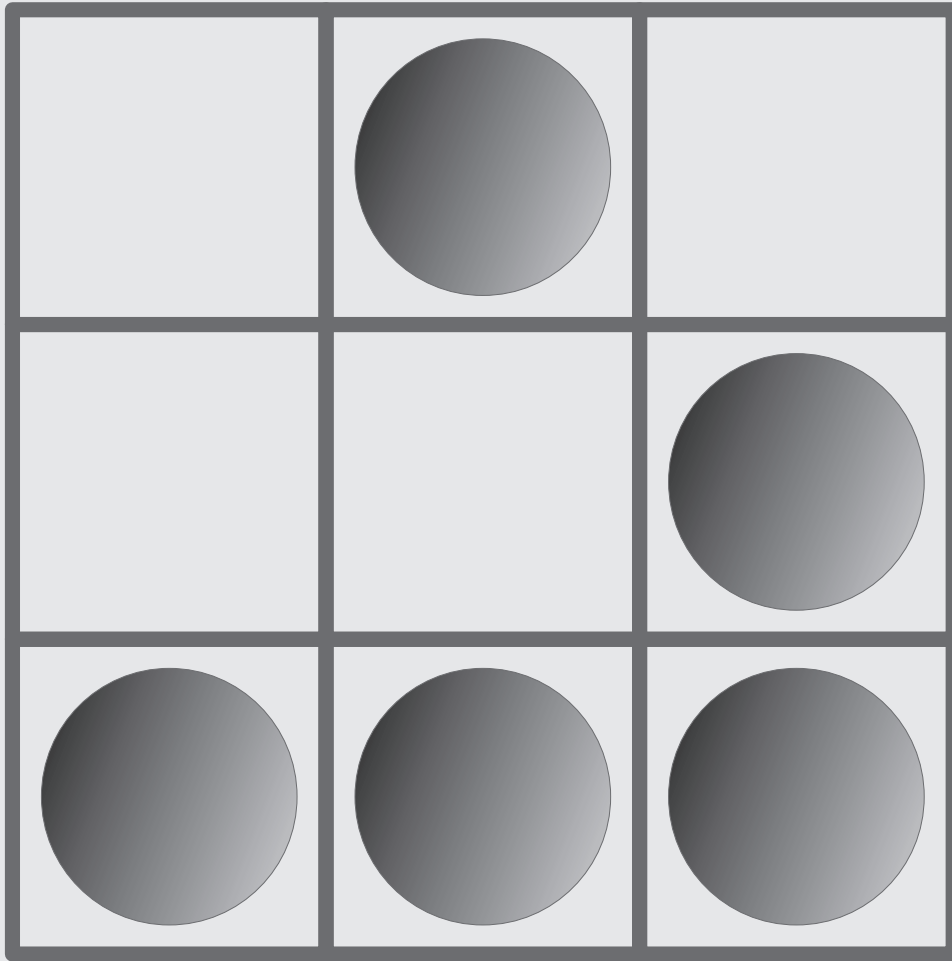


# Práctica 4

GPGPU Clásica: El Juego de la Vida



**Máster Oficial en  
Informática Gráfica,  
Juegos y  
Realidad Virtual**

**Procesadores  
Gráficos**



Universidad  
Rey Juan Carlos

*“If you have the opportunity to play this game of life  
you need to appreciate every moment.  
A lot of people don't appreciate the moment until it's passed”  
—Kanye West*

## Objetivo de la práctica

El objetivo de esta práctica es un primer acercamiento al mundo de la Computación General en GPU (también conocido como GPGPU) desde el punto de vista del cauce clásico. En numerosas ocasiones en clase se ha destacado el gran potencial que tienen los chips gráficos desde el punto de vista de cálculo potencial en coma flotante. Esta práctica nos ayudará a comprender las dificultades con las que se enfrentan los programadores de efectos gráficos cuando tienen que diseñar un algoritmo para el que las tarjetas gráficas no están pensadas en un principio. También ilustraremos cómo se hace la comunicación de datos entre puntos que no siguen el sentido natural del cauce gráfico, y cómo es la comunicación entre CPU y GPU.

La situación ha mejorado notablemente con la aparición de la arquitectura unificada. Así que esta práctica nos sirve tanto para dar el primer paso hacia la introducción de la segunda parte de la asignatura, como para contrastar las diferencias que tenemos en las distintas aproximaciones en la programación de estas arquitecturas.

En principio la práctica iba a girar alrededor de un sofisticado ejemplo de cálculo de dinámica de fluidos. Dado que este tipo de efectos se explican en la asignatura de Animación Avanzada del tercer cuatrimestre, el volumen de la explicación de los detalles matemáticos que hay que adelantar para poder llevar a cabo la simulación oscurecía completamente los mecanismos de comunicación entre CPU y GPU que se pretenden ilustrar con esta práctica de iniciación en la GPGPU, razón por la cual la práctica de simulación de fluidos se realizará en lugar de la práctica 10 anteriormente prevista, y en la práctica 4 nos conformaremos con la realización de un efecto de textura dinámica mediante la aplicación de autómatas celulares.

## Planteamiento

El enunciado de la presente práctica consta de cuatro partes.

En la primera parte expondremos las ideas que han dado lugar a la GPGPU y unas recetas a nivel muy general de cómo se debe afrontar este tipo de desarrollos en GPGPU clásica.

En la segunda parte analizaremos la forma en la que retroalimentamos el cauce para poder llevar a cabo este algoritmo en la tarjeta gráfica y estudiaremos las ventajas e inconvenientes de este mecanismo.

En la tercera parte contemplaremos la nueva forma de encajar los problemas numéricos en GPGPU de cauce clásico y nos centraremos en la comunicación entre GPU y CPU para poder recuperar los datos de los cálculos que deseamos reaprovechar en la CPU, de modo que podamos utilizar la GPU como si fuera un potente coprocesador. En lugar de dar una explicación puramente teórica, construiremos bloque a bloque un primer código de ejemplo completamente funcional que realice una operación algebraica sencilla para ilustrar los

mecanismos de comunicación entre CPU y GPU, así como la nueva forma de plantear los algoritmos de forma que se aproveche el potencial de cálculo que tenemos en el chip gráfico.

En la cuarta parte explicaremos el algoritmo del autómata celular del juego de la vida del matemático británico John Horton Conway, y trabajaremos con el cómo si de una textura dinámica se tratara.

### IMPORTANTE:

- En esta práctica habrá que entregar una memoria escrita en la que se respondan claramente a las preguntas y cuestiones propuestas en cada apartado.
- La entrega se realizará a través del Campus Virtual
- Es necesario incluir el código fuente o, al menos, indicar las modificaciones realizadas sobre el código existente.
- Será absolutamente **obligatorio** comentar los resultados obtenidos.

## Lecturas previas

Con el fin de adecuar la asignatura al nuevo espacio de europeo de enseñanza superior, y dado el reducido tiempo presencial que tenemos en la asignatura, parte de la explicación relativa a los detalles de implementación y la nueva forma de planear los algoritmos en la arquitecturas de *streaming* se deja como ejercicio al alumno para que profundice por su cuenta. Y el tiempo de clase lo aprovecharemos para concentrarnos en resolver las posibles dudas que hayan surgido a partir de las lecturas y “cacharrear” realizando los ejercicios.

Estos textos pueden consultarse en la biblioteca o bien descargarse desde la página web de la asignatura<sup>1</sup>:

- GPU Gems 2, editado por Matt Pharr
  - o Capítulo 30: The GeForce 6 Series GPU Architecture
  - o Capítulo 31: **Mapping computational concepts to GPUs**
  - o Capítulo 32: Taking the plunge into GPU computing
  - o Capítulo 33: Implementing efficient parallel data structures on GPUs
- OpenGL Shader Language (The Orange Book), de Randi J. Rost:
  - o Capítulo 1: A Review of OpenGL Basics
- Tutoriales de NeHe

## Índice del enunciado de la práctica

Objetivo de la práctica.....	1
Planteamiento .....	1
Lecturas previas.....	2
Índice del enunciado de la práctica.....	2
Normativa del laboratorio.....	4
Recomendaciones .....	5
1. Introducción a la GPGPU clásica .....	6
1.1 El cauce clásico .....	6
1.2 Peculiaridades de la GPGPU clásica .....	8

<sup>1</sup> <http://dac.escet.urjc.es/rvmaster/asignaturas/PG/>

1.2.1 Elección del lenguaje de programación .....	8
1.2.2 Receta básica en GPGPU .....	8
1.2.3 Texturas y Arrays.....	9
1.2.4 <i>Shaders</i> en la GPU y bucles internos en la CPU.....	9
1.2.5 Invocación de la subrutina y renderizado .....	10
1.2.6 Coordenadas de textura y dominio del cálculo.....	10
2. Retroalimentación en el cauce clásico.....	10
3. Mecanismos de comunicación.....	11
3. 1 Configuración de OpenGL .....	11
3.1.1 GLUT .....	11
3.1.2 Extensiones de OpenGL.....	12
3.1.3 Configurar OpenGL para renderizar en un lugar distinto a la pantalla .....	12
3.2 Primer símil: Arrays y texturas .....	13
3.2.1 Creación de arrays en la GPU .....	14
3.2.2 Creación de texturas en coma flotante en la GPU .....	14
3.2.3 Correspondencia uno a uno del índice del array a las coordenadas de textura.....	16
3.2.4 Utilización de texturas como render targets.....	17
3.2.5 Transferencia de datos desde los arrays de la CPU a las texturas de la CPU.....	19
3.2.6 Transferencia de los datos de las texturas de la GPU a los arrays de la CPU.....	19
3.2.7 Un pequeño programa de ejemplo .....	20
3.3 Segundo símil: Kernels y <i>shaders</i> .....	21
3.3.1 Implementación basada en bucles de la CPU frente a la implementación orientada a kernels y paralelismo de datos de la GPU.....	21
3.3.2 Creación de un <i>shader</i> en lenguaje Cg .....	23
3.3.3 Configuración del Cg runtime.....	24
3.4 Tercer símil: Computación y dibujado.....	25
3.4.1 Preparación del kernel computacional .....	25
3.4.2 Establecimiento de los arrays / texturas de entrada .....	25
3.4.3 Establecimiento de los arrays / texturas de salida.....	25
3.4.4 Realización del cálculo .....	25
3.5 Cuarto símil: Retroalimentación .....	27
3.5.1 Renderizado en múltiples pasadas.....	27
3.5.2 La técnica “ping pong” .....	27
3.5.3 Algunos detalles sobre el código fuente de ejemplo que acompaña esta sección....	28
4. El juego de la vida .....	28

4.1 Origen.....	29
4.2 Reglas .....	30
4.3 Ejemplos de patrones.....	30
5. Implementación del juego de la vida en GPU y cuestiones .....	33
5.1 Cuestiones y mejoras del código.....	33
Forma y fecha de entrega.....	34
Nota aclaratoria .....	34

## Normativa del laboratorio

- El criterio más importante es la funcionalidad: un programa que funciona siempre tiene más posibilidades de llevarse una buena puntuación; no se valorarán aquellos programas que no funcionen. Una práctica o proyecto modesto será evaluada mucho más favorablemente que un "proyecto" ambicioso que sólo da *core-dumps*. Los siguientes criterios que se tendrán en cuenta (y que hay que cuidar al realizar las prácticas) son:
  - La manera de resolver el problema con el programa
  - Estructuras de datos y diseño de los algoritmos
  - Claridad y documentación en el código
  - Eficiencia y elegancia en la implementación.
- ¡Por favor, no hagáis trampas! Se procura alentar el diálogo y el trabajo en equipo, pero por favor trabajad de forma independiente (a menos que el trabajo sea en grupos). Trabajos muy similares serán considerados como copias, a menos que la naturaleza lo pedido sea tan restrictiva que justifique las similitudes. Y una copia implica el suspenso automático. Simplemente piénsalo de esta manera: hacer trampas dificulta el aprendizaje y la diversión de conseguir hacerlo. Es vuestra responsabilidad proteger vuestro trabajo y asegurarnos que no se convierte en el de otro.
- Si se utiliza (o mejora) código fuente u otro material obtenido a través de internet, la biblioteca... debe darse el crédito a los autores y pedir permiso de ser necesario (si tiene una licencia restrictiva). Tomar código de un libro o de internet sin tener en cuenta estas consideraciones será considerado copia.
- Está terminantemente prohibido la práctica de técnicas de *overclocking* en las tarjetas gráficas del laboratorio, así como desbloquear los procesadores de vértices y fragmentos de los chips gráficos. Este tipo de acciones pueden dañar físicamente el equipo del laboratorio y los alumnos responsables serán amonestados severamente.
- Las prácticas (código y memoria explicativa) deberán entregarse en los plazos indicados con las herramientas del portal de la asignatura en el Campus Virtual. La recepción de los trabajos a través del Campus Virtual restringe el periodo en el que se pueden enviar. Por favor, organizaros bien para evitar retrasos.

