Sistemas Informáticos 2003 - 2004

Facultad de Informática
Universidad Complutense de Madrid

Proyecto CG

Simulación y Visualización de Fenómenos Físicos sobre GPU's de última generación

Autor:

JAVIER ORTEGO LA MONEDA

Dirigido por:

ROBERTO LARIO

INDICE

SECCIÓN **PAG.**

1. Especificación de partida:

PROYECTO:
Simulación y Visualización de Fenómenos Físicos sobre GPU's de última generación

Roberto Lario
Dpto. Arquitectura de
Computadores y Automática
Universidad Complutense Madrid

rlario@dacya.ucm.es

1. INTRODUCCIÓN

El presente proyecto, como bien indica su nombre, tiene por objetivo la **simulación** y **visualización** de ciertos fenómenos físicos que pueden ser de gran utilidad para mostrar escenas 3D con gran realismo. Para ello se pretede hacer uso intensivo de la capacidad de cálculo de los procesadores gráficos de última generación (NVIDIA GeForce FX [1], ATI Radeon 9700 [2]). Además, se realizará cuando sea posible un estudio comparativo a nivel de tiempos entre la versión hardware (sobre GPU) y la versión software (CPU). El principal propósito es analizar que tipo de cálculos pueden realizarse sobre GPU y cómo.

Se utilizará C/C++ como lenguaje de programación. El entorno gráfico se implentará con OpenGL [3], haciendo uso de la librería GLUT. La programación de la GPU se realizará mediante las extensiones *ARB_vertex_program* y *ARB_fragment_program* [4] de OpenGL. Para la generación de programas sobre GPU se aconseja el uso del lenguaje gráfico de alto nivel CG [5,11].

2. PROYECTO

El proyecto está dividido en tres partes independientes (igual al número de alumnos). El encargado de cada parte implementará un módulo de cálculo y visualización, así como una demo que muestre de forma interactiva la funcionalidad del módulo correspondiente. Finalmente, se realizará una demo global que haga uso de la funcionalidad de los tres módulos. A continuación se dan las líneas generales para la implementación de cada módulo, posteriormente se darán más detalles.

2.1 Módulo1: Simulación y visualización de la superficie de un fluido contenido en un recipiente cúbico sobre el que incide una luz direccional.

Para la implementación de este módulo se requiere aplicar técnicas de *environment mapping* (capítulo 7 de [6]). A continuación se muestran las fases de las que consta el módulo:

- Simulación (inicialmente software) del movimiento de la superficie del fluido mediante la Ecuación. de Ondas 2D.
- Cálculo de iluminación teniendo en cuenta la Reflexión, Refracción, Efecto Fresnel y Dispersión Cromática.
- El contenedor cúbico se supone compuesto por 5 planos texturizados. Como recinto exterior al contenedor se supone una habitación con 5 planos texturizados.
- Tras implementar la demo con recipiente cúbico, se procederá a implementar las versiones con recipiente semiesférico, semicilíndrico vértical y horizontal, y cónico.

2.2 Módulo2: Simulación de Ecuaciones Físicas sobre GPU/CPU y comparación entre ambas.

Este módulo está compuesto por las siguiente fases:

- Simulación de la Ecuación de Ondas 2D sobre GPU/CPU y comparación de tiempos.
- Simulación del movimiento de una malla geométrica cuyos puntos siguen la Ec. de Ondas 2D. Este proceso se realizará integramente en GPU. Estudiar el ejemplo sobre simulación de telas que tiene NVIDIA en [7].
- Idem para la Ec. de Reacción-Difusión. Modelo de Gray-Scott. Ver [8].
- Idem para la Ec. de Navier-Stokes 2D. Ver [9].

2.3 Módulo3: Simulación de la dinámica de una malla 3D ante una explosión.

Las fases de este módulo son:

- Simulación del comportamiento dinámico que sufre una malla 3D ante una explosión. Se supone que dicho objeto se descompone en un conjunto de subobjetos que describen ciertas trayectorias. Ver capítulos relacionados en [10].
- Detección de la colisión de los subobjetos resultantes de la explosión con paredes o fronteras.
- Simulación de choques elásticos e inelásticas cuando dichos subobjetos colisionan con paredes o fronteras.

3. REFERENCIAS

[1] http://www.nvidia.com
[2] http://www.ati.com
[3] http://www.opengl.org
[4] http://developer.nvidia.com/object/nvidia_opengl_specs.html
[5] http://developer.nvidia.com/page/cg_main.html
[6] Randima Fernando, Mark J. Kilgard. 2003. *The Cg Tutorial: The definitive guide to programmable real-time graphics*. Addison-Wesley.
[7] http://developer.nvidia.com/object/demo_cloth_simulation.html
[8] http://www.cgshaders.org/shaders
[9] J. Krüger, R. Westermann. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. SIGGRAPH 2003.
[10] David M. Bourg. 2001. *Physics for Game Developers*. O'Reilly.
[11] William R. Mark, R. Steven Glanville, Kurt Akeley, Mark J. Kilgard. Cg: A system for programming graphics hardware in a C-like language.

2. Descripción del Proyecto:

En un inicio, formamos el grupo con el propósito de realizar un Proyecto que fuese a implementar rutinas de gráficos. Al principio no conocíamos muy bien la utilidad de CG, pero en cuanto vimos las posibilidades que ofrecía lo tuvimos claro.

Introducción de CG:

Es un lenguaje de programación para las tarjetas gráficas de última generación. “C for graphics” es una adaptación del lenguaje C, para poder utilizar el hardware gráfico que llevan las tarjetas. El poder programar el proceso interno que se lleva acabo dentro de la tarjeta gráfica, a la hora de pintar las cosas en pantalla, permite realizar una gran cantidad de efectos especiales. Es una forma fácil de decirle a la tarjeta, que es lo que quieres que haga con lo que se le a mandado pintar previamente.

El hardware interno de las tarjetas, actualmente es bastante potente, y comparable con la CPU. La GPU dispone de un hardware interno preparado para procesar unidades vectoriales de 4 float. Esto la hace bastante eficiente en cálculos vectoriales. Actualmente el procesador Intel a 2.4 GHz Pentium 4 tiene 55 millones de transistores, mientras que la GeForce FX de NVIDIA tiene una GPU que dispone de 125 millones de transistores. Con este dato nos podemos hacer una idea de la potencia que se desarrolla en las tarjetas de última generación.

El lenguaje CG nos permite hacer muchas cosas similares ha C, pero todavía tiene bastantes limitaciones debido a las restricciones de la arquitectura de la tarjeta utilizada. La compilación de un programa CG se hace de forma similar a los programas de C. El resultado es un programa en lenguaje ensamblador para la tarjeta gráfica como podría ser el siguiente:

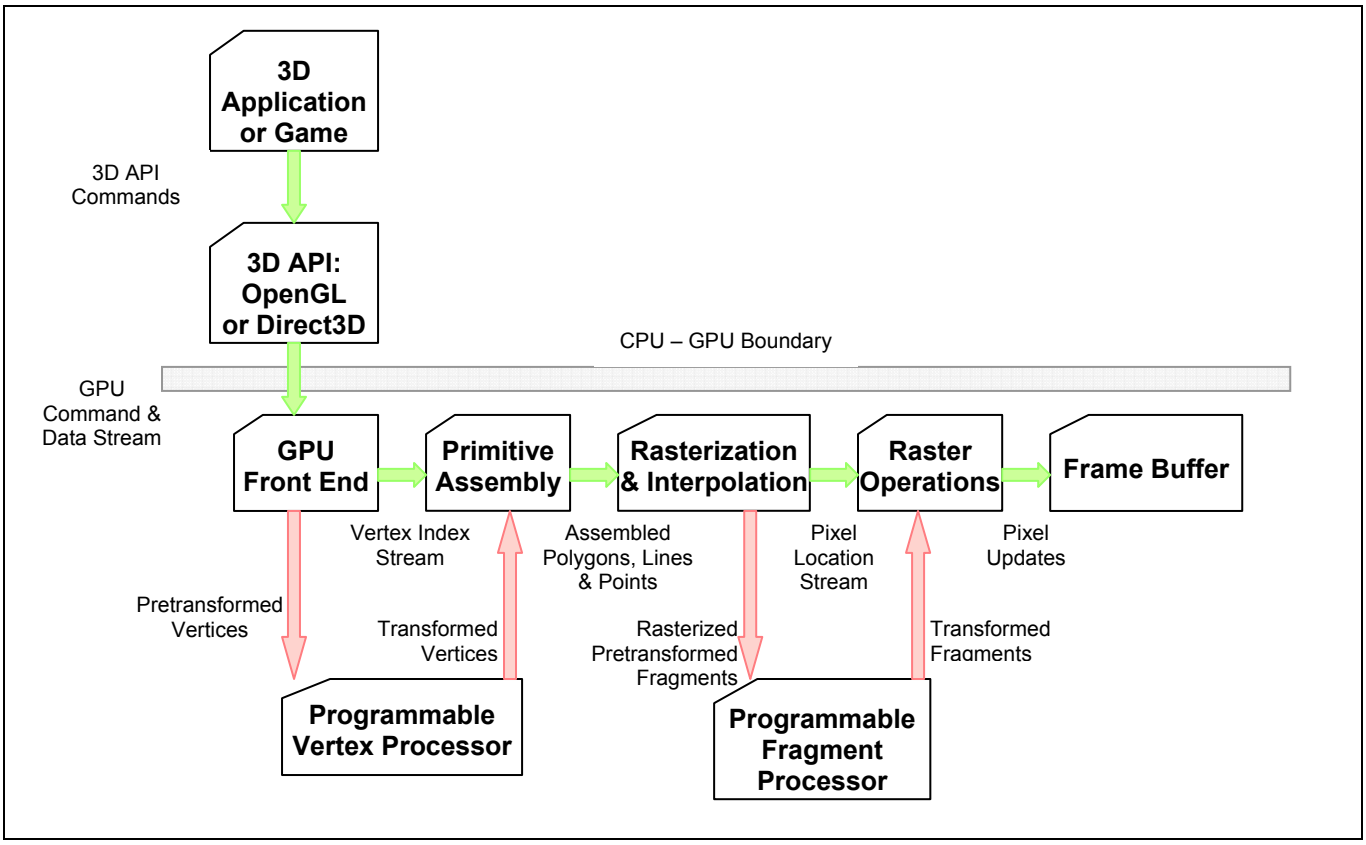
```
. . .
DEFINE LUMINANCE = {0.299, 0.587, 0.114, 0.0};
TEX   H0, f[TEX0], TEX4, 2D;
TEX   H1, f[TEX2], TEX5, CUBE;
DP3X  H1.xyz, H1, LUMINANCE;
MULX  H0.w H0.w, LUMINANCE.w;
MOVH  H2, f[TEX3].wxyz;
. . .
```

Para utilizar CG en la tarjeta se debe escribir un programa que luego se va a cargar, antes de pintar. El programa a ejecutarse en medio del *Pipeline* y dependiendo del tipo de programa se ejecutara en un sitio o en otro. Se pueden hacer dos tipos:

- **Vertex Program**
- **Fragment Program**

Los *Vertex Program*, como su propio nombre indica son para realizar transformaciones sobre los vértices pasados a la tarjeta. Para cada vértice que pase por el *Pipeline* de la tarjeta se ejecutará nuestro *Vertex Program* una vez. Sin embargo el *Fragment Program* es para realizar transformaciones de los fragmentos que van a representar en pantalla nuestro polígono, punto, línea…Básicamente es para hacer transformaciones sobre el color que va representar ese fragmento en la pantalla. De modo que el *Fragment Program* se ejecutará una vez para cada fragmento del objeto pintado. Esto quiere decir que se puede ejecutar una cantidad muy elevada de veces lo cual puede ralentizar mucho la renderización de la escena.

