# Arquitectura Multihilos Para Motores 3D.

**Zurita, Pablo Esteban**

*Universidad Tecnológica Nacional, Facultad Regional Córdoba*

**Abstract**

*"Los microprocesadores de múltiples cores ya están llegando a la masas y dejaron de ser algo exclusivos de los servidores y las supercomputadoras. La mayoría de las computadoras personales y consolas de video juegos usan microprocesadores con múltiples cores que permiten ejecutar de forma paralela varios hilos de ejecución en hardware. Por otra parte la velocidad de cada core se ha estabilizado y por lo tanto no es posible mejorar la performance de ejecución de una aplicación que se ejecuta en un solo hilo de ejecución simplemente agregando más hardware. Pero crear una arquitectura para aplicaciones interactivas en 3D que usen este hardware nuevo es un reto por la necesidad de interacción entre los diferentes subsistemas. En este artículo se muestra la arquitectura que se creó para el Chromaticity Engine, un motor 3D que fue pensado para aplicaciones interactivas. El objetivo final del articulo es mostrar los diferentes inconvenientes al crear un motor 3D multihilo para aplicaciones interactivas y como se solucionaron esos inconvenientes en el Chromaticity Engine."*

**Palabras Clave**

Motor 3D, Engine 3D, Computación Grafica, Arquitectura, Xenon, Cell, multihilo, multithread.

## Introducción

La complejidad de los motores 3D y las aplicaciones que los utilizan hacen muy difícil la separación de un motor de un solo hilo en múltiples hilos. A nivel de la arquitectura los diferentes subsistemas (como por ejemplo el subsistema de simulación de física) de un motor dependen entre sí por lo tanto hay dos maneras de encarar la arquitectura de un motor 3D. La primera es usar la misma arquitectura que se usaba cuando había un solo core y un solo microprocesador con la diferencia que se trata de crear hilos en funciones especificas. El beneficio de este sistema es que en este método se puede mantener bastante código sin modificar y usar APIs (Application Programming Interface o Interfaz de Programación de Aplicaciones) como OpenMP para hacer todo más fácil. El problema con esta solución reside en que se trata de mezclar dos modelos que son fundamentalmente diferentes, el modelo de hilo único y el modelo de múltiples hilos. Este factor hace prácticamente imposible aprovechar al máximo el hardware disponible para obtener buenas ganancias de performance. El otro modelo consiste en dejar de lado la actualización serial de cada subsistema y remplazarlo por un modelo donde cada subsistema se actualiza de manera independiente pero siempre manteniendo cierta coherencia en los tiempos transcurridos en la simulación. Este modelo permite utilizar mejor los recursos disponibles en un microprocesador de múltiples cores y con una arquitectura adecuada es posible especializar para un microprocesador en específico sin requerir grandes cambios al motor entero. Este último modelo es el utilizado en el Chromaticity Engine y es el que va a ser expuesto.

Es importante reconocer también el hecho de que las arquitecturas de los microprocesadores en las diferentes plataformas difieren de manera sustancial y por lo tanto es necesario que esta arquitectura se pueda moldear para adaptarse a las diferentes plataformas. Si observamos los tres microprocesadores más importantes para aplicaciones interactivas en 3D vamos a ver grandes diferencias. Los

tres microprocesadores que vamos a observar son:

- Cell Broadband Engine de Sony, Toshiba e IBM que es utilizado en servidores y la consola PlayStation 3.
- Xenon de IBM que es utilizado en la consola Xbox 360 de Microsoft.
- Core 2 Duo de Intel que es utilizado en PCs y la última serie de Macs de Apple.

Teniendo en cuenta estas diferencias se va a exponer la arquitectura y van a evaluar los casos específicos para cada microprocesador.

**Microprocesadores**

Las diferencias en los microprocesadores que vamos a evaluar son de gran importancia. La primera diferencia se hace ver en la cantidad de cores en los microprocesadores y las capacidades computacionales de los diferentes cores.

En el caso del Core 2 Duo de Intel vemos un microprocesador con dos o cuatro cores, un cache L1 de 64 KB por cada core, un cache L2 compartido de 4 MB, ejecución fuera de orden, y un hilo en hardware por core [1, 2, 3]. Este microprocesador que es fácil de programar ya mantiene la mayoría de las características de los microprocesadores de las generaciones anteriores. Lo más importante en este sentido es que cada core mantiene la ejecución fuera de orden y que el cache L2 compartido tiene un tamaño considerable. Si analizamos esta situación podemos ver que podríamos directamente asignar un core a uno o varios subsistemas del motor hasta utilizar los dos o cuatro cores.

