CENTRO INTERNACIONAL DE MÉTODOS NUMÉRICOS EN INGENIERÍA

Implementación de Algoritmos Numéricos en una Tarjeta Gráfica

M. Pasenau
A. Fernández Jiménez

Monografía CIMNE Nº-99, Julio 2006

Implementación de Algoritmos Numéricos en una Tarjeta Gráfica

M. Pasenau
A. Fernández Jiménez

Monografía CIMNE Nº-99, Julio 2006

Centro Internacional de Métodos Numéricos en Ingeniería
Gran Capitán s/n, 08034 Barcelona, España

CENTRO INTERNACIONAL DE MÉTODOS NUMÉRICOS EN INGENIERÍA
Edificio C1, Campus Norte UPC
Gran Capitán s/n
08034 Barcelona, España
www.cimne.upc.es

Primera edición: Julio 2006

IMPLEMENTACIÓN DE ALGORITMOS NUMÉRICOS EN UNA TARJETA GRÁFICA
Monografía CIMNE M99

ISBN: 84-96736-05-9

Depósito legal: B-46924-2006

Índice:

8. Bibliografía

- GPGPU y GPU
- Tutoriales GPGPU
- Lenguajes generales que abstraen la GPU
- Lenguajes de sombreado
- OpenGL
- Extensiones OpenGL
- Hardware OpenGL y arquitectura OpenGL

1. Introducción

En los últimos 10 años, la evolución más importante en un PC ha sido la de las tarjetas gráficas. Pasando de una mera memoria de puntos a dibujar en pantalla a ser verdaderas mini-supercomputadoras, capaces de crear escenas 3D con tantos detalles y efectos que al usuario final le cuesta distinguirlas de la realidad, y en tiempo real.

En algunos artículos [A.2, A.3, A.4], se habla del uso de estas tarjetas gráficas para resolver problemas algebraicos, por ejemplo sistemas de ecuaciones, de gran tamaño y donde se consigue un rendimiento hasta 10 veces superior a un ordenador de sobremesa convencional.

Dado el alto coste de una estación de trabajo, una Sun Ultra 40 con dos procesadores 'dual core' se puede encontrar a partir de 9.000 €, comparado con el de una de las tarjetas de alto nivel, una XFX GeForce 7800 se puede encontrar a partir de 350 €, parece dibujarse como una alternativa a los clusters de súper computación, teniendo en cuenta que se puede instalar más de una tarjeta gráfica en un mismo ordenador.

Por otro lado, la evolución de los procesadores de sobremesa se ha visto limitada por problemas de calor y consumo eléctrico, mostrándose una tendencia a paralelizar su arquitectura implementando más unidades de proceso en el mismo chip [G.3]. Una tendencia que lleva ya años implementando la industria en sus GPU, Graphical Processor Unit.

Por ello cabe preguntarse ¿puede utilizarse esta potencia para otras aplicaciones además de los juegos? ¿Cómo pueden los ingenieros usar este poderío de cálculo fácilmente? ¿Merece la pena invertir esfuerzo en este campo? Preguntas que intentaré responder en esta monografía. Primero veremos la evolución de las tarjetas gráficas y las CPU, su coste y compararemos el mundo de los juegos 3D y el de la simulación por ordenador.

Definiré los objetivos de la monografía que consistirá en valorar las herramientas existentes para usar estas tarjetas como coprocesador y desarrollar una librería para usarlas, estableceré un plan de desarrollo y el entorno de trabajo.

Contrastaremos las herramientas existentes para programar estas tarjetas gráficas desde un punto de vista de un profano en el ámbito de gráficos, su facilidad de uso y definiremos las líneas maestras de la librería y la implementaremos.
Finalmente analizaremos los resultados obtenidos y enumeraremos futuras líneas de trabajo.

2. Motivación

2.1 Evolución de las tarjetas gráficas

Desde la introducción del PC hace más de 25 años, la evolución de las tarjetas gráficas se puede dividir en dos estadios: los primeros 15 años, en que apenas han evolucionado, y los últimos 10 años, en que su evolución ha sido espectacular.

Desde la primera tarjeta gráfica aparecida en 1982 (Hercules Graphics Card), con una resolución monocroma de 720x350 puntos, hasta la aparición de la tarjeta EGA (Enhanced Graphics Array) en 1984, la única aceleración de que disponían era el modo texto: la CPU enviaba el código de la letra y la tarjeta lo dibujaba en pantalla. La tarjeta EGA contaba con su propia memoria y su propia BIOS lo que abría la etapa de liberar parte del trabajo de la CPU y traspasarlo a la tarjeta gráfica [G.13, G.14].

En esa época los juegos como Elite (1987 Figura 1), Elite Plus (1988 Figura 2) de Acornsoft, considerados como los primeros juegos 3D en PC estaban íntegramente escritos en ensamblador y usaban enteros como tipo básico de datos en gráficos. O los primeros juegos FPO (First Person Shooter) de Id Software: Hovertank 3D (1992 Figura 3), Catacomb 3D (1991 Figura 4) y su célebre sucesor Wolfstein 3D (1992 Figura 5) que comenzaban a usar texturas y todo un motor 3D (usando ray-casting) en enteros. Todo el trabajo de creación de la imagen 3D se realizaba en la CPU.

Figura 1: Elite de Acornsoft que apareció en 1987 para PC con EGA. Imagen de un BBC Micro

Figura 2: Elite Plus de Acornsoft que apareció en 1988 para PC con VGA

Figura 3: Hovertank 3D de Id Software apareció en Abril 1991 para PC con VGA

Figura 4: Catacomb 3D lanzado el Noviembre de 1991 para PC con VGA

Figura 5: Publicado por Apogee Software, Wolfstein 3D fue creado por Id Software para PC con VGA

Todos estos juegos llevaban incorporado su motor 3D que utilizaba la CPU para crear las imágenes del juego.

No sería hasta 1996 cuando la compañía 3Dfx presentó la primera aceleradora 3D Voodoo para PC, pero requiriendo de una tarjeta 2D para visualizar los resultados en pantalla. Previamente S3 introdujo su tarjeta Virge que incorporaba z-buffer pero pronto se la conoció como 'desaceleradora' 3D, pues la emulación 3D en la CPU sobrepasaba en velocidad a la tarjeta. [G.13] En la figura 6 podemos ver uno de los primeros ejemplos en que se podía activar y desactivar la aceleración gráfica, aunque a una resolución muy baja [G.3].

3Dfx publicó entonces su librería 3D 'Glide' para usar la aceleración de sus tarjetas Voodoo usada por la mayor parte de los juegos como el Monster Truck Mandess de Microsoft (1996 Figuras 6), que también posibilitaba realizar el render por software. 3Dfx también realizó una implementación reducida de la librería OpenGL ampliamente utilizada en las estaciones de trabajo de SGI (antes Silicon Graphics), Sun, HP e IBM. Se llamaba 'miniGL' y la usaba, entre otros, Quake de Id Software.

Figura 6: Monster Truck Mandes de Microsoft ejecutándose en un Intel Pentium 166MHz con una resolución de 640x480 y 256 colores en una S3 Virge 325.

Con la compra de la compañía RenderMorphics, que desarrollaba una API 3D llamada Render Lab, en 1995 Microsoft llevó el mundo 3D a Windows 95, cambiando el nombre de dicha librería a Direct3D. Gracias a una serie de acuerdos con las principales compañías desarrolladoras de juegos, impulsó el uso de esta librería y, junto a su implementación de OpenGL para Windows NT, y más tarde, para Windows 95 OSR2, estableció dos estándares para el desarrollo de aplicaciones 3D en el PC [G.13].

A partir de entonces, hace 10 años, la evolución más importante en el mundo del PC ha sido el de las tarjetas gráficas. Con sólo dos estándares con los que hacer compatibles sus desarrollos, la industria, liderada por NVIDIA y, más tarde, junto con ATI, ha implementado por hardware, y con esto acelerado, las diferentes etapas de dibujo de escenas 3D, con todos sus efectos de luz, agua, lluvia, sombras, reflejos, traslucimientos, etc. Sirva como ejemplo las siguientes figuras de Age of Empires III de Ensemble Studios y publicado por Microsoft, aparecido en Octubre de 2005 y de Oblivion de Bethesda Softworks lanzado en Marzo de 2006 [G.3, G.12, G.13, G.15].

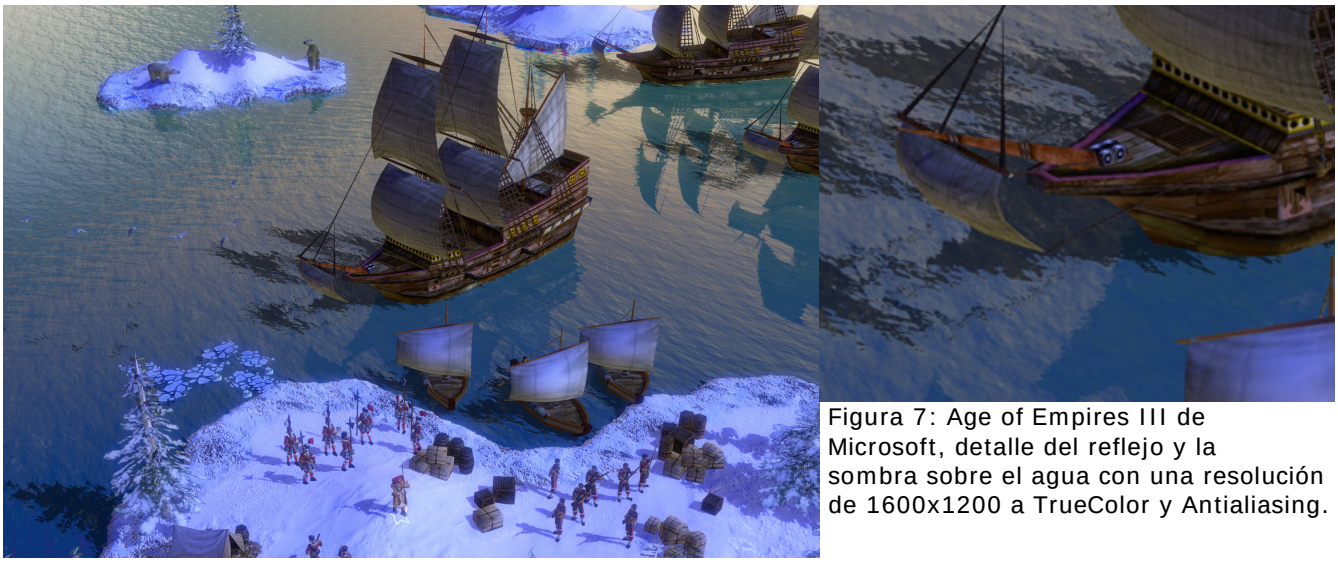


Figura 7: Age of Empires III de Microsoft, detalle del reflejo y la sombra sobre el agua con una resolución de 1600x1200 a TrueColor y Antialiasing.

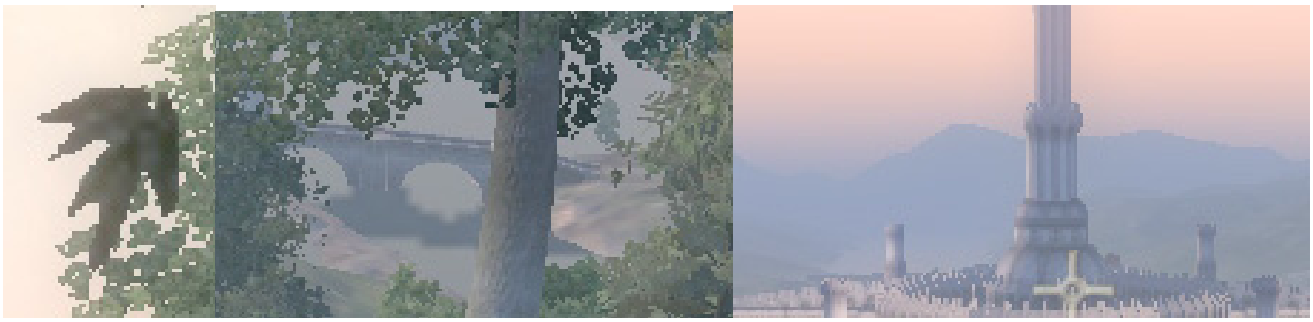


Figura 8: Oblivion de Bethesda Softworks, detalle de la hoja de árbol, reflejo del puente en el agua y el ocaso

Los juegos de ahora permiten resoluciones de hasta 1920x1200 puntos, incluso 3840x1024 puntos con 3 pantallas, y color verdadero (16.777.216 colores) y conjugan diferentes efectos como transparencias, traslucimientos, objetos con relieve fijo y variable, diversas luces fijas y dinámicas, reflejos, sombras, superficies con olas de agua, aceite, efectos meteorológicos como lluvia, niebla, amanecer, ocaso, fuego, lava, etc.. Todo esto se dibuja en tiempo real y a una velocidad de entre 30 y 100 veces por segundo. Con la introducción, por parte de NVIDIA, del chip GeForce 256 para tarjetas gráficas en 1999 se acuña el término de GPU (Graphical Processor Units) para los chips que aceleran todo el proceso gráfico, desde triángulos del mundo real hasta la pantalla, en las tarjetas aceleradoras 3D [G.3].

.

Figura 9: Oblivion de Bethesda Softworks, detalle del reflejo del puente en la superficie ondulante del agua, del relieve de la base de la estatua y de las nubes en el cielo, junto con la difuminación en la distancia

2.2 Comparativa CPU ←→ GPU

La evolución de las tarjetas gráficas ha sido mucho más acelerada que la de los procesadores: el Intel Pentium 4 sigue siendo básicamente el mismo que el de hace 10 años, un procesador serial con algunas extensiones añadidas. Las únicas innovaciones importantes, a parte de multiplicar por 23 su velocidad de reloj (de 166MHz, a 3,8 GHz), que no su velocidad de proceso, y las instrucciones SSE/3dNow!, han sido implementadas en el último par de años: procesamiento de 64 bits, ahorro de energía y doble núcleo.

Si comparamos el rendimiento en coma flotante y la velocidad de transferencia entre el procesador y la memoria observamos que las GPUs son rápidas:

- 24,6 Gflops en una CPU Intel Pentium 4 Dual Core
- 165 Gflops en una GPU NVIDIA GeForce 7800
- 8,5 GBytes/s en un Intel Pentium 4 Extreme Edition
- 37,8 GBytes/s en una tarjeta ATI Radeon X850 XT Platinum edition

Y si comparamos su evolución en los últimos años, serán más rápidas aún:

- Factor de crecimiento para las CPU's 1,4x
- Factor de crecimiento para las GPU's 1,7x (píxeles) y 2,3x (vértices)

En la siguiente figura podemos contrastar esta evolución [A.6]:

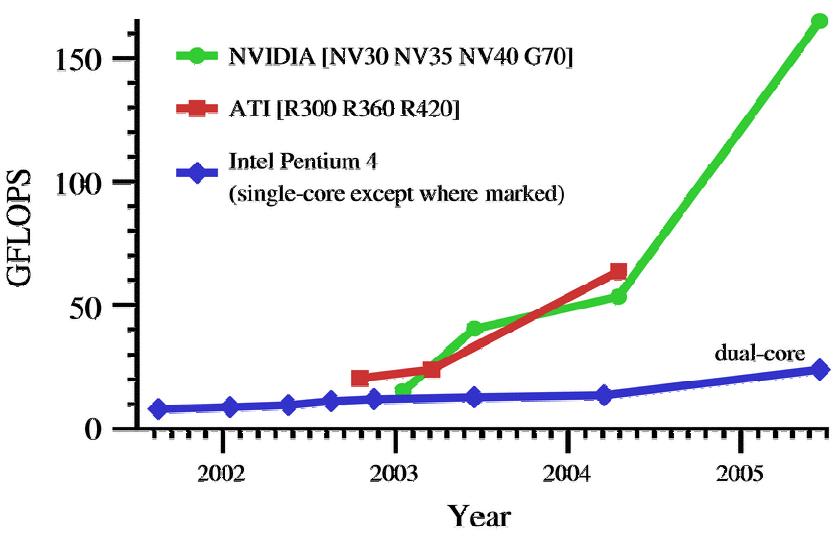


Figura 10: curvas de crecimiento

NVIDIA:
NV30 – GeForce FX 5800
NV35 – GeForce FX 5900
NV40 – GeForce 6800
G70 – GeForce 7800

ATI:
R300 – Radeon 9700
R360 – Radeon 9800XT
R420 – Radeon X800

En los últimos años el aumento del rendimiento de los procesadores se ha visto frenado por el calor que se ha de disipar (hasta 130W en los Intel Pentium 4), como se puede observar en la figura 11, y su consumo eléctrico, por ejemplo, no se ha llegado a la barrera de los 4GHz. Para solventar este problema, junto a algoritmos de ahorro de energía, como desactivar partes del procesador que no se usan, o bajar su velocidad cuando esta inactivo, se está rediseñando su arquitectura y se usa tecnología de 65nm. Actualmente para aumentar su rendimiento, se está aumentando la caché o incrustando dos núcleos de proceso en el mismo chip, es decir se está paralelizando. Un Intel Pentium 4 EE a 3,2 GHz tiene 178 millones de transistores de los cuales, 149 corresponden a la caché de 2MBytes y sólo 29 al procesador

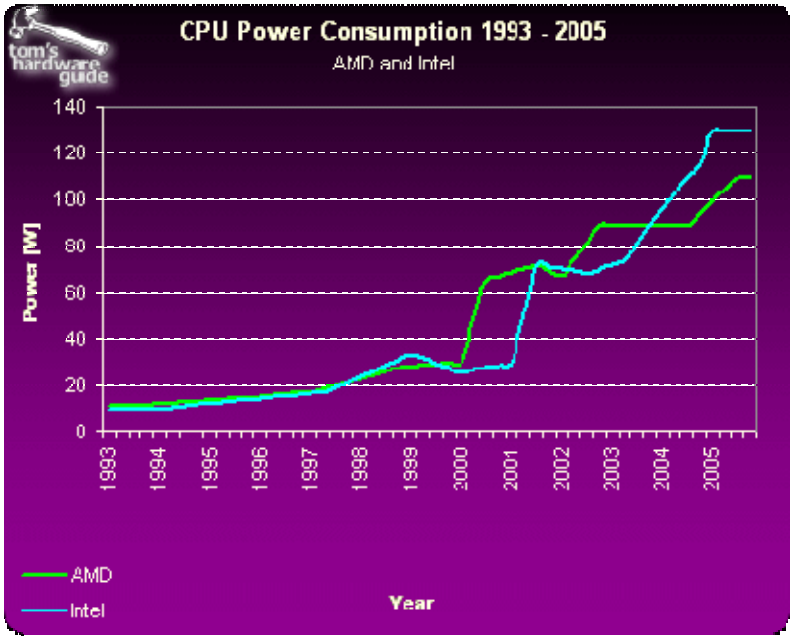


Figura 11: evolución del consumo de los procesadores de Intel y AMD. Ya desde 1999 AMD siguió su propio camino en el rediseño de la arquitectura de sus procesadores, siendo ahora los más eficientes tanto energéticamente como en cuanto a rendimiento.

Si bien las tarjetas llegan a consumir más de 120W, en la NVIDIA 7800 GTX 512, dentro de este consumo también se cuenta, no sólo la GPU, sino también la memoria y toda la circuitería para generar las imágenes. La GPU NVIDIA 7800GTX está fabricada con tecnología de 110nm y 302 millones de transistores, y actualmente las ATI X1900X, x1800x se están fabricando con procesos de 90nm, también la NVIDIA 7900 se está fabricando a 90nm y rediseñado su arquitectura con lo que se ha rebajado el número de transistores a 278 millones de transistores [G.3]. Aún les queda camino a las GPU's para aplicar la tecnología y experiencia obtenida en las CPU respecto ahorro de energía.

2.3 Arquitectura paralela

El motivo de esta formidable evolución es debida, básicamente, a la arquitectura altamente paralela del proceso de dibujado.

Según el proceso de dibujado definido en OpenGL [E.1], cada vértice de las primitivas de entrada (listas de vértices, de triángulos, 'tiras' u otros) puede ser procesado por separado. Este procesado consiste en operar normales del vértice, vectores de colores, vectores de iluminación, cálculo de coordenadas de texturas u otros efectos ('operaciones por vértice' en la Figura 12) y se transforman a coordenadas homogéneas (0.0-1.0).

Más tarde se agrupan estos vértices en sus respectivas primitivas: triángulos, cuadrados, etc. ('ensamblado de primitivas' en la Figura 12). Después de tener los vértices agrupados

por primitivas, por ejemplo en triángulos, estos se transforman a coordenadas de pantalla según la vista definida, 'viewport', por ejemplo a 0-800x600, se descartan los triángulos cuyas normales no apunten al usuario, 'culling' si éste está definido, y se recortan si parte cae dentro del semiespacio definido por un plano de corte, 'cliping' ('Recortes' en la Figura 12).

Después de esta etapa, se pasa a la 'digitalización' de las primitivas, ya transformadas a coordenadas de pantalla y recortadas. Cada primitiva crea una serie de píxeles asociados, por ejemplo los que 'pintan' el interior de un triángulo, con su color interpolado de entre los colores definidos en los vértices, con sus coordenadas de textura, si éstas se definieron, su información de profundidad, 'Z'. A esta 'discretización' de la figura geométrica triángulo, se le conoce como 'rasterización' (en la Figura 12), y a todo este conjunto de información: las coordenadas del píxel y la información antes comentada, se le conoce como 'fragmento'.

Con toda esta información pasamos a la siguiente etapa, que consiste en realizar operaciones con cada uno de los fragmentos ('operaciones por fragmento' en la Figura 12). Entre estas operaciones se cuentan el operar con los texels (elementos de textura que le corresponden al fragmento) de las diferentes texturas, los colores, valores de niebla, máscaras de recortado, 'scissor' (recorte de la vista), de transparencia para obtener el que finalmente se aplicará al fragmento. Si además pasa el test de profundidad y el de 'stencil' (buffer con valores según los cuales se pinta o no) pasará a la última etapa: operaciones en el buffer de pantalla (en la Figura 12). Dentro de esta etapa se escribirá el píxel en pantalla según las máscara de escritura de color, las operaciones lógicas definidas o el 'dithering' (patrón de pintado para simular con pocos colores, millones de colores), si está habilitado, y su profundidad, 'Z', si lo permite también su máscara [E.1, G.4].

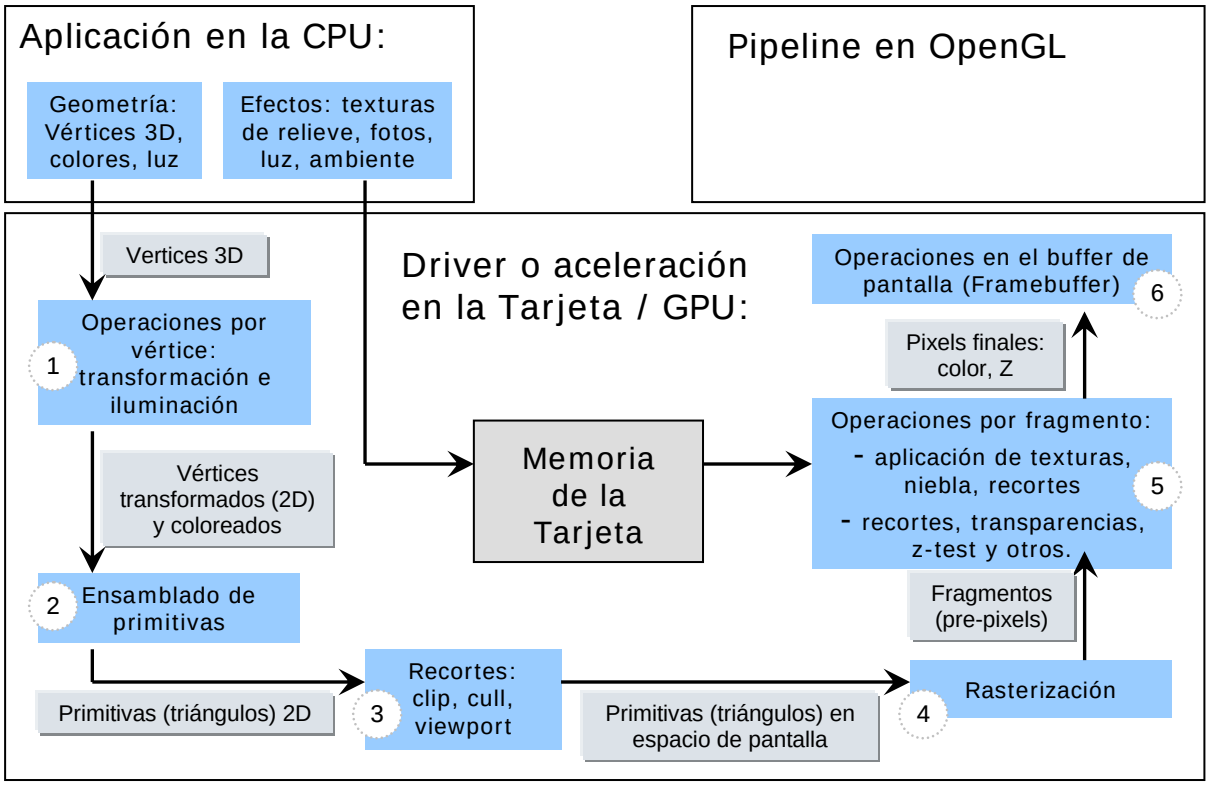


Figura 12: Diferentes etapas que forman el proceso de dibujado en OpenGL

En la siguiente figura podemos ver un ejemplo de un triángulo con textura a través de las diferentes etapas.