Veamos como interfieren ambos programas en el *Pipeline*:



Objetivos generales:

El objetivo principal del proyecto es explotar la capacidad de cálculo de las tarjetas, haciendo uso de CG. Para ello hemos escogido la representación de varios fenómenos físicos de los típicos, que requieren gran cantidad de cálculo.

En un inicio, ninguno de los tres componentes del grupo conocemos en profundidad el lenguaje de programación de CG, y nuestros conocimientos sobre programación de gráficos es el básico. Así que, nuestra primera meta es un aprendizaje exhaustivo del lenguaje CG, y las tarjetas de última generación. Para ello realizamos una primera etapa de documentación.

Una vez conocido el manejo de las tarjetas, limitaciones, etc. El proyecto se divide en tres módulos. De este modo cada uno de los miembros podrá centrarse en ciertos fenómenos físicos, y desempeñar su tarea exclusivamente en ellos. Esto resulta el modo más adecuado, puesto que cada uno va ha utilizar diferentes técnicas de implementación, y el desarrollo de estas, es completamente modular.

Los fenómenos físicos que se van a implementar buscan el modo de explotar la potencia de las tarjetas. En el caso del primer y el tercer módulo se realizarán las pruebas directamente sobre CG, para renderizar en tiempo real los fenómenos. Con el fin demostrar la potencia gráfica de la que disponemos con las tarjetas. En el tercer módulo se emplearán ambas técnicas para renderizar los fenómenos. Usando la CPU y la GPU, con el fin de sacar una comparativa del rendimiento.

3. Módulo 2:

Simulación de Ecuaciones Físicas sobre GPU/CPU y comparación entre ambas

El módulo consta de las siguientes fases:

1. **Simulación de la Ecuación de Ondas 2D sobre GPU/CPU y comparación de tiempos.**
2. **Simulación del movimiento de una malla geométrica cuyos puntos siguen la Ec. de Ondas 2D. Este proceso se realizará íntegramente en GPU.**
3. **Simulación de la Ec. de Reacción-Difusión. Modelo de Gray-Scott.**
4. **Simulación de Fuego sobre GPU/CPU y comparación de tiempos.**

Este módulo, va orientado a la comparación de rendimientos. En teoría se busca demostrar que utilizar la tarjeta gráfica para los cálculos aumenta el rendimiento.

Conseguir unos buenos resultados para la GPU, dependerá mucho de la lo fácil que sea exportar la implementación de CPU. La transformación consistirá en crear los *Vertex Program* y *Fragment Program* necesarios, para sustituir los algoritmos del cálculo de las ecuaciones físicas.

Más adelante veremos los resultados obtenidos, y como la GPU puede superar la velocidad de cálculo de la CPU en situaciones determinadas. En general, realizando una implementación que aproveche la capacidad vectorial de cálculo de la GPU, los resultados obtenidos superan a los de la CPU.

Para representar las ecuaciones y los cálculos, he integrado todo dentro de una sencilla y manejable demo, de la cual ire comentando cosas más adelante.

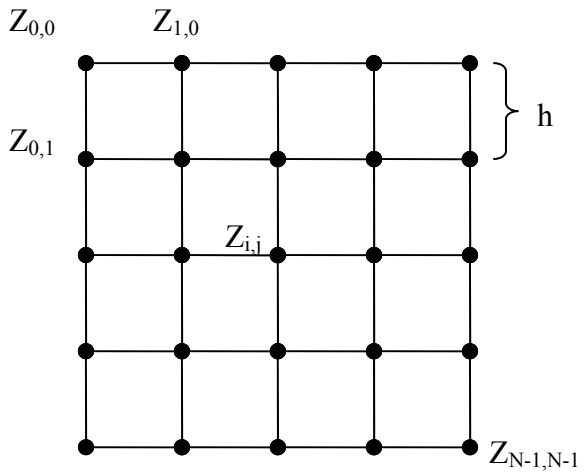
3.1 Simulación de la Ecuación de Ondas 2D sobre GPU/CPU y comparación de tiempos:

Descripción de la simulación:

Vamos a realizar la simulación de la Ecuación de Ondas 2D sobre una malla de (2^Nx2^N). La ecuación que vamos a implementar está en función del tiempo. Con ella calculamos la altura de un punto en un instante dado. Vamos a utilizar la siguiente:

$$\frac{\partial^2 z}{\partial t^2} = c^2\left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2}\right)$$

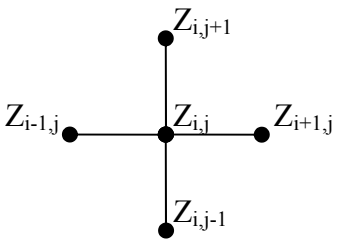
Donde ***c*** representa la velocidad a la que las ondas viajan a través de la superficie. La representación de esta superficie la vamos a hacer con una malla de la forma siguiente:



La ecuación discretizada nos quedaría en función de los dos estados anteriores (actual y anterior al actual). Si despejamos la altura ($z_{i,j}^{n+1}$), que es valor que nos interesa actualizar, dado que la onda se propaga por la superficie 2D, desplazando los puntos de la malla en el eje Z, nos quedaríamos con la siguiente ecuación:

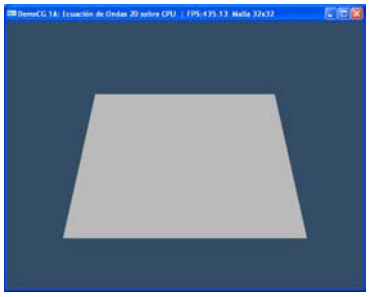
$$z_{i,j}^{n+1} = \frac{c^2\Delta t^2}{h^2}\left(z_{i+1,j}^{n} + z_{i-1,j}^{n} + z_{i,j+1}^{n} + z_{i,j-1}^{n}\right) + \left(2 - \frac{4c^2\Delta t^2}{h^2}\right)z_{i,j}^{n} - z_{i,j}^{n-1}$$

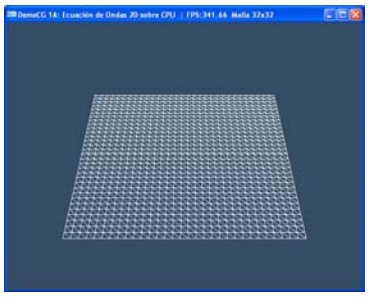
Como podemos observar en la ecuación, el valor de z en el instante n+1 depende de los dos anteriores, y de los cuatro puntos más cercanos.

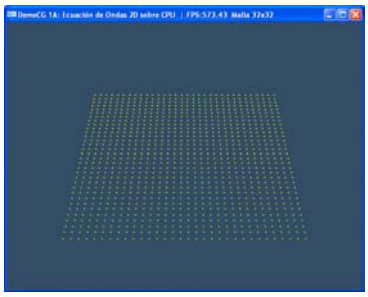


Métodos utilizados para la implementación:

Para representar la malla vamos a utilizar una estructura llamada *Grid,* que va a contener todos los datos necesarios, posiciones, tamaño del lado, número de lados… La malla creada formaría una superficie que se visualizaría del siguiente modo:


Superficie de color liso


Malla que forma la superficie


Vértices de la malla

La malla mapeada con una textura

El *Grid* contiene tres listas que almacenan las posiciones de los puntos. Una para la posición actual, otra para la anterior, y otra para futura. Además tenemos otra lista para las normales en cada punto, que se irá actualizando a la par que las de posición.

A cada paso de tiempo se llama a una función "*update*" que nos hace el cálculo de la Ecuación de Ondas 2D para cada punto de la malla.

Implementación de la **CPU**:

En el caso de la implementación para CPU, su función para actualizar los valores, va a consistir en la ejecución de un simple algoritmo. Este algoritmo está formado por un bucle, que recorre los puntos de la malla aplicándoles la ecuación, y actualizando sus normales. Este algoritmo es el siguiente:

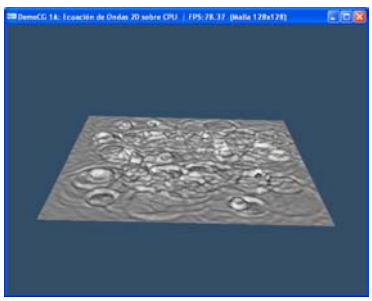
```
// Calculamos las nuevas posiciones en funcion de la Ec. de ondas
// para evitar cálculos innecesarios, calculamos A y B antes de entrar en el bucle
float A = (speed * time / grid.sideX) * (speed * time / grid.sideX);
float B = 2 - 4 * A;
long i, j;
float n[3];
// Recorremos un bucle que actualiza los valores de las Alturas y las nuevas normales
for (j = 1; j < (grid.numVertexY-1); j++)
	for (i = 1; i < (grid.numVertexX-1); i++) {
		// Calculamos Z según la Ecuación de Ondas 2D discretizada
		grid.vertex[grid.dest][j*grid.numVertexX + i][2] =
			A * (grid.vertex[grid.current][j*grid.numVertexX + (i-1)][2] +
			grid.vertex[grid.current][j*grid.numVertexX + (i+1)][2] +
			grid.vertex[grid.current][(j-1)*grid.numVertexX + i][2] +
			grid.vertex[grid.current][(j+1)*grid.numVertexX + i][2]) +
			B * grid.vertex[grid.current][j*grid.numVertexX + i][2] -
			grid.vertex[grid.previous][j*grid.numVertexX + i][2];
		grid.vertex[grid.dest][j*grid.numVertexX + i][2] *= damping;
		// Ahora actualizamos las normales
		n[0] = grid.vertex[grid.dest][j*grid.numVertexX + (i-1)][2] -
			grid.vertex[grid.dest][j*grid.numVertexX + (i+1)][2];
		n[1] = grid.vertex[grid.dest][(j-1)*grid.numVertexX + i][2] -
			grid.vertex[grid.dest][(j+1)*grid.numVertexX + i][2];
		n[2] = grid.sideX * 2;
		// Normalizamos n
		float mod = sqrt(n[0]*n[0] + n[1]*n[1] + n[2]*n[2]);
		n[0] /= mod;
		n[1] /= mod;
		n[2] /= mod;
		grid.normals[j*grid.numVertexX + i][0] = n[0];
		grid.normals[j*grid.numVertexX + i][1] = n[1];
		grid.normals[j*grid.numVertexX + i][2] = n[2];
	}
// Actualizamos las variables que indican en que lista se encuentra la información
int temp = grid.current;
grid.current = grid.dest;
grid.dest = grid.previous;
grid.previous = temp;
```

En la demo realizada, está implementado como Demo1A. Se puede ver el resultado visualmente, o ver la velocidad de cálculo, cancelando la renderización en pantalla. Al cancelar la renderización en pantalla, la demo a cada paso de tiempo realiza un *update*, de este modo podemos ver a que velocidad corre la demo en la parte superior de la ventana con o sin visualizar.

