

Apuntes para el cursillo de Introducción a la
Programación en 3D

Autor: Iago De la Fuente González (Programador de Gargore Software).

CONTENIDO

TEORÍA:

- 1 - Introducción (qué forma un motor 3D).
- 2 - Primitivas básicas (línea, polígono, triángulo).
- 3 - Primitivas avanzadas (polígono texturado, sombreado goraud y phong, corrección de perspectiva).
- 4 - Entornos 3D (transformación de coordenadas cartesianas, cilíndrica y esférica).
- 5 - Introducción a OpenGL y Direct3D.
- 6 - Introducción a las técnicas de organización de espacio (CSP, QSP, BSP y Polychoop).

PRÁCTICA

- 7 - Sesión práctica sobre OpenGL.

TEMA 1

-Introducción-

¿Qué es programación 3D?

El concepto es totalmente subjetivo y depende de cada uno. Para unos incluye parte de gráficos y vectores, para otros también incluye el sonido y como le afectan las posiciones relativas y la velocidad, para otros es simplemente saber usar las Direct3D u OpenGL. También hay gente que opina q lo de la programación 3D son los juegos... como se ve, para gustos, hay colores.

Yo opino que se trata de todo un poco, pero hay que ordenar algo todo esto. Programación 3D incluye parte de programación gráfica y de vectores -necesaria para todo tipo de implementaciones Software/Hardware-, incluye también el sonido espacial -pese a que no sea necesario, es muy recomendable para crear ambiente-. Además, si deseamos usar las aceleradoras Hardware que incluyen las tarjetas gráficas actuales, deberemos saber cómo programar en OpenGL o Direct3D (GLiDE o SGL también son válidas para estos fines).

Y como todo en esta vida, podemos usar código propio, o usar alguna de las infinitas librerías que purulan por internet; eso sí, es muy recomendable saber las bases del funcionamiento de dichas librerías para varios fines, como optimizar nuestro código.

Y finalmente, los juegos... realmente no es necesario que todo el trabajo que realizamos esté destinado a un juego, podríamos hacer simuladores u otros programas serios. Pero eso sí, un juego 3D requiere todos y cada uno de los elementos de la programación 3D para llegar a ser algo medianamente real.

¿En qué consiste un motor 3D?

Siempre oímos que el Quake tiene un motor 3D buenisimo o que el Blade tiene el motor más realista que hay. Muchas cosas similares se tienen oído por ahí. Esto demuestra dos cosas:

1º la técnica avanza con el tiempo... pasados un par de años ya nadie habla de los méritos de aquel juego que en su tiempo fue rompedor.

2º los juegos siempre tienen las mejores implementaciones de "realidad virtual", ya que nadie habla de los programas X o Y. También es verdad que los programas profesionales relativamente buenos, necesitan grandes ordenadores para ejecutarse.

Y por último se plantea la pregunta ¿que es un motor?

Igual que la programación 3D, un motor engloba muchas cosas por definición:

- * Presentación (gráficos y imagen).
- * Movimientos (física y colisión de objetos).
- * Sonidos (posiciones, volumen y velocidad, doppler).
- * Organización 3D (mapas, objetos, ...).
- * Interacción (teclado, ratón, ...).

De todas estas cosas, las hay que forman parte del motor propiamente dicho, y otras que ya no tanto. Para saber cuales son, propongo entender QUE HACE UN MOTOR, ya que ¿todos los programas 3D incorporan motor? En realidad, hay programas de visualización 3D de objetos muy sencillos que también son motores gráficos, pero, esencialmente, un motor gráfico, más que encargarse de visualizar directamente se encarga de lograr meter un mundo virtual dentro de la pantalla.

Para ello se compone de 2 partes:

- * Organización del espacio: se encarga de lograr introducir la información gráfica adecuada al render, esto es, si tenemos un mapa hecho en 3D Studio, pues eliminará aquellas partes que no se van a ver, disminuye la calidad de aquellas que se encuentran lejos, ...
- * Física: crea la sensación de movimiento, gravedad y solidez de los objetos (colisión entre objetos).

En esencia, un motor como el del Quake es bueno no sólo por renderizar polígonos rápido -esto es lo de menos- sino, por poder representar el entorno que ve el jugador con el menor número posible de polígonos para que no se pierda calidad y aumente la velocidad.

TEMA 2

-Primitivas básicas-

Se entiende por una primitiva una operación sencilla a partir de un conjunto de las cuales podemos realizar una operación más compleja. En gráficos, las primitivas más básicas son los puntos, pero para lograr una cierta eficiencia, se consideran también primitivas a las líneas, los polígonos -los triángulos y quads incluidos- y los decals también.

Líneas

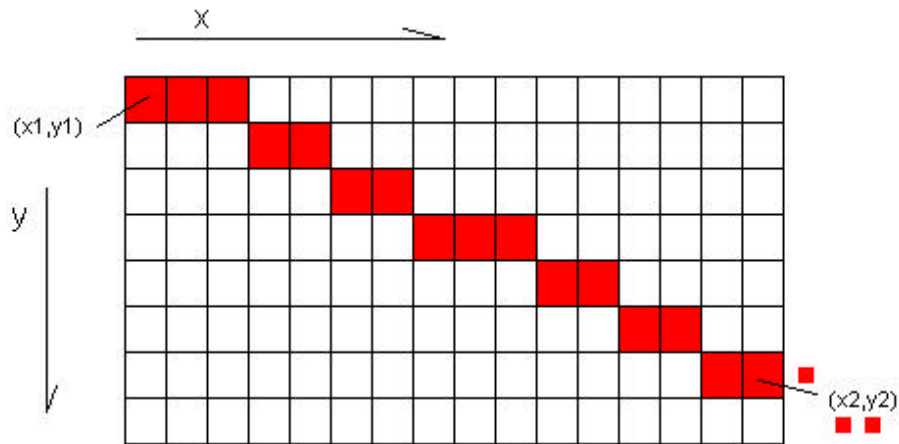
Pese a no valer de mucho, las líneas son la parte básica y el fundamento del resto de las primitivas. El saber realizar una línea correctamente implica directamente en el resto de las primitivas. ¿Porqué? Pues sencillo, tanto un polígono, como un quad, como un decal usan INTERPOLACIÓN LINEAL. INCLUSO ES ÚTIL PARA TEXTURIZAR polígonos. Y eso es justamente UNA LÍNEA. Veremos más en el triángulo y quad, y también en el decal.