En el caso del Xenon de IBM vemos que es un microprocesador con tres cores, con un cache L1 de 64 KB por cada core, un cache L2 de 1 MB compartido por los tres cores y el GPU, ejecución en orden, y dos hilos simétricos en hardware por cada core [4, 5]. Aquí es importante observar el hecho de que aunque parece muy diferente al Core 2 Duo en la arquitectura, vamos a ver que las diferencias no son tan notables que las que hay con el Cell. De todas formas hay diferencias significativas que van a impactar a la arquitectura del motor. Primeramente esta la ejecución en orden de las instrucciones lo que va a causar que si uno no organiza bien las instrucciones a ejecutar es posible perder bastante performance mientras un hilo ejecuta una instrucción intensiva. En este caso vamos a tener en cuenta la salida del compilador para ver que instrucciones generamos y crear código que sea más amigable con este factor. Por otra parte el tamaño del cache L2 es bastante más pequeño que el cache L2 en un Core 2 Duo pero sigue estando en los parámetros aceptables como para no hacer decisiones de gran impacto basados en el tamaño del cache L2. En general podemos decir que se va a poder asignar un core a cada subsistema aunque en este caso hay que tener más cuidado al momento de manejar la información y el orden de ejecución por las consideraciones ya mencionadas.

Por último vemos el Cell Broadband Engine de Sony, Toshiba e IBM. Este es un microprocesador con un “Power Processor Element” (“PPE”) que es encargado de controlar seis u ocho “Synergistic Processing Elements” (“SPE”). El PPE tiene un cache L1 de 64 KB y un cache L2 de 512 KB, dos hilos de ejecución por hardware, este core esta hecho

principalmente para controlar los SPEs. Cada SPE tiene 256 KB de memoria sin cache, tiene un modelo de ejecución en orden y un solo hilo en hardware [6, 7, 8]. Este es el microprocesador más complejo para programar. Los SPE tienen un cache muy chico lo que implica que hay que separar todos los trabajos de manera muy fina para mantener a todos los SPEs ocupado y en este caso definitivamente no es posible asignar un SPE a cada subsistema porque la combinación de cache pequeño y ejecución en orden crearían estragos desde el punto de vista de la performance. Por lo tanto hay que darle especial importancia al planificador de tareas o scheduler del motor de tal formas que los SPEs se mantengan ocupados pero sin estancarse en una tarea en particular [9].

**Paralelismo**

Teniendo en claro las diferencias en las diferentes plataformas es importante encontrar un modelo de paralelismo de manera que las plataformas sean bien aprovechadas sin crear una arquitectura difícil de mantener y expandir. Esto implica ignorar la posibilidad de agregar paralelismo función por función ya que las posibles ganancias de performance son muy limitadas [10]. Por otro lado tenemos que tener en cuenta que la interacción entre los diferentes subsistemas en un motor 3D es constante por lo tanto no es posible simplemente separar cada subsistema en diferentes cores como si estuvieran en el vacio (Figura 1). Dada la naturaleza de las aplicaciones interactivas hay que buscar soluciones de paralelismo que permitan la interacción entre los diferentes subsistemas sin que esto implique excesiva complejidad en mantenimiento y costo en performance.