## Recomendaciones

En el enunciado de esta práctica daremos por supuesto que los alumnos tienen ciertos conocimientos de OpenGL, y su mecanismo de máquina de estados, así como del cauce clásico ya introducidos en las asignaturas de Informática Gráfica y Procesadores Gráficos.

En principio la práctica se puede hacer en parejas, si alguien no encuentra pareja o por problemas de horario tiene complicado el poder equilibrar el trabajo también es posible hacerla de forma individual, esta práctica es sencilla y no debería suponer una dificultad añadida el no tener compañero/a para realizarla.

Esta práctica ha sido diseñada de modo que pueda realizarse sin muchos problemas en las dos horas de tiempo de laboratorio que se nos han asignado. Si algún alumno no tuviera tiempo para terminarla y no dispone del hardware gráfico necesario para realizarla tiene dos opciones:

- Puede realizarla cuando le resulte más cómodo en el aula de libre acceso S09 del edificio de Laboratorios II (Campus de Móstoles) que dispone de 15 ordenadores con tarjeta GeForce7900. Indícale al responsable del aula que necesitáis un ordenador con este hardware para realizar la práctica y os dirá podéis utilizar.
- El hardware necesario para poder realizar esta práctica es una nVIDIA GeForce FX una ATI Radeon 9500. Las tarjetas más antiguas no dan la posibilidad de almacenar y realizar cálculos en formatos de coma flotante. Por ello la opción alternativa sería utilizar NVemulate (que puede conseguirse a partir de un enlace en la página web de la asignatura<sup>2</sup>), que es un pequeño programa que emula el *driver* de la tarjeta gráfica y os permite probar diferentes perfiles. La versión que a aparecido a principios de Octubre del 2006 soporta incluso las extensiones para el Shader Model 4.0 de la nueva serie de tarjetas de nVIDIA<sup>3</sup>. Como es lógico, al emular sus características por software es muy muy lento, y se recomienda la primera de las opciones si os podéis desplazar al Campus.

La búsqueda de errores (*debugging*) en los programas que implican la comunicación entre CPU y GPU es bastante compleja, ya que la GPU actúa como caja negra y el driver no aporta demasiada información. Por ello es importante trazar cada una de las funciones y llamadas a OpenGL que hagamos. En el caso de utilizar Cg, este trozo de código puede resultar muy útil:

```
void checkGLErrors(const char *label) {
    GLenum errCode;
    const GLubyte *errStr;
    if ((errCode = glGetError()) != GL_NO_ERROR) {
        errStr = gluErrorString(errCode);
        printf("OpenGL ERROR: ");
        printf((char*)errStr);
        printf(" (Label: ");
        printf(label);
        printf(")\n.");
    }
}
```

<sup>2</sup> <http://developer.nvidia.com/object/nvemulate.html>

<sup>3</sup> Durante el mes de Febrero aparecerá en el mercado la serie 9 de nVIDIA, en el que se supone que algunos modelos darán soporte al Shader Model 4.1, y por tanto aparecerá una nueva versión de NVemulate con estas nuevas características.

## 1. Introducción a la GPGPU clásica

El juego de la vida fue uno de los primeros ejemplos de programación genérica en GPU, más conocida como GPGPU. Con este ejemplo, Mark Harris, pretendía ilustrar las capacidades que tiene la GPU para realizar cálculos dentro y también fuera del ámbito de la síntesis de gráficos pura. Este ejemplo estaba orientado a la creación de texturas dinámicas (procedurales) en tiempo real<sup>4</sup>. Pronto le siguieron otros pequeños *shaders* que ampliaban esas posibilidades hacia métodos numéricos aritméticamente intensivos en los que la dependencia entre datos era baja, como la resolución de ecuaciones en derivadas parciales para la simulación de fenómenos físicos.

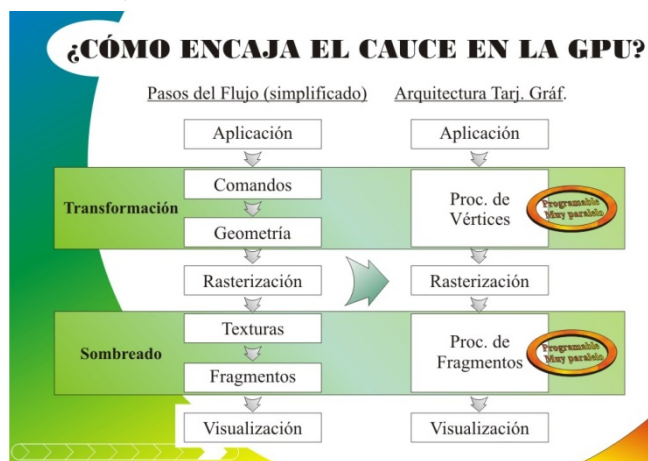
El campo de la GPGPU tuvo una calurosa aceptación por parte de la comunidad científica, ya que abría las puertas a unos sistemas muy baratos, disponibles en cualquier ordenador personal con un potencial de cálculo muy alto y un crecimiento exponencial. Algo que no tenía rival frente a las otras arquitecturas de aquella época (ni tampoco con las que disponemos ahora).

Sin embargo su programación ha sido durante mucho tiempo una tarea especialmente ardua y compleja, ya que las tarjetas gráficas y su cauce clásico<sup>5</sup> están únicamente pensados para realizar cálculos para la síntesis de gráficos geométricos 3D y nada más. Lo que obliga a sus programadores a adaptar sus algoritmos a la secuencia segmentada del cauce, y para lidiar con la tarjeta gráfica es necesario conocer su arquitectura a muy bajo nivel si queremos aprovechar sus características potenciales.

Así fue creciendo el movimiento de la GPGPU hasta lo que hoy día es un campo de desarrollo e investigación muy importante, hasta el punto de que los planes de negocio de nVIDIA, AMD-ATI, IBM e Intel han cambiado debido al potencial que tienen las arquitecturas que se utilizan en procesadores gráficos para realizar grandes cantidades de cálculos en coma flotante, lo que nos dirige cada vez más rápido a los llamados procesadores heterogéneos.

### 1.1 El cauce clásico

Como ya hemos visto en clase en repetidas ocasiones, el cauce clásico tardó muchos años en establecerse, y una vez se convirtió en un estándar, empresas con una gran visión de futuro –como 3DFX- decidieron crear chips aceleradores gráficos que aliviaban a la CPU de tener que realizar las tareas más tediosas y computacionalmente intensivas del renderizado. Se comenzó por las etapas finales del cauce, hasta poco a poco completar todo el cauce en forma de chip gráfico a medida que se iban sucediendo las generaciones de tarjetas, hasta brindar la posibilidad de reconfigurar y programar parte de las etapas de transformación y sombreado.



<sup>4</sup> Como puede verse en la presentación “Dynamic Textures” que hizo en nVIDIA, poco después de terminar la tesis y ser contratado por esta empresa. Hoy día, Mark Harris es considerado uno de los precursores y máximos exponentes del movimiento GPGPU. También es el webmáster de GPGPU.ORG

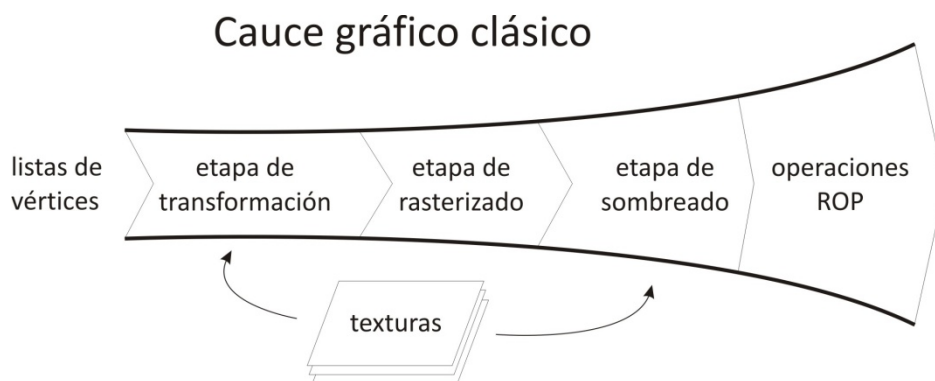
<sup>5</sup> Esto ha cambiado en el planteamiento de la arquitectura unificada con la aparición de soluciones como CUDA, CTM, RapidMind...

El chip gráfico, es por tanto, la representación hardware del cauce clásico, que gracias a elaboradas estrategias de arquitectura de computadores logra tener un *throughput* que nos permite realizar todo los cálculos necesarios para poder sintetizar la representación 2D de complejas escenas 3D en tiempo real. Este tipo de hardware se basa en el modelo de la arquitectura de *streaming* y un cauce tremendamente segmentado (el procesador de fragmentos de la etapa de sombreado puede llegar a tener más de 800 etapas, algo impensable en arquitecturas orientadas a control / instrucciones como las de Von Neumann).

Pero esta gran potencia de cálculo tiene un coste en generalidad. Los chips gráficos son conjuntos de procesadores muy especializados que no están pensados para tareas que vayan más allá de la representación de datos tridimensionales geométrico-superficiales (lo que excluye muchas otras formas de representación gráfica<sup>6</sup>).

Si queremos aprovechar los chips gráficos para realizar otro tipo de cálculos (como es el caso de GPGPU) es necesario que nos adaptemos a la arquitectura, características y limitaciones del cauce gráfico. Lo que nos obliga a realizar un fuerte cambio de planteamiento en el diseño de los algoritmos.

En el cauce clásico sólo disponemos de listas de vértices con atributos (como el color, las coordenadas de textura, el vector normal...) y texturas para introducir los datos en nuestros algoritmos.



De modo que tenemos que adaptar también nuestras estructuras de datos para que se adecúen a las dos estructuras que podemos consultar en las dos zonas programables del cauce.

Así traduciremos en geometría aquellos datos que deseemos que se procesen en la etapa de transformación en los procesadores de vértices. También sería posible consultar texturas, pero para ello necesitamos que la tarjeta gráfica cumpla las especificaciones del *shader model 3.0* y, como toda consulta de datos a memoria de vídeo, tiene un cierto impacto en el rendimiento cuando muchos de los procesadores realizan peticiones simultáneamente, por lo que es preferible que se utilicen aquellas estructuras que de forma natural fluyen por el cauce.

Y las texturas serán más utilizadas en la etapa de sombreado por parte de los procesadores de fragmentos como si se tratase de la consulta a un array multidimensional mediante las coordenadas de texturas.

Esta etapa es la preferida por los programadores de GPGPU, no sólo por su mayor capacidad de cómputo —ya que debido a la mayor cantidad de datos en esta etapa se dedica un 80% de los transistores del chip gráfico a sus procesadores para equilibrar la carga en aplicaciones gráficas— sino porque su salida es mucho más directa.

<sup>6</sup> Como la basada en imagen, la volumétrica, la basada en puntos...

## 1.2 Peculiaridades de la GPGPU clásica

El campo de la GPGPU clásica es uno de los más dados a optimizaciones extrañas y recetas empíricas, a pesar del rigor con el que ha evolucionado en los últimos años en el seno de la comunidad científica. En buena parte esto es debido al celo con el que los fabricantes de chips gráficos guardan la información relativa al funcionamiento de sus drivers, hasta el punto de no revelar más que lo justo e imprescindible para programar a través de las APIs gráficas, de manera que sus arquitecturas queden en el más estricto secreto industrial.

Por ello se han ideado multitud de aproximaciones a la hora de llevar algoritmos de programación genérica a la GPU. En esta práctica veremos la estrategia más simple posible como un primer acercamiento. Aquellos alumnos que deseen profundizar en este tema, descubrirán el elevado grado de sofisticación y la forma en la que hay que “retorcer” algoritmos más complejos para adaptarlos a las restricciones del cauce, razón por la cual, hasta hace bien poco, el escribir un nuevo algoritmo de forma eficiente en la GPU era considerado de gran interés para la comunidad y publicado en revistas de renombre con cierta facilidad.

### 1.2.1 Elección del lenguaje de programación

Dado que el objetivo es aprovechar al máximo la capacidad potencial que nos brinda el chip gráfico, nos interesa programarlo a muy bajo nivel. De ahí que en esta práctica sigamos utilizando como lenguaje *shader* Cg. Es cierto que existen otras alternativas más orientadas a la programación genérica de procesadores de cauce clásico como Brook o Sh, pero el nivel de abstracción que tienen para facilitar su aprendizaje, también nos aleja de todas las posibilidades y el jugo que se puede obtener.

## PROYECTANDO CONCEPTOS (IX)


### ANALOGIAS CPU-GPU

Para no sentirnos tan perdidos a la hora de comprender la arquitectura y la forma de programación en GPUs vamos a hacer una serie de símiles que nos ayuden a relacionar e identificar las ideas de programación gráfica tradicional (en CPU) con sus correspondientes mecanismos en las GPUs (salvando las distancias)

- Texturas en GPU = Arrays en CPU

También tenemos arrays de vértices, pero el similitud es más patente en el procesador de fragmentos (que también es más versátil)

No podemos reservar memoria dinámicamente, dependemos de la configuración que se haya establecido en la API/interfaz (OpenGL, DirectX...)