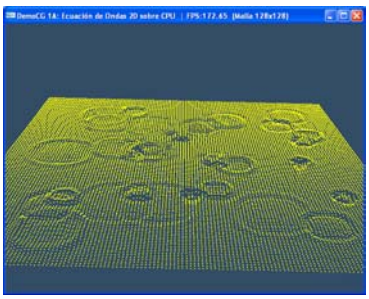
** Para intercambiar entre modo de renderizado en pantalla, o modo sin renderizado (solo cálculo), en la demo, debemos utilizar la tecla 'R'.*

Para que el resultado sea una sensación visible de lo que realiza la Ecuación de Ondas 2D, hemos metido perturbaciones en la superficie de un modo aleatorio. Cada perturbación, desplaza una pequeña zona de vértices de la malla, en dirección positiva sobre el eje Z. La impresión resultante es como si una gota de agua acabase de perturbar una superficie líquida. El efecto visual resultante al insertar perturbaciones sería el siguiente:

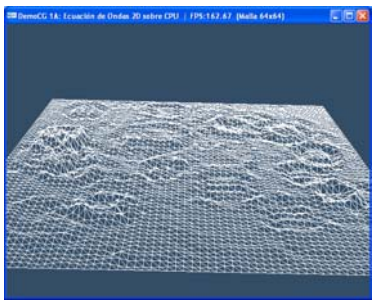
La superficie mapeada con al textura tras haber insertado perturbaciones.



Superficie



Vértices



Malla

** Para insertar una perturbación en la demo se utiliza la tecla 'B'.*

Implementación de la **GPU**:

Lo primero que nos planteamos llegado este punto, es, como pasarle la información de los estados anteriores a la tarjeta, para que pueda calcular las nuevas posiciones y las nuevas normales. Además, al igual que hacíamos en el otro caso, nos interesa separar lo que es el cálculo, que sería la actualización, de lo que es la visualización de la malla. De modo que podamos comparar las velocidades de cálculo sin necesidad de renderizar. Esto lo logramos utilizando los *Pixel Buffer*.

Los *Pixel Buffer* son unos buffers que almacenan valores de píxel como pueden ser los de una textura, en la memoria de la tarjeta. Nosotros los vamos a utilizar para guardar los valores de las posiciones de los puntos, de modo que al igual que antes disponíamos de tres listas de floats, ahora dispondremos de tres *PBuffers*. Y además de estos tres, tendremos uno más que utilizaremos para las normales, al igual que antes teníamos una lista.

Los *PBuffers* que vamos a utilizar tendrán que almacenar, al menos tres valores de float, para guardar los valores XYZ de las posiciones. Por lo tanto utilizaremos *PBuffers* con formato "`float=32 rgb textureRECT`", que indica que el formato para los colores del píxel es un float de 32bits, y que los pixels van a utilizar sus valores rgb que representarán xyz. Además la textura del *PBuffer* va a ser una *textureRECT* que es el tipo de textura rectangular, adecuada para nuestra malla rectangular. Para cargar unos valores iniciales en estos PBuffers, lo haremos como si fuésemos a pintar con una textura cualquiera, del siguiente modo:

```
// Activamos el PBuffer que queremos cargar
_pbuffer->Activate();
// Llamamos a la funcion que pinta en la textura active, los valores del array "vertex"
glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_FLOAT_RGB_NV, grid.numVertexX,
              grid.numVertexY, 0, GL_RGB, GL_FLOAT, (GLfloat *) &vertex[0]);
// Ahora para que se cargen esos valores en el PBuffer que tenemos activado como salida
// llamamos a esta función que pinta un Quad de la dimensiones del PBuffer
drawQuad();
_pbuffer->Deacivate();
```

El *PBuffer* se activa para que la salida de la tarjeta vaya a pintarse en sus valores de píxel. Además de está funcionalidad de los *PBuffers* vamos a ver la otra que nos interesa, que es la de usar lo que tienen almacenado como una textura cargada en memoria.

Ahora ya sabemos como guardar la información en la tarjeta. Vamos a ver, cómo podemos utilizarla con ayuda de un *Fragment Program*. Para realizar el algoritmo de cálculo de la Ecuación de Ondas 2D, necesitamos acceder a los *PBuffers* que guardan la información de atrás, y con estos actualizar los valores del otro. La solución es la siguiente, realizamos un *Bind* de los *PBuffers* que guardan al información que necesitamos leer, y activamos como salida el *PBuffer* restante. Ahora para que se actualicen todos los valores de píxel del *PBuffer* de salida llamamos a la función *drawQuad* que nos pinta un cuadrado de las dimensiones del *PBuffer*. En este momento ya tenemos que tener cargado el Fragment Program que se encargará de calcular la ecuación. Veamos como quedaría el código de la función *update*:

```
// Activamos el PBuffer al que le vamos a actualizar los datos
vPBuffer[dest]->Activate();
// Ponemos el anterior y el acutal como texturas
glActiveTextureARB(GL_TEXTURE0_ARB);
glBindTexture(GL_TEXTURE_RECTANGLE_NV, vPBufferTexture[previous]);
vPBuffer[previous]->Bind(WGL_FRONT_LEFT_ARB);
glActiveTextureARB(GL_TEXTURE1_ARB);
glBindTexture(GL_TEXTURE_RECTANGLE_NV, vPBufferTexture[current]);
vPBuffer[current]->Bind(WGL_FRONT_LEFT_ARB);
CFragmentProgram::on();
```

```
// Cargamos el Fragment Program que cálcula la Ec. de Onda 2D
waveEc->bind();
// Calculamos A y B para no tener que hacerlo en
// todas las veces que se ejecuta el Fragment Program
float A = (speed * time/grid.sideX);
A = A*A;
float B = 2.0f - 4.0f * A;
waveEc->setNamedParam( "param", A, B, 0.0f, damping);
// Pintamos el cuadrado para que se realicen los cálculos
drawQuad();
CFragmentProgram::off();
vPBuffer[previous]->Release(WGL_FRONT_LEFT_ARB);
vPBuffer[current]->Release(WGL_FRONT_LEFT_ARB);
vPBuffer[dest]->Deactivate();
int temp = previous;
previous = current;
current = dest;
dest = temp;
// Actualizamos las normales
calculateNormals();
```

Veamos el Fragment Program, en el hemos adaptado el cálculo de la Ec. de Ondas 2D, que se realizará para cada píxel que representa un vértice de la malla:

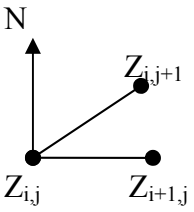
```
void main( in  float2 inTexCoord : TEX0,
           out float3 outColor   : COLOR,
           uniform texobjRECT previous,
           uniform texobjRECT current,
           uniform float4 param = float4(1.0f, 0.1f, 1.0f, 0.99f))
{
        float A = param.x;
        float B = param.y;
        float2 s = inTexCoord.xy;
        // Cálculamos la nueva altura
        float3 c  =    f3texRECT(current, s);
        float3 c1 =    f3texRECT(current, s + float2(-1, 0));
        float3 c2 =    f3texRECT(current, s + float2( 1, 0));
        float3 c3 =    f3texRECT(current, s + float2( 0,-1));
        float3 c4 =    f3texRECT(current, s + float2( 0, 1));
        float3 p  =    f3texRECT(previous, s);
        float h = ((A * (c1.z + c2.z + c3.z + c4.z)) + (B * c.z )- p.z)*param.w;
        // Actualiza el valor de la posición
        outColor = float3(c.x, c.y, h);
}
```

Como se puede ver en la función *update* tenemos que recalcular las normales por separado. Esto se debe, a que este método, solo da una salida que es la posición de los vértices. Para actualizar las normales necesitamos reescribir el PBuffer que las contiene. Y para calcularlas necesitamos el PBuffer de las nuevas posiciones. A partir de este nos será fácil sacar las normales, puesto que con unos simples productos vectoriales de 3 posiciones conseguiremos el vector resultante. Veamos el *Fragment Program* que lo realiza:

```
void main( in  float4 inColor0   : COL0,
           in  float2 inTexCoord : TEX0,
           out float3 outColor   : COLOR,
           uniform samplerRECT current   )
{
       // Sacamos la perpendicular a la superficie formada por los tres puntos
        float3 x = f3texRECT(current, inTexCoord );
        float3 x2 = f3texRECT(current, inTexCoord + float2(1.0, 0.0) );
        float3 x3 = f3texRECT(current, inTexCoord + float2(0.0, 1.0) );
        float3 N = cross(x2 - x, x3 - x);
       // La normalizamos y se la escribimos a la salida
        N = normalize(N);
        outColor = N;
}
```

Esto representa lo que hace el *Fragment Program* para calcular las normales:

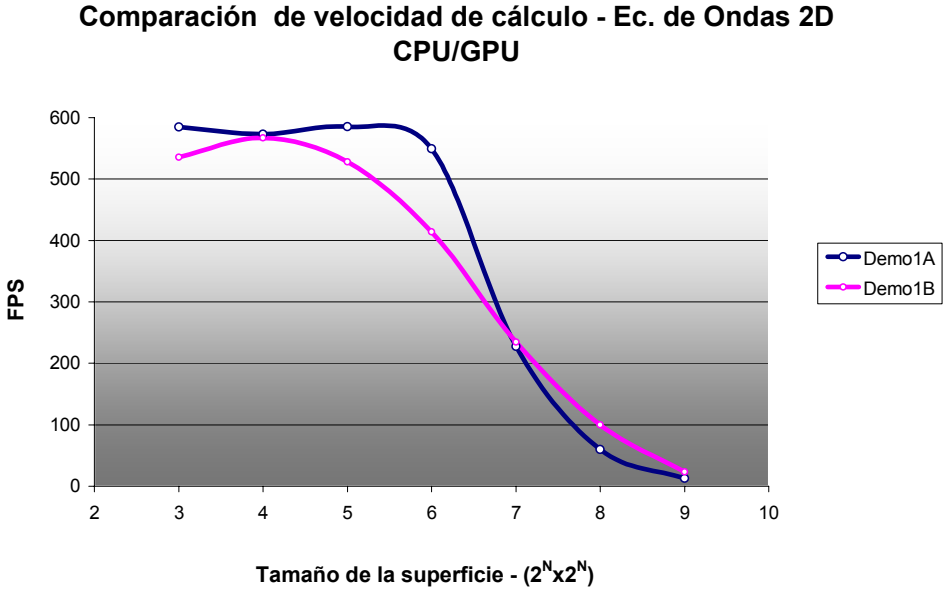
$$N = \text{cross}\ (Z_{i,j+1} - Z_{i,j},\ Z_{i+1,j} - Z_{i,j})$$



En la demo realizada, está implementado como Demo1B. Y los resultados visuales son los mismos que nos da la implementación en CPU.

Comparación de tiempos:

Para realizar una comparación de tiempos, en la demo hay una forma de generar un Log que nos va guardando los tiempos, y el estado de la demo. A partir de este log hemos sacado los siguientes datos que nos muestran la velocidad de cálculo de ambas implementaciones:



Se puede apreciar que en un principio la implementación en CPU, es más rapida haciendo el cálculo. Pero luego, cuando el tamaño de la malla se hace considerablemente mayor, la velocidad de la implementación de GPU duplica la velocidad de la CPU. A simple vista no parece un aumento muy elevado pero hay que tener en cuenta que la escala del eje X que tiene el tamaño de la malla, crece exponencialmente.