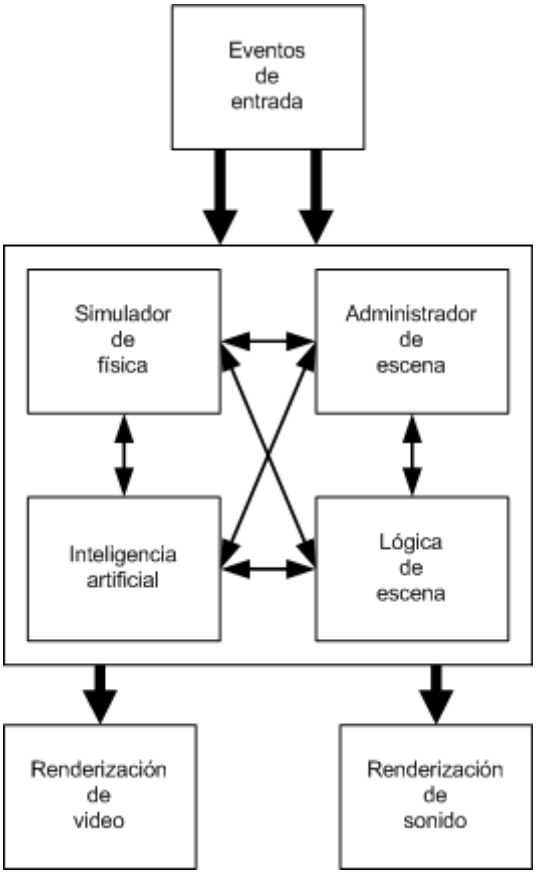


**Figura 1: Ejemplo de cómo es la interacción entre diferentes subsistemas de un motor 3D para aplicaciones interactivas. Como se puede ver hay dependencia entre los diferentes subsistemas hasta el momento de renderizar el video o sonido.**

Uno de los modelos de paralelismo posibles implica en mantener el modelo del bucle principal de un motor 3D para microprocesadores con un solo core y simplemente paralelizar los subsistemas que no interactúan entre si [11]. Por ejemplo si tenemos un subsistema simulador de partícula entonces lo podemos mantener paralelo a la inteligencia artificial y ejecutar de manera simultánea en diferentes cores los dos subsistemas (Figura 2). Pero este modelo tiene muy pocas aplicaciones en un motor 3D para aplicaciones interactivas ya que la cantidad de subsistemas que se pueden paralizar de esa manera son limitados. En este modelo procesadores como el Cell Broadband Engine quedan esperando instrucciones para ejecutar en los

diferentes SPEs pero en definitiva el 80 por ciento de los SPEs van a quedar libres. Este modelo tampoco permite da control sobre cómo se van a utilizar los diferentes cores.

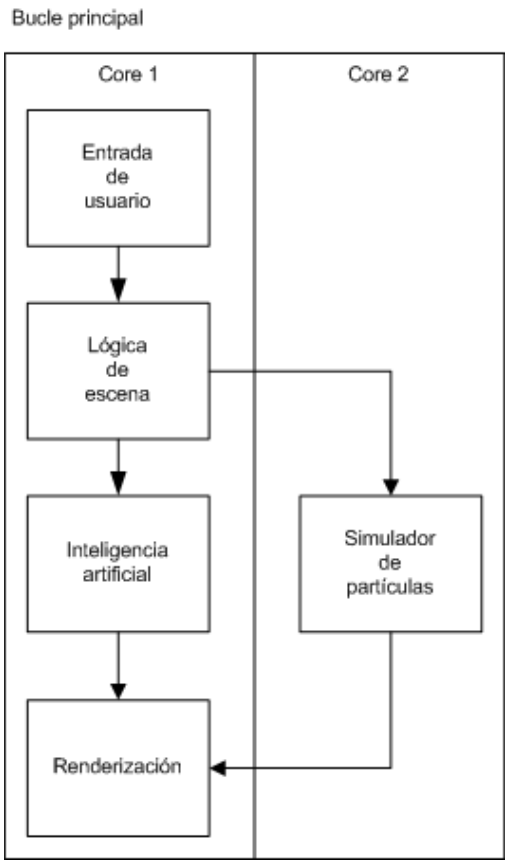


**Figura 2: Ejemplo de paralización de subsistemas independientes entre sí. En este caso solo la inteligencia artificial y el simulador de partículas son independientes entre sí por lo tanto se ejecutan de manera paralela. Pero el resto del bucle permanece en un solo core tomando la mayoría del tiempo de ejecución.**

El Chromaticity Engine utiliza un modelo hibrido de paralelismo donde los subsistemas se actualizan de manera totalmente paralela siempre trabajando con la última información disponible del resto de los subsistemas, y a la vez dentro de cada subsistema otro nivel de paralelismo es posible a través de la actualización de objetos independientes dentro del subsistema en sí.

En el primer nivel de paralelismo se busca tener todos los subsistemas sean de ejecución independiente del resto de los subsistemas (Figura 3). La dependencia entre los diferentes subsistemas sigue existiendo, pero en vez de que un subsistema termine de actualizarse para pasar al siguiente, todos los subsistemas se actualizan en paralelo basado en la última información disponible. El gran beneficio de paralelizar de esta manera el motor es que se puede escalar a procesadores con múltiples cores sin problemas, es posible mantener cada subsistema en su propio core.

