ESCUELA SUPERIOR POLITÉCNICA DE CHIMBORAZO

FACULTAD DE INFORMÁTICA Y ELECTRÓNICA

ESCUELA DE INGENIERÍA EN SISTEMAS

"ANÁLISIS DE LA PROGRAMACIÓN CONCURRENTE SOBRE LA CPU Y GPU EN EL DESARROLLO DE FRACTAL BUILD"

TESIS DE GRADO

Previa la obtención del Título de

INGENIERO EN SISTEMAS INFORMÁTICOS

Presentado por:

José Vicente Anilema Guadalupe

Riobamba-Ecuador

2012

AGRADECIMIENTOS

Primero que todo, Gracias a Dios por haberme llenado de experiencias nuevas en esta etapa que culmina.

Un agradecimiento al Tribunal de Tesis, Dr. Alonso Álvarez e Ing. Jorge Menéndez que con el conocimiento y la capacidad supieron guiarme en los procesos más difíciles de esta tesis.

DEDICATORIA

Con todo mi cariño dedico esta tesis a mis padres, Carmen y José que con el sacrificio y perseverancia diaria me enseñaron que siempre tengo que luchar por mis ideales y sueños.

A mis amigas y amigos que siempre me acompañan en los buenos y malos momentos, demostrándome con una sonrisa la verdadera solución a un problema.

FIRMAS DE RESPONSABILIDAD Y NOTA DEL TRIBUNAL

NOMBRES	**FIRMAS**	**FECHA**
Ing. Iván Menes **DECANO DE LA FACULTAD** **INFORMÁTICA Y ELECTRÓNICA**	_____	_____
Ing. Raúl Rosero **DIRECTOR DE ESCUELA** **INGENIERÍA EN SISTEMAS**	_____	_____
Dr. Alonso Álvarez **DIRECTOR DE TESIS**	_____	_____
Ing. Jorge Menéndez **MIEMBRO DE TESIS**	_____	_____
Tlgo. Carlos Rodríguez **DIRECTOR DEL CENTRO DE** **DOCUMENTACIÓN**	_____	_____

NOTA: _____

FIRMA DE RESPONSABILIDAD DEL AUTOR

"Yo, José Vicente Anilema Guadalupe, soy responsable de las ideas, doctrinas y resultados expuestos en esta Tesis; y, el patrimonio intelectual de la Tesis de Grado pertenece a la ESCUELA SUPERIOR POLITÉCNICA DE CHIMBORAZO"

_____

José Vicente Anilema Guadalupe

INDICE DE ABREVIATURAS Y ACRÓNIMOS

API Applicaction Programming Interface(Interfaz de Programación de Aplicaciones)

CPU Central Processing Unit (Unidad Central de Proceso)

FPS Frames Per Second (Fotogramas por segundo)

FLOPS Floating point Operations per Second

GB/s Gigabytes per second

GDDR Graphics Double Data Rate

GLSL OpenGL Shading Language (Lenguaje de Sombreado de OpenGL)

GPGPU General-Purpose Computing on Graphics Processing Units

GPU Graphic Processing Unit (Unidad de Procesamiento Gráfico)

HLSL High Level Shader Language (Lenguaje de Sombreador de Alto Nivel)

Hz Hertz

MIMD Multiple Instruction Multiple Data(Múltiples Instrucciones, Múltiples Datos)

OpenCL Open Computing Language

PC	Personal Computer (Computador Personal)
RAM	Random Access Memory (Memoria de Acceso Aleatorio)
SIMD	Single Instruction Multiple Data (Única Instrucción, Múltiples Datos)
SM	Stream Multiprocessor
SP	Stream Processor
SO	Sistema Operativo

INDICE GENERAL

INDICE FIGURAS

INDICE DE TABLAS

CAPÍTULO I

MARCO REFERENCIAL

1.1 Antecedentes

Hoy en día nos encontramos en una situación límite en el cumplimiento de la ya conocida *Ley de Moore* sobre el rendimiento hardware. Se ha alcanzado el rendimiento pico en CPUs mono procesadores, comenzando a utilizar los nuevos transistores disponibles para construir multiprocesadores, de doble o cuádruple núcleo.

Esto indica que estamos en una generación de ordenadores *de muchos núcleos* en los cuales los algoritmos secuenciales tradicionales perderán importancia frente a los algoritmos que permitan el procesamiento paralelo de datos.

Si bien pensar en este tipo de procesadores actualmente está vinculado a grandes cantidades de dinero, existe una posibilidad de bajo coste que resulta novedosa en el área de la programación de algoritmos paralelos: utilizar la enorme potencia de cálculo y el gran número de procesadores de las tarjetas gráficas (en adelante GPUs, de

Graphics Processing Units) para llevar a cabo la ejecución de tareas no vinculadas con actividades gráficas, es decir llevar a cabo lo que se conoce como programación de propósito general sobre GPUs (GPGPU).

Las tarjetas actuales dedican la mayor parte de sus transistores a crear unidades de procesamiento más que al control de flujos de datos. En esta línea, una tarjeta gráfica de coste medio puede tener alrededor de 128 procesadores, con capacidad de ejecutar tanto tareas inherentes al renderizado de imágenes como programas de propósito general, todo ello de forma paralela.

Estas condiciones de ejecución hacen atractivo establecer una competición entre el rendimiento de una CPU (Unidad Central de Proceso, procesador clásico) y una GPU (Unidad de Proceso Gráfico). En términos generales, los algoritmos a ejecutar en una GPU van a ser versiones paralelas de los que se ejecutan en la CPU, pues es esta la forma de conseguir un mayor rendimiento de la misma.

Por otra parte, muchas aplicaciones que trabajan sobre grandes cantidades de datos se pueden adaptar a un modelo de programación paralela, con el fin de incrementar el rendimiento en velocidad de las mismas. La mayor parte de ellas tienen que ver con el procesamiento de imágenes y datos multimedia, como la codificación/descodificación de vídeos y audios, el escalado de imágenes, y el reconocimiento de patrones en imágenes, etc.

En la actualidad existen varias APIs que permiten trabajar de forma más amigable con la GPU al momento de desarrollar una aplicación de propósito general, como son: CUDA propietaria de Nvidia, ATI Stream y OpenCL. Esta última constituye una

propuesta libre, de tal forma que una aplicación elaborada con esta API podrá ser ejecutada en cualquier hardware gráfico.

Es en las aplicaciones que requieren gran poder de cómputo en donde se hace atractiva la utilización de los procesadores gráficos como un co-procesador gracias a su gran capacidad de computación paralela. Uno de los más claros ejemplos de esto, es en la generación de fractales y específicamente la construcción de edificios fractales.

En la Escuela de Ingeniería en Sistemas se dicta la materia de Computación Gráfica, uno de los temas tratados en dicha materia es la creación de fractales, para lo cual se requiere de una aplicación real de los fractales. Como resultado de la tesis "ANÁLISIS DE LA PROGRAMACIÓN CONCURRENTE SOBRE LA CPU Y GPU EN EL DESARROLLO DE FRACTAL BUILD" se desarrollará una aplicación gráfica (Fractal Build) para la generación de un edificio 3D en base a fractales, que sirva como aplicación real de los fractales para la materia antes mencionada.

1.2 Justificación

1.2.1 Justificación Teórica

La programación de GPU para aplicaciones de propósito general ha abierto un nuevo e interesante campo de desarrollo, una oportunidad para explotar soluciones basadas en un paralelismo de datos masivo sin necesidad de recurrir a clústeres de ordenadores o supercomputadores, sino haciendo uso de los procesadores gráficos.

A medida que el número de núcleos de proceso en las GPU's se han ido incrementando, y ganando en rendimiento al operar con datos en coma flotante, se

hizo cada vez más patente la necesidad de aprovechar esa potencia bruta de cálculo para propósitos alternativos, aparte de la evidente aplicación en videojuegos de última generación.

1.2.2 Justificación Aplicativa

Actualmente dentro de uno de los temas dictados en la materia de Computación Gráfica de la Escuela de Ingeniería en Sistemas está la generación de fractales, mediante la investigación propuesta se desarrollará una aplicación para la generación de un edificio fractal en 3D que servirá como herramienta para esta materia al ser una aplicación real de los fractales.

1.3 Objetivos

1.3.1 Objetivo General

Realizar el estudio estadístico entre la programación concurrente sobre CPU y GPU para mejorar el rendimiento y tiempo de respuesta en la aplicación gráfica Fractal Build para la generación de un edificio 3D en base a fractales.

1.3.2 Objetivos Específicos

- Estudiar cómo se desarrolla la programación concurrente en CPU y GPU.
- Determinar los parámetros y pruebas para realizar el estudio estadístico entre la programación concurrente en CPU y GPU.
- Desarrollar la aplicación gráfica para la generación de un edificio 3D en base a fractales, la cual requiere un gran poder de cómputo.

1.4 Hipótesis

La comparación estadística entre la programación concurrente CPU y GPU permitirá escoger la mejor tecnología para desarrollar la aplicación gráfica para la generación de un edificio 3D en base a fractales.

CAPÍTULO II

MARCO TEÓRICO

2.1 Introducción

Hasta hace poco, uno de los mejores métodos para mejorar el rendimiento de una aplicación, era colocarla en un equipo con un procesador más potente, que por lo general era un procesador con una frecuencia de reloj más alta. Pero actualmente, esto no sirve del todo, ya que la tendencia actual para construir los procesadores es incluir a éstos un mayor número de núcleos, y no sólo aumentar su frecuencia del reloj. De aquí que se hace necesario poder aprovechar la capacidad de cómputo de esos núcleos y esto es mediante la concurrencia y el paralelismo.

Para aplicaciones que requerían gran capacidad de cálculo era necesario disponer de equipos potentes o superordenadores. Pero en este sentido, es decir potencia de cálculo, existen otros tipos de procesadores que no requieren de una gran inversión, éstos son los procesadores gráficos, los cuales están diseñados con una gran cantidad de núcleos especializados.

Para aprovechar esta potencia de cálculo nació la Programación de Propósito General sobre Unidades de Procesamiento Gráfico ó más conocida como GPGPU.

2.2 Conceptos

2.2.1 Proceso

El concepto central de todo sistema operativo es el proceso: abstracción de un programa en ejecución. Todo lo demás se organiza en relación a este concepto. Los ordenadores modernos realizan varias tareas al mismo tiempo. Por ejemplo, mientras se ejecuta un programa de usuario, el ordenador puede estar leyendo del disco e imprimiendo un documento en la impresora.

2.2.2 Ejecución Serial

Las tareas o instrucciones de un programa se ejecutan de forma secuencial, es decir una tras otra, una a la vez. En un sistema multitarea, la CPU alterna de un programa a otro, ejecutando cada uno durante milisegundos, y conmutando a otro inmediatamente, de tal forma que al usuario se le proporciona cierta sensación de ejecución paralela[1], como si el ordenador realizase varias tareas al mismo tiempo.

[1] Al menos esto era cierto hasta hace algunos años. Con la aparición de los sistemas multinúcleo se puede llegar a tener de un verdadero paralelismo entre tareas

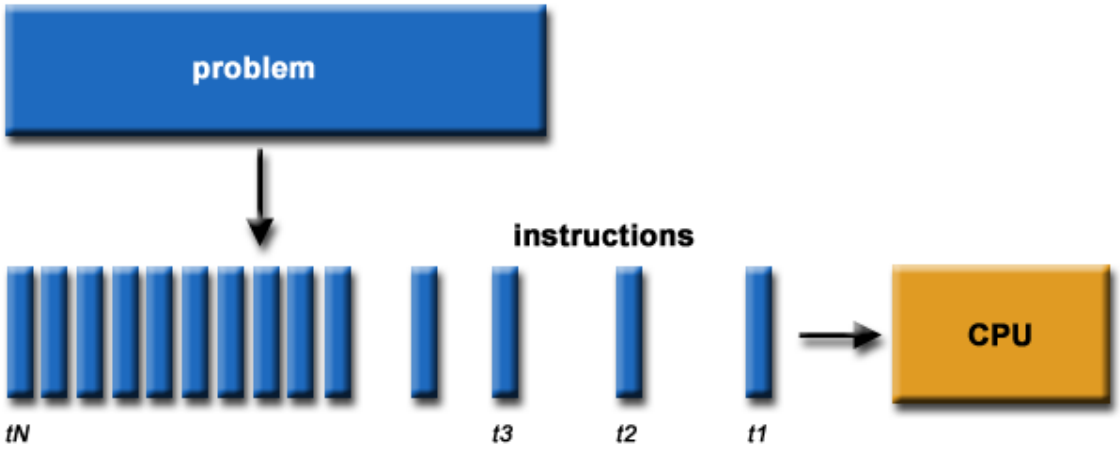


Figura.I.1. Programación Secuencial

2.2.3 Ejecución Paralela

Varias tareas o instrucciones de un programa son ejecutados de manera simultánea. Para esto cada una de estas tareas se ejecuta en un CPU diferente(Figura 2).

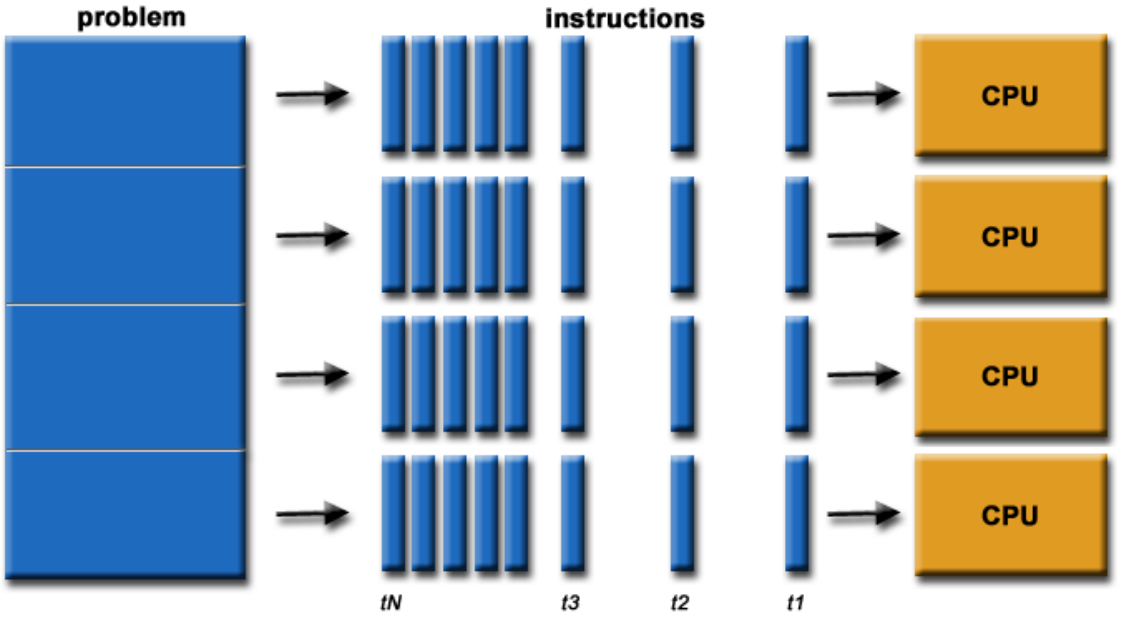


Figura.II.2. Programación Concurrente y Paralela

La diferencia entre proceso y programa es sutil, pero crucial. El programa es el algoritmo expresado en una determinada notación, mientras que el proceso es la actividad, que tiene un programa, entrada, salida y estado. Una CPU puede compartirse entre varios procesos empleando un algoritmo de planificación, que determina cuando se debe detener un proceso en ejecución para dar servicio a otro distinto.

2.2.4 Paralelismo

Parece claro que a pesar de los avances tecnológicos conseguidos en los últimos años, la tecnología del silicio está llegando a su límite. Si se quieren resolver problemas más complejos y de mayores dimensiones se deben buscar nuevas alternativas tecnológicas. Una de estas alternativas en desarrollo es el paralelismo. Mediante el paralelismo se pretende conseguir la distribución del trabajo entre las diversas CPU disponibles en el sistema de forma que realicen el trabajo simultáneamente, con el objetivo de aumentar considerablemente el rendimiento total.

Existen dos visiones del paralelismo:

- A nivel ***hardware***: Es el paralelismo definido por la arquitectura de la máquina.

- A nivel ***software***: Es el paralelismo definido por la estructura del programa. Se manifiesta en las instrucciones que son independientes y consiste en el diseño de algoritmos que se ejecuten requiriendo el menor tiempo posible.

El paralelismo se presenta, a su vez, en dos formas:

- Paralelismo *de control*: Se pueden realizar dos o más operaciones simultáneamente. Se presenta en los pipelines y las múltiples unidades

funcionales. El programa no necesita preocuparse de este paralelismo, pues se realiza a nivel hardware.

- Paralelismo *de datos*: Una misma operación se puede realizar sobre varios elementos simultáneamente.

En relación al paralelismo hardware, Michael Flynn realizó la siguiente clasificación de arquitecturas de computadores:

- ***SISD*** *(Simple Instruction Simple Data)*: Un flujo simple de instrucciones opera sobre un flujo simple de datos, es el modelo clásico de Von Neumann. Es la arquitectura de las máquinas secuenciales convencionales de un sólo procesador.
- ***SIMD*** *(Simple Instruction Multiple Data)*: Un flujo simple de instrucciones opera sobre múltiples flujos de datos; todos los procesadores ejecutan la misma instrucción aunque sobre distintos datos. Permite un gran número de procesadores. Es la arquitectura de las computadoras con hardware para proceso vectorial.
- ***MISD*** *(Multiple Instruction Simple Data):* Es la arquitectura de las computadoras que poseen un conjunto de procesadores que ejecutan diferentes instrucciones sobre los mismos datos. Es teóricamente equivalente al tipo SISD.

- ***MIMD*** *(Multiple Instruction Multiple Data)*: Es la arquitectura más genérica para los computadores paralelos, ya que es aplicable a cualquier tipo de problema, al contrario que las dos anteriores, en donde, cada procesador ejecuta su propio código sobre datos distintos a los de otros procesadores.

		Datos	
		Simples	**Múltiples**
Instrucciones	**Simples**	SISD	SIMD
	Múltiples	MISD	MIMD

Tabla.II.I. Clasificación del Paralelismo

(Michael Flynn)

2.2.5 Concurrencia

Se puede definir a la Programación concurrente como las notaciones y técnicas empleadas para expresar el paralelismo potencial y para resolver los problemas de comunicación y sincronización resultantes. La programación concurrente proporciona una abstracción sobre la que estudiar el paralelismo sin tener en cuenta los detalles de implementación. Esta abstracción ha demostrado ser muy útil en la escritura de programas claros y correctos empleando las facilidades de los lenguajes de programación modernos.

Existen dos formas de concurrencia:

- Concurrencia *implícita*: Es la concurrencia interna al programa, por ejemplo cuando un programa contiene instrucciones independientes que se pueden realizar en paralelo, o existen operaciones de Entrada/Salida que se pueden realizar en paralelo con otros programas en ejecución. Está relacionada con el paralelismo hardware.

- Concurrencia *explícita*: Es la concurrencia que existe cuando el comportamiento concurrente es especificado por el diseñador del programa. Está relacionada con el paralelismo software.

2.2.5.1 Características de la Concurrencia

Los procesos concurrentes tienen las siguientes características:

- *Indeterminismo*: Las acciones que se especifican en un programa secuencial tienen un orden total, pero en un programa concurrente el orden es parcial, ya que existe una incertidumbre sobre el orden exacto de ocurrencia de ciertos sucesos, esto es, existe un indeterminismo en la ejecución. De esta forma si se ejecuta un programa concurrente varias veces puede producir resultados diferentes partiendo de los mismos datos.

- *Interacción entre procesos*: Los programas concurrentes implican interacción entre los distintos procesos que los componen:
  - Los procesos que comparten recursos y compiten por el acceso a los mismos.
  - Los procesos que se comunican entre sí para intercambiar datos.

  En ambas situaciones se necesita que los procesos sincronicen su ejecución, para evitar conflictos o establecer contacto para el intercambio de datos. La interacción entre procesos se logra mediante variables compartidas o bien mediante el paso de mensajes. Además la interacción puede ser explícita, si aparece en la descripción del programa, o implícita, si aparece durante la ejecución del programa.

- *Gestión de recursos*: Los recursos compartidos necesitan una gestión especial. Un proceso que desee utilizar un recurso compartido debe solicitar dicho

recurso, esperar a adquirirlo, utilizarlo y después liberarlo. Si el proceso solicita el recurso pero no puede adquirirlo en ese momento, es suspendido hasta que el recurso está disponible. La gestión de recursos compartidos es problemática y se debe realizar de tal forma que se eviten situaciones de retraso indefinido (espera indefinidamente por un recurso) y de deadlock (bloqueo indefinido o abrazo mortal).

- *Comunicación*: La comunicación entre procesos puede ser síncrona, cuando los procesos necesitan sincronizarse para intercambiar los datos, o asíncrona, cuando un proceso que suministra los datos no necesita esperar a que el proceso receptor los recoja, ya que los deja en un buffer de comunicación temporal.

2.3 CPU. Unidad Central de Proceso

2.3.1 Conceptos

La unidad central de procesamiento o CPU (por el acrónimo en inglés de Central Processing Unit), o simplemente el procesador o microprocesador, es el componente central del computador, que interpreta las instrucciones contenidas en los programas y procesa los datos. Los CPU proporcionan la característica fundamental de la computadora digital, es decir, la programabilidad y son uno de los componentes necesarios encontrados en las computadoras de cualquier tipo, junto con el almacenamiento primario y los dispositivos de entrada/salida[20].

Se conoce como microprocesador al CPU que es manufacturado con circuitos integrados. Desde mediados de los años 1970, los microprocesadores de un solo chip

han reemplazado casi totalmente todos los tipos de CPU, y hoy en día, el término "CPU" es aplicado usualmente a todos los microprocesadores.

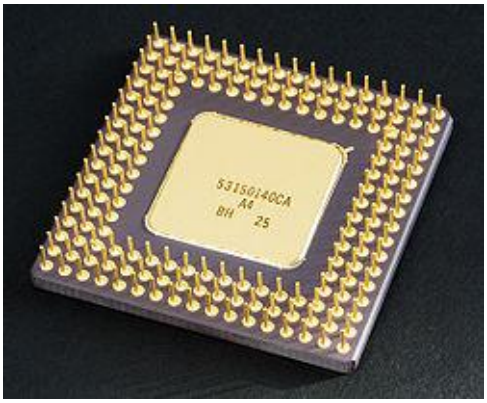


Figura.II.3. CPU

La expresión "unidad central de proceso" es, en términos generales, una descripción de una cierta clase de máquinas de lógica que pueden ejecutar complejos programas de computadora. Esta amplia definición puede fácilmente ser aplicada a muchos de los primeros computadores que existieron mucho antes que el término "CPU" estuviera en amplio uso. Sin embargo, el término en sí mismo y su acrónimo han estado en uso en la industria de la informática por lo menos desde el principio de los años 1960. La forma, el diseño y la implementación de los CPU ha cambiado drásticamente desde los primeros ejemplos, pero su operación fundamental ha permanecido bastante similar.

2.3.2 Historia

Los primeros CPU fueron diseñados a la medida como parte de una computadora más grande, generalmente una computadora única en su especie. Sin embargo, este costoso método de diseñar los CPU a la medida, para una aplicación particular, ha desaparecido en gran parte y se ha sustituido por el desarrollo de clases de procesadores baratos y estandarizados adaptados para uno o muchos propósitos. Esta

tendencia de estandarización comenzó generalmente en la era de los transistores discretos, computadoras centrales, y microcomputadoras, y fue acelerada rápidamente con la popularización del circuito integrado, éste ha permitido que sean diseñados y fabricados CPU más complejos en espacios pequeños (en la orden de milímetros). Tanto la miniaturización como la estandarización de los CPU han aumentado la presencia de estos en dispositivos digitales en la vida moderna mucho más allá de las aplicaciones limitadas de máquinas de computación dedicadas. Los microprocesadores modernos aparecen en todo, desde automóviles, televisores, neveras, calculadoras, aviones, hasta teléfonos móviles o celulares, juguetes, entre otros.

2.3.3 Tendencia Actual

Todos los ordenadores cuentan con una CPU o microprocesador (en ocasiones con varios) y, desde hace unos años, dichos circuitos integrados dejaron de competir en velocidad bruta, regida únicamente por la frecuencia de funcionamiento, para incrementar su potencia por otra vía: la integración de *múltiples núcleos*. Una configuración hardware con un único microprocesador de 4 núcleos resulta mucho más barata que otra que cuente con 4 microprocesadores (que también existen) de un único núcleo, ya que en la placa base es necesario un único zócalo y el número de interconexiones es también mucho menor, aparte de que fabricar un único *chip* es más barato que fabricar cuatro. La potencia de estos dos hipotéticos sistemas no es idéntica, pero sí muy similar.

Además de varios núcleos ciertos fabricantes, como es el caso de Intel con su tecnología Hyperthreading, diseñan sus microprocesadores de manera que cada núcleo (duplicando ciertas unidades operativas) tiene capacidad para ejecutar dos hilos simultáneamente. Con un microprocesador de 4 núcleos podrían ejecutarse 8

hilos con paralelismo real (sin recurrir a técnicas de tiempo compartido). Este fabricante ya cuenta con microprocesadores que ofrecen 6 núcleos/12 hilos y en muestras de tecnología futura, puesta a disposición de universidades, experimenta con un *chip* que dispone de 48 núcleos. Es fácil darse cuenta de que en pocos años tendremos ordenadores cuyas CPU integrarán decenas sino cientos de núcleos, equivalente cada uno de ellos a un procesador tipo *Pentium*.

2.3.4 Modelo de Programación

Los ordenadores no tendrían utilidad alguna si no existiesen programadores que creasen software que les hiciese funcionar y ofrecer a los usuarios finales aquello que necesitan. Dicho software puede tomar distintas formas, desde el firmware que se incluye en ROM (o alguna variante PROM/EPROM/etc.) y se encarga de la puesta en marcha y ofrece los servicios más básicos hasta las aplicaciones de control de procesos o diseño asistido, pasando por los sistemas operativos y los propios compiladores e intérpretes de multitud de lenguajes.

En los primeros ordenadores el software se ejecutaba en forma de procesos por lotes: cada programador ponía su tarea en cola y esperaba a que llegase su turno para obtener la salida correspondiente. Cada tarea era intrínsecamente secuencial, en el sentido de que no ejecutaba más de una instrucción de manera simultánea. Posteriormente llegaron los sistemas de tiempo compartido capaces de atender interactivamente a varios usuarios y, en apariencia, ejecutar múltiples tareas en paralelo, si bien la realidad era que el sistema operativo se encargaba de que el procesador fuese saltando de una tarea a otra cada pocos ciclos, consiguiendo esa ilusión de paralelismo o multitarea.

Para los programadores de un sistema operativo la implementación del tiempo compartido implicaba codificar algoritmos relativamente complejos, de los cuales el

más conocido es Round Robin, capaces de seleccionar en cada momento el proceso que ha de pasar a ejecutarse e impedir situaciones indeseadas como, por ejemplo, que una tarea no obtenga nunca tiempo de procesador. Con el tiempo los microprocesadores implementaron en hardware una gran cantidad de lógica que facilitaba el intercambio rápido de tareas, haciendo más fácil el trabajo de los programadores de sistemas.

Los programadores de aplicaciones, por el contrario, siguieron durante décadas diseñando software asumiendo que sus programas obtenían el control total del ordenador, implementando los algoritmos para que se ejecutasen secuencialmente de principio a fin y dejando que el sistema operativo los interrumpiese cada cierto tiempo para volverlos a poner en marcha un instante después, todo ello a tal velocidad que los usuarios tenían la sensación esperada: la aplicación les atendía de manera continua sin problemas. Esto es especialmente cierto en aquellos programas en los que existe comunicación con el usuario, ya que la mayor parte del tiempo se encuentran a la espera de una acción por parte de éste: una entrada por teclado, la pulsación de un botón, etc.

Dado que los microprocesadores solamente eran capaces de ejecutar una tarea en un instante dado, antes de que contasen con pipelines primero, varias unidades funcionales después (procesadores superescalares) y múltiples núcleos finalmente; los programadores de aplicaciones que querían ejecutar más de un trabajo en paralelo recurrían a diversos trucos, como el uso de interrupciones. De esta forma era posible, por ejemplo, imprimir un documento o realizar otra tarea lenta mientras se seguía atendiendo al usuario. De ahí se pasó a la programación con múltiples hilos o threads, de forma que un programa podía ejecutar múltiples secuencias de instrucciones en paralelo. Esos hilos eran gestionados por el sistema operativo mediante el citado algoritmo de tiempo compartido: el procesador seguía ejecutando únicamente una tarea en cada instante.

Aunque el multiproceso simétrico existe desde la década de los 60 en máquinas tipo mainframe, no fue hasta finales de los 90 cuando los servidores y estaciones de trabajo con zócalos para dos procesadores se hicieron suficientemente asequibles como para adquirir cierta popularidad. Estas máquinas contaban con dos procesadores y, en consecuencia, tenían capacidad para ejecutar dos tareas con paralelismo real. Obviamente los algoritmos de tiempo compartido seguían estando presentes, ya que el número de procesos en ejecución suele ser mucho mayor, pero el rendimiento era muy superior al ofrecido por los ordenadores personales.

El escenario de la computación personal ha cambiado drásticamente desde el inicio del nuevo milenio. Si en los años previos los fabricantes de microprocesadores competían casi exclusivamente en velocidad, lo que permitía al software aprovechar la mayor potencia sin trabajo adicional por parte del programador, en la última década han aparecido los microprocesadores multinúcleo y los procesadores gráficos con capacidades GPGPU (General-Purpose Computation on Graphics Hardware), lo que ha traído el final del Free Lunch para los programadores. Ahora aprovechar la potencia de un ordenador implica necesariamente el uso de todo ese paralelismo de una forma u otra.

A los microprocesadores multi-núcleo y las GPU habría que sumar una opción cada vez más alcance de cualquiera: los clúster de ordenadores. Si bien antes eran una opción reservada a centros de supercomputación, en la actualidad hay multitud de usuarios que disponen de varias máquinas conectadas en red local, lo cual abre las puertas (con el software adecuado) a la configuración en clúster para aprovechar el paralelismo y ejecutar software distribuyendo el trabajo entre múltiples máquinas.

En conjunto la popularización de estos mecanismos de paralelización implican la aparición de un nuevo modelo de diseño de software y el necesario reciclaje de los programadores. Ya no basta con escribir sentencias que se ejecutarán una tras otra, sin

más, siendo preciso planificar una arquitectura de mayor complejidad si se quiere obtener el mayor beneficio del hardware disponible.

2.4 GPU. Unidad de Procesamiento Gráfico

2.4.1 Introducción

La unidad de procesamiento gráfico o GPU (acrónimo del inglés Graphics Processing Unit) es un procesador dedicado exclusivamente al procesamiento de gráficos, para aligerar la carga de trabajo al procesador central en aplicaciones como los videojuegos y/o aplicaciones 3D interactivas. De esta forma, mientras gran parte de lo relacionado con los gráficos se procesa en la GPU, la CPU puede dedicarse a otro tipo de cálculos. Las GPU modernas son muy eficientes a la hora de mostrar gráficos de alta resolución gracias a su forma paralela de trabajar.