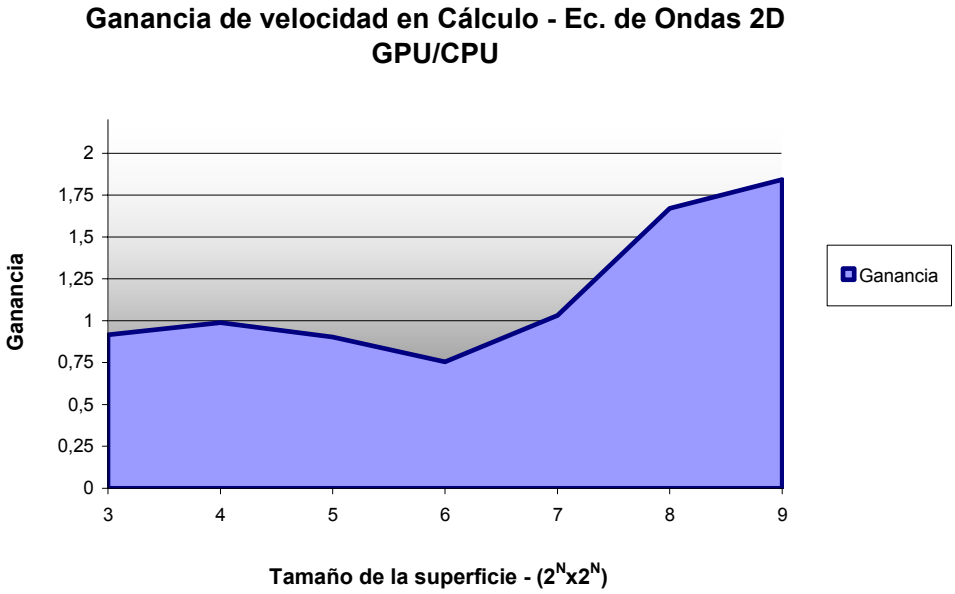
La implementación de CPU solo necesita realizar las operaciones, mientras que la GPU tiene que activar los *PBuffers* y el *Fragment Program*, y realizar el cálculo. Cuando el numero de vértices es muy pequeño. Este retardo para la segunda implementación resulta un retraso considerable, pero cuando el numero de vértices es mucho mas grande este retardo se hace despreciable. Es entonces cuando se ve la capacidad de cálculo vectorial que tiene la GPU. En la gráfica resultante los valores se ven poco espaciados con tamaños grandes, pero los valores numéricos que representan son muy diferentes. Por ejemplo con (2^9x2^9) la CPU tiene una media de FPS igual a 12,733 mientras que la GPU tiene 23,4735 que es casi el doble.

Si nos fijamos en la implementación de la GPU, podemos hacernos una idea de donde se aprovecha la capacidad de calculo, de donde sale esa diferencia de velocidad de cálculo. En el *Fragment Program* que calcula la Ec. de Onda 2D no se hace gran uso de la capacidad de la tarjeta. Pero sin embargo, en el que actualiza las normales se hace todo con cálculos vectoriales.

Si hacemos una comparativa obteniendo la ganancia en velocidad de este modo:

$$G = \frac{FPS_{implementación\_GPU}}{FPS_{implementación\_CPU}}$$

Obtenemos este resultado:



En conclusión, podemos decir que la implementación de este fenómeno físico se puede realizar con la tarjeta obteniendo una ganancia de velocidad. A pesar de que la evaluación de la Ec. de Ondas 2D no nos permita un aprovechamiento al máximo de sus posibilidades.

3.2 Simulación del movimiento de una malla geométrica cuyos puntos siguen la Ec. de Ondas 2D. Integramente en GPU:

Descripción de la simulación:

Vamos a realizar la simulación de la Ecuación de Ondas 2D sobre una malla de (2^Nx2^N) al igual que hacíamos en la fase anterior. La finalidad de esta fase del proyecto, es buscar una visualización de la implementación en GPU que aproveche los recursos que tiene la tarjeta.

No solo vamos a tener un cálculo más rápido, sino que vamos a tener una visualización más rápida. Para conseguir esto, lo mejor que podemos hacer es trasladar toda la información necesaria para calcular, y para visualizar, a la tarjeta.

Los resultados de visualización son idénticos a los casos anteriores, pero con un aumento considerable de los FPS.

Métodos utilizados para la implementación:

Los cálculos y el almacenamiento de los datos en los *PBuffers* se continúan haciendo como en la fase anterior. Ambos están integrados en la GPU que es lo que queremos para está simulación.

En cambio, las estructuras que nos guardaban los datos que más tarde se utilizaban para pintar, van a ser cambiadas por memoria de la tarjeta gráfica. Para pintar la malla, necesitamos una array con los valores de las posiciones de los vértices, otro para sus normales, y otro para las coordenadas de textura. Estos arrays los vamos a almacenar en memoria de la tarjeta utilizando extensiones de NVIDIA.

Para guardar estos arrays en memoria de la tarjeta, hemos implementado la clase *RenderableVertexArray*, que nos permite almacenar toda la información de la malla. Esta clase nos facilitará el manejo de la memoria, especialmente para nuestro caso que tenemos que intercambiar datos con los de los *PBuffers*. Nos va ha permitir leer los datos de un *PBuffer* para actualizar los valores con una simple llamada a una función:

```
// Leemos las nuevas posiciones que se han calculado en el PBuffer
vertexArray->ReadVertex(vPBuffer[current]);
// Ahora hacemos lo mismo para las nuevas normales
vertexArray->ReadNormal(nPBuffer);
```

Solo se actualizan los valores modificados. Las coordenadas de textura se mantienen constantes durante todo el tiempo, no hace falta actualizarlas.

También nos permite dibujar la malla a través de otra simple llamada:

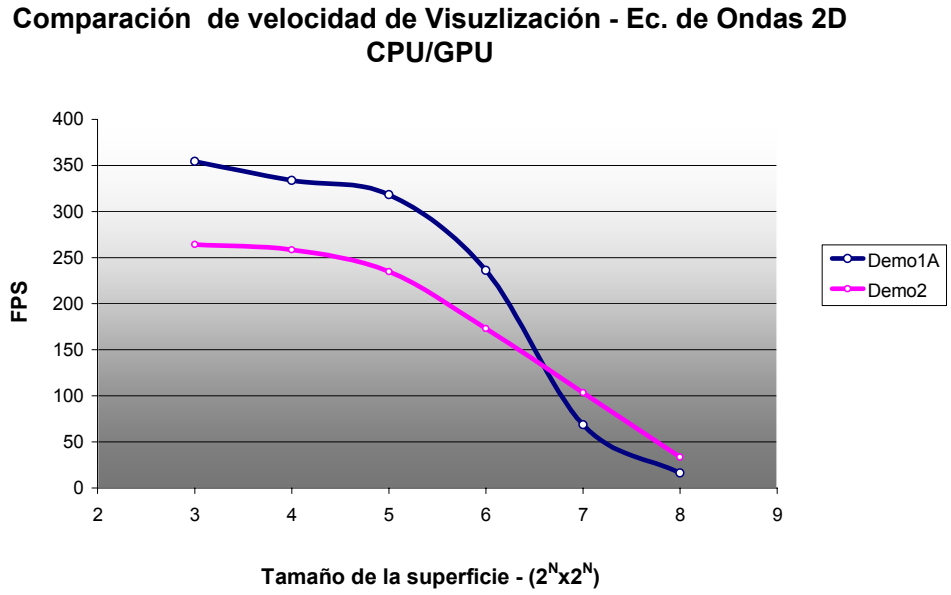
```
// Antes de pintar el array configuararamos los modos de pintado, las texturas…
glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
// Llamamos a la function pasandole los indices del array que represetan la malla
vertexArray->Draw(GL_TRIANGLE_STRIP, numIndices, indices);
```

Ahora ya tenemos independizado todo el cálculo y toda la visualización en la tarjeta gráfica. Ahora cada vez que pintemos la superficie no nos hará falta enviar de la CPU a la GPU todos los vértices, normales, y coordenadas de textura porque ya están en la tarjeta. Los resultados deberían ser bastante mejorados, al haber eliminado la continua tasa de transferencia con la tarjeta. Pero, hay que tener en cuenta que hoy en día las tarjetas gráficas tienen unos puertos de comunicación con la CPU que alcanzan unas velocidades muy grandes, como pueden ser los AGP x8 (es el que he utilizado para tomar las muestras de tiempos).

En la demo realizada, está implementado como Demo2.

Comparación de tiempos:

Vamos a comparar los tiempos de visualización de la implementación integra en CPU que teníamos en la Demo1A, con los de esta versión integra en la GPU, la Demo2. La diferencia es notable como podremos ver:



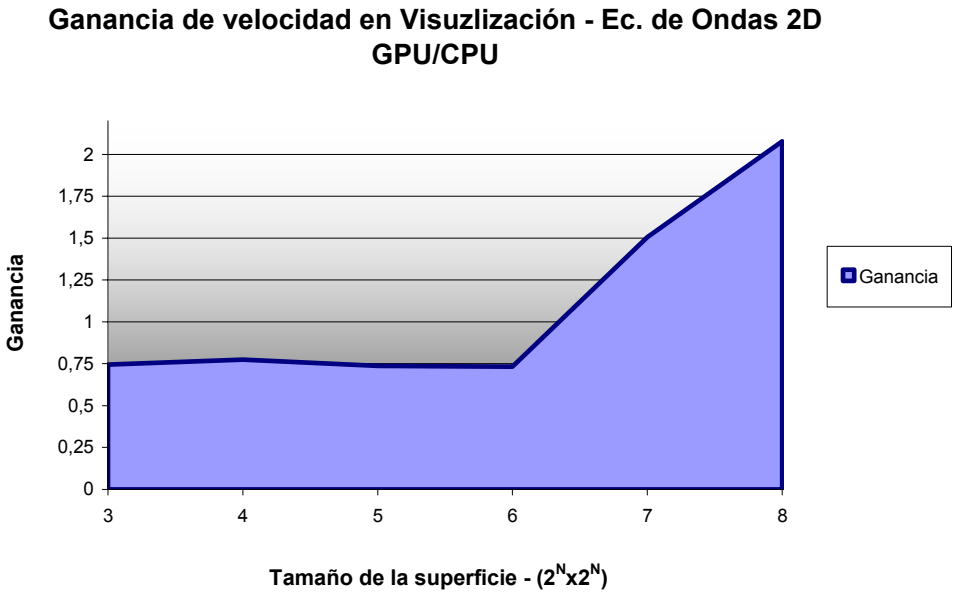
La gráfica que nos muestra los datos aparece muy similar a la de antes. Esto se debe a que el tiempo de cálculo influye en gran parte sobre la velocidad.

El método que hemos utilizado ahora nos permite visualizar la superficie, sin a penas retardo al leer de vuelta los valores calculados en el *PBuffer*. Esto es algo muy importante, porque si tuviésemos que leer de vuelta a la CPU los valores, esa ganancia que nos da utilizar CG se vería contrarrestada. De hecho, se mantiene esa ganancia como muestra gráfica. A simple vista pasa lo mismo que antes, parece una ganancia pequeña pero hay que tener en cuenta la escala del eje X que es exponencial.

Si hacemos una comparativa obteniendo la ganancia en velocidad de este modo:

$$G = \frac{FPS_{implementación\_GPU}}{FPS_{implementación\_CPU}}$$

Obtenemos este resultado:



Con esto, podemos concluir que una implementación en la GPU puede ser más eficiente utilizando las técnicas adecuadas, y dependiendo de la cantidad de cálculo.

3.3 Simulación de la Ecuación de Reacción-Difusión. Modelo de Gray-Scott:

Descripción de la simulación:

Esta simulación consiste en representar la Ecuación de Reacción-Difusión. La ecuación representa el comportamiento que tienen dos especies químicas que reaccionan entre si. La vamos a mostrar sobre una textura de (2^Nx2^N), que representa los niveles de concentración de las especies químicas que están reaccionando, según el color del píxel.