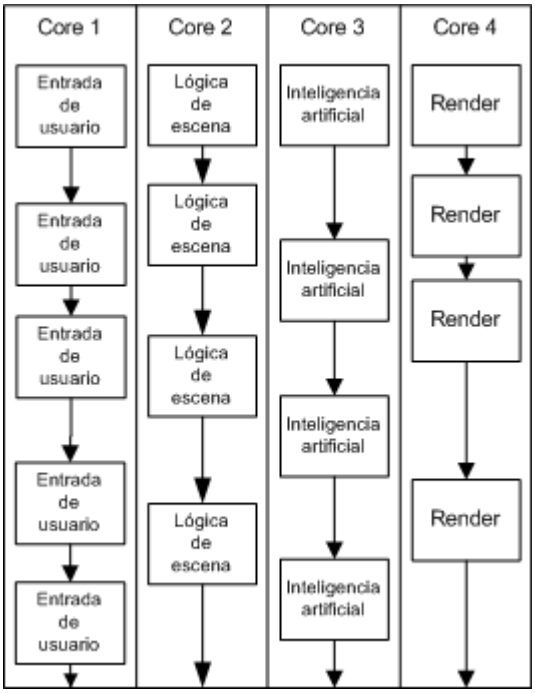


**Figura 3: Ejemplo del primer nivel de paralelismo en el Chromaticity Engine. En este caso podemos ver que los diferentes subsistemas están en cores diferentes y corren de manera totalmente independiente. Al momento de actualizarse cada subsistema, la última información disponible de cada subsistema se obtiene y se utiliza.**

El segundo nivel de paralelismo va al nivel de los subsistemas en sí. Dentro de un subsistema ciertas operaciones no dependen entre si y por lo tanto es posible realizar estas operaciones en paralelo (Figura 4). Por ejemplo si tenemos un subsistema de animación que va a modificar la geometría

de los objetos dinámicos en una escena, es posible animar esos dos objetos de manera paralela ya que un objeto no depende del otro. Este modelo de paralelismo no es muy útil en una arquitectura como la del Intel Core 2 Duo ya que la cantidad de cores es igual o menor que la cantidad de subsistemas en un motor en la mayoría de los casos. Pero en cambio en un microprocesador como el Cell Broadband Engine este modelo es muy importante ya que los SPEs pueden manejar muy poca información y hay varios SPEs.

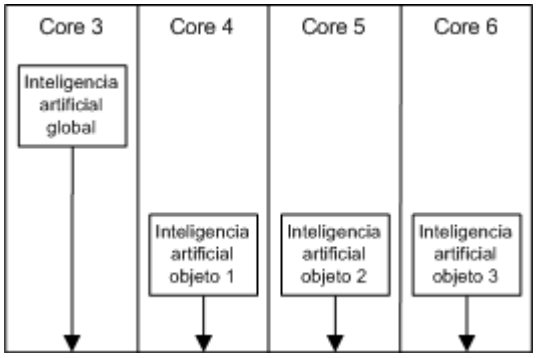


**Figura 4: Ejemplo del segundo nivel de paralelismo en el Chromaticity Engine. En este caso la inteligencia artificial global y la inteligencia artificial de dos objetos (por ejemplo, 3 personajes en la escena) se actualizan de manera paralela. Los objetos no van a actualizar siempre en el mismo core necesariamente.**

Una característica que todas las plataforma van a compartir desde el punto de vista de la ejecución paralela tiene que ver con la necesidad de crear puntos de sincronización cuando la ejecución de uno o más subsistemas va mas rápido o más lento que el resto de los subsistemas. Por ejemplo si el administrador de escena no actualiza lo suficientemente rápido la posición de un generador de sonidos entonces al momento de renderizar el sonido la salida va a ser totalmente inapropiada. Por eso es necesario definir un umbral de cuadros antes de crear un punto de sincronización para que todo vuelva a estar sincronizado.