El algoritmo más óptimo de todos es el llamado de Bresenham -que viene descrito en el PC Interno, PC al límite y otros libros de la biblioteca-. Fundamentalmente se parte de varios contadores, de manera que:

El pseudocódigo es:

```
dx = x2 - x1
dy = y2 - y1
cx = dy
y = y1
repetir mientras y < y2
    si (cx < 0) entonces
        cx = cx + dy
        x = x + 1
    fin-si
    cx = cx - dx
    y = y + 1
fin-repetir
```

Y un ejemplo:



Se aplicaría así:

- (x=1) $16-7=9$
- (x=2) $9-7=2$
- (x=3) $2-7=-5$; $-5+16=11$; bajar un punto
- (x=4) $11-7=4$
- (x=5) $4-7=-3$; $-3+16=13$; bajar un punto
- (x=6) $13-7=6$
- (x=7) $6-7=-1$; $-1+16=15$; bajar un punto
- (x=8) $15-7=8$
- (x=9) $8-7=1$
- (x=10) $1-7=-6$; $-6+16=10$; bajar un punto
- (x=11) $10-7=3$
- (x=12) $3-7=-4$; $-4+16=12$; bajar un punto
- (x=13) $12-7=5$
- (x=14) $5-7=-2$; $-2+16=14$; bajar un punto
- (x=15) $14-7=7$
- (x=16) $7-7=0$; fin, la siguiente interacción sería:
- (x=17 ** ya no se usa **) $0-7=-7$; $-7+16=9$; bajar un punto

Y pese a que sea sencillo, cómo se observa, es el fundamento del resto de las primitivas, como ahora se verá:

Polígonos

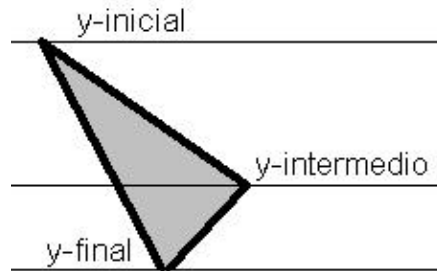
Para trazar un polígono se pueden seguir varios algoritmos, pero todos **por lo menos los que yo he visto** pasan por primero trazar las rectas (con el algoritmo antes dado) y luego rellenar usando las rectas como si fuesen interruptores. Yo propongo lo siguiente:

1) se trazan todos los lados del polígono (para ello se usan unos ARRAYS de tantos elementos como pixels de altura tenga la pantalla, y se usarán tantos ARRAYS como aproximadamente lados/2 tenga el polígono como máximo).

2) se va recorriendo todas las líneas horizontales de la pantalla. Ordenando los valores X de los arrays y posteriormente trazando líneas entre la coordenada del primer array y la del segundo, de la del tercero a la del cuarto, de la del quinto a la del sexto... podeis usar los patrones que más os gusten.

Optimización para el triángulo

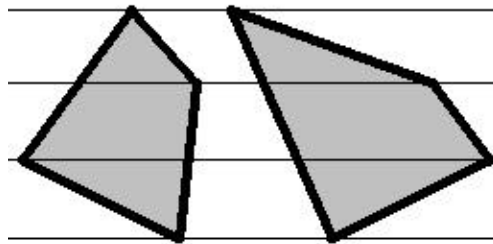
Por motivos obvios -para Texturizar y acelerar- se usan algoritmos específicos para triángulos. Para ello se usa exactamente el mismo sistema que para un polígono, salvo que se sabe que hay dos ARRAYs únicamente y se sabe de antemano que un triángulo va a consistir en una línea desde el y-inicial al y-final y dos líneas desde el y-inicial al y-intermedio y desde el y-intermedio al y-final...



Esto nos permite hacer un código especializado para este caso.

Optimización para el quad

Para los polígonos de 4 lados pasa algo similar, si son convexos:



Y además menciono aquí que permiten ciertas "optimizaciones" en calidad para la texturación lineal, que permiten una apariencia más correcta.

Decals

3/4 de lo mismo, se usa el algoritmo de bresenham, pero en este caso para escalar. Esto es, para un lado dado X, usamos Y-inicial=0 Y-final=N-1 (tamaño del SPRITE) y X-inicial y X-final las coordenadas en pantalla, y lo mismo ambos lados:



Gracias a las líneas estamos INTERPOLANDO linealmente entre los puntos.

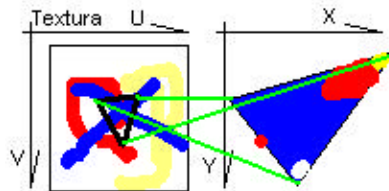
TEMA 3

-Primitivas avanzadas-

Texturación

Vista la línea, cómo se ha demostrado, es cuestión de organizarse el lograr escalar algo. Y precisamente para texturar tenemos que escalar o interpolar **si no utilizamos corrección de perspectiva** las llamadas coordenadas de textura (U,V).