Figura.II.4. Tarjeta Gráfica

Una GPU implementa ciertas operaciones gráficas llamadas primitivas optimizadas para el procesamiento gráfico. Una de las primitivas más comunes para el

procesamiento gráfico en 3D es el antialiasing[2], que suaviza los bordes de las figuras para darles un aspecto más realista. Además existen primitivas para dibujar rectángulos, triángulos, círculos y arcos. Las GPU actualmente disponen de gran cantidad de primitivas, buscando mayor realismo en los efectos; incluyen también funciones relacionadas con el video digital y son capaces de soportar entornos 3D.

2.4.2 Evolución de los Procesadores Gráficos

Los procesadores gráficos han ido evolucionado considerablemente en las últimas dos décadas, por la incorporación de más capacidad de cálculo y programación con el objetivo de atender las crecientes demandas multimedia, ingeniería, aplicaciones de visualización científica y especialmente los videojuegos. Los primeros procesadores gráficos eran esencialmente dispositivos de función fija que implementaban un pipeline gráfico orientado a la rasterización.

La "revolución 3D" comenzó en el ámbito de los juegos de computadora a mediados de los 90', la tendencia ha sido trasladar el trabajo de procesamiento de gráficos tridimensionales, desde la CPU hacia la tarjeta de video.

En primer lugar se introdujo el filtro de las texturas mediante la función de Transformación e Iluminación (Transform & Lighting) ó T&L para lo cual se crearon chips especialmente dedicados para realizar esta tarea. Así nacieron las famosas placas aceleradoras 3D, que incorporaban dichos chips y una cantidad de memoria propia en

[2] Proceso que permite minimizar el aliasing cuando se desea representar una señal de alta resolución en un sustrato de más baja resolución. En la mayoría de los casos, consiste en la eliminación de la información de frecuencia demasiado elevada para poder ser representada.

la misma tarjeta. Luego, con la salida del GeForce 256 de NVIDIA, el procesador gráfico pasó a encargarse de lo que, hasta ese momento, realizaba la CPU.

El gran cambio se dio a partir de la incorporación de los Píxel shaders y Vertex shaders. Esto permitió a los programadores una mayor libertad a la hora de diseñar gráficos en tres dimensiones, ya que puede tratarse a cada píxel y cada vértice por separado. De esta manera, los efectos especiales y de iluminación puede crearse mucho más detalladamente, sucediendo lo mismo con la geometría de los objetos.

Y es mediante esta incorporación, la de los pixel y vertex shaders, es que los dispositivos modernos se han convertido en procesadores gráficos muy programables y ejecutan el software de forma muy similar a una CPU, cada generación de las GPU dependen cada vez más de grandes conjuntos de unidades de procesamiento completamente programables, en lugar del hardware con funciones fijas utilizadas en los dispositivos de la generación anterior.

En los últimos cinco años, se han alcanzado numerosos avances importantes en la adaptación de la arquitectura de hardware de la GPU para un mejor apoyo a la computación de propósito general, además de los avances en el campo de los gráficos 3D. La creciente necesidad de programación de software para el procesamiento geométrico y otras operaciones más allá del sombreado de las GPU modernas, condujo a la transición de dejar de usar efectos especiales de formatos numéricos a las representaciones de la máquina estándar para los números enteros y números en punto flotante, junto con una simple y doble precisión (IEEE-754) de punto flotante a la capacidad de la aritmética comparable a la proporcionada por la respectiva instrucción en la CPU.

2.4.3 Visión General de la Arquitectura

Desde la perspectiva física de estos dispositivos hardware se puede afirmar que una unidad de procesamiento gráfico o GPU se compone de un número limitado de multiprocesadores, cada uno de ellos consta de un conjunto de procesadores simples que operan de la forma Única Instrucción, Múltiples Datos o SIMD, es decir, todos los procesadores de un multiprocesador ejecutan de manera sincronizada la misma aritmética o la operación lógica al mismo tiempo, operando potencialmente sobre datos diferentes. Por ejemplo, la GPU de la generación más reciente GeForce GTX 590 con dos GPU y 1024 multiprocesadores, un total de 512 procesadores dentro de cada GPU. Su nivel de proceso alcanza casi los 2 Teraflops, lo que la convierte en la tarjeta más rápida y potente diseñada hasta el momento(de realizada la tesis). La GPU GTX 590 es la nueva generación de la arquitectura de NVIDIA Fermi, incorporado en la GeForce GTX serie 4x0 y la serie Tesla. La arquitectura de la GPU GTX590 es similar en muchos aspectos a sus predecesores G80 y G92, que fueron los primeros con soporte para NVIDIA CUDA (Compute Unified Device Architecture) para el cálculo de la GPU.

Los multiprocesadores dentro de un grupo de procesadores de textura se ejecutan completamente independiente el uno del otro, aunque comparten algunos recursos.

Un rasgo clave de la arquitectura de las GPUs modernas es la utilización de múltiples sistemas de memoria de alto ancho de banda para mantener la gran variedad de unidades de procesamiento con los datos suministrados. La arquitectura de la GTX690, la tarjeta más potente del mercado(2012), es compatible con un gran ancho de banda de (192 GB/s)[7]. El sistema de memoria principal es complementado con hardware dedicado a texturas que ofrecen las unidades de almacenamiento en caché de sólo lectura, con soporte de hardware para la localidad espacial multidimensional de referencia, de múltiples texturas de filtrado y modos de interpolación.

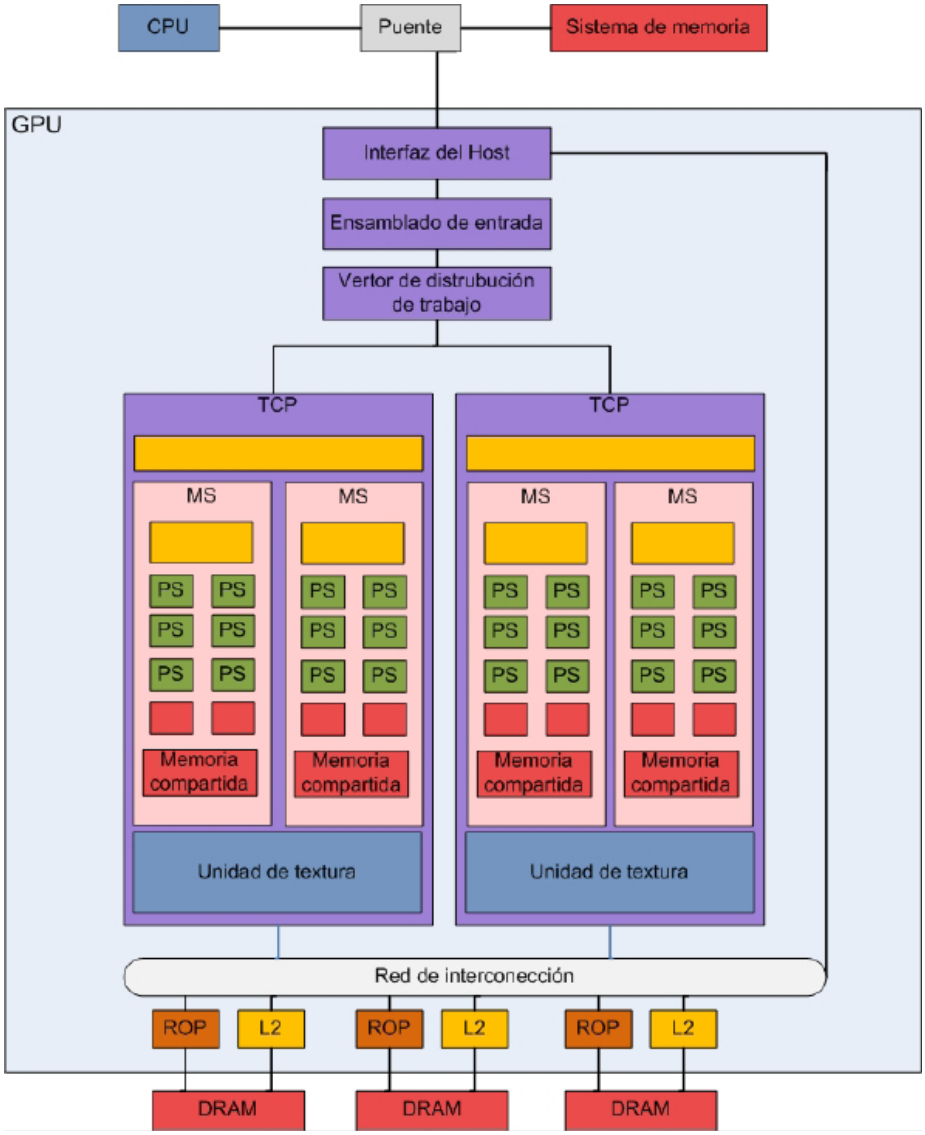


Figura.II.5. Arquitectura GPU

La figura 5 muestra la arquitectura "clásica" de una GPU NVidia en la cual se identifican los distintos elementos que la conforman:

- SM, o un stream multiprocessor, es la agrupación de SPs. En la generación anterior de GPUs NVidia(anterior a la generación Fermi[3]) consistía en la agrupación de 8 SPs.

[3] Es la arquitectura de GPUs creada por NVidia para las tarjetas gráficas Tesla, las cuales están orientadas a la HPC(High Performance Computing)

- SP, es un núcleo ó core dentro de una GPU, es el acrónimo de Stream Processor
- Bloque es una agrupación fija de hilos. Todos los bloques se definen con el mismo número de hilos. Además, un SM puede tener asignado varios bloques, pero un bloque se ejecuta en un solo SM.

La capacidad de 64 KB de caché constante es un medio eficaz de transmisión de sólo lectura de los elementos idénticos de datos a todos los temas dentro de un multiprocesador de transmisión a la velocidad de registro. El recuerdo constante puede ser una herramienta eficaz para lograr un alto rendimiento de los algoritmos que requieren recorrer y leer datos idénticos solamente. Los subprocesos que se ejecutan en el mismo multiprocesador(SM) pueden cargar y manipular los bloques de datos en este registro usando la rápida velocidad de la memoria compartida, evitando costosos accesos a la memoria global. Los accesos a la zona de memoria compartida son coordinados a través del uso de una barrera de sincronización del hilo de ejecución primitivo, garantizando que todas las peticiones han terminado sus actualizaciones de memoria compartida antes de iniciar las discusiones de otros resultados para acceder a la misma. La memoria compartida puede ser utilizado con eficacia como uno de los programas gestionados por la caché, la reducción de la utilización del ancho de banda y la latencia de las cargas repetitivas y tiendas para el sistema de memoria global.

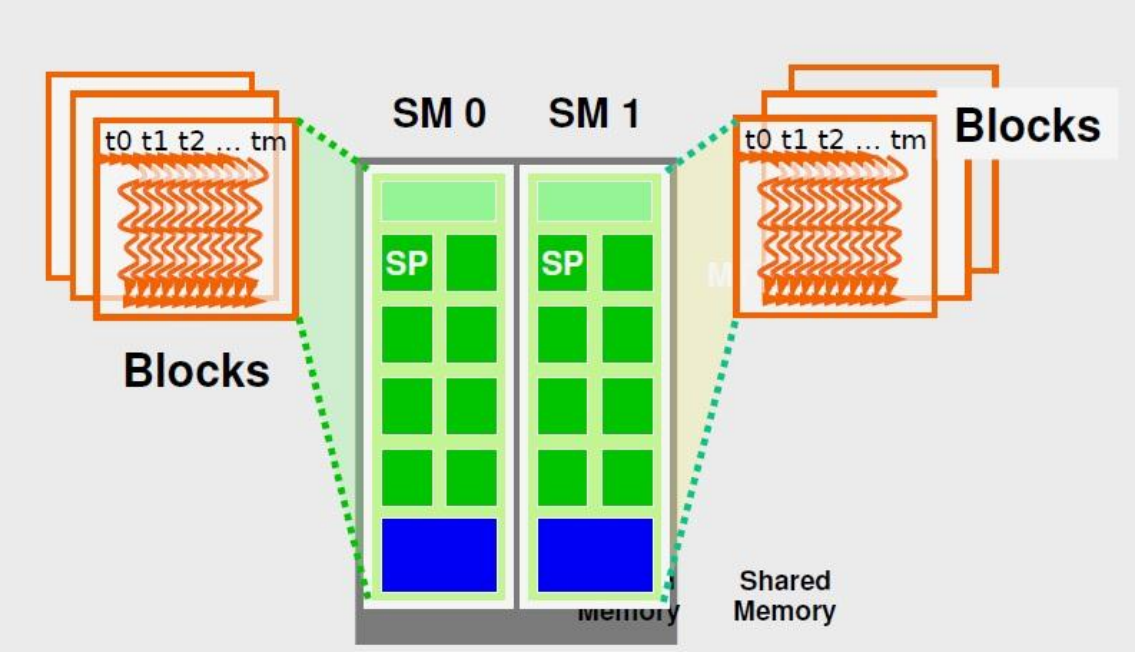


Figura.II.6. Bloque

2.4.4 Pipeline Gráfico (Etapas del Procesamiento gráfico)

En la figura 6 se muestra la estructura simplificada del renderizado del pipeline gráfico de la GPU. Donde un procesador de vértices recibe un vértice de dato y luego es capaz de convertir una imagen vectorial en una foto computarizada que es construida usando los fragmentos a cada localización de pixel cubierto por una primitiva geométrica simple.

Para lograr este objetivo el pipeline realiza las siguientes operaciones:

1. Procesar vértices.
2. Agrupar en primitivas simples.
3. Realizar las operaciones de, rotar, trasladar, escalar e iluminar, ó mejor conocido como T&L (Transform and Lighting)
4. Acotar texturas.
5. Convertir primitivas en mallas de puntos(Rasterizar).
6. Procesar píxeles.

7. Mezclar elementos (blending).

El resultado de todo este proceso se almacena en la memoria de video para posteriormente ser mostrado en el monitor.

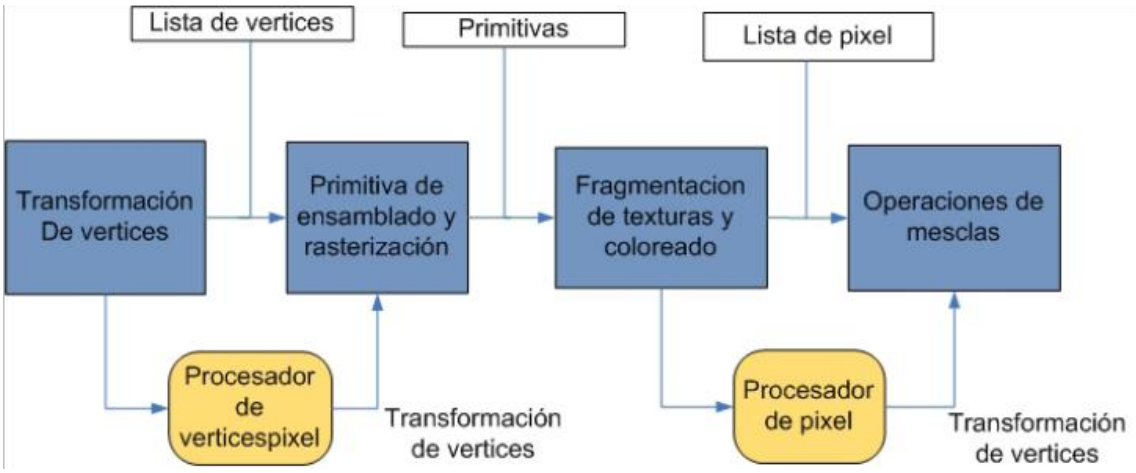


Figura.II.7. Pipeline Gráfico

2.4.5 Modelo de Memoria

Aparte de algunas unidades de memoria con un propósito especial en el contexto de los gráficos, cuentan con tres grandes tipos de memoria, la memoria local, la memoria compartida (Shared Memory) que es una unidad de memoria de acceso rápido que se comparte entre todos los procesadores de un multiprocesador, esta no solo puede ser utilizada por las variables locales, sino también para el intercambio de información entre los hilos de ejecución en diferentes procesadores del mismo multiprocesador. Esta memoria no puede ser utilizada para la información que se comparte entre los diferentes hilos de ejecución en los multiprocesadores. La memoria compartida es muy rápida, pero tiene una muy limitada capacidad (16 KB por multiprocesador). El tercer tipo de memoria es la memoria del dispositivo (Device Memory) ó memoria Global, que es la memoria RAM de video real de la tarjeta gráfica (se usa también para memoria de video, etc.), esta es significativamente y más lenta que la Memoria compartida [5].

En la Tabla 2 se muestra un resumen con los tipos de memoria de la GPU.

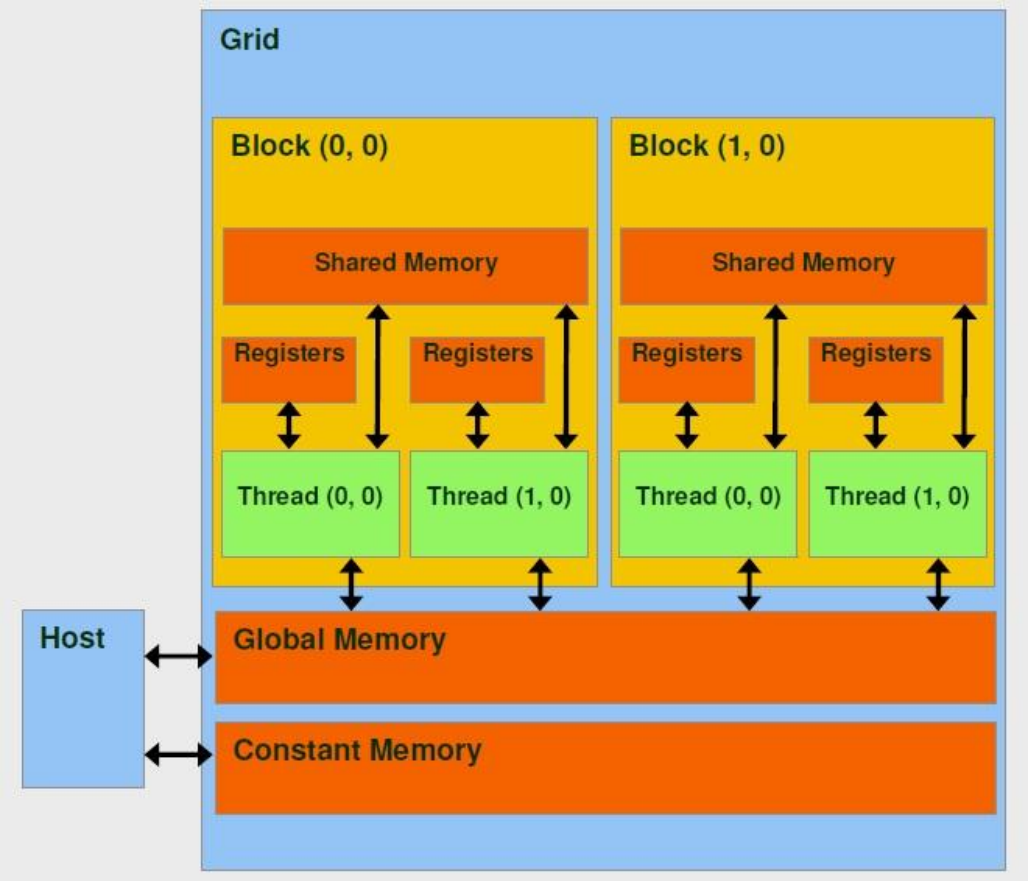


Figura.II.8. Distribución de la Memoria en la GPU

Cabe destacar que el ancho de banda para transferir datos entre la memoria del dispositivo y GPU (192 GB/s en la GTX 690) es mucho mayor que el de la CPU.

Tipo de memoria	Uso / Descripción	Accesible por
Global Memory	Para lectura y escritura	Los hilos y por la CPU
Local Memory	Es parte de la memoria global	privada para cada hilo(memoria virtual).
Constant Memory	• Global de solo lectura • Puede cargarse en caché de SM para acelerar las transferencias	Accesible para todos
Shared Memory	De tamaño limitado (16KB) (impuesto por los videojuegos).	Accesible para lectura y escritura por los hilos de un mismo bloque
Registro	Es el mismo número para todos los hilos	Accesible en lectura y escritura de forma privada

Tabla.II.II. Tipos de memoria

Elaborado por: Autor

2.4.6 Comparación con la CPU

Tradicionalmente las GPUs están diseñadas como unidades de procesamiento de rendimiento orientado al uso hardware sofisticado para programar la ejecución de decenas de miles de hilos simultáneamente en un gran número de unidades de procesamiento, o como su nombre lo indica una unidad de procesamiento gráfico de una computadora.

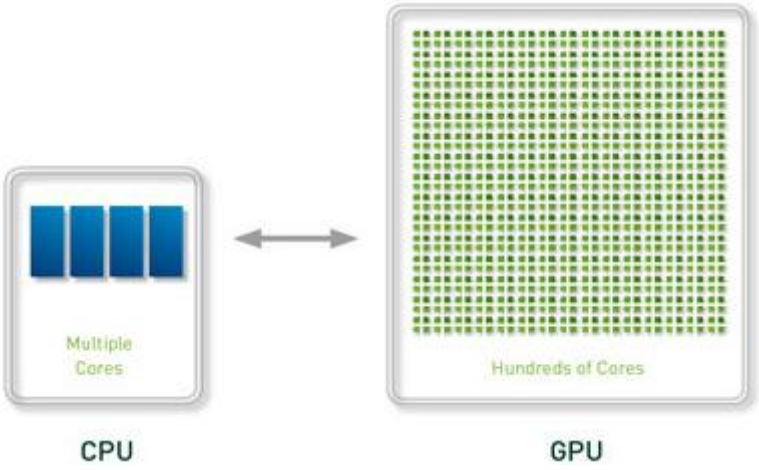


Figura.II.9. Núcleos en una CPU y en una GPU

La CPU está diseñada con memorias caché de gran tamaño para reducir la latencia hacia las localidades de memoria de acceso frecuente en la memoria principal. Estas memorias caché consumen una gran parte en los diseños de la CPU, espacio que podrá utilizarse para las unidades de procesamiento adicionales de la aritmética. La GPU adopta un enfoque muy diferente y emplea el hardware multi-hilo de ejecución para ocultar la latencia de acceso a la memoria en el cambio de un subproceso ejecutable cuando el hilo de ejecución se encuentra con una activa dependencia de una operación a la espera de memoria global. Con un número suficiente de hilos multiplexados en cada una de las unidades de procesamiento, la latencia es realmente oculta, y el dispositivo logra un alto rendimiento sin la necesidad de una memoria caché. GPU utiliza el área que habrían sido consumidos por la memoria caché en la CPU para las unidades de la aritmética adicionales, dando un rendimiento de la arquitectura orientada a aritmética máxima de alto rendimiento.

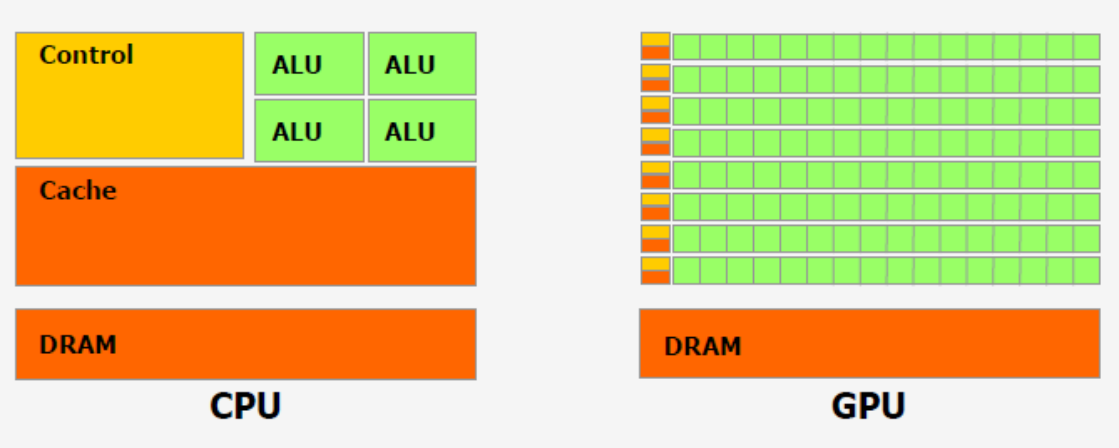


Figura.II.10. Las GPU's dedican más transistores para el procesamiento de datos

En la siguiente tabla se resume las diferencias entre un CPU y una GPU desde el punto de vista de diseño:

	CPU	GPU
Optimizado para	Código secuencial	Videojuegos
Núcleos	Pocos, pero muy complejos	Cientos de núcleos, muy sencillos
Gran capacidad para Paralelismo	De tareas	De datos
Frecuencia de Reloj	Altas frecuencia de reloj, sobre el 1.5 GHz	Bajas frecuencia de reloj, por debajo del 1 GHz

Tabla.II.III. Diferencias desde el punto de vista de diseño

Elaborado por: Autor

El rendimiento de una GPU es mucho mayor que en la CPU. Una computadora personal o PC con una unidad central Core i7 a 2.0 GHz puede teóricamente procesar alrededor de 54 billones de puntos flotantes o float-point por segundos, sin embargo un programa ejecutado en una GPU de modelo NVIDIA GeForce GTX 460M procesa

alrededor de 297 billones de puntos flotantes por segundo, lo que constituye en una gran diferencia. En cuanto a la memoria de ancho de banda disponible, en la GPU puede alcanzar un tope de 192 GB/s comparando con un procesador actual que ronda entre los 10 GB/s. El balanceo de carga es una característica fundamental que hay que tener en cuenta, la CPU limita el rendimiento de la aplicación dejando la GPU con ciclos inactivos, limitando la descarga del trabajo a la GPU para lograr un aumento de velocidad en la aplicación. Por estas razones queda claro que la GPU es mucho mejor para realizar cálculos complejos que en una CPU. Y esta diferencia se hace más notable con el pasar de los años; en la figura 11 se puede ver la evolución de este salto que hay entre el rendimiento de la CPU y el de la GPU en los últimos años; en la figura 12 se muestra ésta misma evolución, pero desde el punto de vista del ancho de banda de la memoria.

Una de las, pocas, desventajas de usar una GPU es que no se pueden implementar todos los algoritmos en ella, ni puede ser vista como una plataforma de elección para la implementación de cualquier algoritmo, debido a su obligatoria implementación en paralelo.

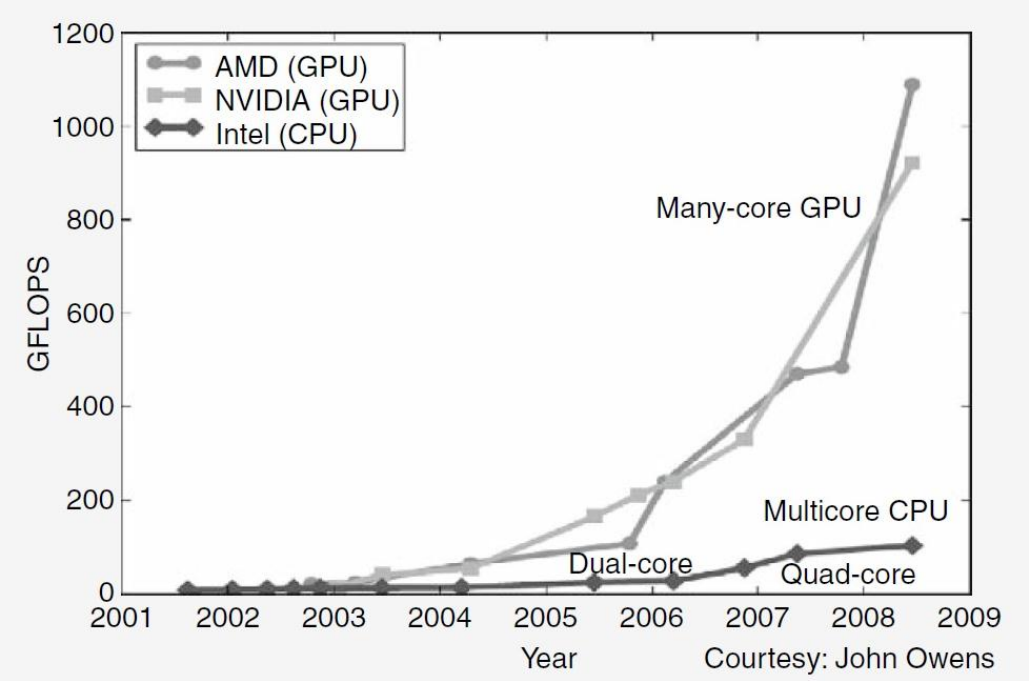


Figura.II.11. Evolución de procesadores(GFLOPS)

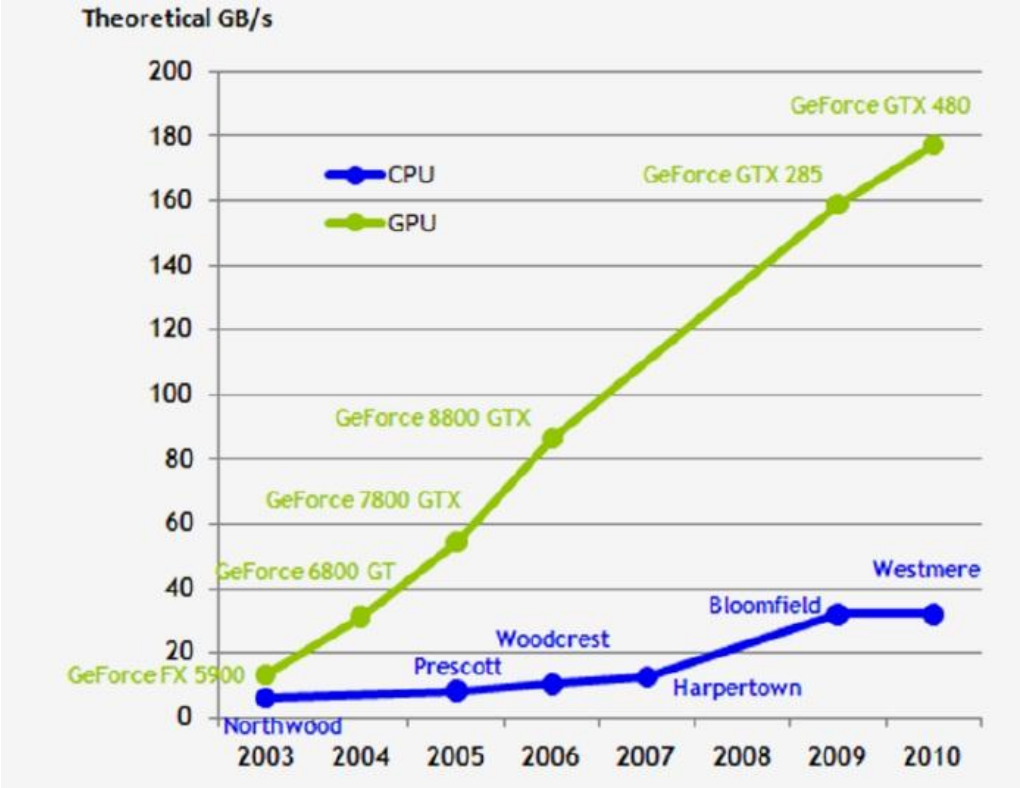


Figura.II.12. Evolución del ancho de banda de la memoria en los procesadores(GPU y CPU)

2.4.7 Modelo de Programación

La principal característica de la GPU es que no es un procesador en serie, también conocido como arquitectura de Von Neumann (Von Neumann 1945), pero si pertenecen al grupo de los procesadores de flujos o stream-processor, que se diferencian ligeramente por la ejecución de una función en un conjunto de registros de entrada produciendo un conjunto de archivos en paralelo. Estos procesadores se refieren a esta función como un módulo central de un sistema o kernel y al conjunto de registro como un flujo.