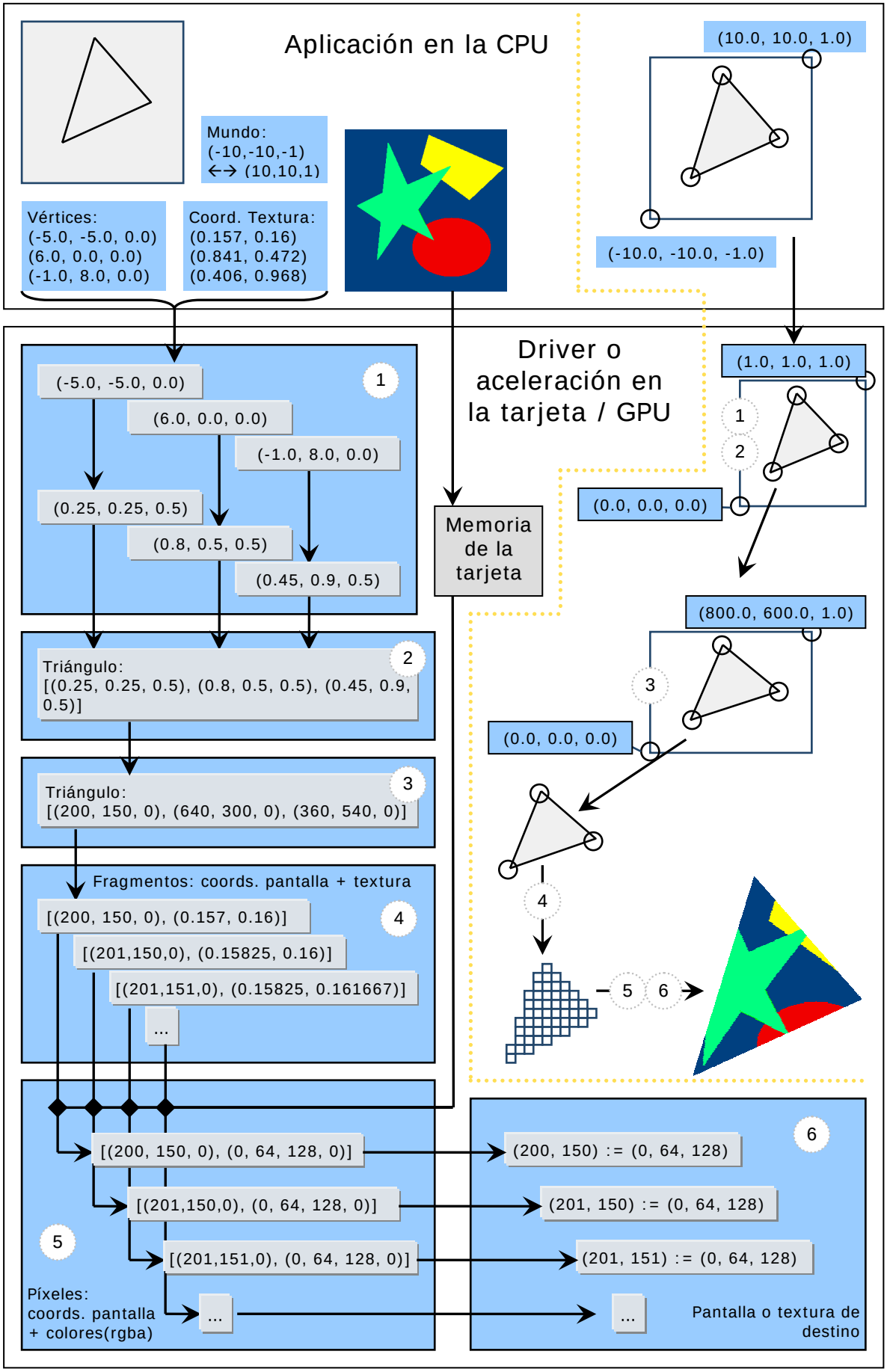

Aplicación en la CPU
Mundo:
(-10,-10,-1)
↔ (10,10,1)
Vértices:
(-5.0, -5.0, 0.0)
(6.0, 0.0, 0.0)
(-1.0, 8.0, 0.0)
Coord. Textura:
(0.157, 0.16)
(0.841, 0.472)
(0.406, 0.968)
(10.0, 10.0, 1.0)
(-10.0, -10.0, -1.0)
Driver o aceleración en la tarjeta / GPU
Memoria de la tarjeta
(-5.0, -5.0, 0.0)
(6.0, 0.0, 0.0)
(-1.0, 8.0, 0.0)
(0.25, 0.25, 0.5)
(0.8, 0.5, 0.5)
(0.45, 0.9, 0.5)
1
Triángulo:
[(0.25, 0.25, 0.5), (0.8, 0.5, 0.5), (0.45, 0.9, 0.5)]
2
Triángulo:
[(200, 150, 0), (640, 300, 0), (360, 540, 0)]
3
Fragmentos: coords. pantalla + textura
[(200, 150, 0), (0.157, 0.16)]
[(201,150,0), (0.15825, 0.16)]
[(201,151,0), (0.15825, 0.161667)]
...
4
[(200, 150, 0), (0, 64, 128, 0)]
[(201,150,0), (0, 64, 128, 0)]
[(201,151,0), (0, 64, 128, 0)]
...
5
Píxeles:
coords. pantalla
+ colores(rgba)
(1.0, 1.0, 1.0)
(0.0, 0.0, 0.0)
(800.0, 600.0, 1.0)
(0.0, 0.0, 0.0)
(200, 150) := (0, 64, 128)
(201, 150) := (0, 64, 128)
(201, 151) := (0, 64, 128)
...
6
Pantalla o textura de destino

Figura 13: Paso por las diferentes etapas de un triángulo con textura.

Como podemos ver es un proceso altamente paralelizable donde la entrada está bien definida, vértices o texturas, y la salida también: la pantalla. El procesamiento de cada vértice es independiente, por tanto no afecta a los demás vértices, y lo mismo para los fragmentos, son independiente entre ellos.

La implementación por hardware todas estas etapas en un solo chip apareció por primera vez con la NVIDIA GeForce 256 en 1999 y se le llamó 'hardware transform and lighting (T&L)', capaz de procesar 10 millones de polígonos por segundo. No fue hasta el 2002 que ATI le siguió con su Radeon 8500 [G.3].
A medida que los desarrolladores de juegos demandaban nuevos efectos, más complejos y variados, se vio la necesidad de implementar unidades programables. Se comenzó con el 'operador de fragmentos', también conocido como 'píxel' o 'fragment shader' y enseguida le siguió el operador de vértices, conocido como 'vertex shader' (véase la Figura 14). Con su definición en la librería Microsoft DirectX 8 apareció la GeForce 3 en el 2001 con su implementación por hardware [G.3].
Pero se vio pronto que estaba limitado: la resolución era entera, no disponía por tanto de coma flotante, y tampoco disponía de instrucciones condicionales ni de salto. Pronto salió a la luz DirectX 9 y DirectX 9.0b con el 'Shader Model' 2.0 con soporte de instrucciones de salto, bucles y resolución en coma flotante de 96 y 128 bits. NVIDIA lo implementó por hardware con su GeForce FX en el 2002 y ATI respondió con su Radeon 9700 (R300) [F.7].
OpenGL tardó en incorporar esta programabilidad y no fue hasta el 2003 que lo incorporó como extensión, GLSL, en su versión 1.5 y formalmente dentro de OpenGL 2.0 en el 2004 [E.2, E.3].

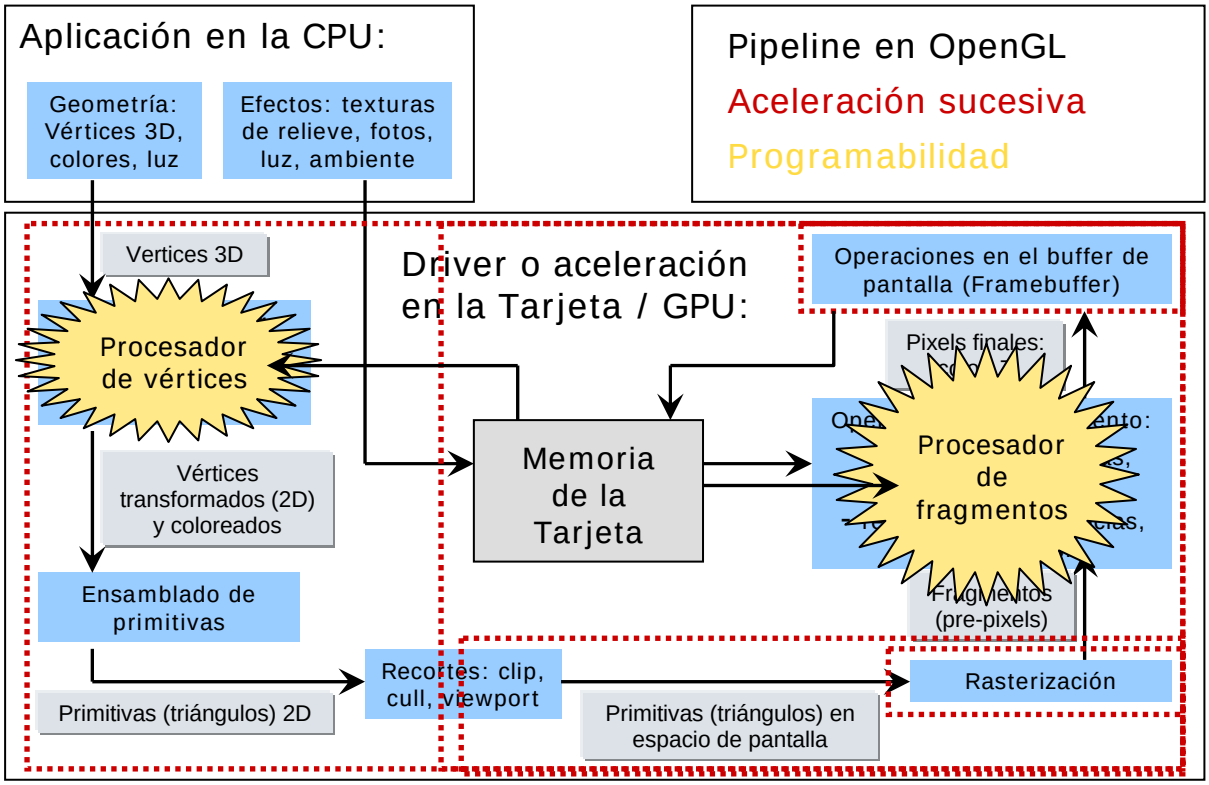


Figura 14: Programabilidad añadida en el proceso de dibujado.

Dos de las últimas incorporaciones han sido: una la de poder usar texturas dentro del procesador de vértices, y la de poder pintar 'fuera de pantalla', es decir en una textura que

después, una vez acabado el proceso de dibujado, se pueda utilizar dentro del siguiente proceso de dibujado (véase la Figura 15) [F.1, F.2, F.3, F.4].

Para no complicar mucho la arquitectura de la tarjeta gráfica, y como el objetivo es la visualización gráfica, las texturas sólo se pueden acceder de una manera dentro de un bucle de pintado: o son leídas desde el procesador de vértices o de fragmentos o son escritas, como destino del bucle de pintado, como podemos observar en la figura 15. Con lo que para poder usar parte del resultado de la textura pintada el programa ha de esperar a finalizar todo el proceso de pintado entero para en la siguiente vuelta definir otra textura como destino del dibujo y así poder usar el resultado de la anterior.

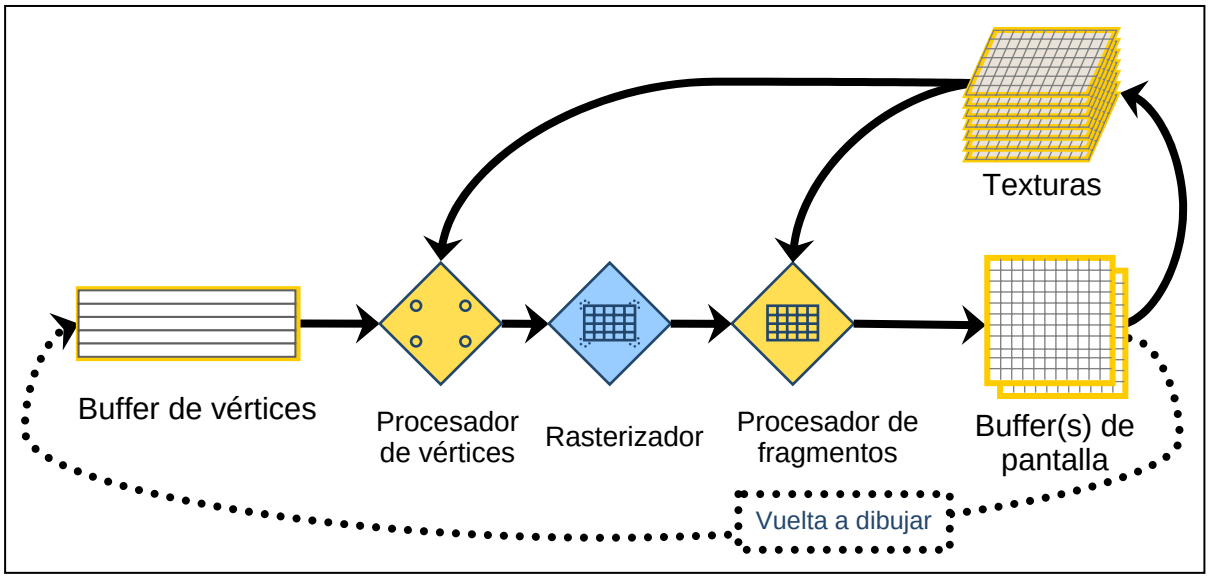


Figura 15: Pipeline gráfica actual resumida.

En las figuras 16 y 17 tenemos dos ejemplos de implementación del 'pipeline' gráfico en la arquitectura de la GPU de NVIDIA GeForce 6800 Ultra y de la ATI Radeon X1800X [G.3].

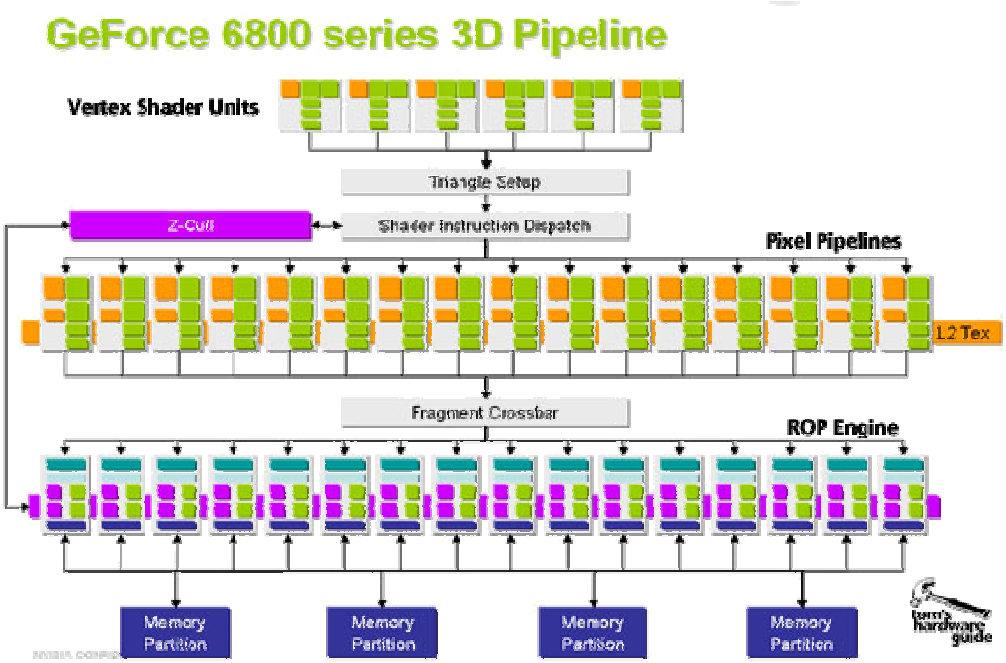


Figura 16: Esquema de la arquitectura de la GPU NVIDIA GeForce 6800 Ultra

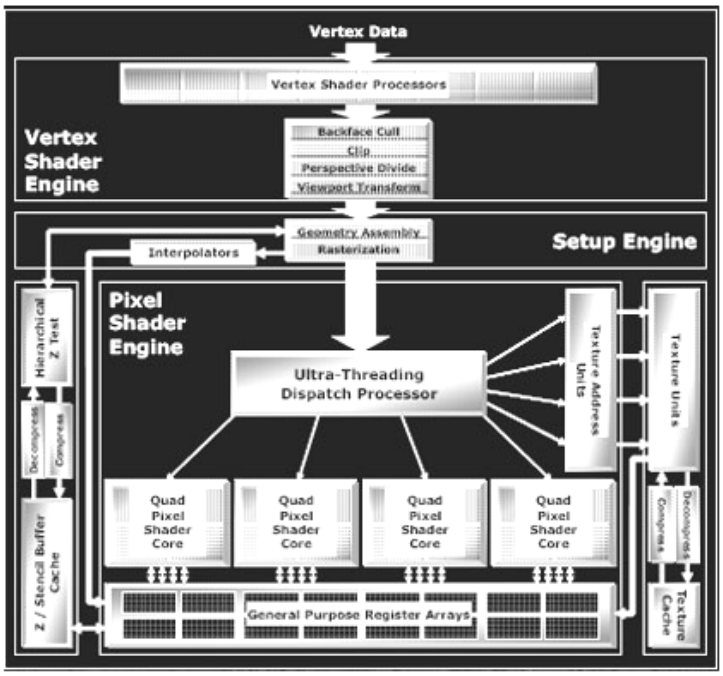


Figura 17: Esquema de la arquitectura de la GPU ATI Radeon X1800X

En la figura 18 podemos ver los detalles de uno de estos procesadores de vértices y de fragmentos.

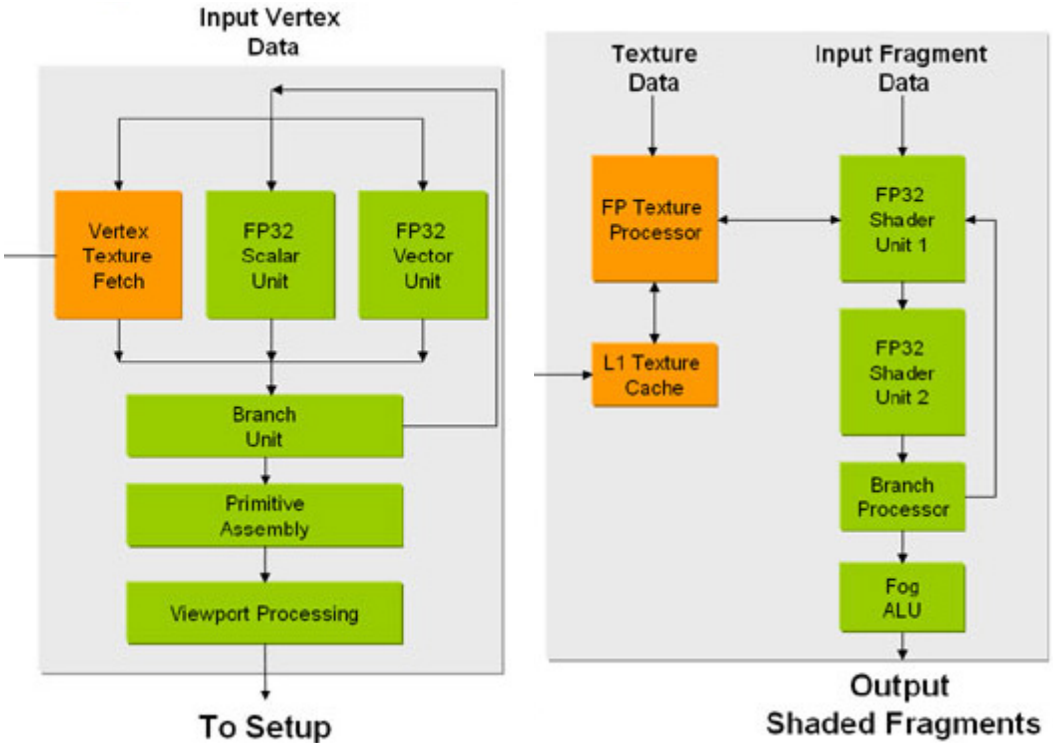


Figura 18: Detalle del procesador de vértices, izquierda, y de fragmentos, derecha, de la GPU NVIDIA GeForce 6800 Ultra

A título indicativo, las especificaciones de las últimas tarjetas en el mercado de las compañías NVIDIA y ATI son estas:

NVIDIA GeForce 7900 (Marzo 2006):

- 24 procesadores de fragmentos
- 8 procesadores de vértices: 1400 millones de vértices/s
- 278 millones de transistores
- 51,2 GBytes/s de transferencia en memoria de la tarjeta
- GPU a 650 MHz y 512 MBytes de memoria GDDR3 a 800 MHz y ancho de bus de 256 bits

ATI Radeon X1900XTX (Enero 2006):

- 48 procesadores de fragmentos
- 8 procesadores de vértices: 1300 millones de vértices/s
- 384 millones de transistores
- 49,6 GBytes de transferencia en memoria de la tarjeta
- GPU a 640 MHz y 512 MBytes de memoria GDDR3 a 775 MHz y ancho de bus de 256 bits

Por último mencionar que la tarjeta ATI FireGL V7350 dispone de 1GByte de memoria en la tarjeta [G.2].

2.4 El mundo de la simulación

Ingenieros simulan por ordenador varios procesos industriales y naturales: análisis de estructuras de edificios, presas, estampación de chapas, para capós y puertas de coches, aerodinámica de aviones, hidrodinámica de barcos y submarinos, procesos de fundición, de rellenado de moldes y enfriamiento, procesos de erosión, transporte de partículas, inundaciones, incendios y un largo etcétera [A.16, A.17, A.19, A.20 y A.21].

Para calcular estas simulaciones se utilizan varios métodos, entre ellos diferencias finitas, elementos finitos, volúmenes finitos, métodos sin malla y métodos de partículas.

Visto a 'grosso modo' todos ellos consisten en:

- Discretizar una geometría compleja en:
  - un conjunto de elementos simples: líneas, triángulos, tetraedros entre otros, para diferencias finitas, elementos finitos y volúmenes finitos conocido como malla
  - nubes de puntos o partículas con volumen para el esto
- Imponer condiciones iniciales, por ejemplo velocidad del fluido o la estructura, pesos, fuerzas y presiones en la estructura, y condiciones de contorno, por ejemplo estructura empotrada en el suelo, restricciones de movimiento o paredes por donde no puede salir el fluido)
- Asignar las propiedades de los materiales: agua, hormigón, acero,
- Definir las incógnitas: por ejemplo los desplazamientos de una deformación en los nodos, la presión del fluido, su velocidad
- Con todo esto obtenemos un sistema de ecuaciones con sus incógnitas. Para resolverlo incorporamos la contribución de cada uno de los elementos a una matriz grande, proceso que se conoce como 'ensamblar la matriz' y resolver:

  $$K \cdot a = f \qquad \text{(o el equivalente } A \cdot x = b\text{)}$$

  Donde 'K' es la matriz, 'a' el vector de incógnitas y 'f', los valores iniciales [A.21].

Para resolver esta última ecuación se utilizan diferentes métodos: gradientes conjugados, Gauss-Jordan, Cholesky, etc.
En la siguiente figura podemos ver algunos ejemplos de análisis hecho por programas de simulación numérica.