Gráficamente es fácil de entender:



Si se observan las líneas verdes, se verá que lo que se hace al texturar el polígono es convertir las coordenadas desde el espacio (U,V) al (X,Y) del polígono. Podría hacerse usando una matriz de cambio de espacio vectorial, cómo se dio en Álgebra, pero es más sencillo, dado que se conocen las (U,V) de todos los vértices, interpolar a lo largo de los tres lados los valores de (U,V) y posteriormente interpolar línea a línea horizontal de pantalla los valores de los lados, obteniendo así un polígono perfectamente texturado.

Por cierto, la diferencia con este método entre hacer BILINEAR o MOST-NEAR (sin filtro) es el realizar con U,V un trabajo de interpolación entre los colores en función de los decimales de las posiciones en el espacio de U,V.

Sombreado Goraud y Phong

Es más sencillo que texturizar un polígono, si a cada vértice del polígono le damos una LUMINANCIA, e interpolamos los valores de manera análoga a U o V, tendremos un sombreado suavizado.

El que el suavizado que demos sea PHONG o GORAUD depende de los valores de luminancia en sí. Veamos, los valores de luminancia se obtienen mediante la técnica del producto escalar entre el vector normal al un triángulo y el vector triángulo-iluminación:



Así obtendremos la cantidad de luz que absorbe una cara dada (así obtendremos sombreado FLAT). Para obtener la iluminancia de cada vértice promediaremos las iluminancias de todas las caras adyacentes **o bien promediaremos los vectores normales y en lugar de usar la normal de la cara usaremos la normal del vértice**. Así obtendremos sombreado GORAUD. Si además para los vectores que son muy similares a los de iluminación-cara aumentamos el valor de la luminancia mucho más (siguiendo un factor exponencial) obtendremos resplandores y sombreado PHONG.

Corrección de Perspectiva

Esto ya es entrar en caliente... los algoritmos de corrección de perspectiva son varios, y buscan reconstruir en la textura, el espacio original. Hay 3 algoritmos básicos: el primero es hacer el camino inverso en las ecuaciones de proyección 2D/3D, otro añade un factor parabólico al algoritmo estandar lineal, y otro mezcla el algoritmo lineal y el anterior (este último es el que usan el Quake y otros muchos programas y juegos 3D).

TEMA 4

-Entornos 3D-

En este tema se tratarán de explicar los principales sistemas de proyección de coordenadas 3D sobre la pantalla -una superficie 2D-. El escoger una u otra depende de los resultados que deseemos.

Existen muchísimas técnicas para realizar cada una de las proyecciones, y algunas orientadas a objetivos concretos (polígonos, voxel, ...) pero todas se basan en aplicar una transformación de (x,y,z) en (x,y) .

Personalmente distingo 3 tipos: cartesiana, cilíndrica y esférica. La primera es la habitual y la que son capaces de usar las aceleradoras, usan los principales juegos 3D,... y permite 2 estilos: distancia Z y distancia cuadrática.

La cilíndrica es una proyección diseñada para acelerar ciertas aplicaciones como voxel, pero también se puede usar con polígonos. Se basa en hacer que la coordenada X de la pantalla sea un ángulo horizontal. Sin embargo se crea un efecto visual un tanto raro, que se soluciona reasignando las coordenadas X, con lo que volveremos al mismo efecto que causaría una proyección cartesiana. La limitación suele estar en la Y (porque nos obliga a usar una perspectiva vertical similar al Duke3D) por lo que se suele limitar la movilidad en la Y.

La esférica es hacer coincidir a X -de la pantalla- con el ángulo horizontal y a Y con el vertical. Este tipo de proyección tiene aplicaciones limitadas, pero existen.

Ya que la proyección más sencilla y más usada es la cartesiana, será la que me limite a discutir: para ello expondré las fórmulas de proyección.

Lo primero es establecer (X,Y,Z) , el sistema de referencia. Usaré distancia Z -para usar distancia cuadrática basta usar distancia = Raiz ($X^2 + Y^2 + Z^2$).

Y he aquí el pseudocódigo de una función usada para proyectar:

```
[X,Y]=Proyecta[x,y,z]
  d=z
  X=x/d
  Y=y/d
fin
```

¿Sencillo verdad? Pues no lo es tanto, ¿Qué pasa si $Z \leq 0$? Pues que obtendremos valores incongruentes. Ante este problema hay tres soluciones:

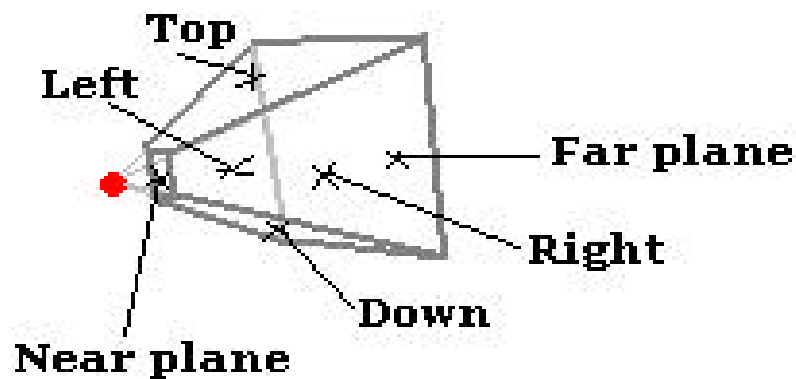
1) Hacer continua a d: para ello se usará un cálculo similar a este...

```
limite=0'1
si z>limite entonces d=z
si z<=limite entonces d=1/((1/limite)-z)
```

El problema generado en este caso es una pérdida de rendimiento y una posición de cámara virtualmente más atrás de la posición (0,0,0).