La Ecuación de Reacción-Difusión según el modelo de Gray-Scott es la siguiente:

Ecuaciones:

$$\frac{\partial u}{\partial t} = r_u \nabla^2 u - uv^2 + f(1-u)$$

$$\frac{\partial v}{\partial t} = r_v \nabla^2 v + uv^2 - (f+k)v$$

Reacción química:

$$U + 2V \rightarrow 3V$$

$$V \rightarrow P$$

Donde ***U***, ***V*** y ***P*** son las especies químicas, ***u*** y ***v*** representan las concentraciones, r_u y r_v son sus velocidades de difusión, ***k*** representa la velocidad de conversión de ***V*** a ***P***, y ***f*** representa la velocidad a la que se alimenta ***V*** y se vacía ***U***, ***V*** y ***P***.

Métodos utilizados para la implementación:

Para representar las concentraciones vamos a utilizar un *PBuffer,* que nos va servir como textura a la hora de mostrarlo en pantalla. Y a su vez nos va a servir para aplicar la ecuación con los *Fragment Progam* necesarios, que van a evaluar los valores de cada píxel en función de la reacción.

La ecuación está en función del estado anterior de la reacción. Por lo tanto vamos a necesitar dos *PBuffers*, uno para el estado anterior, y otro para el actual que es el que vamos a calcular. Como en las demos anteriores, vamos a tener una función *update* que nos actualiza los valores de píxel de la textura de concentración.

La forma de hacerlo seria la siguiente, para cargar los valores del estado anterior, pondríamos el *PBuffer* con el estado pasado como textura de entrada a la tarjeta. Para poder leer los datos con el *Fragment Program*. Activaríamos el *PBuffer* que va a almacenar los datos calculados, y cargaríamos el *Fragment Program* que realiza la Ec. de Reacción-Difusión. Y al final, para que se calculen los valores dibujamos un cuadrado con las dimensiones del *PBuffer*.

El siguiente código muestra la función *update*:

```
// Activamos como salida el PBuffer que recibe los nuevos valores de concentración
pbuffer[dest]->Activate();
// Lee limpiamos los valores residuales que pudiese llevar dentro
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// Ponemos los valores del cálculo anterior en una textura de entrada
glActiveTextureARB(GL_TEXTURE0_ARB);
glBindTexture(GL_TEXTURE_RECTANGLE_NV, concentrationTex[current]);
pbuffer[current]->Bind(WGL_FRONT_LEFT_ARB);

// Activamos los programas de CG que calculan la Ec. de Reacción-Difusión
CVertexProgram::on();
reactDif_vp->bind();
// c[0]..c[3] = modelview-projection matrix
CVertexProgram::trackModelProjMtxIde( 0 );
CFragmentProgram::on();
reactDif_fp->bind();

//Posibles valores para los parámetros de la Ecuación:
//                  rK= 0.0575f , 0.06f , 0.065f , 0.05f  , 0.05f
//                  rF= 0.025f  , 0.04f , 0.035f , 0.012f , 0.025f
float rK = 0.05f;
float rF = 0.012f;
float rDiffusionV = 0.0002f;
float rDiffusionU = 0.0004f;
reactDif_fp->setNamedParam("windowDims",sizeX,sizeY,-rK-rF+(1-655.36*rDiffusionV), 0);
reactDif_fp->setNamedParam("reactDifParam",655.36f*rDiffusionU,
                                    655.36f*rDiffusionV,rK,rF);

// Pintamos un cuadrado con las dimensiones del PBuffer
glBegin(GL_QUADS);
       glTexCoord2f(0    ,     0);   glVertex2f(0    ,     0);
       glTexCoord2f(sizeX,     0); glVertex2f(sizeX,     0);
       glTexCoord2f(sizeX, sizeY); glVertex2f(sizeX, sizeY);
       glTexCoord2f(0    , sizeY); glVertex2f(0    , sizeY);
glEnd();

// Desactivamos los PBuffers y el CG, y actualizamos las variables de estado
CFragmentProgram::off();
CVertexProgram::off();
pbuffer[current]->Release(WGL_FRONT_LEFT_ARB);
pbuffer[dest]->Deactivate();
int temp = current;
current = dest;
dest = temp;
```

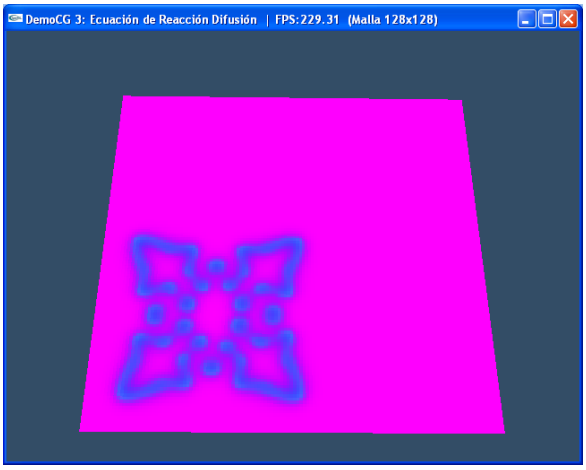
El *Fragment Program* recibe los valores del otro *PBuffer*, y con los parámetros de entrada calcula la ecuación. La Ec. de Reacción-Difusión se calcula tomando los valores de concentración de los píxel de alrededor. El *Vertex Program* utilizado, básicamente sirve para dejar pasar los valores introducidos por OpenGL. Veamos en profundidad con el *Fragment Program* cuales son los cálculos necesarios:

```
void main( in  float3 inColor    : COL0,
           in  float2 inTexCoord0 : TEX0,
           in  float3 inTexCoord1 : TEX1,
           out float3 outColor    : COLOR,
           uniform samplerRECT concentrationField,
           uniform float4 windowDims,
           uniform float4 reactDifParam )
{
       float2 uv = inTexCoord0;
       float2 centerSample = f2texRECT(concentrationField, uv);
       //Calculamos la difusion con los valores de los pixels de alrededor
       uv.x = (inTexCoord0.x > windowDims.x - 1.0f) ? 0.5f  : inTexCoord0.x + 1.0f;
       float2 diffusion = f2texRECT(concentrationField, uv);
       uv.x = (inTexCoord0.x <= 1.0f) ? windowDims.x - 0.5f : inTexCoord0.x - 1.0f;
       diffusion += f2texRECT(concentrationField, uv);
       uv = inTexCoord0;
       uv.y = (inTexCoord0.y <= 1.0f) ? windowDims.y - 0.5f : inTexCoord0.y - 1.0f;
       diffusion += f2texRECT(concentrationField, uv);
       uv.y = (inTexCoord0.y > windowDims.y - 1.0f) ? 0.5f  : inTexCoord0.y + 1.0f;
       diffusion += f2texRECT(concentrationField, uv);
       // Calculamos la media de los 4 pixels
       diffusion *= 0.25f * reactDifParam.xy;
       float2 reaction = centerSample.xx * centerSample.yy * centerSample.yy;
       reaction.x *= -1.0f;
       reaction.x += (1.0f - reactDifParam.x) * centerSample.x +
                        reactDifParam.w * (1.0f - centerSample.x);
       reaction.y += windowDims.z * centerSample.y;
       // Juntamos los valores de reacción y difusión para obtener el resultado
       outColor.xy = diffusion + reaction;
       outColor.z = inColor.z;
}
```

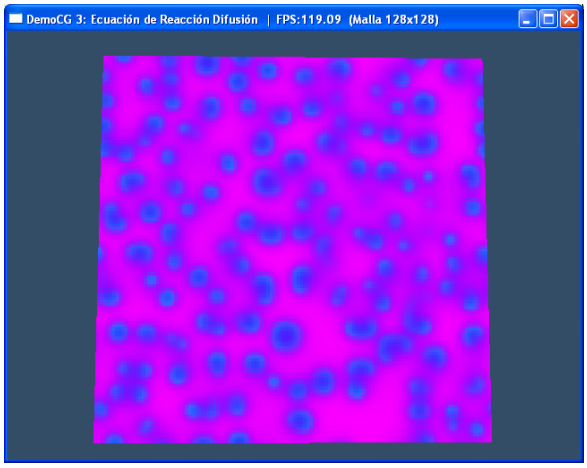
Todas las comprobaciones que realiza en función del tamaño de la ventana, son para que la reacción sea continua de un extremo a otro de la textura. De modo, que cuando la concentración está aumentando hacia arriba, y llega al borde de la textura, reaparece por debajo en el borde inferior. Y del mismo modo pasa con los bordes laterales.

Las comprobaciones de tipo *if-else,* no proporcionan un buen rendimiento de la GPU. Por eso, esta simulación no alcanza unas velocidades muy elevadas.

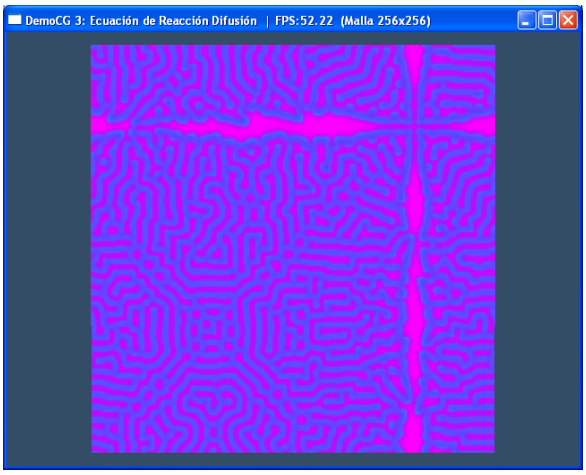
En la demo realizada, está implementado como Demo3. Y los resultados visuales son los siguientes, según sus parámetros de entrada ***f*** y ***k***:



f = 0,025f k = 0,0575f



f = 0,012f **k** = 0,05f



f = 0,04f **k** = 0,06f

f = 0,035f **k** = 0,065f

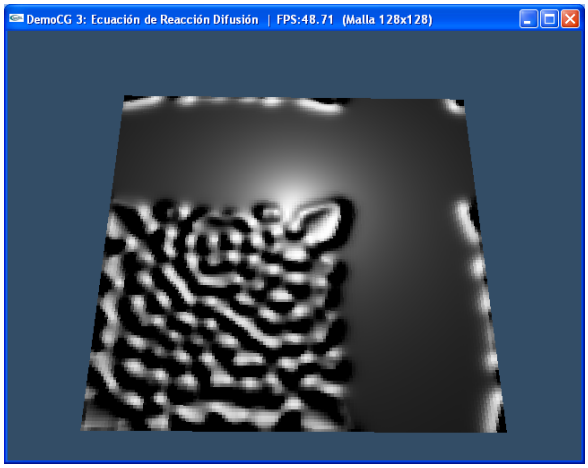
Como podemos observar la textura de concentraciones resultante, es muy similar a las texturas utilizadas para hacer un *Bump-Mapping*. Vamos a aprovechar esta característica de la textura que genera la reacción, para realizar un efecto similar al de la técnica del *Bump-Mapping*.

Para hacerlo introducimos un programa de CG, que nos va a calcular la supuesta pendiente de píxel a píxel dentro de la textura, en función de los valores de color de cada píxel. El *Fragment Program* obtendrá la pendiente, de la cual es fácil sacar una normal que nos servirá para asignar la cantidad de luz que reflejada. Para ello vamos a necesitar la posición del foco de luz como entrada.