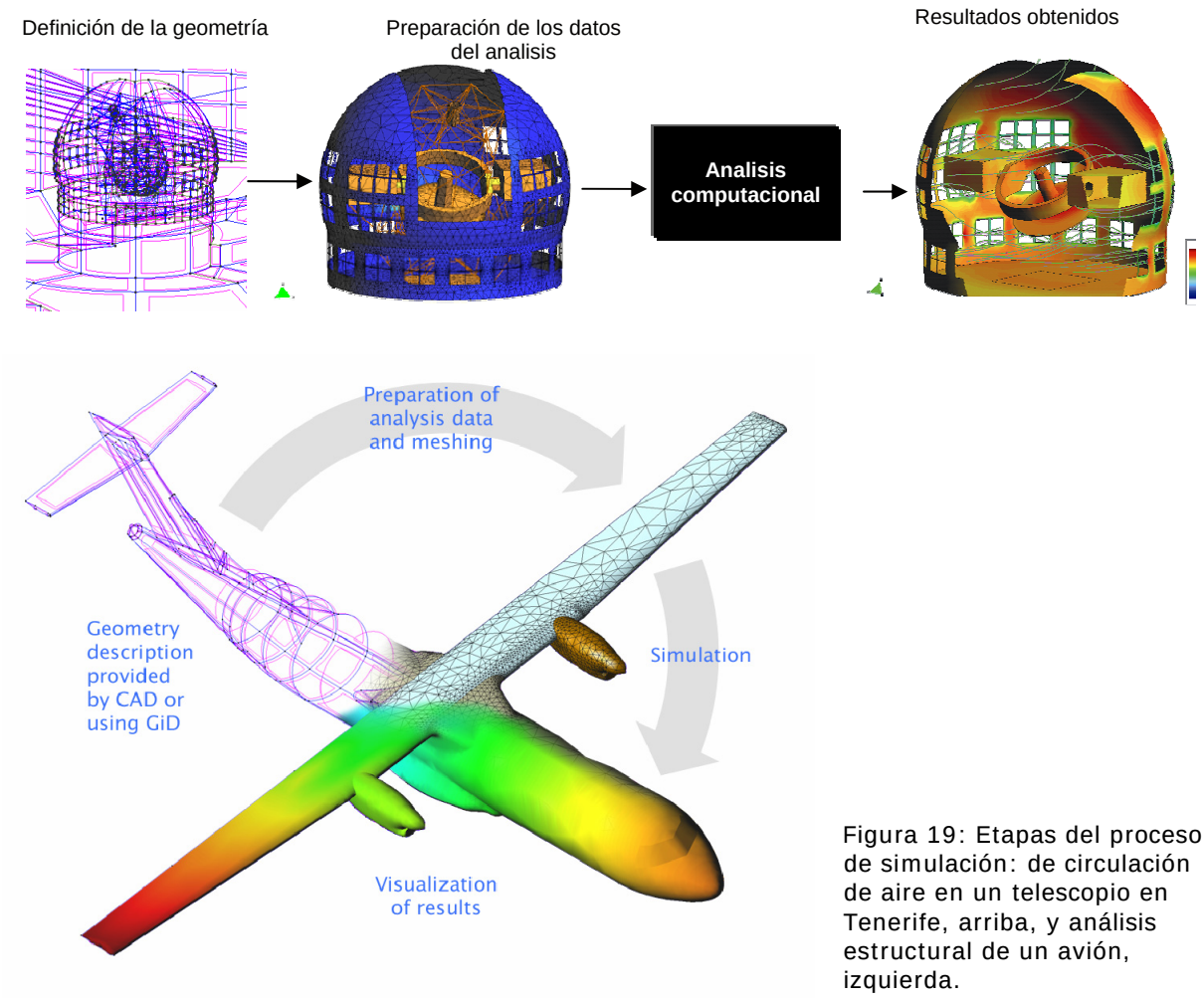


Figura 19: Etapas del proceso de simulación: de circulación de aire en un telescopio en Tenerife, arriba, y análisis estructural de un avión, izquierda.

2.5 Símil Juego 3D ←→ Simulación

Si analizamos con un poco más de detalle, observamos que un juego 3D, dejando de lado la parte de IA que hay detrás del comportamiento del ordenador, consiste en:

- Mallas de triángulos, cuadriláteros y algunas líneas para representar tanto objetos inanimados: edificios, farolas, terrenos de paisajes, superficie líquidas, como animados: personas, coches, animales, la hierba movida por el viento, con diferentes niveles de detalle, LOD, que se usan según la distancia al observador
- Muchas texturas que contienen:
    - Imágenes y fotos reales o creadas por ordenador para simplificar la geometría: por ejemplo una pared de ladrillos se visualiza con un par de triángulos y una imagen con unos cuantos ladrillos.
    - Transparencias, también para simplificar geometría: por ejemplo, en la Figura 8, en el detalle de la hoja de árbol, no se dibuja cada hoja de árbol con una malla de triángulos, si no que se pinta un cuadrilátero y se mapea una imagen con transparencia.
    - Información de relieve: conteniendo normales para cada texel de la imagen, por ejemplo en la Figura 9 para pintar la base de la estatua simplemente se usan un par de triángulos para cada cara y se mapea una textura con la imagen de la base y otra con las normales para darle el aspecto de relieve;

  - Información de iluminación, para simplificar el cálculo de luces: la luz de una farola que ilumina una pared es siempre la misma, a no ser que se apague, con lo que, para ahorrar este cálculo a la tarjeta se añade una textura de iluminación a la del relieve y de la imagen de la pared.
- Efectos 3D: animaciones y cálculo de luces, por ejemplo los faros de un coche en marcha, mallas en movimiento, cálculo de las ondas de agua en la superficie de un lago, simulación de eventos atmosféricos: lluvia, niebla, tormenta, movimiento del sol, evolución de sombras, reflejos transparencias y traslucimientos.

En la figura 20 vemos un ejemplo de la imagen que ve un jugador y las mallas de soporte de dibujo, de cálculo de sombras y ocultaciones.

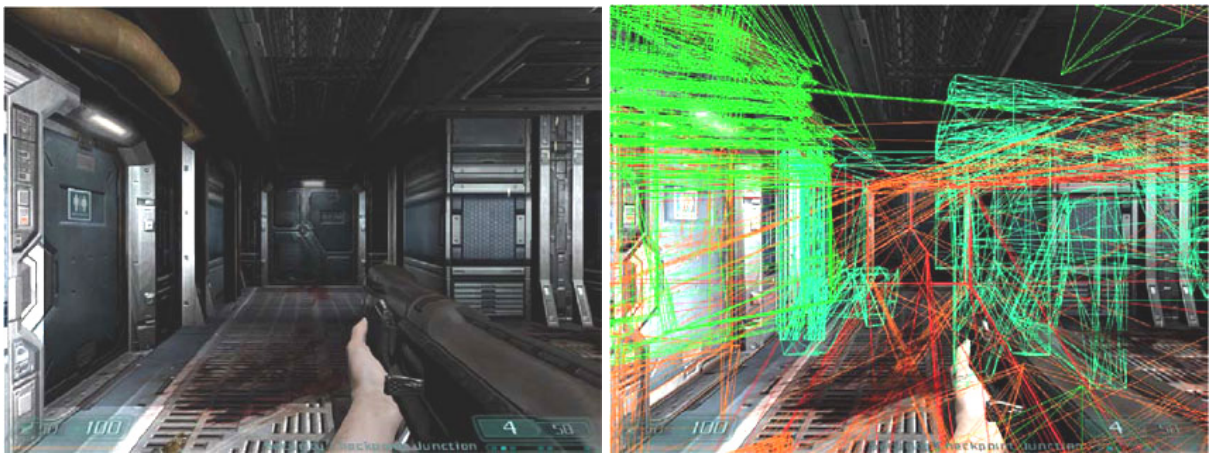


Figura 20: Malla de triángulos para pintar y para el cálculo de sombras en un juego 3D.

En resumen: muchas operaciones relativamente sencillas entre texturas y mallas.

Por otro lado, en un programa de cálculo, como ya hemos visto, consiste en montar todas las ecuaciones en una matriz grande y resolver el sistema ‘K · a = f’. En resumen, operaciones más o menos complejas sobre vectores grandes y matrices grandes y casi vacías como podemos comprobar en la siguiente figura [A.18 y A.21]:

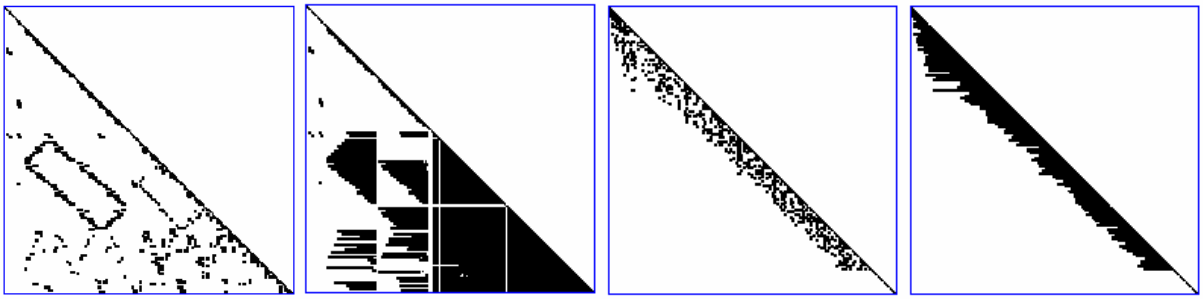


Figura 21: Matrices dispersas para una malla de tetraedros de 111 nodos según diferentes esquemas de renumeración y almacenamiento.

Si ponemos lado a lado las necesidades de ambos tipos de programa:

Juegos 3D	Simulación
Malla de triángulos y cuadriláteros	Mallas de diferentes tipos: 2D y 3D
Operaciones sobre datos simples:	Operaciones sobre datos simples:

• Entre texturas (matrices 2D) donde cada elemento son vectores de 2, 3 o 4 componentes: Imagen * ilum. * Transp. + relieve • Entre vectores: operaciones con normales, vectores de color, de luz…	• Matrices y vectores bastante grandes • Algunas multiplicaciones, sumas y escalados

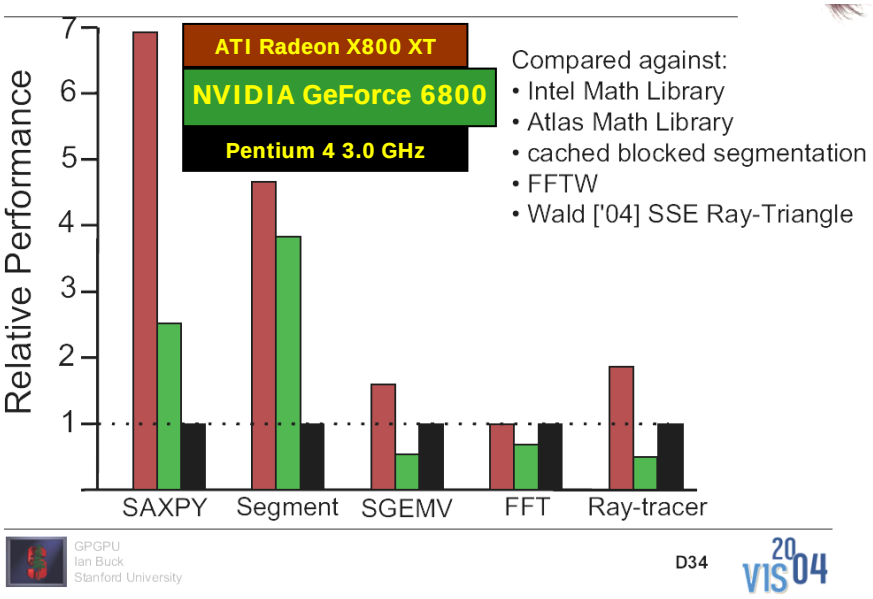
Vemos que las similitudes son muchas, y más si tenemos en cuenta que los juegos actuales incorporan cada vez más y mejores modelos de comportamiento: detector de colisiones, efectos de líquidos, movimiento de partículas.

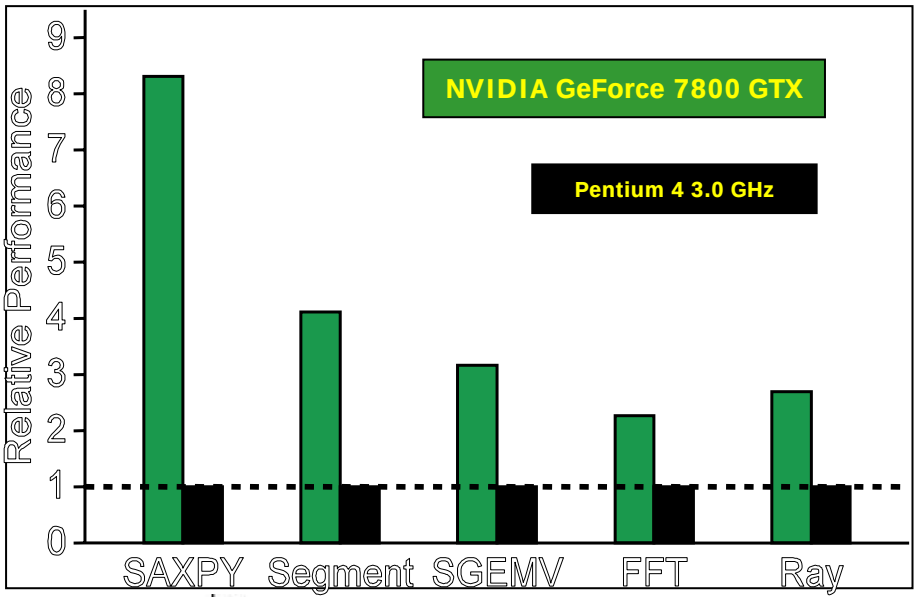
2.6 Rendimiento y coste

Dada la evolución y el rendimiento de las tarjetas gráficas tiene sentido aprovechar dicho recurso para otras aplicaciones a parte de los juegos. En la literatura se puede encontrar varios artículos [A.4, A.5, A.6, A.10, C.1, C.3] que evalúan el rendimiento de estas tarjetas y lo comparan con las CPU's.

En la siguiente figura podemos comparar los diferentes rendimientos que publican diferentes artículos, según el modelo de la tarjeta gráfica usada y el modo de codificar estos algoritmos:

- SAXPY que computa y := alpha * x + y siendo 'x' e 'y' vectores y 'alpha' un escalar,
- SGEMV que computa y = alpha * A * x + beta * y siendo 'X' e 'Y' vectores, 'alpha' y 'beta' escalares y 'A' una matriz,
- 'segment' que segmenta una imagen en zonas, por ejemplo en imágenes médicas detectar zonas de tejidos blandos o huesos, y en imágenes por satélite zonas de agricultura, pantanos, ríos, ciudades
- FFT una transformada rápida de Fourier que halla el espectro en frecuencias de una señal
- Ray-tracer, un trazador de rayos.
-





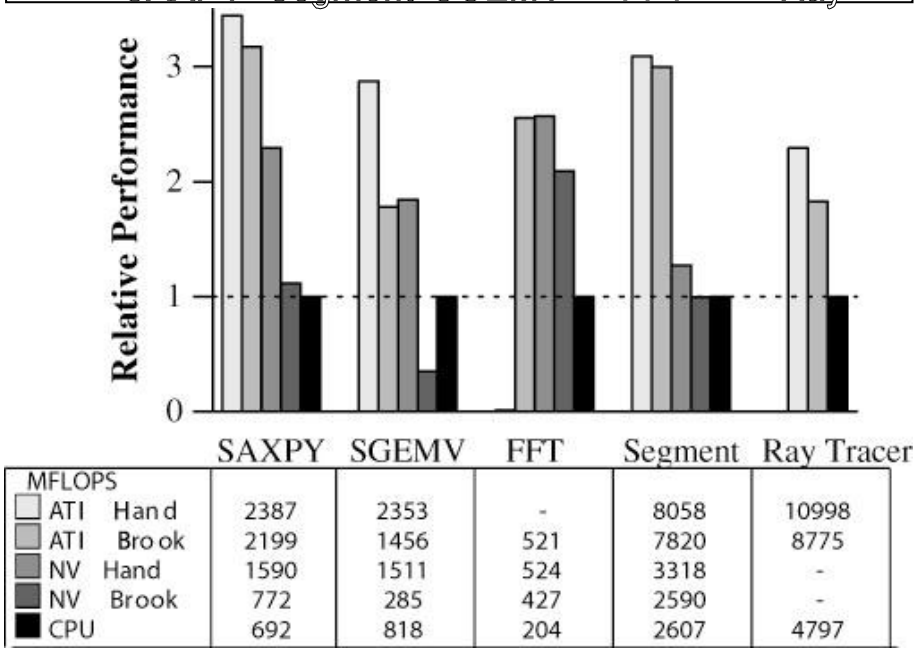


Figura 22: Diferentes comparativas de rendimiento según la publicación y la tarjeta usada: arriba izquierda, publicada por Ian Buck, en su presentación de Brook; arriba derecha, publicada por Mark Harris de NVIDIA; izquierda publicada por ATI.

Como podemos observar, dependiendo de la fuente, la fecha y la tarjeta usada varía el rendimiento entre un factor 3x y un factor 8,5x para la operación SAXPY. Incluso hay artículos que hablan de un factor 20x respecto de la CPU. Para la operación SGEMV el cálculo en la tarjeta es entre un 1,7x y un 3,3x más rápido que la CPU.

Si tenemos en cuenta el factor económico:

- el precio de una tarjeta NVIDIA GeForce 7800 GTX de 512MBytes se encuentra por 766 € (www.comprapc.com IVA incluido), una ATI Radeon X1900X 512MBytes por 561 € (www.comprapc.com IVA incluido), y si no necesitamos tanta potencia, una NVIDIA GeForce 6600 con 256 MB se puede encontrar por 129€ (www.comprapc.com IVA incluido)
- para conseguir un factor 4x, nos conformamos con este factor en vez del 8x que aparece en la gráfica de la Figura 22, una estación de trabajo SUN Ultra 40 con 2 procesadores AMD Optaron duales y 16 GBytes cuesta unos 9.428 € (www.spi.es IVA incluido).

Es decir que podemos disponer de un mini ‘super-ordenador’ ocupando una simple ranura dentro de un PC de sobremesa a un precio asequible.

También hay que tener en cuenta la versatilidad que nos ofrece el tener toda una estación de trabajo con cuatro procesadores, pero a nivel de cálculo estricto en precisión simple (veremos más adelante), al menos merece la pena estudiar el aprovechamiento de la capacidad de cálculo de las tarjetas gráficas, pues también las podemos usar en conjunción con la estación de trabajo.

Otra vía que se abre es la de poder utilizar cualquier ordenador de sobremesa, aun siendo algo viejo, para calcular simplemente insertándole una tarjeta gráfica.

Visto el rendimiento y el hecho de que ya hay algunas implementaciones de algoritmos generales vale la pena aprovechar el potencial de las tarjetas gráficas para hacer simulaciones y resolver sistemas de ecuaciones.

Una última nota, debido a este uso general de las tarjetas gráficas, fuera del mundo de los juegos, las GPU han pasado a denominarse GPGPU: General Purpose Graphical Processor Unit [A.6].

3. Definición del trabajo

3.1 Definición del objetivo y etapas

Después de ver que es posible usar las tarjetas gráficas para otros fines que no sean lúdicos y que merece la pena usar estas tarjetas como procesadoras de cálculo masivo pasaremos a definir un esquema de trabajo cuyo objetivo principal es proporcionar una herramienta para que se pueda usar este recurso sin tener que interactuar con la tarjeta. Es decir desarrollaremos una librería de manera que un programador, ya sea un ingeniero de caminos, aeronáutico, u otro, que esté realizando simulaciones numéricas pueda usar este recurso en su propio beneficio sin que se dé cuenta que usa una tarjeta gráfica.

Las etapas que seguiremos son:

Primero estudiaremos, por una parte, la arquitectura y limitaciones de las tarjetas gráficas que nos ayudará a definir una serie de conceptos como ‘streams’ y ‘kernels’; y por otra estudiaremos las diferentes herramientas disponibles que abstraen la arquitectura de la tarjeta, y valoraremos sus puntos fuertes y sus puntos débiles, siempre desde un punto de vista del ingeniero y que nos ayudarán a definir una serie de requerimientos de nuestra librería.

Segundo: implementaremos la función ‘saxpy’ (calcula Vector_Y = Vector_Y + alpha * Vector_X) que nos servirá como aprendizaje de la filosofía GPGPU, para definir los recursos que necesitará la librería y establecer una metodología de evaluación del rendimiento. Para realizar esta implementación también definiremos el entorno de trabajo y librerías a usar.

Tercero: desarrollaremos una librería:

- con operaciones básicas de vectores y matrices,
- que oculte toda interacción con la tarjeta al programador numérico final,
- fácil de usar y de mínimo coste al programador final,
- portable, que se pueda usar tanto en Microsoft Windows como Linux.

Hay que remarcar que parte del primer objetivo ya se ha cubierto en el punto ‘2.3 Arquitectura paralela’ del capítulo anterior que nos ha servido para valorar la ‘idoneidad’ del proyecto.

3.2 Herramientas

Para el desarrollo del proyecto disponemos de un ordenador PC AMD Athlon64 3000+ con 2GBytes de memoria y una tarjeta gráfica NVIDIA GeForce 6800GT AGP con 256 MBytes. Y para evaluar el rendimiento tenemos acceso a los siguientes ordenadores:

Procesador y memoria RAM	Tarjeta gráfica	bus
AMD Athlon 850, 512 MBytes RAM	NVIDIA GeFoce FX5200 256 MBytes	PCI
AMD Athlon XP 1600 512 MBytes RAM	NVIDIA GeForce FX5500 128 MBytes	AGP
AMD Athlon64 3000+ 1GBytes RAM	NVIDIA GeForce 6600 256 MBytes	PCI Express
AMD Athlon64 3200+ 1GBytes RAM	ATI Mobility Radeon 9600 64 MBytes	AGP
Intel Pentium 4 2,8 GHz 2 GBytes RAM	NVIDIA GeForce FX 5790 128 MBytes	AGP

La librería se desarrollará en C++ sobre OpenGL para asegurar la portabilidad entre Windows y Linux.

4. Estudio

4.1 Aprovechamiento de la arquitectura

Si recordamos el apartado 2.3 y la figura 15, la única forma de enviarle datos a la tarjeta gráfica es enviar vértices o enviar texturas. Y la única forma de operar con ellos es enviar a la tarjeta un programa que opere con los datos y 'pintar' estos datos. En este proceso de pintado se ejecuta nuestro programa sobre los datos, un esquema simplificado lo vemos en la siguiente figura:

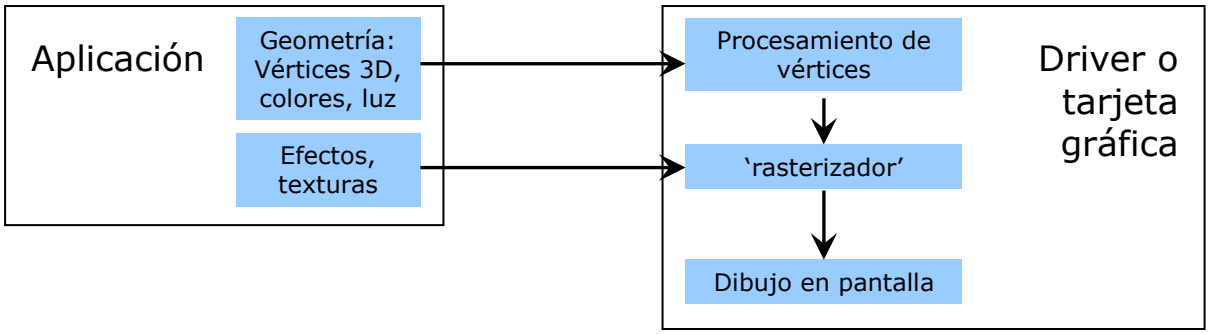


Figura 23: Esquema simplificado

Como podemos observar, lo que hace la tarjeta es recibir ráfagas de datos: listas de vértices y/o texturas y sobre ellos aplicar algún algoritmo.

A las ráfagas de datos se les conoce como 'streams' de datos y el programa que opera con ellos: 'kernels'. Este símil está basado en la arquitectura 'streams' en la que se aplica un algoritmo a todos los datos del 'stream' de entrada dejando los resultados en el 'stream' de salida.

Las GPU le añaden a este modelo, una arquitectura SIMD, donde se ejecuta una operación, en nuestro caso un algoritmo, sobre múltiples datos, diferentes ráfagas de datos independientes entre ellas. Es como una 'Connection Machine' [G.16] (máquina de paralelo masivo que constaba de 1024 procesadores con su memoria local) en la que la GPU además se encarga de dividir nuestro 'stream' de datos en varias partes a las que se les aplica nuestro algoritmo en paralelo y después reunifica en un único 'stream' de salida.

En la figura 24 vemos una representación de este modelo.