2) Ignorar aquellas primitivas que posean uno o más vértices por debajo de un límite, 0'01, por ejemplo. En este caso desaparecerán polígonos enteros y los polígonos grandes crearán malos efectos.

3) Frustum Clipping... es la solución ideal, pero más costosa de programar. Se trata de cortar los polígonos, mientras aún son 3D con un NEAR-PLANE, situado inmediatamente delante de la cámara. También se puede cortar con un FAR-PLANE y con los lados de una base de una pirámide cuyo vértice superior es la cámara. Esta imagen lo explica bastante bien:



La idea es que si cortamos un polígono con los planos Frustum, ya no tendremos que preocuparnos de que esté fuera de ángulo de cámara o pueda estar demasiado cerca de la cámara, ya que sólo dibujaremos lo que esté dentro del Frustum. A cambio perderemos tiempo en el proceso de cortar cada uno de los polígonos con el volumen. Y ahora es cuando entran en juego los engines de organización del espacio 3D, ya que muchas veces permiten realizar simplificaciones muy drásticas (BSP por ejemplo).

TEMA 5

-OpenGL, Direct3D y otras librerías-

Y hemos llegado al cúlmen del temario. Si en los temas anteriores se discutían los elementos básicos de un renderizador, a partir de ahora vamos a suponer que ya no vamos a ser nosotros quienes nos encargamos de realizar semejante tarea. Las librerías de OpenGL y Direct3D se diseñaron al propósito de quien no quería machacarse la cabeza con lo anterior -pese a que conviene tener algo de idea del tema para entender muchas de las cosas que contienen-. Sin embargo, el propósito de diseñar OpenGL o Direct3D -incluso GLiDE o SGI, Phigs o otras muchas librerías- va más allá... se trata de estandarizar los accesos a los servicios 3D para introducir un nuevo concepto... la aceleración 3D por Hardware. En realidad de nuevo no tiene nada, ya que OpenGL se diseñó para máquinas que eran TODAS ELLAS en su conjunto una enorme aceleradora 3D, si, hablo de las Silicon Graphics y otras muchas estaciones de trabajo. Sin embargo, hasta hace unos 5 años no llegó al entorno doméstico el concepto de aceleradora. Las primeras fueron las 3DFX y las PowerVR -siendo las primeras las más famosas dada su facilidad de programación mediante GLiDE-. Estas tarjetas eran simples y de bajos recursos -4Mb. de RAM de Video- e incluso se obtenían por separado de las tarjetas de video -que también eran aceleradoras, pero de mucho menos calibre... S3 ViRGE, ...-. Se puede decir que la primera generación de tarjetas había salido a la luz. La segunda generación no se hizo derrogar más de un año... impresionantes tarjetas de hasta 16 Mb. de RAM de Video, que ya iban en comunión con la tarjeta gráfica. Entre ellas, la más famosa es la Banshee Voodoo II, con unas características realmente buenas. Con estas tarjetas de segunda generación ya se podía empezar a pensar en juegos y programas más potentes, pero aún tenían sus limitaciones. La última generación de tarjetas salió a la venta no mucho después. Se trata de la tercera Voodoo y similares. Ahora ya se podía decir que tratamos con tarjetas muy potentes, que alcanzan rendimientos muy altos y permiten mejorar el aspecto de los entornos virtuales.

Por último salió la generación de las TNT (Twin Pixel Transfer) y TNT2, hasta la GeForce2, que se caracterizan por poder mezclar más de una textura en una sola operación de render -una pasada como se suele decir-. La última generación es la GeForce3, que apenas lleva a la venta unos meses -esto lo estoy escribiendo a fecha de Noviembre de 2001-, y se caracteriza por realizar filtrados y efectos sobre las texturas y las imágenes finales que pueden lograr efectos que antes solo eran posibles en las Silicon Graphics en tiempo real -alguien me comentó que podían renderizar la película de Final Fantasy en tiempo real-.

¿Y para que todo esto? Además de para aquellos que no estén puestos en el tema, para decir que, actualmente se pueden renderizar -sin ningún tipo de caché de polígonos- aproximadamente 100000 polígonos en una tarjeta de tercera generación con interfaz AGP 2x/4x o 10000 en una con interfaz PCI (yo tengo una TNT2 conectada a un Pentium 200 a través de un puerto PCI, y le importa poco la calidad o resolución, lo que realmente importa es el número de polígonos, dado el cuello de botella que se forma en el puente PCI, ya que a la TNT2 no se le puede poner a máximo rendimiento, excepto que se usen cachés).

Este tema, como observareis no es para explicar OpenGL o Direct3D, sino para comentar algunos aspectos de ambas API's -al final explicaré la ardua tarea de iniciar Direct3D, que es lo que le brindó tan pocos amigos-. Os puedo decir que OpenGL es más fácil de iniciar y poner a funcionar en modo básico que Direct3D, a cambio Direct3D es más sencillo de utilizar una vez que se ha iniciado. También, que ni uno ni otro puede competir con las facilidades de GLiDE, que solo necesita un par de instrucciones para inicializar el conjunto (mirad <http://www.voodooextreme.com/glide3tutorial/>).