El modelo básico de programación de la GPUs es a través de los hilos de ejecución o threads en paralelo. Estos hilos de ejecución no son más que ligeros procesos que son fáciles de crear y sincronizar a diferencia de los procesos de la CPU[5].

Debido a las diferencias fundamentales entre las arquitecturas de la GPU y la CPU, no cualquier problema se puede beneficiar de una implementación en la GPU. En concreto, el acceso a memoria plantea las mayores dificultades. Las CPU están diseñadas para el acceso aleatorio a memoria. Esto favorece la creación de estructuras de datos complejas, con punteros a posiciones arbitrarias en memoria. En cambio, en una GPU, el acceso a memoria está mucho más restringido. Por ejemplo, en un procesador de vértices (la parte de una GPU diseñada para transformar vértice en aplicaciones 3D), se favorece el modelo scatter, en el que el programa lee en una posición predeterminada de la memoria, pero escribe en una o varias posiciones arbitrarias. En cambio, un procesador de píxeles, o fragmentos, favorece el modelo gather, pudiendo el programa leer de varias posiciones arbitrarias, pero escribir en sólo una posición predeterminada.

La tarea del diseñador de algoritmos GPGPU consiste principalmente en adaptar los accesos a memoria y las estructuras de datos a las características de la GPU. Generalmente, la forma de almacenar datos es en un buffer 2D, en lugar de lo que normalmente sería una textura. El acceso a esas estructuras de datos es el equivalente a una lectura o escritura de una posición en la textura. Puesto que generalmente no se puede leer y escribir en la misma textura, si esta operación es imprescindible para el desarrollo del algoritmo, éste se debe dividir en varias pasadas.

Pese a que cualquier algoritmo que es implementable en una CPU lo es también en una GPU , esas implementaciones no serán igual de eficientes en las dos arquitecturas. En concreto, los algoritmos con un alto grado de paralelismo, sin necesidad de estructuras de datos complejas, y con una alta intensidad aritmética, son los que mayores beneficios obtienen de su implementación en la GPU.

2.5 GPGPU. Computación de Propósito General sobre Unidades de Procesamiento Gráfico

2.5.1 Introducción

GPGPU o General-Purpose Computing on Graphics Processing Units es un concepto reciente dentro de informática que trata de estudiar y aprovechar las capacidades de cómputo de una GPU.

Una GPU es un procesador diseñado para los cómputos implicados en la generación de gráficos 3D interactivos. Algunas de sus características (bajo precio en relación a su potencia de cálculo, gran paralelismo, optimización para cálculos en coma flotante), se consideran atractivas para su uso en aplicaciones fuera de los gráficos por

computadora, especialmente en el ámbito científico y de simulación. Así, se han desarrollado técnicas para la implementación de simulaciones de fluidos, bases de datos, algoritmos de clustering, etc[8].

Teóricamente, las GPUs son capaces de realizar cualquier cálculo que puede transformar el modelo de paralelismo y que permitan la arquitectura específica de la GPU. Este modelo ha sido explotado por múltiples áreas de investigación.

2.5.2 Historia de la Computación de Propósito General sobre GPU

Tradicionalmente, el desarrollo de software GPGPU se había hecho bien en ensamblador, o bien en alguno de los lenguajes específicos para aplicaciones gráficas usando la GPU, como GLSL, Cg o HLSL. Pero recientemente han surgido herramientas para facilitar el desarrollo de aplicaciones GPGPU, al abstraer muchos de los detalles relacionados con los gráficos, y presentar una interfaz de más alto nivel. Existen herramientas ofertadas por las principales empresas de procesadores gráficos como son nVidia y ATI, así como propuestas libres, como es el caso de OpenCL.

2.5.3 Aplicaciones GPGPU

Cada vez son más campos en los que se aplica la GPGPU, a continuación se lista algunos ejemplos de aplicaciones de propósito general:

- *Grid computing* (red virtual de computación)
- Simulaciones físicas
- Procesado de señal de audio, uso de las GPU's para técnicas de DSP (*digital signal processing*)
- Procesado del habla

- Aplicaciones científicas
- Predicción del tiempo
- Investigación climática (como por ejemplo investigación del calentamiento global)
- Modelos moleculares
- Mecánica cuántica
- Bioinformática
- Aplicaciones de predicción financiera y de mercados
- Imagen médica
- Visión artificial
- Redes neuronales
- Bases de Datos
- Minería de Datos
- Criptografía y criptoanálisis
- Computación Gráfica

2.5.4 API para GPGPU

Existen algunas APIs diseñadas específicamente para hacer uso de las GPUs en aplicaciones que no tienen que ver con gráficos, y que facilitan en gran medida el desarrollo de aplicaciones que aprovechen el poder de cómputo de estos procesadores sin la necesidades de utilizar una API gráfica ó un lenguaje de sombreado(shading). A

continuación se mencionan las más populares y que mayor progreso han tenido en los últimos años:

2.5.4.1 CUDA

El SDK/API de NVIDIA es probablemente el que a día de hoy está más avanzado y el que también ha sido más aceptado por algunos desarrolladores.

Figura.II.13. CUDA, la API GPGPU de nVidia

Para poder acceder a esta API se necesita de una GPU que lo soporte (en la actualidad, la gran mayoría lo hacen) y los controladores NVIDIA GeForce que van añadiendo más y mejor soporte para dicha tecnología. CUDA es una arquitectura de cálculo paralelo de NVIDIA que aprovecha la gran potencia de la GPU (unidad de procesamiento gráfico) para proporcionar un incremento extraordinario del rendimiento de sistemas.

Con una base instalada de más de 128 millones de GPUs aptas para CUDA, miles de desarrolladores, científicos e investigadores están encontrando innumerables aplicaciones prácticas para esta tecnología en campos como el procesamiento de vídeo, la astrofísica, la biología y la química computacional, la simulación de mecánica de fluidos, la interferencia electromagnética, la reconstrucción de imágenes, el análisis sísmico o el trazado de rayos entre otras. [11]

2.5.4.2 ATI Stream

Es la alternativa de AMD destinada a lograr lo mismo que NVIDIA, pero gracias a las GPUs de AMD. ATI Stream es una tecnología que constituye en un set de hardware y software que permite a los procesadores gráficos de AMD a trabajar en conjunto con la CPU para acelerar las aplicaciones de propósito general [19]

Figura.II.14. ATI Stream

2.5.4.3 OpenCL

Esta API trata de ofrecer un estándar unificado y universal para poder desarrollar aplicaciones con soporte GPGPU. Las dos principales empresas fabricantes soportan OpenCL , aunque de momento ambas apoyan públicamente su propio estándar. Es un estándar apoyado por una gran cantidad de compañías como Sony, Apple, Nokia, Intel, Toshiba, etc; y su principal ventaja es la portabilidad, ya que incluso está disponible en teléfonos celulares.

Figura.II.15. Logotipo de OpenCL

2.5.4.4 Direct Compute

Es la propuesta de Microsoft orientada a las aplicaciones GPGPU. DirectCompute se incluye en las actuales GPU's compatibles con DX10 y las futuras GPU's DX11. La nueva especificación permite a los desarrolladores aprovechar la extraordinaria capacidad de procesamiento paralelo de las GPU's NVIDIA para crear aplicaciones de cálculo altamente eficientes tanto para el mercado profesional como el de consumo.

Una ventaja de Direct Compute es que funciona independiente de la plataforma hardware en donde se ejecuta, es decir funciona tanto sobre tarjetas gráficas NVIDIA como en tarjetas AMD.

2.6 OpenCL

Open Computing Language es nuevo un estándar multiplataforma para computación heterogénea, es decir que tiene soporte para poder ser ejecutada sobre cualquier GPU e incluso sobre los CPUs. Su uso permite a los desarrolladores aprovechar la extraordinaria capacidad de procesamiento paralelo de las GPUs para crear aplicaciones de cálculo altamente eficientes.

OpenCL es una API compatible con la especificación C89, por lo cual tiene como lenguaje nativo a C, pero también se han hecho muchos bindings o wrappers [4]a otros lenguajes como C++, C#, java, python, etc.

[4] wrapper es un componente que obtiene su funcionalidad de una librería externa ya sea una dll o un ocx. De esta forma el componente es solo una interface entre dicha librería y el sistema a desarrollarse.

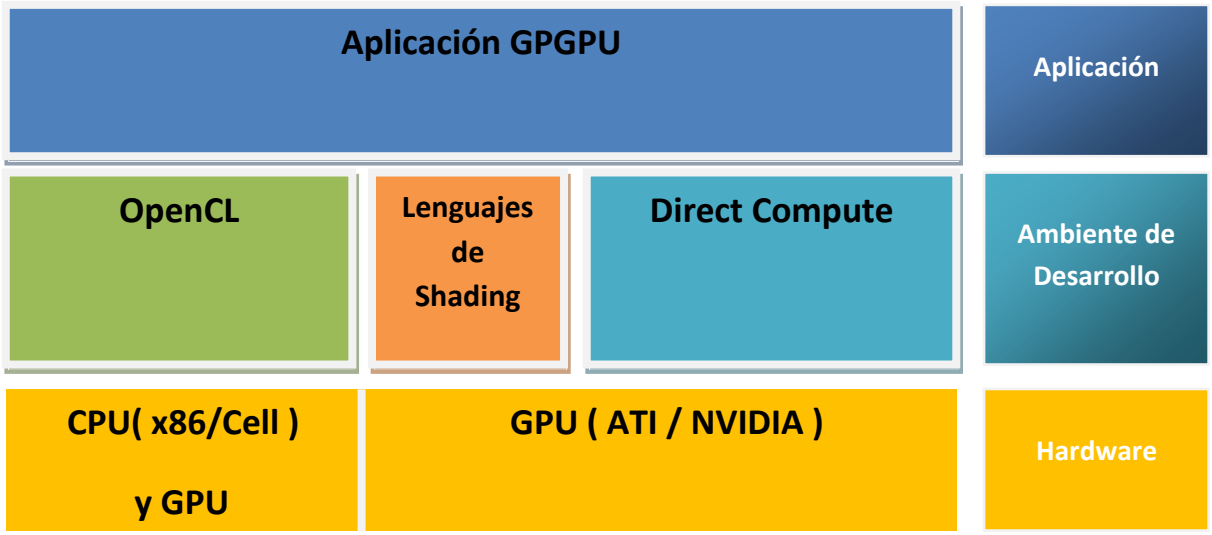


Figura.II.16. Arquitectura de Aplicaciones GPGPU

OpenCL es un estándar propuesto inicialmente por Apple y desarrollado por Khronos Group. La primera especificación de OpenCL fue lanzada en Diciembre de 2008, después de lo cual, AMD y Nvidia, no tardaron mucho en dar soporte a éste estándar en sus respectivas plataformas.

2.6.1 Características

OpenCL es una API que ofrece muchas posibilidades:

- **Aceleración** en procesos paralelos.
- **Interoperabilidad** con API gráficas como OpenGL y DirectX.
- **Portabilidad** a través de distintos fabricantes, es decir es independiente de la plataforma sobre la cual se ejecute, ya que puede ejecutarse sobre CPU's, GPU's, procesadores basados en Cell y dispositivos DSP.

2.6.2 Requisitos para ejecutar aplicaciones OpenCL

- Un dispositivo que soporte OpenCL
- Los respectivos drivers.

2.7 Fractales

La geometría euclidiana trata con rectas, círculos, polígonos, poliedros, entre otros, lo cual nos permite estudiar muchas formas de la naturaleza y las construidas por los humanos. Pero con la sola ayuda de la geometría euclidiana, no se pueden explicar algunas formas de la naturaleza tales como: líneas costeras, ramificaciones arbóreas o bronquiales, rocas, montañas, nubes, sistema neuronal, brócolis, coliflor, corales, sistemas montañosos, cortezas de árboles, y ciertos objetos matemáticos.

La palabra fractal procede del adjetivo latino fractus que significa “interrumpido”, “irregular” o “fraccionario”. Y, en cierta forma, algunos objetos de la naturaleza son fragmentados, irregulares, rugosos. Surge la geometría fractal como la geometría de la naturaleza.

Un fractal es una figura geométrica en la que un motivo (patrón) se repite pero siempre disminuyendo su escala en el mismo porcentaje.

2.7.1 Historia

Los fractales fueron concebidos aproximadamente en 1980 por el francés Henri Poincaré. Sus ideas fueron extendidas más tarde fundamentalmente por dos matemáticos también franceses, Gastón Julia y Pierre Fatou, hacia 1918. Se trabajó mucho en este campo durante varios años, pero el estudio quedó congelado en los años 20.

El estudio fue renovado a partir de 1974 en IBM y fue fuertemente impulsado por el desarrollo de la computadora digital. El Dr. Mandelbrot, de la Universidad de Yale, con sus experimentos de computadora, es considerado como el padre de la geometría fractal. En honor a él, uno de los conjuntos que él investigó fue nombrado en su nombre. Otros matemáticos, como Douady, Hubbard y Sullivan trabajaron también en esta área explorando más las matemáticas que sus aplicaciones.

Maldelbrot tratando de hacer un análisis del ruido y perturbaciones eléctricas, encontró, mientras realizaba dichos estudios, un patrón en su comportamiento y por lo tanto comenzó a descifrar una estructura escondida. Algo así como jerarquías de fluctuaciones en todas las escalas.

Lo que sí es cierto es que esas fluctuaciones no podían ser descritas por la matemática estadística que existía. Mientras seguía adelante con sus tareas empezó a imaginar en que otros sistemas podrían encontrar patrones similares que no puedan ser descritos con exactitud por la matemática existente y que se comportaran de igual manera. Su visión lo llevó a hacerse una pregunta que para la mayoría puede resultar obvia y hasta para muchos otros ser trivial, o en el mejor de los casos sin sentido. Su famosa pregunta fue: ¿Cuánto mide realmente la costa de Inglaterra? Pues bien, cualquiera que tome un libro de geografía o un mapa va a poder contestar esto sin ningún tipo de problema. Suponiendo que el dato que se encuentra es de 2.000 kilómetros. Ahora

bien, esos 2.000 KM., ¿de dónde provienen? ¿Cómo se midieron? Para contestar esto voy a exponer tres situaciones diferentes, con distintos puntos de vista:

1) Si medimos las costas de Inglaterra desde un satélite, se va a ver que sus bordes son suaves, armónicos, con líneas casi rectas y ángulos prácticamente redondeados.

2) Midiendo la misma distancia, pero desde un avión que vuela mucho más bajo que el satélite. ¿Qué pasa en este caso? En esta ocasión las cosas se ven con más detalle por estar más próximos, por lo tanto se ve que los bordes no eran en realidad tan suaves como se había observado anteriormente, sino que se nota muchas más rugosidades.

3) Por último un tercer punto de partida, algo extremista, pero vale. Esta vez situados sobre la misma costa de Inglaterra con una regla como la que se usa en la escuela, y midiendo roca por roca, rugosidad por rugosidad, detalle por detalle.

¿Cuál será el resultado de las distintas mediciones? ¿Siempre habrá arrojado el mismo resultado? ¿Si fue variando, cuál habrá sido el de mayor extensión?

La realidad y la geometría tradicional nos muestra que una figura con bordes rectos o redondeados tiene una longitud menor que otra equivalente pero llena de rugosidades. Volviendo al caso de la costa de Inglaterra y la pregunta de Mandelbrot. Si se acaba de decir que una longitud sin rigurosidades es menos extensa que una totalmente irregular, entonces se puede asegurar que los resultados de las 3 mediciones serán en todos los casos diferentes, y el de mayor extensión será el tercer caso, ya que es en el cual aparece con más detalles. En realidad el resultado de este último caso se acercaría a infinito en el marco teórico.

¿De qué dependerán las mediciones que se hagan? Justamente de la escala que se utilice para medirlas, y no es para nada una casualidad que estas deducciones se desprendan de los mismos patrones que encontró Mandelbrot en sus estudios sobre flujo electrónico. Esas escalas como Mandelbrot reconoció poseían un patrón, y ese patrón las relacionaba diciendo que si bien no eran iguales a diferentes escalas, si lo eran de manera estadísticamente similar, y ésta es una de las características principales de los fractales.

Euclídea	Fractal
Tradicional (más de 2000 años)	Moderna (aprox. 10 años)
Dimensión entera	Dimensión fractal
Trata objetos hechos por el hombre	Apropiada para formas naturales
Descripta por fórmulas	Algoritmo recursivo (iteración)

Tabla.II.IV. Diferencias fundamentales entre la Geometría Euclídea y la Fractal

(JORGE ANDRES DUSSAN PASCUAS)

2.7.2 Aplicaciones

Música

Realmente se habla de música fractal, una aplicabilidad reciente, pero más que aplicación de lo que se trata es de encontrar en los fractales notas musicales. ¿Cómo? Tomando en cuenta que un fractal es la iteración de una expresión algebraica simple. Si a cada punto asociamos una nota musical, obtendremos lo que conocemos como Música Fractal.

Se puede decir que existe un patrón de puntos congruentes entre sí que generan el fractal, a los que, por estar determinados, podemos asociarles notas musicales

determinadas. Pero si un fractal es "infinito" en puntos, ¿entonces la melodía es infinita? Pues sí.

En el Arte

Por increíble que pueda parecer o por el contrario, obvio, los Fractales tienen también sus aplicaciones dentro del Arte. Se podría decir que es una de las aplicaciones más interesantes, claro que no es tanto así como una aplicación, es la belleza del gráfico de estos cuerpos matemáticos. La mayoría de los fractales, una vez representados y asociados con algunos colores, arrojan formas muy hermosas y de nuevos colores. Como dice Mandelbrot, la plástica se explica por sí misma.

Esta es una de las aplicaciones más "eterea" Eso, porque no se utilizan los fractales en su forma más "matemática". Tampoco se pintan los fractales. Sino que se utilizan los conceptos de orden y caos que van ligados con éstos para la localización de los elementos en la obra.

En las Ciencias de la Computación

Dentro de las aplicaciones que se dan a los Fractales, las que se presentan en la Computación son verdaderamente impresionantes, creativas y sobre todo muy importante, permiten el desarrollo de muchas cosas distintas (técnicas) y se considera pionera en el campo de sus aplicaciones.

La aplicación más común, es la de la Transformación Fractal, proceso que se utiliza en el tratamiento de imágenes para reducir su espacio "físico" (o peso en bytes) mediante esta técnica; se utilizan en programas como winzip y winrar muy comunes en los ordenadores.

La primera vez (o mejor dicho, la primera vez "conocida", ya que se hacía desde antes) que el público pudo observar esta forma de utilización del proceso fractal fue en las imágenes incluidas en la Enciclopedia Multimedia Encarta. Aunque ahora esto se aprecia de manera muy común y no sólo en imágenes estáticas, sino que en complejas animaciones de videojuegos y también en algunas cintas de cine muy populares, especialmente de ciencia ficción.

En la Geografía

La primera de las aplicaciones que hoy en día se dan a los fractales es en el cálculo más cercano o acertado de distancias. Esto, para determinar las verdaderas distancias que separan costas de continentes y cosas por el estilo. Además, se ha usado mucho lo que son las curvas de Koch para efectos similares al que mencionamos. Esto presenta un interesante problema "¿qué importancia tiene calcular mejor una distancia?". Pues en asuntos como la exploración espacial, por ejemplo, errar un "pequeño" cálculo, a escalas diferentes, puede significar millones y millones de kilómetros de "error" lo que puede incurrir en serios problemas.

Quizá la forma más útil de aplicar los fractales en la geografía es la elaboración de mapas en tres dimensiones muy detallados (y elaborados, por lo demás) que permiten entregar una imagen 99.9% real en comparación de la forma de nuestro planeta y su geomorfología.

2.8 Aplicación de los Fractales en la Computación Gráfica. Paisajes Fractales

Los paisajes fractales son una representación de un paisaje real o imaginado, producido mediante fractales. Originalmente se conoció como paisaje fractal a una forma bidimensional de la forma de una línea de costa fractal, la que puede ser

considerada una generalización estocástica de la curva de Koch. Su dimensión fractal es una fracción entre 1 y 2.

Para construir este tipo de paisajes, básicamente se subdivide un cuadrado en cuatro cuadrados iguales y luego se desplaza aleatoriamente su punto central compartido. El proceso se repite recursivamente en cada cuadrado hasta que se alcanza el nivel de detalle deseado.

Dado que hay muchos procedimientos fractales (como el "ruido de Perlin") que pueden ser utilizados para crear datos de terreno, la expresión paisaje fractal se utiliza actualmente en forma genérica.

Aunque los paisajes fractales parezcan naturales a primera vista, la exposición repetida puede defraudar a quienes esperen ver el efecto de la erosión en las montañas. La crítica principal es que los procesos fractales simples no reproducen (y quizás no puedan hacerlo) las funciones geológicas y climáticas reales. [13]

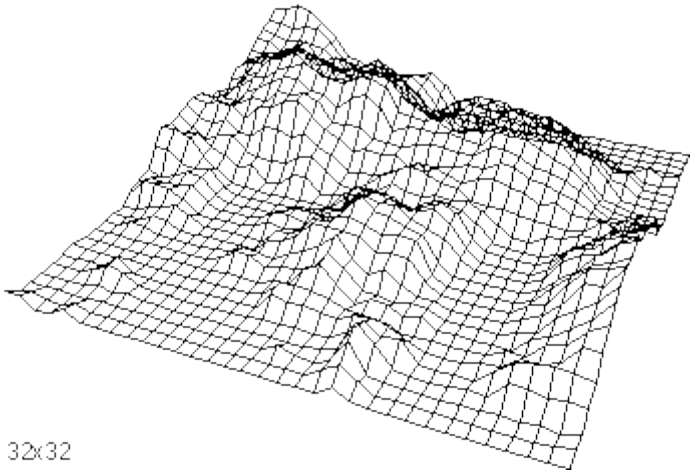


Figura.II.17. Paisaje Fractal

CAPÍTULO III

ANÁLISIS DE LA PROGRAMACIÓN CONCURRENTE EN LA CPU Y GPU

3.1 Introducción

Para determinar la tecnología más adecuada para el desarrollo de una aplicación gráfica se debe realizar un análisis minucioso en función de algunos criterios y factores que se deben tomar en cuenta al momento de la ejecución de la aplicación, en este caso específico, una aplicación gráfica para la generación de un paisaje fractal.

Los criterios tomados en cuenta en el presente análisis, han sido seleccionados en función de los principales requerimientos que una aplicación gráfica, como Fractal Build (aplicación para la generación de un paisaje fractal), debería tener.

Primeramente se hará un enfoque de cómo funciona la programación concurrente en cada una de las plataformas a analizar, después se determinará los parámetros con los cuales se realizará el análisis. Y por último con la ayuda de un prototipo se realizarán las mediciones respectivas para poder determinar la plataforma más óptima para implementar la aplicación Fractal Build.

3.2 Programación Concurrente en la CPU y GPU

En el capítulo 2 se ha hecho una introducción a la programación concurrente y paralela, además se ha realizado un estudio tanto de la CPU como de la GPU para dar un punto de vista general de su arquitectura hardware y dar un enfoque de cómo se programa en cada una de éstas plataformas. Con todo ello se ha finalizado con una comparación entre la CPU y GPU, mencionando las principales diferencias al momento de programar en ellas debido, principalmente, a su arquitectura.

Para poder comparar objetivamente la ejecución en cada una de las tecnologías, se ha seleccionado una API de programación paralela para arquitecturas multinúcleo, lo que permitirá ejecutar el mismo programa en las dos plataformas con pocos cambios, lo que permitirá realizar las comparaciones en las mismas condiciones, desde el punto de vista de programación, sobre las dos tecnologías.

3.3 OpenCL como API para la programación en Arquitecturas Multi-núcleo

3.3.1 Arquitectura de OpenCL

OpenCL es un estándar abierto para la programación de sistemas heterogéneos, es decir sobre CPU o GPU, por lo tanto es mucho más que un lenguaje de programación, consiste en un framework para la programación paralela que incluye además de una API, librerías y un sistema de ejecución para permitir el desarrollo. Mediante el uso de OpenCL se puede construir aplicaciones de propósito general que puedan ser ejecutadas sobre GPUs de distintos fabricantes sin la necesidad de que se tenga que mapear los algoritmos a una API gráfica como OpenGL o DirectX, así como también sobre los procesadores tradicionales y también sobre procesadores como Cell[5].

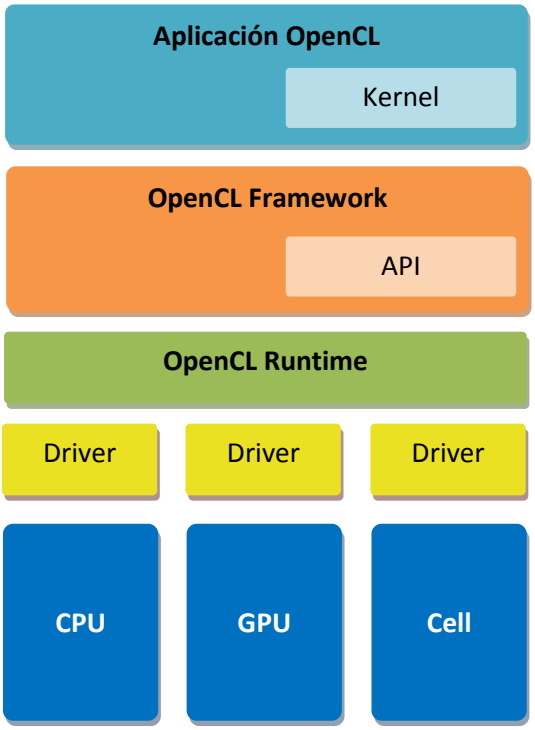


Figura.III.18. Arquitectura OpenCL

[5] Cell es una arquitectura de microprocesador desarrollada conjuntamente por Sony Computer Entertainment, Toshiba, e IBM. Cell emplea una combinación de la arquitectura de núcleo PowerPC, de propósito general y medianas prestaciones, con elementos coprocesadores3 en cascada, los cuales aceleran notablemente aplicaciones de procesado de vectores y multimedia, así como otras formas de computación dedicada

La figura anterior muestra todos elementos que intervienen en la ejecución de un aplicación OpenCL, están ordenados en capas ordenadas desde el más alto nivel al más bajo. A continuación se describe cada una de ellas:

- **Aplicación OpenCL.** es un aplicación que hace uso de funcionalidades específicas de OpenCL para llevar a cabo tareas computacionales en las unidades de cómputo disponibles(CPU/GPU). Las tareas computacionales se representan mediante funciones llamadas kernels que son pequeños subprogramas que van a ser ejecutados sobre una unidad de cómputo.
- **Framework OpenCL.** Es el encargado de ofrecer los recursos necesarios para el desarrollo de aplicaciones OpenCL. Básicamente es un API en lenguaje C que contiene un conjunto de funciones que permite controlarla ejecución de tareas sobre los dispositivos o unidades de cómputo. Mediante llamadas a la API se consigue controlar todo el proceso: inicialización de dispositivos, preparación del entorno de ejecución, ejecución de kernels y la recuperación de los resultados obtenidos tras la ejecución. Dentro del framework también está un compilador OpenCL que se encarga de procesar los kernels escritos en OpenCL C para generar ejecutables binarios que puedan ser ejecutados en las unidades de cómputo.
- **Runtime OpenCL**. Lleva a cabo la ejecución de todas las tareas enviadas mediante llamadas a la API de OpenCL.
- **Drivers y Hardware.** El último nivel para la ejecución de aplicaciones OpenCL. El hardware, que es donde se ejecutan los kernels, que son accesibles mediante el driver proporcionado por el fabricante.

3.3.2 Jerarquía de Modelos OpenCL

El objetivo de OpenCL es ofrecer a los programadores una API para elaborar código eficiente y portable, de tal manera que pueda ser ejecuta en multitud de plataformas.

Para describir el núcleo de OpenCL se usa un jerarquía de modelos:

- Modelo de la Plataforma
- Modelo de Memoria
- Modelo de Ejecución
- Modelo de Programación

3.3.3 Modelo de la Plataforma

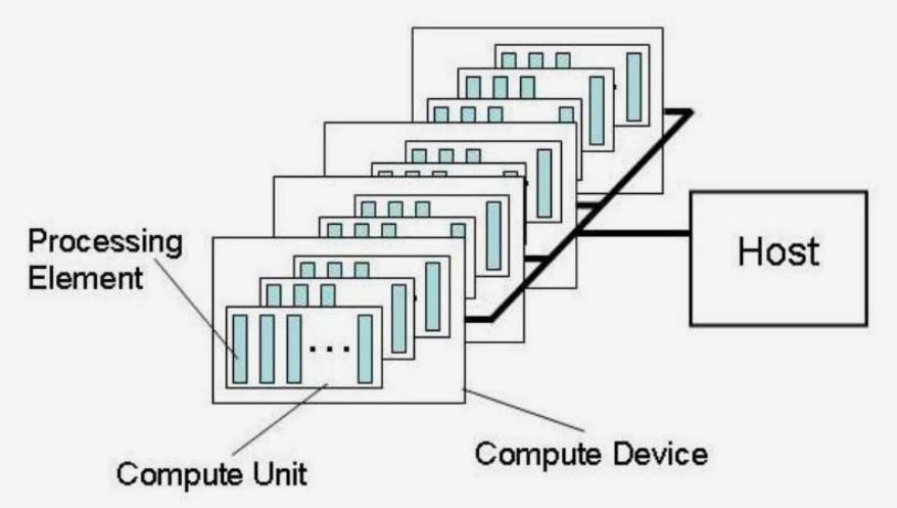


Figura.III.19. Modelo de Plataforma OpenCL

El modelo de plataforma de OpenCL es definido como un Host que se conecta a uno o más dispositivos OpenCL. La Figura 19. Modelo de Plataforma OpenCL, muestra el modelo de la plataforma el cual comprende de un host, que generalmente es el procesador principal, más algunos dispositivos de cómputo (compute devices), cada

uno de los cuales tiene múltiples unidades de cómputo (Compute Units) y a la vez, cada uno tiene varios elementos de procesamiento (processing elements)[12]

A continuación se describen 4 elementos mostrados en la figura 19:

- **Host**, es cualquier computador con un CPU corriendo un sistema operativo estándar, el cual controla la ejecución del programa.

