SISTEMAS INFORMATICOS
CURSO 2003-2004

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

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

Autor: Raúl Muñoz López

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

ÍNDICE

1. INTRODUCCIÓN

1.1 OBJETIVOS

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 pretende hacer uso intensivo de la capacidad de cálculo de los procesadores gráficos de última generación (NVIDIA GeForce FX y ATI Radeon 9700). 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 implementará con OpenGL, haciendo uso de la librería GLUT. La programación de la GPU se realizará mediante las extensiones *ARB_vertex_program* y *ARB_fragment_program* de OpenGL. Para la generación de programas sobre GPU se aconseja el uso del lenguaje gráfico de alto nivel CG.

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.

Aunque lo puesto hasta ahora son los objetivos iniciales se ha de mencionar un cambio que se ha producido a lo largo del desarrollo: las extensiones finalmente utilizadas para la programación de la GPU, y cuyas clases han sido suministradas por el profesor Roberto Lario, han sido finalmente las extensiones GL_VERTEX_PROGRAM_NV y GL_FRAGMENT_PROGRAM_NV, las cuales a su vez son propietarias de nvidia y por lo cual las únicas tarjetas válidas son las NVIDIA GeForce FX.

1.2 DESCRIPCION GENERAL

Las fases de las que consta el módulo son las siguientes:

- El contenedor cúbico se supone compuesto por 5 planos texturizados. Como recinto exterior al contenedor se supone una habitación con 5 planos texturizados.
- Simulación (inicialmente software) del movimiento de la superficie del fluido, la simulación tanto a nivel de software como a nivel de hardware (CG) mediante la Ecuación de Ondas 2D está diseñado para realizarse en el módulo 2 de este mismo proyecto.
- Cálculo de iluminación teniendo en cuenta la Reflexión, Refracción, Efecto Fresnel y Dispersión Cromática.
- Tras implementar la demo con recipiente cúbico, se procederá a implementar las versiones con recipiente semiesférico, semicilíndrico vertical y horizontal.

La implementación de este módulo se pensó inicialmente para ser desarrollado por las técnicas de *environment mapping*, pero 2 fueron los motivos principales por los cuales se decidió no utilizarse:

1) Esta técnica se basa principalmente en la idea de que el cubo sobre el que se realiza el environment mapping se sitúa sobre el infinito, es decir, se suele utilizar cuando sobre lo que se hace el environment mapping son las imágenes de fondo (cielo y horizonte), por lo que para realizarlo no se tiene en cuenta la posición del punto a calcular (puesto que la posición de un punto sobre el infinito es inapreciable), y lo cual produciría sobre la demo de este módulo una distorsión de la realidad, puesto que la altura del agua es variable y modificable, el color de un determinado punto del agua no puede ser el mismo si está sobre la parte mas alta de la piscina que si está sobre el fondo, ya que no se refractaria ni se reflejaría lo mismo.
2) El segundo motivo de esta decisión es que el environment mapping sobre la reflexión, y a su vez el efecto fresnel puesto que usa a la reflexión, tiene en cuenta solo las imágenes sobre un único cubo, pero sobre esta escena al hacer la reflexión sobre el agua y el nivel del agua no tiene porqué estar simulando una piscina llena, sino que puede estar a un nivel medio, o incluso a ras de suelo, es necesario calcular la reflexión no solamente sobre el cubo de la habitación sino que también quedaría reflejado parte de las paredes de la piscina, lo cual son 2 cubos.

3*) Un tercer motivo menos importante podríamos considerar que es el hecho de que environment mapping necesita cubos completos de 6 lados, y tanto nuestra piscina como la habitación poseen solamente 5 lados, es decir son cubos incompletos, aunque esto es solamente secundario puesto que al compilador de CG se le podría pasar un sexto lado negro.

El cambio fundamental de esta decisión provoca que en vez de hacer uso de 2 funciones geométricas propias de CG que son reflect(I,N) y refract(I,N,eta) las cuales son aceleradas por hardware, tenga que ser necesario implementar en CG cada una de las 2 funciones supliendo así los fallos que se cometerían al usarlos directamente, pero como consecuencia el tamaño de los programas aumentan y el rendimiento y eficiencia de los programas en CG decaen.

Por último comentar que aunque las técnicas de environment mapping son bastante eficientes solo hubieran podido utilizarse sobre la piscina cúbica, no así sobre los otros tipos de piscina solicitados en este módulo.

1.3 HERRAMIENTAS UTILIZADAS

Las herramientas, librerías y programas usados para la realización de este proyecto han sido el entorno de programación de Microsoft Visual C++ 6.0 apoyándome en las librerías de OpenGL, devil y glut junto con la programación sobre CG, más concretamente:

- Microsoft Visual C++ 6.0, es el entorno de programación usado para programar sobre C para la realización del proyecto, es uno de los prerrequisitos exigidos por el profesor. Fue escogido el lenguaje de programación C++ por una mayor facilidad de uso de la librería de OpenGL y de las glut, mientras que el entorno gráfico se escogió pensando en una mayor integración de las parte de CG con el código de C++, puesto que el Microsoft Visual C++ permite integrar los archivos de CG dentro de su compilador, compilando todo el código de C++ y de CG a la vez, produciendo tanto los .obj generados a partir de los .cpp y .h del código de C++, como los .vp y .fp generados a partir de los .cg del código de CG añadiendo las opciones de compilación adecuadas a los archivos de CG.
- OpenGL, es la librería escogida para hacer el marco de trabajo sobre el que se apoyará CG, ya que se necesitaba una librería de dibujo en 3D puesto que aunque CG sea capaz de modificar vértices y píxeles no es capaz de crearlos. Para la realización del este proyecto se han barajado tanto la opción de escoger OpenGL como la opción de escoger DirectX, ya que CG soporta las 2 librerías, pero por facilidad de uso, un mayor conocimiento de OpenGL y porque la mayor parte de la documentación de CG viene implementada sobre OpenGL se escogió finalmente esta opción.
- Devil, esta es la librería encargada de la carga de imágenes para posteriormente poder manejarlas con OpenGL. En los prerrequisitos mostrados por el profesor del proyecto se exigían imágenes guardadas por archivos .tga por lo que se eligió esta librería por ser capaz de abrir casi todo tipo de formatos de imágenes, por su eficiencia y porque a su vez está diseñada para ser usada con OpenGL.
- Glut, esta es la librería en la que se apoya C++ para crear y manejar las ventanas de Windows, para el renderizado, para la creación de menús o para otras funciones auxiliares como por ejemplo la detección y localización de eventos de ratón o teclado. Esta librería ha sido escogida por ser un prerrequisito del proyecto y por la falta de una buena herramienta de Microsoft Visual C++ para la creación y gestión de ventanas y eventos.
- CG, es el lenguaje de programación en el cual se basa y está centrado todo el proyecto (explicado en un apartado posterior).
- Adobe Photoshop 7.0 para la creación de las texturas.

1.4 REQUISITOS Y RESTRICCIONES

Como CG es un lenguaje de programación para la programación interna de tarjetas gráficas parece lógico pensar que para que el ejecutable de este proyecto funcione correctamente hace falta tener una tarjeta programable instalada y funcionando, en especial y debido al uso de unas extensiones propietarias de nvidia (GL_VERTEX_PROGRAM_NV y GL_FRAGMENT_PROGRAM_NV) en las clases del proyecto que interrelacionan la parte de CG con la parte de C++ y proporcionadas por el profesor supervisor del proyecto es también necesario que dicha tarjeta sea una GeForce, y debido a que dentro del código de CG se usan procedimientos y funciones avanzadas es necesario también que dicha tarjeta soporte las últimas versiones de CG, es decir vp30 y fp30, por lo que dentro de los modelos de la GeForce tendrá que ser un modelo igual o superior a la GeForce 5 Fx 5200, que es la primera tarjeta en soportar dichas versiones de CG.

Por último en cuestión de hardware recomendar que cuanto mayor sea el modelo de la tarjeta mucho mejor irá, puesto que en este proyecto se hace un uso intensivo de la tarjeta gráfica y que con la GeForce 5 Fx 5200 el refresco de pantalla para la realización de determinados efectos gráficos sobre el agua se ralentiza bastante, mientras que pruebas con la GeForce 5 Fx 5900 la velocidad de refresco aumenta en más de un 400% dependiendo del efecto gráfico y de la forma de la piscina utilizada.

En cuestión de software para hacer funcionar el ejecutable solo es necesario que estén en la misma carpeta que el ejecutable todos los ficheros .fp y .vp, el fichero de configuración *config.rml*, las librerías devil.dll, glut32.dll, ilu.dll y ilut.dll, y una carpeta llamada *Texturas* con todas las texturas usadas en el proyecto, todo lo cual viene proporcionado junto al ejecutable.

1.5 BREVE EXPLICACIÓN DE CG

CG es un lenguaje que hace posible el control de sombras, apariencia, y movimientos de objetos usando tarjetas gráficas programables. CG (C for grafic) es un lenguaje derivado del lenguaje de C, con notación similar pero especializado en programación de hardware, con procedimientos y funciones acelerados por hardware y con posibilidad de procesamiento en paralelo, acelerando de esta forma los cálculos de los puntos en 3D, vectores y matrices.

CG fue desarrollado en colaboración con Microsoft Corporation y está diseñado para ser usado con librerías gráficas como son OpenGL API y Microsoft High-Level Shading (HLSL) para DirectX 9.0, y actualmente la última versión tanto para el vertex program como para el fragment program (las 2 partes programables con CG) el la 3.0 (vp30 y fp30).

Los programas en CG pueden manejar vértices o fragmentos (se puede relacionar con píxeles) que se procesan cuando se renderiza la imagen, básicamente es capaz de transformar los vértices o los píxeles casi al antojo del programador, esto nos empieza a dar una idea de las 2 formas o modelos de programación de CG, los vertex program y los fragment program.

Básicamente un programa en CG puede ser activado y desactivado independientemente, con la única restricción de que solo puede haber un único vertex program junto con un único fragment program activados a la vez, por lo que se puede renderizar un objeto diseñado en OpenGL y ser manipulado por un vertex program y un fragment program y el siguiente objeto por otro vertex program o fragment program totalmente distintos.

Cada uno de los 2 tipos de programas en CG afectan al pipeline gráfico en un momento determinado, el pipeline gráfico está formado por 4 partes:

- ✓ Vertex transformación:

  Todos los vértices tienen una posición, pero generalmente también tienen otra serie de propiedades como por ejemplo el color, un color secundario, una posible lista de coordenadas de textura y una normal que indica hacia qué posición está mirando la cara del objeto y que se suele utilizar para el cálculo de la iluminación.

  La vertex transformación es la primera de todas las transformaciones que realiza el pipeline gráfico, estas operaciones incluyen transformación de la posición del vértice, ya sean trasformaciones afines o para la adaptación del vértice a la posición de la cámara, generación de coordenadas de textura para el texturizado y determinar el color del vértice por la iluminación.

- ✓ Primitivas de ensamblaje y rasterización:

  Los vértices se quedan ensamblados los unos a los otros dependiendo de ciertas primitivas geométricas, después son recolocados en un espacio 3D definido por *view frustrum* (o región del espacio visible en 3D), una vez hecho esto alguno de los polígonos que se sobresalen de este espacio han de ser recortados.

  Los vértices que pasan este corte han de ser rasterizados, el resultado de esta rasterización es un conjunto de localizaciones de píxeles o un conjunto de fragmentos.

- ✓ Interpolación, texturizado y coloreado:

  Una vez tenemos ese conjunto de cero o más fragmentos, la interpolación, texturizado y coloreado interpola los parámetros de los fragmentos dados en los vértices para calcular el color final y la profundidad de cada fragmento.

- ✓ Operaciones de rasterizado:

  Estas operaciones son propias de OpenGL o de DirectX como pueden ser el scissor test, alpha test, stencil test, depth test o el blending.

El vertex program empieza cuando son cargados los atributos de los vértices, tiene como objetivo principal la manipulación de los datos vértice por vértice, la salida de un vertex program será como mínimo el color del vértice y su posición, pudiendo cambiar y sacar también como salida otros datos como las coordenadas de textura, colores secundarios o las normales.

El fragment program empieza una vez son creados los fragmentos, su única salida es el color final que mostrará por fragmento, al igual que el vertex program se pueden realizar con él casi todo tipo de operaciones matemáticas, pero a diferencia de este tiene operaciones de texturizado para el cálculo del color a partir de las coordenadas de textura y de las texturas.

2. DISEÑO E IMPLEMENTACIÓN

2.1 PLANIFICACIÓN TEMPORAL

El proyecto al completo fue planificado por el profesor supervisor del proyecto de forma que se dividiese en tres módulos completamente aislados y separados, cada uno de dichos módulos asignado a cada uno de los tres alumnos para dicha realización, y con el fin de que en junio, si las tres partes iban correctamente y se completaban con tiempo suficiente se pudiese intentar juntar algunos módulos.

Con respecto al módulo1 a parte de algunos prerrequisitos dictados por el profesor supervisor del proyecto como por ejemplo algunas de las herramientas a utilizar o la forma y extensiones de las imágenes se dio libertad para hacer el proyecto como se quisiera y sin plazos de entrega, por lo cual y tras una breve planificación del diseño y del tiempo se decidió hacer la realización de este modulo en tres iteraciones mas una cuarta iteración en caso de juntar los distintos módulos:

1. La primera iteración consta de la lectura de los distintos manuales, conseguir familiarizarse con las herramientas a utilizar, y la realización de un framework en Microsoft Visual C++ en el que se hace uso de las librerías Glut, OpenGL y Devil, junto con el diseño de ciertas clases necesarias para la programación de escenas en 3D, y a partir de el cual montar toda la estructura del programa en C++. La finalización de esta primera iteración coincidió con la finalización de las navidades.