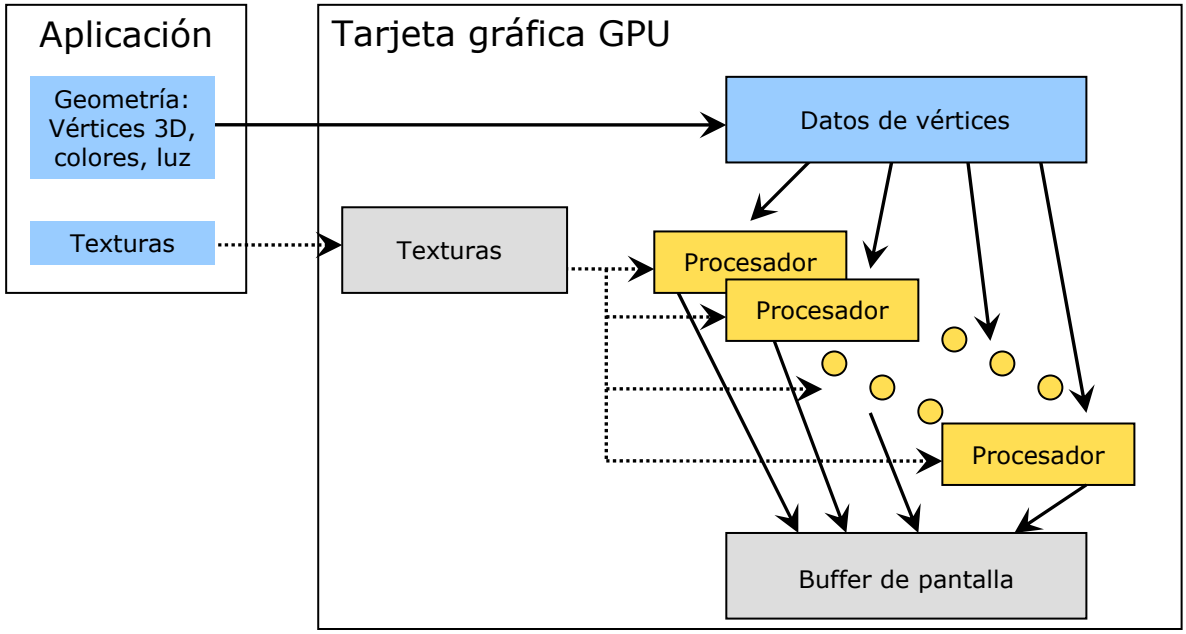


Figura 24: Esquema paralelo

Este tipo de arquitectura explota la localidad de los datos, con lo que la mayor velocidad de transferencia entre la memoria y la GPU juega a favor de las tarjetas gráficas, a pesar de contar con una pequeña, prácticamente inexistente, caché.

Otra característica de las GPU's es que el tipo básico ya no sólo es real o el entero, si no que además puede operar directamente con:

- vectores de 2, 3 y 4 elementos, y con
- matrices de 2x3, 3x3 y 4x4 elementos

Es decir que con una operación puede ejecutar una multiplicación matriz4x4 * vector4.

Así pues, si queremos operar con la tarjeta simplemente tenemos que seguir estos pasos (figura 25):

- Empaquetar nuestros datos, matrices y vectores, en texturas
- Reprogramar nuestro algoritmo en algún código que sea entendible para la tarjeta
- Enviar las texturas y el programa a la tarjeta
- Decirle a la tarjeta que calcule, que dibuje en este caso, y que el resultado nos lo deje en otra textura
- Recoger esta textura y traerla a nuestra aplicación
- Y, finalmente, desempaquetar nuestro resultado de la textura.

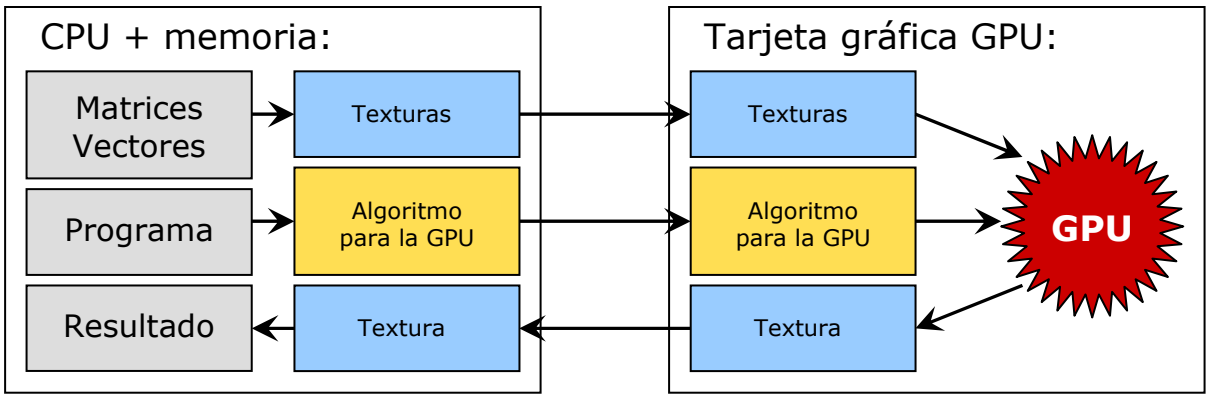


Figura 25: Proceso de cálculo en la tarjeta

Como vemos hay bastante sobrecoste a la operación de calcular: empaquetar los datos, enviarlos a la tarjeta, crear y enviar el programa, recoger el resultado y desempaquetarlo. ¿Bajo qué condiciones es rentable todo este sobrecoste y no nos echa a perder toda la mejora de rendimiento que hemos conseguido al calcular en la GPU?
También hay que contar con la inicialización del contexto gráfico, así como de los recursos que necesitaremos de la tarjeta.

Hemos comentamos en la sección 2.3 que no podemos la misma textura para leer y para escribir, es decir que si queremos realizar una operación de acumulación, por ejemplo a += b, hemos de usar tres 'streams': a_destino = a_original + b_original, y que no podremos acceder al resultado hasta que haya acabado todo el bucle de 'calculo', pintado.

Otro de los condicionantes es que tenemos que escoger el formato adecuado de la textura, en nuestro caso algún formato que soporte números reales, para que el algoritmo que enviemos pueda operar con ellos como reales. Y el real más preciso que soportan los procesadores de vértices y fragmentos son los reales de 32 bits ('float' en 'C/C++' o 'real*4' en 'Fortran') a partir de la GPU NVIDIA GeForceFX 5200 y la ATI Radeon X1300 (las ATI Radeon anteriores a esta serie sólo soportaban reales de 24 bits, a todas luces insuficiente para la simulación, pero suficiente para aplicaciones gráficas) [G.11, G.2, G.9].

Éste limite de 4 octetos, 'bytes', para representar un real es una limitación si queremos hacer algunas de las simulaciones de fluidos, u otras que requieran cálculos realmente complejos, pero para buena parte de los análisis, como por ejemplo los estructurales, los análisis de sólidos o algunos métodos de partículas sí que es suficiente.

Sólo mencionar que Dominik Göddeke, en su artículo 'Accelerating double precision FEM simulations with GPUs' [A.8] presenta un método para conseguir doble precisión, la que se consigue con reales de 64 bits ('doubles' en 'C' y 'real*8' en Fortran). Göddeke usa un algoritmo de aproximación por defecto de precisión mixta, en la que primero ejecuta un precondicionador en la GPU con precisión simple y después lo acaba de corregir usando precisión doble en la CPU. Aún pareciendo más costoso consigue un factor 2.3x respecto a ejecutar el código enteramente en la CPU.

Podemos observar que un programador de simulaciones numéricas ha de tener en cuenta muchos detalles y readaptar mucho su código para poder aprovechar el rendimiento de las tarjetas. Esfuerzo que requerirá mucho tiempo si no ha programado ni una aplicación gráfica, como es la mayoría de los casos.

Actualmente existen herramientas que ocultan toda la interacción con la tarjeta gráfica y la abstraen a un modelo de streams y kernels.

4.2 Herramientas disponibles

Las herramientas para programar las tarjetas gráficas las podemos dividir en tres clases: los lenguajes de 'alto nivel', que esconden toda interacción con la tarjeta al programador de simulaciones numéricas; las de 'medio nivel' con los que se puede crear el algoritmo a ejecutar en la tarjeta en alto nivel, parecido a 'C/C++', pero que hay que inicializar el contexto gráfico, compilar y enviar el algoritmo a la tarjeta y después calcular 'pintando'; o los de 'bajo nivel', que además de interactuar con la tarjeta se programaría el algoritmo en código máquina. En la figura 26 podemos ver esta clasificación:

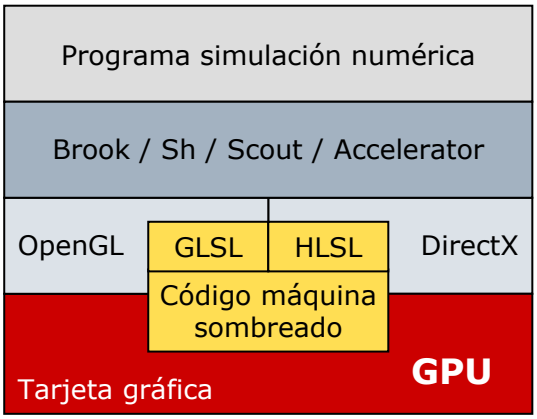


Figura 26: Capas de software entre la aplicación y la tarjeta gráfica

Brook

Dentro de los lenguajes de 'alto nivel' se encuentra BrookGPU desarrollado por Ian Buck [C.1, C.3, B.2] en la Universidad de Stanford. Se trata de un compilador que le añade extensiones al lenguaje 'C' para habilitar la GPU como coprocesador de streams y un 'runtime' para escoger la plataforma final de ejecución. Brook se desarrolló como lenguaje para procesadores de 'streams' como el Merrimac de Stanford, el procesador Imagine de Kapasi entre otros. Y simplemente se extendió para soportar el nuevo hardware gráfico.

Las abstracciones que el lenguaje provee se encuentran:

- la memoria se maneja como 'streams': listas de datos que pueden tener varias dimensiones. Son parecidos a los 'arrays' de 'C' pero sólo se puede acceder sus datos:
    - mediante estas dos funciones especificas:
        streamRead: para pasar los datos de la aplicación al 'stream'
        streamWrite: para pasar los datos del 'stream' a la aplicación
    - mediante las funciones 'kernel'.

- las operaciones paralelas, es decir que operan sobre los 'streams' son funciones especiales, designadas con la palabra clave 'kernel', que realizan un bucle implícito sobre los elementos del 'stream' ejecutando el cuerpo de la función para cada uno de ellos. Hay diferentes tipos de argumentos: 'streams' de entrada y de salida, designados con las palabras clave 'in y 'out' y 'streams' recogidos ('gather'), designados como los arrays en 'C' 'array[]', para recuperar elementos vía el operador de indexación '[]', y todo argumento que no sean 'streams' son constantes de sólo lectura.

- operaciones de reducción, tipo 'muchos a uno', es un método paralelo para calcular un valor simple de una lista de datos. Por ejemplo, sumas aritméticas, búsquedas de máximo o mínimo de los elementos de un vector, etc. Se definen como funciones con la palabra clave 'reduction', especificando el argumento destino de la reducción con la misma palabra.

También dispone de otras peculiaridades como por ejemplo: 'indexof' para obtener el índice de un elemento de un 'stream' dentro de una función 'kernel', tablas de valores secuenciales precalculados, etc.

En la figura 27 tenemos un ejemplo de un programa escrito en Brook que simplemente actualiza un vector de posiciones en función de la velocidad, el tiempo y la posición inicial.

```
kernel void ActualizaPosicion( float PosIni<>, float f, float Vel<>, float t<>,
                               out float PosFinal<>) {
  PosFinal = PosIni + f * Vel * t;
}

void main( void) {
  float PosIni[ 1000];
  float Vel[ 1000];
  float t[ 1000];
  float PosFinal[ 1000];

  float f = 0.5;

  float B_PosIni< 1000>;
  float B_Vel< 1000>;
  float B_t< 1000>;
  float B_PosFinal< 1000>;

  streamRead( B_PosIni, PosIni);
  streamRead( B_Vel, Vel);
  streamRead( B_t, t);
  ActualizaPosicion( B_PosIni, f, B_Vel, B_t, B_PosFinal);
  streamWrite( B_PosFinal, PosFinal);
}
```

```
void main( void) {
  float PosIni[ 1000];
  float Vel[ 1000];
  float t[ 1000];
  float PosFinal[ 1000];
  float f = 0.5;

  for ( int i = 0; i < 1000; i++)
    PosFinal[ i] = PosIni[ i] + f * Vel[ i] * t[ i];
}
```

Figura 27: Ejemplo de un programa escrito con Brook y su equivalente en 'C'.
La única forma de pasar datos a un 'stream' o recogerlos es
con 'streamRead' o 'streamWrite' respectivamente

Una vez hayamos escrito nuestro código ('hola.br' en la figura 28) lo compilamos con el compilador de Brook, 'brcc', y obtenemos, por un lado el código 'Cg' (lenguaje de sombreado de alto nivel, que más adelante explicaremos) de nuestras funciones 'kernel' de Brook, y por otro código 'C++' ('hola.cg' y 'hola.cc' respectivamente, en la figura 28) que usa la librería 'runtime', 'brt', para llamar a los 'kernels', ejecutarlos en la plataforma elegida, la tarjeta o el procesador.

La librería 'runtime' de Brook es una capa de software independiente de la arquitectura que provee un interfaz común para las plataformas finales soportadas por el compilador. Entre las plataformas finales de ejecución soportadas se encuentran 'OpenGL' ('OGL' en la Figura 28), 'DirectX' ('DX9' en la figura 28) y la implementación en CPU ('x86' en la figura 28).

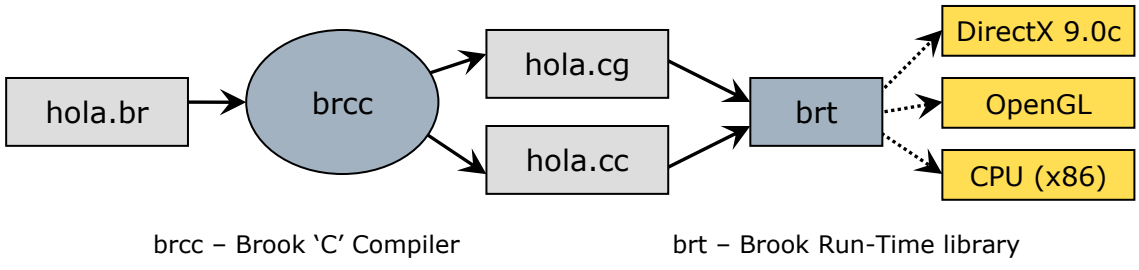


Figura 28: Esquema de funcionamiento de Brook.

Según Ian Buck, esta generación automática del código de sombreado a partir del código escrito en Brook, consigue un rendimiento, en el caso de la función saxpy (y = y + a * x) de hasta el 80% respecto a escribir el algoritmo manualmente para la tarjeta gráfica.

La última versión disponible de Brook es la 0.4 del 15 de Octubre del 2004. Desde hace un año y medio no ha habido versión nueva alguna. En una herramienta que explota las últimas

tecnologías, o está muy bien hecha, o no se mantiene. Por ejemplo, la extensión 'GL_EXT_framebuffer_object' incorporada a OpenGL 2.0 (Octubre 2004) [E.2, E.3], que acelera la creación del contexto para pintar en una textura, su dibujado y su recuperación (se verá más adelante), no se usa dentro de Brook. Cabe otra posibilidad, y es que se desarrolle a puertas cerradas para su posterior comercialización.

Desde el punto de vista del programador de simulaciones numéricas se presentan varios inconvenientes, a parte de bajarse el compilador y la librería, compilarla e instalarla:

- no se pueden indexar los elementos del 'stream' en el resto de la aplicación, es decir, fuera de las funciones 'kernel' y 'reduce', por ejemplo, para evaluar el error de una solución;
- ha de aprender un nuevo lenguaje de programación y sin un entorno IDE, como el Microsoft Visual Studio o el Kdeveloper;
- obliga al programador a distinguir entre 'streams' de entrada en una función 'kernel', que se acceden secuencialmente, y los 'streams' recogidos ('gathered') que pueden accederse aleatoriamente, cuando los dos tipos se almacenan de la misma manera en la tarjeta.

Sh

Otro de los lenguajes de 'alto nivel' se encuentra Sh desarrollado por la Universidad de Waterloo [C.4, B.2]. Se trata de una librería C++ que proporciona un lenguaje de 'meta-programación' dentro de C++. También consigue ocultar la arquitectura de la tarjeta gráfica para el programador final.

Al estar implementado en C++ y proporcionar este 'meta-lenguaje' dentro de C++ aprovecha toda su potencia: comprobación de tipos, estructuración en clases con sus constructores, funciones, y demás características de los programas orientados a objetos. Este hecho también supone una facilidad al programador habitual en este lenguaje, pues la sintaxis de Sh es muy similar a la de C++.

En la figura 29 vemos la clasificación por niveles de la funcionalidad dentro de Sh.

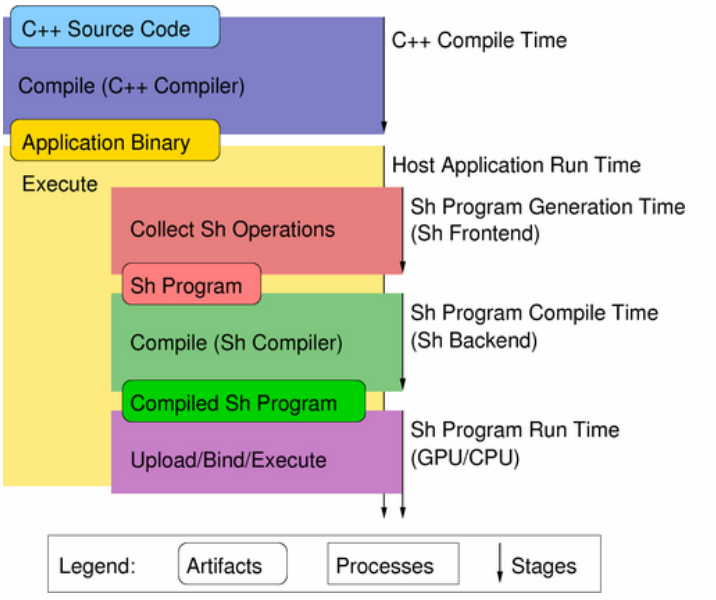


Figura 29: Esquema de funcionamiento de Sh

El programa se puede compilar con un compilador de C++ estándar. Cuando el binario final se ejecuta, Sh recopila las instrucciones a evaluar y las convierte a una representación

interna. Este código se transforma a uno que pueda entender la GPU particular sobre la que estemos ejecutando y se compila para la plataforma final elegida, como en el caso de Brook, ya sea OpenGL, DirectX, la CPU o 'SM simulator' (también dispone de una herramienta de simulación de sombreado). Este código ensamblador se puede ejecutar o guardar aparte para poderlo utilizar en otro programa.

Otra ventaja de Sh es que dispone de dos modos, el modo inmediato, 'inmediate', en el que las operaciones se computan al momento, si se suman dos vectores, se calcula el resultado de esa suma. Este es el modo por defecto. El segundo modo, es el retenido, 'retained', que se activa con un flan en tiempo de ejecución. En este modo, las operaciones, dentro de la zona 'retenida' se van acumulando y cuando se sale de esa zona 'retenida', se compila y se optimiza todas las operaciones recogidas en conjunto. Este modo permite, por ejemplo, usar variables temporales dentro del espacio de Sh para acumular resultados parciales y que, una vez obtenido el resultado final, se puede despreciar, y sin que pasen por el espacio de aplicación.

Aquí tenemos un ejemplo del programa escrito anteriormente en Brook, pero esta vez en Sh:

```
void ActualizaDatos( ShChannel< ShAttrib1f> &ST, float *datos) {
  ShHostStoragePtr ST_storage = shref_dynamic_cast<
                                ShHostStorage>( ST.memory()->findStorage( "host"));
  ST_storage->dirty(); // para marcar que los datos actuales están en el host
  float *ST_datos = ( float *)ST_storage->data();
  for ( int i = 0; i < 1000; i++) {
    // inicializamos los vectores
    ST_datos[ i] = datos[ i];
  }
  ST_storage->sync(); // para que sincronice los datos, los baje a la tarjeta
}

float *GetDatos( ShChannel< ShAttrib1f> &ST) {
  ShHostStoragePtr ST_storage = shref_dynamic_cast< ShHostStorage>(
                                  ST.memory()->findStorage( "host"));
  ST_storage->sync(); // para que sincronice los datos, los suba de la tarjeta
  return ( float *)ST_storage->data();
}

void main() {
  float PosIni[ 1000], Vel[ 1000], t[ 1000], PosFinal[ 1000];
  ShAtrrib1f f = 0.5;

  ShProgram ActualizaPosicion = SH_BEGIN_PROGRAM() {
    ShInputAttrib1f PosIni, Vel, t;
    ShOutputAttrib1f PosFinal;
    PosFinal = PosIni + f * Vel * t;
  }
  ShCompile( ActualizaPosicion);

  ShChannel< ShAttrib1f> S_PosIni( 1000);
  ShChannel< ShAttrib1f> S_Vel( 1000);
  ShChannel< ShAttrib1f> S_t( 1000);
  ShChannel< ShAttrib1f> S_PosFinal( 1000);

  ActualizaDatos( S_PosIni, PosIni);
  ActualizaDatos( S_Vel, Vel);
  ActualizaDatos( S_t, t);

  S_PosFinal = ActualizaPosicion << ( S_PosIni & S_Vel & S_t & S_PosFinal);
  // otra forma sería usando ShStream, que referencia a los canales, no los copia
  // ShStream datos_in = ( S_PosIni & S_Vel & S_t & S_PosFinal);
  // S_PosFinal = ActualizaPosicion << datos_in;

  float *ret = GetDatos( S_PosFinal);
  for ( int i = 0; i < 1000; i++)
    PosFinal[ i] = ret[ i];
}
```

```
void main( void) {
  float PosIni[ 1000], Vel[ 1000];
  float t[ 1000], PosFinal[ 1000];
  float f = 0.5;

  for ( int i = 0; i < 1000; i++)
    PosFinal[ i] = PosIni[ i] +
      f * Vel[ i] * t[ i];
}
```

Figura 30: Ejemplo de un programa escrito con Sh y su equivalente en 'C'.

Los vectores declarados al inicio del programa, (PosIni, Vel, t y PosFinal)
son a título indicativo y contienen los valores iniciales de los
vectores que podían leerse de un archivo.

Como puede comprobarse la forma de acceder a los datos de los canales, 'streams', es un tanto complicada (también puede ser que no hayamos encontrado una forma más fácil) aunque intenta seguir el paradigma de los 'streams' al pie de la letra, usando incluso el operador '<<' de C++.

Comparado con Brook, esta vez sí se puede acceder a los elementos por separado y no hace falta tener explícitamente dos copias de los datos en la aplicación. En Brook teníamos la copia local y la de la estructura Brook, aunque bien es cierto que se aloja en la tarjeta. En cambio, en Sh, la copia local está enmascarada dentro de la estructura 'ShChannel', aunque es a cargo del programador el mantener las copias sincronizadas.
Otro inconveniente es que los nuevos tipos de datos son algo crípticos, por ejemplo ShAttrib1f y ShPoint3f, y en ellos se ve reflejado el nacimiento de Sh como lenguaje de sombreado, pues la sintaxis parece basarse en OpenGL, donde teníamos, por ejemplo, GLfloat, gl_Position.

Desde el punto de vista del programador de simulaciones numéricas se puede apreciar una separación entre el código que se ejecuta en la CPU y en Sh, que puede acabar ejecutándose en la CPU, obligándole, además, a mantener una sincronización explícita entre los datos de la aplicación y la librería.

Scout

Un lenguaje de 'alto nivel' interesante pues combina dos tareas que pueden parecer claramente separadas: el análisis de datos y la representación gráfica.

Desarrollado por Patrick McCormick en los laboratorios de Los Alamos, Scout [C.6] combina dos tareas que normalmente se solventan separadas: extraer información de un espacio de datos mediante consultas basadas en expresiones y la representación de esta información en pantalla. Para ello desarrolla un nuevo lenguaje basado en 'C*', parecido a 'C' con extensiones para computación paralela y que se usaba para programar la Connection Machine de Thinking Machines Corporation [G.16]. Con este lenguaje, el programador puede realizar cualquiera de estas dos tareas sin darse cuenta de que la computación final se hace en la GPU.