- **Compute Device**, puede ser un GPU, DSP o un CPU multi-core (en éste caso se incluye también un procesador Cell). Un dispositivo OpenCL se compone de un conjunto de uno o varios Compute Units.

- **Compute Unit**, está compuesto de uno o más processing elements. En una GPU es cada uno de los Stream Multiprocessors(SM)

- **Processing Element**, Son los que ejecutan las instrucciones. En una GPU es cada Stream Processor(SP)

3.3.4 Modelo de Ejecución

El modelo de ejecución de OpenCL comprende 2 elementos: los programas kernels y los programas host.

El núcleo del modelo de ejecución de OpenCL es definido por cómo los kernels se ejecutan. Cuando un kernel es llamado por el host para su ejecución, un espacio de índices es definido. Una instancia del kernel se ejecuta por cada punto en éste espacio de índices. Esta instancia del kernel es llamada un **work-Item** y es identificado por su punto en el espacio de índices, el cual provee un ID global para el work-item. Cada work-item ejecuta el mismo código pero los datos sobre los que trabaja pueden variar.

Work-items son organizados dentro de **work-groups**. Los work-groups proveen una decomposición más granular del espacio de índices. Cada work-group tiene asignado un ID único. Los work-items poseen un ID local único dentro de un work-group de tal forma que un work-item puede ser identificado de manera única por su ID global ó por la combinación de su ID local y ID del work-group al que pertenece.

El espacio de índices soportado en OpenCL 1.0 es llamado NDRange. Un NDRange es un espacio de índices n-dimensional, donde n puede ser uno, dos o tres. Un NDRange es definido por un array de enteros de longitud N especificando la extensión del espacio de índices en cada dimensión.

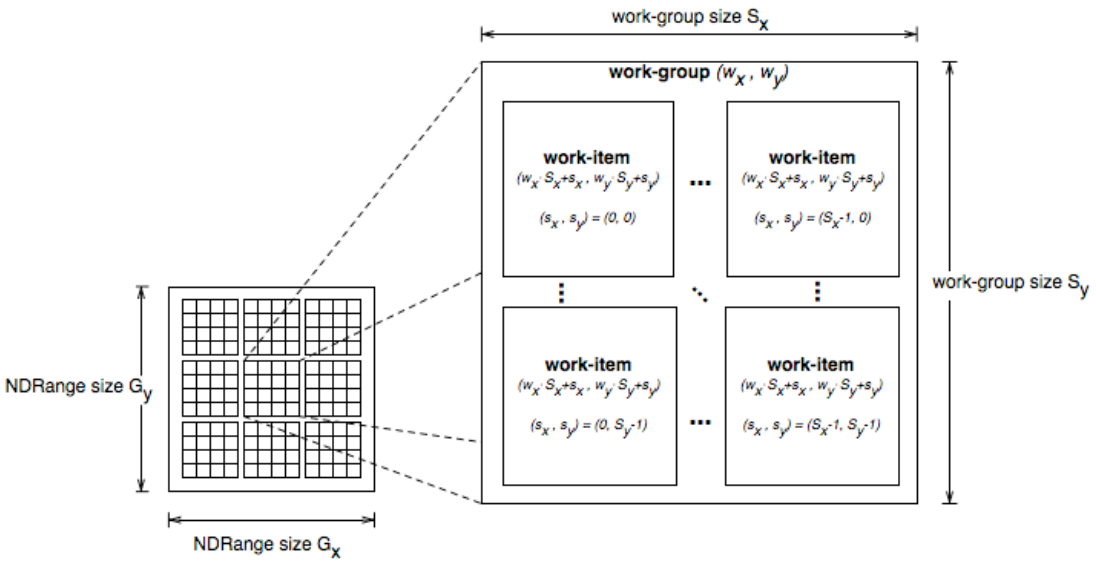


Figura.III.20. Ejemplo de un NDRange

3.3.4.1 Programas Kernels

Kernel es la unidad básica de código que se ejecuta en un dispositivo OpenCL. Es un archivo que contiene funciones C. Mediante un kernel se expresa el paralelismo de

tareas o datos, se diseña tomando en cuenta que esta función será la que se ejecute en cada núcleo del dispositivo OpenCL.

OpenCL explota la computación paralela en un dispositivo de cómputo definiendo el problema dentro de un espacio n-dimensional indizado.

3.3.4.2 Programas Host

Se ejecutan sobre el host. El programa host define el contexto sobre el cual se ejecutan los programas kernels y administra su ejecución.

Es el que envía los kernels a ser ejecutados en un dispositivo OpenCL para lo cual hace uso de las colas de comandos.

3.3.5 Modelo de Memoria

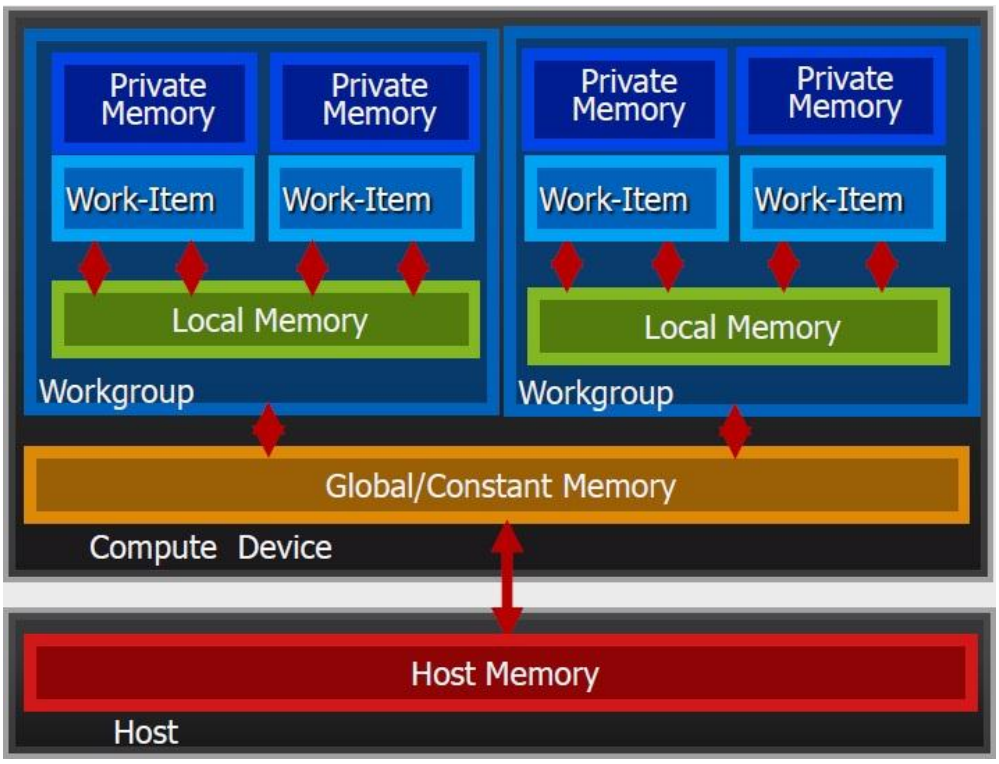


Figura.III.21. Modelo de Memoria de OpenCL

Los Work-items ejecutan un kernel teniendo acceso a 4 regiones de memoria distintos:

- **Memoria Global (Global Memory)**. Esta región de memoria permite acceso de lectura y escritura para todos los work-items en todos los work-groups. Los work-tems pueden escribir a... ó leer de... cualquier elemento de un objeto de memoria.
- **Memoria Constante (Constant Memory)**. Es una región de memoria global que se mantiene constante durante la ejecución de un kernel. El host ubica e inicializa objetos de memoria situados dentro de memoria constante.
- **Memoria Local (Local Memory).** Es una región de memoria local para un work-group. Esta región de memoria puede ser usada para asignar variables que se compartan por todos los work-items en un work-group.
- **Memoria Privada (Private Memory).** Es una región de memoria privada para un work-item. Las variables definidas en memoria privada de un work-item no son visibles para otro work-item.

3.3.6 Modelo de Programación

El modelo de ejecución de OpenCL soporta el paralelismo de datos y de tareas.

3.3.6.1 Paralelismo de Datos

El paralelismo de datos se define como una secuencia de instrucciones aplicado a múltiples elementos. En OpenCL esto consiste en un mapeo uno a uno entre un work-item y el elemento en un objeto de memoria sobre el cual un kernel puede ser ejecutado en paralelo.

3.3.6.2 Paralelismo de Tareas

En OpenCL el paralelismo de tareas se expresa de tal forma un una instancia de un kernel es ejecutado independientemente de cualquier espacio de índices. Esto es equivalente a ejecutar un kernel sobre una Unidad de Cómputo con un work-group conteniendo un solo work-item.

3.3.6.3 Sincronización

Hay dos dominios de sincronización en OpenCL:

- Work-items en un solo work-group, esto se logra usando una barrera de work-group.
- Comandos en una cola de comandos en un solo contexto. Mediante una barrera de cola de comandos ó esperando un evento.

3.3.7 Estructura de un programa en OpenCL

La estructura principal que tiene una aplicación que hace uso de OpenCL es la siguiente:

- Crear un contexto de OpenCL asociado a un dispositivo OpenCL
- Crear una cola de comandos.
- Crear un "programa" con el archivo fuente que contiene las funciones kernel.
- Compilar el "programa" creado
- Crear un kernel el cual se le asocia el "programa" creado previamente y la función a ser llamada
- Crear buffer para entrada y salida

- Pasar los parámetros a la función kernel a ser llamada
- Encolar el comando del kernel para ser ejecutado
- Copiar los resultados de la salida en un buffer

3.4 Determinación de los parámetros para comparar la programación concurrente en la CPU y GPU

Una aplicación gráfica es muy "sensible" en lo que tiene que ver con el rendimiento, incluso más sensibles que cualquier otro tipo de aplicación. Es así que una aplicación de este tipo puede ser muy interactiva y manejable al tener un rango de dibujado entre 24 y los 30 fps, puede ser difícil de manejar entre los 15 y 24 fps, y nada usable a a menos de 15 fps[6]. Por lo cual el rendimiento es un factor importantísimo y a tener muy en cuenta al momento de desarrollar una aplicación gráfica en tiempo real. Esta es precisamente la razón por la cual este requerimiento toma suma relevancia ya que influye directamente en la funcionalidad de la aplicación, y de allí su importancia.

Los parámetros seleccionados van a medir el comportamiento en el tiempo y el uso de recursos de la aplicación, todo esto mediante el uso de un prototipo a desarrollar.

Comportamiento en el tiempo:

- **Tiempo de ejecución**

  Es el índice de prestaciones más intuitivo y más fácil de medir. Es un parámetro absoluto ya que permite medir la rapidez de ejecución del algoritmo en cada una de las tecnologías.

  Consiste en el tiempo que tardará el algoritmo ó código en ser ejecutado.

- **FPS(Fotogramas por segundo)**

En una aplicación gráfica lo más importante es la velocidad a la que se dibujan los objetos en pantalla, éste factor se lo mide a través de los FPS ó fotogramas por segundo. Y en una aplicación que requiere gran poder de cómputo, como es el caso de fractales, este factor se ve seriamente afectado ya que el dibujado de la escena se realiza sólo después de haber procesado toda la información.

Uso de Recursos:

- **Uso de CPU**

El Procesador no sólo es el encargado de procesar las tareas de la aplicación gráfica, sino también del resto de procesos del sistema operativo.

Éste parámetro se refiere al tiempo que utiliza un proceso o programa para realizar una tarea determinada.

- **Uso de memoria**

La memoria es un recurso importante de la computadora ya que determina el tamaño y número de programas que pueden ejecutarse al mismo tiempo, así como también la cantidad de datos que pueden ser procesados, ya que las instrucciones son cargadas en memoria para después ser ejecutadas; aunque hoy en día no se está muy limitado en cuanto a cantidad de memoria RAM respecta, es algo a tomar muy en cuenta cuando se trate de aplicaciones más complejas, de ahí su importancia en la investigación.

Criterios	Parámetros		Escenario	
Comportamiento en el tiempo	Tiempo	FPS	CPU	GPU
Uso de recursos	Uso de Procesador	Uso de memoria		

Tabla.III.V. Criterios y Parámetros de Análisis

Elaborado por: Autor

3.5 Definición de las escalas de Evaluación

Dada que la principal necesidad es el rendimiento es necesario dar un peso, de acuerdo a su importancia dentro de la aplicación gráfica, a cada uno de los parámetros a medir. El resultado del valor medido posteriormente se asignará en la escala.

Para el análisis de resultados que se obtendrán se ha tomado como base comparaciones cuali-cuantitativas definidas bajo criterio propio del autor, en base a la importancia de los parámetros a evaluar en una aplicación gráfica. La siguiente tabla detalla los valores:

Parámetros	Valor porcentual
Tiempo de ejecución	30%
Fotogramas por segundo	50%
Uso de Memoria	10%
Uso de Procesador	10%
Total	**100%**

Tabla.III.VI. Parámetros de medición y su respectivo valor porcentual

Elaborado por: Autor

El peso que se le da a cada parámetro para la puntuación global está tomado de acuerdo a la importancia en una aplicación de este tipo, es decir en un aplicación gráfica es más importante la tasa de frames por segundo a la que se está presentando la escena, ya que de la fluidez de esta depende cuán bien se visualice y se interactúa con la aplicación.

El segundo parámetro con mayor peso es el tiempo de ejecución, ya que este influye directamente en la tasa de frames por segundo, y es un factor a tomar muy en cuenta en la aplicación gráfica a desarrollar.

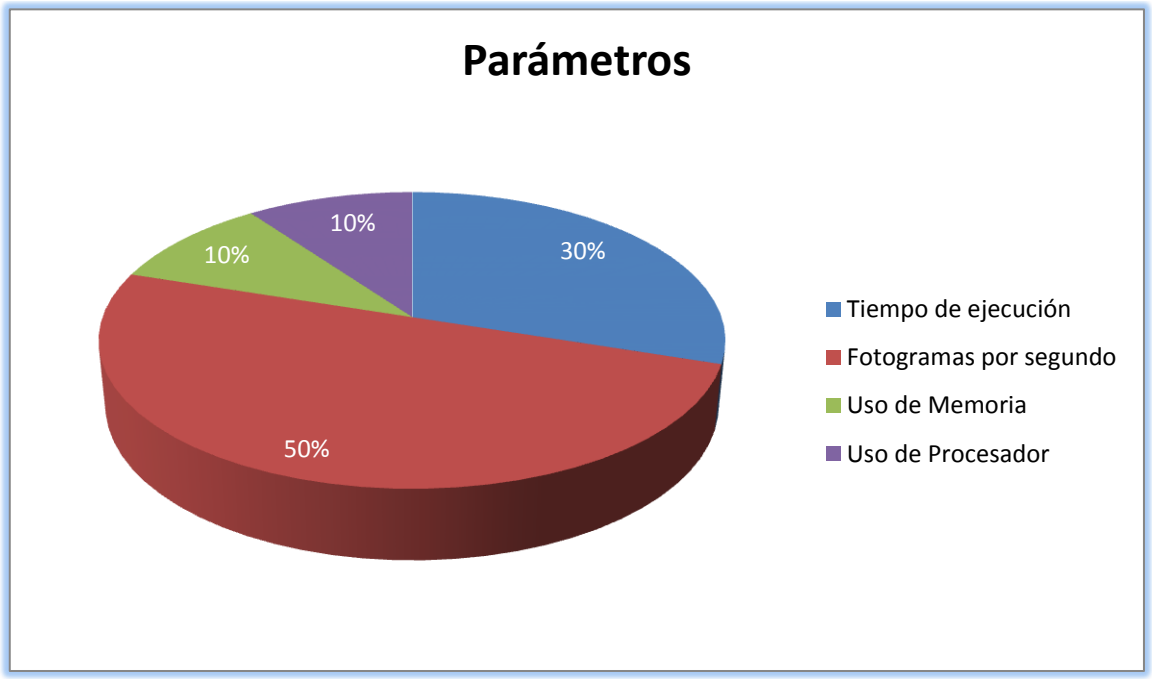


Figura.III.22. Escala de los parámetros de Medición

Una vez que se tiene la escala definida y con los parámetros evaluados en cada una de las tecnologías se obtendrá como resultado la tecnología más adecuado en la que se implementará la aplicación Fractal Build.

Para que la medición sea más precisa se detallan escalas para cada uno de los parámetros, donde los valores que se obtengan serán sumados y mapeados en la escala general anteriormente definida.

- **Tiempo de ejecución**

Mediante éste parámetro se determinará el tiempo de respuesta, es decir el tiempo que toma en completarse la ejecución del algoritmo. Para obtener el tiempo de ejecución se hará uso de librerías propias del lenguaje C++, específicamente **time**.

La mayor calificación la obtendrá la tecnología cuyo tiempo de ejecución sea el menor.

t = menor tiempo de ejecución

Parámetro	Valor	Valor Cualitativo
t	100	Excelente
t + 30%t	75	Muy Bueno
t + 60%t	50	Bueno
t + 80%t	25	Regular
> t + 100%t	0	Malo

Tabla.III.VII. Escala Cuali-cuantitativa para el parámetro tiempo de ejecución

Elaborador por: Autor

- **FPS(Fotogramas por segundo)**

Los fotogramas por segundo es la medida de la frecuencia a la cual un reproductor de imágenes genera distintos fotogramas (frames). Estos fotogramas están constituidos por un número determinado de píxeles que se distribuyen a lo largo de una textura. La frecuencia de los fotogramas es proporcional al número de píxeles que se deben generar, incidiendo en el rendimiento del ordenador que los reproduce.[21]

Mediante éste parámetro se refleja la fluidez con la que se dibuja la escena.

Para definir la escala, se toma en cuanto a algunos factores con respecto a la visión humana, y es en función de éstos sobre los cuales se elaboró la escala de evaluación; sobre los 30 fps el ojo humano no detecta el cambio entre frames, es decir que la animación se nota fluida y no se percibe cada cuadro dibujado, entre 24 y 29 se empieza a percibir alguna ralentización, entre 15 y 23 la ralentización es claramente

perceptible y por debajo de esta es claramente visible cada frame dibujado. Por debajo de los 5 fps, no se percibe la animación como debería ser.

$$\boldsymbol{f} = cantidad\ de\ FPS$$

Parámetro	Valor	Valor Cualitativo
$\boldsymbol{f \geq 30}$	100	Excelente
$\boldsymbol{24 < f < 30}$	75	Muy Bueno
$\boldsymbol{15 < f < 24}$	50	Bueno
$\boldsymbol{5 < f < 15}$	25	Regular
$\boldsymbol{f < 5}$	0	Malo

Tabla.III.VIII. Escala Cuali-cuantitativa para el parámetro Frames por segundo

Elaborado por: Autor

- **Uso de CPU**

Otro parámetro importante a medir es el uso del procesador, es decir qué porcentaje del procesador pasa ocupado durante la ejecución del programa. Esto es importante, ya que el CPU no sólo debe ejecutar la aplicación gráfica sino también debe encargarse del resto de tareas relacionadas con el sistema operativo.

Se hará uso del monitor del sistema para medir la carga del procesador con el contador % de tiempo de procesador que es el porcentaje de tiempo que el procesador invierte en ejecutar el proceso.

Para establecer la escala de calificación se considera la mayor calificación a la tecnología cuyo tiempo de uso del procesador sea menor.

t_p = menor tiempo de uso del procesador

Parámetro	Valor	Valor Cualitativo
$\mathbf{t_p}$	100	Excelente
$\mathbf{t_p + 30\% t_p}$	75	Muy Bueno
$\mathbf{t_p + 60\% t_p}$	50	Bueno
$\mathbf{t_p + 80\% t_p}$	25	Regular
$\mathbf{> t_p + 100\% t_p}$	0	Malo

Tabla.III.IX. Escala Cuali-cuantitativa para el parámetro uso de CPU

Elaborado por: Autor

- **Uso de memoria**

Consiste en medir la cantidad de memoria RAM utilizada al momento de ejecución de la aplicación. Para obtener éste dato se hará uso de la herramienta gDEbugger y el monitor del sistema de Windows.

La mayor calificación la obtendrá la tecnología que use menor cantidad de memoria.

m = menor cantidad de memoria

Parámetro	Valor	Valor Cualitativo
m	100	Excelente
m + 30%t	75	Muy Bueno
m + 60%t	50	Bueno
m + 80%t	25	Regular
> m + 100%t	0	Malo

Tabla.III.X. Escala Cuali-cuantitativa para el parámetro de uso de memoria

Elaborado por: Autor

3.6 Descripción del Escenario de Pruebas

3.6.1 Equipo Utilizado

El escenario de pruebas está constituido por una única máquina en la cual se han realizado todos las pruebas. La máquina es un computador portátil que tiene las siguientes especificaciones hardware:

	Características
Procesador	Intel Core i7 @ 1.73 GHz
Tarjeta Gráfica	nVidia Geforce GTX 460M
Memoria RAM	8 GB
Disco Duro	2x500 GB

Tabla.III.XI. Características PC para las pruebas

Elaborado por: Autor

En cuanto a software, el equipo de pruebas presenta la siguiente configuración:

Sistema Operativo	**Windows 7 Professional 64 bits**
.Net Framework	3.5
SDK OpenCL	SDK CUDA Tollkit 3.2
OpenCL	1.1
Drivers GPU	nVidia Geforce 296.10

Tabla.III.XII. Configuración software del equipo de pruebas

Elaborado por: Autor

3.6.2 Tecnologías a comparar

La siguiente tabla muestra las plataformas a comparar:

	CPU	**GPU**
	Intel Core i7 740 QM	NVIDIA GTX 460M
Frecuencia Procesador	1.73 GHz	675 MHz
Frecuencia Memoria	667 MHz	1250 MHz
Tipo de Memoria	DDR3	GDDR5
RAM	8 GB	1.5 GB
Transferencia GPU/CPU - Memoria	7 GB/s	60 GB/s
GIGAFLOPS	54	297.202

Tabla.III.XIII. Plataformas a comparar

Elaborado por: Autor

3.7 Herramientas software utilizadas en la investigación

El software utilizado para desarrollar el prototipo sobre el cual se va a realizar las pruebas y el análisis de resultados es el siguiente:

3.7.1 Para el desarrollo del prototipo

Las herramientas software necesarias para desarrollar el prototipo son:

API para el manejo de la programación concurrente y paralela	**OpenCL**
Lenguaje de programación	C++
IDE	Visual Studio 2008
API gráfica	OpenGL

Tabla.III.XIV. Software utilizado para el desarrollo del prototipo

Elaborado por: Autor

La selección de OpenGL como API gráfica tiene como principal razón la interoperabilidad que permite con OpenCL.

3.7.2 Para la medición de los parámetros

Para realizar la medición de los parámetros se hace uso de algunas herramientas:

- gDEBugger. Es un avanzado depurador, profiler y analizador de memoria para OpenGL y OpenCL. gDEBugger permite rastrear la actividad de aplicaciones en la parte superior de la API OpenGL y OpenCL y ver lo que está sucediendo dentro de la implementación del sistema [9]

- FRAPS. Es una aplicación que puede ser usada con las tecnologías gráficas OpenGL y DirectX para realizar las siguientes tareas:
  - Benchmarking de software
  - Captura de pantalla de la aplicación gráfica
  - Captura de video de la aplicación gráfica

  En la presente investigación FRAPS será usado para medir la tasa de fotogramas por segundo a los que trabaja el prototipo.

- Monitor del sistema de Windows. Es una herramienta gráfica integrada en el sistema operativo. Es la principal herramienta de monitorización en tiempo real y de análisis de datos del uso de recursos del sistema.

3.8 Evaluación de los parámetros

Una vez definidos los parámetros, la escala de valoración y las herramientas a usar para recolectar los datos, se procede a evaluar y mostrar los resultados obtenidos en las mediciones. Las pruebas para medir los parámetros se realizarán sobre un prototipo que consiste en una aplicación que grafica el fractal de Mandelbrot, este se eligió debido a su dificultad, desde el punto de vista computacional; sobre el cual se realizará una animación que consiste en hacer zoom a un punto específico.

Las pruebas serán ejecutadas sobre implementaciones de la aplicación hechas para las tecnologías GPU y CPU como se especifica en la descripción del escenario de pruebas.

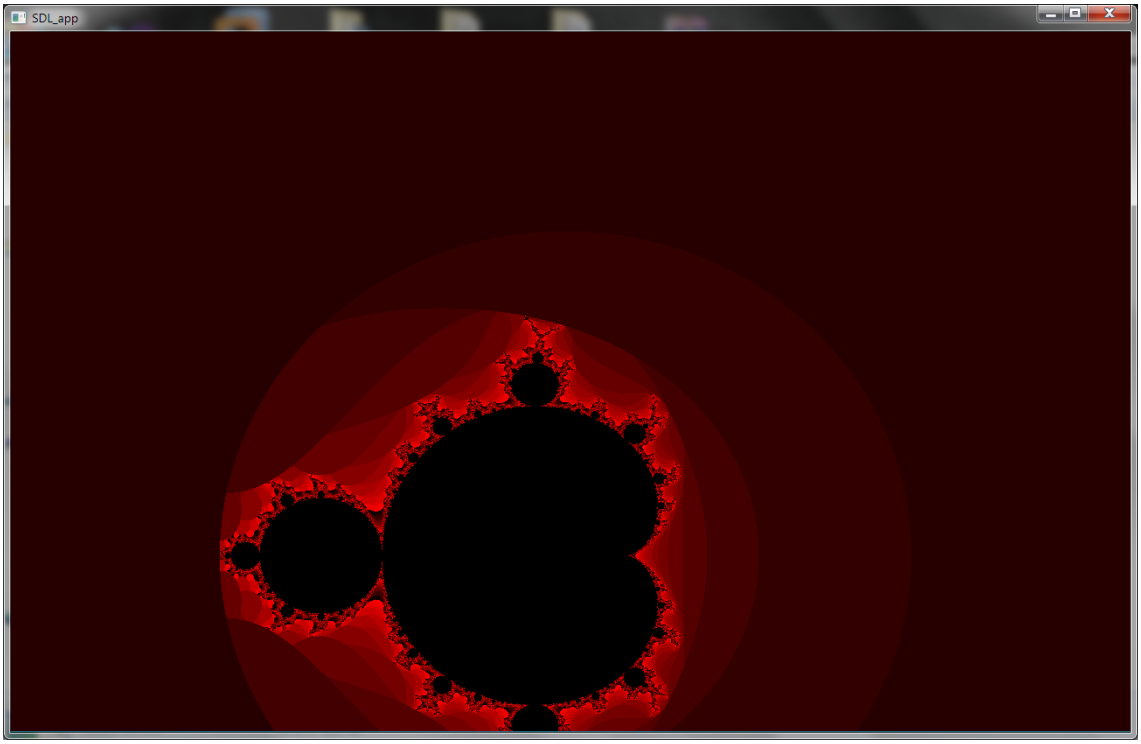


Figura.III.23. Captura del prototipo para la medición de los parámetros de evaluación

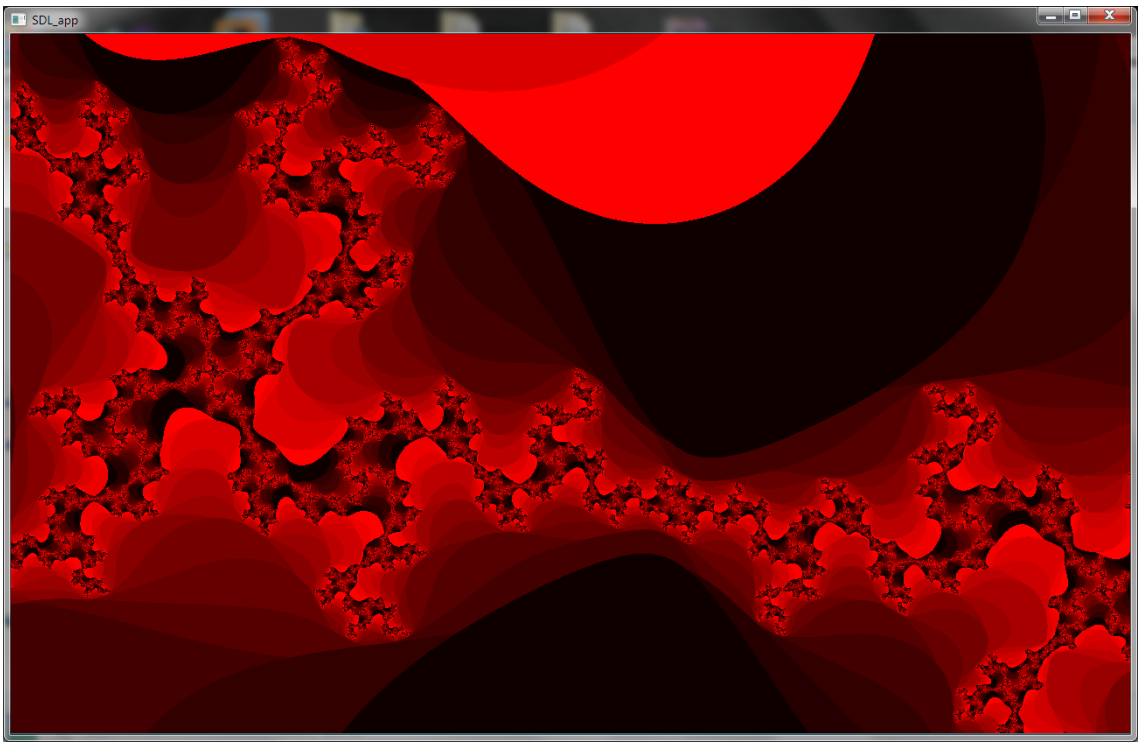


Figura.III.24. Captura del prototipo para la medición de los parámetros de evaluación

3.9 Análisis de Resultados

El análisis se lo realizará mediante pruebas en el prototipo, debido a que las medidas probarán diferencias en el rendimiento, estas medidas serán tomadas, dependiendo del criterio analizado, al ejecutarse en una u otra tecnología.

Las pruebas a realizarse sobre el prototipo consiste en realizar una animación en la cual se hace un Zoom de alrededor de 90X a un punto específico del fractal, con esto se podrá medir la los distintos parámetros a medida que se va generando el fractal y mientras se realiza la animación.