Para iniciar OpenGL o Direct3D, la tarea es un poco más ardua -con OpenGL, usando las librerías independientes del sistema se simplifica mucho la cosas, hasta alcanzar a GLiDE, pero luego es un poco más complejo de usar para renderizar (en cualquier caso mirad <http://nehe.gamedev.net/> para más información)-. Me voy a centrar en Direct3D, ya que últimamente es el API más usado, dada su compatibilidad. Para ejecutar esto se necesitan las cabeceras de las DirectX 5 en adelante (Dx 5, 5.1, 6, 6.1, 7, 8 o 8.1, ya que supongo que no tendréis las Dx9, además que no probé en ellas el ejemplo, pero debería de ir igual). Hay dos maneras de iniciar

las Surfaces DirectDraw, de una vez o por partes, ahora voy a iniciarlas de golpe, con triple buffer, trilinear filter y sin Z-Buffer. Mirad:

```
dds:=nil;  
ddsback:=nil;  
d3do:=nil;
```

Primero se limpian los objetos DirectX

```
InitAblFullscreen('Direct3D Sample',video800x600);
```

Luego se inicializa la ventana a pantalla completa (yo lo tengo todo metido en esta función para ganar tiempo, pero finalmente esta función ejecuta:

```
CreateWindow(...)  
CreateDirectDraw(...)  
dd.setCooperativeLevel(hWnd,DDSCCL_EXCLUSIVE or DDSCCL_FULLSCREEN)  
dd.setVideoMode(800,600,16)
```

```
writeln(logfile,'Window & DirectDraw created');  
TEXTURESIZE:=128;  
SCREENWIDTH:=800;  
SCREENHEIGHT:=600;  
zeromemory(@ddsd,sizeof(ddsd));  
ddsd.dwSize:=sizeof(ddsd);  
ddsd.dwFlags:=DDSD_CAPS or DDSD_BACKBUFFERCOUNT;// or DDSD_ZBUFFERBITDEPTH;  
ddsd.ddscaps.dwCaps:=DDSCAPS_PRIMARYSURFACE or DDSCAPS_FLIP or  
DDSCAPS_COMPLEX or DDSCAPS_3DDEVICE or DDSCAPS_VIDEMEMORY;  
ddsd.dwBackBufferCount:=2;  
if dd.CreateSurface(ddsd,dds,nil)<>DD_OK then exit;  
writeln(logfile,'Screen Primary Surface Created (with triple-buffer)');
```

Ahora hay que obtener el objeto de la BackSurface -ya que no la creamos aparte-.

```
zeromemory(@ddscaps,sizeof(ddscaps));  
ddscaps.dwCaps:=DDSCAPS_BACKBUFFER;  
if dds.GetAttachedSurface(ddscaps,ddsback)<>DD_OK then exit;  
writeln(logfile,'BackBuffer obtained (no Z-Buffer used)');
```

Y empezamos con Direct3D, hay que despertar a la aceleradora...

```
if dd.QueryInterface(IID_IDirect3D2,d3do)<>S_OK then exit;  
writeln(logfile,'Direct3D created');
```

Creamos un device ligado al backbuffer -con IID_IDirect3DHALDevice lo que logramos es usar el dispositivo de aceleración por defecto-.

```
d3do.createdevice(IID_IDirect3DHALDevice,ddsback,d3ddev);  
writeln(logfile,'Created HAL device');
```

Ahora hay que crear el Viewport, configurarlo y ligarlo.

```
zeromemory(@viewcfg,sizeof(viewcfg));  
viewcfg.dwSize:=sizeof(viewcfg);  
viewcfg.dwWidth:=SCREENWIDTH;  
viewcfg.dwHeight:=SCREENHEIGHT;  
viewcfg.dvScaleX:=SCREENWIDTH/2.0*48/64;  
viewcfg.dvScaleY:=SCREENHEIGHT/2.0;  
viewcfg.dvMaxX:=D3DVAL(1.0);  
viewcfg.dvMaxY:=D3DVAL(1.0);  
viewcfg.dvMinZ:=D3DVAL(0.0);  
viewcfg.dvMaxZ:=D3DVAL(1.0);  
d3do.createviewport(d3dview,nil);  
d3ddev.addviewport(d3dview);  
d3dview.setviewport(viewcfg);  
d3ddev.setcurrentviewport(d3dview);  
writeln(logfile,'ViewPort OK');
```

Fijar las matrices de proyección, mundo y visión (que realizaran la conversión 3D/2D).

```

mat_world:=IdentityMatrix;
mat_view:=ViewMatrix(D3DVECTOR(0,0,0),D3DVECTOR(0,0,1),D3DVECTOR(0,1,0),0);
mat_proyect:=ProjectionMatrix(1.01,2000.0,60*PI/180);
d3ddev.SetTransform(D3DTRANSFORMSTATE_WORLD, mat_world);
d3ddev.SetTransform(D3DTRANSFORMSTATE_VIEW, mat_view);
d3ddev.SetTransform(D3DTRANSFORMSTATE_PROJECTION, mat_proyect);