Capítulo 31: Mapping computational concepts to GPUs

### 1.2.2 Receta básica en GPGPU

En el apartado 3 de esta práctica profundizaremos con ejemplos en esta primera aproximación que exponemos. La idea básica es llegar a realizar el cálculo a través del renderizado de una geometría simple<sup>7</sup>. Para ello se procura que haya una correspondencia directa entre nuestras fuentes de datos (que generalmente son texturas) y los fragmentos que hacen uso de ellas (para luego convertirse en píxeles en el framebuffer).

Los pasos se pueden resumir en:

1. Establecimiento de un *viewport* con una correspondencia 1 a 1 entre t́xeles y futuros píxeles
2. Creación y asociación de texturas del tamaño determinado en el paso anterior
3. Transferencia de los datos de entrada a las texturas
4. Carga y asociación de los *shaders* (en general *shaders* de fragmentos) que se comportarán como kernels computacionales del modelo *streaming*
5. Renderizado de una geometría simple para llevar a cabo el cálculo

<sup>7</sup> Geometrías más complejas nos pueden permitir realizar algoritmos más sofisticados.

6. Mostrar por pantalla o recuperar el resultado para iterar con él

### 1.2.3 Texturas y Arrays

Ya hemos comentado en clase las características de la interfaz de memoria de la tarjeta gráfica, en la que no podemos leer y escribir de las mismas estructuras de datos en cauce clásico. La lectura puede realizarse de los datos que vienen a través del cauce como atributos de las estructuras de datos que “sobreviven” a las etapas anteriores, y de texturas.

Las texturas son una forma muy cómoda y socorrida para almacenar información de entrada, aunque su utilización indiscriminada puede reducir el rendimiento del algoritmo al saturar el bus de memoria innecesariamente. En ese último caso estaríamos ante una aplicación limitada en ancho de banda, que por tanto no puede aprovechar toda la capacidad computacional de los procesadores del chip gráfico.

Las texturas tienen una estructura muy similar a la de los arrays, son accedidos mediante coordenadas de textura que vienen heredados como parte de los atributos de los vértices que son interpolados de los vértices originales, en lugar de simples índices como en los arrays de la CPU.

Este tema se tratará en mayor profundidad, con ejemplo, en la sección 3.2

### 1.2.4 Shaders en la GPU y bucles internos en la CPU

Aquellos que no conocen el funcionamiento y la arquitectura interna de la GPU pueden encontrar muy extraño que los chips gráficos estén basados en un tipo de arquitectura que se “lleva mal” con las ramificaciones condicionales (aunque a partir del *shader model 3.0* se supone que tienen soporte, éste es muy precario). Ya que en la CPU los algoritmos de procesamiento de imagen suelen recorrer todo el array que almacena los datos de la imagen mediante bucles anidados.

Estos bucles no son necesarios en la GPU, ya que el rasterizador realiza la labor de “fragmentar” el problema y un conjunto de *dispatchers* no programables (ni configurables) reparten las estructuras de datos resultantes a los procesadores de fragmentos. Así, estos procesadores sólo tienen que aplicar la parte del algoritmo que se encontraría dentro del bucle más interno del código equivalente en la CPU.

Estudiaremos estos conceptos en profundidad en el apartado 3.3

**PROYECTANDO CONCEPTOS (X)**  
**ANALOGIAS CPU-GPU**

- Programas de Fragmentos en la GPU = Bucles internos en CPU

Virtualmente podemos imaginar que tenemos un procesador independiente por “pixel” (fragmento).

En los programas en CPU al hacer tratamiento de señal 2D o proc. de imagen (como aplicar un filtro convolucionando) se suele tener una serie de bucles anidados en los que el código que afecta a cada pixel se encuentra en el más interno

Ese “trozo de código” vendría a ser nuestro programa del procesador de fragmentos

**PROYECTANDO CONCEPTOS (XI)**  
**ANALOGIAS CPU-GPU**

- Render to Texture = Retroalimentación  
Habitualmente los cálculos tienen dependencias entre elementos, pero se puede dividir el algoritmo en varios pasos y utilizar este mecanismo para utilizar el flujo de salida como entrada en los distintos kernels
- Rasterización de la geometría = Invocación de “la rutina”  
Para invocar el procesamiento de los cálculos tenemos que dibujar una geometría. El procesador de vértices transforma la geometría el rasterizador determina qué píxeles del buffer de salida “cubrirá” y genera un fragmento para cada uno.

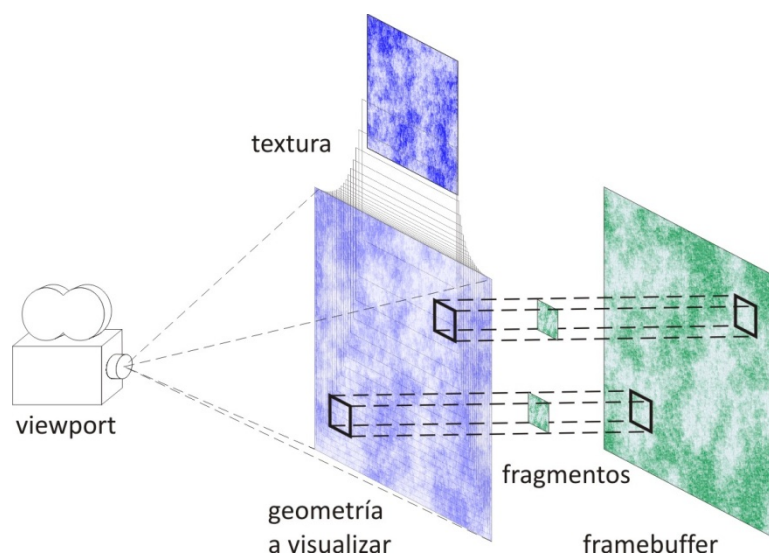
Lo más normal es que en GPGPU es tener un grid cuadrado

### 1.2.5 Invocación de la subrutina y renderizado

Dado que estamos utilizando una arquitectura fuertemente orientada al procesado de gráficos, la forma en la que llevamos a cabo los cálculos se traduce en lo que sería una o varias pasadas de renderizado. Por tanto no existen “invocaciones a subrutinas” sino que los cálculos se suceden en cada *frame* para la geometría que hayamos decidido representar.

### 1.2.6 Coordenadas de textura y dominio del cálculo

Dado que en este caso de iniciación forzamos una correspondencia 1 a 1 entre los datos de entrada (la textura) y los datos de salida (fragmento y píxeles a la salida del framebuffer), tendremos que preparar la forma de mapear la textura sobre la geometría de manera que cuando el rasterizador interpole las coordenadas de textura a lo largo de la geometría, cada fragmento reciba la posición del centro del t́xel de los datos de entrada. En este ejemplo haremos la proyección de la forma más simple posible mediante un quad.



## 2. Retroalimentaci3n en el cauce clásico

Las arquitecturas *streaming* puras est́n ideadas como cauces muy segmentados en los que los datos siempre evolucionan en el mismo sentido, raz3n por la cual no se admiten bucles ni retroalimentaciones que puedan alterar dinámicamente la velocidad de una determinada etapa. El modelo del cauce gráfico clásico sigue estas ideas, pero se deja una puerta medio abierta al permitir que el resultado de una renderizaci3n pueda ser reutilizado como parte de los datos de entrada del siguiente renderizado. Esto es lo que ha dado lugar a la representaci3n multipasada, que se ha convertido en una práctca extremadamente com3n en

videojuegos, al dotar de una flexibilidad al cauce que permite componer distintas capas (pasadas de cálculo) para obtener una escena más rica y elaborada.

Sin embargo, este mecanismo de retroalimentación es muy precario en el cauce clásico, ya que nos tenemos que limitar a las estructuras de datos mencionadas (listas de vértices y texturas) y la salida sólo puede convertirse en una textura y consultarse como tal (en pasadas posteriores).

Además, este mecanismo reduce drásticamente el rendimiento de la representación ya que la latencia del cauce es muy grande y tenemos que esperar a que se complete y comunicarnos con la CPU para poder iniciar la siguiente pasada (lo que tiene una fuerte penalización).

Esta técnica fue demandada por los programadores gráficos durante años y finalmente tuvieron respuesta. De modo que en cierto modo podemos tener una cierta comunicación entre los procesadores de una pasada a la siguiente (es posible consultar el estado en el renderizado anterior).

Los fabricantes de chips gráficos han habilitado dos conjuntos de extensiones que permiten reutilizar la salida del cauce como textura en los procesadores: los pixel buffers y los framebuffer objects, que no se explican en esta sección ya que se deja al alumno esa investigación como parte de las preguntas de esta práctica.

### 3. Mecanismos de comunicación

Dado que estos conceptos de GPGPU nos pueden parecer todavía extraños, ya que en clase de Procesadores Gráficos nos hemos centrado en las aplicaciones gráficas, vamos a volver a ver paso a paso la programación de los mecanismos de comunicación del cauce clásico junto con los trozos de código que nos van a permitir explotar este potencial.

Esta sección está fuertemente inspirada en el magnífico tutorial de Dominik Goddeke de programación GPGPU, muchas de las subsecciones –incluido el código– se han tomado y adaptado del mismo con su permiso.

#### 3.1 Configuración de OpenGL

##### 3.1.1 GLUT

La librería GLUT (OpenGL Utility Toolkit) nos proporciona funciones para poder gestionar los eventos de ventanas, crear sencillos menús, etc. En nuestro caso de estudio, simplemente crearemos un contexto de OpenGL, a través del cual nos comunicaremos con la tarjeta gráfica para mandarle y recoger los datos que nos interesa procesar.

```
// include the GLUT header file
#include <GL/glut.h>

// call this and pass the command line arguments from main()
void initGLUT(int argc, char **argv) {
    glutInit ( &argc, argv );
    glutCreateWindow("SAXPY TESTS");
}
```

### 3.1.2 Extensiones de OpenGL

La mayor parte de las funcionalidades que necesitamos para realizar cálculos en coma flotante en la GPU aún no forman parte del núcleo de OpenGL<sup>8</sup>, por lo que hemos de recurrir a las extensiones<sup>9</sup> para utilizar estas características de la GPU mediante la API.

Como ya se ha comentado en clase, el estándar OpenGL ya no es llevado de la mano de Silicon Graphics, sino que desde el año 2000 un comité técnico -llamado Grupo Kronos<sup>10</sup>- se encarga de él. Dentro de este grupo están representadas empresas con gran peso en la industria de los gráficos, de muy distintas ramas (hardware: AMD, Intel, Texas Instrument, nVIDIA, SGI, Sun...; software: Discreet...) y con intereses económicos muy dispares. Este último factor hace que en no pocas ocasiones no lleguen a un consenso en lo que debería estar o no estar como estándar base, por esa razón se permite el realizar extensiones a la versión original, que pueden ser diferentes para cada uno de los fabricantes.

Estas extensiones le permiten cierto dinamismo y adaptabilidad en un sector en el que el hardware cambia muy rápidamente, pero también obliga a los programadores a conocer las discrepancias entre las soluciones que proponen los distintos fabricantes. Cuando una extensión es comúnmente aceptada por los desarrolladores, el comité se plantea su integración en el núcleo y el resto de fabricantes han de darle soporte en sus drivers y chips gráficos.

Sin embargo, no todo el hardware que está disponible en el mercado es compatible con las extensiones<sup>11</sup> y por ello hay que tener cuidado a la hora de usarlas y comprobar antes si el ordenador en el que se ejecuta el programa lo soporta antes de hacer esas llamadas a la API (y buscar un camino alternativo en el caso de no disponer de ellas). En esta práctica supondremos que el perfil de las tarjetas es suficientemente alto para las extensiones a utilizar.

Utilizaremos GLEW como librería para poder acceder a estas extensiones en OpenGL, ya que envuelve todo lo que necesitamos en esta práctica con una interfaz minimalista.

```
void initGLEW (void) {
    // init GLEW, obtain function pointers
    int err = glewInit();
    // Warning: This does not check if all extensions used
    // in a given implementation are actually supported.
    // Function entry points created by glewInit() will be
    // NULL in that case!
    if (GLEW_OK != err) {
        printf((char*)glewGetErrorString(err));
        exit(ERROR_GLEW);
    }
}
```

### 3.1.3 Configurar OpenGL para renderizar en un lugar distinto a la pantalla

El final del cauce clásico es, por lo general, el framebuffer. Allí es a donde se destinan las estructuras de datos que han sobrevivido a los distintos tests ya en forma de fragmentos, y se convierten en píxeles (información de color) que se almacena en un área reservada de

<sup>8</sup> Es posible que OpenGL 3.0 ya las incluya, pero a día de hoy todavía la especificación no se ha hecho pública.