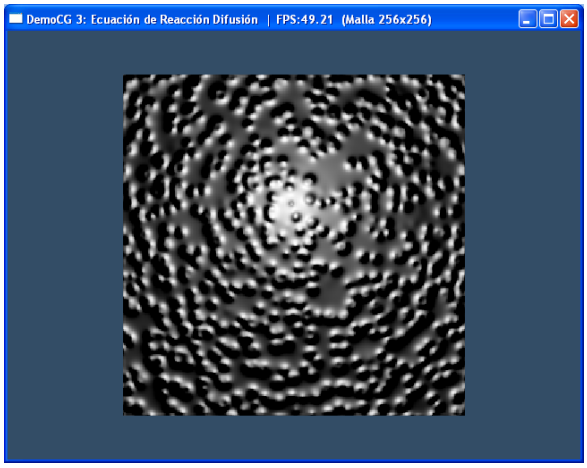
El *Fragment Program* implementado es muy similar al que se utiliza para realizar un *Bump-Mapping* normal y corriente. Veamos el código más en profundidad:

```
void main( in  float3 inColor   : COL0,
           in  float2 inTexCoord0 : TEX0,
           in  float3 inTexCoord1 : TEX1,
           out float3 outColor   : COLOR,
           uniform samplerRECT concentrationField,
           uniform float4 pixelSize,
           uniform float4 lightPosition )
{
        float3 gradientBottom = { 1, 1, 1};
        float3 gradientTop    = inColor;
        float2 texCoord= float2(inTexCoord0.x*pixelSize.z, inTexCoord0.y*pixelSize.w);
        float3 t = f3texRECT(concentrationField, texCoord).xxx;
        float2 uv = texCoord;
        uv.y += 1;
        float3 topSample = f3texRECT(concentrationField, uv);
        uv.y -= 2;
        float3 bottomSample = f3texRECT(concentrationField, uv);
        uv.y = texCoord.y;
        uv.x += 1;
        float3 rightSample = f3texRECT(concentrationField, uv);
        uv.x -= 2;
        float3 leftSample = f3texRECT(concentrationField, uv);
        // Calculamos la normal de la textura
        float3 tangentSpaceNormal;
        tangentSpaceNormal.x = (float)(4*(rightSample.x - leftSample.x));
        tangentSpaceNormal.y = (float)(4*(topSample.x - bottomSample.x));
        tangentSpaceNormal.z = 1;
        tangentSpaceNormal = normalize(tangentSpaceNormal);
        // Hacemos los cálculos necesarios para saber la iluminación que recibe
        float3 l = normalize( lightPosition.xyz - inTexCoord1 );
        float ndotl = max( dot(tangentSpaceNormal,l), 0 );
        outColor = ndotl * lerp(t, gradientBottom, gradientTop);
}
```

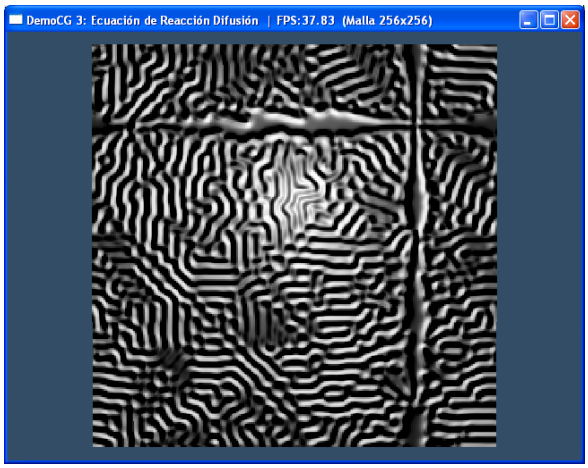
Utilizando esta técnica obtendremos los siguientes resultados visuales:



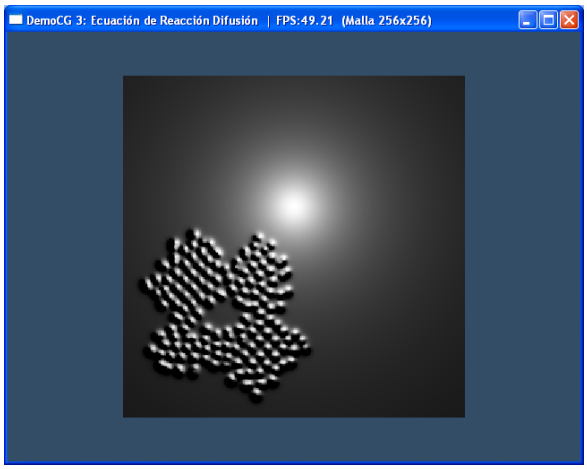
f = 0,025f k = 0,0575f



f = 0,012f **k** = 0,05f

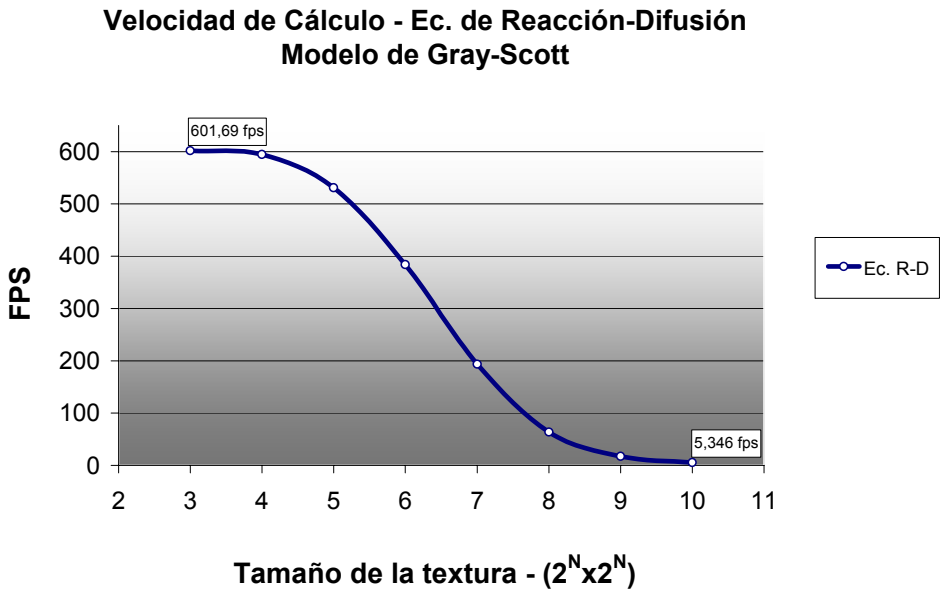


f = 0,04f **k** = 0,06f



f = 0,035f **k** = 0,065f

Haciendo un Log de los FPS para representar la velocidad de cálculo sin renderizar la imagen, obtenemos la siguiente gráfica:

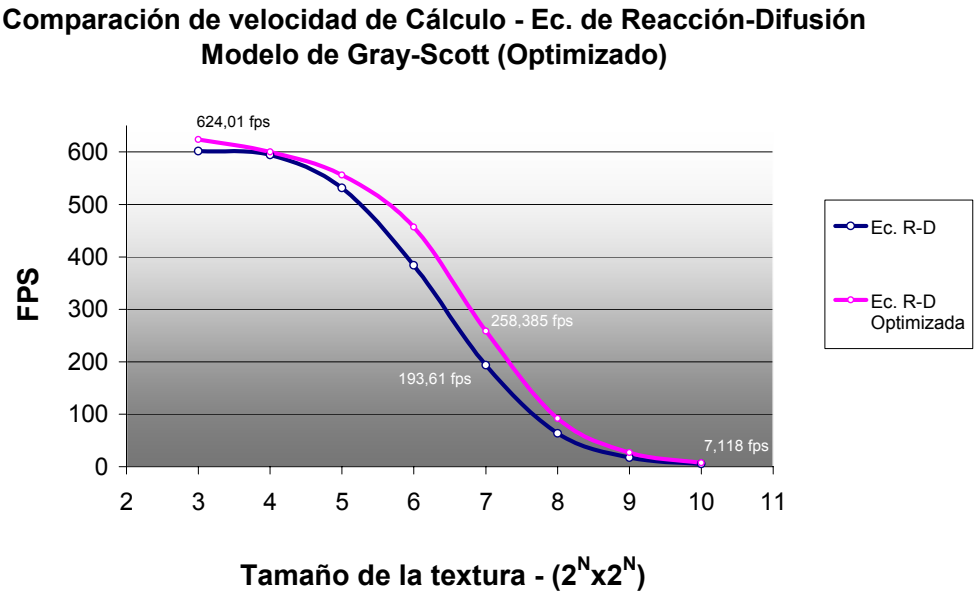


Como hasta ahora la gráfica la hemos tomado con tamaños que crecen exponencialmente. Por eso la curva decrece tan rápido.

Los resultados obtenidos son los que buscábamos. Como ya he comentado antes, en el cálculo de la ecuación hemos utilizado cuatro instrucciones de control de flujo del tipo ***"condición? instr1 : instr2;"***. Estas instrucciones hacen el programa de CG considerablemente más lento. Esto se debe, a que la GPU por alguna razón no resulta muy eficiente a la hora de ejecutar dichas instrucciones.

Si evitamos estas comprobaciones quitándolas, podemos comprobar que el programa hace el cálculo más deprisa. El quitarlas, influirá en la continuidad que conservaba la textura en sus extremos. Ahora cuado la concentración vaya aumentando hacia un extremo de la textura, se perderá al escapar por el borde, al contrario que antes que reaparecía por el borde opuesto.

Veamos los resultados de tiempo al modificarlo comparados con la versión anterior:



Viendo estos resultados, que muestran una gran diferencia entre una y otra, podemos demostrar la incapacidad de la tarjeta, al ejecutar las instrucciones de control de flujo, de tipo condicional.

De esta última observación, podemos concluir que para realizar un aprovechamiento al máximo de la GPU, utilizando CG debemos evitar este tipo de instrucciones.

3.4 Simulación de Fuego sobre GPU/CPU y comparación de tiempos:

Descripción de la simulación:

Para hacer la simulación del fuego, hay muchas técnicas. Nosotros hemos elegido utilizar un sistema de partículas. Buscamos la comparación de una implantación en CPU con la de una en GPU, por ello no nos vamos a centrar en un modelo que nos sea fácilmente exportable a CG.

El movimiento de las partículas lo vamos a calcular empleando la formula de la aceleración de toda la vida. Cada partícula tendrá su vector de velocidad inicial, su posición inicial, y su aceleración. Para calcular su posición en función del tiempo tendremos la siguiente formula:

$$p_{actual} = p_{inicial} + v_{inicial} * t + a * t^2$$

Si queremos hacer un fuego haremos un sistema de partículas, en el que las partículas nacen con poca velocidad, y con una aceleración que las hace subir cada vez mas rápido según va envejeciendo su llama hasta convertirse en humo.

Teniendo el sistema de partículas de Fuego de este modo, nos sería muy fácil hacer otro tipo de fenómeno a partir de la misma implementación. Por ello hemos integrado a la vez, la simulación de una especie de fuente, en la cual solo cambian los valores de velocidad inicial y aceleración que se verá afectada por la gravedad.