2. La segunda iteración es la creación de la escena completa en 3D haciendo uso fundamentalmente de OpenGL y de patrones de diseño como la Factoría Abstracta, completando así toda la programación necesaria de C++ para este proyecto y quedando solamente la parte de CG propiamente dicha. El plazo de esta entrega fue finales de febrero o final del primer cuatrimestre, momento en el cual ya existía algo sólido que poder mostrar al profesor supervisor, el cual mostró su conformidad con lo realizado, quedando libre la programación sobre CG.

3. La tercera iteración ha sido la programación de los programas de CG y la unión entre la parte de C++ y la parte de CG, esta iteración es el bloque fundamental en el que está basado el proyecto, y el cual, aunque teniendo menos código ha sido el que más tiempo ha sido necesario para su realización, dicha entrega ha durado hasta finales de mayo, finalizando de esta forma el módulo1 del proyecto.

4. Una posible cuarta iteración hubiera sido la unión de este módulo con el segundo módulo, pero por problemas ajenos a la parte correspondiente al modulo1 no ha podido realizarse.

2.2 PRIMERA ITERACIÓN

2.2.1 MODELO DEL DOMINIO

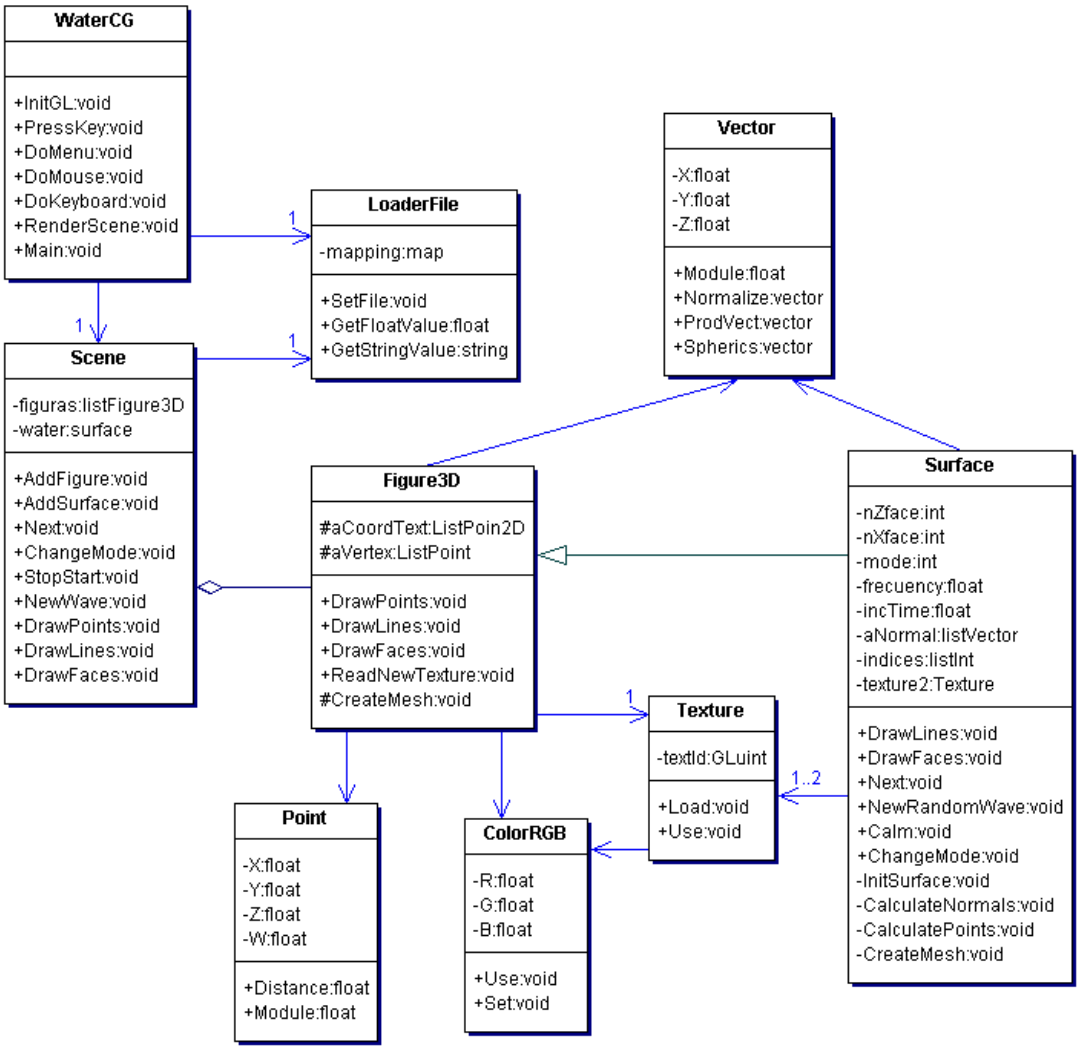
En esta primera iteración se va a desarrollar un framework básico en C++ el cual sea capaz de representar bajo la librería de OpenGL cualquier escena en 3D, para lo cual, a parte de las clases básicas que han de aparecer para poder representar un objeto en 3D aparecen otras funcionalidades extras, que, aunque no fuesen directamente necesarias para la representación de una escena en 3D, nos orientan hacia las necesidades que posteriormente tendrá el módulo y a su vez aumentando la funcionalidad de dicho framework.

Una de estas funcionalidades a desarrollar es la posibilidad de cargar mediante un fichero de configuración ciertos parámetros pudiendo hacer que sean variables (aunque no en tiempo de ejecución), como por ejemplo las dimensiones de la escena, los parámetros de el agua o las texturas a cargar, debido a esta funcionalidad aparece sobre el framework básico una nueva clase llamada LoaderFile la cual es la encargada del almacenamiento de todos los parámetros almacenados en el fichero de configuración, y a través de la cual otros objetos pueden consultar los parámetros que necesiten en cada momento. Esta mejora producen en el módulo una dinamización de la escena, pudiendo crear escenas distintas cambiando solamente un parámetro en el fichero de configuración.

La otra funcionalidad básica implementada en este framework es la carga de casi cualquier tipo de formato de imágenes para ser usados como texturas, aunque en la práctica se han usado solamente ficheros .tga, se ofrece la posibilidad de usar casi cualquier otro tipo de formato, siempre y cuando la imagen que contengan tenga la forma necesaria para que se dibuje correctamente (sobre todo la forma de las imágenes de las piscinas), puesto que aunque el fichero sería válido, el resultado obtenido sobre la imagen en 3D no sería de buena calidad. Esta nueva funcionalidad se ha implementado con la clase Texture, la cual hace uso de la librería Devil para la carga de las imágenes y cuya clase es usada por todo aquel objeto para el que se necesite una textura asociada.

2.2.2 DISEÑO

El diseño de esta primera iteración consta de las clases necesarias para la representación de una escena en 3D, las clases para la implementación de las funcionalidades de la carga de parámetros y de la carga de imágenes para las texturas y de las relaciones entre ellas, el diagrama UML que queda de estas clases es el siguiente:



2.2.3 IMPLEMENTACIÓN

Lo primero que habría que explicar de la implementación de esta primera parte sería la clase principal, en este caso llamada waterCG en el cual se hace uso de las librerías glut, las cuales han sido utilizadas entre otras cosas para la creación y manejo de las ventanas de Windows y de los eventos.

En esta clase podemos encontrar los siguiente grupos de funciones:

- Funciones para el uso de openGL, como la inicialización de OpenGL, los eventos de cambiar el tamaño de la ventana o la colocación y orientación de la cámara.

- La función propia del renderizado de la imagen, en la cual aparte de la comprobación de ciertas variables, hace el cálculo de los frames por segundo y llama a las funciones de renderizado que sean necesarias (renderShaded. RenderWired, renderVertex).

- Las funciones de creación de los menús y capturas de los eventos de ratón y de teclado, cada una de las cuales llamará a la función necesaria para cada evento.

- Funciones necesarias para mostrar la ayuda, puesto que la escena es una escena en 3D, si se desea una ayuda interactiva en 2D es necesario poner la escena de una forma ortográfica, cargar la matriz identidad, escribir la ayuda con ayuda de la librería glut y volver a restablecer la proyección perspectiva:

```
setOrthographicProjection();
glPushMatrix();
glLoadIdentity();
renderBitmapString(5,10,(void *)font,"Arrows - move");
renderBitmapString(5,25,(void *)font,"Re pag - zoomIn");
renderBitmapString(5,40,(void *)font,"Av pag - zoomOut");
(.....)
glPopMatrix();
resetPerspectiveProjection();
```

- La función principal (main), la cual registra las funciones correspondientes a cada cosa para que la librería glut sepa cómo manejarlas:

```
// Creación de la ventana
unsigned int mode = GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH;
glutInit( &argc, argv );
glutInitWindowSize( windowWidth, windowHeight );
glutInitWindowPosition( 200, 200 );
glutInitDisplayMode( mode );
int mainWinID = glutCreateWindow( title );

//para cg;
initExtensions();
```

```
//para el uso de la librería Devil
ilInit();
ilEnable(IL_CONV_PAL);
ilutEnable(ILUT_OPENGL_CONV);
ilutRenderer(ILUT_OPENGL);

// cargamos el fichero de configuración y creamos los objetos de la escena
Load();
// Inicializamos los atributos de renderizado y las luces
initGL();
// Creamos el menu de la ventana
createMenu();

// Definimos los oyentes de los eventos
glutIgnoreKeyRepeat(1);          //ignora el mantener una tecla pulsada
glutSpecialFunc(pressKey);       //registramos evento de tecla especial
glutSpecialUpFunc(releaseKey);   //registramos evento de levantar tecla especial
glutKeyboardFunc( doKeyboard );  //registramos función de eventos de teclado

glutDisplayFunc ( renderScene ); //registramos función de render
glutIdleFunc    ( idleHandler ); //registramos función de render
glutReshapeFunc ( changeSize );  //registramos función de eventos de ratón (hay
más de una)
glutMouseFunc   ( doMouse );     //registramos función de eventos de ratón
glutMotionFunc  ( doMoveMouse ); //registramos función de eventos de ratón
time=glutGet(GLUT_ELAPSED_TIME);

glutMainLoop();

//se ha terminado la ejecución
delete scene;
delete loader;
delete factory;
```

Otra de las clases importantes en esta primera iteración es la clase Figure3D, de la cual heredarán todos los objetos que se pintarán sobre la escena. Esta clase almacena los vértices, las normales y las coordenadas de textura del objeto que representa, y posee procedimientos para su pintado de vértices, malla o caras con sus respectivas texturas.

Un detalle importante a tener en cuenta de esta clase es la forma que tiene de dibujarse con openGL, típicamente sobre OpenGL se usan los comando glBegin y glEnd para encapsular cada cara, guardando una tabla de vértices, una tabla de normales, una tabla de coordenadas de textura y una tabla de caras con índices de cada cara a cada uno de sus elementos en cada una de las otra 3 caras, pero esta forma de dibujo de OpenGL es bastante ineficiente,

por lo que se ha optado por acelerar el dibujado de la imagen haciendo uso de los vertexArrays, normalArrays y textureCoordArrays, los cuales almacenan una sola lista de valores y los recorre secuencialmente, con esto se consigue una mejora considerable puesto que en vez de pasar a la tarjeta gráfica paquetes de 4 vértices, le pasamos un único paquete con todos los vértices del objeto, además para una mayor eficiencia se han diseñado dichas listas de vértices, normales y coordenadas de textura de forma que se puedan recorrer con el comando GL_TRIANGLE_STRIP el cual va dibujando un triángulo por cada vértice nuevo, haciendo uso de los dos últimos vértices, minimizando así la cantidad de vértices que le tenemos que pasar a la tarjeta gráfica (pasando un solo vértice con una única normal y una única coordenada de textura por cada triángulo más los dos vértices iniciales en vez de pasarle tres puntos con sus datos asociados por cada triángulo), haciendo uso de triángulos puesto que las tarjetas gráficas actuales están diseñadas para la aceleración de dibujos con triángulos:

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);

glVertexPointer(4, GL_FLOAT, 0, aVertex);
glNormalPointer(GL_FLOAT, 0, aNormal);
glTexCoordPointer(2, GL_FLOAT, 0, aCoordTex);

glPolygonMode( GL_FRONT, GL_FILL );
glDrawElements(GL_TRIANGLE_STRIP, numIndex, GL_UNSIGNED_INT, indices);

glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_TEXTURE_COORD_ARRAY);
```

La clase Surface es la que representa al agua, y es una de las más importantes a lo largo del proyecto, puesto que será sobre esta clase sobre la que se enlacen los programas de CG. Esta clase hereda de Figure3D y añade ciertas funcionalidades para el manejo de las olas del agua.

La clase Scene es la encargada de manejar todos los objetos en 3D, su atributo más importante es una lista de punteros a objetos de la clase Figure3D la cual es usada para almacenar cada uno de los objetos de las escena y para dibujarlos. Esta clase además contiene un puntero a un objeto de tipo surface, aunque dicho objeto hereda también de la clase Figure3D y también podría ser incluido en la lista de objetos, se ha preferido mantener aparte con un puntero por eficiencia, ya que debido a la gran funcionalidad de este objeto es referenciado y usado cada vez que se quiere modificar cualquier atributo sobre el agua, las olas o los efectos del agua, evitando así una búsqueda innecesaria sobre la lista.

La clase Texture es la encargada de la carga de ficheros de imágenes, siendo capaz de cargar casi todo tipo de formatos y extensiones de imágenes gracias al uso de las librerías Devil, y guardando dichas texturas como imágenes de OpenGL, encapsulando todo el código del manejo de dichas texturas sobre OpenGL:

```
ilGenImages(1, &ImageName);
ilBindImage(ImageName);
if (!ilLoadImage(const_cast<char *>(nameFileAux))) {
        ilDeleteImages(1, &ImageName);
        return;
}
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

TexID = ilutGLBindTexImage();
ilDeleteImages(1, &ImageName);
```

Por último están las clases Vector, Point y ColorRGB las cuales son necesarias para la creación de una escena en 3D.

2.3 SEGUNDA ITERACIÓN

2.3.1 MODELO DEL DOMINIO

En la segunda iteración se va a completar el diseño y el código del framework creado en la primera iteración, completando todo el código de C++ necesario para el posterior desarrollo de las parte de CG, es decir, al diseño e implementación de las clases de los objetos que se van a mostrar, los cuales inicialmente serían la habitación, la piscina cúbica y el césped.

Teniendo en cuenta los requisitos finales del proyecto y anticipándonos al último apartado en el que nos pedían los efectos sobre la piscina cilíndrica vertical y horizontal y semiesférica se han diseñado e implementado también sobre esta iteración la creación de los objetos necesarios para mostrar dichas escenas.

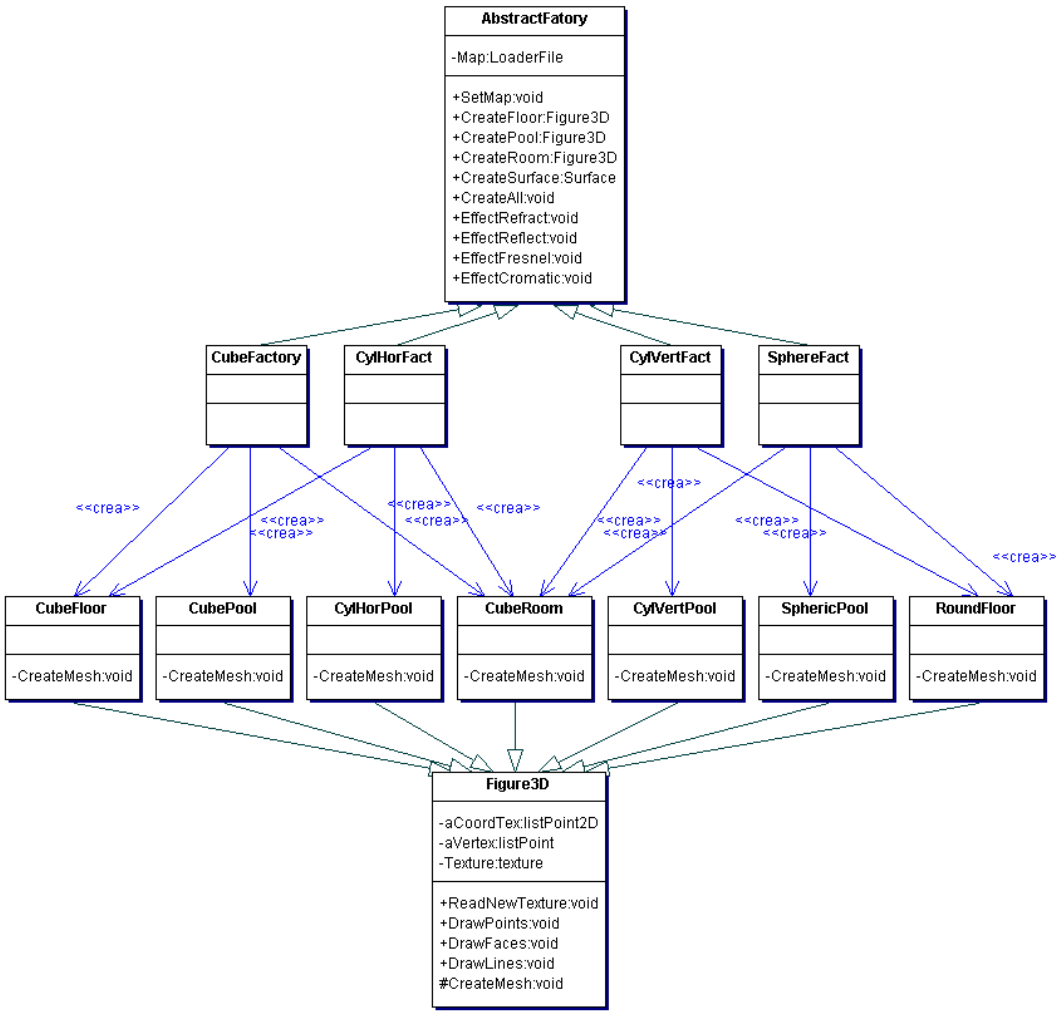
Aunque se ha dejado para la tercera y última iteración tanto la programación de CG como la interacción entre el código producido en C++ y los programas implementados con CG, en esta iteración se han creado algunas de las clases que van a interactuar como intermedias entre un código y otro, y que han aparecido gracias al diseño de esta iteración.

En esta iteración también se han tenido que crear las texturas para cada una de las piscinas, más las texturas para la habitación y el césped, las cuales han sido creadas con el Adobe Photoshop 7.0. Cada una de estas texturas han sido creadas con una forma característica para cada objeto, de forma que la textura envuelva completamente al objeto, por ejemplo, para la semiesférica la forma de la textura es circular mientras que para la piscina cúbica tiene forma de cubo cuyos lados se abren para formar una cruz en 2D.

2.3.2 DISEÑO

Pensando en la posibilidad de la extensión de este proyecto a otros tipos de piscinas (como por ejemplo una piscina cónica), y con la intención de que los tipos de piscinas puedan ser intercambiables en tiempo de ejecución se ha diseñado esta parte con un patrón de diseño llamado Factoría Abstracta, cuyo objetivo es independizar la manera en que se crean, componen y representan los objetos, encapsulando el conocimiento acerca de qué clase concreta se instancia y de cómo se crean o ensamblan las instancias de los objetos.

Gracias es este patrón y usando a su vez la herencia sobre la clase objeto3D, obtenemos el siguiente diagrama UML, el cual habría que juntar con el diagrama de la primera iteración:



En este diagrama se puede ver como cada una de las factorías concretas, las cuales heredan de la factoría abstracta, crean los objetos con la forma necesaria dependiendo de cómo sea la escena solicitada, por ejemplo, mientras la escena de la piscina cúbica hace uso de las clases CubeRoom, CubeFloor y CubePool, la escena de la piscina semiesférica usa las clases CubeRoom, RoundFloor y SphericPool.

Aprovechando la idea de la factoría abstracta y teniendo en cuenta que para cada tipo de piscina, y para cada tipo de agua tendremos un programa diferente de CG, se ha aprovechado este patrón para la interacción entre C++ y CG, incluyendo en la factoría abstracta una función para cada tipo de efecto solicitado en el módulo del proyecto y que deberán de ser implementados por cada una de las factorías concretas.

2.3.3 IMPLEMENTACIÓN

La factoría abstracta podríamos considerar que es la clase más importante de esta segunda iteración, puesto que es el eje a partir del cual se ha diseñado el patrón factoría y es el padre de cada una de las factorías concretas. Esta clase podría haberse diseñado como una interfaz de C++, pero puesto que hay partes comunes en todas las factorías hijas se ha decidido implementarla como una clase abstracta (no pueden existir objetos de esta clase).

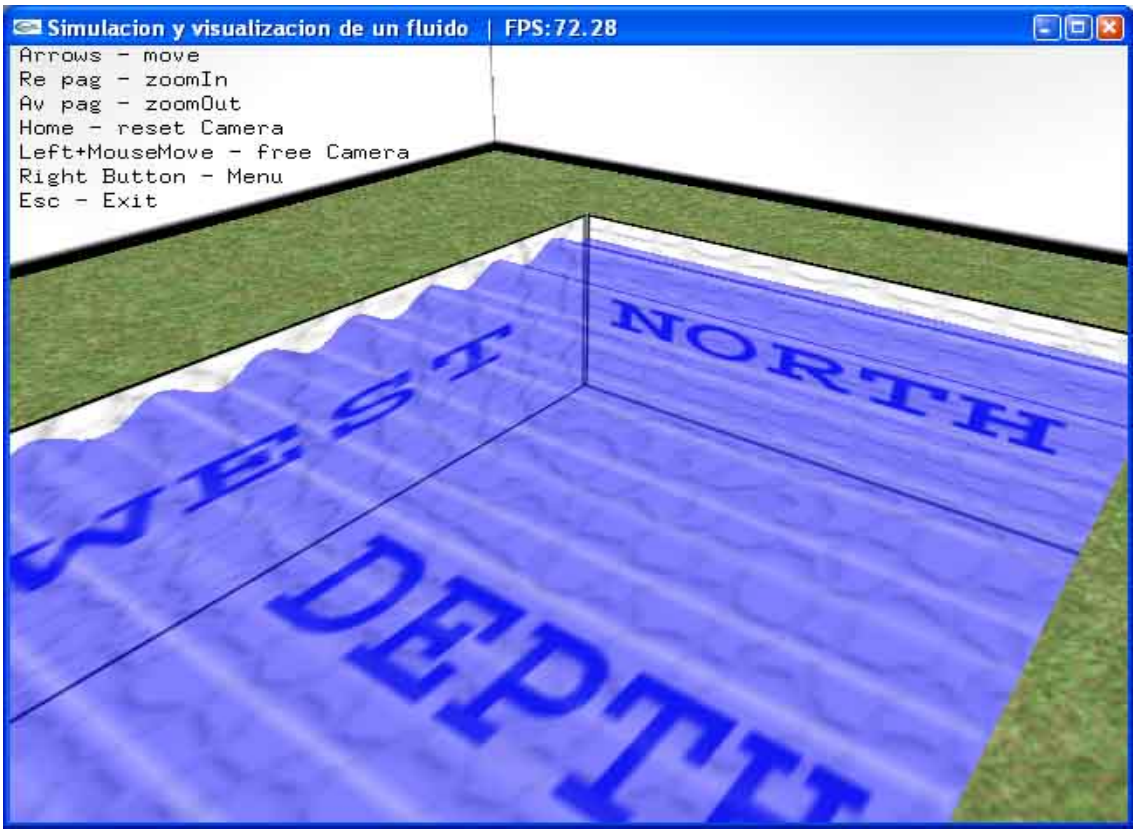
La factoría abstracta, al igual que todas las factorías concretas heredadas de ella tienen dos funciones fundamentales, la primera es la creación de la escena, seleccionando los objetos que se han de crear, y la segunda es la selección del tipo de efecto que cuando esté implementado el código de CG queramos mostrar:

```
void SetMap(const LoaderFile &mapp);          //carga el fichero de configuración
virtual Figure3D* CreateFloor() = 0;          //crea el suelo de la escena
virtual Figure3D* CreateRoom();               //crea la habitación
virtual Figure3D* CreatePool() = 0;           //crea la piscina
virtual Surface* CreateSurface() = 0;         //crea la malla del agua
virtual void CreateAll(Scene** scene);        //compone la escena completa
virtual void EffectRefract(Scene* scene){};   //selecciona el efecto de refracción
virtual void EffectReflect(Scene* scene){};   //selecciona el efecto de reflexión
virtual void EffectFresnel(Scene* scene){};   //selecciona el efecto Fresnell
virtual void EffectCromatic(Scene* scene){};//selecciona el efecto cromático
```

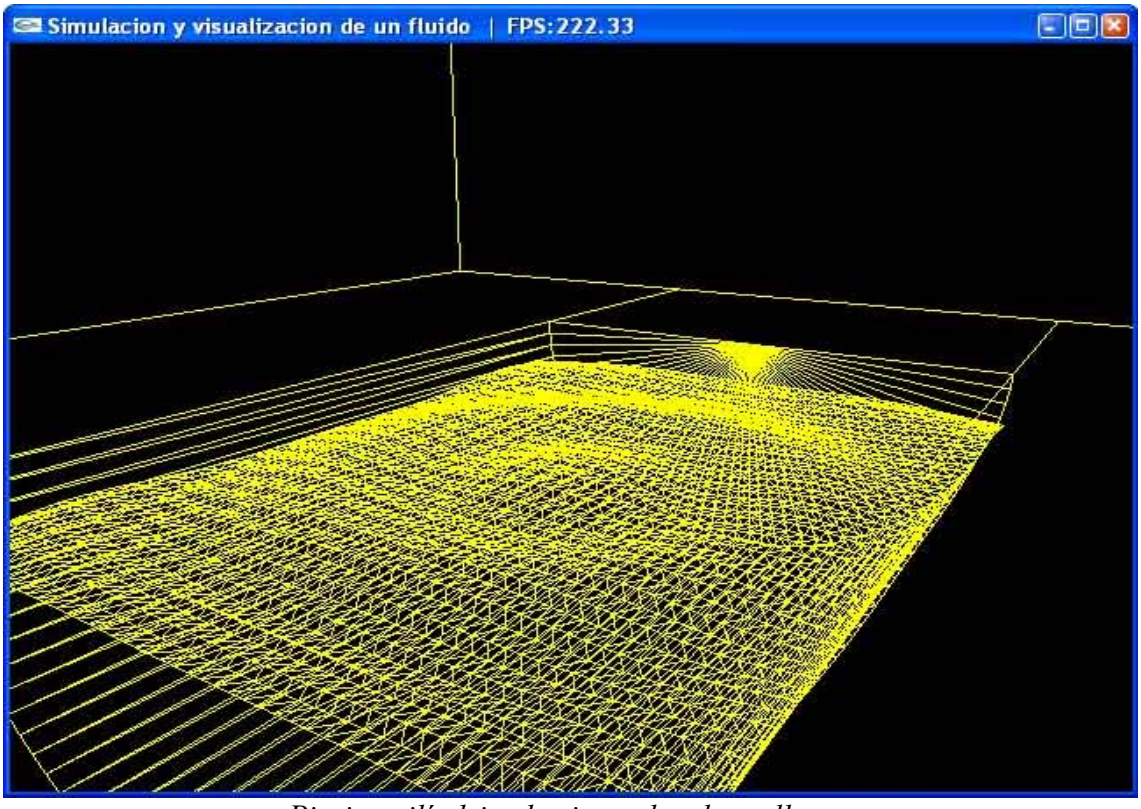
Puesto que tenemos cuatro tipos de escena es normal que tendremos también cuatro tipos de factorías concretas, las cuales son CubeFactory, CylHorFactory, CylVertFactory y SphereFactory, dichas factorías implementan cada uno de los procedimientos de la factoría abstracta.

Además de las factorías en esta iteración se han implementado cada uno de los objetos que se muestran es la escena (excepto la superficie del agua que ya se tenía de la primera iteración), es decir, los 4 tipos de piscina, la habitación y la malla que representa el césped para piscinas rectangulares y circulares. Como ya se comentó en la primera iteración se han usado los vertexArrays, normalArrays y coordTexArrays de OpenGL para la visualización de la malla, usando triángulos encadenados para una mayor eficiencia, pero cuya forma de almacenar y seleccionar los vértices hacen que el cálculo de los vértices y su ordenación sea más complicado.

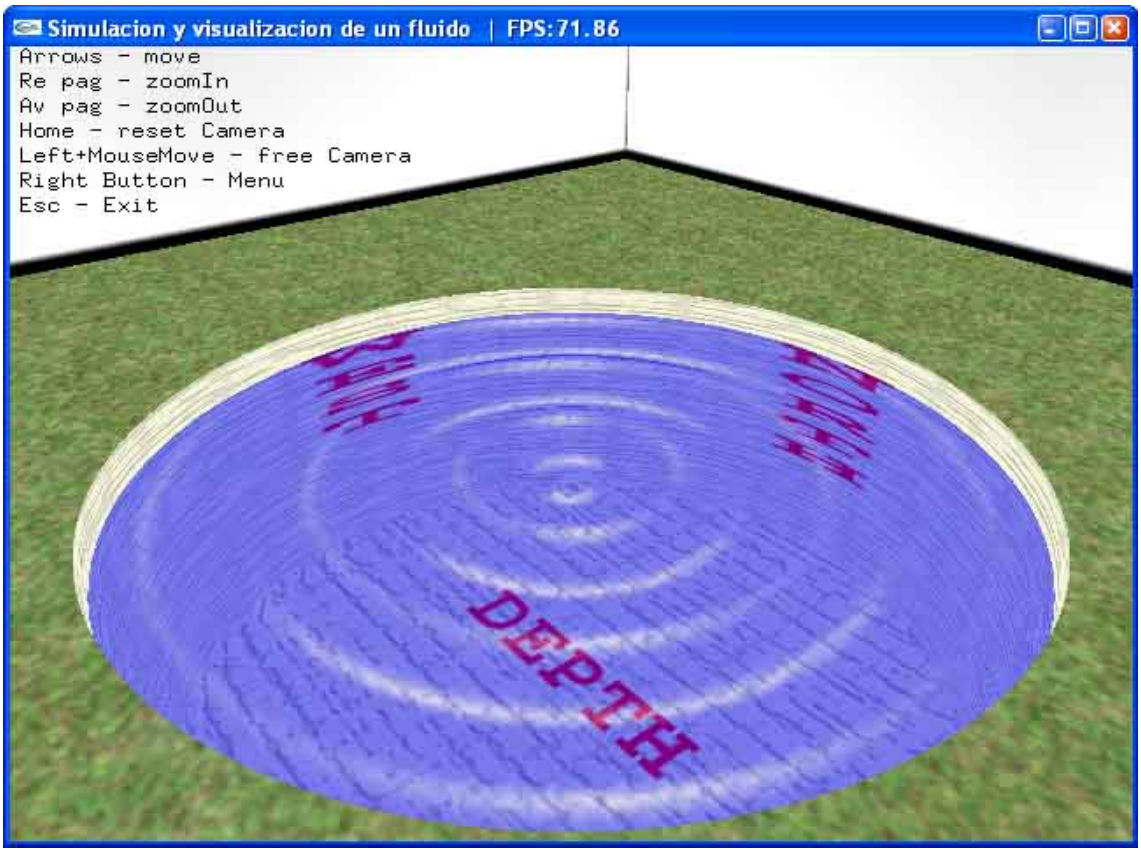
2.3.4 CAPTURA DE IMAGENES



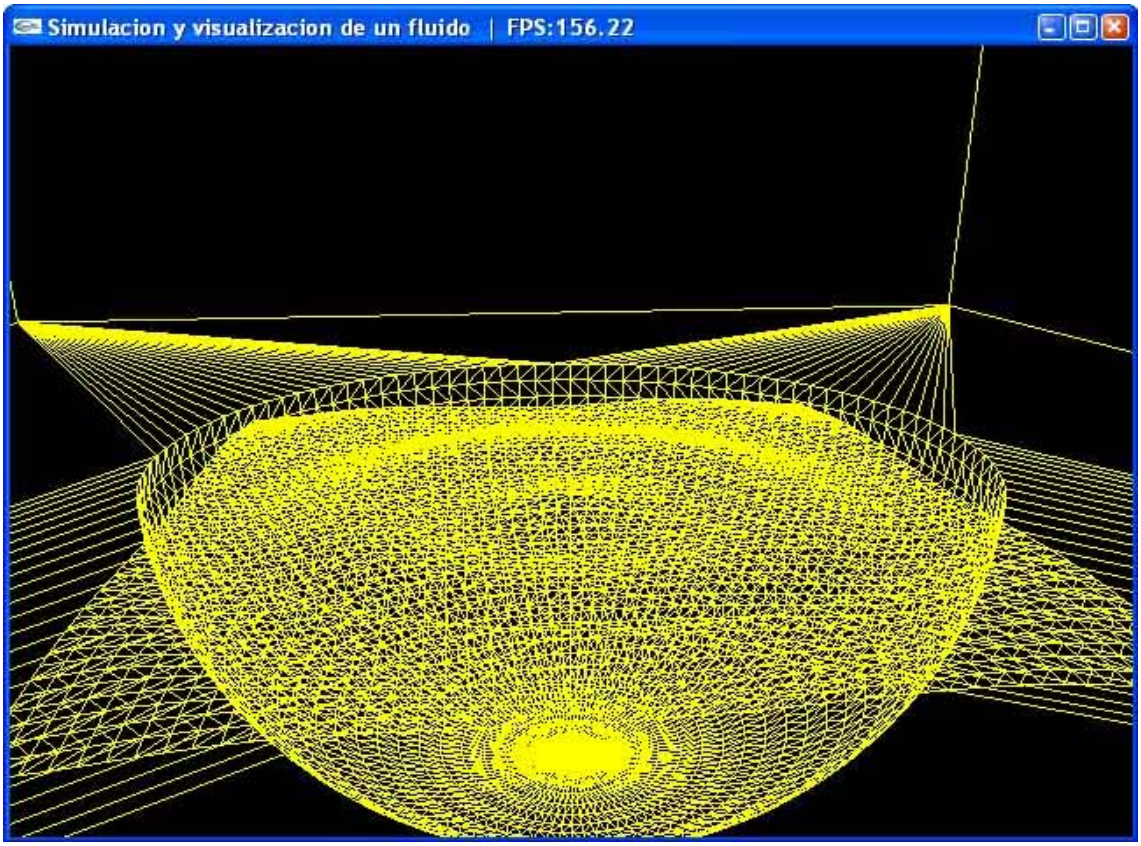
Piscina cúbica con olas trasversales



Piscina cilíndrica horizontal, solo malla



Piscina cilíndrica vertical con olas centradas sobre el origen



Piscina semiesférica, solo malla

2.4 TERCERA ITERACIÓN

2.4.1 MODELO DEL DOMINIO

Como es de suponer, al ser esta la última iteración, se ha planeado terminar el módulo1, para ello faltan fundamentalmente dos aspectos a terminar:

1. Cada uno de los programas en CG, tanto los vertex program como los fragment program, este es el bloque fundamental y más importante del módulo1, y que, como se verá más adelante, con un solo vertex program es suficiente, pero se necesitan tantos fragment program como efectos diferentes por cada uno de los tipos de piscinas.

2. El código que hace interactuar los programas de CG con el código en C++ y los objetos dibujados con OpenGL, para este fin, el profesor supervisor del proyecto ha suministrado unas clases que facilitan el uso de CG (ext, CFragmentProgram, CVertexProgram), las cuales hacen uso de las extensiones de nvidia GL_VERTEX_PROGRAM_NV y GL_FRAGMENT_PROGRAM_NV, siendo estas clases un sustituto para las CG runtime (herramienta suministrada por CG para dicho fin), faltando únicamente hacer que resto de las clases del módulo interactúen con las clases suministradas por el profesor.

**Nota: esta iteración no requiere de diseño, puesto que no aparecen clases nuevas, sino el uso de los programas de CG y la interacción entre C++ y CG lo cual ya fue planeado en la iteración anterior.*

2.4.2 IMPLEMENTACIÓN

La implementación de esta iteración es la más costosa de todas, no por la cantidad de código que puedan contener lo programas en CG, sino por el coste de la depuración del código de CG, puesto que no tiene ningún tipo de debuger y carece de cualquier tipo de herramienta que te ayude a depurar el código, la única manera posible de saber algo sobre algún tipo de variable interna es acotar o ponderar la variable entre cero y uno y sacarlo como un color (ejemplo el rojo) sobre la imagen renderizada, pudiéndote hacer una idea del valor aproximado de dicha variable.

Para empezar a ver lo realizado sobre esta iteración habría que empezar a mirar cómo interactúa el código posteriormente escrito de CG con el código de C++ y los objetos creados de OpenGL. Como ya hemos comentado antes para este uso el profesor supervisor del proyecto nos ha proporcionado unas clases que hacen de intermediarios entre los dos lenguajes (ext, CFragmentProgram, CVertexProgram), dichas clases son ensambladas dentro del proyecto y son usadas principalmente en tres lugares:

Para el uso correcto de estas tres clases es necesario la previa inicialización de la clase ext, esta inicialización está dentro del código main de la clase waterCG y se realiza llamando a la función initExtensions().

Un uso correcto de CG implica también una previa inicialización de CG indicándoles cuáles van a ser las variables que se les van a pasar del código de C++ al código de CG y cuáles van a ser sus valores, y a su vez cuál es el nombre de los ficheros de CG que se van a usar para el renderizado, es decir, que vertex program y qué fragment program se van a utilizar para el renderizado de un determinado objeto, todo esto es dependiente del tipo de piscina que se está mostrando en la escena, por este motivo se hace uso del patrón factoría y de cada una de las factorías concretas para pasarle estos datos a CG, además también es dependiente de cada uno de los efectos deseados, por lo cual cada una de las factorías tendrá una función para cada uno de los efectos, el resultado de esto es un código como este:

```
//-------------------------------------------------------------//
// variables necesarias para el vertex program //
//-------------------------------------------------------------//
float etaRatio[1];
etaRatio[0] = map.GetFloatValue("etaRatio");

//-------------------------------------------------------------//
// variables necesarias para el fragment program //
//-------------------------------------------------------------//
float fresnelBias[1];
fresnelBias[0] = map.GetFloatValue("fresnelBias");
float fresnelScale[1];
fresnelScale[0] = map.GetFloatValue("fresnelScale");
float fresnelPower[1];
fresnelPower[0] = map.GetFloatValue("fresnelPower");
```

```
float radius[1];
radius[0] = (map.GetFloatValue("Xpool")+map.GetFloatValue("Zpool"))/4;
float room[3];
room[0] = map.GetFloatValue("Xroom")/2;
room[1] = map.GetFloatValue("Yroom");
room[2] = map.GetFloatValue("Zroom")/2;

//----------------------------//
//--- vertex program ---//
//----------------------------//
water->vp->loadFile( "VertexProgram.vp" );
water->vp->trackModelProjMtxIde( 0 );
water->vp->trackModelMtxInvTra( 4 );

//-----------------------------//
//-- fragment program --//
//-----------------------------//
water->fp->loadFile( "SphereFresnelFP.fp" );
water->fp->setNamedParam( "etaRatio", etaRatio );
water->fp->setNamedParam( "fresnelBias", fresnelBias );
water->fp->setNamedParam( "fresnelScale", fresnelScale );
water->fp->setNamedParam( "fresnelPower", fresnelPower );
water->fp->setNamedParam( "radius", radius );
water->fp->setNamedParam( "room", room );
```

En donde:

- map.GetFloatValue es un procedimiento propio de LoaderFile que carga los datos necesarios del fichero de configuración.
- loadFile es un procedimiento tanto de CFragmentProgram como de CVertexProgram que indica a CG quién contiene el código del fragment program o del vertex program que se va a utilizar de ahora en adelante.
- trackModelProjMtxIde y trakModelMtxInvTra son procedimientos que indican al vertex program cómo obtener las matrices de proyección y la inversa traspuesta, usados por el vertex program para el cálculo de los puntos y de las normales a los puntos, y en qué posición de memoria intermedia serán almacenadas.
- setNamedParam es un procedimiento que le indica al fragment program qué variable de C++ se asocia a qué variable del código de CG.

Por último es necesario activar el uso de los códigos del fragment y vertex program en aquellos objetos que queramos que se modifiquen, en nuestro caso el único objeto que ha de ser modificado por el vertex program y el fragment program es la superficie del agua (surface). El código de la activación y desactivación del uso de CG es el siguiente:

```
//Activación del fragment y vertex program
vp->on();
fp->on();
vp->bind();
fp->bind();
```

```
//desactivación del uso del fragment y vertex program
vp->off();
fp->off();
```

vp y fp son atributos de la clase surface, en donde vp es un objeto de la clase CVertexProgram y fp es un objeto de la clase CFragmentProgram.

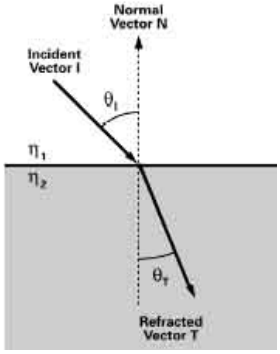
Inicialmente se había planteado también un vertex program por cada uno de los efectos en cada uno de los tipos de piscina, haciendo dentro del vertex program ciertos cálculos por vértices ahorrando así tener que hacerlos por píxeles. Finalmente se decidió hacer un único vertex program en el que simplemente le pasa ciertos valores necesarios para el fragment program, principalmente porque el cálculo por fragmento aunque sea menos eficiente consigue un realismo de la escena muy superior al cálculo por vértice, quedando el siguiente código:

```
postion = position;  //posicion original
oPosition = mul(modelViewProj, position);
oEyePos.xyz = eyePosition.xyz;
oNormal = normal;
oColor = color;
```

- A continuación pasaremos a hacer una breve explicación de cada uno de los efectos diseñados en este módulo.

REFRACCIÓN:

La refracción se produce cuando una onda, por ejemplo la luz, pasa de un material a otro distinto con distinta densidad (del aire al agua por ejemplo), el efecto de este fenómeno es un cambio en la dirección de la onda, ya que la onda cambia de velocidad al pasar de un material al otro. Este cambio de inclinación depende del ángulo de entrada de la onda sobre la normal a la superficie que forman los dos materiales y del cambio de densidad entre los dos materiales.



Antes de empezar a explicar el código realizado en la tercera iteración sobre CG, se ha de mencionar la ineficiencia de CG para las instrucciones condicionales, un programa puesto en CG con una instrucción condicional (if-else) es más lento un ese mismo programa en CG ejecutando directamente las dos partes de instrucción condicional. Por este motivo se ha intentado reducir al mínimo las instrucciones condicionales o en su defecto el código contenido dentro de dichas instrucciones condicionales. A su vez se ha intentado hacer el mayor uso posible de los cálculos en paralelo que gracias a la aceleración por hardware de CG.

Para el cálculo del vector de refracción primero se ha calculado el vector de incidencia sobre el agua, haciendo la resta entre la posición del punto y de la cámara, y usando este vector ,el valor de la variable “etaRatio” que se supone el coeficiente de refracción de la longitud de una onda con respecto a la diferencia de densidades entre los dos materiales y la normal al punto se llama a la función geométrica de CG llamada refract(I, N, eta). Este cálculo es común para todos los tipos de piscina, en lo que se diferencian cada uno de los tipos de piscinas es, que una vez con este vector y el punto de origen, ver en qué parte de la piscina impactaría dicho vector y calcular también dependiendo de la forma de la imagen el píxel asociado a esa posición de la piscina.

- Piscina cúbica:

Para el calculo del punto de choque entre la piscina y el vector se ha usado un algoritmo que se basa en, poniendo todas las paredes de la piscina en forma paramétrica, se calculan todos los tHits del vector con cada una de las paredes (o valores de la “t” en la forma paramétrica que hace cierto que el punto está a la vez en el vector y en el plano), de todos ellos, nos quedamos con el menor valor de “t” no negativo, puesto que el menor de ellos será con el primero que impacte el vector de todos los planos de la piscina, a su vez no permitimos que sea negativo puesto que esto significaría que ese punto está en la dirección contraria al vector, es decir, por encima del agua.

Una vez sabemos sobre qué pared impacta el vector y en qué posición exactamente, ponderamos el valor de la anchura y de la altura del punto de intersección sobre 0 y 0.33, y dependiendo de qué pared sea miramos sobre la textura en la posición correspondiente a ese plano, por ejemplo, si fuese el suelo, lo colocaríamos sobre el eje X entre el 0.33 y el 0.66 y sobre el eje Y

sobre el 0.33 y el 0.66, averiguando así el píxel que le corresponde a la piscina en ese punto.

Algo que es común a todos los efectos y a todas las formas de las piscinas es una ligera tonalidad azulada que se le ha dado al agua para que exista una diferencia más sustancial entre la pared de la piscina que queda por encima del agua y la que queda por debajo.

Código de CubeRefractFP.cg:

```
  float3 I = position.xyz - eyePos;
  float3 vector = refract(I,normal.xyz, etaRatio);

 //cara Norte
  float t1=(-pool.z-position.z)/vector.z;
 //cara Sur (el negativo es el que no nos sirve)
  float t2=(pool.z-position.z)/vector.z;
  float tmin=max(t1,t2);
 //cara Este
  float t3=(pool.x-position.x)/vector.x;
 //cara Oeste
  float t4=(-pool.x-position.x)/vector.x;
  tmin = min(tmin, max(t3,t4));
 //Base
  float t0=(-pool.y-position.y)/vector.y;
  tmin=min(tmin,t0);

/* Ahora tengo tmin el t más pequeño de la recta en forma paramétrica
   el más pequeño positivo es contra el plano con el que chocará   */
  float2 coordTex;
  float3 coordTexAux;

 //Cálculos en paralelo que se pueden sacar de las instrucciones condicionales
  float3 aux = position.xyz + vector.xyz * tmin;
  coordTexAux.xz = ((aux.xz+pool.xz)/(6*pool.xz))+0.333333;
  coordTexAux.y = ((aux.y)/(3*pool.y));

//en las instrucciones condicionales hacemos el menor número de cálculos posibles
  if(tmin == t0)  //base
  {
    coordTex.x = coordTexAux.x;
    coordTex.y = 1-coordTexAux.z;
  }
  if(tmin == t1) //norte
  {
    coordTex.x = coordTexAux.x;
    coordTex.y = coordTexAux.y;
  }
  if(tmin == t2) //sur
  {
    coordTex.x = coordTexAux.x;
    coordTex.y = 1-coordTexAux.y;
```

```
}
if(tmin == t3) //este
{
  coordTex.x = coordTexAux.y;
  coordTex.y = 1-coordTexAux.z;
}
if(tmin == t4) //oeste
{
  coordTex.x = 1-coordTexAux.y;
  coordTex.y = 1-coordTexAux.z;
}

//a partir de aquí es común para todos
 float4 decalColor = tex2D(texPool, coordTex); //cálculo del píxel a partir de la pos
oColor = lerp(decalColor, color, 0.1);  //le ponemos una ligera tonalidad azulada
```

- Piscina semiesférica:

El cálculo del punto de intersección de esta piscina se basa únicamente en, sabiendo el radio de la piscina, ver cuándo el vector atraviesa la esfera formada por ese vector, quedándonos solo con la parte positiva.

Código de SphereRefracFP.cg:

```
//calculo del vector de la refracción por pixel, algo menos eficiente pero más exacto
float3 I = position.xyz - eyePos;
float3 vector = refract(I,normal.xyz, etaRatio);

float3 vectorAux = vector * vector;
float a = vectorAux.x + vectorAux.y + vectorAux.z;

vectorAux = position.xyz * vector;
float b = 2*(vectorAux.x + vectorAux.y + vectorAux.z);

vectorAux = position.xyz * position.xyz;
float c = vectorAux.x + vectorAux.y + vectorAux.z - (radius * radius);

float t = (-b +  sqrt((b * b) - (4 * a * c)))/ (2 * a);

float3 aux = position.xyz + vector * t;
float alfaNor = acos(dot(normalize(aux),float3(0,-1,0)))*0.6366;

float dir = acos(dot(normalize(aux.xz), float2(1,0)));
//el producto escalar solo te da hasta pi, hay que mirar si esta entre pi y 2*pi
if( aux.z < 0 ) dir = (6.2832)-dir;
float2 ang;

//ang.y = sin(dir), ang.x = cos(dir) más eficiente que calcularlos individualmente
sincos(dir, ang.y, ang.x);
ang.y = -ang.y;
```

```
float2 coordTex = 0.5 + alfaNor * ang * 0.5;

float4 decalColor = tex2D(texPool, coordTex);
oColor = lerp(decalColor, color, 0.1);
```

- Piscina cilíndrica horizontal:

La idea del cálculo con la piscina horizontal es bastante similar a la semiesférica, para el cálculo de la parte circular calculamos cuando el vector sale de cilindro, para ello, hacemos los mismos cálculos que en la semiesférica pero ignorando la coordenada Z, por lo cual obtenemos cuando intersecan el vector con un tuvo del radio que nos interese orientado en el eje Z, para terminar de acotarlo calculamos cuando intersecan el vector y las 2 pareces usando las formas paramétricas de los planos que contienen las paredes.

Código de CylHorRefract.cg:

```
float3 I = position.xyz - eyePos;
float3 vector = refract(I,normal.xyz, etaRatio);

float2 vectorAux = vector.xy * vector.xy;
float a = vectorAux.x + vectorAux.y;

vectorAux = position.xy * vector.xy;
float b = 2*(vectorAux.x + vectorAux.y);

vectorAux = position.xy * position.xy;
float c = vectorAux.x + vectorAux.y - ((radius/2) * (radius/2));

float t, t1, t2;
float3 aux, aux1, aux2;
t = (-b +  sqrt((b * b) - (4 * a * c)))/ (2 * a);

aux = position.xyz + vector * t;

float dir = acos(dot(normalize(aux.xy), float2(1,0)));
float2 coordTex;

t1=(width-position.z)/vector.z;
t2=(-width-position.z)/vector.z;

aux1 = position.xyz + vector * t1;
aux2 = position.xyz + vector * t2;

if((aux.z<width) && (aux.z>(-width)))
{
  coordTex.x = 1-(aux.z/(4*width)+0.5);
  coordTex.y = dir*0.31831;                    // = dir/pi;
```

```
}
if(aux.z>width)
{
  coordTex.x = 1-(aux1.y/(2*radius));
  coordTex.y = 0.5-aux1.x/radius;
}
if(aux.z<-width)
{
  coordTex.x = aux2.y/(2*radius);
  coordTex.y = 0.5-aux2.x/radius;
}

float4 decalColor = tex2D(texPool, coordTex);
oColor = lerp(decalColor, color, 0.1);
```

- Piscina cilíndrica vertical:

En el cálculo de las paredes laterales de las piscinas, y usando otra vez la misma idea que para la semiesférica y la cilíndrica horizontal, lo que ignoramos en este caso es la coordenada de la altura, o coordenada Y, obteniendo la intersección del vector con un cilindro centrado sobre el eje Y, por último y con la forma paramétrica calculamos la intersección del vector con el plano que forma el suelo.

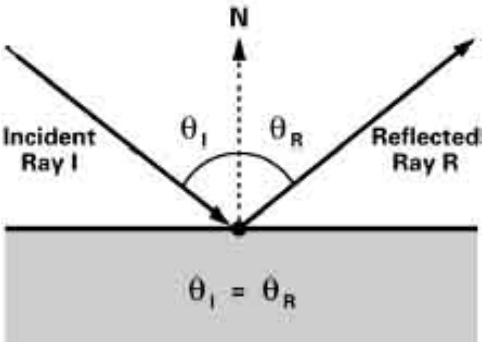
Código de CylVertRefractFP.cg:

```
float3 I = position.xyz - eyePos;
float3 vector = refract(I,normal.xyz, etaRatio);
float radAux = radius*0.5;

float2 vectorAux = vector.xz * vector.xz;
float a = vectorAux.x + vectorAux.y;

vectorAux = position.xz * vector.xz;
float b = 2*(vectorAux.x + vectorAux.y);

vectorAux = position.xz * position.xz;
float c = vectorAux.x + vectorAux.y - (radAux * radAux);

float t = (-b +  sqrt((b * b) - (4 * a * c)))/ (2 * a);

float3 aux = position.xyz + vector * t;

float dir = acos(dot(normalize(aux.xz), float2(1,0)));

//el producto escalar solo te da hasta pi, hay que mirar si esta entre pi y 2*pi
if( aux.z < 0 ) dir = (6.2832)-dir;
dir = dir*0.07957747;                    // = dir/4*pi;

float t0=(deep -position.y)/vector.y;
```

```
float3 aux2 = position.xyz + vector.xyz * t0;
float2 coordTex;

if (length(float3(aux2.x,0,aux2.z))<=radAux)
{
   coordTex.x = (aux2.x+radAux)/(4*radAux);
   coordTex.y = 1-(aux2.z+radAux)/(2*radAux);
}
else
{
  coordTex.x = 0.5 + dir;
   coordTex.y = 1-(aux.y/deep);
}

float4 decalColor = tex2D(texPool, coordTex);
oColor = lerp(decalColor, color, 0.1);
```

REFLEXIÓN:

La reflexión es un efecto producido cuando la luz ha de atravesar de un material a otro con mayor densidad, en el cual en vez de entrar sale reflejado con un ángulo de salida igual al ángulo de entrada, que se produzca este efecto depende tanto de el ángulo con el que intenta entrar la luz en el nuevo material, como de la densidad y composición del nuevo material.



El cálculo del vector de reflexión es simétrico al de refracción, se calcula el vector de incidencia como la resta entre la posición del punto y la posición de la cámara y después se usa el procedimiento de CG llamado reflect(I, N).

El método usado para cada una de las piscinas es muy similar en todos ellos, primero se calcula el punto de intersección entre el vector de reflexión y la piscina, usando los métodos mencionados en la refracción para cada una de ellas y evitándonos cálculos innecesarios como por ejemplo los cálculos con el suelo, y posteriormente se calcula el punto de intersección del vector con la habitación usando el mismo método que el usado en la refracción con la piscina cúbica.

Por similitud solo pondré el código de la piscina esférica (SphereRefractFP.cg):