<sup>9</sup> <http://opengl.org/resources/features/OGLExtensions/>

<sup>10</sup> <http://www.khronos.org/>

<sup>11</sup> En <http://www.opengl.org/registry/> podéis encontrar una lista de todas las posibles extensiones.

memoria de vídeo que alternativamente es leída por el RAMDAC y traducida a señales que se plasman en el monitor del PC. En general, la representación que sale a pantalla se suele hacer en RGB de 24 bits (8 para cada canal) y un canal extra A para las transparencias, lo que suman más de 16 millones de posibles colores. Sin embargo, a la hora de operar en coma flotante, 8 bits por componente se nos quedan muy cortos, ya que no podemos conformarnos con un rango entre<sup>12</sup> [0,1] y, aunque así fuese, en este tipo de representación el rango/exponente se determina de forma explícita, lo que consume varios bits y reduce la precisión que podemos alcanzare con los cálculos.

Las GPUs modernas nos evitan tener que trabajar con modelos aritméticos complejos, construidos a partir de enteros de 8 bits, mediante la utilización de extensiones de OpenGL que nos permiten dar salida a los datos en 32 bits hacia el *framebuffer* e incluso cambiar el “target” de renderizado hacia una zona de memoria que no es consumida por el RAMDAC sino que se preserva de forma semejante a una textura. Es la llamada `EXT_framebuffer_object` (FBO), y facilita el uso de valores en coma flotante de 32 bits sin tener que preocuparnos de las acotaciones de los valores que se hacen en algunas partes de la implementación hardware del cauce.

Para aplicar esta extensión y desactivar el *framebuffer* tradicional bastan unas líneas de código. Es importante destacar que al volver a asociar el FBO número 0 volveremos a activar el *framebuffer* en cualquier momento.

```
GLuint fb;

void initFBO(void) {
    // create FBO (off-screen framebuffer)
    glGenFramebuffersEXT(1, &fb);
    // bind offscreen buffer
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
}
```

### 3.2 Primer símil: Arrays y texturas

La disposición lineal de la memoria en los arrays es muy típica de la programación en CPU, así cuando trabajamos con arrays de un número mayor de dimensiones se accede a los valores mediante un desplazamiento en una estructura de datos que en realidad es lineal. Por ejemplo, si tuviésemos un array bidimensional de MxN y quisiéramos acceder a un determinado valor, en lenguajes como C y C++ podríamos hacerlo con la convención de acceso con índices `a[i][j]` o bien tratando el inicio de este array como el puntero que es y calculando el desplazamiento necesario `*(a + i*M + j)`.

En GPUs la disposición de los datos es diferente porque su arquitectura está optimizada en el acceso de estructuras tipo array bidimensionales. Los arrays unidimensionales y tridimensionales también están soportados, pero tienen una fuerte penalización en su aplicación.

---

<sup>12</sup> En realidad, al llegar al framebuffer el rango se traduce linealmente a enteros entre [0, 255], lo que también resulta insuficiente

A estas estructuras de datos, que nos recuerdan a arrays, las llamamos comúnmente texturas ó *texture samples*. Y su tamaño viene limitado por el *shader model* y la propia API<sup>13</sup>, lo que podemos comprobar con el siguiente pedazo de código:

```
int maxtexsize;
glGetIntegerv(GL_MAX_TEXTURE_SIZE, &maxtexsize);
printf("GL_MAX_TEXTURE_SIZE, %d\n", maxtexsize);
```

Las tarjetas actuales rondan los 4096 ó 8192 por dimensión, en cualquier caso hay que tener cuidado con las texturas tridimensionales, porque aunque en teoría están soportadas, la memoria de vídeo se nos puede quedar corta.

Del mismo modo en el que antes hablábamos de los índices de los arrays en la CPU, las texturas son accedidas mediante coordenadas de textura (u,v). Estas coordenadas pueden tener cualquier valor real dentro del dominio de la textura (en general está acotado entre [-1, 1]), y la unidad de acceso a memoria de texturas se encargará de devolvernos el valor aproximado más parecido a lo que se ha solicitado, mediante una interpolación más o menos sofisticada, como si la textura fuera una función muestreada en  $R^n$ .

En el caso de GPGPU, las texturas almacenarán los datos que deseamos calcular y rara vez nos va interesar tener aproximaciones locales a la forma espacial que se han almacenado, por lo que es importante calcular las coordenadas de texturas de forma que se correspondan con el centro de las posiciones de los téxeles.

Además, las GPUs de cauce clásico trabajan con cuatro componentes a la vez, de modo que en las texturas se almacenan los cuatro canales de color en bloque (rojo, verde, azul y alfa). Esto puede y debe ser aprovechado en nuestras aplicaciones para mejorar su rendimiento.

### 3.2.1 Creación de arrays en la GPU

Para ilustrar los mecanismos de programación de la GPGPU vamos a hacer una operación lineal entre dos conjuntos de datos A y B de un tamaño N. Por ejemplo:  $y = y + \alpha x$   
En el caso de la CPU necesitaríamos dos arrays de números y un valor escalar en coma flotante.

```
float* dataY = (float*)malloc(N*sizeof(float));
float* dataX = (float*)malloc(N*sizeof(float));
float alpha;
```

Como deseamos hacer el cálculo en la GPU, hemos de convertir estos datos a una estructura que sea compatible con su arquitectura.

### 3.2.2 Creación de texturas en coma flotante en la GPU

En la CPU la decisión es simple, con dos arrays de números en coma flotante es suficiente, sin embargo en la GPU tenemos que hacerlo en forma de texturas en punto flotante y eso nos lleva a una elección entre una gama más amplia de posibilidades.

<sup>13</sup> Hay que tener un poco de cuidado con este aspecto, ya que hay ocasiones en las que la implementación de la API no se corresponde con las posibilidades que potencialmente nos ofrece el hardware y nos podemos encontrar con algún “bug” o funcionalidad a la que todavía no se le da soporte ni siquiera como extensión. Es algo atípico, pero que nos puede suceder en al tratar con texturas de tamaños muy grandes.

Si descartamos las texturas no nativas (de otras dimensiones) y nos quedamos con las bidimensionales, hemos de escoger entre dos posibilidades cuyas notables diferencias de uso se describen en la siguiente tabla:

	<b>Textura 2D</b>	<b>Textura rectangular</b>
<b>Texture target</b>	GL_TEXTURE_2D	GL_TEXTURE_RECTANGLE_ARB
<b>Coordenadas de textura</b>	Las coordenadas deben ser normalizadas al rango [0,1], independientemente de las dimensiones [0,M]x[0,N] de los texeles de la textura	Las coordenadas no están normalizadas
<b>Dimensiones de la textura</b>	Las dimensiones están limitadas a potencias de dos, a menos que el driver esté preparado para OpenGL 2.0 ó soporte la extensión ARB_non_power_of_two	Las dimensiones pueden tener cualquier tamaño

En cuanto al formato de la textura, las GPUs pueden trabajar con datos escalares, y vectores compuestos de varios datos: duplas, triplas, y cuádruplas. En esta sección ilustraremos la utilización de las texturas escalares y las de 4 elementos, al ser las opciones más comunes.

El caso más sencillo es el primero, con un único valor de punto flotante por texel, que en OpenGL se determina con GL\_LUMINANCE y consume 32 bits por texel.

En el segundo almacenamos 4 valores de coma flotante por texel lo que hace un total de 128 bits (16 bytes) a transferir y procesar de golpe, y se determina con GL\_RGBA.

Además tenemos tres extensiones de OpenGL que determinan el formato interno en coma flotante: NV\_float\_buffer, ATI\_texture\_float y ARB\_texture\_float. Cada extensión define una serie de constantes enumeradas (por ejemplo GL\_FLOAT\_R32\_NV) y símbolos (como 0x8880) que pueden ser utilizadas para crear y localizar las texturas.

En nuestro caso sólo nos serán útiles las constantes enumeradas GL\_FLOAT\_R32\_NV, para el almacenamiento de valores escalares en 32 bits, y GL\_FLOAT\_RGBA32\_NV, para cuádruplas. Las extensiones ATI\_texture\_float y ARB\_texture\_float son idénticas desde el punto de vista práctico, salvo por el hecho de que definen distintos valores de constantes enumeradas (GL\_LUMINANCE\_FLOAT32\_ATI, GL\_RGBA\_FLOAT32\_ATI y GL\_LUMINANCE32F\_ARB, GL\_RGBA32F\_ARB) para los mismos símbolos.

Por último tenemos que mapear los vectores de datos de la CPU en las texturas de la GPU. En el caso de las texturas con un valor por texel (formato de LUMINANCE) la forma más sencilla es meter los datos del vector de tamaño N en una textura de tamaño  $\sqrt{N} \times \sqrt{M}$ , suponiendo que N y M son potencias de 2. O de tamaño  $\sqrt{\frac{N}{4}} \times \sqrt{\frac{M}{4}}$  en el caso de formatos RGBA. Por ejemplo, un vector de 1048575 elementos cabrá en un array RGBA de 512x512.

	<b>NV3x</b>	<b>NV4x, G7x, G8x (RECT)</b>	<b>NV4x, G7x, G8x (2D)</b>	<b>ATI</b>
<b>Target</b>	Textura rectangular	Textura rectangular	Textura 2D	Textura 2D y rectangular

Formato	LUMINANCE y RGBA (y RG y RGB) <sup>14</sup>			
Formato interno	NV_float_buffer	NV_float_buffer	ATI_texture_float ARB_texture_float	ATI_texture_float ARB_texture_float

La reserve del espacio para una textura es muy fácil una vez ya hemos decidido el texture target, su formato y su formato interno:

```
// create a new texture name
GLuint texID;
glGenTextures (1, &texID);
// bind the texture name to a texture target
glBindTexture(texture_target, texID);
// turn off filtering and set proper wrap mode
// (obligatory for float textures atm)
glTexParameterf(texture_target, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameterf(texture_target, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameterf(texture_target, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameterf(texture_target, GL_TEXTURE_WRAP_T, GL_CLAMP);
// set texenv to replace instead of the default modulate
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
// and allocate graphics memory
glTexImage2D(texture_target, 0, internal_format,
             texSize, texSize, 0, texture_format, GL_FLOAT, 0);
```

En la última llamada a OpenGL cabe destacar varios parámetros. El primer 0 que aparece indica a OpenGL que no vamos a utilizar varios niveles de textura. El segundo de los 0 le indica que desactive los bordes de la textura porque no los necesitamos. El formato de textura le indica el número de canales a almacenar. GL\_FLOAT no tiene que ver con la precisión de los datos a almacenar en la GPU, sino con el formato de los datos que se quieren pasar desde la CPU. El último 0 le dice a OpenGL que en este momento no vamos a pasar los valores de la textura. Este trozo de código nos permite reservar el espacio necesario en memoria de vídeo para poder almacenar más tarde la textura, el símil con la CPU sería el de la función malloc().

Es importante pensar bien cuáles han de ser las características de las texturas a utilizar, ya que si no se hace bien, el algoritmo GPGPU tendrá un rendimiento notablemente inferior.

### 3.2.3 Correspondencia uno a uno del índice del array a las coordenadas de textura

Para poder ser capaces de decidir exactamente qué elementos de datos accedemos y calculamos en la textura que está almacenada en memoria, tenemos que prestar especial atención a las transformaciones que proyectan desde el mundo 3D (coordenadas del mundo) al 2D (espacio de pantalla). Esto es más sencillo si tenemos una correspondencia 1 a 1 entre los píxeles (que serán los fragmentos en los que queremos “renderizar” el resultado de las operaciones matemáticas) y los téxeles (de los que tomamos los datos de partida); y para ello podemos recurrir a una proyección ortogonal y un apropiado punto de vista que nos permita obtener esta relación entre las coordenadas geométricas (que utilizamos en las transformaciones), las coordenadas de texturas (con las que accedemos a los datos de entrada) y las posiciones de los píxeles (que serán nuestros datos de salida). Así pues la correspondencia estará basada en el único valor del que disponemos por ahora: el tamaño de la textura<sup>15</sup>.

<sup>14</sup> Estos son formatos de texturas que están soportados, pero pueden no ser válidos como render targets

<sup>15</sup> En el caso de texturas 2D será necesario un escalado adicional, pero no supone mayor dificultad

Para lograrlo establecemos la coordenada z a cero en el sistema de referencia del mundo y aplicamos el mapeado / correspondencia 1:1. Las siguientes instrucciones (o equivalentes) pueden añadirse al código de la rutina initFBO:

```
// viewport for 1:1 pixel=texel=geometry mapping
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, texSize, 0.0, texSize);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glViewport(0, 0, texSize, texSize);
```

### 3.2.4 Utilización de texturas como render targets

Las texturas no sólo se pueden utilizar como entrada de datos, sino también como salida mediante la extensión framebuffer\_object, esto es, podemos renderizar directamente a una textura. Pero no es posible hacerlo con la misma flexibilidad que en la CPU debido a que el interfaz hacia la memoria de vídeo no lo permite en la arquitectura *streaming* de las tarjetas de cauce gráfico clásico. Como se indicó en clase, las texturas son sólo de lectura o de escritura, pero no pueden tener habilitados ambos mecanismos a la vez.