3.9.1 Tiempo de ejecución

La siguiente tabla muestra los resultados obtenidos:

Tecnología	Media-Tiempo de Ejecución (segundos)	Desviación Estándar
CPU	45.16	0.4533
GPU	20.38	0.4582

Tabla.III.XV. Resultados obtenidos de la medición de Tiempo de Ejecución

Elaborado por: Autor

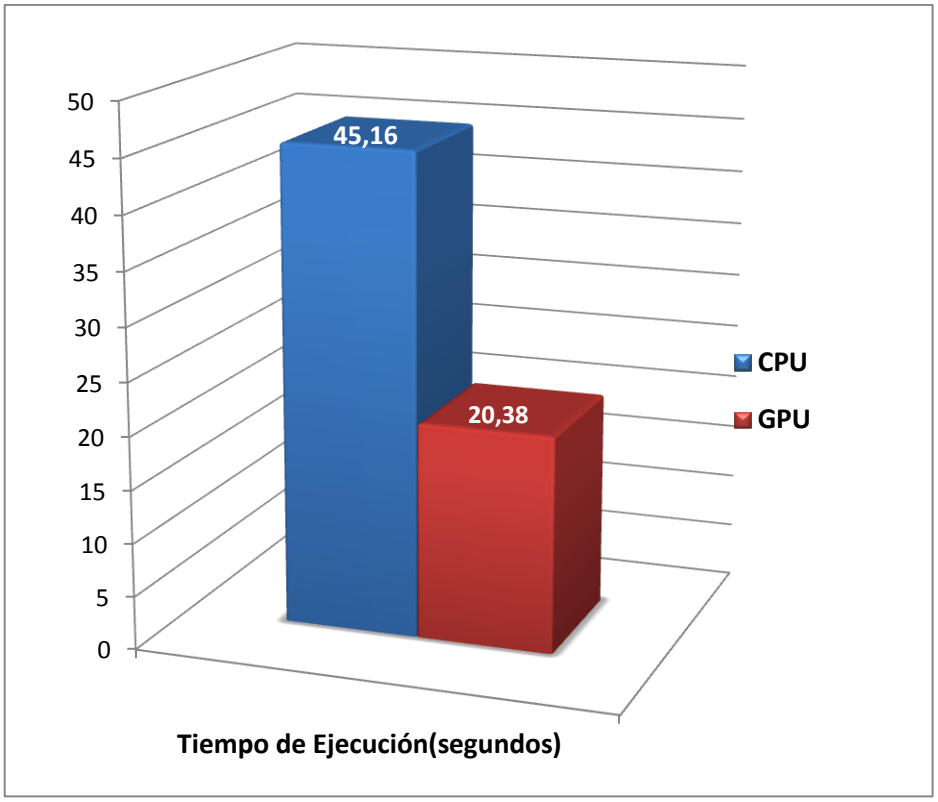


Figura.III.25. Gráfico de Resultados de Tiempo de Ejecución

Como se observa hay una gran diferencia de tiempos cuando el algoritmo es ejecutado sobre cada una de las tecnologías analizadas, para el análisis vamos a mapear los datos en función de la escala definida con anterioridad, con lo cual obtenemos el siguiente resultado:

Tecnología	Valor	Valor Cualitativo
CPU	0	Malo
GPU	100	Excelente

Tabla.III.XVI. Mapeo de los resultados obtenidos con la escala de evaluación del Tiempo de Ejecución definida anteriormente.

Elaborado por: Autor

La ejecución del prototipo sobre la tecnología GPU obtuvo el menor tiempo, por lo cual tiene la mayor calificación (100), en el caso de CPU equivale a t(menor tiempo) + 122% por lo cual su calificación es 0. Analizando los resultados obtenidos se observa una

gran diferencia con respecto al tiempo en que se tardan las 2 plataformas en completar la animación.

3.9.2 Fotogramas por segundo

Para medir este parámetro se hizo uso de la herramienta FRAPS. Los resultados de la medición son los siguientes:

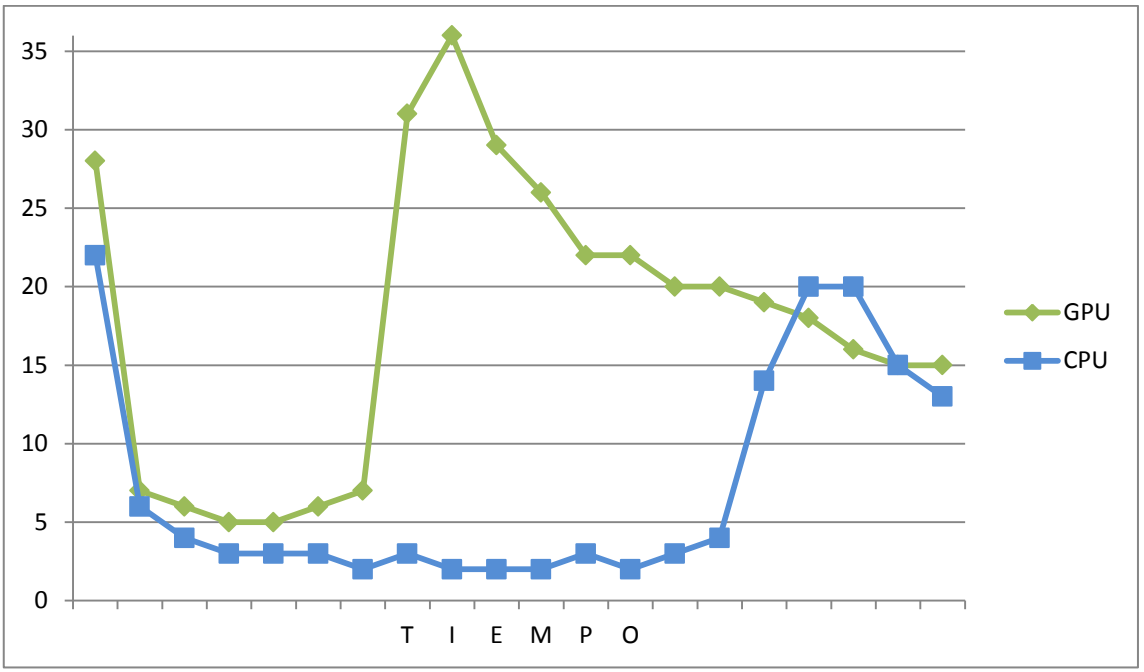


Figura.III.26. Comportamiento de FPS en función del tiempo

Tecnología	FPS		
	Mínimo	Máximo	Media
CPU	2	22	8
GPU	4	36	18

Tabla.III.XVII. Resultados obtenidos de la medición de FPS

Elaborado por: Autor

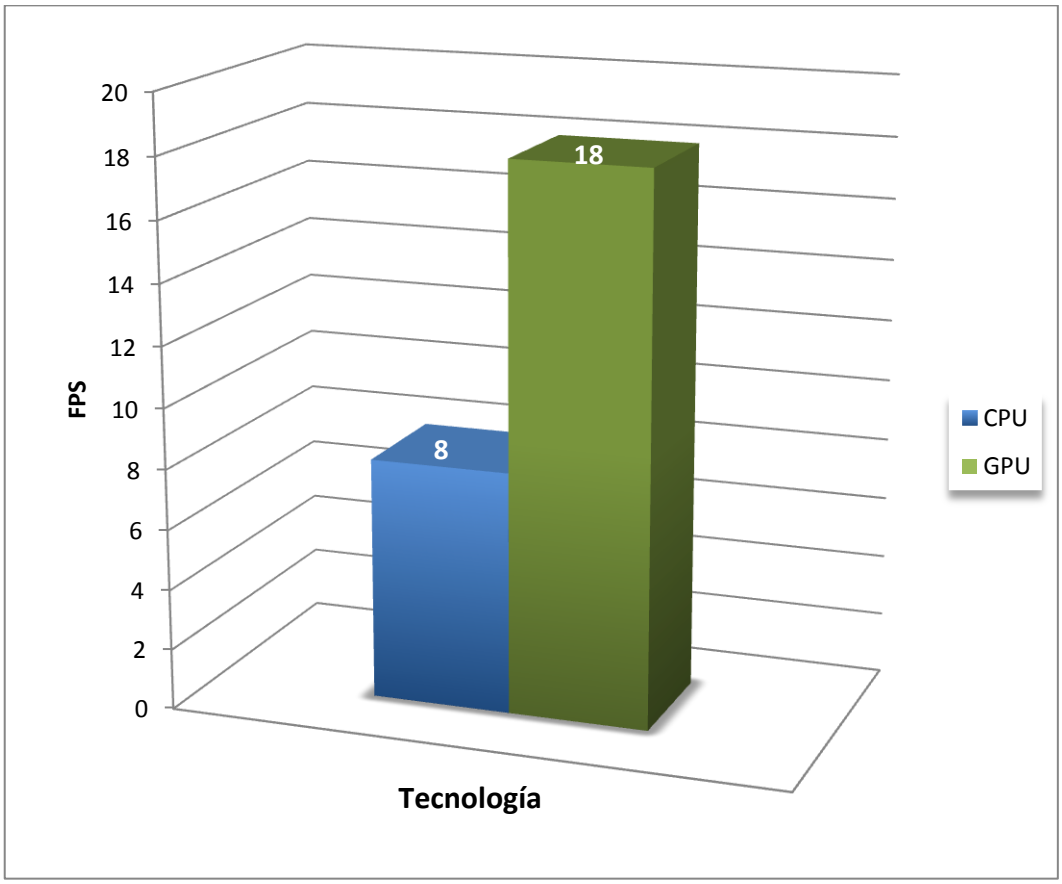


Figura.III.27. Gráfico de Resultados de la medición de FPS

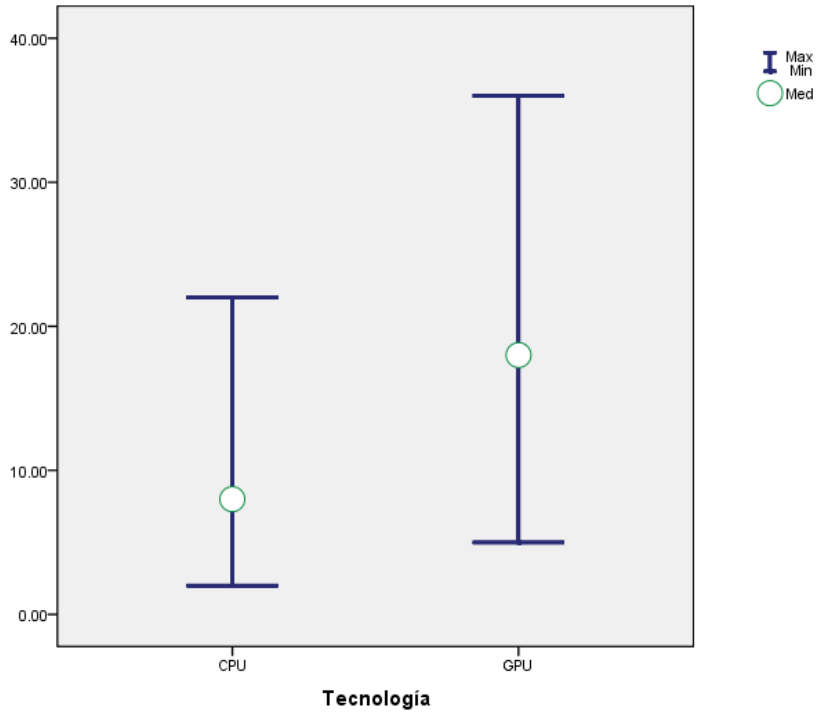


Figura 28. Diagrama de Máximos y Mínimos en base a los datos de la medición de FPS

En función de la escala definida con anterioridad, obtenemos el siguiente resultado:

Tecnología	Valor	Valor Cualitativo
CPU	25	Regular
GPU	50	Bueno

Tabla.III.XVIII. Mapeo de los resultados obtenidos según la escala de evaluación

Elaborado por: Autor

En el caso de este parámetro se había definido que sobre los 30 fps se considera una aplicación que se visualiza de manera fluida, pero como se observa ninguna de las tecnologías alcanza dicha cifra (lo cual también se lo representa mediante el siguiente gráfico), por lo tanto en CPU con una tasa de fps de 8 obtenemos una calificación de 25 (Regular) y en GPU con 18 fps obtenemos una calificación de 50 o equivalente a Bueno.

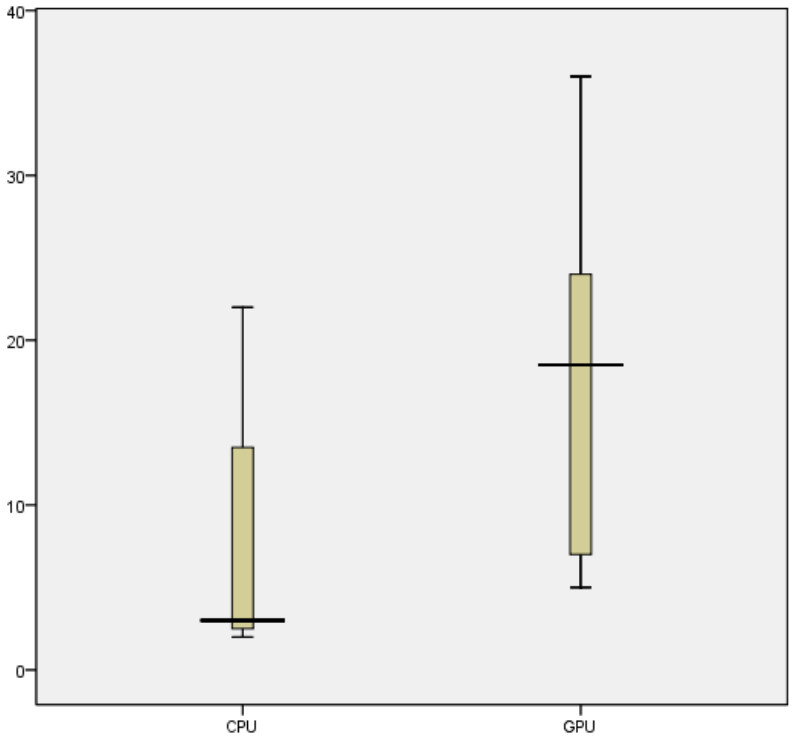


Figura.III.29. Diagrama de Caja y Bigotes con los datos de la medición de FPS

3.9.3 Uso de Memoria

Tecnología	Uso de Memoria (MB)		
	Mínimo	Máximo	Media
CPU	20.52	88.90	86.82
GPU	12.93	48.21	48.20

Tabla.III.XIX. Resultados obtenidos de la medición de Uso de Memoria

Elaborado por: Autor

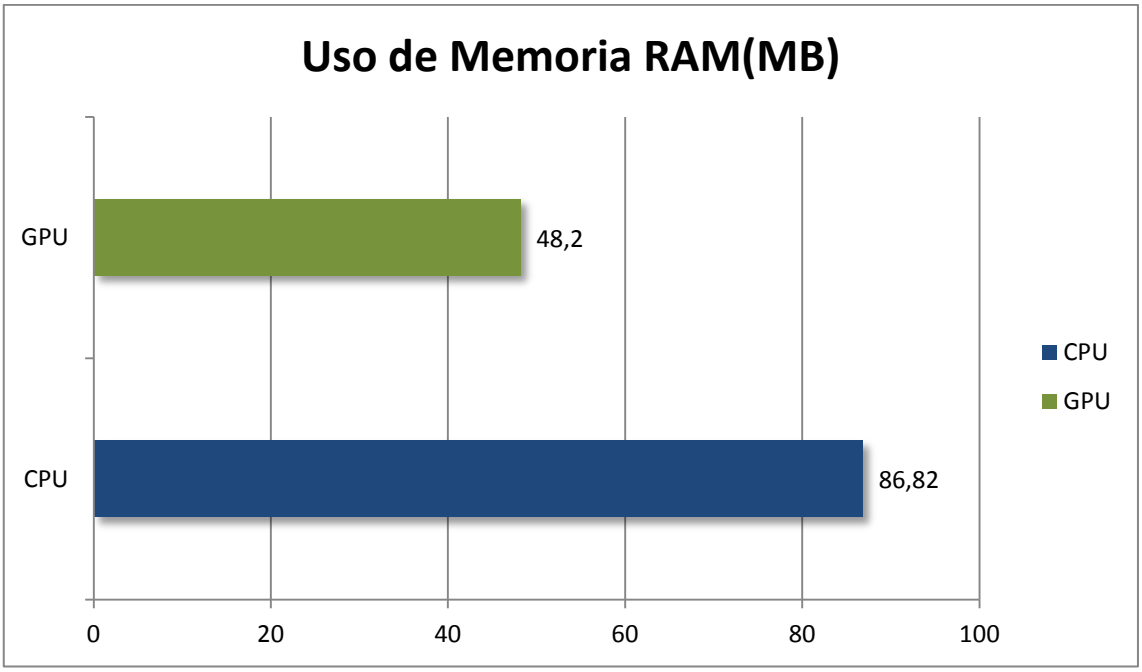


Figura.III.30. Gráfico de Resultados de la medición de Uso de Memoria

La ejecución sobre la tecnología GPU obtuvo el menor consumo de memoria, por lo cual tiene la mayor calificación (100) y en el caso de CPU equivale a m(menor uso de memoria) + 80% por lo cual su calificación es 25.

Tecnología	Valor	Valor Cualitativo
CPU	25	Regular
GPU	100	Excelente

Tabla.III.XX. Mapeo de los resultados obtenidos según la escala de evaluación

Elaborado por: Autor

3.9.4 Uso de Procesador

Se obtuvieron los siguientes resultados:

Tecnología	Uso de Procesador(%)		
	Mínimo	Máximo	Media
CPU	8.20	86.70	67.40
GPU	1.50	13.60	10.48

Tabla.III.XXI. Resultados obtenidos de la medición del Uso de Procesador

Elaborado por: Autor

El siguiente gráfico muestra el comportamiento en el tiempo del uso del Procesador mientras se ejecuta el prototipo, se puede observar claramente la diferencia entre la ejecución en cada una de las tecnologías.

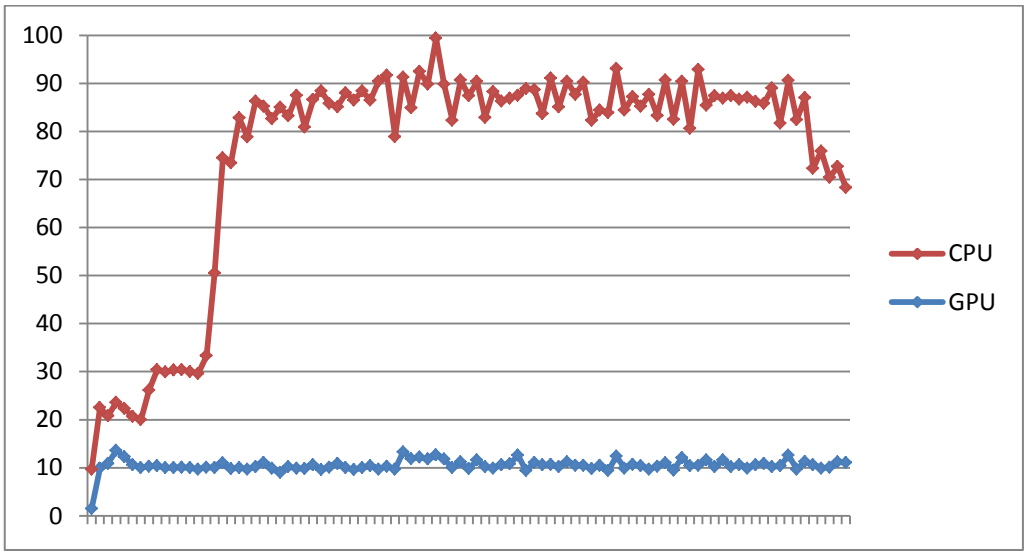


Figura.III.31. Gráfico de Resultados de la medición del Uso de Procesador

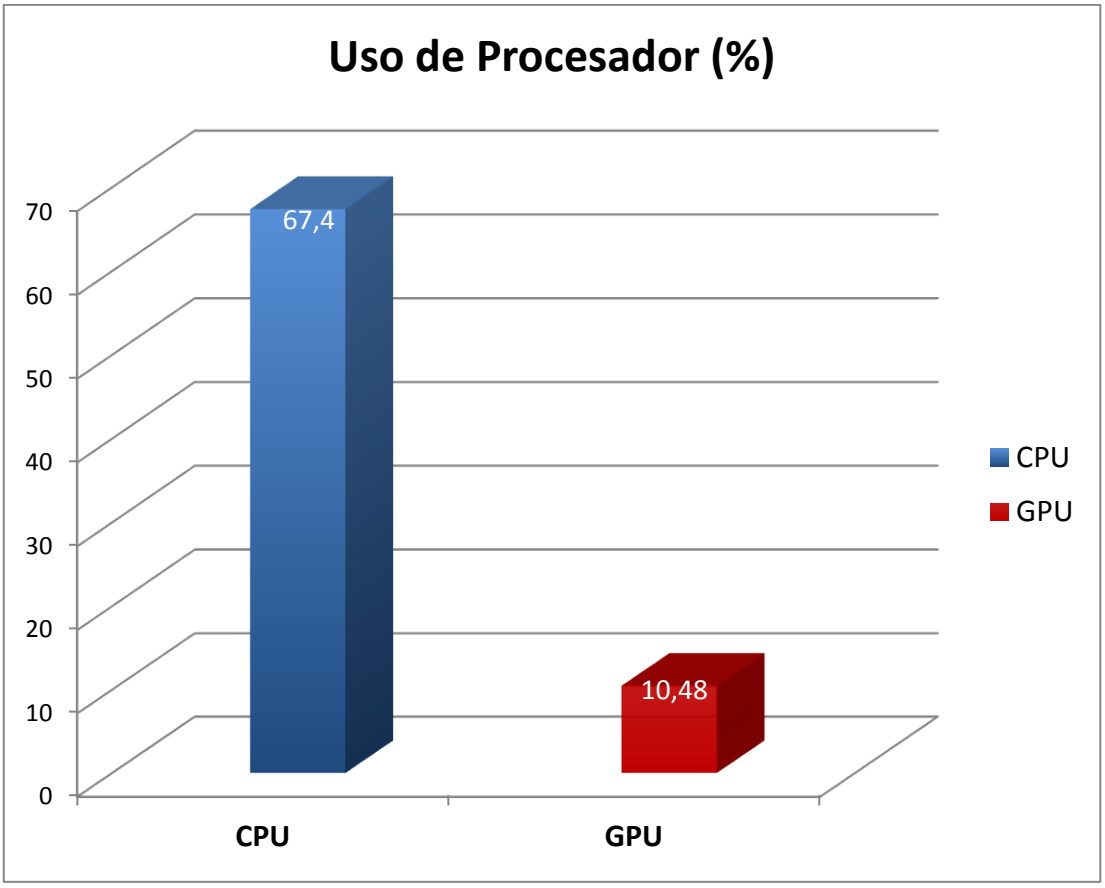


Figura 32. Gráfico de resultados de la medición del Uso del Procesador

En función de la escala definida con anterioridad, obtenemos el siguiente resultado:

Tecnología	Valor	Valor Cualitativo
CPU	0	Malo
GPU	100	Excelente

Tabla.III.XXII. Mapeo de los resultados obtenidos según la escala de evaluación

Elaborado por: Autor

La ejecución sobre GPU es la que menor tiempo de procesador consume, y esto es algo lógico ya que se está usando en mayor medida la capacidad de cómputo del procesador gráfico, es decir de la GPU. según la escala definida, la diferencia es muy grande, ya que en el caso de CPU constituye el menor tiempo de uso + 543%.

3.10 Análisis de Resultados y Comprobación de Hipótesis

A continuación se realizará el análisis de resultados de los datos recolectados, y la sumatoria de las calificaciones obtenidas en cada uno de los parámetros definidos en la evaluación, con lo cual se concluirá con la mejor tecnología para la implementación.

	Tiempo de Ejecución		Frames por Segundo		Uso del Procesador		Uso de Memoria	
	Valor	Calificación	Valor	Calificación	Valor	Calificación	Valor	Calificación
CPU	0	Malo	25	Regular	0	Malo	25	Regular
GPU	100	Excelente	50	Bueno	100	Excelente	100	Excelente

Tabla.III.XXIII. Resultados de las mediciones de cada uno de los parámetros

Elaborado por: Autor

Los puntajes obtenidos en cada tecnología se mapean a continuación, según la escala de evaluación definida anteriormente:

	Tiempo de Ejecución	Frames por Segundo	Uso del Procesador	Uso de Memoria	TOTAL
CPU	0	12.5	0	2.5	14.5
GPU	30	25	10	10	75

Tabla.III.XXIV. Sumatoria de los puntos obtenidos por tecnología

Elaborado por: Autor

El siguiente gráfico representa la sumatoria de los puntos obtenidos por cada tecnología:

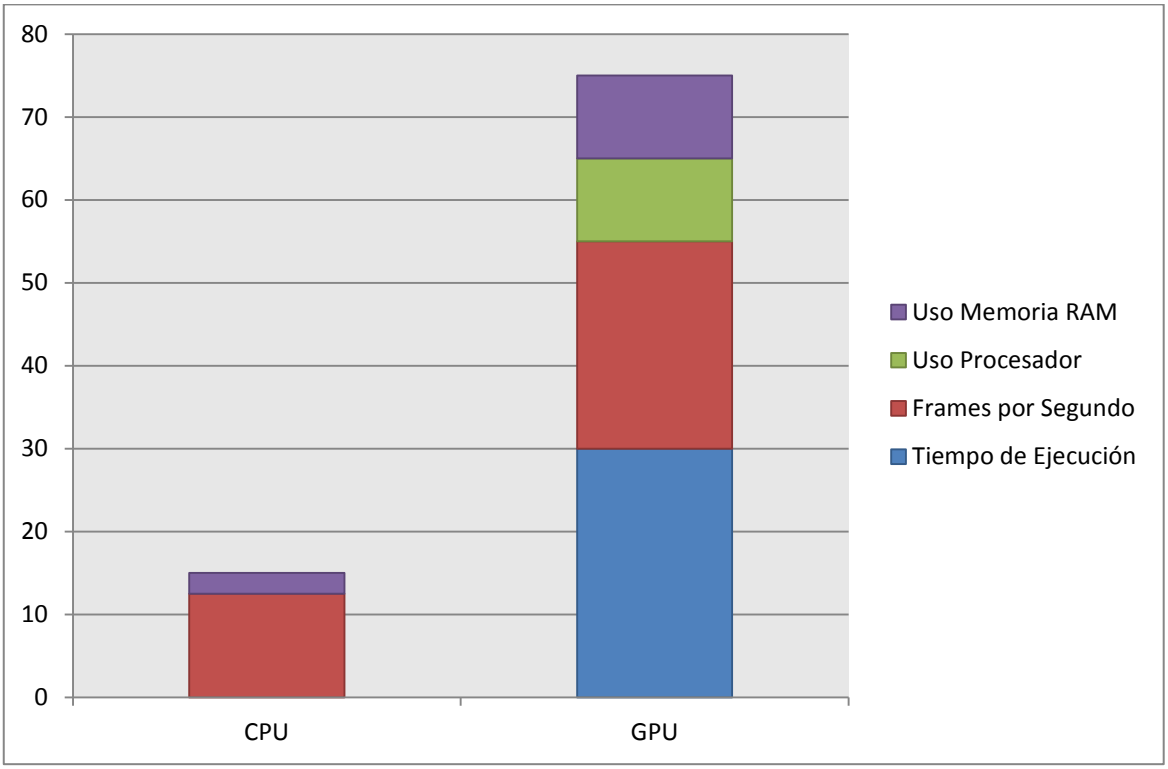


Figura.III.33. Gráfico resumen del resultado total de la medición de los parámetros

Resultados:

- CPU 14,5 puntos
- GPU 75 puntos

En base a los resultados obtenidos se puede concluir lo siguiente:

- De acuerdo a los resultados obtenidos que se resumen en la tabla anterior se puede afirmar que la tecnología más adecuada para implementar la aplicación es GPU .
- La tecnología GPU obtuvo el mayor puntaje, 75 puntos y se observa una clara diferencia con respecto a la otra tecnología, CPU, que obtuvo 14,5.
- En el caso del parámetro Tiempo de ejecución, la diferencia es clara, la tecnología CPU obtuvo 0 puntos frente a 30 puntos en GPU.

- En lo que se refiere al parámetro Uso de procesador, existe igualmente un claro ganador, ya que CPU obtiene 0 puntos, esto es debido a que en el caso de GPU el uso de procesador baja radicalmente debido a que la mayor cantidad de procesamiento se realiza sobre el procesador gráfico.

Comprobación de Hipótesis

En base a las mediciones realizadas y a los datos obtenidos para su posterior análisis haciendo uso de la estadística descriptiva se puede llegar a comprobar la hipótesis planteada en la presente tesis, mediante la cual se puede afirmar que la tecnología GPU es la más adecuada al momento de implementar la aplicación gráfica Fractal Build, esto debido a las claras ventajas con respecto a la tecnología CPU que se observan en cada una de las pruebas realizadas, es así, que la tecnología GPU obtuvo 75 puntos en la sumatoria de los puntajes de todos los parámetros, siendo mejor que CPU la cual obtuvo 14.5 puntos.

CAPÍTULO IV

DESARROLLO DE FRACTAL BUILD

4.1 VISIÓN

4.1.1 Definición del Problema

En la Escuela de Ingeniería en Sistemas se dicta la materia de Computación Gráfica, uno de los temas tratados en dicha materia es la creación de fractales, para lo cual se requiere de una aplicación real de los fractales. Como resultado de la tesis “ANÁLISIS DE LA PROGRAMACIÓN CONCURRENTE SOBRE LA CPU Y GPU EN EL DESARROLLO DE FRACTAL BUILD” se desarrollará una aplicación gráfica (Fractal Build) para la generación de un paisaje fractal, que sirva como aplicación real de los fractales para la materia antes mencionada.

4.1.2 Visión del proyecto

Se plantea la solución mediante el desarrollo de una aplicación gráfica que permite dibujar un paisaje fractal, en donde se pueda visualizar terrenos, nubes y un edificio fractal. Las principales funcionalidades son la navegación por el paisaje mediante el uso del teclado y mouse, así como la posibilidad de guardar lo que se visualice en un archivo JPG.

4.1.3 Ámbito del proyecto

4.1.3.1 Alcance del Proyecto

El sistema Fractal Build se centrará principalmente en graficar varios objetos 3d en base a fractales. Además ofrecerá la interfaz adecuada para realizar una navegación por el paisaje fractal que generará y la posibilidad de guardar lo que se visualice en un archivo JPG.