```
//calculo del vector de la reflexión por píxel
float3 I = position.xyz - eyePos;
float3 vector = reflect(I,normal.xyz);

// calculo de la t para la piscina
float3 vectorAux = vector * vector;
float a = vectorAux.x + vectorAux.y + vectorAux.z;
vectorAux = position.xyz * vector;
float b = 2*(vectorAux.x + vectorAux.y + vectorAux.z);
vectorAux = position.xyz * position.xyz;
float c = vectorAux.x + vectorAux.y + vectorAux.z - (radius * radius);

float tmin1 = (-b +  sqrt((b * b) - (4 * a * c)))/ (2 * a);

//cara Norte habitación
float t21=(-room.z-position.z)/vector.z;
//cara Sur habitación (el negativo es el que no nos sirve)
float t22=(room.z-position.z)/vector.z;
//cara Este habitación
float t23=(room.x-position.x)/vector.x;
//cara Oeste habitación
float t24=(-room.x-position.x)/vector.x;
```

```
float tmin2 = min(max(t21,t22), max(t23,t24));
//Base habitación
float t20=(room.y-position.y)/vector.y;
tmin2=min(tmin2,t20);

float2 coordTex1, coordTex2;
float4 decalColor;
float3 coordTexAux, aux1, aux2;
aux1 = position.xyz + vector * tmin1;
aux2 = position.xyz + vector * tmin2;

coordTexAux.xz = ((aux2.xz+room.xz)/(6*room.xz))+0.333333;
coordTexAux.y = ((aux2.y)/(3*room.y));

//techo habitación
if(tmin2 == t20)
{
  coordTex2.x = coordTexAux.x;
  coordTex2.y = 1-coordTexAux.z;
}
//norte habitacion
if(tmin2 == t21)
{
  coordTex2.x = 1-coordTexAux.x;
  coordTex2.y = 1-coordTexAux.y;
}
//sur habitacion
if(tmin2 == t22)
{
  coordTex2.x = 1-coordTexAux.x;
  coordTex2.y = coordTexAux.y;
}
//este habitacion
if(tmin2 == t23)
{
  coordTex2.x = 1-coordTexAux.y;
  coordTex2.y = coordTexAux.z;
}
//oeste habitación
if(tmin2 == t24)
{
  coordTex2.x = coordTexAux.y;
  coordTex2.y = coordTexAux.z;
}

float alfaNor = acos(dot(normalize(aux1),float3(0,-1,0)))*0.6366;

float dir = acos(dot(normalize(aux1.xz), float2(1,0)));
//el producto escalar solo te da hasta pi, hay que mirar si esta entre pi y 2*pi
if( aux1.z < 0 ) dir = (6.2832)-dir;
```

```
float2 ang;
//ang.y = sin(dir), ang.x = cos(dir) más eficiente que calcularlos individualmente
sincos(dir, ang.y, ang.x);
ang.y = -ang.y;

coordTex1 = 0.5 + alfaNor * ang * 0.5;

if(aux1.y < 0)
  decalColor = tex2D(texPool, coordTex1);
else
  decalColor = tex2D(texRoom, coordTex2);
oColor = lerp(decalColor, color, 0.1);
```

EFECTO FRESNEL

El efecto Fresnel se basa en la idea de que no todos los rayos de luz son refractados o reflectados, sino que una parte de los rayos son reflectados y el resto refractados dependiendo de un factor llamado Coeficiente de reflexión, este factor es calculado como:

$RefectionCoefficient = max(0, min(1, bias + scale \times (1 + 1 \bullet N)^{power}))$

Una idea intuitiva de este coeficiente es, que cuanta mayor sea la inclinación o el ángulo formado entre la normal del punto y el vector más es la parte reflejada, y cuando menos inclinación mayor será la refracción.

La forma de programar este efecto es calcular primero la reflexión y la refracción individualmente, calcular el coeficiente de reflexión y mezclar las imágenes producidas por la reflexión y la refracción dependiendo de dicho coeficiente, las partes que cambiarían serían, por ejemplo para la piscina cúbica

Código de CubeFresnelFP.cg:

```
//calculo del vector de la reflexion por pixel
float3 I = position.xyz - eyePos;
float3 vector = reflect(I,normal.xyz);
float fNdotV = dot(I, normal.xyz)/360; //parametro de fresnell
float reflectionFactor = fresnelBias + fresnelScale * pow(1 + fNdotV, fresnelPower);

float t11=(-pool.z-position.z)/vector.z;    //cara Norte piscina
float t12=(pool.z-position.z)/vector.z;     //cara Sur piscina (el negativo es el que no nos sirve)
float t13=(pool.x-position.x)/vector.x;     //cara Este piscina
float t14=(-pool.x-position.x)/vector.x;    //cara Oeste piscina
float tmin1 = min(max(t11,t12), max(t13,t14));

float t21=(-room.z-position.z)/vector.z;    //cara Norte habitacion
float t22=(room.z-position.z)/vector.z;     //cara Sur habitacion (el negativo es el que no nos sirve)
float t23=(room.x-position.x)/vector.x;     //cara Este habitacion
float t24=(-room.x-position.x)/vector.x;    //cara Oeste habitacion
float tmin2 = min(max(t21,t22), max(t23,t24));
float t20=(room.y-position.y)/vector.y;     //Base habitacion
tmin2=min(tmin2,t20);

float2 coordTex1, coordTex2, coordTex;
float4 decalColor1, decalColor2;
float3 coordTexAux, aux1, aux2, who, aux;
aux1 = position.xyz + vector.xyz * tmin1;
aux2 = position.xyz + vector.xyz * tmin2;
```

```
if(aux1.y < 0)
{
   who = pool;
   aux = aux1;
}
else
{
   who = room;
   aux = aux2;
}

coordTexAux.xz = ((aux.xz+who.xz)/(6*who.xz))+0.333333;
coordTexAux.y = ((aux.y)/(3*who.y));

if(tmin1 == t11) //norte piscina
{
  coordTex1.x = coordTexAux.x;
  coordTex1.y = coordTexAux.y;
}
if(tmin1 == t12) //sur piscina
{
  coordTex1.x = coordTexAux.x;
  coordTex1.y = 1-coordTexAux.y;
}
if(tmin1 == t13) //este piscina
{
  coordTex1.x = coordTexAux.y;
  coordTex1.y = 1-coordTexAux.z;
}
if(tmin1 == t14) //oeste piscina
{
  coordTex1.x = 1-coordTexAux.y;
  coordTex1.y = 1-coordTexAux.z;
}

if(tmin2 == t20)  //techo habitacion
{
  coordTex2.x = coordTexAux.x;
  coordTex2.y = 1-coordTexAux.z;
}
if(tmin2 == t21) //norte habitacion
{
  coordTex2.x = 1-coordTexAux.x;
  coordTex2.y = 1-coordTexAux.y;
}
if(tmin2 == t22) //sur habitacion
{
  coordTex2.x = 1-coordTexAux.x;
  coordTex2.y = coordTexAux.y;
}
```

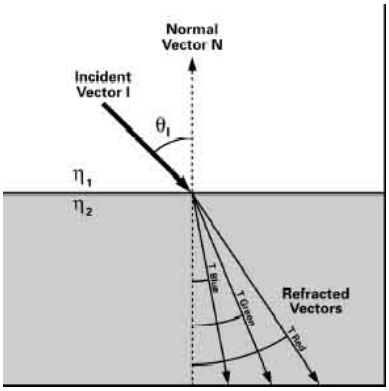
```
if(tmin2 == t23) //este habitacion
{
  coordTex2.x = 1-coordTexAux.y;
  coordTex2.y = coordTexAux.z;
}
if(tmin2 == t24) //oeste habitacion
{
  coordTex2.x = coordTexAux.y;
  coordTex2.y = coordTexAux.z;
}

if(aux1.y < 0)
  decalColor1 = tex2D(texPool, coordTex1);
else
  decalColor1 = tex2D(texRoom, coordTex2);


//calculo del vector de la refracción por pixel, algo menos eficiente pero más exacto
vector = refract(I,normal.xyz, etaRatio);

float t1=(-pool.z-position.z)/vector.z;    //cara Norte
float t2=(pool.z-position.z)/vector.z;   //cara Sur (el negativo es el que no nos sirve)
float t3=(pool.x-position.x)/vector.x;     //cara Este
float t4=(-pool.x-position.x)/vector.x;    //cara Oeste
float tmin = min(max(t1,t2), max(t3,t4));
float t0=(-pool.y-position.y)/vector.y;    //Base
tmin=min(tmin,t0);

/* Ahora tengo tmin el t más pequeño de la recta en forma paramétrica
   el más pequeño positivo es contra el plano con el que chocará   */

aux = position.xyz + vector.xyz * tmin;
coordTexAux.xz = ((aux.xz+pool.xz)/(6*pool.xz))+0.333333;
coordTexAux.y = ((aux.y)/(3*pool.y));

if(tmin == t0)  //base
{
  coordTex.x = coordTexAux.x;
  coordTex.y = 1-coordTexAux.z;
}
if(tmin == t1) //norte
{
  coordTex.x = coordTexAux.x;
  coordTex.y = coordTexAux.y;
}
if(tmin == t2) //sur
{
  coordTex.x = coordTexAux.x;
  coordTex.y = 1-coordTexAux.y;
}
```

```
if(tmin == t3) //este
{
  coordTex.x = coordTexAux.y;
  coordTex.y = 1-coordTexAux.z;
}
if(tmin == t4) //oeste
{
  coordTex.x = 1-coordTexAux.y;
  coordTex.y = 1-coordTexAux.z;
}

decalColor2 = tex2D(texPool, coordTex);