Desde el punto de vista de la arquitectura esto tiene una clara explicación, ya que dentro del chip gráfico tenemos multitud de procesadores que están accediendo continuamente en paralelo y forma independiente a memoria de textura en busca de datos. El permitir lecturas y escrituras simultáneas requeriría un mecanismo de bloqueo para evitar leer un dato que está siendo modificado en ese momento por otro procesador. Eso no sólo requiere una gran cantidad de transistores para gestionar el control, sino que además tiene un impacto muy negativo en el rendimiento de la síntesis de gráficos<sup>16</sup>, razón por la cual en cauce clásico la lectura y escritura en la interfaz de memoria están completamente separadas (lo que a su vez también es coherente con la arquitectura *streaming* de este tipo de procesadores).



En nuestra aplicación de ejemplo necesitamos dos texturas de sólo-lectura en las que están los datos a procesar, y una tercera textura de sólo-escritura en la que guardaremos los resultados de los cálculos. De modo que el cálculo  $y = y + \alpha x$  pasará a ser  $y_{nuevo} = y_{viejo} + \alpha x$

<sup>16</sup> En los chips gráficos de arquitectura unificada ya es posible solventar esos problemas, pero su programación es diferente a lo que tratamos de ilustrar en esta práctica.

La extensión framebuffer object nos proporciona una interfaz muy simple a *render to texture*<sup>17</sup>. De modo que utilizaremos una textura como render target, y esta debe ser asociada al FBO (en el caso de que todavía esté al *framebuffer*), con el siguiente código:

```
glFramebufferTexture2D(GL_FRAMEBUFFER_EXT,
                      GL_COLOR_ATTACHMENT0_EXT,
                      texture_target, texID, 0);
```

El primer parámetro es obligatorio. El segundo parámetro define el destino de la textura, es necesario indicarlo explícitamente porque cada FBO puede llegar a contener hasta 4 texturas<sup>18</sup>. El último de los parámetros selecciona el nivel del mipmap en el que queremos renderizar, en nuestro caso será 0 al no hacer uso de esta posibilidad.

Podemos pensar en los FBOs como estructuras de punteros que redirigen las operaciones de renderizado a la textura asociada, y, conceptualmente, lo que hacemos a través de OpenGL es modificar esa indirección para cambiar de target.

Puesto que está pensado para hacer operaciones con gráficos, la especificación de `framebuffer_object` sólo define asociaciones a texturas con el formato `GL_RGB` y `GL_RGBA`. Para poder aplicar texturas de tipo LUMINANCE necesitamos una extensión posterior no completamente soportada,.

En la siguiente tabla tenemos un resumen de los texture targets y sus combinaciones de parámetros que son realmente soportados (existen más, pero aquí nos centramos en los formatos de punto flotante):

	NV3x	NV4x, G7x, G8x	ATI
texture 2D, ATI/ARB_texture_float, LUMINANCE	no	no	no
texture 2D, ATI/ARB_texture_float, RGB, RGBA	no	sí	sí
texture 2D, NV_float_buffer, LUMINANCE	no	no	no
texture 2D, NV_float_buffer, RGB, RGBA	no	no	no
texture RECT, ATI/ARB_texture_float, LUMINANCE	no	no	no
texture RECT, ATI/ARB_texture_float, RGB, RGBA	no	sí	sí
texture RECT, NV_float_buffer, LUMINANCE	sí	sí	no
texture RECT, NV_float_buffer, RGB, RGBA	sí	sí	no

<sup>17</sup> Ya utilizada en la primera y segundas prácticas de Técnicas Avanzadas de Gráficos 3D.

<sup>18</sup> Depende un poco del hardware, se puede consultar el número de texturas que soporta el FBO con la llamada a `GL_MAX_COLOR_ATTACHMENTS_EXT`

La conclusión a la que llegamos tras leer esta tabla es que, a día de hoy, no es posible escribir código portable para todas las posibles GPUs. Si queremos utilizar texturas LUMINANCE, necesariamente tenemos que optar por las texturas rectangulares en tarjetas de NVIDIA. El hardware más moderno supera estas restricciones, pero hemos de tenerlo en cuenta en el caso de querer que nuestras aplicaciones funcionen en un amplio rango de configuraciones<sup>19</sup>.

### 3.2.5 Transferencia de datos desde los arrays de la CPU a las texturas de la CPU

Para transferir los datos (como los vectores del ejemplo) a una textura, tenemos que asociar la textura a un texture target y establecer la copia de los datos a la tarjeta mediante una llamada de OpenGL.

Es muy importante que el array que pasamos mediante esta función esté adecuadamente dimensionado y se indique correctamente el tipo de datos a pasar (GL\_FLOAT), ya que el driver se encargará de todas estas tareas de forma transparente.

En tarjetas nVIDIA el siguiente código está acelerado por hardware:

```
glBindTexture(texture_target, texID);
glTexSubImage2D(texture_target, 0, 0, 0, texSize, texSize,
                texture_format, GL_FLOAT, data);
```

Los tres ceros son los parámetros que definen el desplazamiento y el nivel mipmap (que en este caso no vamos a utilizar).

En el caso de tener una tarjeta de AMD-ATI, se suele preferir transferir los datos a una textura ya asociada al framebuffer object, como en el siguiente código:

```
glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);
glRasterPos2i(0, 0);
glDrawPixels(texSize, texSize, texture_format, GL_FLOAT, data);
```

La primera llamada redirecciona la salida. En la segunda utilizamos el origen como posición de referencia para volcar todo el array en la textura en la última llamada a OpenGL.

En ambos casos, el array de la CPU es enviado por filas a la textura<sup>20</sup>. En una textura LUMINANCE los dos primeros elementos del array deberían contener el primer y el segundo texel de la primera fila, mientras que en el caso de una textura RGBA los primeros cuatro elementos en el array se corresponderán con las cuatro componentes del primer texel.

### 3.2.6 Transferencia de los datos de las texturas de la GPU a los arrays de la CPU

Tenemos dos alternativas para implementar el paso de la información de las texturas de la GPU a arrays en la CPU. La más tradicional se hace asociando la textura a un texture target y mediante la llamada a glGetTexImage de la siguiente manera:

```
glBindTexture(texture_target, texID);
glGetTexImage(texture_target, 0, texture_format, GL_FLOAT, data);
```

<sup>19</sup> Los cambios necesarios para lograr que el código sea compatible no son demasiado intrincados, pero no los trataremos en esta práctica. La fuente de información a partir de la que se ha elaborado esta sección pronto quedará obsoleta con la aparición de la especificación de OpenGL 3.0 en la que es más probable que este tipo de aspectos cambien a mejor.

<sup>20</sup> Este es un aspecto a tener en cuenta en lenguajes como FORTRAN, en los que los arrays están organizados por columnas. Pero en el caso de C y C++ es lo más natural.

Si la textura devuelta a la CPU y ya está asociada a un FBO, podemos utilizar la técnica de la redirección de punteros:

```
glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
glReadPixels(0,0,texSize,texSize,texture_format, GL_FLOAT,data);
```

Una vez obtenemos de vuelta la textura de la GPU a la memoria principal de la CPU pasamos el origen como los dos primeros parámetros. Esta es la técnica más recomendada.

Es muy importante destacar que las transferencias entre GPU y CPU llevan mucho tiempo, por ello se consideran muy “caras” ya que en su lugar se podrían realizar gran cantidad de cálculos en lugar de la transferencia de los datos. Por ello han de ser utilizadas sólo cuando sea estrictamente necesario y con cierta planificación.

### 3.2.7 Un pequeño programa de ejemplo

Antes de continuar avanzando, juntemos todas las piezas que se han presentado para mostrar una mini-aplicación que he usado de los mecanismos explicados.

Se recomienda al alumno que lea y trate de comprender el código el código, y que vuelva a los sub-apartados anteriores en caso de tener dudas sobre algún punto concreto.

```
#include <stdio.h>
#include <stdlib.h>
#include <GL/glew.h>
#include <GL/glut.h>

int main(int argc, char **argv) {
    // declare texture size, the actual data will be a vector
    // of size texSize*texSize*4
    int texSize = 2;
    // create test data
    float* data = (float*)malloc(4*texSize*texSize*sizeof(float));
    float* result = (float*)malloc(4*texSize*texSize*sizeof(float));
    for (int i=0; i<texSize*texSize*4; i++)
        data[i] = i+1.0;
    // set up glut to get valid GL context and
    // get extension entry points
    glutInit (&argc, argv);
    glutCreateWindow("TEST1");
    glewInit();
    // viewport transform for 1:1 pixel=texel=data mapping
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0,texSize,0.0,texSize);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glViewport(0,0,texSize,texSize);
    // create FBO and bind it (that is, use offscreen render target)
    GLuint fb;
    glGenFramebuffersEXT(1,&fb);
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,fb);
    // create texture
    GLuint tex;
    glGenTextures (1, &tex);
    glBindTexture(GL_TEXTURE_RECTANGLE_ARB,tex);
    // set texture parameters
    glTexParameterf(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);
// define texture with floating point format
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB,0,GL_RGBA32F_ARB,
             texSize,texSize,0,GL_RGBA,GL_FLOAT,0);
// attach texture
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          GL_COLOR_ATTACHMENT0_EXT,
                          GL_TEXTURE_RECTANGLE_ARB,tex,0);
// transfer data to texture
glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB,0,0,0,texSize,texSize,
               GL_RGBA,GL_FLOAT,data);
// and read back
glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
glReadPixels(0, 0, texSize, texSize,GL_RGBA,GL_FLOAT,result);
// print out results
printf("Data before roundtrip:\n");
for (int i=0; i<texSize*texSize*4; i++)
    printf("%f\n",data[i]);
printf("Data after roundtrip:\n");
for (int i=0; i<texSize*texSize*4; i++)
    printf("%f\n",result[i]);
// clean up
free(data);
free(result);
glDeleteFramebuffersEXT (1,&fb);
glDeleteTextures (1,&tex);
return 0;
}

```

### 3.3 Segundo símil: Kernels y shaders

En este apartado trataremos las diferencias fundamentales entre el modelo de computación de las GPUs y las CPUs en programación genérica debido a las diferencias arquitecturales entre la aproximación *streaming* y la de Von Neumann, así como el cambio de forma de afrontar los problemas y algoritmos en cauce clásico.

#### 3.3.1 Implementación basada en bucles de la CPU frente a la implementación orientada a kernels y paralelismo de datos de la GPU

En la programación de algoritmos gráficos que han de recorrer un dominio espacial, y en nuestro caso más humilde del ejemplo de la operación lineal  $y = y + \alpha x$ , en la CPU utilizamos una estructura de control tipo bucle para procesar todos los elementos:

```

for (int i=0; i<N; i++)
    dataY[i] = dataY[i] + alpha * dataX[i];

```

En este caso tenemos dos niveles de computación activos al mismo tiempo:

- En el exterior del bucle, el contador es actualizado (incrementado) y comparado con la longitud de nuestros vectores en cada pasada.
- Y dentro del bucle, accedemos a los arrays en posiciones determinadas por ese contador para realizar las operaciones que nos interesan, en este caso una multiplicación y una suma en cada elemento

Es importante resaltar que los cálculos que realizamos en cada elemento son *independientes* de los demás, para una posición dada accedemos a posiciones diferentes de memoria y no hay dependencias entre los datos del vector resultante.

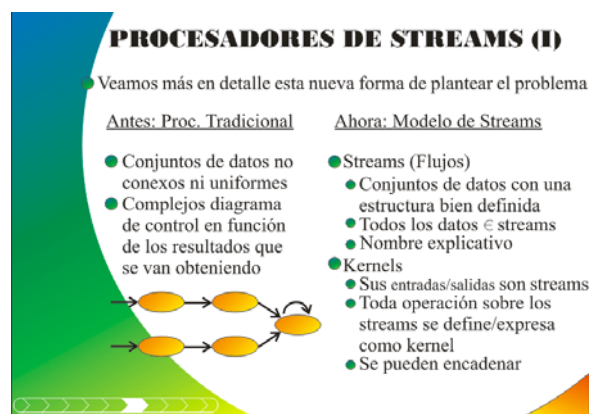
Si en lugar de tener una CPU convencional<sup>21</sup>, dispusiéramos de un procesador vectorial capaz de realizar N operaciones de golpe, ó de una CPU multicore con N núcleos, no necesitaríamos el bucle para nada, cada uno de los elementos sería calculado simultáneamente. A este paradigma se le suele llamar SIMD (una instrucción múltiples datos).

Aunque la arquitectura de las GPUs nos permite múltiples posibilidades, una aproximación desde el punto de vista de programación es que todos los elementos (vectores o fragmentos) son procesados como si tuviéramos un procesador distinto para cada uno de ellos. En realidad no es exactamente así, ya que tenemos un número limitado de recursos, pero dado que los procesadores no tienen estados y que en cauce clásico se aplica el mismo *shader* a todos los elementos de una misma etapa, este símil es posible.