```

Y Fijar el estado del renderer...

```

d3ddev.SetRenderState(D3DRENDERSTATE_ANTI_ALIAS,INTEGER(D3DANTI_ALIAS_SORTINDEPENDENT));
d3ddev.SetRenderState(D3DRENDERSTATE_TEXTUREADDRESS,INTEGER(D3DTEXTUREADDRESS_WRAP));
d3ddev.SetRenderState(D3DRENDERSTATE_TEXTUREPERSPECTIVE,INTEGER(TRUE));
d3ddev.SetRenderState(D3DRENDERSTATE_FILLMODE,INTEGER(D3DFILL_SOLID));
d3ddev.SetRenderState(D3DRENDERSTATE_SHADEMODE,INTEGER(D3DSHADE_GOURAUD));
d3ddev.SetRenderState(D3DRENDERSTATE_MONOENABLE,INTEGER(FALSE));
d3ddev.SetRenderState(D3DRENDERSTATE_TEXTUREMAG,INTEGER(D3DFILTER_LINEAR));
d3ddev.SetRenderState(D3DRENDERSTATE_TEXTUREMIN,INTEGER(D3DFILTER_LINEAR_MIPMAP_LINEAR));
d3ddev.SetRenderState(D3DRENDERSTATE_CULLMODE,INTEGER(D3DCULL_NONE));
d3ddev.SetRenderState(D3DRENDERSTATE_DITHERENABLE,INTEGER(TRUE));
d3ddev.SetRenderState(D3DRENDERSTATE_SPECULARENABLE,INTEGER(TRUE));
d3ddev.SetRenderState(D3DRENDERSTATE_SUBPIXEL,INTEGER(TRUE));

writeln(logfile,'Renderer Status Updated');

```

Crear la textura...

```

zeromemory(@ddsd,sizeof(ddsd));
ddsd.dwSize:=sizeof(ddsd);
ddsd.dwFlags:=DDSD_CAPS or DDSD_WIDTH or DDSD_HEIGHT;
ddsd.dwWidth:=TEXTURESIZE;
ddsd.dwHeight:=TEXTURESIZE;
ddsd.ddscaps.dwCaps:=DDSCAPS_TEXTURE or DDSCAPS_VIDEMEMORY;
if dd.CreateSurface(ddsd,ddstexture,nil)<>DD_OK then exit;
writeln(logfile,'Texture Surface OK');
filltexture;
ddstexture.queryinterface(IID_IDirect3DTexture2,d3dtex);
d3dtex.getHandle(d3ddev,htex);

```

Y fijarla en uso:

```

d3ddev.SetRenderState(D3DRENDERSTATE_TEXTUREHANDLE, INTEGER(htex));
writeln(logfile,'Texture OK');

```

Yo creo que ya se ha comprendido porque Direct3D es difícil, o al menos costosa de inicializar. Lo bueno es que una vez que funciona (si lo probais en varios ordenadores vereis que no siempre funciona, a veces se cuelga, no arranca o se ve mal) ya no hay que volverse a preocupar de esta árdua tarea. Os recomiendo que mireis algunas páginas del tema:

<http://francis.dupont.free.fr/coindev/english/dd.htm>
<http://www.microsoft.com/mspress/books/WW/sampchap/2831.asp>

Y como andan a cambiar las direcciones de otras páginas, os digo los nombres y los meteis en el Google (<http://www.google.com/>):

- The DirectX Experience.
- CodeMaster.
- Programmer's Lair
- Delphi Jedi Project.

Y muchas más páginas.

TEMA 6

-Introducción a las técnicas de organización espacial- <Motores 3D>

Y, finalmente, a modo de introducción contaré un poco de las bases de organización espacial. Supongo que os habreis dado cuenta de que con un limite de unos 2500~5000 poligonos para un buen rendimiento, el Quake pone en pantalla escenarios enormes, y ya no digamos el Quake 2. ¿Cómo es que se pueden permitir el lujo de usar mapas de más de 1000000 de poligonos si su limite real (y el de cualquier renderer software para un ordenador medio) es de aproximadamente 10000 poligonos? Pues sencillo, renderizando sólo los necesarios en cada momento. Esto, es, los ocultos y lejanos no se renderizan, o se usan menos poligonos o muchas otras técnicas.

BSP

En el caso del Quake y sus sucesores se usa una técnica muy famosa -gracias al propio Quake- llamada BSP. Sin duda decir BSP implica un cierto tabú entre ciertos circulos, dada la aparente complejidad subyacente. Sin embargo, lo que hace complejo al Quake no es precisamente el BSP, sino las optimizaciones que incorpora, y que sólo John Carmack puede decir en que consisten...

El BSP es un algoritmo muy sencillo de división espacial. Se basa en usar un plano nodal que divide todo el espacio en 2 subespacios (si algún elemento está en ambos lados del plano, se divide en dos elementos). Una vez tengamos los dos subespacios creados, volvemos a dividirlos, y así recursivamente hasta que tan sólo quede 1 elemento por subespacio. Con esto hemos construido el llamado árbol BSP. Por cierto, sería interesante que el número de elementos en cada subespacio fuera aproximadamente igual para no despinfarrar memoria.

Que yo conozca, los arboles BSP se pueden realizar mediante elementos enlazados con punteros o con un HEAP lineal, usando el hecho de que el primer elemento de cada nivel siempre tiene un índice de la forma 2^n con n entero.

Para renderizar un BSP se usa -normalmente- un algoritmo recursivo. Podremos usar el hecho de que si un plano no corta al Frustum, almenos uno de los dos subespacios se puede ignorar.

División espacial ordenada

Frente al BSP, que divide el espacio de forma arbitraria, existen técnicas más sencillas de dividir el espacio en zonas geometricamente estables. Por ejemplo, crear un grid 2D (división en prismas) o crear uno 3D (división en prismas).

En ámbos casos (los más usados) obtendremos un universo dividido. Podremos aplicar dos tipos de técnicas: ocultación por profundidad (usando FOG) o bien Polychop.

Ocultación en profundidad se logra reduciendo la distancia al FAR PLANE del frustum, creando un efecto de niebla y limitando la profundidad de la selección de elementos del espacio (cubos o prismas).