Seguidamente se pueden apreciar la filosofía de Scout en dos ejemplos: en el primero se dibuja con colores del arco iris (azul para frío y rojo para caliente) el valor de las temperaturas en el océano y la superficie terrestre negra. En el segundo ejemplo, primero calculamos el valor de la entropía y el módulo de velocidad de un gas que después se usarán para pintarlos.

```
// Ejemplo 1:

render with( shapeof( pt)) {
  // tierra y pt han de tener la misma forma ( shape)
  where ( tiera) // no colorear los continentes
    image = 0;
  else
    image = hsva( 240 - norm( pt) * 240, 1.0, 1.0, 1.0);
}
```

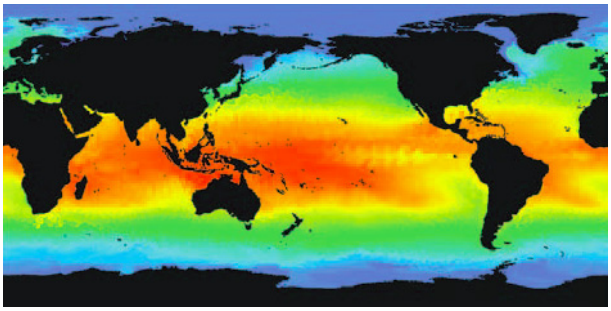
Paso de pintado

```
// Ejemplo 2:

// calcular la entropia y el módulo de la velocidad

float:shapeof( presion) entropia;
float:shapeof( presion) vmod; // módulo de velocidad

compute with( shapeof( presion)) {
  entropia = presion / pow( densidad, 4.0 / 3.0);
  vmod     = sqrt( dot3( velocidad, velocidad);
}
```

Paso de cálculo

```
// calcular las normales del gradiente para el sombreado
volren with( shapeof( entropia)) {
  // seleccionar la region interna de la entropia y
  // recortar a lo largo del eje X.
  where ( ( i > 115) && ( entropia > 0.07) && ( entropia < 0.076)) {
    image = hsva( 240 - norm( vmod) * 240.0, 1.0, diffuse, 1.0);
  } else where( ( entropia > 0.01) && ( entropia < 0.04)) {
    // esta es la onda de choque
    image = hsva( 240 - norm( vmod) * 240.0, 1.0, 1.0, 0.1);
  } else {
    image = 0; // negro
  }
}
```

Paso de pintado



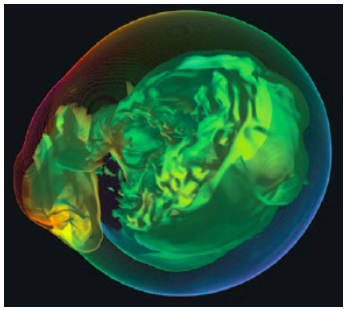


Figura 31: Dos ejemplos de programas escritos con Scout y su salida en pantalla.
En el ejemplo 1, simplemente pintamos las temperaturas en el mundo allí donde no es tierra.
En el ejemplo 2, calculamos primero la entropía y el módulo de la velocidad y
después pintamos este módulo según un rango de entropías.

Otra de las ventajas de Scout es que se proporciona un entorno de desarrollo de código (GUI/IDE) y una ventana de renderización, además del compilador de línea de comandos. La parte de representación gráfica la realiza en OpenGL, pero escondiéndolo al programador de simulaciones numéricas, con lo que no hace falta sepa cómo usar una librería gráfica. También es multiplataforma, Linux, Windows y MacOS X, y soporta varios tipos de GPU.

Una vez hayamos escrito nuestro código en C* (‘hola.src’ en la Figura 32) lo compilamos con el compilador de Scout, ‘scc’, y obtenemos código ‘C++’ (‘hola.cc’ en la Figura 32) que se puede compilar con el compilador de ‘C++’.

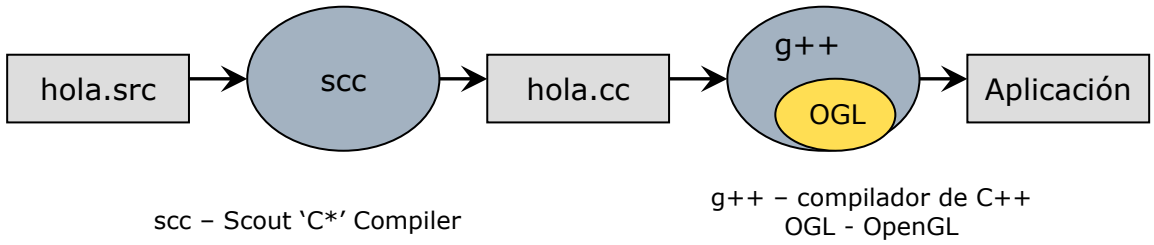


Figura 32: Esquema de funcionamiento de Scout.

Desde el punto de vista del programador de simulaciones numéricas, ha de aprender una nueva sintaxis, además de usar otra herramienta que la habitual para desarrollar. El problema más importante es que su salida estaba prevista para Octubre del 2005, y aún no se encuentra disponible.

Accelerator

Dentro del departamento de investigación de Microsoft se está desarrollando una librería que proporciona un modelo de programación de alto nivel para datos paralelos: Accelerator [C.2]. Traduce operaciones de datos paralelos 'al vuelo' a código optimizado para la GPU. Según el artículo publicado en Diciembre de 2005, esta automatización consigue un rendimiento de aproximadamente un 50% respecto a los mismos algoritmos codificados 'a mano'. Incluso algunas pruebas de rendimiento, superan en un factor de 18 la velocidad del mismo algoritmo en 'C'. En principio, esta librería esta pensada para poderse usar con cualquier lenguaje.

El tipo de dato básico en Accelerator en el array paralelo (class ParallelArray; class FloatParallelArray;) y una de sus limitaciones es que no se puede indexar. En la Figura 33 tenemos un ejemplo de Accelerator, que le aplica una convolución a una imagen. Los parámetros que se le pasan a la función, un array 2D con la imagen y otro unidimensional con los pesos, los convierte a parallelArray y calcula los valores ponderados de la imagen final, primero en la dirección x y después en la y usando la sobrecarga del operador '*'.

```
using Microsoft.Research.DataParallelArrays;

float[,] Blur( float[,] array, float[] kernel) {
  using ( DFPA parallelArray = new DFPA( array)) {
    FPA resultX = new FPA( 0.0f, parallelArray.Shape);
    for ( int i = 0; i < lernel.Length; i++) {
      retulrX += parallelArray.Shift( 0, i) * kernel[ i];
    }
    FPA resultY = new FPA( 0.0f, parallelArray.Shape);
    for ( int i = 0; i < lernel.Length; i++) {
      retulrX += resultX.Shift( i, 0) * kernel[ i];
    }
    using ( DFPA result = resultY.Eval()) {
      return result.ToArray() as float[,];
    }
  }
}
```

Figura 33: Ejemplo de un programa escrito con Accelerator y C#

La única plataforma que soporta Accelerator es DirectX 9 y necesita de Microsoft Visual Studio 2003 o 2005 y el entorno de desarrollo .NET 1.1.
Esta librería se puede encontrar esta librería en la web de Microsoft desde abril del 2006.

Aquí también se ve diferenciado el código que se evalúa con Accelerator y el de la aplicación aunque la librería usa la sobrecarga de operadores. Otro punto es que al programador se le obliga a tener dos copias de los datos en memoria, la de la aplicación y la que se usará con Accelerator y a sincronizarlas. Tampoco se pueden acceder a los elementos del parallelArray por separado.

Lenguajes de sombreado

Dentro de esta categoría se hallan los lenguajes de 'alto nivel' que se utilizan para programar la tarjeta gráfica:

- GLSL que se usa con OpenGL a partir de la versión 1.5, aunque el soporte para bucles y saltos condicionales es a partir de la 2.0 [D.1].
- HLSL que se usa con DirectX 9 de Microsoft [D.3].
- Cg de NVIDIA, previo a los dos anteriores y a los que les sirve de base [D.2]. Se puede usar con cualquiera de las dos librerías: DirectX o OpenGL.

En GLSL y HLSL, el código de sombreado es una cadena de texto, que la aplicación tiene dentro de su código o que lee de un archivo, lo envía a la librería gráfica para que lo compile y lo enlace. Después se dibuja la escena y se le dice a la librería que use el código, ya compilado. Todo esto en tiempo de ejecución.

Cg usa un mecanismo similar. Cg da nombre a un lenguaje y a una librería que lo compila. Esta librería se ha de enlazar, junto a la librería gráfica deseada, con la aplicación. Dentro de la aplicación, el código Cg se envía a su 'propia' librería para que se compile y se enlaza. Una vez obtenido el código compilado sigue el mismo proceso que antes, se dibuja la escena, usando la librería gráfica que queramos, y se le envía el código compilado a la tarjeta con la librería Cg. También todo esto en tiempo de ejecución.

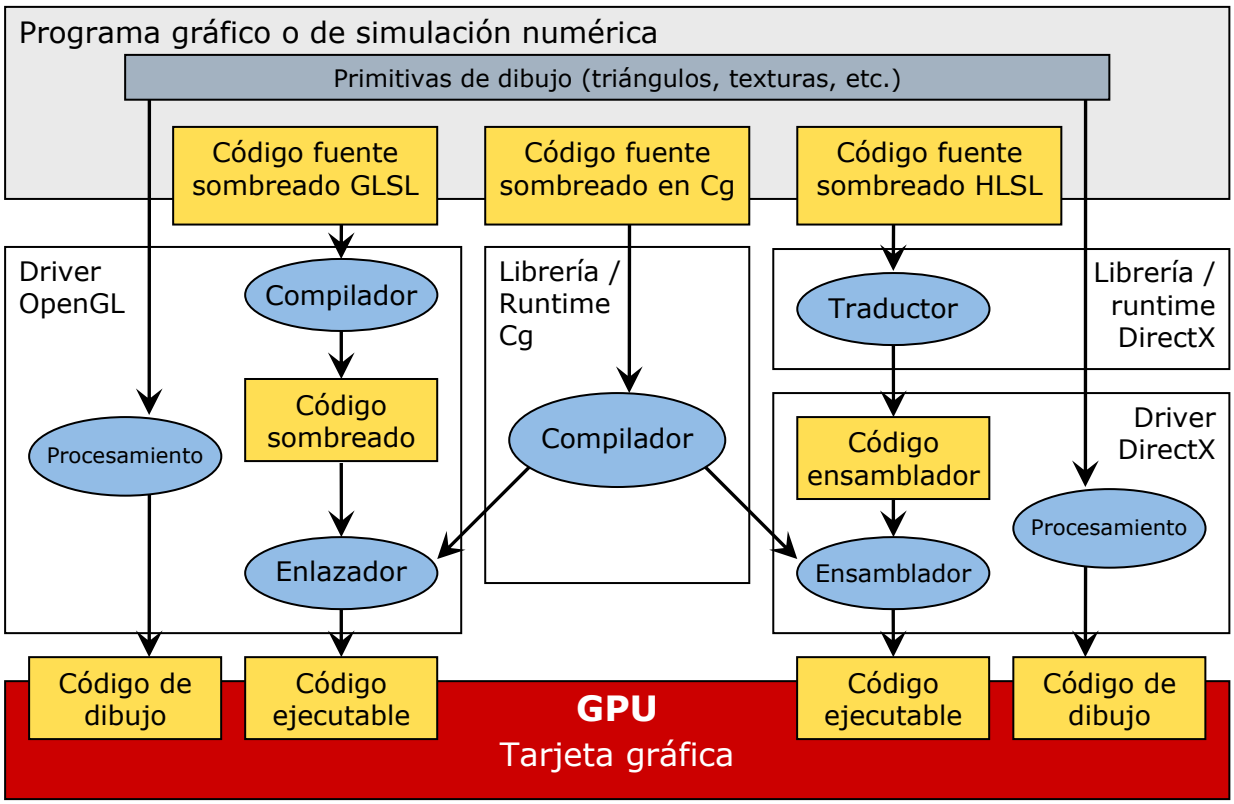


Figura 34: Esquema de funcionamiento de GLSL, Cg y HLSL

En la figura 34 vemos un esquema del funcionamiento interno de estos tres lenguajes.

Los tres lenguajes comparten características similares, como podemos observar en la figura 35:

- mismos tipos de datos, aunque se llamen diferente: vec4, mat3 en OpenGl, float4, float3x3 en Cg y HLSL definen un vector de 4 reales y una matriz de 3 por 3 reales.
- tienen un objetivo común: posibilitar al programador el uso de todas las posibilidades de los procesadores de vértices y fragmentos.

GLSL

```
uniform sampler2D VectorY;
uniform sampler2D VectorX;
uniform float alpha;

void main( void) {
  float2 idx = gl_TexCoord[0];
  vec4 y = texture2D( VectorY, idx.st);
  vec4 x = texture2D( VectorX, idx.st);
  gl_FragColor = y + alpha * x;
}
```

Cg

```
float4 saxpy ( in float2 idx: TEXCOORD0,
               uniform sampler2D VectorY,
               uniform sampler2D VectorX,
               uniform float alpha ) :
COLOR {
  float4 y = tex2D( VectorY, idx);
  float4 x = tex2D( VectorX, idx);
  return y + alpha * x;
}
```

HLSL

```
sampler2D VectorY;
sampler2D VectorX;
float alpha;

void main( in float2 idx: TEXCOORD0,
           out reultado: COLOR0) {
  float4 y = tex2D( VectorY, idx);
  float4 x = tex2D( VectorX, idx);
  resultado = y + alpha * x;
}
```

C

```
void main( void) {
  float VectorX[ 1000];
  float VectorY[ 1000];
  float resultado[ 1000];
  float alpha = 0.5;

  for ( int i = 0; i < 1000; i++)
    resultado[ i] = VectorY[ i] +
                    alpha * VectorX[ i];
}
```

Figura 35: Ejemplo de la función saxpy, Z = Y + a * X, escrito en GLSL, Cg, HLSL y en C.

Si recordamos la figura 25, un programador de simulación numérica se ha de pelear con la inicialización de la tarjeta, la verificación de los recursos disponibles, así como de empaquetar sus datos en texturas y reprogramar su algoritmo en alguno de los tres lenguajes (figura 35). Después lo tiene que enviar a la tarjeta, pintar un simple cuadrado para que calcule, recoger la textura destino y desempaquetar el resultado. Como puede verse, es un proceso laborioso, complicado y lleno de detalles sobre todo para un programador profano en el mundo de los gráficos.

Código de sombreado

Este es el código que resulta de compilar los lenguajes anteriores y que se pueden utilizar directamente en la tarjeta, parecido al código máquina. Según la librería gráfica que se esté usando, OpenGL o DirectX, este código varía un poco pero existe la utilidad Babelshader [C.1, B.2], de la universidad de Stanford, que convierte este código de una librería a la otra, como se muestra en la figura 36. Permite una comparación más 'justa' entre los diferentes compiladores en la misma plataforma, con la misma demo y el mismo driver.

```
texld r9, t5, s4 ; fetch
dp2add r0.a, r0, r0,-c0.z
rsq r0.a, r0.a
rcp r0.b, r0.a
mad r3, r8, c0.w, -c0.z
mad r6, r3, c4.r, r0
mad r3, r9, c0.w, -c0.z
mad r7, r3, c4.g, r0
mad r1.a, r5.r, c3.x,c3.y
dp3 r4.a, t4, t4
rsq r4.a, r4.a ;1/view
mul r4, t4, r4.a ;norm
; reflection vector
mul r2.rgb, r0.x, t1
mad r2.rgb, r0.y, t2, r2
mad r2.rgb, r0.z, t3, r2 ;
 transform bump map normal
```

```
TEX r9, t5, texture[4], 2D;
MAD R0.w, R0.x, R0.x, -c0.z;
MAD R0.a, R0.y, R0.y, R0.w;
RSQ r0.a, r0.a;
RCP r0.b, r0.a;
MAD r3, r8, c0.w, -c0.z;
MAD r6, r3, c4.r, r0;
MAD r3, r9, c0.w, -c0.z;
MAD r7, r3, c4.g, r0;
MAD r1.a, r5.r, c3.x, c3.y;
DP3 r4.a, t4, t4;
RSQ r4.a, r4.a;
; 1/ view
MUL r4, t4, r4.a;
MUL r2.rgb, r0.x, t1;
MAD r2.rgb, r0.y, t2, r2;
MAD r2.rgb, r0.z, t3, r2;
```

Figura 36: Ejemplo de conversión entre HLSL (PS 2.0) y GLSL (ARB)

Comprensiblemente esto no facilita la tarea del programador de simulaciones numéricas.

4.3 Requerimientos de la librería

Después de comparar las diferentes alternativas, y aprender de ellas, hemos llegado a la siguiente conclusión: un programador de simulaciones numéricas (ingeniero de caminos, naval, aeronáutico, etc.), y por tanto profano en el mundo de los gráficos ha de:

- aprender una sintaxis ampliada en el mejor de los casos o un nuevo lenguaje;
- tener en mente la nueva filosofía de programación, la de los 'streams' y resignarse a no poder acceder a elementos sueltos de los 'streams' directamente, por ejemplo para valorar el factor de daño en la carga de una estructura;
- ha de mantener dos copias de los datos en la aplicación y explicitar el intercambio entre los datos de la aplicación y de la librería o lenguaje, además de mantenerlos sincronizados;
- reprogramar los algoritmos en la mayoría de los casos
- en algunos casos ha de interactuar con la librería gráfica
- en otros tendrá que abandonar su entorno IDE/GUI preferido
- algunos de los lenguajes no pueden utilizarse en Linux o MacOSX

Por ello, requeriremos a nuestra librería:

- que esconda toda terminología relativa a gráficos: librerías, drivers, definición del entorno necesario y recursos;
- que sea posible mantener, de cara al programador final, una sola copia de los datos y que se sincronice automáticamente con los datos de la tarjeta cuando sea necesario;
- que opera con la sintaxis C++ estándar, si es posible, y, si no, que al menos se suponga el mínimo trauma posible;
- si no se encuentran los recursos necesarios, no hay tarjeta disponible o lo pide explícitamente el programador, se ejecute en CPU;
- que sea portable: Windows y Linux.

Ejemplos que queremos resolver serían como los mostrados en las figuras 37 y 38:

```
float a[ 1000], b[ 1000], c[ 1000];

c = a + b;
printf( “Valor: %g\n”, c[ 45]);
```

```
float a[ 1000], b[ 1000], c[ 1000];

for( int i = 0; i < 1000; i++)
  c[ i] = a[ i] + b[ i];
printf( “Valor: %g\n”, c[ 45]);
```

Figura 37: Ejemplo de uso de la librería, izquierda, y su equivalente en C, derecha.

```
void Solve( Matrix &A, Vector &B, Vector &X) {
  mBNorm = B.mod();
  if (mBNorm == 0.00) return;

  Vector r( B), p( B), q( B.size());
  double roh0 = r * r, roh1 = roh0, beta = 0.00;
  double pq, alpha;

  do {
    q = A * p;
    pq = p * q;
    if ( pq <= epsilon /*0.0*/) break;

    alpha = roh0 / pq;
    X =  alpha * p + X;
    X = -alpha * q + r;
    roh1 = r * r;
    beta = ( roh1 / roh0);
    p = r + beta * p;
    roh0 = roh1;
    mResidualNorm = sqrt( roh1);
    mIterationNumber++;
  } while ( ( mIterationNumber < mMaxIterations) &&
            ( mResidualNorm > mTolerance*mBNorm) && ( roh0 != 0.00) );
}
```

Figura 38: Algoritmo para resolver b = A * x usando gradientes conjugados y usado en Emant/Kratos.

Quizá el algoritmo de gradientes conjugados [A.18] quede fuera del trabajo presente pero sirve como objetivo al cual se pretende llegar. Al menos se implementará la función Vector = Matriz * Vector.

Las funciones que se implementarán en esta librería serán, y por este orden:

- operaciones entre vectores: suma, resta y multiplicación por componentes
- operaciones entre reales y vectores
- multiplicación matriz-vector

Con estas operaciones básicas definidas sobrecargando los operadores de C++ ‘+’, ‘-‘ y ‘*’ se podrán construir expresiones como esta:

```
GVector x( 1000), y( 1000), z( 1000), r( 1000); // vectores de floats
GMatrix A( 1000, 1000), B( 1000, 1000); // matrices cuadradas de floats
float c, d;

r = c * A * x + d * y + B * Z;
```

Figura 39: Objetivo del trabajo

Las matrices serán cuadradas y para operar entre ellas y con los vectores todos serán de la misma dimensión, como en el ejemplo de la figura anterior.

5. Caso concreto

5.1 Conceptos e implementación de V_Y = V_Y + a * V_X

Si recordamos el apartado 4.1 y la figura 25, para poder calcular en la tarjeta necesitamos:

- Empaquetar nuestros datos, matrices y vectores, en texturas
- Reprogramar nuestro algoritmo en algún código que sea entendible para la tarjeta
- Enviar las texturas y el programa a la tarjeta
- Decirle a la tarjeta que calcule, dibuje, y que el resultado nos lo deje en otra textura
- Recoger esta textura y traerla a nuestra aplicación
- Y, finalmente, desempaquetar nuestro resultado de la textura.

En este apartado concretaremos todos y cada uno de estos pasos. La siguiente figura es una ampliación de la 15 aplicándole el esquema:

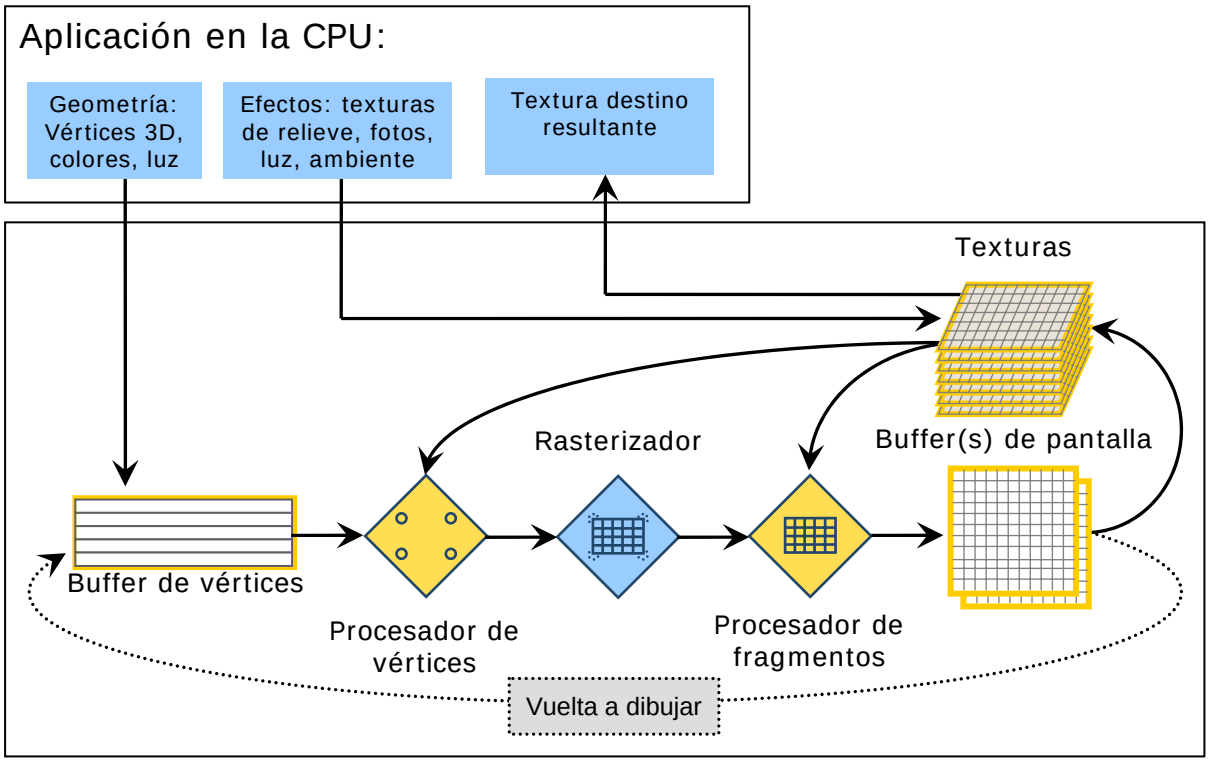


Figura 40: Pipeline gráfica actual resumida.

Los recursos que usaremos en la tarjeta son:

- texturas para empaquetar los vectores y matrices,
- el procesador de fragmentos para calcular nuestro algoritmo, y
- una textura destino para obtener el resultado de nuestro cálculo

Usaremos texturas para empaquetar nuestros vectores y matrices, pues es lo más directo. En el caso más trivial se podría pensar en una textura 1D para almacenar nuestro vector.

Usaremos el procesador de fragmentos porque es el más 'controlable', es decir que podemos escribir directamente en la textura del resultado. El procesador de vértices, en

cambio, no puede hacerlo directamente, y si pretendemos hacerlo indirectamente hemos de tener en cuenta que la información que sale del procesador pasa por todo el pipeline de dibujo de la tarjeta, y esto conlleva controlar muchos detalles del proceso de dibujo. Otra razón es que las compañías fabricantes de tarjetas, en su carrera por el rendimiento, aumentan antes el número de procesadores de fragmentos que el de vértices, como podemos ver en la siguiente tabla [G.3]:

Tarjeta gráfica	Número PV	Número PF	Ancho Bus
NVIDIA GeForce 7900 GTX	8	24	256
NVIDIA GeForce 7900 GT	8	24	256
NVIDIA GeForce 7800 GTX	8	24	256
NVIDIA GeForce 7800 GT	7	20	256
NVIDIA GeForce 7800 GS	6	16	256
NVIDIA GeForce 7600 GT	5	12	128
NVIDIA GeForce 6800 Ultra	6	16	256
NVIDIA GeForce 6800 GT	6	16	256
NVIDIA GeForce 6800 GS	6	12	256
NVIDIA GeForce 6800	6	12	256
NVIDIA GeForce 6800 XT	6	8	256
NVIDIA GeForce 6600 GT	3	8	128
NVIDIA GeForce 6600	3	8	128
NVIDIA GeForce 6600 LE	3	4	128
NVIDIA GeForce 6200	3	4	128
NVIDIA GeForce 6200 TurboCache	3	4	64 / 32
ATI Radeon X1900 XTX	8	48	256
ATI Radeon X1900 XT	8	48	256
ATI Radeon X1800 XT	8	16	256
ATI Radeon X1800 XL	8	16	256
ATI Radeon X1800 GTO	8	12	256
ATI Radeon X1600 XT	8	12	128
ATI Radeon X1600	8	12	128
ATI Radeon X1300 Pro	4	4	128
ATI Radeon X1300	4	4	128
ATI Radeon X1300 HyperMemory	4	4	128

Figura 41: Comparativa entre número de procesadores de vértices (PV), procesadores de fragmentos (PF) y ancho del bus de acceso a memoria

Además tenemos que tener en cuenta que:

- hay que inicializar la librería;
- definir qué recursos de la tarjeta usaremos e inicializarlos;
- para poder definir la precisión y el formato del resultado que queremos obtener tenemos que dibujar fuera de pantalla, 'offscreen', y asociar al buffer de destino una textura para poder recoger el resultado.