Así desde el punto de vista de la GPGPU separamos el cálculo que se realiza en el exterior del bucle, y el procesado que tenemos en el interior para cada elemento.

En arquitecturas *streaming* se conoce como kernel a los elementos de proceso dentro del cauce segmentado, en el caso del cauce clásico tenemos dos elementos que son programables: los procesadores de vértices y los de fragmentos.

Cada uno de ellos es alimentado con estructuras de datos de elementos que son procesados de forma idéntica, igual que si se tratase del interior de un bucle en el que se van seleccionando los elementos a calcular. Dado que entre elementos de salida no hay dependencias de datos podemos extrapolar los cálculos que se hacen en el bucle más profundo al interior de un kernel.



En el ejemplo inicial, los índices de los arrays de entrada y salida son idénticos, ó más rigurosamente, las posiciones de los arrays de datos de entrada son las mismas desde el punto de vista de un elemento de salida. En el caso de tener que calcular una fórmula en diferencias finitas tendremos una cierta dependencia de los datos de entrada (como ocurriría en el caso de hacer simulación de fluidos y de forma similar al caso del autómatas celular del juego de la vida que trataremos más adelante en esta práctica) y se tendrán que resolver las diferencias locales en las posiciones de los elementos de entrada. Por ejemplo, con la fórmula  $y[i] = -x[i - 1] + 2x[i] - x[i + 1]$  el kernel tendrá que consultar los valores de x en la misma posición que en el elemento de salida, pero también a su “izquierda” y su “derecha”.

En GPGPU se suele preferir el uso de *shaders* en la etapa de sombreado, ya que los procesadores de fragmentos tienen una salida directa a la textura que se renderiza (y que podemos reenviar fácilmente a la CPU). De cara a un programador es como tener un procesador vectorial del tamaño de las texturas que queramos procesar, aunque como ya se ha comentado en clase estos procesadores no funcionan según el paradigma SIMD y los

<sup>21</sup> Los compiladores suelen desenrollar los bucles y aprovechar las capacidades SIMD de las CPUs convencionales con instrucciones vectoriales tipo SSE, SSE2... pero se describe de esta forma por simplicidad.

fragmentos pueden encontrarse en distintos puntos de ejecución<sup>22</sup> (en realidad tenemos muchos fragmentos siendo procesados a la vez en etapas de datos muy segmentadas).

La lógica encargada del reparto de estructuras de datos no es programable o configurable en el chip gráfico, por lo que no podemos controlar el orden en que los fragmentos son procesados, aunque si podemos conocer su “dirección” (la posición a la que corresponden, como se vió en la práctica 1) y las coordenadas de textura de los datos con los que trabajan.

### 3.3.2 Creación de un *shader* en lenguaje Cg

En este sencillo caso de ejemplo nos centraremos en la etapa de sombreado y dejaremos el procesado de vértices<sup>23</sup> tal y como funciona por defecto en OpenGL.

El código necesario para realizar la operación del ejemplo ( $y = y + \alpha x$ ) debería ser pan comido para los alumnos. El interior del bucle que examinábamos en el código de la CPU tenía algunos cálculos, dos consultas a arrays y un valor constante. Dado que ya hemos visto que las texturas en la GPU son la estructura de datos equivalente a los arrays, la búsqueda se hará a través del muestreo de las texturas con las coordenadas de texturas interpoladas por el rasterizador (el cómo estas coordenadas se corresponden de forma directa con los datos a los que queremos acceder lo veremos más adelante). El valor de la constante en coma flotante podría introducirse en duro en el propio código y modificarlo cada vez que sea necesario, pero resulta mucho más cómodo pasar su valor como un parámetro *uniform* (que en realidad se comporta como una constante al no poder modificarse durante el renderizado de una misma pasada del cauce). En el siguiente pedazo de código podemos ver la parte central del *shader* de fragmentos necesario:

```
float saxpy (
    float2 coords : TEXCOORD0,
    uniform sampler2D textureY,
    uniform sampler2D textureX,
    uniform float alpha ) : COLOR
{
    float result;
    float y = tex2D(textureY,coords); // float yval=y_old[i];
    float x = tex2D(textureX,coords); // float xval=x[i];
    result = y + alpha * x;          // y_new[i]=yval+alpha*xval;
    return result;
}
```

La interpretación de este código a estas alturas debería ser muy sencilla. Tan sólo destacar que la función de muestreo de las texturas depende del texture target que estemos utilizando:

	Textura 2D	Textura rectangular
Definición de muestreador (sampler)	Uniform sampler2D	Uniform samplerRECT
Buscador de texturas	Tex2D (nombre, coords)	texRECT(nombre, coords)

<sup>22</sup> Por ejemplo, en el caso de que se produzca un bloqueo en una de ellas al no estar preparado el valor de una textura con el sistema de prebúsqueda y no queden más fragmentos en el procesador cuádruple.

<sup>23</sup> El procesador de vértices en GPGPU se suele utilizar para poder establecer condiciones en los datos a tratar, y proyecciones especiales. En esta práctica de iniciación no adelantaremos esas técnicas, pero en cualquier caso, la etapa de transformación en GPGPU clásica no se aprovecha tan bien como sería deseable, y tenemos un cierto desequilibrio en la carga del cauce clásico.

Si deseamos utilizar texturas de cuatro canales en lugar de texturas con formato de luminancia, sólo tenemos que modificar el tipo de variable que se encarga de la búsqueda del valor de textura (y del valor devuelto). Como podemos hacer “4 operaciones por el precio de una”, el código sería el siguiente:

```
float4 saxpy (
    float2 coords : TEXCOORD0,
    uniform samplerRECT textureY,
    uniform samplerRECT textureX,
    uniform float alpha ) : COLOR
{
    float4 result;
    float4 y = texRECT(textureY,coords);
    float4 x = texRECT(textureX,coords);
    result = y + alpha*x;
    // equivalent: result.rgb=y.rgb+alpha*x.rgb
    //             or: result.r=y.r+alpha*x.y; result.g=...
    return result;
}
```

El *shader* se puede guardar en una array tipo char (como veremos en la práctica 5) o en un fichero aparte para utilizarlo en tiempo de ejecución desde el programa de OpenGL con el Cg runtime justo antes de establecer la configuración del cauce gráfico en nuestra aplicación.

### 3.3.3 Configuración del Cg runtime

Esta sección ya se ha tratado en la práctica anterior y debería resultar familiar a los alumnos. Para poder hacer uso del Cg runtime es necesario incluir los ficheros de cabecera de Cg (con <Cg/cgGL.h> debería ser suficiente), los *paths* y las librerías en las opciones del compilador y el enlazador en el fichero de descripción del proyecto (en las propiedades del menú de Visual Studio), junto con la declaración de las siguientes variables:

```
// Cg vars
CGcontext cgContext;
CGprofile fragmentProfile;
CGprogram fragmentProgram;
CGparameter yParam, xParam, alphaParam;
char* program_source = "float saxpy( [...] return result; } ";
```

El CGcontext define el punto de entrada del Cg runtime, también son necesarios el perfil del procesador de fragmentos y un contenedor para el programa que se ha mostrado en la sección anterior. En la memoria de esta práctica no entraremos en detalles relativos a su configuración ya que se trataron con profundidad en la práctica anterior y pueden consultarse en el manual de Cg.

```
void initCG(void) {
    // set up Cg
    cgContext = cgCreateContext();
    fragmentProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);
    cgGLSetOptimalOptions(fragmentProfile);
    // create fragment program
    fragmentProgram = cgCreateProgram (
        cgContext,CG_SOURCE,program_source,
        fragmentProfile,"saxpy",NULL);

    // load program
    cgGLLoadProgram (fragmentProgram);
    // and get parameter handles by name
    yParam = cgGetNamedParameter (fragmentProgram,"textureY");
}
```

```

    xParam = cgGetNamedParameter (fragmentProgram, "textureX");
    alphaParam = cgGetNamedParameter (fragmentProgram, "alpha");
}

```

### 3.4 Tercer símil: Computación y dibujado

En esta sección veremos la relación que tiene el hecho de calcular el contenido del bucle interno que teníamos en la CPU y el “dibujado” mediante el paso a través del cauce clásico. Para ello son necesarios tres pasos: la activación del kernel, la asignación de las texturas de entrada y salida en el Cg runtime y, finalmente, los cálculos son llevados a cabo mediante la renderización de una geometría adecuada.

#### 3.4.1 Preparación del kernel computacional

Para activar el kernel mediante el Cg runtime, el perfil de fragmentos creado anteriormente debe ser activado y el *shader* correspondiente ha de cargarse y asociarse. Sólo un *shader* de cada tipo puede estar activo simultáneamente. Como en este caso no vamos a “oscurecer” las tareas de la etapa de transformación, el siguiente código debería ser suficiente:

```

// enable fragment profile
cgGLEnableProfile(fragmentProfile);
// bind saxpy program
cgGLBindProgram(fragmentProgram);

```

#### 3.4.2 Establecimiento de los arrays / texturas de entrada

El enlazado y habilitación de las texturas de entrada  $y_{viejo}$ ,  $x$  y la constante (uniform), los identificadores de textura  $y\_oldTexID$  y  $xTexID$  es muy parecido a lo que ya se ha visto en prácticas anteriores:

```

// enable texture y_old (read-only)
cgGLSetTextureParameter(yParam, y_oldTexID);
cgGLEnableTextureParameter(yParam);
// enable texture x (read-only)
cgGLSetTextureParameter(xParam, xTexID);
cgGLEnableTextureParameter(xParam);
// enable scalar alpha
cgSetParameter1f(alphaParam, alpha);

```

#### 3.4.3 Establecimiento de los arrays / texturas de salida

Definir el array de salida es básicamente la misma operación que la que se mostró para transferir datos a una textura ya asociada al FBO. Utilizaremos la misma manipulación de punteros que vimos a través de llamadas de OpenGL (simplemente redireccionamos la salida). Si no lo hemos hecho todavía, asociamos el texture target a nuestro FBO y mediante llamadas de OpenGL lo ponemos como render target.

```

// attach target texture to first attachment point
glFramebufferTexture2D(GL_FRAMEBUFFER_EXT,
                      GL_COLOR_ATTACHMENT0_EXT,
                      texture_target, y_newTexID, 0);

// set the texture as render target
glDrawBuffer (GL_COLOR_ATTACHMENT0_EXT);

```

#### 3.4.4 Realización del cálculo

Hasta ahora hemos realizado una proyección uno a uno entre los píxeles finales, las coordenadas de textura y la geometría que estamos a punto de dibujar. Y hemos preparado el *shader* de fragmentos para que se ejecute en cada estructura de datos que salga del

rasterizador. Todo lo que queda por hacer es renderizar una “geometría adecuada” que asegure que el *shader* de fragmentos es ejecutado para cada elemento almacenado en la textura, esto es, asegurarnos que cada fragmento se corresponde con un único texel. Dados la proyección y el punto de vista de la cámara que hemos escogido esto es tremendamente fácil. Sólo necesitamos un *quad* que cubra todo el *viewport*.

Mediante la llamada de OpenGL sólo tenemos que definir las cuatro esquinas del quad y asignar las coordenadas de textura como atributos de los vértices. Los cuatro vértices serán transformados a espacio de pantalla de forma automática por la etapa de transformación que no modificamos en esta práctica.

El rasterizador se encargará de realizar una interpolación bilineal de cada fragmento cubierto por el quad, interpolando tanto la posición (en espacio de pantalla) como los atributos de los vértices (las coordenadas de textura) y estos datos pasarán al procesador de fragmentos como estructuras de datos. Esto es, el renderizado de un simple quad nos sirve como generador de flujo de datos en GPGPU, y se consigue el efecto deseado para el ejemplo que tenemos entre manos.

Si utilizamos texturas rectangulares, y, por tanto, las coordenadas de textura son idénticas a las coordenadas de los fragmentos, podemos utilizar el siguiente fragmento de código:

```
// make quad filled to hit every pixel/texel
glPolygonMode(GL_FRONT, GL_FILL);
// and render quad
glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0);
    glVertex2f(0.0, 0.0);
    glTexCoord2f(texSize, 0.0);
    glVertex2f(texSize, 0.0);
    glTexCoord2f(texSize, texSize);
    glVertex2f(texSize, texSize);
    glTexCoord2f(0.0, texSize);
    glVertex2f(0.0, texSize);
glEnd();
```

En el caso de utilizar texturas 2D, es necesario hacer un escalado para normalizar las coordenadas de textura, y el código sería el siguiente:

```
// make quad filled to hit every pixel/texel
glPolygonMode(GL_FRONT, GL_FILL);
// and render quad
glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0);
    glVertex2f(0.0, 0.0);
    glTexCoord2f(1.0, 0.0);
    glVertex2f(texSize, 0.0);
    glTexCoord2f(1.0, 1.0);
    glVertex2f(texSize, texSize);
    glTexCoord2f(0.0, 1.0);
    glVertex2f(0.0, texSize);
glEnd();
```

En el caso de desear realizar operaciones más complejas, sería posible emplear más de un conjunto de coordenadas de texturas por vértice, como ya hemos visto en clase<sup>24</sup>.