Métodos utilizados para la implementación:

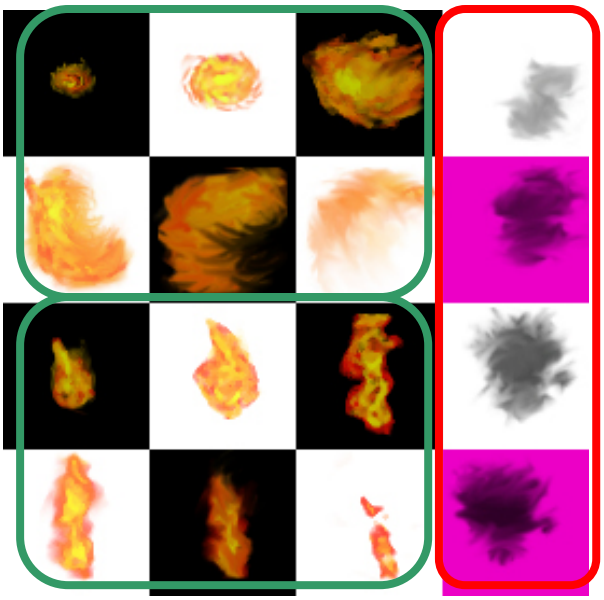
Como no queremos que todas las partículas nazcan a la vez y con la misma velocidad, vamos a introducir una variación de modo aleatorio dentro de un margen determinado. De este modo, el sistema de partículas nos da una sensación mucho más real.

Para mostrar las partículas y que parezca un fuego real, vamos a utilizar una secuencia de *sprites*. Con la secuencia intentaremos representar la vida de la partícula desde que nace siendo una llama pequeña, hasta que se convierte en humo que desaparece en el aire.

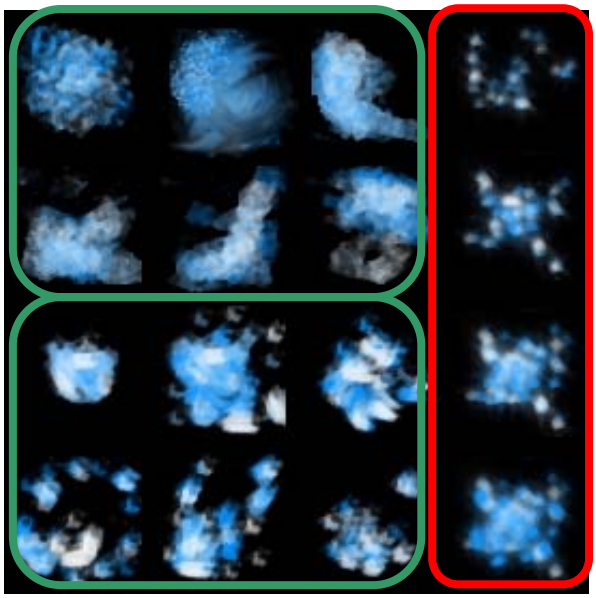
El sistema tendrá un tiempo de vida, una velocidad inicial, una aceleración, y una posición. A partir de estas se sacarán aleatoriamente las de las partículas aplicando un margen determinado. Las partículas una vez lleguen al final de su vida, renacerán en un nuevo origen. La posición inicial de las partículas se generará dentro de una circunferencia alrededor de la posición del sistema de generación.

Los *sprites* tendrán un nivel alfa muy bajo, para hacer un *alpha-blending* que mezcle todos los *sprites* de las partículas formando el fuego. El modo de pintar las partículas, es uno de los factores más importantes para conseguir más realismo en el fuego. Si pintamos todas las partículas con la misma secuencia de *sprites*, el fuego va ha tener demasiada homogeneidad. Por lo tanto vamos a aplicar los siguientes *trucos* para conseguir una variedad mayor en los *sprites*. Para empezar vamos a meter en la secuencia de sprites dos tramas

distintas para la parte de la llama. La parte de humo la dejaremos igual porque no nos compensa mucho hacer dos tramas distintas. Con esto conseguiremos un fuego mucho más variado. Las que he utilizado en la demo las he hecho a mano, no es mala, pero podría ser mejor, con lo que conseguiríamos más realismo. Veamos como quedan las secuencias de *sprites*, incluida la que vamos a utilizar para esa especie de fuente de agua:



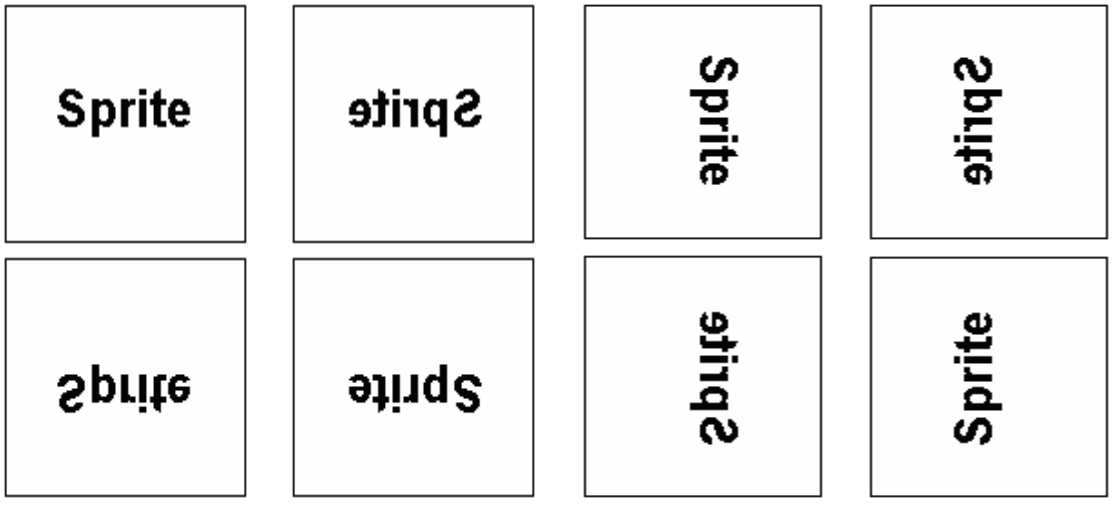
Matriz de 16 sprites utilizada para el Fuego



Matriz de 16 sprites utilizada para la Fuente

Los recuadros verdes engloban las distintas tramas que pueden tener las partículas al nacer. El orden de los sprites seria de la esquina izquierda superior a la derecha inferior. Para la parte del humo, es el recuadro rojo, y las *sprites* se aplican de abajo a arriba. Van desde un humo denso hasta desvanecerse en el *sprite* de arriba.

Ahora vamos a emplear otro truco más para conseguir mayor variedad de nuestro fuego y más realismo. El método consiste en utilizar coordenadas de textura que roten el *sprite*. Veamos como se haría para sacar de un *sprite* 8 diferentes:



Con esto hemos conseguido 16 secuencias diferentes, ya que tenemos dos tramas en la textura, y podemos sacar 8 de cada una haciendo los cambios necesarios en las coordenadas de textura.

Para visualizar los *sprites* en las tarjetas de NVIDIA han hecho la extensión GL_POINTS_SPRITE_NV, que nos permite cargar a un punto una textura y visualizarla por pantalla. Esta extensión funciona de un modo muy eficiente, y es realmente útil en los sistemas de partículas. Utilizando esta extensión solo es necesario decir donde queremos la partícula, y ya la tarjeta se encarga del resto. Nos evita tener que dar las coordenadas de textura y calcular el *billboarding* para que la textura del *sprite* en todo momento sea perpendicular a la cámara.

Pero a nosotros nos interesa tener partículas de un tamaño considerable, ya que un *sprite* puede representar una llama considerablemente grande. Además, la textura que vamos a utilizar contiene todos los *sprites,* y tenemos que aplicarle las coordenadas de textura que vamos a crear para conseguir más variedad. Por eso nos vamos a calcular el *billboarding* por nuestra cuenta, y vamos a dar las coordenadas de textura que hemos generado.

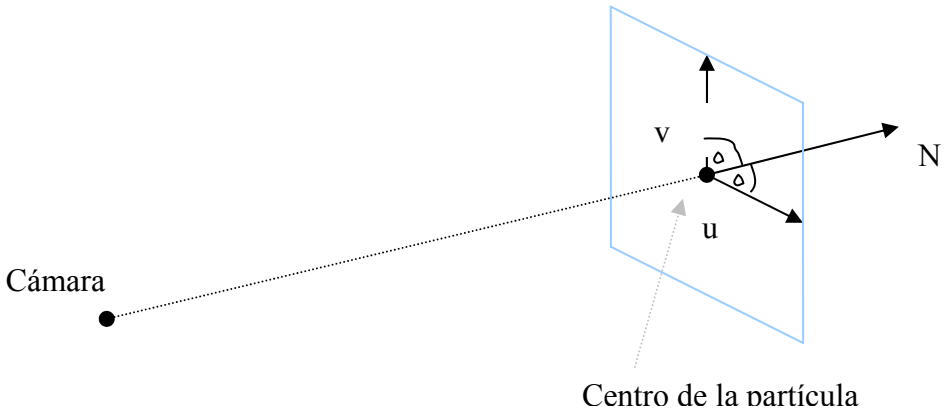
Veamos por separado y en profundidad las dos implementaciones.

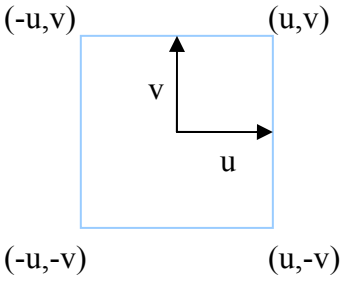
Implementación de la **CPU**:

En esta implementación antes de llamar a la función de render se llama a la función *update*. En esta, se realiza el cálculo principal de la posición de la partícula, y los puntos que definen el *billboard*.

La posición se calcula simplemente aplicando la formula del movimiento vista anteriormente.

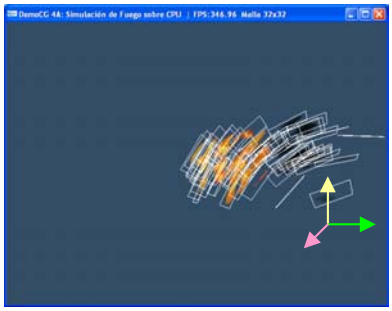
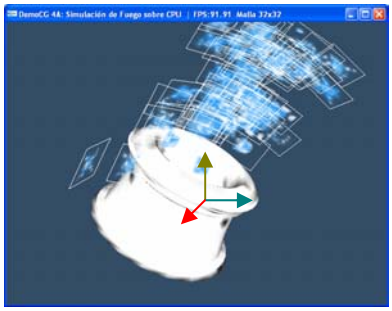
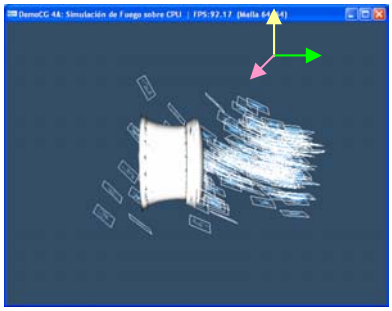
Para el *billboarding* vamos a sacar un vector dirección, de la posición de la cámara al centro de la partícula, calculado en el paso anterior. Una vez tenemos ese vector sacamos los otros dos vectores que forman un sistema ortogonal con este otro. Con unas simples multiplicaciones vectoriales es fácil sacarlo. Ahora normalizamos los vectores, y ya podemos calcular las posiciones de los cuatro puntos que definen el Quad que pinta la partícula. En la siguiente imagen se muestra el modo de hacerlo:





Una vez tenemos calculado los puntos, cuando se llame a la función de dibujado, simplemente hay que mandar pintar los GL_QUADS con dichos vértices.

Si parasemos la ejecución en un instante determinado, y rotasemos la cámara para ver las partículas desde otro ángulo, sin que estas actualizasen sus vértices, podriamos comprobar claramente como actua el *billboarding*. Veamos unas imágenes de ejemplo:

Los ejes representan donde estaría situada la camara en el momento en el que se hizo la pausa. Se ve claramente como todas las particulas encaran al punto donde estaría situada.

