/Individual_Samples/DEMOS/Direct3D9/src/HLSL_Instancing/docs/HLSL_Instancing.pdf
Gamma E., Helm R., Johnson R, and Vlissides J.. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 1997.
Hecker C., Lander J., Product Review of Physics Engines, Part One: The Stress Test. *Technical report*, 2000. [Online] Disponible en: http://www.gamasutra.com/features/20000913/lander_01.htm
Hecker C., Lander J., Product Review of Physics Engines, Part Two: The Rest of the Story. *Technical report*, 2000. [Online] Disponible en: http://www.gamasutra.com/features/20000920/lander_01.htm
Hansson H. Craft Physics Interface. *Master Thesis*, Linköping University. Sweden. ISRN LiTH-ISY-EX--07/3887--SE. 2007. [Online] Disponible en: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-8497
Göransson J. AgentPhysics: Software Design for Pluggable Real Time Physics Middleware. *Master Thesis*, Lulea Tekniska Universitet. Sweden. ISSN 1402-1617 / ISRN LTU-EX--05/270--SE / NR 2005:270. 2005. [Online] Disponible en: http://epubl.ltu.se/1402-1617/2005/270/LTU-EX-05270-SE.pdf
NGD, Newton Game Dynamics Home Page. [Online]. Disponible en: http://www.newtondynamics.com/
NVIDIA, NVIDIA PhysX Home Page. [Online]. Disponible en: http://www.nvidia.com/object/nvidia_physx.html.
ODE, Open Dynamics Engine Home Page. [Online]. Disponible en: http://www.ode.org/
Fischer A., Reinot A., Streeter T.. OPAL, Open Physics Abstraction Layer Home Page. [Online] Disponible en: http://opal.sourceforge.net/. (2005)
OpenGL ARB "Architecture Review Board". [Online] Disponible en: http://www.opengl.org/about/arb/
TPE, Tokamak Physics Engine Home Page. [Online]. Disponible en: http://www.tokamakphysics.com/
Zelsnack J. Pseudo Instancing. NVIDIA *Technical report*, 2004. [Online] Disponible en: http://http.download.nvidia.com/developer/SDK/Individual_Samples/DEMOS/OpenGL/src/glsl_pseudo_instancing/docs/glsl_pseudo_instancing.pdf