<sup>24</sup> Conviene revisar la documentación de `glMultiTexCoord()`

### 3.5 Cuarto símil: Retroalimentación

Una vez los cálculos se han realizado, el resultado queda almacenado en la textura `target_y_new`

#### 3.5.1 Renderizado en múltiples pasadas

En una aplicación real, ya sea gráfica (como la textura dinámica del juego de la vida) o un método numérico iterativo, el resultado es utilizado como entrada en el siguiente cálculo. En la GPU esto implica atravesar de nuevo todo el cauce de renderizado y asociar diferentes texturas de entrada y salida (y eventualmente un kernel distinto). Para ello se suele recurrir a la técnica “ping pong”.

#### 3.5.2 La técnica “ping pong”

La técnica “ping pong” nos permite utilizar alternativamente la salida de una pasada de renderizado como entrada en la siguiente. En nuestro caso de ejemplo,  $y_{nuevo} = y_{viejo} + \alpha x$ , intercambiamos el papel que juegan las texturas  $y_{nuevo}$  e  $y_{viejo}$ , ya que no necesitamos los valores de  $y_{viejo}$  más adelante, en este espacio de memoria de vídeo podemos guardar nuevos valores (en una pasada de renderizado diferente debido a las limitaciones de la interfaz de memoria de las tarjetas de cauce clásico). Tenemos tres formas distintas de implementar este tipo de reutilización de estructuras de datos<sup>25</sup>:

- Utilizar FBOs distintos con asociaciones a cada una de las texturas hacia la que se renderiza y cambiarlos con `glBindFramebufferEXT()`
- Utilizar un FBO, reasociar la textura del render target en cada pasada utilizando `glFramebufferTexture2DTEXT()`
- Utilizar un FBO con múltiples puntos de asociación, e intercambiarlos con `glDrawBuffer()`

Dado que es posible asociar hasta 4 texturas por FBO, la última alternativa parece la más rápida. Para llevarlo a cabo necesitamos algunas variables de control:

```
// two textures identifiers referencing y_old and y_new
GLuint yTexID[2];
// ping pong management vars
int writeTex = 0;
int readTex = 1;
GLenum attachmentpoints[] = { GL_COLOR_ATTACHMENT0_EXT,
                               GL_COLOR_ATTACHMENT1_EXT
                             };
```

Durante el cálculo todo lo que tenemos que hacer es pasar el valor correcto de estas dos tuplas a las llamadas de OpenGL correspondientes, e intercambiar las variables de índice después de cada pasada:

```
// attach two textures to FBO
glFramebufferTexture2DTEXT(GL_FRAMEBUFFER_EXT,
                           attachmentpoints[writeTex],
                           texture_Target, yTexID[writeTex], 0);
glFramebufferTexture2DTEXT(GL_FRAMEBUFFER_EXT,
                           attachmentpoints[readTex],
                           texture_Target, yTexID[readTex], 0);
// enable fragment profile, bind program [...]
```

<sup>25</sup> Esto está explicado en mayor detalle en la presentación de Simon Green: “OpenGL FrameBuffer extension” que podeis encontrar en [http://http.download.nvidia.com/developer/presentations/2005/GDC/OpenGL\\_Day/OpenGL\\_FrameBuffer\\_Object.pdf](http://http.download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_FrameBuffer_Object.pdf)

```

// enable texture x (read-only) and uniform parameter [...]
// iterate computation several times
for (int i=0; i<numIterations; i++) {
    // set render destination
    glBindBuffer (attachmentpoints[writeTex]);
    // enable texture y_old (read-only)
    cgGLSetTextureParameter(yParam, yTexID[readTex]);
    cgGLEnableTextureParameter(yParam);
    // and render multitextured viewport-sized quad
    // swap role of the two textures (read-only source becomes
    // write-only target and the other way round):
    swap();
}

```

### 3.5.3 Algunos detalles sobre el código fuente de ejemplo que acompaña esta sección

El código del ejemplo que acompaña esta sección de la práctica (pensado para que los alumnos no tengan que empezar la práctica desde cero) realiza las siguientes operaciones:

- Crea una textura en coma flotante por vector
- Transfiere los datos iniciales a estas texturas
- Crea un *shader* en Cg (también están las instrucciones para que sea compatible con GLSL)
- Itera varias veces los cálculos para demostrar el uso de la técnica “ping pong”
- Transfiere de vuelta los datos a memoria principal
- Compara los resultados con la solución de referencia calculada en la CPU

Para que el código no sea tan farragoso muchos de los parámetros de las llamadas de OpenGL han sido encapsulados en estructuras de datos, como en el siguiente ejemplo:

```

rect_nv_r_32.name           = "TEXRECT - float_NV - R - 32";
rect_nv_r_32.texTarget     = GL_TEXTURE_RECTANGLE_ARB;
rect_nv_r_32.texInternalFormat = GL_FLOAT_R32_NV;
rect_nv_r_32.texFormat     = GL_LUMINANCE;
rect_nv_r_32.shader_source = "float saxpy (\\"
    "in float2 coords : TEXCOORD0,\\"
    "uniform samplerRECT textureY,\\"
    "uniform samplerRECT textureX,\\"
    "uniform float alpha ) : COLOR {\\"
    "float y = texRECT (textureY, coords);\\"
    "float x = texRECT (textureX, coords);\\"
    "return y+alpha*x; }";

```

Además, el código de ejemplo toma algunos parámetros de la línea de comandos para su configuración. Al ejecutarlo sin ellos da una explicación de los distintos parámetros (el parseado es bastante mejorable, por lo que puede fallar si no se sigue al pie de la letra). Posee un modo de test y otro de benchmark.

## 4. El juego de la vida

El juego de la vida es un autómata celular ideado por el matemático británico John Horton Conway in 1970, y, sin duda, es el ejemplo más conocido de este tipo de modelos matemáticos de sistemas dinámicos.

En realidad se trata de un “juego” sin jugadores, ya que la evolución viene determinada sólo por su estado inicial y no necesita ninguna entrada de datos por parte de un oponente humano (aunque existen variantes de dos jugadores).

## 4.1 Origen

Conway estudió un problema que John Von Neumann había presentado en los años 40. Von Neumann ideó una máquina imaginaria que fuera capaz de hacer copias de sí misma y trató de modelarlo matemáticamente en una malla rectangular con éxito, el resultado fue un conjunto de reglas bastante complejo que publicó<sup>26</sup> en el libro *Theory of Self-reproducing Automata* (editado y completado por A. W. Burks). Conway trató de simplificar las ideas y reglas de Von Neumann, lo que unido a la influencia de sus anteriores trabajos en el problema de las regillas de Leech<sup>27</sup> le hicieron llegar al “juego de la vida”.



El caparazón de *Conus textile* muestra un patrón caracterizable en términos de autómatas celulares (imagen de la Wikipedia)

La primera publicación del juego de la vida fue en la famosa columna de Martin Gardner “Mathematical Games” del número de Octubre de 1970 de *Scientific American*. Desde un punto de vista teórico, es interesante porque se trata de una máquina de Turing

universal, esto es, cualquier cosa que pueda ser calculada algorítmicamente puede ser calculada dentro de un Juego de la Vida. Además, una estructura puede contener un conjunto de patrones especiales que se combinen para construir nuevos objetos, incluso copias de la estructura original. Se puede construir un “constructor universal” que contenga un ordenador Turing-completo y que pueda generar muchos tipos de objetos complejos, incluso nuevas copias de sí mismo (se pueden consultar descripciones de estas construcciones en *Winning Ways for your Mathematical Plays* de Conway, Elwyn Berlekamp y Richard Guy)

Martin Gardner escribió:

“The game made Conway instantly famous, but it also opened up a whole new field of mathematical research, the field of cellular automata ... Because of Life’s analogies with the rise, fall and alterations of a society of living organisms, it belongs to a growing class of what are called ‘simulation games’ (games that resemble real life processes) ”

Desde el mismo momento de su publicación el Juego de la Vida atrajo mucho interés debido a la siempre sorprendente evolución de los patrones que se forman. El juego de la vida es un ejemplo de interacción y autoorganización en sistemas complejos. Así, físicos, biólogos, economistas, matemáticos, filósofos, y muchos otros científicos observaron cómo podían aparecer patrones complejos a partir de reglas muy simples, y esto dio lugar a un buen número de teorías que han calado profundamente en la forma de entender nuestro entorno a nivel social, natural, evolutivo...

Otra de las razones por las que el Juego de la Vida se hizo tan popular fue debida a la aparición en el mercado de las minicomputadoras, cuya primitiva capacidad era suficiente para ejecutar este tipo de algoritmos y podían dejarse durante horas visualizando este tipo de patrones por pantalla. Del mismo modo en el que más tarde se hicieron famosos los fractales. Era una forma muy divertida de gastar ciclos de procesador ;-)

<sup>26</sup> Aunque John von Neumann puso en práctica los autómatas celulares dentro del contexto de la física computacional, éstos fueron concebidos en los años 40 por Konrad Zuse y Stanislaw Ulam. Zuse los planteó en los “espacios de cómputo” (*computing spaces*), como modelos discretos de sistemas físicos. Las contribuciones de Ulam vinieron al final de los 40, poco después de haber inventado con Nicholas Metropolis el Método de Monte Carlo (del que hablaremos en Técnicas Avanzadas de Gráficos 3D).

<sup>27</sup> [http://en.wikipedia.org/wiki/Leech\\_lattice](http://en.wikipedia.org/wiki/Leech_lattice)

Conway eligió las reglas de forma muy cuidadosa, tras un largo periodo de experimentación, hasta dar con tres criterios:

- No habría un patrón inicial para el que se pueda demostrar fácilmente que se llegue a una población que pueda crecer sin límite.
- Habría patrones iniciales que aparentemente podrían crecer sin límite
- Habría patrones iniciales que crecerían y cambiarían durante un considerable periodo de tiempo antes de evolucionar de alguna de las siguientes maneras:

Desapareciendo completamente, ya sea por superpoblación o por

## 4.2 Reglas

El universo del juego de la vida es una malla ortogonal bidimensional de celdas cuadradas, cada una de las cuales puede tener dos posibles estados: vivo o muerto. Cada celda interactúa con las 8 celdas colindantes, que son las celdas vecinas horizontal, vertical y diagonalmente.

En cada paso de tiempo pueden darse las siguientes transiciones:





- Una celda /célula viva con menos de 2 vecinos vivos muere (de soledad)
- Una celda /célula viva con más de 3 vecinos vivos muere (por superpoblación)
- Una celda /célula viva con 2 ó 3 vecinos vivos sobrevive, no se produce ningún cambio hasta la próxima generación.
- Una celda /célula muerta con justo 3 vecinos vivos vuelve a la vida.

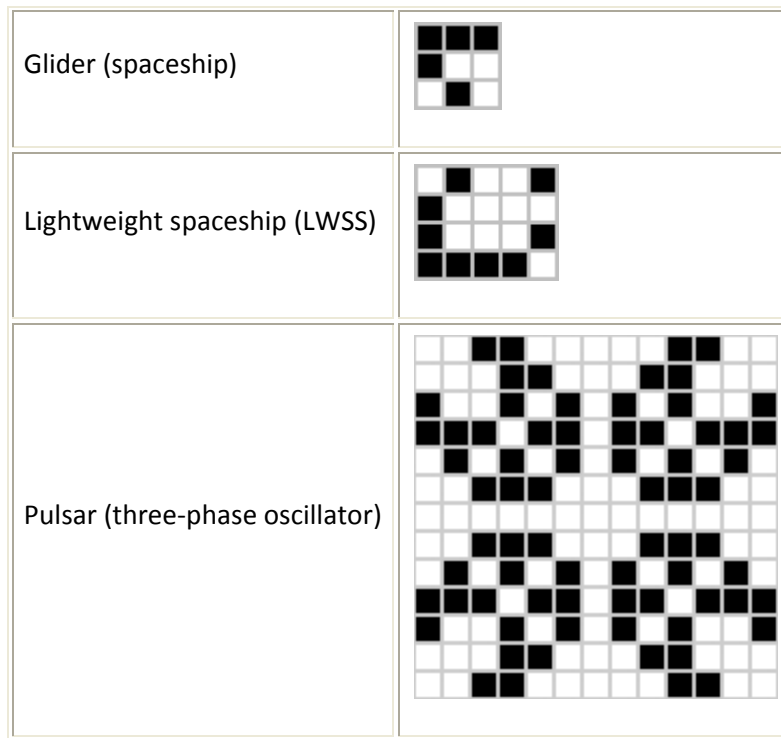
El patrón inicial constituye la “semilla” del sistema. La primera generación se crea al aplicar las reglas anteriores a cada una de las celdas de la semilla, y estas reglas se vuelven a aplicar en cada nuevo paso de tiempo.