**Scheduler o planificador de tareas**

Dejar que cada subsistema decida que core utilizar seria una muy mala decisión de diseño. La razón es que, o el subsistema no tiene información suficiente como para tomar una buena decisión sobre que core utilizar, o la lógica sobre que hilo usar se repite a través de cada subsistema creando un gran problema de mantenimiento y performance. Por esa razón en el Chromaticity Engine hay un scheduler o planificador que se encarga de manejar todas las operaciones que tienen que ser ejecutadas en los diferentes hilos. El scheduler va a ser diferente para cada tipo de microprocesador porque por ejemplo no sería efectivo aplicar las mismas políticas de scheduling basado en un Core 2 Duo a un Cell con ocho SPEs. En el caso del Core 2 Duo se puede dejar un o un par de subsistemas en cada hilo y obtener una performance más que aceptable, pero en el caso del Cell el scheduler va a tener un trabajo más complejo al ir asignando diferentes SPEs a cada mini proceso en cada subsistema. También hay que tener en cuenta que el scheduler no puede crear y destruir hilos constantemente ya que bajo ninguna de las plataformas la creación y destrucción de hilos es barata. Y con respecto a la destrucción de hilos, un hilo nunca tiene que ser matado sino que ese hilo tiene que cometer un “suicidio” porque matar un hilo es una de las operaciones más costosas en todas las plataformas [12, 13].

La política de scheduling para el Intel Core 2 Duo consiste en tener los subsistemas balanceados en los diferentes hilos. Para hacer el balanceo correcto es necesario recolectar métricas de los diferentes subsistemas y hay que acomodar los subsistemas en los diferentes hilos. Uno de los problemas para lograr el balanceo

correcto a través de los diferentes cores es que no hay forma de especificar definitivamente en que core correr un hilo. En Windows se puede sugerir que core usar usando la función SetThreadAffinityMask pero esto no asegura que efectivamente un hilo se ejecute en el core que queremos. Pero incluso esta práctica no es efectiva porque el entorno donde se usan los Intel Core 2 Duo hay muchos hilos en ejecución y por lo tanto es mucho mejor dejar al scheduler de Windows manejar a que core asignar cada hilo.

La política de scheduling para el Xenon es bastante similar a la del Intel Core 2 Duo, pero hay algunas diferencias. Primero es necesario asignar específicamente que hilo en hardware usar de lo contrario todos los hilos que creamos van a estar ejecutándose en el mismo hilo en hardware del hilo que lo creo. Esto se hace usando la función XSetThreadProcessor con el argumento 0 o 1 para el core uno, 2 o 3 para el core dos, y 4 o 5 para el core tres. Salvando esta diferencia, el scheduler va a hacer básicamente lo mismo que hace para el Intel Core 2 Duo. Los subsistemas van en hilos en hardware, cualquier hilo extra puede ser insertado en cualquiera de los cores pero hay que balancear la utilización de cores.

Por último la política de scheduling para el Cell es totalmente diferente y mucho más compleja. En este caso los trabajos enviados a cada SPE van a tener que ser mucho más pequeños que mandar todo lo que hace un subsistema. Por lo tanto el scheduler va a residir en el PPE y se va a tener que encargar de que el motor use todos los SPEs. Para hacer esto el scheduler va a necesitar información en tiempo real del tiempo de ejecución de un hilo en cada SPE. Básicamente el modelo a seguir un modelo mostrado en [14] donde el scheduler tiene un queue FIFO (First In First Out) de trabajos donde los trabajos se van ejecutando en los SPEs con menos carga (Figura 5). Es muy importante que los trabajos que agreguemos al queue sean lo más atómicos posible. La razón para esto reside en el cache y capacidad de procesamiento de los SPEs. Como se vio en la sección de microprocesadores los SPEs tienen un cache pequeño y además las instrucciones se ejecutan de manera ordenada por lo que ejecutar trabajos grandes causa problemas de performance rápidamente. Lo mejor es tener los trabajos más pequeños posibles de tal manera que el SPE se libere de trabajo lo antes posible.