Los conceptos que manejamos son:

- Vectores → texturas
- Resultado → textura de destino
- Algoritmo de cálculo → programa de sombreado
- Calcular → pintar

Estudiaremos estos conceptos mientras implementamos la función [B.1]:

VectorY = VectorY + alpha * VectorX

Vectores → texturas

Para almacenar nuestros vectores y matrices usaremos texturas. Se podría pensar que, dado que hay texturas 1D, las usemos para almacenar los vectores. Pero si queremos usar vectores de millones de elementos, estaremos limitados por el tamaño de las texturas 1D, que según la tarjeta será de 1024, 2048 o 4096 elementos, o más.

El formato de texturas nativo de las GPU es el de las texturas bidimensionales y donde podremos guardar hasta 1, 4 ó 16 millones de elementos. Como las texturas también sirven para guardar imágenes observamos cada elemento de la textura, 'texel', corresponde con un píxel de esa imagen y ese píxel consta de 4 componentes: el color rojo, el verde, el azul y la transparencia. Es decir que podemos guardar 1, 4, ó 16 millones de vectores de 4 elementos, con lo que podremos guardar vectores de hasta 4, 16 ó 64 millones de reales. En la figura 42 podemos ver las diferentes maneras de guardar un vector de 10 elementos en una textura.

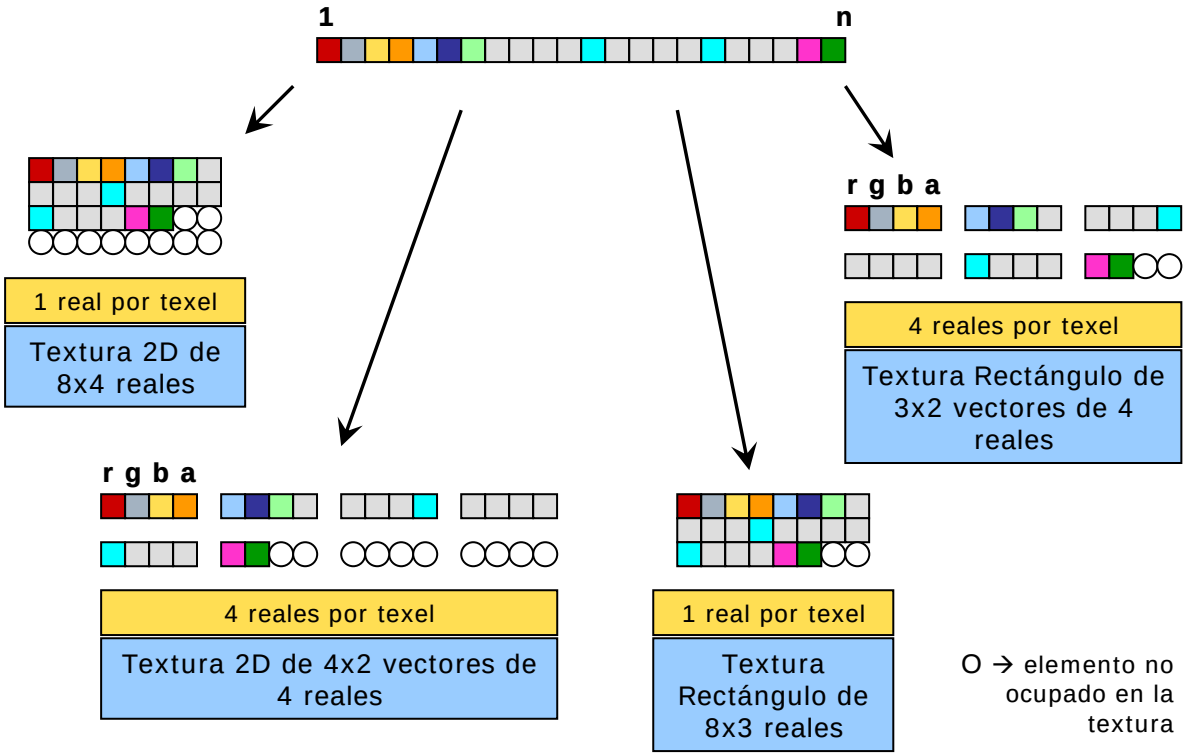


Figura 42: Diferentes formas de guardar un vector según la textura escogida.

Desde la introducción de las texturas en OpenGL sus dimensiones siempre tenían que se potencias de 2. El formato de estas texturas es GL_TEXTURE_2D. NVIDIA, y más tarde ATI, comenzaron a implementar la extensión NV_texture_rectangle y EXT_texture_rectangle que posibilitaba las texturas rectangulares sin esta restricción en sus dimensiones, el nombre del formato es GL_TEXTURE_RECTANGLE_EXT, y que se incorporó a OpenGL con la versión 1.4 y la denominación GL_TEXTURE_RECTANGLE_ARB [G.11, A.14].

A la hora de empaquetar un vector en una textura, pocas veces ocuparemos todos los elementos de una textura como podemos ver en la figura 45. Pero podemos rellenar estos huecos con '0', ceros, pues a la hora de sumar, restar o multiplicar no nos afectaran al resultado final, y sólo devolveremos al usuario los elementos calculados.

A la hora de definir una textura, si queremos almacenar reales y después traspasarle la información de nuestro vector, hemos de especificar tres parámetros:

- el formato básico de las texturas: textura 2D ó textura rectángulo,
- el formato de los elementos de la textura, texel: 1 escalar, por ejemplo luminancia o el color rojo, o un vector de 4 reales por elemento, las cuatro componentes 'rgba',
- el formato de representación interna de la textura en la librería o hardware.

Con este último punto hay que vigilar y escoger un formato que esté soportado por hardware, pues si no la librería nos penalizará con conversiones entre el formato nativo y el deseado, disminuyendo el rendimiento de nuestra librería.

Entonces ¿cuál de los formatos disponibles escogeremos para guardar nuestros vectores? Hemos utilizado estos tres criterios para escoger el formato de la textura:

- minimizar la memoria perdida, pues la memoria en la tarjeta gráfica es un recurso escaso
- maximizar las operaciones aritméticas por ciclo, es decir, minimizar el tiempo de cómputo
- escoger un formato que soporte reales de 32 bits.

Para cumplir con el primer criterio utilizaremos preferentemente las texturas tipo rectángulo, pero como no todas las tarjetas lo soportan, también hemos implementado las texturas 2D.

Como ya hemos visto en el apartado 4.1 se pueden operar vectores de 4 elementos de una sola vez, así pues, para cumplir con el segundo criterio, escogeremos el formato 'rgba' para cada elemento de la textura. Este formato también nos ayuda a maximizar el tamaño de los vectores que el programador podrá usar con nuestra librería.

Para poder cumplir con el tercer criterio, el formato a usar dependerá del escogido con los dos criterios anteriores, como podemos comprobar en esta tabla [B.1, G.1, G.2, G.8, G.9, G.11, F.9 y F.2]:

	NVIDIA	ATI
Textura 2D: extensión requerida para reales	ATI_textura_float ARB_texture_float	ATI_textura_float ARB_texture_float
Textura rectángulo: extensión requerida	NV_float_buffer	ATI_textura_float ARB_texture_float
Textura 2D: 1 componente (luminancia)	GL_FLOAT_R32_NV	GL_LUMINANCE32F_ARB ó GL_LUMINANCE_FLOAT32_ATI
Textura 2D : 4 componentes (rgba)	GL_RGBA32F_ARB	GL_RGBA_32F_ARB ó GL_RGBA_FLOAT32_ATI
Textura rectángulo : 1 componente (luminancia)	GL_FLOATR32_NV	GL_LUMINANCE32F_ARB ó GL_LUMINANCE_FLOAT32_ATI
Textura rectángulo : 4 componentes (rgba)	GL_FLOAT_RGBA32_NV	GL_RGBA32F_ARB ó GL_RGBA_FLOAT32_ATI

Figura 43: Extensiones necesarias para poder almacenar reales en las texturas y formatos de representación interna soportados por el hardware.

En la librería hemos implementado las texturas 2D y las rectangulares, usando preferentemente esta última. Cada texel será un vector de 4 reales, y, dependiendo de la tarjeta y del formato de la textura escogeremos la representación interna según la figura 46.

Una vez definidos los formatos de la textura también hay que definir los parámetros que se usarán para acceder a ésta, pues la librería gráfica dispone de varios filtros según apliquemos la textura a la geometría y la visualización de ésta (no nos olvidemos que estamos usando una librería gráfica): filtro anisotrópico, interpolación linear, bilinear, trilinear, etc., dependiendo de la resolución con la que se pinte la textura en pantalla. En nuestro caso, no nos interesa ninguno de estos filtros que, por otro lado, tampoco se pueden aplicar a texturas con reales. En la siguiente figura se muestra un ejemplo de definición de textura, de los parámetros que usamos en nuestra librería y el traspaso de datos a la tarjeta gráfica.

```
float VectorX[ 1000]; // nuestro vector de reales

int tam_tex = ( 1000 + 3) / 4;
int ancho_tex = 0, alto_tex = 0;
// calculamos las dimensiones de la textura desperdiciando el menor espacio
get_ancho_alto( tam_tex, ancho_tex, alto_tex);

// reajustamos el vector con 0.0's para cuadrarlo con el tamaño de la textura
reajustar( VectorX, ancho_tex * alto_tex * 4);

GLuint id_texVectorX; // identificador de la textura

glEnable( GL_TEXTURE_RECTANGLE_ARB);
glGenTextures( 1, &idx_texVectorX);
glBindTexture( GL_TEXTURE_RECTANGLE_ARB, id_texVectorX);
glTexParameteri( GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri( GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri( GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri( GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_T, GL_CLAMP);

glTexImage2D( GL_TEXTURE_RECTANGLE_ARB, 0, GL_FLOAT_RGBA32_NV,
              ancho_tex, alto_tex, 0, GL_RGBA, GL_FLOAT, NULL);

// ahora transferimos los datos a la tarjeta:
glTexSubImage2D( GL_TEXTURE_RECTANGLE_ARB, 0, 0, 0, ancho_tex, alto_tex,
                 GL_RGBA, GL_FLOAT, VectorX);
```

Figura 44: Pasos a seguir para definir la textura que contenga nuestro vector y enviarlo a la memoria de la tarjeta gráfica.

Comenzamos empaquetando los vectores VectorX y VectorY en dos texturas, pero para realizar la operación 'VectorY = VectorY + alpha * VectorX' usamos como origen y destino el mismo VectorY y, como ya explicamos en el apartado 2.3 y el apartado 4.1, no se puede utilizar la misma textura para leer y para escribir, por lo que necesitaremos de una tercera textura temporal. Al finalizar el cálculo recogeremos los datos de ésta textura temporal en el mismo VectorY.

En la figura 45 vemos la representación de la operación con las texturas y después explicamos cómo se define la textura destino: sobre un cuadrado que dibujamos pintamos dos texturas, la del vector 'x' y la del vector 'y'. Para pintar y obtener los píxeles finales a dibujar, que se guardarán en la textura destino, por cada píxel final, obtenemos el elemento de la textura del vector 'x' y le sumamos el elemento de la textura del vector 'y' multiplicado por 'alpha'

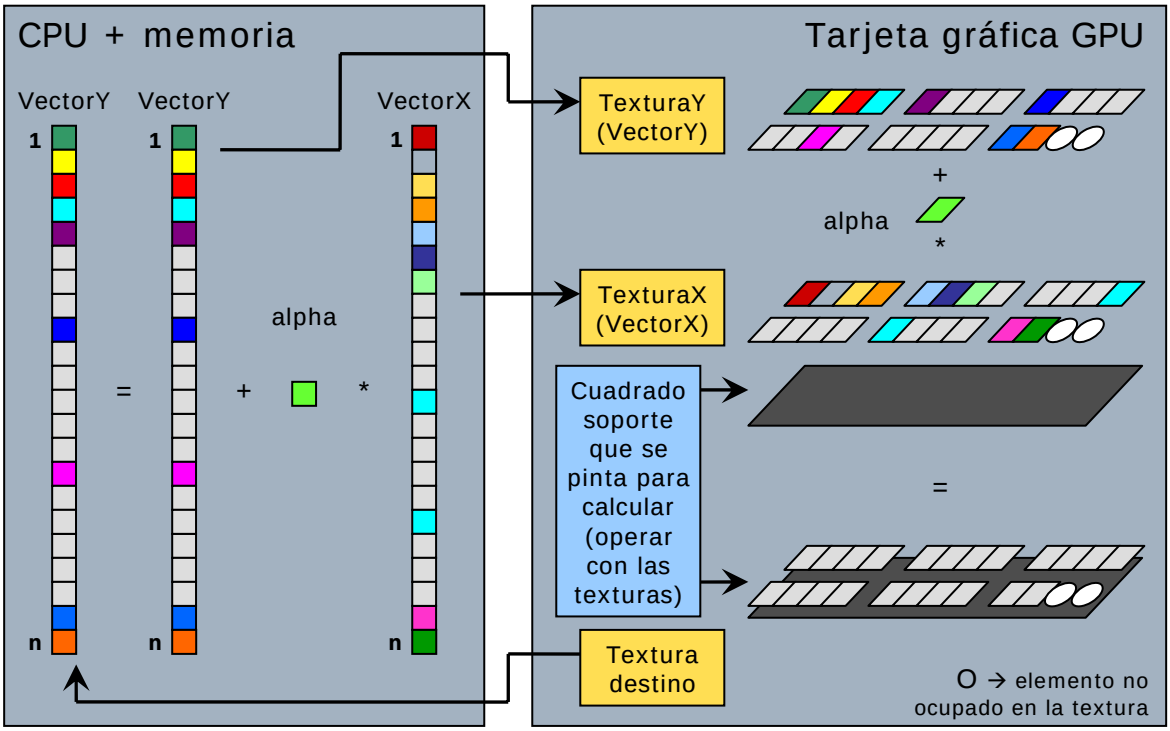


Figura 45: Operación VectorY = VectorY + alpha * VectorX representada con texturas.

Resultado → textura de destino

Para calcular y obtener resultados hemos mencionado que dibujaremos fuera de pantalla 'offscreen'. Para ello existen dos técnicas [F.1, F.2, F.4, F.9]:

- Píxel buffers, o pbuffers, similar a ventanas pero no visibles, y no están pensados para que sea usados extensivamente para cálculos y son pesados de usar:
  - Cada pbuffer tiene su propio contexto de dispositivo y de dibujado, con lo que el cambio de contexto entre diferentes pbuffers tiene un gran coste y, por defecto, no se comparten los objetos entre ellos, hay que habilitarlo específicamente
  - La creación de un pbuffer puede fallar si no hay memoria suficiente y no están pensados para ser creados y destruidos continuamente, con lo que hay que crearlos al principio de la aplicación y liberarlos al salir.
  - El formato de la textura destino viene determinado por el formato del pbuffer.
  - Extensión específica del sistema de ventanas, depende de la parte cliente de la librería gráfica.
  - Si hay un cambio de resolución en la pantalla, los pbuffers pueden perderse.

- FrameBuffer Object provee una interfaz simple para pintar sobre destinos que no sean los buffers del sistema de ventanas.
  - Son creados dentro de un mismo contexto gráfico con lo que el cambio entre FBO es más rápido que entre pbuffers, pues sólo contiene el estado del framebuffer y una estructura de punteros a los objetos de memoria que son el destino a pintar.
  - El formato del FBO viene determinado por el formato de la textura
  - Las texturas de imagen de destino y renderbuffers, para el buffer Z, stencil, pueden ser compartidos entre diferentes FBO.
  - Independientes del sistema de ventanas usado.

La implementación de pbuffer se introdujo en la versión 1.4 de OpenGL y la extensión 'FrameBuffer Object' en OpenGL 2.0 en octubre de 2004. La extensión FBO es bastante reciente y hay drivers que no lo soportan, como por ejemplo los drivers de NVIDIA para Linux. Por eso en la librería he implementado las dos técnicas, usando preferentemente la FBO. En la figura 46 vemos como hay que definir el buffer off-screen usando la extensión 'FrameBuffer Objects' especificar, la textura destino y asociar esta textura al buffer off-screen.

```
// definición de la textura destino
GLuint id_texDestino;
glGenTextures( 1, &id_texDestino);
// definición de la textura según la figura 46, excepto el paso de datos
define_formato_y_parametros_textura( id_texDestino, ancho_tex, alto_tex);

// definición del buffer off-screen
GLuint id_fbo;
glGenFramebuffersEXT( 1, &id_fbo);
glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, id_fbo);

// ligamos la textura al FBO
glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
                           GL_TEXTURE_RECTANGLE_ARB, id_texDestino, 0);

// activamos el buffer de destino para 'calcular' en él
glDrawBuffer( GL_COLOR_ATTACHMENT0_EXT);

(…)

// ahora recogemos el resultado
glReadBuffer( GL_COLOR_ATTACHMENT0_EXT);
glReadPixels( 0, 0, ancho_tex, alto_alto, GL_RGBA, GL_FLOAT, VectorY);
```

Figura 46: Definimos la textura y el buffer de destino, los ligamos y recogemos el resultado.

Algoritmo de cálculo → programa de sombreado

Ahora tenemos definidas las tres texturas, las del VectorX, VectorY y la temporal de destino que después copiamos al VectorY. Pero para realizar la operación 'TexturaTemporal = TexturaVectorY + alpha * TexturaVectorX' aún nos falta definir el parámetro alpha y definir el programa de sombreado que usaremos para operar las texturas. En C estaría claro, si nos fijamos en la siguiente figura:

```
float VectorY[ 1000]; // nuestro vector de reales
float VectorX[ 1000]; // nuestro vector de reales
float alpha;

for ( int i = 0; i < 1000; i++) {
  VectorY[ i] = VectorY[ i] + alpha * Vector_X[ i];
}
```

Figura 47: Función Vector_Y = Vector_Y + alpha * Vector_X en C.

El programa de sombreado se ejecutará por cada fragmento que se dibujará en el buffer de destino y así por cada elemento de la textura de destino, si hemos definido correctamente la matriz de proyección, lo que trataremos más adelante. En el programa de sombreado, además de implementar el algoritmo también hemos de definir los parámetros de entrada. El de salida está claro, el píxel de destino. Como se ejecuta por cada fragmento, no hace falta especificar el bucle de la figura 47, lo que hacemos es acceder a los elementos de las texturas de origen, operar con ellos y escribirlo en la salida. En las siguientes figuras podemos observar que los elementos de las texturas de origen, así como los de la textura de salida son vectores de 4 elementos y que estos lenguajes permiten operar con ellos.

```
uniform samplerRect texturaY;
uniform samplerRect texturaX;
uniform float alpha;

void main( void) {
  vec4 y = textureRect( texturaY,
              gl_TexCoord[ 0].st);
  vec4 x = textureRect( texturaX,
              gl_TexCoord[ 0].st);
  gl_FragColor = y + alpha * x;
}
```

```
uniform sampler2D texturaY;
uniform sampler2D texturaX;
uniform float alpha;

void main( void) {
  vec4 y = texture2D( texturaY,
              gl_TexCoord[ 0].st);
  vec4 x = texture2D( texturaX,
              gl_TexCoord[ 0].st);
  gl_FragColor = y + alpha * x;
}
```

Figura 48: Implementación de la función en GLSL.

```
float4 saxpy(
    in float2 coords: TEXCOORD0,
    uniform samplerRECT texturaY,
    uniform samplerRECT texturaX,
    uniform float alpha ) : COLOR {
  float4 y = texRECT( texturaY, coords);
  float4 x = texRECT( texturaX, coords);
  return y + alpha * x;
}
```

```
float4 saxpy(
    in float2 coords: TEXCOORD0,
    uniform sampler2D texturaY,
    uniform sampler2D texturaX,
    uniform float alpha ) : COLOR {
  float4 y = tex2D( texturaY, coords);
  float4 x = tex2D( texturaX, coords);
  return y + alpha * x;
}
```

Figura 49: Implementación de la función en Cg.

Con ‘uniform samplerXXX’ especificamos las texturas de entrada, con ‘uniform float’ el parámetro alpha y con ‘gl_FragColor’ en GLSL, y el valor de retorno en Cg, especificamos el valor a escribir en el buffer de pintado, que en nuestro caso será la textura de destino. Para acceder a cada elemento, si hemos definido bien la matriz de proyección, el hardware de la tarjeta gráfica nos calcula las coordenadas s y t para acceder a las texturas de origen con ‘gl_TexCoord[0].st’ y ‘ TEXCOORD0’ en GLSL y Cg respectivamente [D.1, D.2].

El programa se guarda en una cadena de texto que hay que enviar a compilar a la librería, ya sea OpenGL o Cg y recoger los identificadores de los parámetros de nuestro programa para asociarlos a las texturas que usaremos. Esto aplicado a nuestro caso concreto lo podemos ver en la siguiente figura:

```
// creamos los objetos de programa y sombreado
GLhandleARB ObjPrograma = glCreateProgramObjectARB();
GLhandleARB ObjSombreado = glCreateShaderObject( GL_FRAGMENT_SHADER_ARB);

// definimos y compilamos el código
GLcharARB codigo = \
  "uniform samplerRect texturaY;" \
  "uniform samplerRect texturaX;" \
  "uniform float alpha;" \
  "void main( void) { " \
  "  vec4 y = textureRect( texturaY, gl_TexCoord[0].st);" \
  "  vec4 x = textureRect( texturaX, gl_TexCoord[0].st);" \
  "  gl_FragColor = y + alpha * x;"\
  "}"; // el texto de la figura 50
glShaderSource( ObjSombreado, 1, &codigo, NULL);
glCompilerShaderARB( ObjSombreado);
glLinkProgramnARB( ObjPrograma);
GLint exito;
glGetObjectParameterivARB( ObjPrograma, GL_OBJECT_LINK_STATUS_ARB, &exito);
if ( !exito) printf( “Error compilando programa\n”);

// ahora cogemos los identificadores de las variables que necesitamos
Glint id_VariableTexturaX = glGetUniformLocationARB( ObjPrograma, "texturaX");
Glint id_VariableTexturaY = glGetUniformLocationARB( ObjPrograma, "texturaY");
Glint id_VariableTexturaAlpha = glGetUniformLocationARB( ObjPrograma, "alpha");
```

Figura 50: Compilación del programa en GLSL y identificación de los parámetros de entrada.

Para Cg el procedimiento es el mismo.

Calcular → pintar

Tenemos las texturas definidas, el programa compilado y los identificadores de los parámetros de entrada del programa. En estos parámetros le indicamos al programa en qué unidad de textura se encuentra nuestra textura de origen. Estas unidades de textura se encargan de acceder a nuestra textura y a aplicarles el filtro requerido, en nuestro caso, ninguno.

Para comenzar a dibujar hemos de activar el programa y asociar las texturas con las unidades de textura y estas con los identificadores del programa, como vemos en la siguiente figura:

```
// siguiendo con la figura 46 y 52
glUseProgramARB( ObjPrograma);

// asociamos las texturas con sus unidades de texturas y con los identificadores
// el VectorY
glActiveTexture( GL_TEXTURE1);
glBindTexture( GL_TEXTURE_RECTANGLE_EXT, id_texVectorY);
glUniform1iARB( id_VariableTexturaY, 1); // unidad de textura 1

// el parámetro alpha
glUniform1fARB( id_VariableAlpha, alpha); // directamente el parámetro


// el VectorX
glActiveTexture( GL_TEXTURE0);
glBindTexture( GL_TEXTURE_RECTANGLE_EXT, id_texVectorX);
glUniform1iARB( id_VariableTexturaX, 0); // unidad de textura 0
```

Figura 51: Activación y asociación de las texturas con los parámetros del programa.

Una vez enviadas las texturas a la tarjeta y el programa de sombreado para poder operar con ellas y obtener la textura destino, tenemos que dibujar. Como no podemos acceder directamente a cada elemento de la textura, para obtener un resultado exacto hemos de escoger un dibujo y una proyección, transformación de mundo a pantalla, de manera que haya una correspondencia 1 a 1 entre los píxeles de destino, textura donde queremos pintar, y la(s) textura(s) de donde sacamos los datos de cálculo.