oColor = lerp(lerp(decalColor2, decalColor1, reflectionFactor), color, 0.1);
```

Donde decalColor1 tiene el color que proporcionaría el cálculo de la reflexión y decalColor2 el color resultante del cálculo de la refracción.

EFECTO CROMÁTICO:

Para explicar este efecto primero deberíamos saber que la luz está compuesta por tres colores primos y que todo color que existe está formado por la combinación de estos tres colores. Este efecto se basa en la idea de que estos tres colores al ser ondas de distinta longitud tienen por tanto un coeficiente de refracción ligeramente distinto, lo cual produce que los colores se dispersen en la refracción.



Para la realización de este efecto se ha tenido que calcular un vector de refracción por cada uno de los tres colores primos usados en OpenGL (rojo, verde y azul), y para cada uno de ellos hacer los mismos cálculos que hacíamos con el efecto de refracción para cada una de las piscinas, teniendo que hacer de esta forma el triple de cálculos que para la refracción normal y bajando apreciablemente los frames por segundo obtenidos.

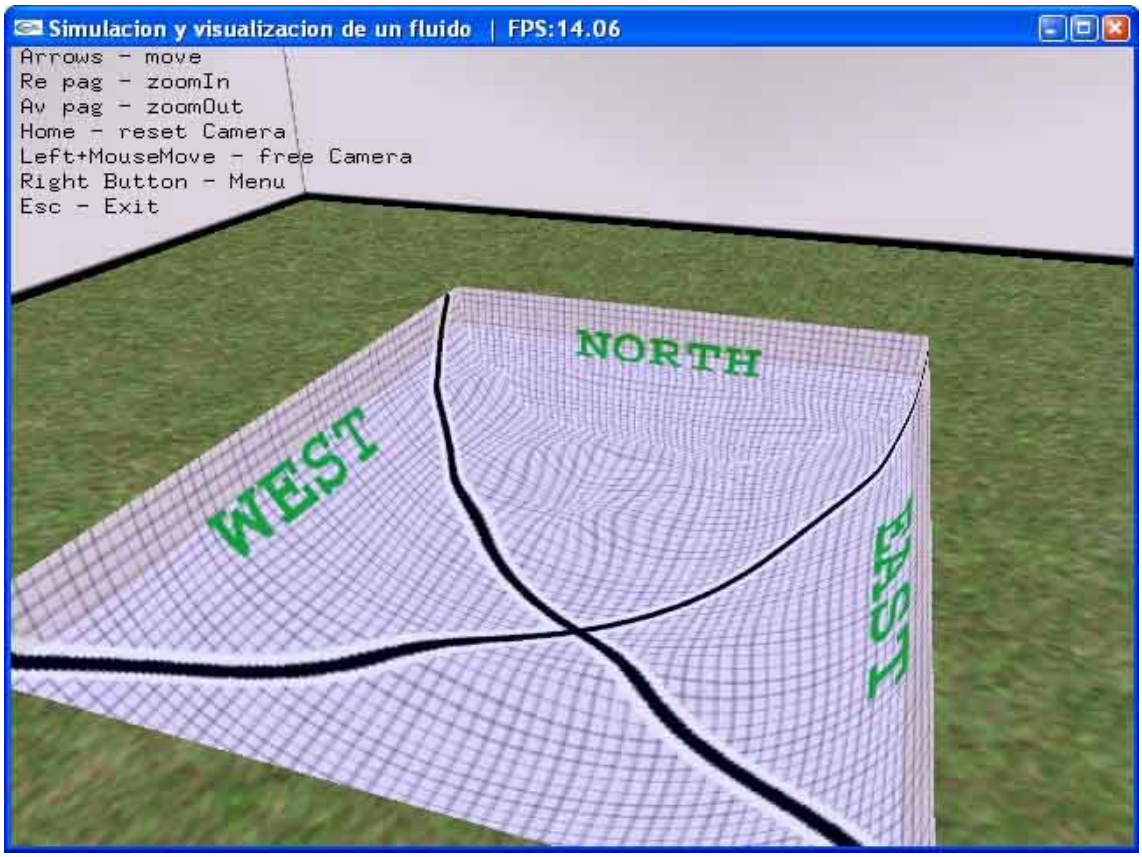
Para no repetir el mismo código y pondremos una parte orientativa del código del efecto cromático sobre la piscina semiesférica.

Código de SphereCromaticFP.cg:

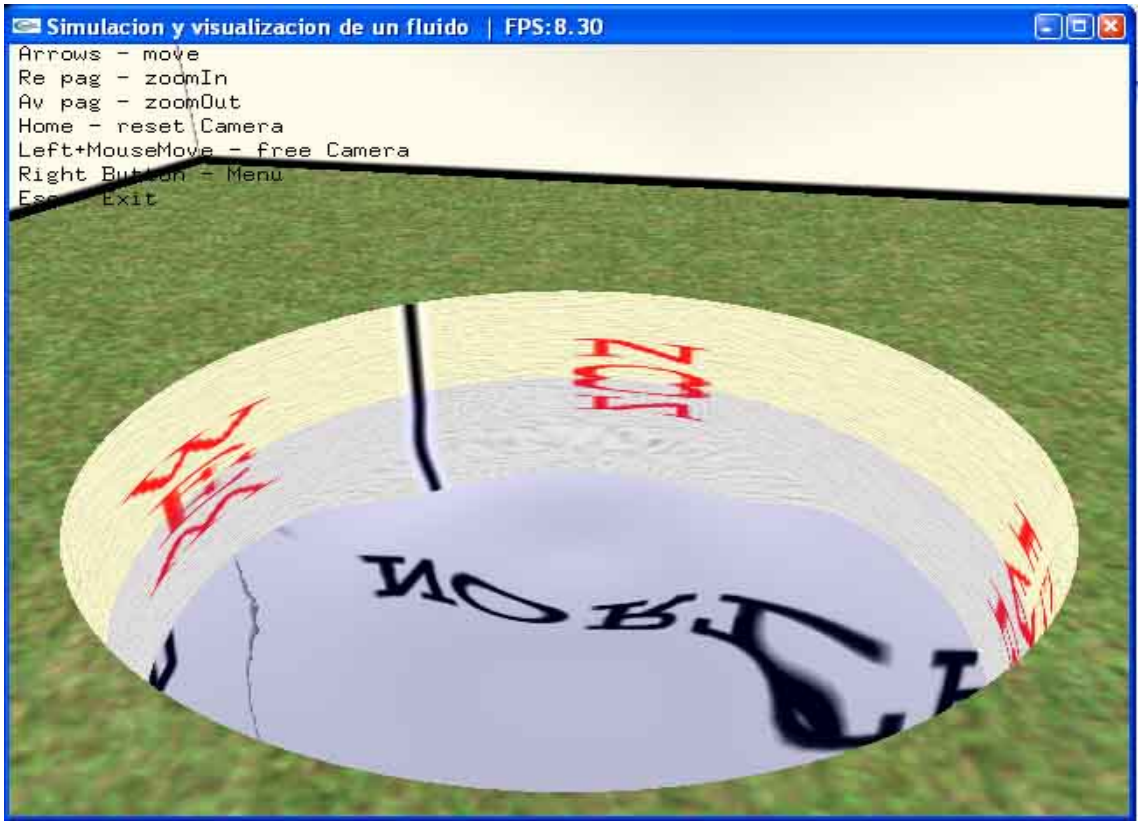
```
//calculo del vector de la refracción por pixel, algo menos eficiente pero más exacto
float3 I = position.xyz - eyePos;
float3 vectorR, vectorG, vectorB;
vectorR = refract(I,normal.xyz, etaRatio.r);
vectorG = refract(I,normal.xyz, etaRatio.g);
vectorB = refract(I,normal.xyz, etaRatio.b);

float3 a, b, t, vectorAuxR, vectorAuxG, vectorAuxB, auxR, auxG, auxB, alfaNor, dir;
float c;

vectorAuxR = vectorR * vectorR;
vectorAuxG = vectorG * vectorG;
vectorAuxB = vectorB * vectorB;
a.r = vectorAuxR.x + vectorAuxR.y + vectorAuxR.z;
a.g = vectorAuxG.x + vectorAuxG.y + vectorAuxG.z;
a.b = vectorAuxB.x + vectorAuxB.y + vectorAuxB.z;

vectorAuxR = position.xyz * vectorR;
vectorAuxG = position.xyz * vectorG;
vectorAuxB = position.xyz * vectorB;
b = 2*(vectorAuxR.x + vectorAuxR.y + vectorAuxR.z);
```

```
b = 2*(vectorAuxG.x + vectorAuxG.y + vectorAuxG.z);
b = 2*(vectorAuxB.x + vectorAuxB.y + vectorAuxB.z);

vectorAuxR = position.xyz * position.xyz;
c = vectorAuxR.x + vectorAuxR.y + vectorAuxR.z - (radius * radius);

t = (-b +  sqrt((b * b) - (4 * a * c)))/ (2 * a);

auxR = position.xyz + vectorR * t;
auxG = position.xyz + vectorG * t;
auxB = position.xyz + vectorB * t;
alfaNor.r = acos(dot(normalize(auxR),float3(0,-1,0)))*0.6366;
alfaNor.g = acos(dot(normalize(auxG),float3(0,-1,0)))*0.6366;
alfaNor.b = acos(dot(normalize(auxB),float3(0,-1,0)))*0.6366;

dir.r = acos(dot(normalize(auxR.xz), float2(1,0)));
dir.g = acos(dot(normalize(auxG.xz), float2(1,0)));
dir.b = acos(dot(normalize(auxB.xz), float2(1,0)));
//el producto escalar solo te da hasta pi, hay que mirar si esta entre pi y 2*pi
if( auxR.z < 0 ) dir.r = (6.2832)-dir.r;
if( auxG.z < 0 ) dir.g = (6.2832)-dir.g;
if( auxB.z < 0 ) dir.b = (6.2832)-dir.b;

float2 angR, angG, angB;
sincos(dir.r, angR.y, angR.x);
sincos(dir.g, angG.y, angG.x);
sincos(dir.b, angB.y, angB.x);
angR.y = -angR.y;
angG.y = -angG.y;
angB.y = -angB.y;

float2 coordTexR, coordTexG, coordTexB;
coordTexR = 0.5 + alfaNor.r * angR * 0.5;
coordTexG = 0.5 + alfaNor.g * angG * 0.5;
coordTexB = 0.5 + alfaNor.b * angB * 0.5;

float4 decalColor;
decalColor.r = tex2D(texPool, coordTexR).r;
decalColor.g = tex2D(texPool, coordTexG).g;
decalColor.b = tex2D(texPool, coordTexB).b;
decalColor.a = 1;