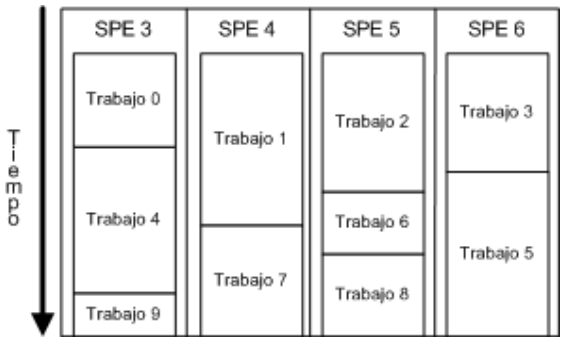


**Figura 5: Ejemplo del sistema de scheduling del Chromaticity Engine. El PPE va mandando los diferentes trabajos que tiene en el queue según van completando en los SPEs.**

## Portabilidad y Mantenimiento

Mantener un motor 3D para múltiples plataformas es una tarea complicada en sí mismo, por lo tanto es sumamente importante diseñar el motor teniendo en cuenta la portabilidad y mantenimiento. Los entornos de desarrollo y APIs disponibles para las diferentes arquitecturas de microprocesadores que analizamos son diferentes entre sí. Por lo tanto si aspiramos a mantener la cantidad de versiones del motor 3D al mínimo es necesario crear un diseño de motor donde el código especifico para cada plataforma se mantenga lo mas separado posible del resto del motor. Por

ejemplo no sería bueno tener diferentes forks o usar sistemas de defines simplemente para crear un hilo de ejecución nuevo simplemente porque las plataformas usan APIs diferentes. Es necesario abstraer todas las operaciones específicas de tal manera que el scheduler pueda crear un hilo de ejecución de manera transparente sin necesidad de saber cómo crear este hilo en la plataforma donde el motor 3D se está utilizando. Cuando todo el código esta abstraído de una manera modular es más fácil mantener una sola versión del motor, portar a otras plataformas, e incluso hace testing unitario. Obviamente esto implica tener en cuenta las diferentes APIs para las diferentes plataformas, por eso por ejemplo el soporte para la creación de hilos de ejecución se baso en Boost Threads [15]. El resultado no es una clase que soporta todo lo que una librería tendría sino que es el conjunto de operaciones mínimas para obtener los resultados deseados en todas las plataformas. Una vez que tenemos todo abstraído el resto del motor va utilizar esta capa y en ninguno de los casos va a utilizar funciones específicas a una plataforma.

**Resultados**

El Chromaticity Engine fue testeado bajo diferentes plataformas aunque no pudo ser probado en todas las plataformas que se nombraron. Para probar la performance se creó una escena con 500,000 objetos dinámicos cada uno con 18 triángulos. En ningún momento se renderizo ni video ni sonido para evitar influenciar los resultados con valores que fluctúan basados en el GPU. El motor en este caso tenía que mantener un scene graph con toda la geometría de la escena y además el motor realizaba consultas a un octree que subdividía todo el espacio.

En la versión de PC el motor fue probado en Windows Vista en un Core 2 Duo E6700, Core 2 Quad QX6600 y AMD Athlon 64 X2 6000+. En estos casos el ganador fue el Core 2 Quad QX6600 por el uso de los cuatro cores (Figura 6). Como se menciono anteriormente, no es posible especificar con seguridad que core utilizaron cada hilo creado.

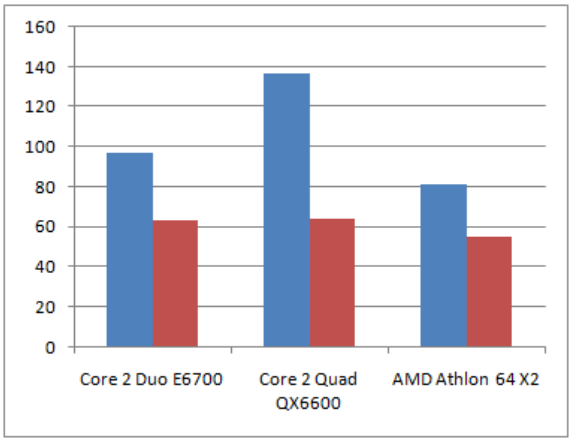


**Figura 6: Performance de los diferentes microprocesadores en Hertz. En azul es la performance del motor corriendo con múltiples hilos, y el rojo el motor corriendo en un solo hilo.**

Las ganancias en velocidad fueron notables comparado con el motor corriendo en un solo core con un solo hilo. Como se ve la velocidad en un solo hilo llega a un techo como [10] argumenta.