Como las texturas son rectangulares escogeremos una vista que cubra toda la textura destino y una proyección ortogonal de manera que se correspondan las coordenadas de la geometría, para dibujar, con las coordenadas de las texturas, para obtener los datos de entrada, y con las coordenadas de los píxeles, para escribir los datos de salida. En la siguiente figura vemos estas definiciones.

```
// siguiendo con el ejemplo de la figura 46

glViewport( 0, 0, ancho_tex, alto_tex);
glMatrixMode( GL_PROJECTION);
gluOrtho2D( 0.0, ancho_tex, 0.0, alto_tex);
glMatrixMode( GL_MODELVIEW);
glLoadIdentity();
glDisable( GL_DEPTH_TEST);
```

Figura 52: Definición de la matriz de proyección.

El poder acceder a los elementos del vector cuando esté empaquetado en una textura dependerá del formato escogido. Si el formato que usamos es el GL_TEXTURE_2D las coordenadas s y t para acceder a cada elemento de la textura varían entre (0.0, 0.0) y (1.0, 1.0) independientemente de las dimensiones de la textura. En cambio, si el formato que usamos es el GL_TEXTURE_RECTANGLE, entonces el rango de las coordenadas para acceder a un elemento es (0.0, 0.0) – (m, n) si la textura es de m x n elementos.

Como tenemos dos formatos de texturas, textura 2D y textura rectángulo, las coordenadas de las texturas varían como podemos ver en la siguiente figura:

```
// siguiendo con los ejemplos de las figuras anteriores

// activamos el buffer de destino:
glDrawBuffer( GL_COLOR_ATTACHMENT0_EXT);

glBegin( GL_QUADS);
glTexCoordf( 0.0, 0.0);
glVertex2f( 0.0, 0.0);
glTexCoord2f( ancho_tex, 0.0);
glVertex2f( ancho_tex, 0.0);
glTexCoord2f( ancho_tex, alto_tex);
glVertex2f( ancho_tex, alto_tex);
glTexCoord2f( 0.0, alto_tex);
glVertex2f( 0.0, alto_tex);
glEnd();
```

```
glBegin( GL_QUADS);
glTexCoordf( 0.0, 0.0);
glVertex2f( 0.0, 0.0);
glTexCoord2f( 1.0, 0.0);
glVertex2f( ancho_tex, 0.0);
glTexCoord2f( 1.0, 1.0);
glVertex2f( ancho_tex, alto_tex);
glTexCoord2f( 0.0, 1.0);
glVertex2f( 0.0, alto_tex);
glEnd();
```

Figura 53: Activación de la textura destino y “cálculo”.

Ahora ya tenemos nuestro programa.

5.2 Análisis y observaciones al realizar este caso

La implementación de la función VectorY = VectorY + alpha * VectorX se realizó en C++ con el Microsoft Visual Studio .NET 2002 en un Microsoft Windows XP sobre un ordenador AMD Athlon64 3000+ con 2 GBytes de memoria de canal simple y con una tarjeta gráfica NVIDIA GeForce 6800GT con 256MBytes.

Lo primero que hemos observado es que la extensión FrameBuffer Object no está implementada en la librería ni en los drivers de NVIDIA para Linux. La versión de OpenGL implementada es la 1.4. Tampoco está implementada esta extensión en el driver de la tarjeta gráfica ATI Radeon 9600 del portátil AMD Athlon64 3200+. Ambos si que tienen la extensión pbuffer, que se implementa en la librería. Las librerías usadas para este caso y que se usan en la librería son:

- OpenGL: librería multiplataforma y libre, para toda la interacción con la tarjeta gráfica
- GLUT: librería multiplataforma y libre, para inicializar la ventana
- GLEW: librería multiplataforma y libre, para la inicialización y definición de las extensiones de OpenGL como FBO y PBuffer entre otras.
- Para este caso también hemos usado la librería Cg, propietaria de NVIDIA, para comparar el rendimiento, pero no la usaremos en nuestra librería.

El primer objetivo con este caso concreto ha sido el determinar el proceso necesario para poder usar las GPU para nuestros propósitos: realizar cálculos. Pero manteniendo la portabilidad.

El proceso que hemos seguido para calcular en la tarjeta ha sido este:

1) creación del contexto gráfico de la aplicación
2) almacenaje de vectores en texturas
3) creación del buffer off-screen y de la textura destino
4) creación, compilación y enlazado del programa de sombreado
5) inicialización del entorno off-screen
6) mapeo de variables del programa a texturas
7) envío de las texturas y del programa a la tarjeta gráfica
8) cálculo, dibujamos un cuadrado
9) recogemos la textura del resultado
10) borramos texturas y destruimos contextos gráficos.

Después de ver que el programa funciona hemos puesto puntos de medición de tiempo para evaluar si realmente se consigue el aumento de rendimiento predicho en los diversos artículos encontrados en Internet.

	AMD Athlon64 3000+ (MFlops) Memoria DDR333 ancho bus 64 bits (programa | teórico)		NVIDIA GeForce 6800GT AGP (MFlops) Memoria GDD3-1000 ancho bus 256 bits				
elementos			programa (*1)	todo GLSL (*2)	bucle GLSL (*3)	sin resultado (*4)	Potencia pico
1.000.000	179,184	222	133,843	492,611	1162,79	1418,44	2666
2.000.000	184,2	222	150,34	673,401	1346,8	1702,13	2666
4.000.000	186,586	222	180,439	839,454	1346,8	1705,76	2666
8.000.000	188,241	222	203,73	947,867	1365,18	1764,06	2666
12.000.000	193,446	222	223,93	1003,76	1384,08	1745,45	2666

Elementos	CPU
1.000.000	182,815
2.000.000	192,493
4.000.000	191,755
8.000.000	190,68
12.000.000	196,36

todo Cg	Bucle Cg	sin res.
533,333	711,744	800
595,238	731,261	824,742
648,298	800	897,868
673,684	819,252	939,518
676,628	817,16	936,768

(*1) – todos los pasos incluidos
(*2) – todos los pasos incluidos excepto el 1)
(*3) – pasos incluidos: 6), 7), 8) y 9)
(*4) – pasos incluidos: 6), 7) y 8)

Figura 54: Resultados obtenidos ejecutando el programa VectorY = VectorY + alpha * VectorX según las dimensiones de los vectores y el lugar de medición.

En la figura anterior la columna (*1) es la medida de todo el programa, que incluye todas las creaciones, inicializaciones y destrucciones de todos los recursos y contextos gráficos.
Si dejamos fuera las inicializaciones de las librerías y del primer contexto gráfico, el de la ventana obtenemos los resultados de la columna (*2), que también incluye la inicialización de los recursos que necesitamos para calcular: creación e inicialización del buffer off-screen, del programa de sombreado, y el almacenaje de los vectores en texturas.
Pero si dejamos fuera esta parte, obtenemos los resultados de la columna (*3), con lo que ya nos acercamos al factor 7x visto en diversos artículos [C.1, C.2, C.3, A.12, A.13, A.15, A.6] y que ya mostramos en el apartado 2.6, y si además no recogemos el resultado obtenemos, en la columna (*4), un resultado mejor.

Si en vez de GLSL usamos Cg el rendimiento es entre un 40% y un 50% menor. La columna con título 'potencia pico' corresponde al cálculo teórico, teniendo en cuenta sólo la transferencia de memoria, de la lectura de dos reales de 32 bits, hacer una operación, y escribir el resultado en memoria.

A raíz de este caso podemos hacer dos observaciones:

- hay dos tipos de operaciones:
  - las que se realizan una sola vez, como la inicialización de las librerías gráficas, la ventana y el buffer offscreen, creación de las texturas a usar y su destrucción;
  - las que se realizan cada vez que queremos calcular en la tarjeta: actualización y envío de las texturas, del programa del procesador de fragmentos y recogida del resultado.

- la tarjeta está lejos del par cpu-memoria donde está nuestro programa:
  - la comunicación es lenta entre CPU y tarjeta, el bus AGP 8x soporta hasta 2,1 GBytes/s pero es asíncrono, aunque el nuevo bus PCI Express la aumenta hasta 4GBytes/s en ambas direcciones y simultáneo [G.7, G.6, G.12].
  - La comunicación es compartida con discos duros, usb, tarjeta de red gigabit, si bien es cierto que la comunicación es a ráfagas tanto ésta como la de la transferencia de las texturas a la tarjeta y hay bastantes buffers repartidos entre la cpu, el puente norte, 'north bridge', y el puente sur, 'south bridge'.

En la figura 55 vemos la disposición de los componentes de un PC de hoy en día, dependiendo de la conexión con la tarjeta gráfica (AGP a la izquierda de la figura y PCI Express a la derecha de la misma).

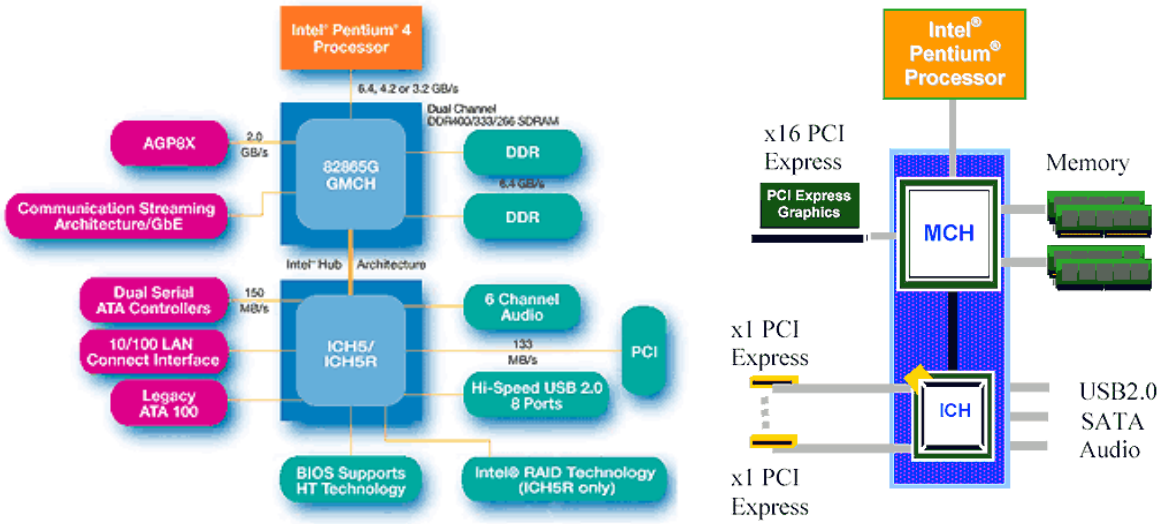


Figura 55: Esquema de comunicación entre la CPU, memoria y la tarjeta gráfica con bus AGP, izquierda, y con bus PCI Express, derecha.

La velocidad de comunicación sólo nos afecta cuando enviamos los datos a la GPU y cuando recogemos el resultado, así pues hay que realizar muchas operaciones en la tarjeta para que el coste de la transferencia resulte rentable.

En la siguiente tabla se puede comprobar el rendimiento de la librería en diferentes plataformas, contando los pasos 6), 7), 8) y 9) del proceso antes descrito:

	elementos					
	1.000.000	2.000.000	4.000.000	8.000.000	12.000.000	16.000.000
Plataforma 1 (CPU / GPU)	180,343	191,022	194,695	195,408	192,957	193,493
	1282,05	1346,8	1346,8	1402,28	1384,08	296,818 (*)
Plataforma 2 (CPU / GPU)	35,920	34,581	36,001	35,822	35,992	35,992
	285,307	246,609	218,221	255,632	242,571	137,67 (*)
Plataforma 3 (CPU / GPU)	130,634	129,282	130,612	130,783	130,833	131,115
	196,85	287,563	184,843	295,094	334,635	Mem. Ins. (*)
Plataforma 4 (CPU / GPU)	216,92	217,037	216,041	216,48	216,041	216,48
	337,268	350,877	281,294	353,123	350,672	98,555 (*)
Plataforma 5 (CPU / GPU)	387,597	400	376,471	384,986	376,471	378,519
	1063,83	522,876	1044,39	76,9897	108,627	141,337 (*)

(*) – bajo rendimiento debido al trasiego de texturas entre la memoria principal y la tarjeta gráfica. En la plataforma 3, OpenGL dio error de memoria insuficiente, la tarjeta dispone de 128 MBytes.

Plataforma 1: AMD Athlont64 3000+ 2.0 GHz 754pins, NVIDIA GeForce 6800GT 256 MB AGP 8x
Plataforma 2: AMD Athlon 850 MHz Socket A, NVIDIA GeForce FX5200 256MB PCI
Plataforma 3: AMD Athlon XP1600+ 1400 MHz Socket A, NVIDIA GeForce FX5500 128MB AGP 8x
Plataforma 4: AMD Athlon64 3000+ 1,8 GHz 939pins, NVIDIA GeForce 6600 256MB PCI Express
Plataforma 5: Intel Pentium 2,8 GHz, ATI Radeon X1600 512MB, AGP 8x

Figura 56: Resultados obtenidos ejecutando el programa VectorY = VectorY + alpha * VectorX según las dimensiones de los vectores y las plataformas.

En la última columna de la figura anterior observamos que el rendimiento de la tarjeta ha bajado si lo comparamos con la columna anterior. Esto es debido al trasiego de las texturas entre la memoria principal y la tarjeta gráfica, al no caber enteras en la tarjeta, pues no toda la memoria de la tarjeta la tenemos a nuestra disposición, también ha de guardar el frame buffer de la pantalla que estamos visualizando, los diferentes contextos gráficos y recursos utilizados. Incluso, en la plataforma 3, la memoria de la tarjeta es insuficiente para manejar las 3 texturas.
En la plataforma 5 podemos observar el efecto de la interacción entre varias tareas que acceden a la tarjeta gráfica: movimiento de ventanas, visualización de gráficos, etc.

Es curioso que incluso una plataforma algo vieja como la plataforma 2, que usa un Athlon a 850MHz, obtiene un rendimiento superior a un AMD Athlon64 3000+ con canal doble de memoria, incluso usando el bus PCI normal. Si comparamos la plataforma 4 con la 1 vemos que el rendimiento es 4 veces inferior, a pesar de contar con una mejor comunicación con la tarjeta gráfica, bus PCI Express en vez de AGP. En la siguiente figura podemos comprobar el porqué:

	GeForce 6800 GT 256 MB AGP 8x	GeForce 6600 256MB PCI Express
Velocidad memoria	GDDR3-1000 (500MHz)	DDR2-800 (400MHz)
Ancho de bus	256 bits	128 bits
Velocidad transferencia	31,25 GBytes/s	12,5 GBytes/s
# procesadores de fragmentos	16	8
Precio	319 € (2006)	90 € (2006)

Figura 57: Comparación características de las tarjetas gráficas de las plataformas 1 y 4.

Podemos observar que además de limitar el número de procesadores de fragmentos también se ha reducido a la mitad el ancho de bus de comunicación con la memoria y la velocidad de ésta.
Vale la pena invertir en la tarjeta de gama alta, es decir con la GPU más potente, sin que sea la de más alto nivel, por ejemplo la GeForce 6800 Ultra, cuyo precio es de 462 € (www.acuista.com), utiliza memoria GDDR3-1100, a 550MHz, en cambio la GeForce 6800 GT que lleva la mima GPU, usa memoria GDDR3-1000, a 500MHz, y su precio es de 319 € (www.acuista.com).

5.3 Recursos a usar por la librería

Después de realizar esta implementación hemos podido determinar los recursos que necesita la librería:

- Texturas para almacenar vectores y matrices, preferentemente usaremos las texturas rectangulares y si no están disponibles, las texturas 2d. Los elementos que guardaremos en cada elemento de la textura serán vectores de 4 reales, 'rgba' y el formato interno el especificado en la figura 44.
- El procesador de fragmentos,
- Para programarlo usaremos GLSL, soportado por OpenGL, así nos ahorramos usar una librería, la de Cg de NVIDIA.
- Como buffer offscreen usamos el FrameBuffer Object si esta presente, y si no, usamos un píxel buffer.
- Unidades de textura para que el programa de sombreado acceda a las texturas con los datos de los vectores.

También evitaremos el movimiento de información innecesario entre la tarjeta y la aplicación:

- enviaremos la textura de nuevo a la tarjeta sólo si el programa ha modificado alguno de los elementos del vector,
- sólo recogeremos el resultado de la tarjeta cuando el programa acceda al resultado.

En la figura siguiente vemos un ejemplo:

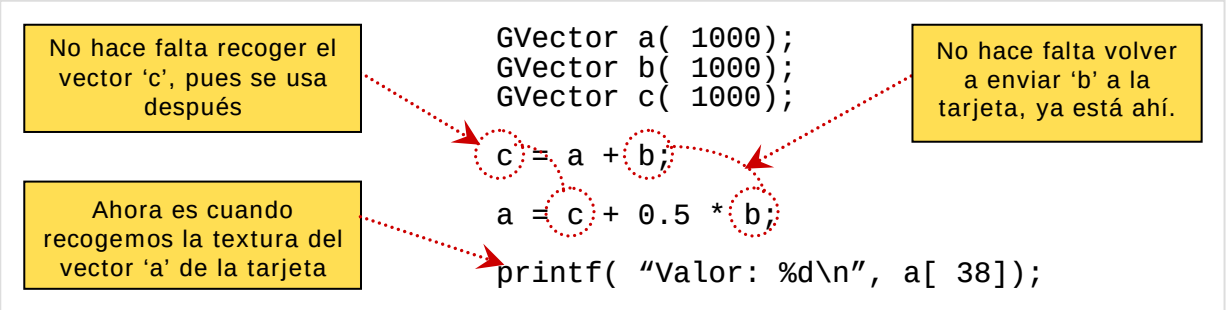


Figura 58: Evitamos el movimiento de información innecesario.

Según lo aprendido de este caso concreto y según el ejemplo anterior cabe diferenciar las operaciones realizadas en dos categorías:

- Operaciones que se hacen una sola vez:
  - inicializaciones de las librerías,
  - creación de la ventana de inicio,
  - creación del buffer off-screen

    - determinación de los recursos disponibles: formato de las texturas,
    - destrucción de los recursos usados

- Operaciones se hacen varias veces:
    - creación de las texturas de los vectores y matrices cuando se declaren,
    - acumulación de las operaciones a realizar
    - creación del programa GLSL cuando se tenga que evaluar
    - envío de texturas a la tarjeta sólo cuando se necesiten y si se han modificado
    - recogida de resultados sólo cuando el programa lo necesite
    - destrucción de las texturas cuando se destruyan los vectores y matrices

De esta manera la librería sólo hará lo estrictamente necesario en las operaciones de cálculo, dejando las inicializaciones y liberaciones de recursos lo más alejadas posibles del cálculo en la tarjeta, para no mermar el rendimiento.

6. Desarrollo

6.1 Definición e implementación de la librería

Una vez definidos los requerimientos de la librería, véase el apartado 4.3, y vistos los recursos que necesitamos y los tipos de operaciones a realizar, véase apartado 5.3, pasaremos a especificar las clases que usará el programador y los pasos seguidos en la implementación.

Las clases que el programador usará serán:

- clase GFloat: para definir escalares y constantes que utilizará para operar con vectores y matrices
- clase GVector: para definir vectores con los que operar
- clase GMatrix: para definir matrices llenas y cuadradas
- clase GSMatrix: para definir matrices dispersas, 'sparse matrix', en las que la mayor parte estará llena de ceros.

Y las operaciones:

- suma, resta y multiplicación de componentes de vectores
- multiplicación de un vector y un escalar
- multiplicación de una matriz con un vector

Vistas las operaciones a realizar y recursos a inicializar y liberar, el funcionamiento de la librería será este (también reflejado en la figura 59):

- la primera declaración de cualquier vector o matriz inicializa la tarjeta y prepara los recursos disponibles, además crea la textura que lo almacena, y las demás declaraciones sólamente crearán la textura que necesitará el vector o matriz;
- los operadores '+', '-' y '*' acumulan operaciones hasta que la asignación los evalúa en la tarjeta;
- en el momento de esta evaluación, se envían los vectores y matrices como texturas a la tarjeta y se calcula, pero no se recoge el resultado;
- sólo cuando se accede a algún elemento del resultado, éste se recoge de la tarjeta.

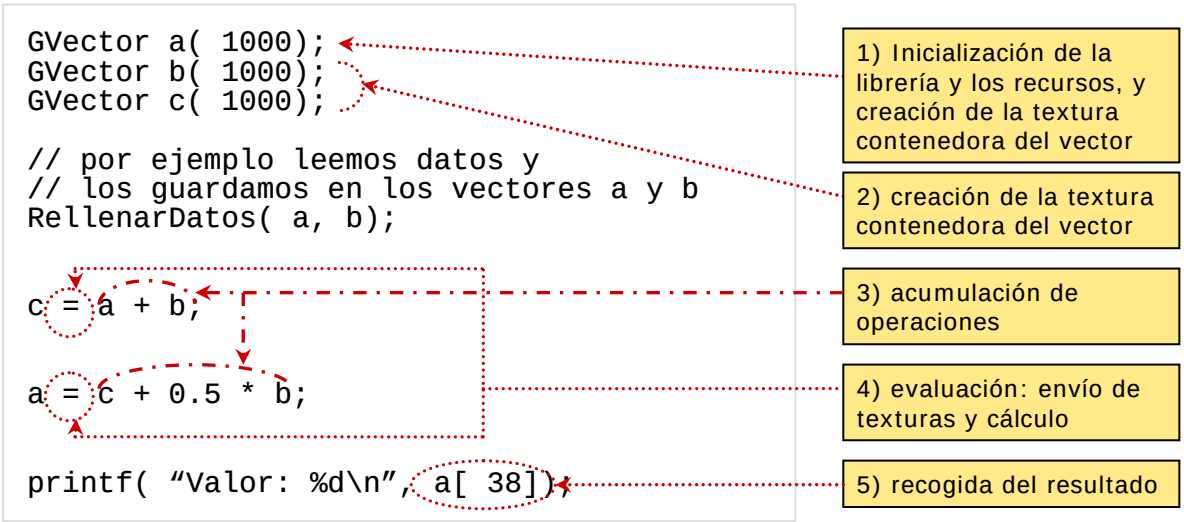


Figura 59: Etapas de funcionamiento de la librería.

Según este funcionamiento y las clases que el usuario verá, la librería se puede dividir en tres capas, como también se puede ver en la figura 60:

- la de interacción con el usuario que contiene las clases GFloat, GVector, GMatrix y los operadores,
- la de inicialización e interacción con la tarjeta gráfica,
- y la de la recolección de operaciones, preparación de datos y del programa de sombreado y cálculo.

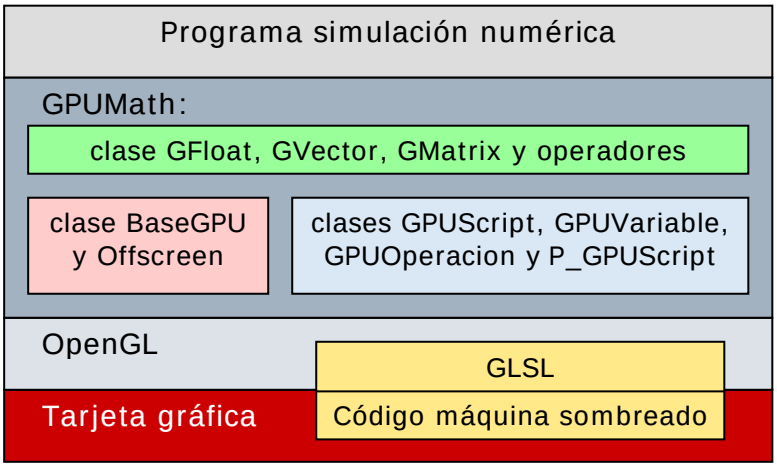


Figura 60: Capas de software entre la aplicación y la tarjeta gráfica

6.2 Implementación

La implementación se ha realizado en C++ definiendo los tipos de datos básicos: GFloat, GVector y GMatrix. Se utiliza la sobrecarga de operadores '+', '-', '*' y '=' para acumular las operaciones entre estos tipos de datos y, en la asignación, construir y evaluar el programa de sombreado en la GPU, o la CPU. También se utiliza la sobrecarga del operador de acceso '[]' para recoger la textura de la tarjeta, si es resultado de un cálculo y todavía está allí, o para indicar a la librería que hay que volver a enviar la textura a la tarjeta.

Primero se ha implementado el esquema de la librería para poder realizar la función: VectorY = VectorY + alpha * VectorX

Para lo que se ha desarrollado las siguientes clases de la estructura básica de funcionamiento y los tipos de datos GFloat y GVector:

- **clase GVector**: contiene el vector de datos de la aplicación, su correspondiente textura y su estado: si se ha enviado a la tarjeta, si hay que enviarla a la tarjeta, si hay que recogerla (si es un resultado). También contiene funciones de declaración y acceso de la variable en el programa GLSL. Además contiene el operador de asignación que evalúa el **GPUScript** (colección de operaciones a realizar y que devuelve como resultado un GVector) que es asignado. Si el objeto global BaseGPU no está inicializado, lo inicializa.

- **clase GFloat**: contiene un escalar y las funciones de declaración en el programa GLSL.