Implementación de la **GPU**:

En este caso, cuando se llame a la función *update* no vamos a realizar el cálculo de las posiciones de los vértices que definen el cuadrado de la partícula. Ni el cálculo de la posición del centro de la partícula. Estas tareas se las vamos a dejar al *Vertex Program* que se carga al utilizar la función de dibujado.

Para que el *Vertex Program* pueda hacer todos los cálculos, vamos a tener que pasarle todos los parámetros necesarios. Vamos a servirnos de las múltiples coordenadas de textura que se le pueden mandar a la tarjeta para pasar todos los parámetros. Básicamente realiza los mismos pasos que la implementación de antes, pero en la GPU con un *Vertex Program*.

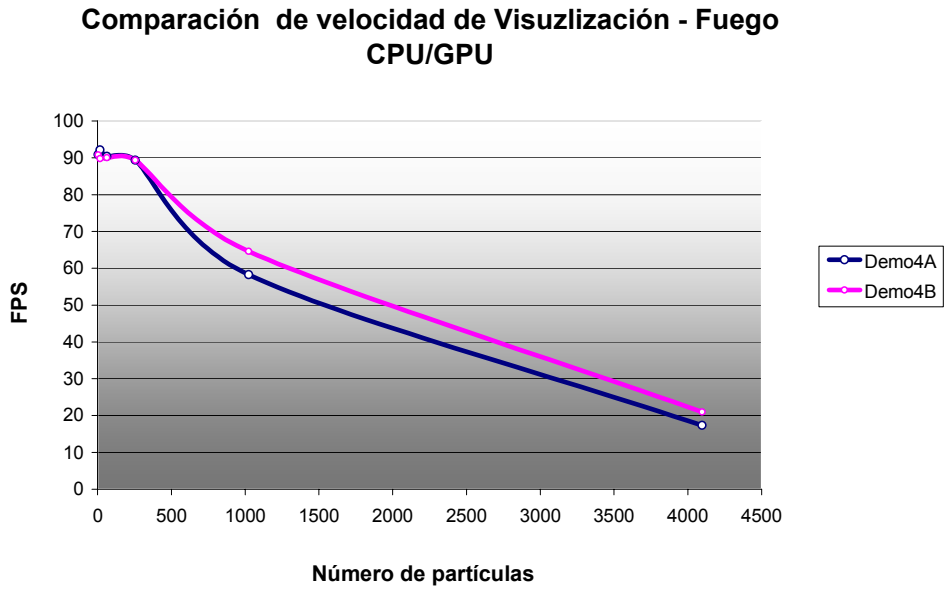
Veamos el código de CG que realiza esto:

```
void main( float4 inPos       : POSITION,
           float4 inColor     : COLOR,
           float4 inTexCoord0 : TEXCOORD0,
           float4 inTexCoord1 : TEXCOORD1,
           float3 inTexCoord2 : TEXCOORD2,
           float3 inTexCoord3 : TEXCOORD3,
           float2 inTexCoord4 : TEXCOORD4,
           out float4 outHpos      : POSITION,
           out float4 outColor     : COLOR,
           out float2 outTexCoord  : TEXCOORD0,
           uniform float4x4 modelViewProj,
```

```
            uniform float4     camara,
            uniform float4    vecX)
{
        // Sacamos el vector dirección de la cámara al centro de la particula
        float3 dir = inPos.xyz - camara.xyz;
        // Obtenemos dos vectores perpendiculares a este y los normalizamos
        float3 v = vecX.xyz;  // vecX.xyz = float3(100.0f, 0.0f, 0.0f)
        float3 u = cross(dir, v);
        v = cross(dir, u);
        u = normalize(u);
        u *= inTexCoord2;
        v = normalize(v);
        v *= inTexCoord3; // El TEXCOORD3 tiene las coordenadas del vértice del QUAD
        float4 particlePos = inPos;
        // El programa recibe la posición inicial , velocidad, aceleración, tiempo…
        // Con ellos aplica la formula del movimiento
        particlePos.xyz += (inTexCoord0.xyz * inTexCoord0.w) + (inTexCoord1.xyz *
                           inTexCoord0.w * inTexCoord0.w * inTexCoord1.w);
        // Desplazamos la posición a la esquina del QUAD que le corresponde
        particlePos.xyz += u + v;
        outHpos = mul(modelViewProj, particlePos);
        outColor = inColor;
        outTexCoord = inTexCoord4;
}
```

Comparación de tiempos:

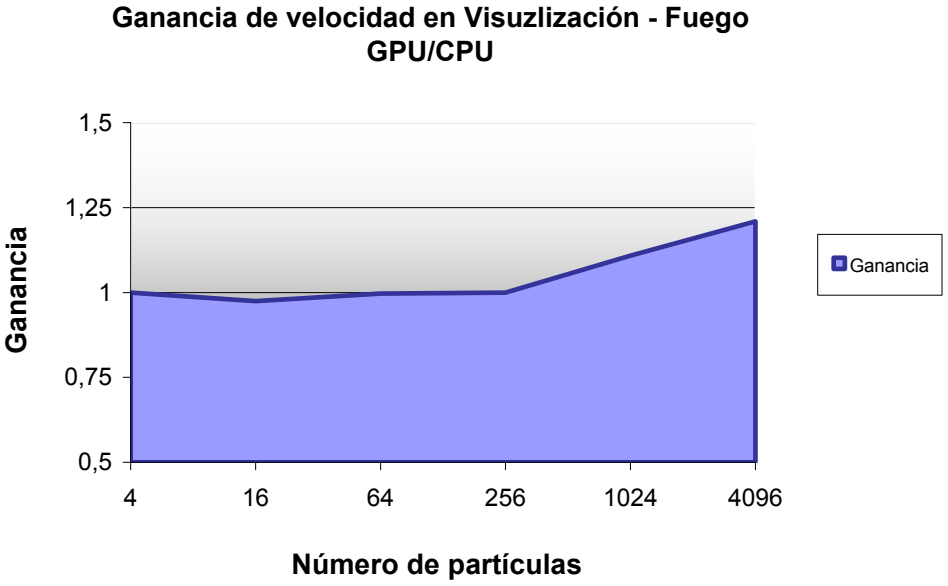
Si observamos el código del programa en CG comprobamos que realiza mucos cálculos vectoriales, productos vectoriales, sumas, normalizaciones… operaciones que la GPU va ha ejecutar muy rápido, con lo cual este programa va a sacar la eficiencia máxima. Sin embargo, hay un problema con esta implementación. Cuando calcula los valores de ***u*** y ***v***, lo está haciendo una vez para cada uno de los cuatro vértices que definen el QUAD. A diferencia de este caso, en la implementación de la CPU sacábamos dichos vectores una única vez y se los aplicábamos de golpe a los cuatro vértices. Aun así, esta implementación que utiliza la GPU es muy rápida, y supera la velocidad de la de la CPU como vamos a comprobar en la gráfica siguiente:



Si hacemos una comparativa obteniendo la ganancia en velocidad de este modo:

$$G = \frac{FPS_{implementación\_GPU}}{FPS_{implementación\_CPU}}$$

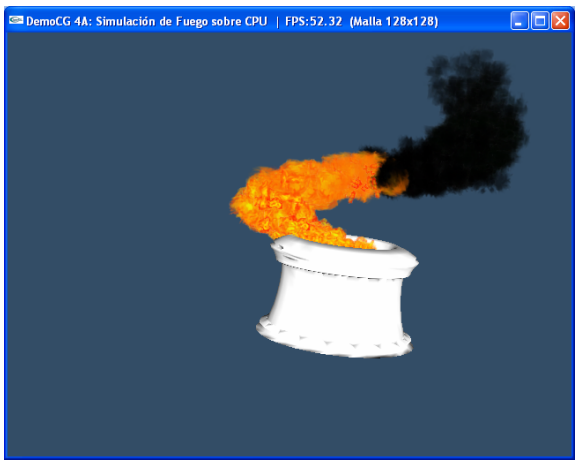
Obtenemos este resultado:



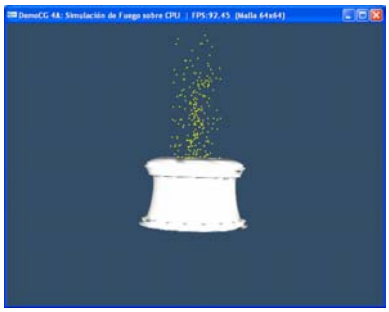
Está visto que esta implementación no es la más adecuada, pero aun así hemos conseguido una pequeña ganancia, ya que el programa explota la capacidad de la tarjeta.

Esta es la visualización de la demo para ambas implementaciones:

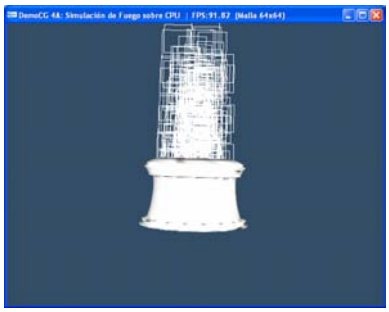
Fuego utilizando el mapa de 16 *sprites*



Fuego tras haber desplazado
el centro de generación en el plano XY

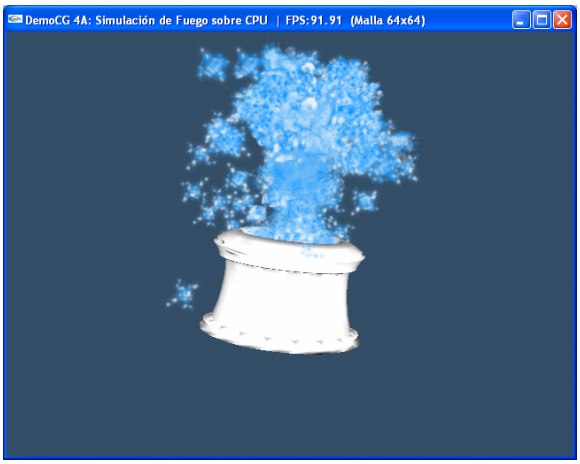


Utilizando
GL_POINTS

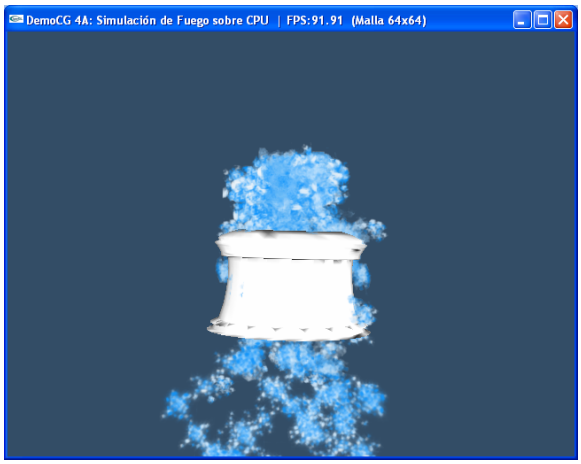


Utilizando
GL_QUADS

Utilizando
GL_POINTS_SPRITE_NV



Fuente utilizando el mapa de 16 *sprites*



Fuente tras haber desplazado
el centro de generación en el eje Z

4. Bibliografía:

1. The Cg Tutorial (NVIDIA Corporation, Addison-Wesley)

2. GPU Gems (NVIDIA Corporation, Addison-Wesley)

3. Páginas Web de CG utilizadas:

   http://www.cgshaders.org
   http://www.markmark.net
   http://developer.nvidia.com/page/cg_main.html
   http://www.gpgpu.org