4.1.3.2 Requerimientos Generales

Número	Requerimiento
REQ1	El sistema debe permitir dibujar un edificio fractal generado en tiempo real
REQ2	El sistema debe permitir dibujar un terreno fractal
REQ3	El sistema debe permitir dibujar nubes fractales
REQ4	El sistema debe permitir la navegación por el paisaje fractal

REQ5	El sistema debe permitir configurar parámetros para el dibujado del terreno fractal
REQ6	El sistema debe permitir configurar parámetros para el dibujado de las nubes fractales
REQ7	El sistema debe permitir guardar en un archivo en disco lo que se visualice en un momento dado(en una imagen)

4.1.4 Concepto de la solución

Fractal Build es una aplicación gráfica para la generación de un paisaje fractal que se va a desarrollar haciendo uso de las siguientes herramientas:

4.1.4.1 Uso de Herramientas

Característica Técnica	Detalle
Plataforma operativa de funcionamiento	Windows 7 como principal sistema operativo, en donde se implementará la solución
Herramientas de Desarrollo	El IDE seleccionado para el desarrollo es Visual Studio 2008
Lenguaje(s) de Programación	C++ y como API gráfica OpenGL
Herramientas de Modelado UML	StarUML
Herramientas de Oficina	Microsoft Word, para la elaboración de la documentación sobre el desarrollo del proyecto
Metodología de Desarrollo	Microsoft Solution Framework (MSF)

4.1.4.2 Planteamiento de la Arquitectura de la Solución

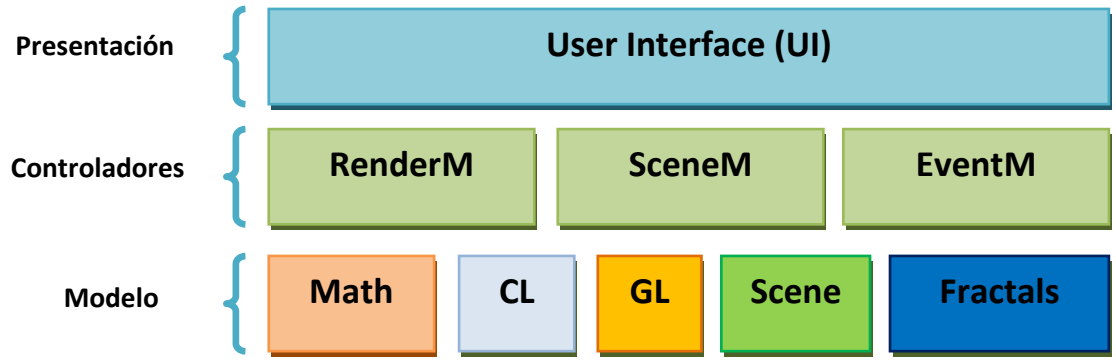


Figura.IV.34. Arquitectura de Fractal Build

Capa 1: Presentación	Engloba toda la interfaz de usuario.
Capa 2: Controladores	Corresponde con el modelo de controlador de MVC. Es la comunicación entre la interfaz de usuario y el núcleo de la aplicación.
Capa 3: Modelo	Constituye el núcleo de Fractal Build. Contiene los elementos necesarios que implementan las funcionalidades de la aplicación Fractal Build.

Tabla XXV. Resumen de la arquitectura de Fractal Build

Elaborado por: Autor

4.1.5 Objetivos del proyecto

4.1.5.1 Objetivos del Negocio

- Tener una aplicación práctica y real de los fractales.
- Brindar la posibilidad de guardar los fractales que se generen.

4.1.5.2 Objetivos del Diseño

- Lograr una aplicación gráfica que genere un paisaje fractal y que sea bastante usable.
- Tener un buen rango de fps para el dibujado de la aplicación.
- Brindar interfaces amigables para el manejo de la aplicación.

4.1.6 Planificación inicial

4.1.6.1 Recursos Software

- Sistema Operativo Microsoft Windows 7 Professional.
- Microsoft Office 2010
- Microsoft Project 2010
- Mozilla Firefox
- Microsoft Visual Studio 2008 Professional

4.1.6.2 Recursos Físicos Hardware

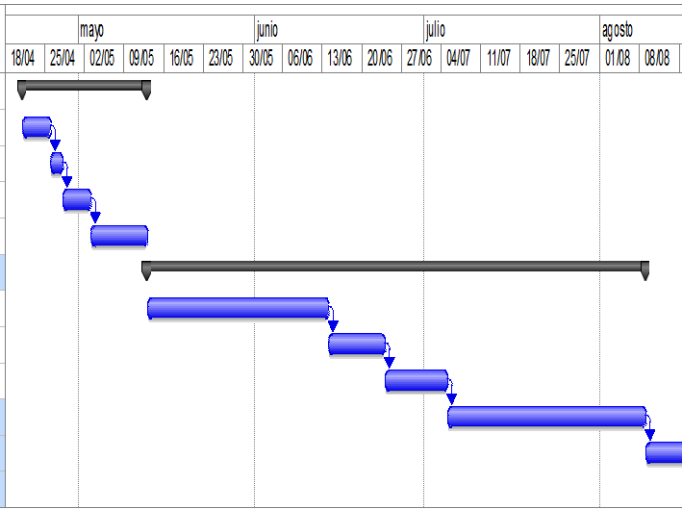
Equipo	Detalle
Máquina 1	Asus Intel Core i7 1.77 GHz, 8GB RAM, 2x500 GB HDD

Tabla XXVI. Recursos Hardware para el desarrollo de la aplicación

Elaborado por: Autor

4.1.6.3 Planificación de Actividades

Id	Nombre de tarea	Duración	Comienzo	Fin
1	**Definición del Problema**	**16 días**	**jue 21/04/11**	**jue 12/05/11**
2	Concepto de la Solución	3 días	jue 21/04/11	lun 25/04/11
3	Definición de Requerimier	2 días	mar 26/04/11	mié 27/04/11
4	Definición de Casos de Us	3 días	jue 28/04/11	lun 02/05/11
5	Definición del Modelo Lóg	8 días	mar 03/05/11	jue 12/05/11
6	**Desarrollo**	**62 días**	**vie 13/05/11**	**lun 08/08/11**
7	Desarrollo del Motor de R	22 días	vie 13/05/11	lun 13/06/11
8	Desarrollo de Interfaces	8 días	mar 14/06/11	jue 23/06/11
9	Selección de Algoritmos	7 días	vie 24/06/11	lun 04/07/11
10	Implementación de Algorit	25 días	mar 05/07/11	lun 08/08/11
11	Pruebas	5 días	mar 09/08/11	lun 15/08/11
12	Despliegue	5 días	mar 16/08/11	lun 22/08/11



4.2 PLANEACIÓN

4.2.1 Especificación Funcional

4.2.1.1 Diseño conceptual

4.2.1.1.1 Requerimientos Funcionales

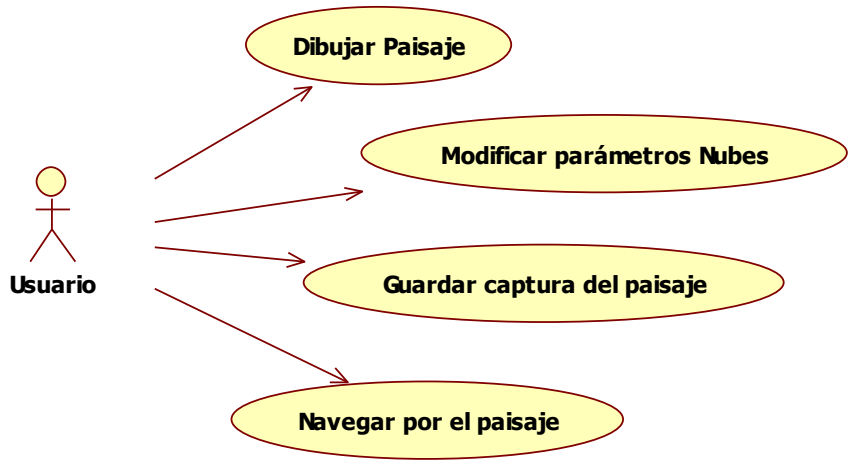
Número	Requerimiento
REQ1	El sistema debe permitir dibujar un edificio fractal generado en tiempo real
REQ2	El sistema debe permitir dibujar un terreno fractal
REQ3	El sistema debe permitir dibujar nubes fractales
REQ4	El sistema debe permitir la navegación por el paisaje fractal

REQ5	El sistema debe permitir configurar parámetros para el dibujado de las nubes fractales
REQ6	El sistema debe permitir guardar en un archivo en disco lo que se visualice en un momento dado(en una imagen)

4.2.1.1.2 Requerimientos No Funcionales

Requerimiento	Detalle
Fácil de manejar	El sistema debe permitir dibujar un edificio fractal generado en tiempo real
Estabilidad	El sistema debe permitir dibujar un terreno fractal
De fácil instalación	El sistema debe permitir dibujar nubes fractales
Buen rendimiento	El sistema debe permitir la navegación por el paisaje fractal

4.2.1.1.3 Casos de Uso y Escenarios

<table>
<tr><td>Titulo:</td><td colspan="2">Proceso General del Sistema</td></tr>
<tr><td colspan="3">Diagrama:
 
</td></tr>
<tr><td>Número de REQ:1, 2,3,4,5 ,6</td><td colspan="2">Usuario: usuario</td></tr>
<tr><td colspan="3">Nombre historia: Proceso General del Sistema</td></tr>
<tr><td colspan="2">Prioridad en negocio: Alta</td><td>Riesgo en desarrollo: Crítica</td></tr>
<tr><td colspan="2">Puntos estimados:3</td><td>Iteración asignada:1</td></tr>
<tr><td colspan="3">Programador responsable: Vicente Anilema</td></tr>
<tr><td colspan="3">Descripción:
• El usuario modifica los parámetros de las nubes
• El usuario guarda la captura del paisaje
• El usuario navega por el paisaje</td></tr>
<tr><td colspan="3">Curso Típico de Eventos:</td></tr>
<tr><td colspan="2">ACCIONES DE ACTORES</td><td>RESPUESTAS DEL SISTEMA</td></tr>
<tr><td colspan="2">1.- El usuario procede a ingresar al sistema.</td><td>2.- El sistema genera el paisaje fractal con los parámetros por defecto.</td></tr>
</table>

3.- El usuario modifica los parámetros de las nubes.	4.- Las nubes fractales se dibujan nuevamente.
1.- El usuario presiona las teclas de dirección y hace uso del mouse.	2.- Empieza la navegación por el paisaje.
1.- El usuario selecciona la opción de guardar imagen	4.- El sistema almacena la imagen en un archivo.

4.2.2 Diseño lógico

La arquitectura de Fractal Build se divide en 3 capas:

Capa 1: Presentación. Engloba toda la interfaz de usuario y es el encargado de dibujar los elementos de la interfaz, es decir, la ventana, el menú y las diferentes opciones.

Capa 2: Controladores. Se corresponde con el modelo de controlador de MVC. Es la comunicación entre la interfaz de usuario y el núcleo de la aplicación. De esta forma se desacopla la interfaz de usuario del núcleo de la aplicación. De esta forma un cambio estructural en la aplicación se disminuye el impacto que produciría. Se tiene 3 controladores, el de Renderizado, el de Escenas y el de Eventos.

Capa 3: Modelo. Constituye el núcleo de Fractal Build. Contiene los elementos necesarios que implementan las funcionalidades de la aplicación Fractal Build. El modelo se divide en 5 bloques:

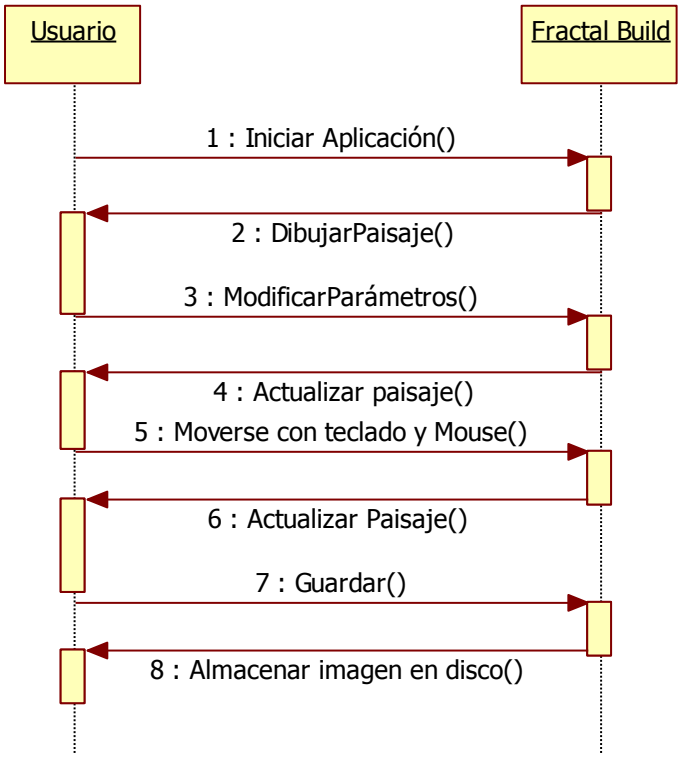
- Math, contiene las funciones que permiten manejar de forma más fácil el dibujado de un mundo 3D.
- CL, todo lo correspondiente al manejo del procesamiento haciendo uso de OpenCL.
- GL, se encarga del proceso de renderizado haciendo uso de OpenGL.
- Scene, se encarga del manejo y construcción de la escena

- Fractals, aquí se encuentran los algoritmos necesarios para la generación del paisaje fractal (nubes, edificio, terreno)

4.2.2.1 Tecnología a utilizar en el proyecto

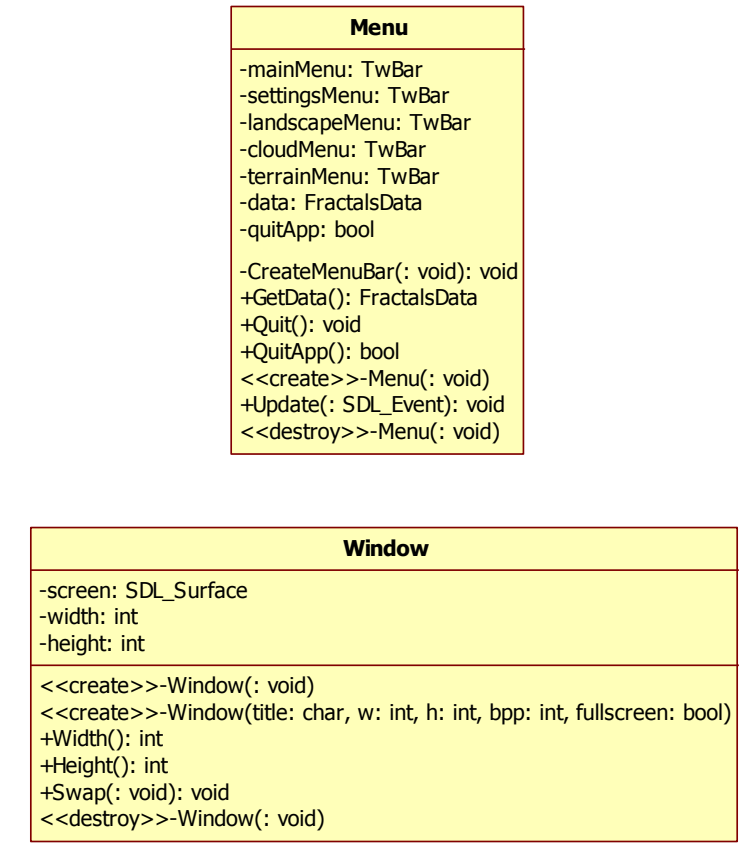
Lenguaje de programación	**C++**
IDE	Visual Studio 2008
API GPGPU	OpenCL
API Gráfica	OpenGL

4.2.2.2 Diagramas de secuencias

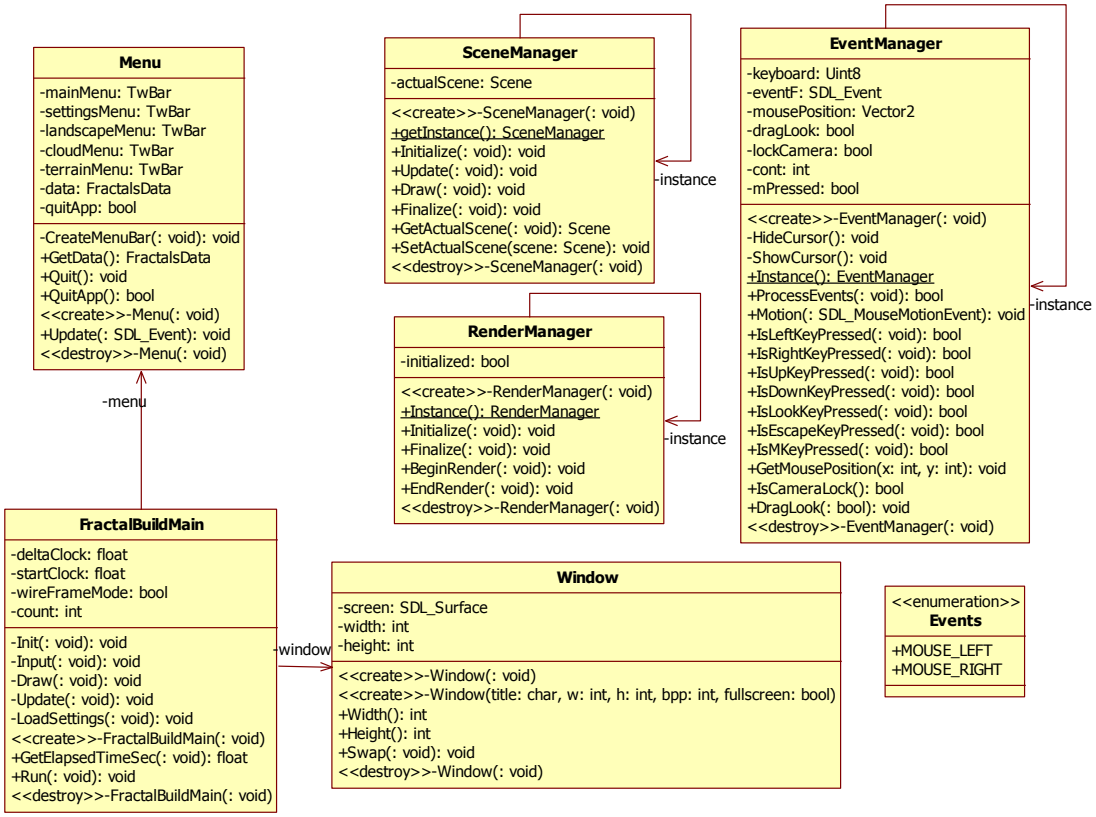


4.2.2.3 Diagramas de clases

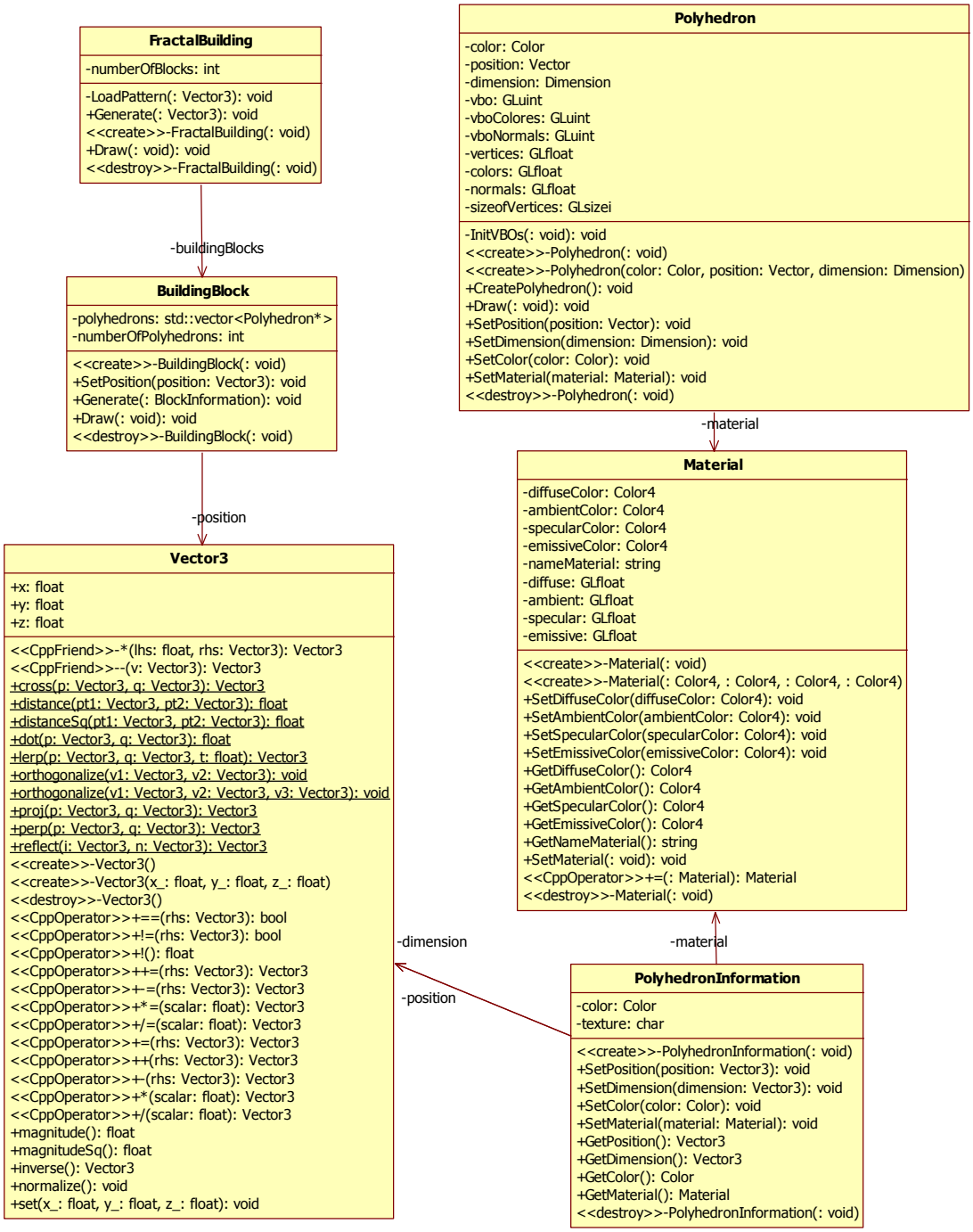
UI:

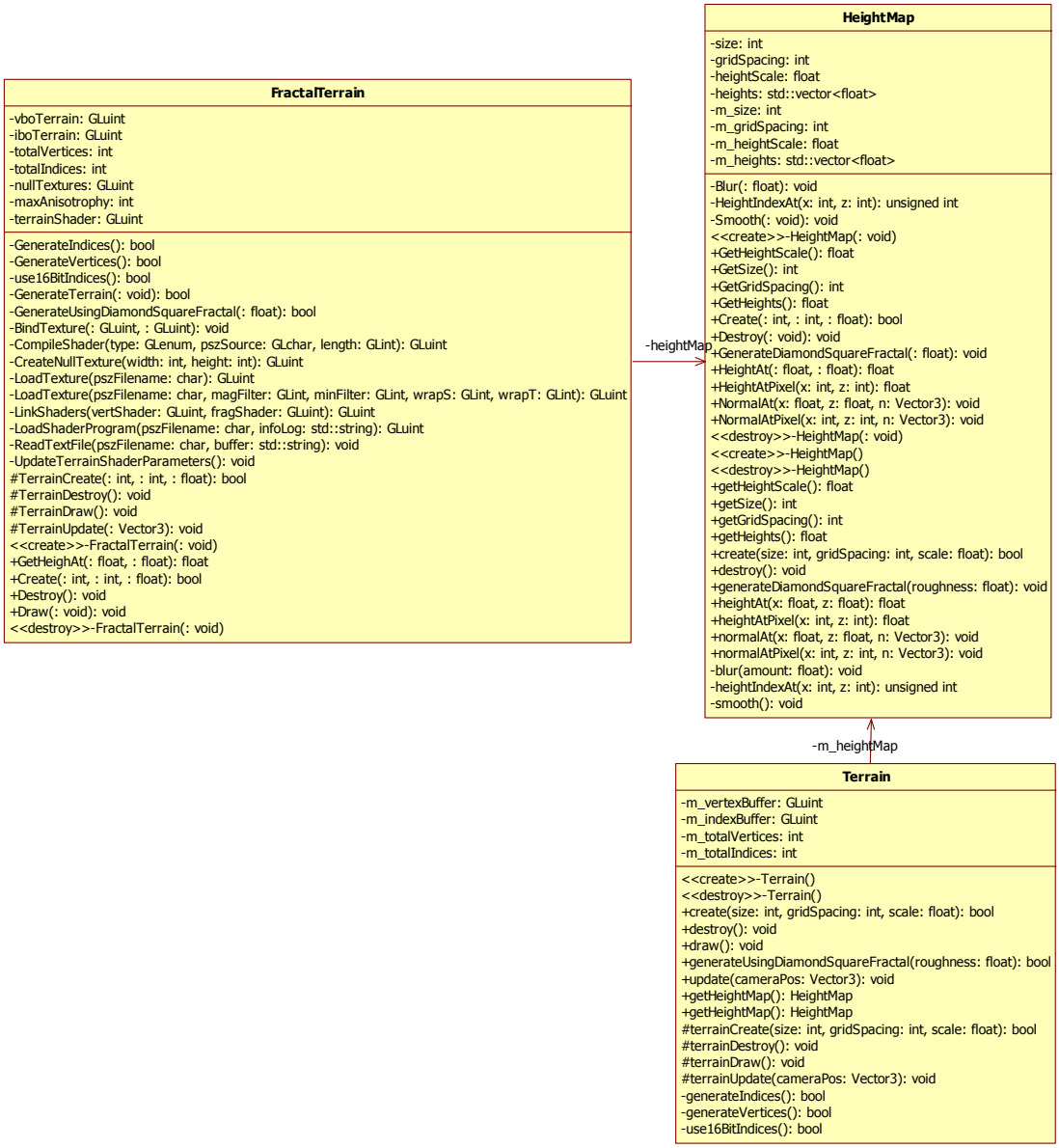


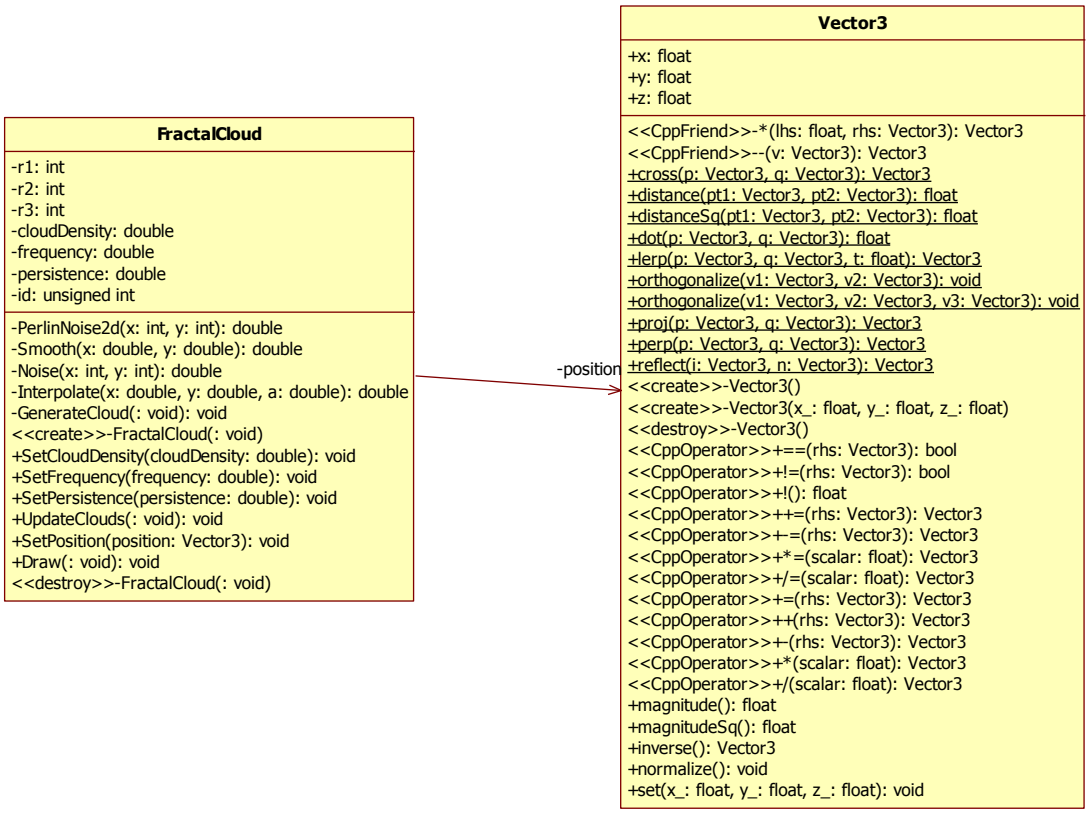
Controladores:

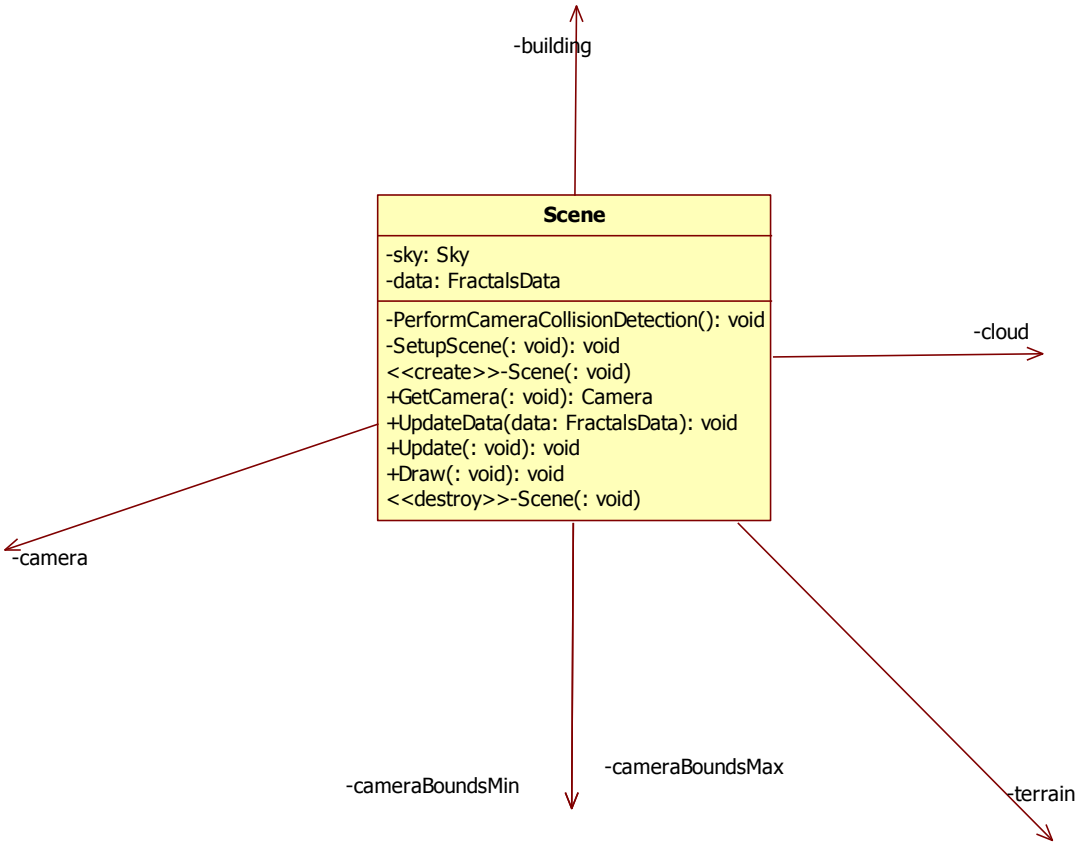


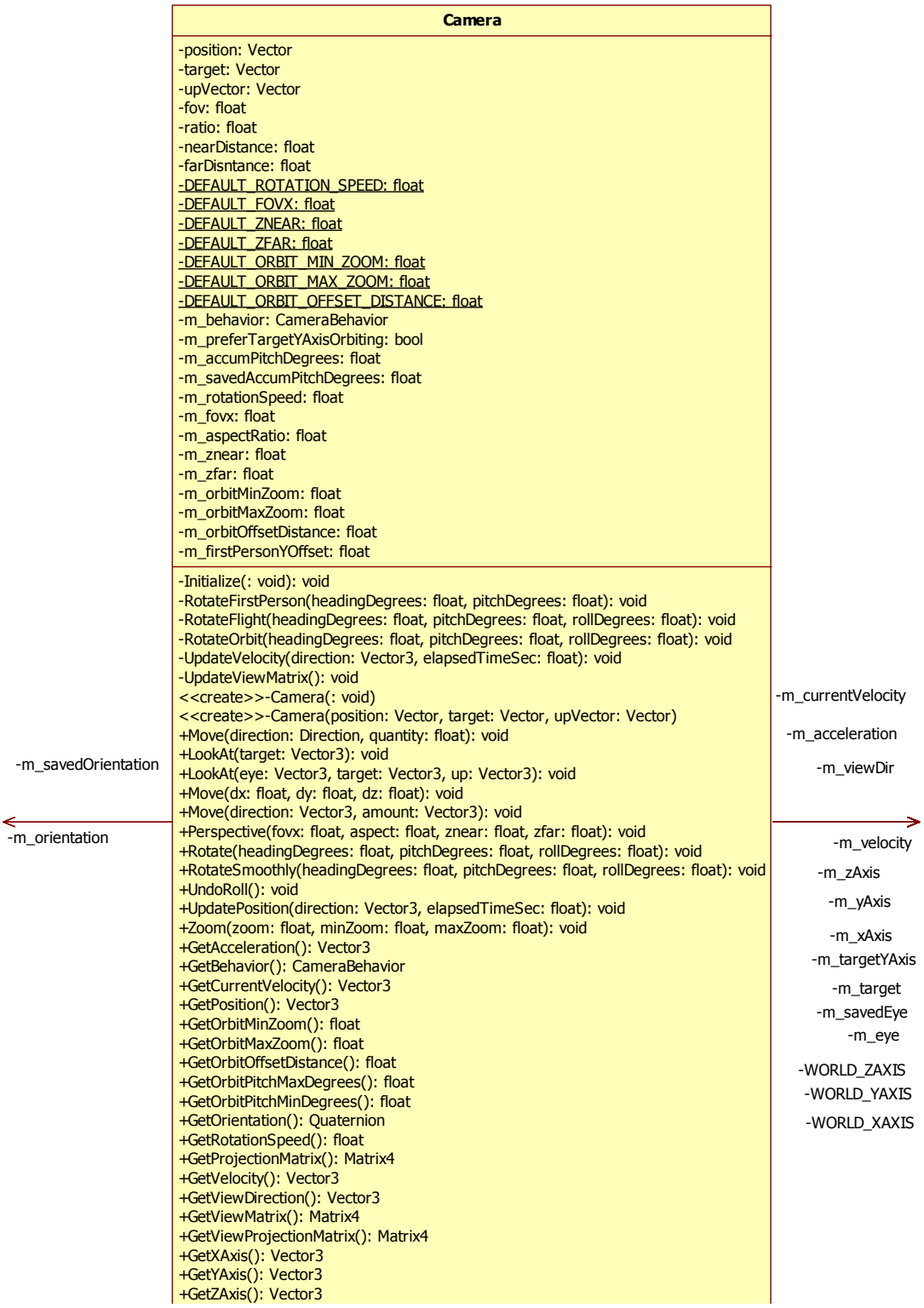
Renderer:

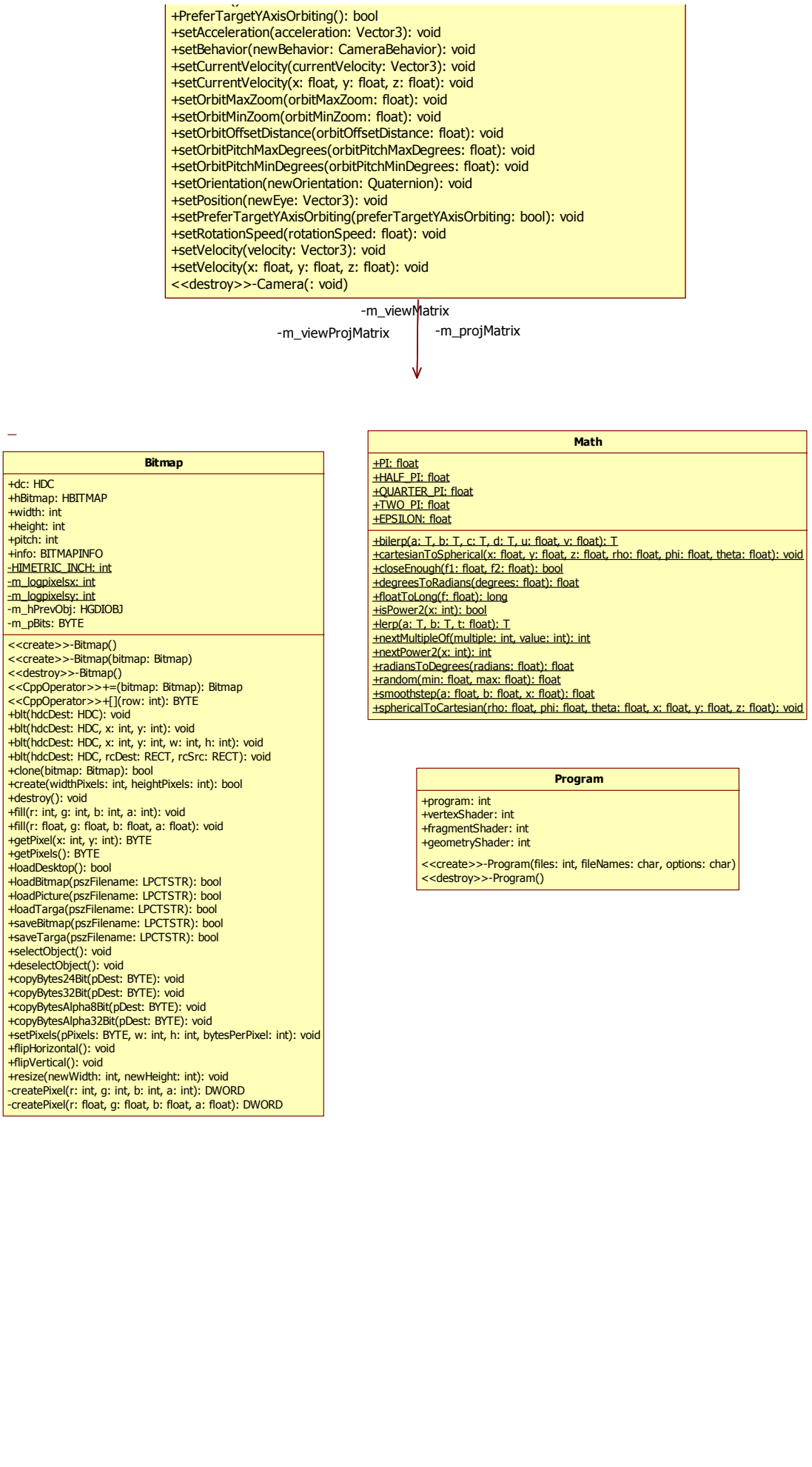

FractalBuilding
-numberOfBlocks: int
-LoadPattern(: Vector3): void
+Generate(: Vector3): void
<<create>>-FractalBuilding(: void)
+Draw(: void): void
<<destroy>>-FractalBuilding(: void)
-buildingBlocks
BuildingBlock
-polyhedrons: std::vector<Polyhedron*>
-numberOfPolyhedrons: int
<<create>>-BuildingBlock(: void)
+SetPosition(position: Vector3): void
+Generate(: BlockInformation): void
+Draw(: void): void
<<destroy>>-BuildingBlock(: void)
-position
Vector3
+x: float
+y: float
+z: float
<<CppFriend>>-*(lhs: float, rhs: Vector3): Vector3
<<CppFriend>>--(v: Vector3): Vector3
+cross(p: Vector3, q: Vector3): Vector3
+distance(pt1: Vector3, pt2: Vector3): float
+distanceSq(pt1: Vector3, pt2: Vector3): float
+dot(p: Vector3, q: Vector3): float
+lerp(p: Vector3, q: Vector3, t: float): Vector3
+orthogonalize(v1: Vector3, v2: Vector3): void
+orthogonalize(v1: Vector3, v2: Vector3, v3: Vector3): void
+proj(p: Vector3, q: Vector3): Vector3
+perp(p: Vector3, q: Vector3): Vector3
+reflect(i: Vector3, n: Vector3): Vector3
<<create>>-Vector3()
<<create>>-Vector3(x_: float, y_: float, z_: float)
<<destroy>>-Vector3()
<<CppOperator>>+==(rhs: Vector3): bool
<<CppOperator>>+!=(rhs: Vector3): bool
<<CppOperator>>+!(): float
<<CppOperator>>++=(rhs: Vector3): Vector3
<<CppOperator>>+-=(rhs: Vector3): Vector3
<<CppOperator>>+*=(scalar: float): Vector3
<<CppOperator>>+/=(scalar: float): Vector3
<<CppOperator>>+=(rhs: Vector3): Vector3
<<CppOperator>>++(rhs: Vector3): Vector3
<<CppOperator>>+-(rhs: Vector3): Vector3
<<CppOperator>>+*(scalar: float): Vector3
<<CppOperator>>+/(scalar: float): Vector3
+magnitude(): float
+magnitudeSq(): float
+inverse(): Vector3
+normalize(): void
+set(x_: float, y_: float, z_: float): void
Polyhedron
-color: Color
-position: Vector
-dimension: Dimension
-vbo: GLuint
-vboColores: GLuint
-vboNormals: GLuint
-vertices: GLfloat
-colors: GLfloat
-normals: GLfloat
-sizeofVertices: GLsizei
-InitVBOs(: void): void
<<create>>-Polyhedron(: void)
<<create>>-Polyhedron(color: Color, position: Vector, dimension: Dimension)
+CreatePolyhedron(): void
+Draw(: void): void
+SetPosition(position: Vector): void
+SetDimension(dimension: Dimension): void
+SetColor(color: Color): void
+SetMaterial(material: Material): void
<<destroy>>-Polyhedron(: void)
-material
Material
-diffuseColor: Color4
-ambientColor: Color4
-specularColor: Color4
-emissiveColor: Color4
-nameMaterial: string
-diffuse: GLfloat
-ambient: GLfloat
-specular: GLfloat
-emissive: GLfloat
<<create>>-Material(: void)
<<create>>-Material(: Color4, : Color4, : Color4, : Color4)
+SetDiffuseColor(diffuseColor: Color4): void
+SetAmbientColor(ambientColor: Color4): void
+SetSpecularColor(specularColor: Color4): void
+SetEmissiveColor(emissiveColor: Color4): void
+GetDiffuseColor(): Color4
+GetAmbientColor(): Color4
+GetSpecularColor(): Color4
+GetEmissiveColor(): Color4
+GetNameMaterial(): string
+SetMaterial(: void): void
<<CppOperator>>+=(: Material): Material
<<destroy>>-Material(: void)
-dimension
-material
-position
PolyhedronInformation
-color: Color
-texture: char
<<create>>-PolyhedronInformation(: void)
+SetPosition(position: Vector3): void
+SetDimension(dimension: Vector3): void
+SetColor(color: Color): void
+SetMaterial(material: Material): void
+GetPosition(): Vector3
+GetDimension(): Vector3
+GetColor(): Color
+GetMaterial(): Material
<<destroy>>-PolyhedronInformation(: void)


FractalTerrain
-vboTerrain: GLuint
-iboTerrain: GLuint
-totalVertices: int
-totalIndices: int
-nullTextures: GLuint
-maxAnisotrophy: int
-terrainShader: GLuint
-GenerateIndices(): bool
-GenerateVertices(): bool
-use16BitIndices(): bool
-GenerateTerrain(: void): bool
-GenerateUsingDiamondSquareFractal(: float): bool
-BindTexture(: GLuint, : GLuint): void
-CompileShader(type: GLenum, pszSource: GLchar, length: GLint): GLuint
-CreateNullTexture(width: int, height: int): GLuint
-LoadTexture(pszFilename: char): GLuint
-LoadTexture(pszFilename: char, magFilter: GLint, minFilter: GLint, wrapS: GLint, wrapT: GLint): GLuint
-LinkShaders(vertShader: GLuint, fragShader: GLuint): GLuint
-LoadShaderProgram(pszFilename: char, infoLog: std::string): GLuint
-ReadTextFile(pszFilename: char, buffer: std::string): void
-UpdateTerrainShaderParameters(): void
#TerrainCreate(: int, : int, : float): bool
#TerrainDestroy(): void
#TerrainDraw(): void
#TerrainUpdate(: Vector3): void
<<create>>-FractalTerrain(: void)
+GetHeighAt(: float, : float): float
+Create(: int, : int, : float): bool
+Destroy(): void
+Draw(: void): void
<<destroy>>-FractalTerrain(: void)
-heightMap
HeightMap
-size: int
-gridSpacing: int
-heightScale: float
-heights: std::vector<float>
-m_size: int
-m_gridSpacing: int
-m_heightScale: float
-m_heights: std::vector<float>
-Blur(: float): void
-HeightIndexAt(x: int, z: int): unsigned int
-Smooth(: void): void
<<create>>-HeightMap(: void)
+GetHeightScale(): float
+GetSize(): int
+GetGridSpacing(): int
+GetHeights(): float
+Create(: int, : int, : float): bool
+Destroy(: void): void
+GenerateDiamondSquareFractal(: float): void
+HeightAt(: float, : float): float
+HeightAtPixel(x: int, z: int): float
+NormalAt(x: float, z: float, n: Vector3): void
+NormalAtPixel(x: int, z: int, n: Vector3): void
<<destroy>>-HeightMap(: void)
<<create>>-HeightMap()
<<destroy>>-HeightMap()
+getHeightScale(): float
+getSize(): int
+getGridSpacing(): int
+getHeights(): float
+create(size: int, gridSpacing: int, scale: float): bool
+destroy(): void
+generateDiamondSquareFractal(roughness: float): void
+heightAt(x: float, z: float): float
+heightAtPixel(x: int, z: int): float
+normalAt(x: float, z: float, n: Vector3): void
+normalAtPixel(x: int, z: int, n: Vector3): void
-blur(amount: float): void
-heightIndexAt(x: int, z: int): unsigned int
-smooth(): void
-m_heightMap
Terrain
-m_vertexBuffer: GLuint
-m_indexBuffer: GLuint
-m_totalVertices: int
-m_totalIndices: int
<<create>>-Terrain()
<<destroy>>-Terrain()
+create(size: int, gridSpacing: int, scale: float): bool
+destroy(): void
+draw(): void
+generateUsingDiamondSquareFractal(roughness: float): bool
+update(cameraPos: Vector3): void
+getHeightMap(): HeightMap
+getHeightMap(): HeightMap
#terrainCreate(size: int, gridSpacing: int, scale: float): bool
#terrainDestroy(): void
#terrainDraw(): void
#terrainUpdate(cameraPos: Vector3): void
-generateIndices(): bool
-generateVertices(): bool
-use16BitIndices(): bool


FractalCloud
-r1: int
-r2: int
-r3: int
-cloudDensity: double
-frequency: double
-persistence: double
-id: unsigned int
-PerlinNoise2d(x: int, y: int): double
-Smooth(x: double, y: double): double
-Noise(x: int, y: int): double
-Interpolate(x: double, y: double, a: double): double
-GenerateCloud(: void): void
<<create>>-FractalCloud(: void)
+SetCloudDensity(cloudDensity: double): void
+SetFrequency(frequency: double): void
+SetPersistence(persistence: double): void
+UpdateClouds(: void): void
+SetPosition(position: Vector3): void
+Draw(: void): void
<<destroy>>-FractalCloud(: void)
-position
Vector3
+x: float
+y: float
+z: float
<<CppFriend>>-*(lhs: float, rhs: Vector3): Vector3
<<CppFriend>>--(v: Vector3): Vector3
+cross(p: Vector3, q: Vector3): Vector3
+distance(pt1: Vector3, pt2: Vector3): float
+distanceSq(pt1: Vector3, pt2: Vector3): float
+dot(p: Vector3, q: Vector3): float
+lerp(p: Vector3, q: Vector3, t: float): Vector3
+orthogonalize(v1: Vector3, v2: Vector3): void
+orthogonalize(v1: Vector3, v2: Vector3, v3: Vector3): void
+proj(p: Vector3, q: Vector3): Vector3
+perp(p: Vector3, q: Vector3): Vector3
+reflect(i: Vector3, n: Vector3): Vector3
<<create>>-Vector3()
<<create>>-Vector3(x_: float, y_: float, z_: float)
<<destroy>>-Vector3()
<<CppOperator>>+==(rhs: Vector3): bool
<<CppOperator>>+!=(rhs: Vector3): bool
<<CppOperator>>+!(): float
<<CppOperator>>++=(rhs: Vector3): Vector3
<<CppOperator>>+-=(rhs: Vector3): Vector3
<<CppOperator>>+*=(scalar: float): Vector3
<<CppOperator>>+/=(scalar: float): Vector3
<<CppOperator>>+=(rhs: Vector3): Vector3
<<CppOperator>>++(rhs: Vector3): Vector3
<<CppOperator>>+-(rhs: Vector3): Vector3
<<CppOperator>>+*(scalar: float): Vector3
<<CppOperator>>+/(scalar: float): Vector3
+magnitude(): float
+magnitudeSq(): float
+inverse(): Vector3
+normalize(): void
+set(x_: float, y_: float, z_: float): void

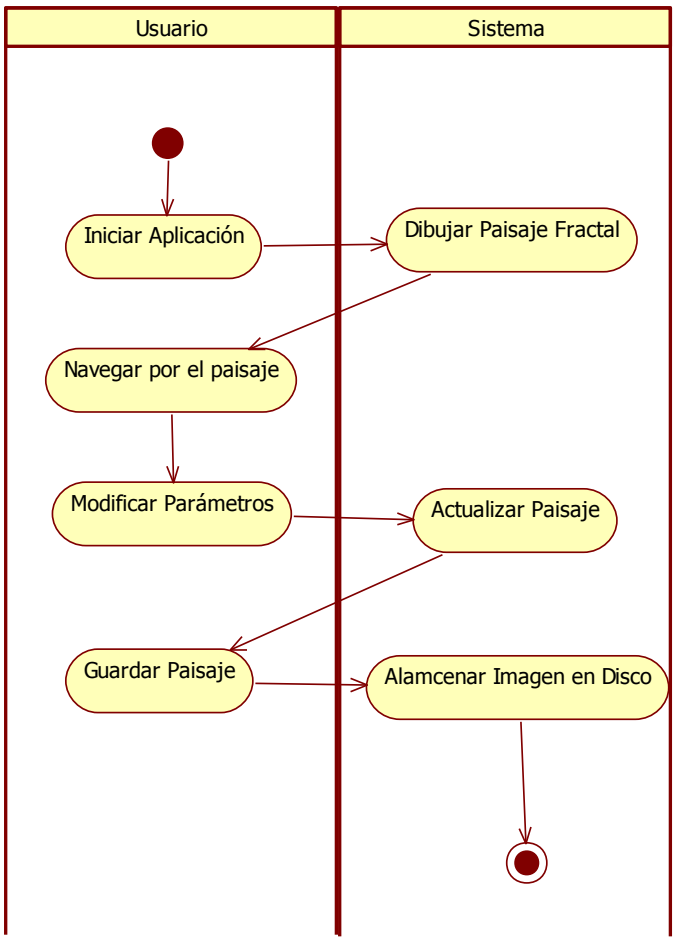

-building
Scene
-sky: Sky
-data: FractalsData
-PerformCameraCollisionDetection(): void
-SetupScene(: void): void
<<create>>-Scene(: void)
+GetCamera(: void): Camera
+UpdateData(data: FractalsData): void
+Update(: void): void
+Draw(: void): void
<<destroy>>-Scene(: void)
-cloud
-camera
-cameraBoundsMin
-cameraBoundsMax
-terrain


Camera
-position: Vector
-target: Vector
-upVector: Vector
-fov: float
-ratio: float
-nearDistance: float
-farDisntance: float
-DEFAULT_ROTATION_SPEED: float
-DEFAULT_FOVX: float
-DEFAULT_ZNEAR: float
-DEFAULT_ZFAR: float
-DEFAULT_ORBIT_MIN_ZOOM: float
-DEFAULT_ORBIT_MAX_ZOOM: float
-DEFAULT_ORBIT_OFFSET_DISTANCE: float
-m_behavior: CameraBehavior
-m_preferTargetYAxisOrbiting: bool
-m_accumPitchDegrees: float
-m_savedAccumPitchDegrees: float
-m_rotationSpeed: float
-m_fovx: float
-m_aspectRatio: float
-m_znear: float
-m_zfar: float
-m_orbitMinZoom: float
-m_orbitMaxZoom: float
-m_orbitOffsetDistance: float
-m_firstPersonYOffset: float
-Initialize(: void): void
-RotateFirstPerson(headingDegrees: float, pitchDegrees: float): void
-RotateFlight(headingDegrees: float, pitchDegrees: float, rollDegrees: float): void
-RotateOrbit(headingDegrees: float, pitchDegrees: float, rollDegrees: float): void
-UpdateVelocity(direction: Vector3, elapsedTimeSec: float): void
-UpdateViewMatrix(): void
<<create>>-Camera(: void)
<<create>>-Camera(position: Vector, target: Vector, upVector: Vector)
+Move(direction: Direction, quantity: float): void
+LookAt(target: Vector3): void
+LookAt(eye: Vector3, target: Vector3, up: Vector3): void
+Move(dx: float, dy: float, dz: float): void
+Move(direction: Vector3, amount: Vector3): void
+Perspective(fovx: float, aspect: float, znear: float, zfar: float): void
+Rotate(headingDegrees: float, pitchDegrees: float, rollDegrees: float): void
+RotateSmoothly(headingDegrees: float, pitchDegrees: float, rollDegrees: float): void
+UndoRoll(): void
+UpdatePosition(direction: Vector3, elapsedTimeSec: float): void
+Zoom(zoom: float, minZoom: float, maxZoom: float): void
+GetAcceleration(): Vector3
+GetBehavior(): CameraBehavior
+GetCurrentVelocity(): Vector3
+GetPosition(): Vector3
+GetOrbitMinZoom(): float
+GetOrbitMaxZoom(): float
+GetOrbitOffsetDistance(): float
+GetOrbitPitchMaxDegrees(): float
+GetOrbitPitchMinDegrees(): float
+GetOrientation(): Quaternion
+GetRotationSpeed(): float
+GetProjectionMatrix(): Matrix4
+GetVelocity(): Vector3
+GetViewDirection(): Vector3
+GetViewMatrix(): Matrix4
+GetViewProjectionMatrix(): Matrix4
+GetXAxis(): Vector3
+GetYAxis(): Vector3
+GetZAxis(): Vector3
-m_savedOrientation
-m_orientation
-m_currentVelocity
-m_acceleration
-m_viewDir
-m_velocity
-m_zAxis
-m_yAxis
-m_xAxis
-m_targetYAxis
-m_target
-m_savedEye
-m_eye
-WORLD_ZAXIS
-WORLD_YAXIS
-WORLD_XAXIS

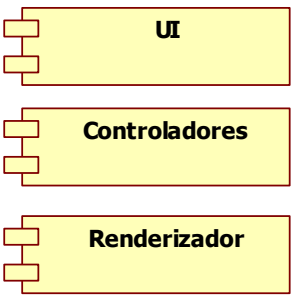

+PreferTargetYAxisOrbiting(): bool
+setAcceleration(acceleration: Vector3): void
+setBehavior(newBehavior: CameraBehavior): void
+setCurrentVelocity(currentVelocity: Vector3): void
+setCurrentVelocity(x: float, y: float, z: float): void
+setOrbitMaxZoom(orbitMaxZoom: float): void
+setOrbitMinZoom(orbitMinZoom: float): void
+setOrbitOffsetDistance(orbitOffsetDistance: float): void
+setOrbitPitchMaxDegrees(orbitPitchMaxDegrees: float): void
+setOrbitPitchMinDegrees(orbitPitchMinDegrees: float): void
+setOrientation(newOrientation: Quaternion): void
+setPosition(newEye: Vector3): void
+setPreferTargetYAxisOrbiting(preferTargetYAxisOrbiting: bool): void
+setRotationSpeed(rotationSpeed: float): void
+setVelocity(velocity: Vector3): void
+setVelocity(x: float, y: float, z: float): void
<<destroy>>-Camera(: void)
-m_viewMatrix
-m_viewProjMatrix
-m_projMatrix
Bitmap
+dc: HDC
+hBitmap: HBITMAP
+width: int
+height: int
+pitch: int
+info: BITMAPINFO
-HIMETRIC_INCH: int
-m_logpixelsx: int
-m_logpixelsy: int
-m_hPrevObj: HGDIOBJ
-m_pBits: BYTE
<<create>>-Bitmap()
<<create>>-Bitmap(bitmap: Bitmap)
<<destroy>>-Bitmap()
<<CppOperator>>+=(bitmap: Bitmap): Bitmap
<<CppOperator>>+[](row: int): BYTE
+blt(hdcDest: HDC): void
+blt(hdcDest: HDC, x: int, y: int): void
+blt(hdcDest: HDC, x: int, y: int, w: int, h: int): void
+blt(hdcDest: HDC, rcDest: RECT, rcSrc: RECT): void
+clone(bitmap: Bitmap): bool
+create(widthPixels: int, heightPixels: int): bool
+destroy(): void
+fill(r: int, g: int, b: int, a: int): void
+fill(r: float, g: float, b: float, a: float): void
+getPixel(x: int, y: int): BYTE
+getPixels(): BYTE
+loadDesktop(): bool
+loadBitmap(pszFilename: LPCTSTR): bool
+loadPicture(pszFilename: LPCTSTR): bool
+loadTarga(pszFilename: LPCTSTR): bool
+saveBitmap(pszFilename: LPCTSTR): bool
+saveTarga(pszFilename: LPCTSTR): bool
+selectObject(): void
+deselectObject(): void
+copyBytes24Bit(pDest: BYTE): void
+copyBytes32Bit(pDest: BYTE): void
+copyBytesAlpha8Bit(pDest: BYTE): void
+copyBytesAlpha32Bit(pDest: BYTE): void
+setPixels(pPixels: BYTE, w: int, h: int, bytesPerPixel: int): void
+flipHorizontal(): void
+flipVertical(): void
+resize(newWidth: int, newHeight: int): void
-createPixel(r: int, g: int, b: int, a: int): DWORD
-createPixel(r: float, g: float, b: float, a: float): DWORD
Math
+PI: float
+HALF_PI: float
+QUARTER_PI: float
+TWO_PI: float
+EPSILON: float
+bilerp(a: T, b: T, c: T, d: T, u: float, v: float): T
+cartesianToSpherical(x: float, y: float, z: float, rho: float, phi: float, theta: float): void
+closeEnough(f1: float, f2: float): bool
+degreesToRadians(degrees: float): float
+floatToLong(f: float): long
+isPower2(x: int): bool
+lerp(a: T, b: T, t: float): T
+nextMultipleOf(multiple: int, value: int): int
+nextPower2(x: int): int
+radiansToDegrees(radians: float): float
+random(min: float, max: float): float
+smoothstep(a: float, b: float, x: float): float
+sphericalToCartesian(rho: float, phi: float, theta: float, x: float, y: float, z: float): void
Program
+program: int
+vertexShader: int
+fragmentShader: int
+geometryShader: int
<<create>>-Program(files: int, fileNames: char, options: char)
<<destroy>>-Program()

4.2.3 Diseño físico

4.2.3.1 Diagramas de actividades



4.2.3.2 Diagrama de componentes



4.3 DESARROLLO

4.3.1 Nomenclatura y estándares para el desarrollo

Este documento refleja los estándares de codificación del lenguaje. Este documento será aplicado a todos los proyectos o librerías de la aplicación.