- **clase BaseGPU**: habrá un objeto global que se encarga de inicializar la librería gráfica, crear la ventana inicial. También creará el buffer de off-screen sobre el que dibujaremos ('calcularemos') según si existe la extensión FrameBufferObject o

PBuffer. Otras de sus funciones será la de ir verificando si ha habido algún error y la de escoger el formato de texturas adecuado según lo explicado en el apartado 5.1. La primera declaración de GVector o GMatrix creará este objeto global y lo inicializará. Y en el momento de destruir el objeto global se liberarán los recursos usados.

- **clase BaseOffScreen**: es la clase base que define las operaciones comunes entre los dos tipos diferentes de buffer offscreen: FrameBufferObject y PixelBuffer

- **clases OS_FBO** y **OS_PBuffer**: según de que recurso dispondrá la tarjeta, la clase BaseGPU usará un objeto OS_FBO u OS_PBuffer para gestionar el buffer off-screen. Preferentemente se usará el OS_FBO, pues es menos costoso de crear y de manejar y más versátil (nos permite usar fácilmente varias texturas destino).

- **clase GOffScreen**: clase genérica a través de la cual BaseGPU escoge el tipo de buffer off-screen (OS_FBO u OS_PBuffer) y lo inicializa.

- **clase GPUScript**: objeto que se crea al definir en C++ las expresiones de las operaciones a realizar entre GFloat y GVector. Este script se va pasando a lo largo de la expresión de C++, encapsulado en la clase P_GPUScript, para ir recolectando las operaciones en una lista. En el momento que es asignado a un GVector, se construye el programa GLSL y se ejecuta en la GPU o en la CPU. C++ nos hace el parking de las operaciones y nos las ordena por prioridad.

- **clases GPUVariable** y **GPUOperacion**: estos objetos los irá usando la librería para acumular las operaciones y operadores en la lista interna del objeto GPUScript.

- **clase P_GPUScript**: clase contenedora que encapsula un puntero al GPUScript que se está construyendo y que se pasa a través de los operadores que se utilizan para especificar la expresión a evaluar.

Y para propagar el GPUScript y realizar la expresión anterior, se usan los operadores:
- P_GPUScript operator+(GVector &v1, GVector &v2);
- P_GPUScript operator*(GFloat &r1, GVector &v2);
- P_GPUScript operator+(GVector &v1, P_GPUScript scr);
- GVector &GVector::operator=(P_GPUScript s);

Que se usan en este código:

```
GFloat a = 0.5;

GVector y( 1000), x( 1000);

for ( int i = 0; i < 1000; i++) {
  x[ i] = sin( i);
  y[ i] = cos( i);
}

// por defecto se evalúa en la GPU, y si no puede, en la cpu
y = y + a * x;

// ahora evaluamos en la CPU
y.usaCPU();
y = y + a * x;

y.usaGPU(); // para volver a evaluar en la GPU
```

Figura 61: función VectorY = VectorY + alpha * VectorX usando la librería GPUMath.

Durante la ejecución los pasos que se siguen son estos:

```
GFloat a = 0.5;
GVector y( 1000), x( 1000);

            y = y + a * x;
```

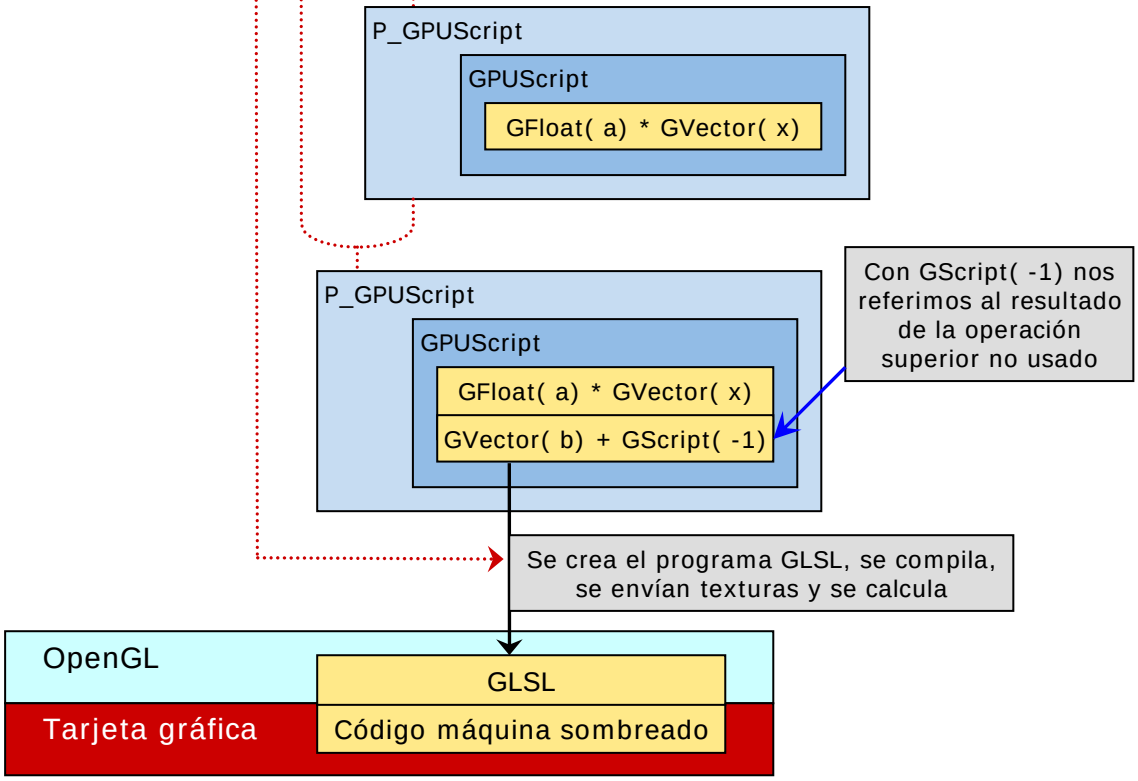


Figura 62: Acumulación de operaciones en el objeto GPUScript.

Seguidamente se ha implementado otros operadores:

- P_GPUScript operator-(GVector &v1, GVector &v2);
- P_GPUScript operator*(GVector &v1, GFloat &r2);
- P_GPUScript operator*(GVector &v1, GVector &v2); // producto componente a componente

// y para encadenar operadores que continúan una expresión

- P_GPUScript operator-(GVector &v1, P_GPUScript scr);
- P_GPUScript operator*(GFloat &r1, P_GPUScript scr);
- P_GPUScript operator*(GVector &v1, P_GPUScript scr); // producto componente a componente
- P_GPUScript operator+(P_GPUScript scr1, P_GPUScript scr2);
- P_GPUScript operator-(P_GPUScript scr1, P_GPUScript scr2);
- P_GPUScript operator*(P_GPUScript scr1, P_GPUScript scr2);

Con las que se puede realizar operaciones como las de la figura siguiente:

```
const int TAM_VECTOR = 8000000;

GVector v1( TAM_VECTOR), v2( TAM_VECTOR), v3( TAM_VECTOR), v4( TAM_VECTOR);
GFloat a, b;

InicializaVectores( v1, v2, v3, v4, a, b);

V3 = v1 + a * v2 - b * v4 + v1 * v2;
```

Figura 63: Operando vectores y escalares.

La lista de las operaciones que se acumulan en la clase GPUScript es esta, recordemos que C++ ya nos ordena esta lista:

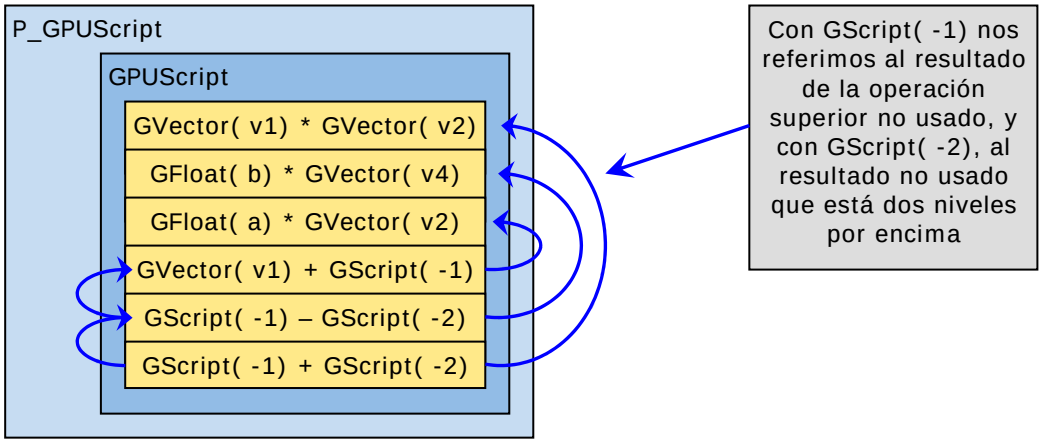


Figura 64: Acumulación de operaciones en el objeto GPUScript de la expresión de la figura 63.

Con las operaciones de la figura 62, el programa que se crea en GLSL es este:

```
 1:  uniform sampler2D Vector1;
 2:  uniform sampler2D Vector2;
 3:  uniform float Float2;
 4:  uniform sampler2D Vector4;
 5:  uniform float Float1;
 6:
 7:  const vec2 CoefIdx2Vector = vec2( 1.0 / 64.000000, 1.0 / 64.000000);
 8:  const vec2 CoefVector2Idx = vec2( 64, 4096);
 9:
10:  void main( void) {
11:    vec4 v4Vector1 = texture2D( Vector1, gl_TexCoord[ 0].st);
12:    vec4 v4Vector2 = texture2D( Vector2, gl_TexCoord[ 0].st);
13:    vec4 v4Vector4 = texture2D( Vector4, gl_TexCoord[ 0].st);
14:    gl_FragColor = ( ( Float2 * v4Vector4) - ( v4Vector1 + ( Float1 *
v4Vector2))) + ( v4Vector1 * v4Vector2);
15:  }
```

Figura 65: Programa GLSL creado a partir del ejemplo anterior.

Los nombres ‘Vector1’, ‘Vector2’, etc., se van generando automáticamente según el orden de creación de los objetos.

Una vez implementadas las operaciones entre vectores he pasado a implementar la clase GMatrix, que contiene una matriz cuadrada y llena, con lo que sus dimensiones quedan limitadas a GL_MAX_TEXTURE_SIZE x GL_MAX_TEXTURE_SIZE. Más adelante se

implementará la matriz dispersa. También se ha implementado el operador P_GPUScript operator*(GMatrix &m1, GVector &v2);.

```
 1:  uniform sampler2D Matrix1;
 2:  uniform sampler2D Vector3;
 3:
 4:  const vec2 CoefIdx2Vector = vec2( 1.0 / 2.000000, 1.0 / 2.000000);
 5:  const vec2 CoefVector2Idx = vec2( 2, 4);
 6:  const vec2 CoefIdx2Matriz = vec2( 1.0 / 8.000000, 1.0 / 8.000000);
 7:  const vec2 CoefMatriz2Idx = vec2( 8, 64);
 8:  const vec2 tamMatriz = vec2( 4, 1);
 9:
10:   // Inicio 'Libreria'
11:
12:  vec2 VectorIdx2D( in float idx, in vec2 factor) {
13:    float idxNorm = idx * factor.x;
14:    vec2 offset = 0.5 * vec2( factor.xy);
15:    vec2 ret = vec2( fract( idxNorm), floor( idxNorm) * factor.y) + offset;
16:    return ret;
17:  }
18:
19:  float Vector2DIdx( in vec2 st, in vec2 factor_v2i, in vec2 factor_i2v) {
20:    vec2 st_00 = st - 0.5 * vec2( factor_i2v.xy);
21:    return dot( st_00, factor_v2i);
22:  }
23:
24:  vec2 MatrizIdx2D( in vec2 idx, in vec2 factor, in vec2 tam) {
25:    float idxLineal = dot( idx, tam); // lo convertimos en un indice linea
26:    //return vec2( idxLineal, idxLineal);
27:    //return factor;
28:    return VectorIdx2D( idxLineal, factor);
29:  }
30:
31:  vec4 ProdMatVec( vec2 pos, sampler2D M, sampler2D V) {
32:    float fila = Vector2DIdx( pos, CoefVector2Idx, CoefIdx2Vector);
33:    vec4 filasMatriz = vec4( 0, 1, 2, 3) + 4.0 * vec4( floor( fila));
34:    vec4 res = vec4( 0.0);
35:    //vec4 res2 = vec4( 0.0);
36:    // Internal compiler error:loop constructs not yet implemented. an ATI
37:    for ( float ic = 0.0; ic < tamMatriz.x; ic += 1.0) {
38:      vec4 mat_1 = texture2D( M, MatrizIdx2D( vec2( filasMatriz.r, ic),
39:                                              CoefIdx2Matriz, tamMatriz));
40:      vec4 mat_2 = texture2D( M, MatrizIdx2D( vec2( filasMatriz.g, ic),
41:                                              CoefIdx2Matriz, tamMatriz));
42:      vec4 mat_3 = texture2D( M, MatrizIdx2D( vec2( filasMatriz.b, ic),
43:                                              CoefIdx2Matriz, tamMatriz));
44:      vec4 mat_4 = texture2D( M, MatrizIdx2D( vec2( filasMatriz.a, ic),
45:                                              CoefIdx2Matriz, tamMatriz));
46:      mat4 elem_matriz = mat4( mat_1, mat_2, mat_3, mat_4);
47:      vec4 elem_vector = texture2D( V, VectorIdx2D( ic, CoefIdx2Vector));
48:      res += elem_matriz * elem_vector;
49:      //float t = elem_vector.x + elem_vector.y + elem_vector.z + elem_vector.w;
50:      //res2 += vec4( t, t, t, t);
51:      // vec2 im = MatrizIdx2D( vec2( filasMatriz.r, ic), CoefIdx2Matriz, tamMatriz);
52:      // res2 = vec4( vec2( filasMatriz.r, ic), im.st);
53:      // vec2 im2 = MatrizIdx2D( vec2( filasMatriz.g, ic), CoefIdx2Matriz, tamMatriz);
54:      // vec2 im3 = MatrizIdx2D( vec2( filasMatriz.b, ic), CoefIdx2Matriz, tamMatriz);
55:      // vec2 im4 = MatrizIdx2D( vec2( filasMatriz.a, ic), CoefIdx2Matriz, tamMatriz);
56:      // res2 = vec4( im.t, im2.t, im3.t, im4.t);
57:      //res2 = mat_1;
58:    }
59:    //res2 = texture2D( V, VectorIdx2D( fila, CoefIdx2Vector));
60:    //vec2 tt = VectorIdx2D( fila, CoefIdx2Vector);
61:    //res2 = vec4( pos, tt);
62:    //res2 = vec4( pos.st, CoefVector2Idx);
63:    //res2 = vec4( fila, filasMatriz.xyz);
64:    return res;
65:  }
66:
67:   // Fin 'Libreria'
68:
69:
70:  void main( void) {
71:    vec4 v4Vector3 = texture2D( Vector3, gl_TexCoord[ 0].st);
72:    gl_FragColor = ProdMatVec( gl_TexCoord[ 0].st, Matrix1, Vector3);
73:  }
```

Figura 66: Programa GLSL que multiplica una matriz con un vector, en comentarios líneas para depurar el código y seguir errores.

6.3 Pruebas realizadas y problemas surgidos

Una vez implementadas las clases GFloat y GVector, se han encontrado los siguientes problemas:

- Aunque Linux soporta pbuffers, no tiene compilador de GLSL
- A la hora de verificar los resultados de operaciones de vectores he observado que la columna 4096 de la textura destino no se actualizaba, pero la fila 4096 si. ¿Error de driver? Con lo que el primer esquema de guardar los vectores en texturas lo más anchas posibles se ha cambiado a lo más cuadradas posibles.
- En la plataforma con la tarjeta ATI Mobility Radeon 9600, a pesar de tener pbuffers y no FrameBuffer Objects, no dispone de bucles para poder hacer la multiplicación matriz x vector como está implementada en la figura 67.
- Debido a que el método de acceso a las matrices y vectores se han de hacer con números reales, y estos tienen una precisión de 32 bits, menor que en la CPU, los errores de redondeo hace que accedamos a posiciones erróneas cuando el tamaño de la matriz excede de 888 x 888, con lo que se ha de cambiar el esquema de acceso a las matrices, por ejemplo con dos texturas de soporte con las coordenadas ‘s’ y ‘t’ del inicio de las filas ya precalculadas.

7. Conclusiones

7.1 Librería GPUMath

Se ha presentado una librería para poder operar efectivamente con escalares y vectores grandes y con matrices llenas que aprovechan la última tecnología en tarjetas gráficas. El objetivo de simplicidad se ha cumplido de cara el programador de simulación numérica, que lo único que ha de hacer es:

- añadir la línea '#include "GPUMath.h"' a su programa
- declarar con los tipos de la librería las variables matriciales, vectoriales y escalares que desea usar
- operar entre ellos como lo haría normalmente en C++
- añadir las librerías GLUT, GLEW y OpenGL en el apartado de enlazado de su proyecto.

7.2 Estudio

Se ha realizado un estudio para la comprensión de la arquitectura de las tarjetas gráficas y su futura evolución que justifica el desarrollo de la librería. De las tres gamas, baja, media y alta, que compañías como NVIDIA y ATI comercializan, hay diferencias sustanciales en cuanto a rendimiento. Incluso con una mínima inversión se pueden aprovechar una plataforma de más de 3 años de antigüedad para obtenga el rendimiento equivalente a un ordenador nuevo de gama media-baja.

7.3 Líneas de trabajo

La librería ha despertado interés en CIMNE, Centro de Investigación de Métodos Numéricos en Ingeniería, lugar donde trabajo, con lo que se seguirá el desarrollo de la librería:
Se implementarán clases de matrices dispersas y algunos algoritmos numéricos, con el de gradientes conjugados o el biconjugado para resolver sistema de ecuaciones grandes (b = A * f).
También se implementarán matrices asimétricas y las correspondientes operaciones.

Pero para que una librería de estas características tuviera éxito entre la comunidad científica, las tarjetas tendría que dar soporte a precisión doble, reales de 64 bits, posibilitar un acceso más directo a las texturas tanto para indexarlas dentro de las tarjetas (que se pueda hacer por enteros y con las dimensiones de la textura) como para mapearlas a memoria para poder traspasar los datos directamente. Y otro punto fuerte sería poder disponer OpenGL 2.0 a Linux.

Pero la preocupación máxima son los planes de Microsoft de no implementar directamente, por hardware, OpenGL en su próximo sistema operativo Windows Vista, si no que sólo se emulará OpenGL 1.4 sobre DirectX 10, con lo que a parte de una merma de su rendimiento supondrá no poder usar toda la potencialidad actualmente existente en OpenGL 2.0 en ese sistema operativo del 'futuro'. Supondrá la muerte de OpenGL en el mundo Microsoft.

8. Bibliografía

A. GPGPU y GPU:

A.1 “GPU Gems 2, programming techniques for high-performance graphics and general-purpose computation”, edited by Matt Pharr and Randita Fernando, © 2005 Addison-Wesley

A.2 “Accelerated Implementation of the S-MRTD Technique Using Graphics Processor Units”, Gerar S. Baron and Costas D. Sarris, University of Toronto.

A.3 “Graphics Processor Unit (GPU) Acceleration of Finite-difference Time-Domain (FDTD) algorithm” Sean E. Krakiwsky, University of Calgary, Canada

A.4 “Linear Algebra on GPUs”, Jens Krüger, Technische Universität München, Eurographics 2005

A.5 “General-Purpose Computation on Graphics Hardware”, David Luebke, University of Virginia, 2005

A.6 “GPGPU: General-Purpose Computation on GPUs”, Mark Harris, NVIDIA Corporation, 2005

A.7 “Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid”, Jeff Bolz, Ian Farmer

A.8 „Accelerating Double Precision FEM Simulations with GPUs“, Dominik Göddeke, Universität Dortmund, 2005

A.9 „Understanding the Efficiency of GPU algorithms for Matrix-Matrix Multiplication“, K.Fatahalin, Stanford University, 2004

A.10 http://www.gpgpu.org

A.11 “Dynamic Particle Coupling for GPU-based Fluid simulation”, Andreas Kolb, 2005

A.12 “A survey of General-Purpose Computation on Graphics Hardware”, John D. Owens, Eurographics 2005,

A.13 “NVIDIA samples: cloth, fluid simulation” NVIDIA, 2005

A.14 “NVIDIA GPU Programming Guide, version 2.4.0”, NVIDIA, 2005

A.15 “Algoritmo gradientes conjugados para electromagnetismo en Kratos”, Rubén Otín, 2005

A.16 http://www.gidhome.com

A.17 http://www.cimne.com

A.18 http://www.cimne.com/kratos

A.19 http://www.quantech.es

A.20 http://www.compassis.com

A.21 “Cálculo de Estructuras por el Método de Elementos Finitos, Análisis estático lineal” Eugenio Oñate, © Septiembre 1995 CIMNE

B. Tutoriales GPGPU:

B.1 “GPGPU – Basic math tutorial” Dominik Göddeke, Universität Dortmund, 2005

B.2 http://www.siggraph.org/s2005/

C. Lenguajes generales que abstraen la GPU:

C.1 “High Level Languages for GPUs”,Ian Buck, NVIDIA, Eurographics 2005

C.2 “Accelerator: simplified programming of graphics processing units for general-purpose uses via data parallelism”, David Tarditi, Microsoft, septiempre 2005 http://research.microsoft.com/act/

C.3 http://graphics.standord.edu/projects/brookgpu/

C.4 http://libsh.org

C.5 “Glift: Generic, Efficient Random-Access GPU Data Structures”, Aarone Lefohn, University of California

C.6 “An Introduction to the Scout Programming Language” Patrick McCormick, Los Alamos National Laboratory GPGPU, 2005

D. Lenguajes de sombreado:

D.1 “OpenGL Shading Language”, Randi J.Rost, © 2004 Addison-Wesley

D.2 “The Cg tutorial, the definitive guide to programmable real-time graphics”, Randima Fernando and Mark J. Kilgard, © 2003 NVIDIA Corporation and Addison-Wesley

D.3 http://www.neatware.com/lbstudio/web/hlsl.html

E. OpenGL:

E.1 “OpenGL Programming Guide, second edition”, Mason Woo, Jackie Neider and Tom Davis, edited by the OpenGL Architecture Review Board, © 1997 Addison-Wesley Developers Press

E.2 “The OpenGL Graphics System: A Specification (version 2.0 – October 22, 2004)”, Mark Segal and Jurt Akeley

E.3 “NVIDIA OpenGL 2.0 Support”, Mark J. Kilgard, NVIDIA, Abril 2005

F. Extensiones OpenGL:

F.1 “The OpenGL Framebuffer Object extension”, Simon Green, NVIDIA Corporation, Marzo 2005

F.2 “OpenGL Render-to-Texture”, Chris Wynn, NVIDIA, 2004

F.3 “OpenGL Vertex programming on Future-Generation GPUs”, Chris Wynn, NVIDIA Corporation, 2005

F.4 “Rendering to an off-screen buffer with WGL_ARB_pbuffer”, Christopher Oat, ATI Research, inc.

F.5 “High Dynamic Range Rendering on the GeForce 6800”, Simon Green / Cem Cebenoyan, NVIDIA Corporation, 2005

F.6 “Shader Model 3.0, Best practices”, Phil Scott, NVIDIA, 2005

F.7 “Whitepaper – Shader Model 2.0 Using Vertex textures” Philipp Gerasimov, Randy Fernando, Simon Green”, NVIDIA Corporation, Junio 2004

F.8 “NVIDIA OpenGL Extension Specifications”, Mark J. Kilgard, NVIDIA, Mayo 2004

F.9 “ATI OpenGL Extension Support”, 2002

G. Hardware, GPU y arquitectura OpenGL:

G.1 http://developer.nvidia.com

G.2 http://www.ati.com/developer/index.html

G.3 http://www.tomshardware.com

G.4 Proyecto de Fin de Carrera: “Generación y compilación de shaders para una GPU programable” de Jordi Roca Monfort

G.5 “Floating Point Unit”, Jidan Al-Eryani, 2005

G.6 “PCI Express technology White paper”, February 2004, DELL

G.7 “AGP v3.0 interface specification”, intelm September 2002

G.8 “Fast Texture Downloads and Readbacks using Pixel Buffer Objects in OpenGL” NVIDIA, August 2005

G.9 “Floating-Point Specials on the GPU” Technical Brief, Cem Cebenoyan, NVIDIA, February 2005

G.10 “What Every Computer Scientist Should Know About Floating-Point Arithmetic”, David Goldberg, Xerox Palo Alto Research Center, 1991

G.11 “NVIDIA OpenGL Texture Formats”, NVIDIA, 2005

G.12 “Procesadores gráficos para PC”, Manuel Ujaldón Martínez, © 2005 Editorial Ciencia-3, s.l.

G.13 http://www.wikipedia.org

G.14 Mueller, Scott (1992) Upgrading and Repairing PCs, Second Edition, Que Books, ISBN 0-88022-856-3

G.15 http://www.nvidia.com

G.16 Arthur Trew and Greg Wilson (1991) Past, Present, Parallel: A Survey of Available Parallel Computing Systems. New York: Springer-Verlag, ISBN 0-387-19664-1.