oColor = lerp(decalColor, color, 0.1);
```

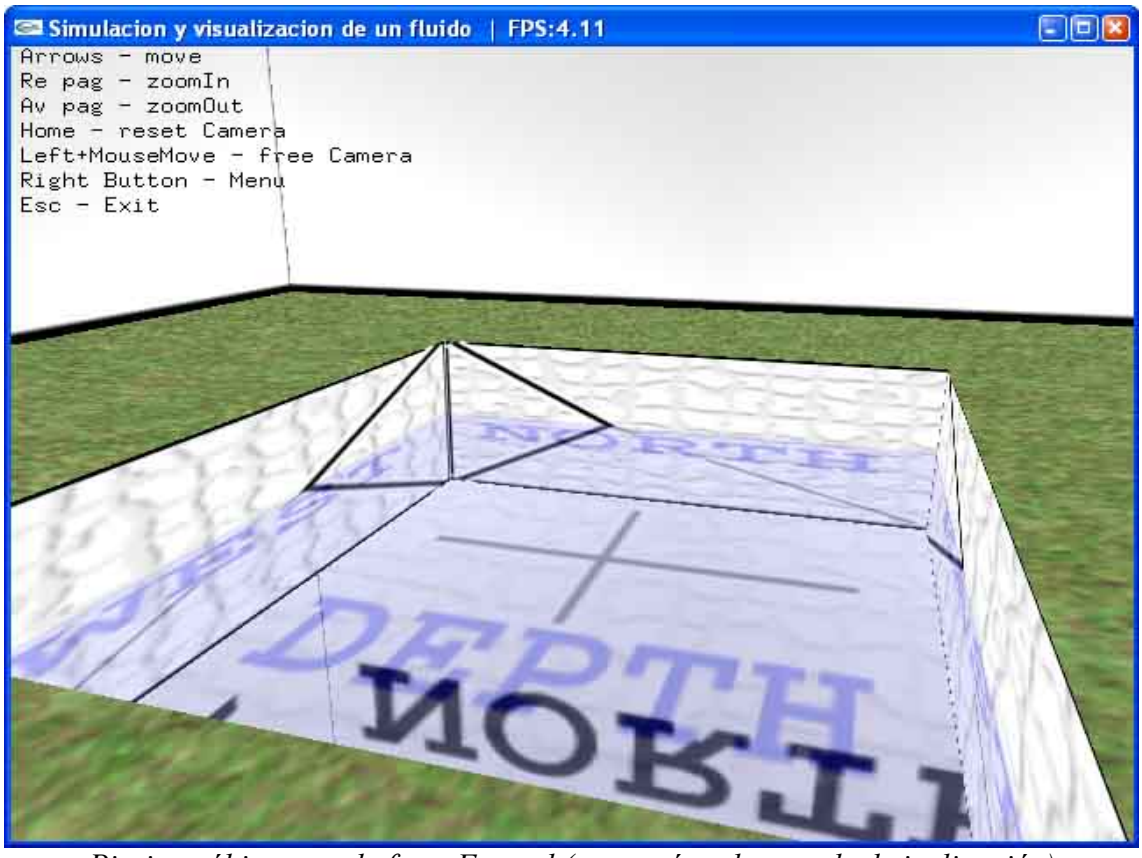
2.4.3 CAPTURA DE IMÁGENES



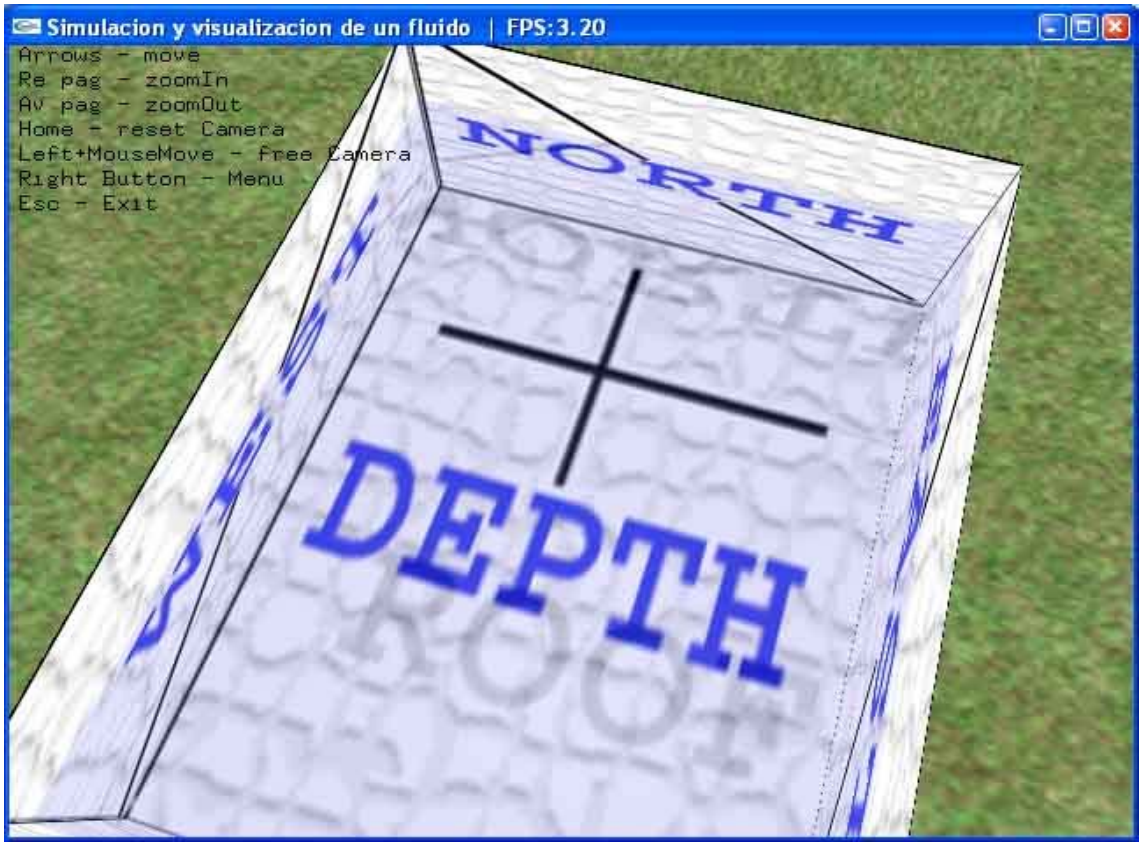
Piscina cilíndrica horizontal con efecto de refracción



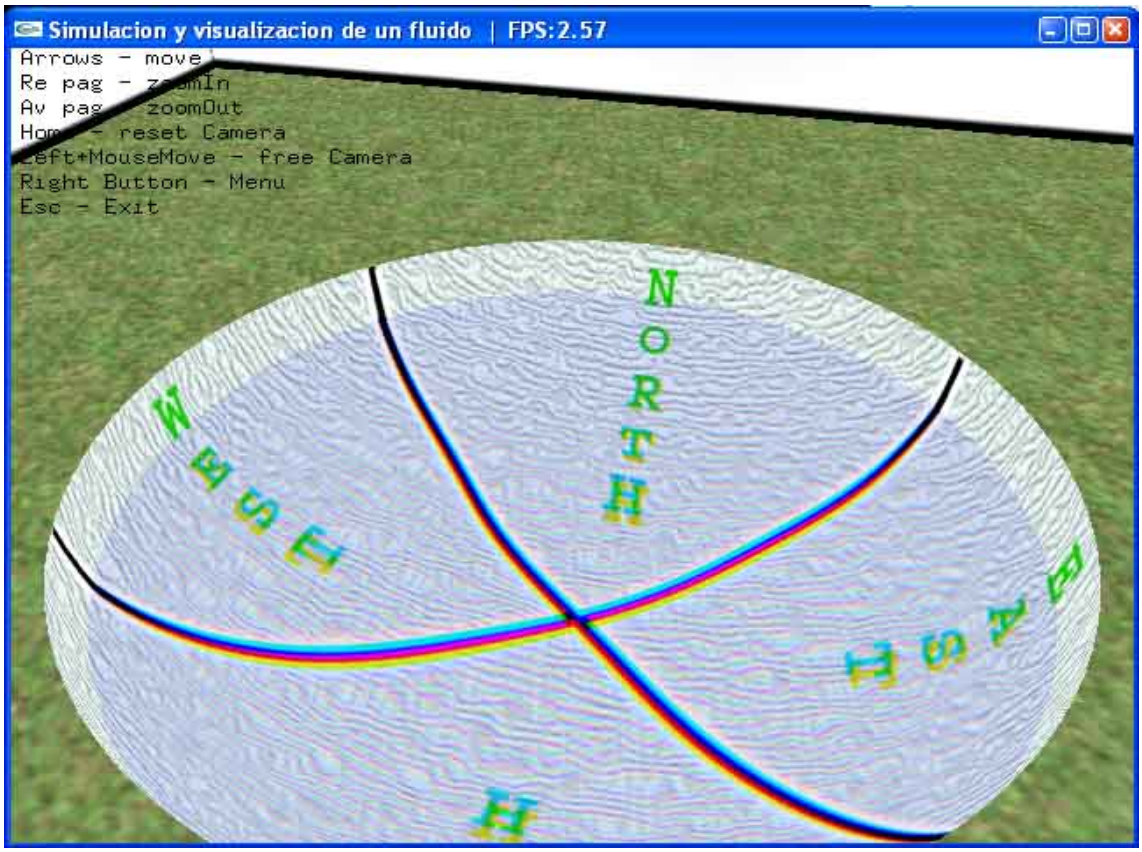
Piscina cilíndrica horizontal con efecto de reflexión



Piscina cúbica con el efecto Fresnel (con un ángulo grande de inclinación)



Piscina cúbica con el efecto Fresnel (con un ángulo pequeño de inclinación)



Piscina semiesférica con el efecto cromático

En las texturas de las piscinas se han incluido líneas blancas con otra línea negra en medio por dos motivos:

El primero es para comprobar su corrección, observando en cada tipo de piscina que efectivamente se cumple una continuidad con respecto a esas líneas de la parte que es directamente de la textura de la piscina, con la parte de la imagen calculada por CG, a su vez también también se puede comprobar una linealidad de la imagen, no existiendo cortes en las líneas, y una coherencia de las letras reflejadas y refractadas.

El segundo es para poder apreciar el efecto cromático, puesto que la mayor dispersión de colores se da cuando están el negro junto al blanco, además la dispersión cromática depende del ángulo de inclinación y de la profundidad del agua, puesto que el mismo ángulo de dispersión se nota más cuanto más lejos está la imagen ya que a los colores les da más tiempo a alejarse los unos de los otros, lo cual también se puede apreciar sobre la última imagen mostrada.

Hacer observar también que sobre el efecto fresnel cuando hay una mayor inclinación hay una mayor reflexión y cuando la inclinación disminuye destaca más la imagen refractada que la reflejada.

3. CONCLUSIONES

La forma más general de poder definir las conclusiones sobre CG sería como un lenguaje de alto nivel que permite hacer gran cantidad de efectos bastantes realistas directamente sobre el hardware, y que CG consigue un alto rendimiento para programas cortos y usando las funciones que tienen las tarjetas gráficas programables aceleradas por hardware.

Más en particular podemos ver ciertos aspectos de CG:

- El tamaño de los programas en CG: la programación con CG es muy eficiente para programas cortos haciendo uso de procedimientos acelerados por hardware (por ejemplo el environment mapping haciendo uso del procedimiento texCUBE), además posee gran cantidad de procedimientos matemáticos, geométricos, para el uso de texturas e incluso derivadas que permiten una realizar una gran variedad de programas en muy pocas líneas de código, pero cuando es necesario realizar una mayor cantidad de cálculo, pasando de las 50 líneas de código compiladas dentro de un programa en CG, el rendimiento de los programas en CG se degrada bastante, esta degradación se nota más cuando los cálculos se realizan sobre el fragment program, puesto que el cálculo por píxel siempre es más costoso que por vértice.

- Vertex program vs fragment program: los fragment program reciben la información desde el vertex program mediante la interpolación lineal de los vértices más cercanos, si un cálculo se puede realizar sobre un vertex program en lugar de en el fragment program la ejecución en genera será mucho más rápida, pero si el cálculo o la función calculada no es totalmente lineal entonces se pierde precisión en el proceso, perdiendo a su vez parte de realismo.

- Programación sobre CPU o programación sobre GPU: en general una CPU (programado con un lenguaje de alto nivel como C++) suele ser más rápida que una GPU (programado con CG), pero como es normal la carga que tiene la CPU también es muy superior a la GPU, el uso de CG y la programación sobre la GPU permite entre otras cosas liberar parte de la carga que posee la CPU, produciendo una mejora general del sistema. Además la GPU está preparada para la aceleración de ciertas funciones gráficas, junto con la posibilidad de realizar cálculos en paralelo sobre puntos, vectores y matrices, producen que la programación sobre la GPU sea más eficiente que la programación sobre la CPU para tareas relacionadas con operaciones gráficas y permite ciertos efectos que no se podrían obtener con la CPU gracias a la programación por fragmento que posee CG.

- Problemas de CG:

    - Un problema básico es la posible sobrecarga de la GPU, como ya se ha comentado la programación sobre la GPU permite una liberación de la carga de la CPU, pero se puede dar el caso contrario, haciendo que la GPU haga casi todo el trabajo, y decrementándose el rendimiento general del sistema.
    - CG posee una gran carencia interna, y es la gran ineficiencia que poseen las instrucciones condicionales, es más eficiente hacer las dos partes de la condicional que usar una condicional (if-else), e incluso resulta perjudicial si el objetivo de la instrucción condicional lo que pretende es ahorrar código de ejecución, puesto que el programa resultante es más lento que el mismo programa eliminando la condicional y dejando que ejecute siempre el código.
    - A nivel del programador CG presenta un gran inconveniente, y es la carencia de un debuger ni de ninguna herramienta que ayude al programador a la hora de intentar depurar el código, teniendo que optar por ponderar las variables entre cero y uno para poder mostrarlas como un color y poder hacerse una idea del valor de la variable, método que ni siquiera se puede usar con variables que cambian continuamente de valores.

4. MANUAL DEL DESARROLLADOR

Si se desea programar sobre CG, o si por ejemplo se deseara continuar este proyecto, habría ciertos programas y librerías que se deberían tener instalados en el ordenador:

Lo primero que habría que instalar sería el Microsoft Visual C++ 6.0, aunque con la versión 7.0 también funciona, solo habría que migrar el código, lo cual lo hace automáticamente el Visual, la instalación se basa en seguir el CD de la instalación, quedando instaladas automáticamente las librerías de OpenGL.

Para la instalación de las librerías glut, hay que poner: el archivo glut.dll MICROSOFT VISUAL STUDIO _HOME\VC98\BIN y en WINDOWS\SYSTEM32, el archivo glut.lib en MICROSOFT VISUAL STUDIO _HOME\VC98\LIB, y el archivo glut.h en el directorio MICROSOFT VISUAL STUDIO _HOME\VC98\INCLUDE\GL.

Para la correcta instalación de las librerías devil, hay que descomprimir los drivers sobre una carpeta vacía (denominaremos a la carpeta como C:\Devil), se recomienda que sea dentro del directorio principal del visual estudio (los drivers de las devil se pueden bajar de la página http://openil.sourceforge.net), una vez hecho esto tenemos que inicial el proyecto de Microsoft visual C++, ir a las “tools” – “options” y pinchar en la pestaña de “directories”, en el cual ha de estar seleccionado “include files”, le damos a añadir uno nuevo, y añadimos como nuevo “C:\Devil\Include”. Después seleccionamos “Library files” y añadimos una nueva entrada como “C:\Devil\lib”, en caso de habernos bajado también los drivers para debug añadiríamos también la entrada “C:\Devil\Lib\Debug”.

Para instalar CG, descomprimimos el archivo “Cg_toolkit.zip” (el cual se puede descargar en la página http://developer.nvidia.com/object/cg_toolkit.html) y ejecutamos “Setup.exe”, y para poder compilar el código de CG directamente sobre Microsoft Visual C++ hay que poner en los settings, en el hueco para “Commands” de todos los vertex program *"$(CG_BIN_PATH)\cgc" $(InputPath) -o $(InputName).vp -profile vp30*, y en el hueco para “Outputs” *$(InputName).vp*, igual hacemos para los fragment program cambiando “.vp” por “.fp” y “vp30” por “fp30”. Hay que tener en cuenta el tipo de opciones que estamos modificando, sería aconsejable hacerlos para cualquier tipo de compilación marcando la opción de “All configurations”.

A partir de aquí, y suponiendo que se posee una tarjeta gráfica programable ya estaría el ordenador configurado para poder generar código en CG haciendo uso de las librerías devil y openGL.

5. MANUAL DEL USUARIO

Para empezar a utilizar la demo de este módulo únicamente hay que descomprimir el archivo Ejecutable.zip, y ejecutar waterCG.exe.

El uso de la demo es bastante sencilla e intuitiva, mostrando una ayuda que si se desea se puede desactivar, básicamente las funcionalidades de la demo son:

- Flechas ←, →: mueven la cámara a derecha e izquierda de forma semiesférica centrada en el origen o centro de la piscina.
- Flechas ↑, ↓: mueven la cámara arriba o abajo de forma semiesférica centrada en el origen o centro de la piscina.
- RevPag: amplía la imagen o hace zoomIn.
- AvPag: encoge la imagen o hace zoomOut.
- Inicio: coloca la cámara en una posición de inicio preestablecida.
- Botón izquierdo del ratón + movimiento del ratón: permite un movimiento libre de la cámara.
- Botón derecho del ratón: despliega el menú.
- Esc: salir de la demo.
- S: para o comienza el movimiento de las olas.
- M: cambia la forma de las olas, de estar centradas en un punto a longitudinales y de longitudinales a trasversales.
- W: si la ola está centrada sobre un punto cambia el punto sobre el que está centrada a otro aleatorio sobre la superficie del agua.

El menú desplegable a su vez tiene las siguientes opciones:

- Opciones de la escena, crea una nueva escena con una piscina con la forma seleccionada.
  - Cúbica
  - Cilíndrica vertical
  - Cilíndrica horizontal
  - Semiesférica

- Opciones de efectos, escoge el efecto de CG deseado, cada vez que se selecciona una nueva escena, el efecto que se muestra sobre el agua vuelve a ser la refracción.
  - Reflexión
  - Refracción
  - Efecto Fresnel
  - Efecto cromático

- Opciones de vista, escoge las partes de las escena que se quieren ver u ocultar
    - Ver/Ocultar vértices
    - Ver/Ocultar malla
    - Ver/Ocultar caras
    - Ver/Ocultar parte de atrás

- Opciones del agua, selecciona la forma de moverse el agua
    - Stop/Start
    - Modo
    - Nueva ola

- Ocultar/Mostrar ayuda.
- FullScreen /Ocultar FullScreen, pone la ventana en pantalla completa o en pantalla reducida.
- Salir, sale de la demo.

6. BIBLIOGRAFIA Y DOCUMENTACIÓN

Libros:

Programación en OpenGL – Una guía de referencia completa de OpengGl
Autor: Richard S. Wright Jr. Michael Sweet
Anaya

The Cg Tutorial – The Definitive Guide to Programmable Real-Time Graphics
Autor: Randima Fernando and Mark J. Kilgard
Nvidia

Enlaces web:

- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/opengl/openglstart_9uw5.asp

- http://www.lighthouse3d.com/opengl/glut/index.php3?2

- http://www.fusionindustries.com/content/lore/code/articles/fi-faq-cg.php3

- http://openil.sourceforge.net/

- http://www.developers.nvidia.com/page/home

- http://opengl.org

- http://www.cgshaders.org/shaders