4.3.1.1 Convenciones del uso de Mayúsculas y minúsculas

Se hará uso de las siguientes:

- Estilo Pascal (PascalCase). La primera letra del identificador y la primera letra de las siguientes palabras concatenadas están en mayúsculas. El estilo de mayúsculas y minúsculas Pascal se puede utilizar en identificadores de tres o más caracteres, por ejemplo:

  ImageSprite

- Estilo camelCase. La primera letra del identificador está en minúscula y la primera letra de las siguientes palabras concatenadas en mayúscula, por ejemplo:

  imageSprite

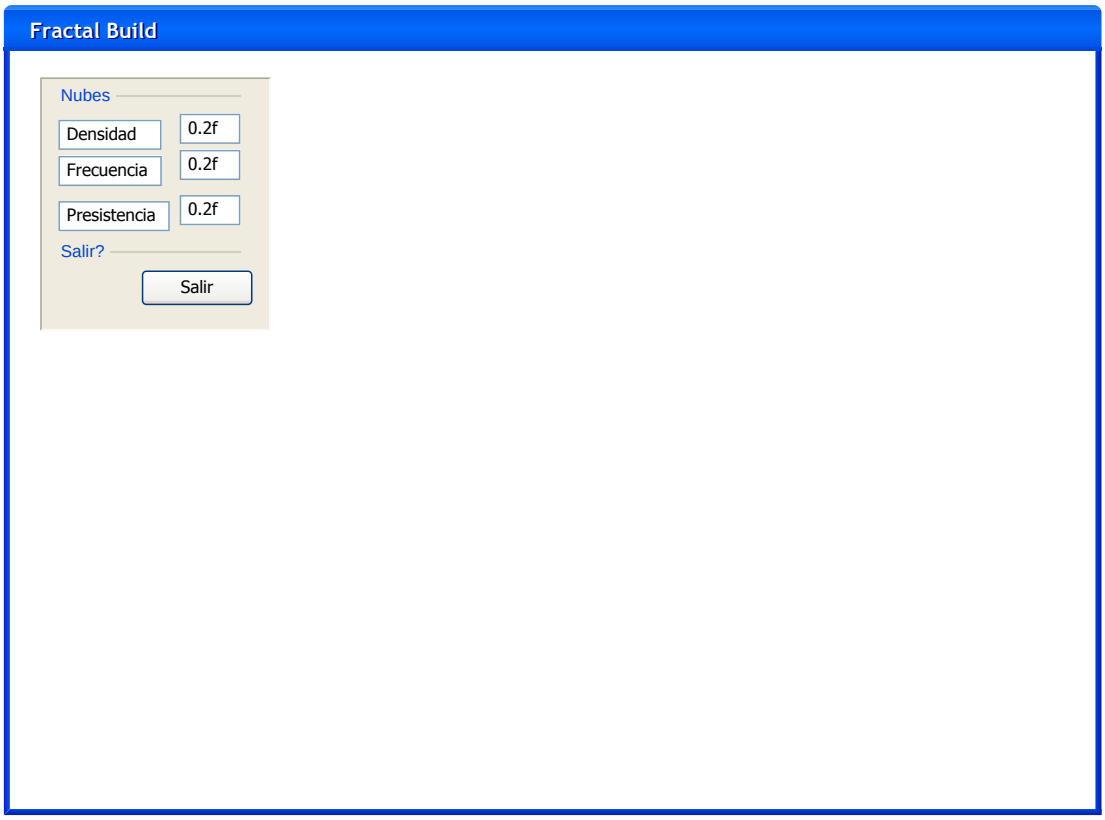
4.3.1.2 Convenciones de nombres

En la siguiente tabla tenemos las convenciones usadas en el desarrollo del proyecto:

TIPO	CONVENCIÓN DE USO DE MAYÚSCULAS
Clase	PascalCase
Constante	PascalCase
Método	PascalCase
Solución	PascalCase
Proyecto	PascalCase
Variable Local	camelCase
Parámetro	camelCase

4.3.2 Capa de presentación

4.3.2.1 Diseño de interfaces de usuario



4.3.3 Capa de Lógica

4.3.3.1 Implementación de los algoritmos de fractales

- Algoritmo para la generación del edificio:

```
for(int i = 0; i<NUMBER_OF_BLOCKS; i++)
 {
      BuildingBlock *buildingBlock = new BuildingBlock();
      if(i%4==0)
            buildingBlock->Generate(block1);
      else
      {
            if((i+1)%4==0)
            {
                  if((i+1)%8==0)
                        buildingBlock->Generate(block3);
                  else
                        buildingBlock->Generate(block21);
            }
```

```
            else
                buildingBlock->Generate(block2);
        }

        buildingBlock->SetPosition(Vector3(position.x + ((i%8) *
3.0),position.y + (i/8) * 3,position.z));

        buildingBlocks[i] = buildingBlock;
    }
```

- Implementación del Algoritmo PerlinNoise para la generación de nubes fractales

```
double FractalCloud::PerlinNoise2d(int x, int y)
{
    double total = 0.0;
    double freq = this->frequency;    // AJUSTABLE
    double persistence = .65;         // AJUSTABLE
    double octaves = 8;               // AJUSTABLE
    double amplitude = 1;             // AJUSTABLE

    double cloudCoverage = 0;          // AJUSTABLE
    double cloudDens = this->cloudDensity; // AJUSTABLE

    for(int lcv = 0; lcv < octaves; lcv++)
    {
        total = total + Smooth(x * freq, y * freq) * amplitude;
        freq = freq * 2;
        amplitude = amplitude * persistence;
    }

    total = (total + cloudCoverage) * cloudDens;

    if(total < 0) total = 0.0;
    if(total > 1) total = 1.0;

    return total;
}
```

- Implementación del algoritmo DiamondSquare para la generación de terrenos fractales

```
void HeightMap::GenerateDiamondSquareFractal(float roughness)
{
    // Generates a fractal height field using the diamond-square
(midpoint
    // displacement) algorithm. Note that only square height
fields work with
    // this algorithm.
    //
```

```
    // Based on article and associated code:
    // "Fractal Terrain Generation - Midpoint Displacement" by
Jason Shankel
    // (Game Programming Gems I, pp.503-507).

    srand(static_cast<unsigned int>(time(0)));

    std::fill(heights.begin(), heights.end(), 0.0f);

    int p1, p2, p3, p4, mid;
    float dH = size * 0.5f;
    float dHFactor = powf(2.0f, -roughness);
    float minH = 0.0f, maxH = 0.0f;

    for (int w = size; w > 0; dH *= dHFactor, w /= 2)
    {
        // Diamond Step.
        for (int z = 0; z < size; z += w)
        {
            for (int x = 0; x < size; x += w)
            {
                p1 = HeightIndexAt(x, z);
                p2 = HeightIndexAt(x + w, z);
                p3 = HeightIndexAt(x + w, z + w);
                p4 = HeightIndexAt(x, z + w);
                mid = HeightIndexAt(x + w / 2, z + w / 2);

                heights[mid] = Math::random(-dH, dH) +
(heights[p1] + heights[p2] + heights[p3] + heights[p4]) * 0.25f;

                minH = min(minH, heights[mid]);
                maxH = max(maxH, heights[mid]);
            }
        }

        // Square step.
        for (int z = 0; z < size; z += w)
        {
            for (int x = 0; x < size; x += w)
            {
                p1 = HeightIndexAt(x, z);
                p2 = HeightIndexAt(x + w, z);
                p3 = HeightIndexAt(x + w / 2, z - w / 2);
                p4 = HeightIndexAt(x + w / 2, z + w / 2);
                mid = HeightIndexAt(x + w / 2, z);

                heights[mid] = Math::random(-dH, dH) +
(heights[p1] + heights[p2] + heights[p3] + heights[p4]) * 0.25f;

                minH = min(minH, heights[mid]);
                maxH = max(maxH, heights[mid]);

                p1 = HeightIndexAt(x, z);
```

```
                p2 = HeightIndexAt(x, z + w);
                p3 = HeightIndexAt(x + w / 2, z + w / 2);
                p3 = HeightIndexAt(x - w / 2, z + w / 2);
                mid = HeightIndexAt(x, z + w / 2);

                heights[mid] = Math::random(-dH, dH) +
(heights[p1] + heights[p2] + heights[p3] + heights[p4]) * 0.25f;

                minH = min(minH, heights[mid]);
                maxH = max(maxH, heights[mid]);
            }
        }
    }

    Smooth();

    // Normalize height field so altitudes fall into range
[0,255].
    for (int i = 0; i < size * size; ++i)
        heights[i] = 255.0f * (heights[i] - minH) / (maxH -
minH);
}

float HeightMap::HeightAt(float x, float z) const
{
    // Given a (x, z) position on the rendered height map this
method
    // calculates the exact height of the height map at that (x,
z)
    // position using bilinear interpolation.

    x /= static_cast<float>(gridSpacing);
    z /= static_cast<float>(gridSpacing);

    assert(x >= 0.0f && x < float(size));
    assert(z >= 0.0f && z < float(size));

    long ix = Math::floatToLong(x);
    long iz = Math::floatToLong(z);
    float topLeft = heights[HeightIndexAt(ix, iz)] *
heightScale;
    float topRight = heights[HeightIndexAt(ix + 1, iz)] *
heightScale;
    float bottomLeft = heights[HeightIndexAt(ix, iz + 1)] *
heightScale;
    float bottomRight = heights[HeightIndexAt(ix + 1, iz + 1)] *
heightScale;
    float percentX = x - static_cast<float>(ix);
    float percentZ = z - static_cast<float>(iz);

    return Math::bilerp(topLeft, topRight, bottomLeft,
bottomRight, percentX, percentZ);
}
```

CONCLUSIONES

- Para usar la GPU para ejecutar aplicaciones, es necesario tomar en cuenta algunos factores, como el hecho de que el algoritmo a implementar sea , preferiblemente, paralelizable.
- Aprender a usar estas herramientas será (si no lo es ya) un requisito indispensable para cualquier programador, no exclusivamente para los desarrolladores de software de sistemas.
- El proceso para aprender estas herramientas, es lento y mientras tanto el hardware de que disponemos en nuestro escritorio estará muy infrautilizado, ya que muy pocas aplicaciones aprovechan su potencia.
- El sistema operativo utiliza los múltiples núcleos de los actuales microprocesadores para repartir la carga de trabajo, pero apenas hace uso de la GPU. De hecho las GPU, salvo en el caso de los juegos y algunas aplicaciones específicas de gráficos/vídeo, son el recurso más desaprovechado. La última generación de de navegadores, no obstante, hace uso de esa potencia a través de la aceleración por hardware de la composición de páginas.
- Aprender a usar un estándar nuevo, como lo es OpenCL, requiere de mucho tiempo, pero se puede convertir en una gran herramienta por los beneficios que puede dar al usarse para acelerar aplicaciones de propósito general.
- El paralelismo, cuando se trabaja en el procesador gráfico, si se lo hacía de la forma tradicional requieren el rediseño de los algoritmos, de tal forma que requería mapear los datos y representarlos como texturas; el uso de API's como OpenCL constituye un gran avance en éste ámbito, pues ya no se requiere el mapeo
- La programación sobre procesadores gráficos se ha convertido en una tendencia, ya que permite aprovechar este hardware para otras tareas.

RECOMENDACIONES

- Esta tesis sirva de base para el estudio y utilización de las arquitecturas heterogéneas(CPU+GPU), es decir que la GPU sea utilizada como co-procesador que sirve de apoyo en las tareas más exigentes y que no necesariamente se traten de aplicaciones gráficas o de videojuegos.
- Se recomienda un estudio en el cual se analice las dos tecnologías(CPU y GPU) desde el punto de vista de la escalabilidad para determinar cuán escalable puede ser un sistema al desarrollarse sobre una u otra plataforma.

RESUMEN

Análisis de la programación concurrente sobre CPU y GPU en el desarrollo de Fractal Build, caso práctico Escuela de Ingeniería en Sistemas de la Escuela Superior Politécnica de Chimborazo.

Se hizo uso del método analítico con el cual se analizó los datos obtenidos tras las mediciones realizadas.

Para el análisis de la programación concurrente sobre las tecnologías GPU y CPU, se utilizó varias las siguientes herramientas: gDebuuger, FRAPS y el Monitor del sistema de Windows, con las cuales se pudo realizar las mediciones sobre el prototipo. Para el desarrollo de la aplicación y del prototipo se usó C++, OpenGL y Visual Studio como IDE de desarrollo.

Los resultados obtenidos fueron: en tiempo de ejecución: GPU ejecutó el algoritmo en 20 segundos frente a los 45 segundos que se tardó en CPU; frames por segundo: 8 fps en CPU y 18 en GPU; uso de procesador: 67% en CPU frente a 10% en GPU; uso de memoria: 48 MB en GPU y 86 MB en CPU. Las mediciones de los parámetros se las hicieron sobre un prototipo construido para el efecto, con lo cual se obtuvo que la tecnología GPU es la más adecuada para implementar la aplicación Fractal Build, siendo superior en cada uno de los parámetros analizados, obteniendo 75 puntos en la sumatoria, sobre 14.5 puntos que obtuvo la tecnología CPU.

Por lo analizado se concluye que el uso de la tecnología GPU es la más adecuada debido a las ventajas que brinda al momento de ejecutar una aplicación gráfica.

Se recomienda continuar con el estudio de las implementaciones sobre GPU ya que el uso de la tecnología GPU, como procesador de propósito general se ha convertido en una tendencia en los últimos años, debido a sus múltiples beneficios.

SUMMARY

Analysis of concurrent programming on CPU and GPU in the development of Fractal Build. Case study of the School of Engineering Systems of the Superior Polytechnic School of Chimborazo.

Analytical method was used by means of which it was analyzed the obtained data after the measurements.

For the analysis of concurrent programming on the GPU and CPU technologies was used the following software tools: gDEBugger, FRAPS, and monitor Windows systems with which measurements could be performed on the prototype. For the development of the application and the prototype was used C++, OpenGL and Visual Studio as development IDE.

The results obtained were as follows: at runtime, GPU run the algorithm in 20 seconds against 45 seconds to lasted the CPU; frames per second: 8fps in CPU and 18 GPU; CPU usage: 67% CPU against 10% in GPU; and memory usage: 48 MB in GPU and 86 MB in CPU. Measurements of the parameters are made on a prototype built for this purpose, therefore, GPU technology is most appropriate to implement the Build Fractal application, being higher in each of the analyzed parameters, getting 75 points in the sum over 14.5 points that was gotten by CPU technology.

From the analysis, it was concluded that use of GPU technology is the most suitable because of the advantages offered by the time you run a graphical application.

It is recommended continuing the study of implementations on GPU since the GPU technology ,as a general purpose processor, has become a trend in recent years due to its multiple benefits.

GLOSARIO DE TERMINOS

ESCALABILIDAD. La escalabilidad es la capacidad de mejorar recursos para ofrecer una mejora (idealmente) lineal en la capacidad de servicio. La característica clave de una aplicación es que la carga adicional sólo requiere recursos adicionales en lugar de una modificación extensiva de la aplicación en sí.

RENDERIZAR. Renderizar es un término usado en informática para referirse al proceso de generar una imagen desde un modelo. Este término técnico es utilizado por los animadores o productores audiovisuales y en programas de diseño en 3D.

FOTOGRAMAS POR SEGUNDO. Las imágenes por segundo ó en inglés frames per second (FPS) es la medida de la frecuencia a la cual un reproductor de imágenes genera distintos fotogramas o frames.

THREAD. En sistemas operativos, es un hilo de ejecución o hebra o subproceso y es la unidad de procesamiento más pequeña que puede ser planificada por un sistema operativo.

KERNEL. Es utilizado en la programación bajo OpenCL para referirse a una función que será ejecutada en un procesador que soporte dicho estándar.

BIBLIOGRAFÍA

1.- KHRONOS, G., The OpenCL Specification., Khronos Group., 2009., Pp. 19-28.

2.- MCREYNOLDS, T., Advanced Graphic Programming Using OpenGL., 2da. Ed.., San Francisco CA-Estados Unidos., Morgan Kaufmann Publishers., 2005., Pp. 35-43,600-630.

BIBLIOGRAFÍA DE INTERNET

3.- ADVANCED PROGRAMMING (GPGPU)

http://www.gpgpu.org

2011-01-15

4.- ARQUITECTURA DE COMPUTADORES II. GPU

http://www.assembla.com/spaces/documents/download/Trabajo.pdf

2011-01-18

5.- ESTADO ACTUAL DE LOS ALGORITMOS DE RECONOCIMIENTO DE ROSTRO USANDO TECNOLOGÍA GPU

http://www.cenatav.co.cu/doc/RTecnicos/RT%20SerieAzul_031web.pdf

201-12-26

6.- FRAME RATE

http://en.wikipedia.org/wiki/Frame_rate

2011-10-10

7.- GEFORCE GTX 690

http://www.guru3d.com/article/geforce-gtx-690-review/2

2012-04-10

8.- GPGPU

http://es.wikipedia.org/wiki/GPGPU

2011-03-14

9.- GRAPHICS REMEDY, GDEBUGGER

http://www.gremedy.com/

2011-05-10

10.- GUÍA DIDÁCTICA PROGRAMACIÓN CONCURRENTE

http://www.uned.es/infor-3-programacion-concurrente/Guia_Didactica_Programac ion_Concurrente.pdf

2011-01-08

11.- NVIDIA

http://www.nvidia.com/

2011-02-01

12.- OPENCL OVERVIEW

http://www.khronos.org/developers/library/overview/opencl_overview.pdf

2011-03-10

13.- PAISAJE FRACTAL

http://es.wikipedia.org/wiki/Paisaje_fractal

2011-05-10

14.- PROGRAMACIÓN CONCURRENTE

http://casidiablo.net/wordpress/wp-content/uploads/2008/01/programacion-concurrente.pdf

2011-01-08

15.- PROGRAMACIÓN CONCURRENTE. CONCEPTOS BÁSICOS

http://www.oocities.com/etsetb/4a_pc_intro.pdf

2011-01-08

16.- PROGRAMACIÓN PARALELA

http://rua.ua.es/dspace/bitstream/10045/10020/4/Zamora%20G%C3%B3mez,%20Antonio_3.pdf

2011-03-14

17.- PROGRAMACIÓN Y PARALELISMO (CPU VS GPU)

http://fcharte.com/Default.asp?noticias=2&a=2010&m=9&d=22

2011-01-09

18.- PROGRAMACIÓN Y PARALELISMO (THREADS, GPGPU, MPI)

http://fcharte.com/Default.asp?noticias=2&a=2010&m=9&d=14

2011-01-29

19.- STREAM TECHNOLOGIES

http://www.amd.com/us/products/technologies/stream-technology/Pages/stream-technology.aspx

2011-05-30

20.- UNIDAD CENTRAL DE PROCESAMIENTO

http://es.wikipedia.org/wiki/Unidad_central_de_procesamiento

2011-03-10

21.- UNIDAD DE PROCESO GRÁFICO

http://es.wikipedia.org/wiki/Unidad_de_Proceso_Gráfico.htm

2010-11-20

ANEXOS

ANEXO 1: Archivo de Profiling de gDebugger de la ejecución del prototipo sobre CPU

//
// This File contains OpenGL Profiling Data information
// Project name: C.MandelbrotOpenCL
// Generation date: Monday_11_June_2012
// Generation time: 17:23:06
//
// Generated by gDEBugger - an OpenGL Debugger and Profiler
// www.gremedy.com
//
Time (sec),Frames/sec: GL Context 1,OGL calls/frame: GL Context 1,Frames/sec: CL Context 1 Queue 0,Available Physical Memory
0.00,0,0,0,6420742144,0
0.20,0,0,0,6420680704,0
0.40,0,0,0,6420385792,11
0.60,0,0,0,6417448960,12
0.80,0,0,0,6400237568,23
1.00,0,0,0,6397886464,11
1.20,0,0,0,6391173120,15
1.40,0,0,0,6389338112,16
1.60,0,0,0,6380937216,16
1.80,0,0,0,6372511744,14
2.00,0,0,0,6359109632,40
2.20,0,0,0,6359027712,40
2.40,0,0,0,6359027712,0
2.60,0,0,0,6359027712,43
2.80,0,0,0,6359027712,41
3.00,0,0,0,6357311488,0
3.20,0,0,2,6330040320,27
3.40,0,0,44,6329122816,48
3.60,0,0,30,6328139776,68
3.80,0,0,20,6328008704,94
4.00,0,0,13,6327881728,95
4.20,0,0,9,6327881728,100
4.40,0,0,7,6327816192,100
4.60,0,0,6,6327721984,96
4.80,0,0,5,6327328768,100
5.00,0,0,5,6327197696,95
5.20,0,0,4,6327132160,98
5.40,0,0,4,6326935552,100
5.60,0,0,4,6326738944,100
5.80,0,0,3,6326079488,97
6.00,0,0,3,6325878784,95
6.20,0,0,3,6325424128,100
6.40,0,0,3,6325293056,100
6.60,0,0,3,6324174848,100
6.80,0,0,3,6324183040,100
7.00,0,0,3,6324125696,100

7.20,0,0,3,6324133888,100
7.40,0,0,3,6324137984,98
7.60,0,0,3,6321942528,93
7.80,0,0,3,6321979392,100
8.00,0,0,3,6321950720,100
8.20,0,0,3,6321979392,100
8.40,0,0,3,6321954816,100
8.60,0,0,3,6321987584,100
8.80,0,0,3,6321963008,100
9.00,0,0,3,6321999872,100
9.20,0,0,3,6321963008,95
9.40,0,0,3,6322098176,100
9.60,0,0,2,6321979392,100
9.80,0,0,2,6322106368,100
10.00,0,0,2,6322028544,100
10.20,0,0,2,6322061312,100
10.40,0,0,2,6322020352,94
10.60,0,0,2,6322049024,100
10.80,0,0,2,6322028544,95
11.00,0,0,2,6322692096,100
11.20,0,0,2,6322638848,95
11.40,0,0,2,6322663424,100
11.60,0,0,2,6322638848,100
11.80,0,0,2,6322769920,100
12.00,0,0,2,6322638848,94
12.20,0,0,2,6322671616,100
12.40,0,0,2,6322642944,94
12.60,0,0,2,6322679808,100
12.80,0,0,2,6322712576,100
13.00,0,0,2,6322647040,94
13.20,0,0,2,6322679808,100
13.40,0,0,2,6322642944,94
13.60,0,0,2,6322679808,100
13.80,0,0,2,6322647040,96
14.00,0,0,2,6322782208,100
14.20,0,0,2,6322647040,94
14.40,0,0,2,6322679808,100
14.60,0,0,2,6322647040,94
14.80,0,0,2,6322683904,100
15.00,0,0,2,6322638848,94
15.20,0,0,2,6322671616,100
15.40,0,0,3,6322647040,95
15.60,0,0,3,6322688000,100
15.80,0,0,3,6322647040,95
16.00,0,0,3,6322782208,100
16.20,0,0,3,6322651136,94
16.40,0,0,3,6322647040,95

16.60,0,0,3,6322688000,100
16.80,0,0,3,6322647040,94
17.00,0,0,4,6322647040,95
17.20,0,0,4,6322647040,95
17.40,0,0,4,6322679808,100
17.60,0,0,5,6322659328,89
17.80,0,0,9,6322655232,90
18.00,0,0,20,6322638848,76
18.20,0,0,24,6322647040,81
18.40,0,0,24,6322753536,76
18.60,0,0,24,6322761728,82
18.80,0,0,24,6322737152,77
19.00,0,0,24,6322753536,75
19.20,0,0,24,6322737152,67
19.40,0,0,23,6322745344,74
19.60,0,0,22,6322745344,75
19.80,0,0,21,6322753536,65
20.00,0,0,20,6322737152,81
20.20,0,0,19,6322757632,82
20.40,0,0,18,6322765824,79
20.60,0,0,17,6322778112,81
20.80,0,0,16,6322782208,80
21.00,0,0,16,6322774016,88
21.20,0,0,15,6322782208,79
21.40,0,0,15,6322765824,79
21.60,0,0,15,6322786304,81
21.80,0,0,15,6322782208,86
22.00,0,0,14,6322790400,83
22.20,0,0,14,6322782208,87
22.40,0,0,14,6322782208,87
22.60,0,0,14,6322802688,87
22.80,0,0,14,6322765824,82
23.00,0,0,14,6322765824,82
23.20,0,0,13,6322774016,87
23.40,0,0,13,6322765824,100
23.60,0,0,12,6322778112,87
23.80,0,0,12,6322802688,81
24.00,0,0,12,6322790400,87
24.20,0,0,12,6322917376,81
24.40,0,0,11,6322991104,86
24.60,0,0,11,6323040256,87
24.80,0,0,11,6323003392,93
25.00,0,0,11,6323011584,82
25.20,0,0,11,6323068928,85
25.40,0,0,11,6323089408,87
25.60,0,0,11,6323089408,87
25.80,0,0,11,6323081216,86

26.00,0,0,11,6323101696,87
26.20,0,0,11,6323081216,88
26.40,0,0,11,6323089408,86
26.60,0,0,11,6323089408,88
26.80,0,0,11,6323089408,86
27.00,0,0,11,6323089408,81
27.20,0,0,11,6323081216,86
27.40,0,0,11,6323089408,86
27.60,0,0,11,6323089408,87
27.80,0,0,11,6323089408,87
28.00,0,0,11,6323081216,87
28.20,0,0,10,6323073024,89
28.40,0,0,10,6323101696,81
28.60,0,0,10,6323089408,87
28.80,0,0,10,6323081216,87
29.00,0,0,10,6323073024,94
29.20,0,0,10,6323073024,87
29.40,0,0,10,6323085312,86
29.60,0,0,10,6323073024,88
29.80,0,0,10,6323081216,87
30.00,0,0,10,6323089408,93
30.20,0,0,10,6323081216,86
30.40,0,0,10,6323089408,87
30.60,0,0,10,6323089408,88
30.80,0,0,10,6323081216,93
31.00,0,0,10,6323089408,87
31.20,0,0,10,6323163136,87
31.40,0,0,10,6323163136,86
31.60,0,0,10,6323208192,94
31.80,0,0,9,6330191872,86
32.00,0,0,9,6329012224,86
32.20,0,0,9,6329139200,93
32.40,0,0,9,6329208832,88
32.60,0,0,9,6329221120,87
32.80,0,0,9,6329233408,87
33.00,0,0,9,6329311232,86
33.20,0,0,9,6329311232,86
33.40,0,0,9,6329319424,93
33.60,0,0,9,6329344000,93
33.80,0,0,9,6329352192,86
34.00,0,0,9,6329352192,89
34.20,0,0,8,6329344000,91
34.40,0,0,8,6329344000,89
34.60,0,0,8,6329511936,76
34.80,0,0,8,6329511936,67
35.00,0,0,8,6329520128,88
35.20,0,0,7,6329528320,94

35.40,0,0,8,6329520128,94
35.60,0,0,8,6329528320,88
35.80,0,0,8,6329520128,93
36.00,0,0,8,6329532416,88
36.20,0,0,8,6329528320,87
36.40,0,0,7,6329536512,93
36.60,0,0,8,6329528320,87
36.80,0,0,7,6329520128,93
37.00,0,0,7,6329528320,87
37.20,0,0,7,6329520128,94
37.40,0,0,7,6329520128,86
37.60,0,0,7,6329520128,93
37.80,0,0,7,6329528320,86
38.00,0,0,7,6329520128,93
38.20,0,0,7,6329528320,94
38.40,0,0,7,6329569280,93
38.60,0,0,7,6329528320,87
38.80,0,0,7,6329520128,100
39.00,0,0,7,6329544704,95
39.20,0,0,7,6329536512,92
39.40,0,0,7,6329536512,100
39.60,0,0,7,6329536512,94
39.80,0,0,7,6329544704,100
40.00,0,0,7,6329659392,100
40.20,0,0,7,6329651200,100
40.40,0,0,7,6329659392,86
40.60,0,0,7,6329651200,93
40.80,0,0,7,6329651200,93
41.00,0,0,7,6329778176,89
41.20,0,0,7,6329778176,100
41.40,0,0,7,6329798656,87
41.60,0,0,7,6329769984,93
41.80,0,0,7,6329786368,87
42.00,0,0,7,6329847808,93
42.20,0,0,7,6329856000,94
42.40,0,0,7,6329876480,86
42.60,0,0,7,6329847808,93
42.80,0,0,7,6329864192,94
43.00,0,0,7,6329868288,87
43.20,0,0,7,6329868288,94
43.40,0,0,7,6329880576,87
43.60,0,0,7,6329860096,94
43.80,0,0,7,6329868288,93
44.00,0,0,7,6329888768,88
44.20,0,0,7,6329880576,94
44.40,0,0,7,6329880576,93
44.60,0,0,6,6329880576,88

44.80,0,0,6,6329880576,93
45.00,0,0,6,6329884672,93
45.20,0,0,6,6335807488,93
Context 1 Queue 0,Available Physical Memory (Bytes),Procesador _Total: % de tiempo de procesador

ANEXO 2: Archivo de Profiling de gDebugger de la ejecución del prototipo sobre GPU

//
// This File contains OpenGL Profiling Data information
// Project name: C.MandelbrotOpenCL
// Generation date: Monday_11_June_2012
// Generation time: 17:35:07
//
// Generated by gDEBugger - an OpenGL Debugger and Profiler
// www.gremedy.com
//
Time (sec),Frames/sec: GL Context 1,OGL calls/frame: GL Context 1,Frames/sec: CL Context 1 Queue 0,Available Physical Memory
0.00,0,0,0,6342885376,0
0.20,0,0,0,6344200192,2
0.40,0,0,0,6344994816,14
0.60,0,0,0,6335127552,14
0.80,0,0,0,6317056000,13
1.00,0,0,0,6300426240,28
1.20,0,0,0,6300360704,24
1.40,0,0,0,6300229632,25
1.60,0,0,0,6300229632,33
1.80,0,0,0,6300229632,18
2.00,0,0,0,6300229632,9
2.20,0,0,0,6300262400,13
2.40,0,0,0,6300229632,57
2.60,0,0,0,6300229632,0
2.80,0,0,0,6300254208,18
3.00,0,0,0,6300254208,49
3.20,0,0,0,6300254208,3
3.40,0,0,0,6300254208,24
3.60,0,0,0,6300188672,39
3.80,0,0,0,6300188672,14
4.00,0,0,0,6300188672,55
4.20,0,0,0,6301908992,0
4.40,0,0,0,6302789632,29
4.60,0,0,0,6302789632,3
4.80,0,0,0,6302793728,37
5.00,0,0,0,6302793728,37
5.20,0,0,0,6302793728,35
5.40,0,0,0,6302806016,40
5.60,0,0,0,6302810112,32
5.80,0,0,0,6302810112,41
6.00,0,0,0,6302810112,41
6.20,0,0,0,6302818304,6
6.40,0,0,0,6302826496,43
6.60,0,0,0,6302797824,0
6.80,0,0,0,6302797824,37
7.00,0,0,0,6302797824,47

7.20,0,0,0,6302797824,0
7.40,0,0,0,6302797824,22
7.60,0,0,0,6302801920,7
7.80,0,0,0,6302801920,18
8.00,0,0,0,6302806016,16
8.20,0,0,0,6304104448,17
8.40,0,0,0,6304100352,19
8.60,0,0,0,6304120832,14
8.80,0,0,0,6304120832,17
9.00,0,0,0,6304120832,21
9.20,0,0,0,6304120832,9
9.40,0,0,0,6303723520,17
9.60,0,0,0,6303719424,13
9.80,0,0,0,6303719424,20
10.00,0,0,0,6303727616,43
10.20,0,0,0,6303727616,0
10.40,0,0,0,6303719424,20
10.60,0,0,0,6303719424,11
10.80,0,0,0,6303719424,33
11.00,0,0,0,6303719424,14
11.20,0,0,0,6303719424,10
11.40,0,0,0,6304309248,16
11.60,0,0,0,6304231424,14
11.80,0,0,0,6304231424,15
12.00,0,0,0,6304231424,29
12.20,0,0,0,6304227328,1
12.40,0,0,0,6304190464,22
12.60,0,0,0,6304104448,18
12.80,0,0,0,6303973376,4
13.00,0,0,0,6303973376,13
13.20,0,0,0,6303092736,22
13.40,0,0,0,6302998528,40
13.60,0,0,0,6302547968,0
13.80,0,0,0,6302547968,42
14.00,0,0,0,6302539776,0
14.20,0,0,0,6302539776,21
14.40,0,0,0,6302474240,12
14.60,0,0,0,6302474240,14
14.80,0,0,0,6302371840,13
15.00,0,0,0,6302388224,24
15.20,0,0,0,6302388224,37
15.40,0,0,0,6302396416,0
15.60,0,0,0,6302396416,23
15.80,0,0,0,6302396416,25
16.00,0,0,0,6302396416,16
16.20,0,0,0,6302396416,28
16.40,0,0,0,6302396416,15

16.60,0,0,0,6302404608,1
16.80,0,0,0,6302404608,25
17.00,0,0,0,6302404608,19
17.20,0,0,0,6302408704,16
17.40,0,0,0,6302355456,14
17.60,0,0,0,6302371840,25
17.80,0,0,0,6302371840,31
18.00,0,0,0,6302371840,8
18.20,0,0,0,6302367744,7
18.40,0,0,0,6302367744,25
18.60,0,0,0,6302367744,17
18.80,0,0,0,6306357248,11
Context 1 Queue 0,Available Physical Memory (Bytes),Procesador _Total: % de tiempo de procesador