## 4.3 Ejemplos de patrones

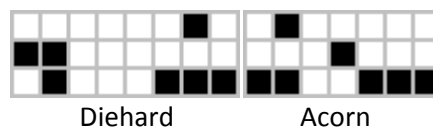
En el juego de la vida tenemos una gran cantidad de patrones característicos: patrones estáticos (“seres inmóviles”), patrones repetitivos (“osciladores”, que son un superconjunto de los anteriores) y patrones que se transforman a sí mismos mientras recorren el tablero / universo de juego (“naves espaciales”).

En la siguiente tabla, extraída de la Wikipedia, podemos ver algunos ejemplos característicos, en el que las celdas vivas aparecen en negro y las muertas en blanco:

Block (still life)	
Boat (still life)	
Blinker (two-phase oscillator)	
Toad (two-phase oscillator)	

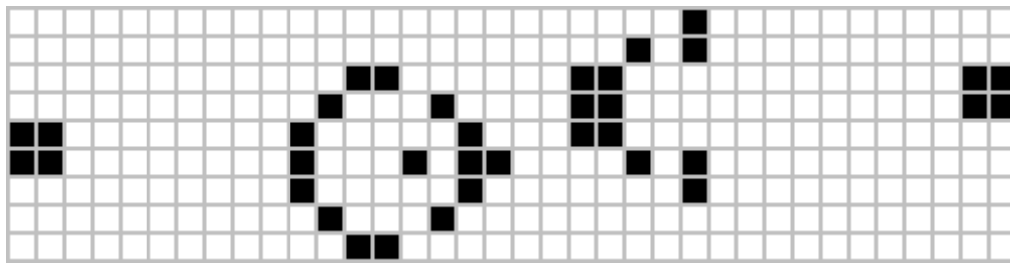


El "pulsar" es la forma más común de oscilador de periodo 3, la gran mayoría de los osciladores que aparecen tienen periodo 2 (como el "blinker" y "toad"), pero pueden verse patrones con periodos mucho más grandes (4, 8, 15, 30...) de vez en cuando. Por ejemplo "diehard" es un patrón que dura 130 generaciones, y "acorn" puede llegar a durar 5206 generaciones y crear al menos 25 "gliders"



En la aparición original del juego en la revista, Conway ofreció un premio de 50 dólares por el descubrimiento de patrones que crecieran indefinidamente (que en principio desbancara la regla de diseño más importante del apartado 4.1). El primero fue descubierto por Bill Gosper en noviembre de 1970. Entre los patrones que crecen indefinidamente se encuentran las "pistolas" (guns), que son estructuras fijas en el espacio que generan planeadores u otras naves espaciales; "locomotoras" (puffers), que se mueven y dejan un rastro de basura y "rastrillos" (rakes), que se mueven y emiten naves espaciales. Gosper descubrió posteriormente un patrón que crece cuadráticamente llamado "criadero" (breeder), que deja atrás un rastro de pistolas. Desde entonces se han creado construcciones más complicadas, como puertas lógicas de planeadores, un sumador, un generador de números primos y una célula unidad que emula el juego de la vida a una escala mucho mayor y una velocidad menor.

El primer planeador que se ha descubierto sigue siendo el más pequeño que se conoce:

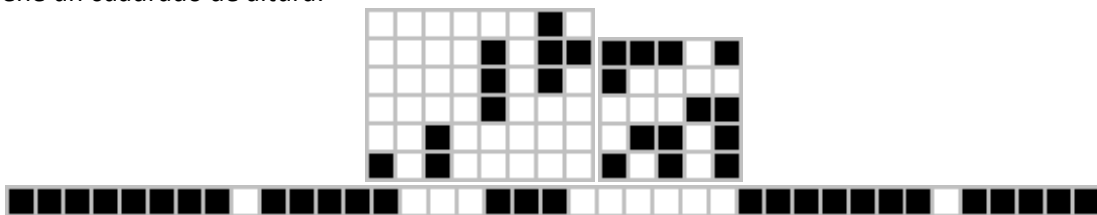


Pistola de Gosper



Pistola de planeadores de Gosper (Gosper Glider Gun)

Se han hallado posteriormente patrones más simples que también crecen indefinidamente. Los tres patrones siguientes crecen indefinidamente. Los dos primeros generan un motor interruptor que deja bloques, mientras que el tercero genera dos. El primero tiene una población mínima de 10 células vivas, el segundo cabe en un cuadrado  $5 \times 5$  y el tercero sólo tiene un cuadrado de altura:



Es posible que los planeadores interactúen con otros objetos de forma interesante. Por ejemplo, si se disparan dos planeadores hacia un bloque contra el que chocan de la forma correcta, el bloque se acercará al origen de los planeadores, pero si se disparan tres planeadores de forma correcta el bloque se alejará. Esta "memoria del bloque deslizante" se puede emplear para simular un contador. Es posible construir puertas lógicas AND (y, conjunción), OR (o, disyunción) y NOT (no, negación) mediante el uso de planeadores.

Como ya se ha indicado anteriormente, se puede construir una estructura que actúe como una máquina de estados finitos conectada a dos contadores. Esto tiene la misma potencia computacional que una máquina universal de Turing, así que el juego de la vida es tan potente como un ordenador con memoria ilimitada: por ello se considera Turing-completo.

En esta práctica crearemos un juego de la vida que nos permita explorar estos patrones repetitivos en grandes áreas de visualización y periodos de tiempo aprovechando la gran capacidad computacional de las tarjetas gráficas<sup>28</sup>.

<sup>28</sup> Existen algoritmos e implementaciones optimizadas muy sofisticadas del juego de la vida para explorar la evolución de los patrones en periodos inmensos de tiempo, como el HashLife que ideó Bill Gosper a principios de los años 80. Pero en esta práctica no nos lo tomaremos tan seriamente y no será

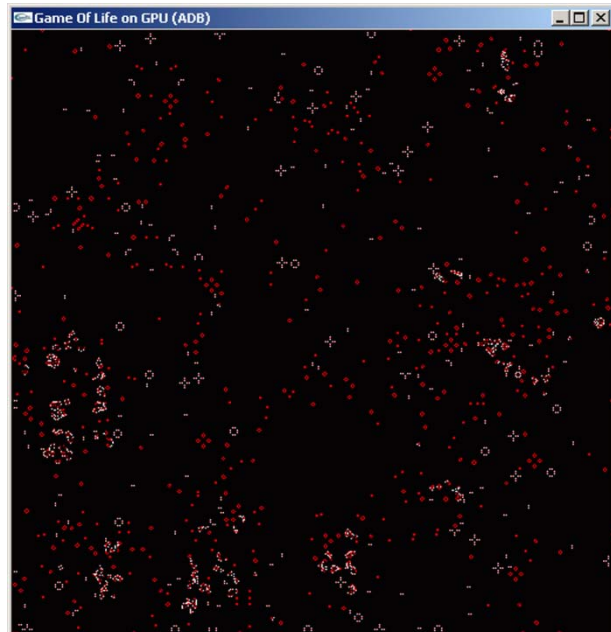
## 5. Implementación del juego de la vida en GPU y cuestiones

La principal tarea a realizar en esta práctica es la implementación de un Juego de la Vida mediante *shaders* con la API de OpenGL.

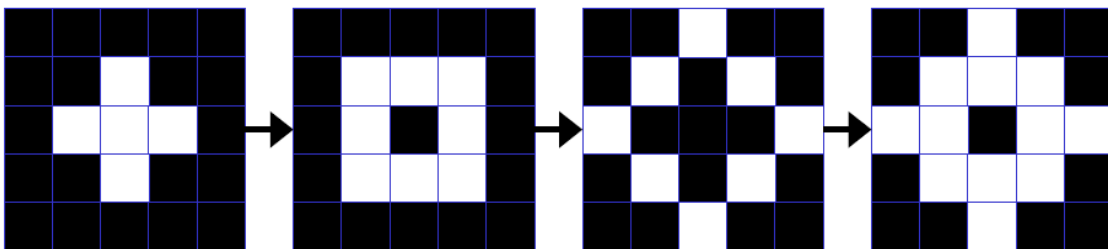
La malla que conforma el universo de juego debe ser de al menos 512x512 píxeles y se inicializará con una semilla completamente aleatoria. El resultado deberá aparecer en pantalla y se volcará a disco en forma de fichero RAW cuando se presione la tecla 's' en el programa.

Las celdas / células tendrán color rojo si están vivas, verde si son recién nacidas, y negras si están muertas.

**TODO el procesamiento del autómata celular debe ser realizado en la GPU.**



Para su realización pueden consultarse la implementación parcial que se ha puesto en la página web de la asignatura con el código original de Mark Harris, pero se recomienda su realización desde cero (a partir del código de ejemplo de la sección 3).



Procurad utilizar de forma inteligente todos los canales de color de los que disponéis (rojo, azul, verde y *alpha*), y experimentar con el número de pasadas por iteración.

El código debe estar limpio y adecuadamente comentado.

### 5.1 Cuestiones y mejoras del código

1. ¿Qué es un pixel buffer? ¿cómo se utilizan?
2. ¿En qué se diferencian de los FrameBuffer Objects? ¿Cuál puede ser mejor para su utilización en GPGPU? ¿por qué? Consulta en internet y trata de razonar las respuestas.
3. El número de consultas que se realizan para averiguar el número de vecinos vivos de cada celda en la implementación óptima es bastante inferior a 8. Trata de reducirlo aplicando los conceptos aprendidos en Procesadores Gráficos y Técnicas Avanzadas de Gráficos 3D sin alterar el resultado del algoritmo.
4. Habilita un mecanismo en el código para contabilizar el número de iteraciones del autómata celular y haz una gráfica de rendimiento en el que en un eje varíe el tamaño del framebuffer (ej. 512x512) y en otro el número de iteraciones por segundo con las medidas que habéis tomado.

necesario “comprimir tiempo y espacio”. Si tenéis curiosidad podéis consultar el artículo del Dr. Jobbs: <http://www.ddj.com/hpc-high-performance-computing/184406478>

5. [OPCIONAL] A partir de la métrica anterior intenta realizar modificaciones en el código del *shader* para lograr mejoras en el rendimiento de la aplicación. Muestra el cambio obtenido en la optimización con un gráfico como el anterior.
6. [OPCIONAL] Crea tu propia variante del juego de la vida  
Desde la creación del juego se han desarrollado nuevas reglas. El juego estándar, en que nace una célula si tiene 3 células vecinas vivas, sigue viva si tiene 2 o 3 células vecinas vivas y muere en otro caso, se simboliza como "23/3". El primer número o lista de números es lo que requiere una célula para que siga viva, y el segundo es el requisito para su nacimiento.

Así, "16/6" significa que "una célula nace si tiene 6 vecinas y vive siempre que haya 1 o 6 vecinas". HighLife ("Alta Vida") es 23/36, porque es similar al juego original 23/3 sólo que también nace una célula si tiene 6 vecinas vivas. HighLife es conocida sobre todo por sus replicantes. Se conocen muchas variaciones del juego de la vida, aunque casi todas son demasiado caóticas o demasiado desoladas.

- /3 (estable) casi todo es una chispa (blink)
  - 5678/35678 (caótico) diamantes, catástrofes
  - 1357/1357 (crece) todo son replicantes
  - 1358/357 (caótico) un reino equilibrado de amebas
  - 23/3 (caótico) "Juego de la Vida de Conway"
  - 23/36 (caótico) "HighLife" (tiene replicante)
  - 235678/3678 (estable) mancha de tinta que se seca rápidamente
  - 245/368 (estable) muerte, locomotoras y naves
  - 34/34 (crece) "Vida 34"
  - 51/346 (estable) "Larga vida" casi todo son osciladores
7. [OPCIONAL] Efecto de fuego.  
Para ello se puede realizar el renderizado mediante el buffer de acumulación y modificando ligeramente la posición de los fragmentos (en realidad de la geometría a partir de la que se crean), de esta forma se puede lograr el efecto de "fuego". El efecto puede quedar muy interesante, como se mostró en el laboratorio.
  8. [OPCIONAL] Aplicar la textura resultante sobre una geometría distinta, como una tetera y experimentar con la posición de las coordenadas (u,v)

## Forma de entrega

Se entregará la memoria en el formato de cualquier procesador de textos (ó PDF) debidamente identificada junto con los ficheros con los programas en Cg en formato texto, tal y como los utilizaríais en el entorno de trabajo que se ha presentado en esta práctica.

Puede enviarse directamente mediante la aplicación correspondiente en el Campus Virtual ó por email dentro del plazo, e indicando "[PG Practica04]". No olvidéis indicar vuestros nombres y apellidos en el cuerpo del mensaje.

Planificaros bien.

## Nota aclaratoria

Todas las marcas y productos mencionados en este enunciado de prácticas están registradas por sus respectivas compañías, y su uso es de carácter descriptivo con fines docentes.

Tal como se aclara en el enunciado, la tercera sección de esta práctica, con contenido teórico, está fuertemente inspirada en el archiconocido tutorial de Dominik Göddeke sobre programación GPGPU, y en la cuarta, relativa a la explicación del Juego de la Vida, se han tomado muchos datos de la Wikipedia.