En la versión para el Cell Broadband Engine, el motor se corrió en una PlayStation 3 sobre Yellow Dog Linux 5.0. En este caso la diferencia entre correr usando un hilo en el PPE y un SPE, contra usar dos hilos en el PPE y los SPEs fue notable. Esto tiene que ver con el hecho de que los SPEs son sumamente especializados y con ciertas limitaciones.

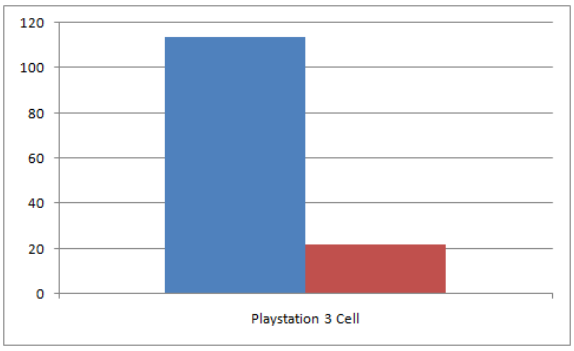


**Figura 7: Performance del PlayStation 3 Cell en Hertz. En azul es la performance del motor corriendo en el PPE y los SPEs, y el rojo el motor corriendo en un solo hilo en el PPE y usando un solo SPE.**

En la versión para el Xenon no pudimos probar realmente la performance ya que para desarrollar en el Xenon es necesario tener el XDK de Microsoft que esta solo disponible para los estudios de video juegos y los creadores de middleware.

## Conclusión

El Chromaticity Engine es un motor 3D que fue escrito en respuesta a los cambios en las plataformas donde tiene que ejecutarse. El cambio fundamental se produjo en los microprocesadores utilizados en estas plataformas. Se paso de disponer un solo hilo en hardware a disponer como mínimo con dos cores. No solo esto sino que los cores mantuvieron la misma velocidad, esto implica que ya no vamos a obtener más ganancias de performances usando un solo hilo [11]. Por eso si hay que escribir un motor 3D para aplicaciones visuales es necesario crear una arquitectura multihilo que sea fácil de mantener y portar a diferentes plataformas. Gracias a los modelos de paralización y la arquitectura expuesta en si es posible escribir motores 3D que aprovechan las arquitecturas de los diferentes microprocesadores sin necesariamente crear un motor sumamente complejo y complicado de mantener.

A futuro va a ser necesario validar mejor la arquitectura ya que la arquitectura expuesta fue probada en una pequeña cantidad de plataformas. En especial hay que analizar el trabajo desde el punto de vista de las plataformas para visualización científica donde las necesidades de visualización no son necesariamente diferentes pero las plataformas donde corren son totalmente diferentes.

### Referencias

[1] O. Wechsler. Inside Intel® Core™ Microarchitecture: Setting new standards for energy-efficient performance, 2006.
[2] J. Doweck. Inside Intel® Core™ Microarchitecture and Smart Memory Access: An in-depth look at Intel innovations for accelerating execution of memory-related instructions, 2006.
[3] R.M. Ramanathan. Intel® Multi-Core Processors: Making the move to Quad-Core and beyond, 2006.
[4] J. Brown. Application-customized CPU Design: The Microsoft Xbox 360 CPU story, 2005.
[5] J. Andrews, N. Baker. Xbox 360 System Architecture. IEEE Micro Volume 26, Issue 2, 2006.
[6] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. IEEE Micro Volume 26, Issue 2.
[7] J.A. Kahle. Introduction to the Cell Multiprocessor, 2005.
[8] D. Pham. The Design and Implementation of a First Generation Cell Processor, 2005.
[9] D.A. Brokenshire. Maximizing the Power of the Cell Broadband Engine Processor: 25 tips to optimal application performance, 2006.
[10] H. Sutter. A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal, 2005.
[11] A. El Rhalibi, D. England, S. Costa. Game Engineering for a Multiprocessor Architecture, 2005.
[12] B. Dawson. Coding For Multiple Cores on Xbox 360 and Microsoft Windows, 2006.
[13] Cell Broadband Engine Programming Handbook, 2006.
[14] D. Mallinson, M. DeLoura. CELL: A New Platform for Digital Entertainment, 2005.
[15] W.E. Kempf. The Boost C++ Libraries: Boost Threads, 2003.

**Datos de Contacto:**
*Pablo Zurita. Universidad Tecnológica Nacional, Facultad Regional Córdoba. E-Mail: pablo@pablo-zurita.com.ar*