El polychop lo que hace es reducir el número de polígonos para objetos lejanos.

Polychop

Ya lo he nombrado, pero tiene otras muchas aplicaciones: aumentar el rendimiento, si se aplica a todos los objetos, para las entidades, también se puede aplicar en mayor o menor medida, dado que el número de polígonos es vital para la velocidad, tanto para software acelerado, como para no acelerado, se puede ver la importancia de la reducción dinámica de polígonos.

El Polychop es visible en muchos juegos al reducir la calidad de visualización -que ahora recuerde, el Undying es un buen ejemplo-.

Lo que hace el algoritmo de reducción es buscar los vértices menos importantes para la malla y eliminarlos. Esto en cada iteración. Al algoritmo se le pasa como parámetro el % de polígonos a dejar. Nótese que lleva un tiempo procesar la malla, y no es reversible, por lo que es recomendable usar una caché con varios niveles de LOD (Level Of Detail) para no andar a ejecutar la reducción demasiado a menudo.

Caché de aceleradora

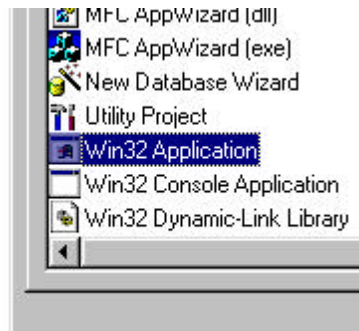
Antes comenté que *...si no se usaba caché, el máximo de poligonos en una tarjeta PCI es aprox. 10000 para un buen rendimiento...* ¿¿a qué me refería con caché?? Pues a las listas, los vértices compilados y otras muchas otras técnicas que ofrecen las API's 3D para no andar a pasar parametros cada vez que queremos generar una escena. En efecto, OpenGL, por ejemplo ofrece listas. Os puedo decir que una esfera de 50000 poligonos se ve a 5 fps sin listas y de 15 a 60 fps con listas (si no se modifican las listas). Por ello se observa la importancia de usar apropiadamente las cachés para escenarios que cambian pocas veces respecto al ritmo de imágenes.

TEMA 7

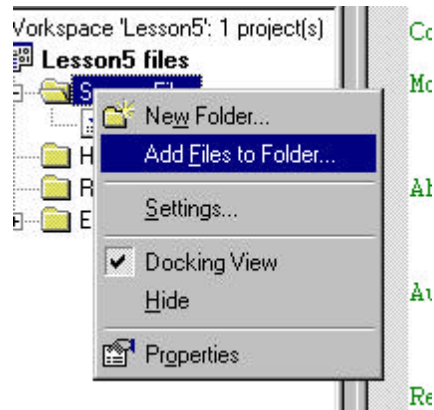
-Sesión práctica-

Para probar el siguiente código fuente si quereis ir probando algo (en el cursillo probaré uno un poco más complejo); necesitareis tener el Visual C con las cabeceras de OpenGL. Para hacerlo funcionar haced lo siguiente:

1) Cread un proyecto de Aplicación Win32 VACIO.

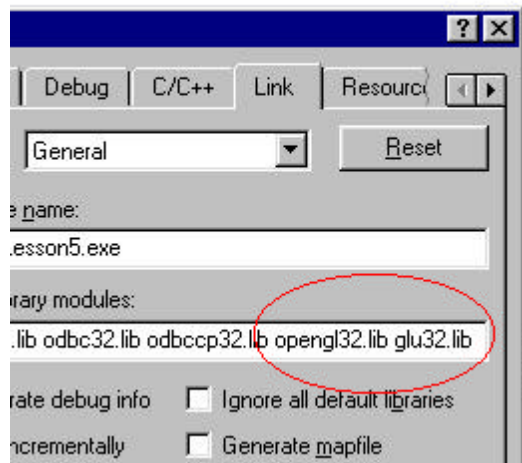


2) Añadid un fichero nuevo en la sección de códigos fuente.



3) Pulsad ALT+F7 y el el Tab de Link, en el campo Object/Library Modules añadid al final:

OpenGL32.lib Glu32.lib



4) Pegad este código fuente en el fichero .Cpp vacio:

```
/*
 *          This Code Was Created By Jeff Molofee 2000
 *          A HUGE Thanks To Fredric Echols For Cleaning Up
 *          And Optimizing The Base Code, Making It More Flexible!
 *          If You've Found This Code Useful, Please Let Me Know.
 *          Visit My Site At nehe.gamedev.net
 */

#include <windows.h>           // Header File For Windows
#include <gl\gl.h>             // Header File For The OpenGL32 Library
#include <gl\glu.h>            // Header File For The GLu32 Library
#include <gl\glaux.h>          // Header File For The Glaux Library

HDC          hDC=NULL;        // Private GDI Device Context
HGLRC        hRC=NULL;        // Permanent Rendering Context
HWND         hWnd=NULL;       // Holds Our Window Handle
HINSTANCE    hInstance;       // Holds The Instance Of The Application

bool         keys[256];        // Array Used For The Keyboard Routine
bool         active=TRUE;      // Window Active Flag Set To TRUE By Default
bool         fullscreen=TRUE; // Fullscreen Flag Set To Fullscreen Mode By Default

GLfloat      rtri;             // Angle For The Triangle ( NEW )
GLfloat      rquad;            // Angle For The Quad ( NEW )

LRESULT      CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Declaration For WndProc

GLvoid ReSizeGLScene(GLsizei width, GLsizei height) // Resize And Initialize The GL Window
{
    if (height==0) // Prevent A Divide By Zero By
    {
        height=1; // Making Height Equal One
    }

    glViewport(0,0,width,height); // Reset The Current
    Viewport

    glMatrixMode(GL_PROJECTION); // Select The
    Projection Matrix
    glLoadIdentity(); // Reset
    The Projection Matrix
```

```

// Calculate The Aspect Ratio Of The Window
gluPerspective(45.0f,(GLfloat)width/(GLfloat)height,0.1f,100.0f);

glMatrixMode(GL_MODELVIEW); // Select
The Modelview Matrix
glLoadIdentity(); // Reset
The Modelview Matrix
}

int InitGL(GLvoid) // All Setup
For OpenGL Goes Here
{
    glShadeModel(GL_SMOOTH); // Enable
Smooth Shading
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Black Background
    glClearDepth(1.0f); //
Depth Buffer Setup
    glEnable(GL_DEPTH_TEST); // Enables
Depth Testing
    glDepthFunc(GL_EQUAL); // The Type
Of Depth Testing To Do
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Really Nice Perspective Calculations
    return TRUE; //
Initialization Went OK
}

int DrawGLScene(GLvoid) // Here's
Where We Do All The Drawing
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Screen And Depth Buffer
    glLoadIdentity(); // Reset
The Current Modelview Matrix
    glTranslatef(-1.5f,0.0f,-6.0f); // Move Left 1.5
Units And Into The Screen 6.0
    glRotatef(rtri,0.0f,1.0f,0.0f); // Rotate The Triangle On The
Y axis ( NEW )
    glBegin(GL_TRIANGLES); // Start
Drawing A Triangle
        glColor3f(1.0f,0.0f,0.0f); // Red
        glVertex3f( 0.0f, 1.0f, 0.0f); // Top Of Triangle (Front)
        glColor3f(0.0f,1.0f,0.0f); // Green
        glVertex3f(-1.0f,-1.0f, 1.0f); // Left Of Triangle
(Front)
        glColor3f(0.0f,0.0f,1.0f); // Blue
        glVertex3f( 1.0f,-1.0f, 1.0f); // Right Of Triangle (Front)
        glColor3f(1.0f,0.0f,0.0f); // Red
        glVertex3f( 0.0f, 1.0f, 0.0f); // Top Of Triangle (Right)
        glColor3f(0.0f,0.0f,1.0f); // Blue
        glVertex3f( 1.0f,-1.0f, 1.0f); // Left Of Triangle (Right)
        glColor3f(0.0f,1.0f,0.0f); // Green
        glVertex3f( 1.0f,-1.0f, -1.0f); // Right Of Triangle
(Right)
        glColor3f(1.0f,0.0f,0.0f); // Red
        glVertex3f( 0.0f, 1.0f, 0.0f); // Top Of Triangle (Back)
        glColor3f(0.0f,1.0f,0.0f); // Green
        glVertex3f( 1.0f,-1.0f, -1.0f); // Left Of Triangle
(Back)
        glColor3f(0.0f,0.0f,1.0f); // Blue
        glVertex3f(-1.0f,-1.0f, -1.0f); // Right Of Triangle
(Back)
        glColor3f(1.0f,0.0f,0.0f); // Red
        glVertex3f( 0.0f, 1.0f, 0.0f); // Top Of Triangle (Left)
        glColor3f(0.0f,0.0f,1.0f); // Blue
        glVertex3f(-1.0f,-1.0f,-1.0f); // Left Of Triangle
(Left)
        glColor3f(0.0f,1.0f,0.0f); // Green
        glVertex3f(-1.0f,-1.0f, 1.0f); // Right Of Triangle
(Left)
    glEnd(); //
Done Drawing The Pyramid

```

```

        glLoadIdentity(); // Reset
The Current Modelview Matrix
        glTranslatef(1.5f,0.0f,-7.0f); // Move Right 1.5
Units And Into The Screen 7.0
        glRotatef(rquad,1.0f,1.0f,1.0f); // Rotate The Quad On The X
axis ( NEW )
        glBegin(GL_QUADS); //
Draw A Quad
        glColor3f(0.0f,1.0f,0.0f); // Set The Color To
Blue
        glVertex3f( 1.0f, 1.0f,-1.0f); // Top Right Of The Quad (Top)
        glVertex3f(-1.0f, 1.0f,-1.0f); // Top Left Of The
Quad (Top)
        glVertex3f(-1.0f, 1.0f, 1.0f); // Bottom Left Of The Quad
(Top)
        glVertex3f( 1.0f, 1.0f, 1.0f); // Bottom Right Of The Quad
(Top)
        glColor3f(1.0f,0.5f,0.0f); // Set The Color To
Orange
        glVertex3f( 1.0f,-1.0f, 1.0f); // Top Right Of The Quad
(Bottom)
        glVertex3f(-1.0f,-1.0f, 1.0f); // Top Left Of The
Quad (Bottom)
        glVertex3f(-1.0f,-1.0f,-1.0f); // Bottom Left Of The
Quad (Bottom)
        glVertex3f( 1.0f,-1.0f,-1.0f); // Bottom Right Of
The Quad (Bottom)
        glColor3f(1.0f,0.0f,0.0f); // Set The Color To
Red
        glVertex3f( 1.0f, 1.0f, 1.0f); // Top Right Of The Quad
(Front)
        glVertex3f(-1.0f, 1.0f, 1.0f); // Top Left Of The Quad (Front)
        glVertex3f(-1.0f,-1.0f, 1.0f); // Bottom Left Of The
Quad (Front)
        glVertex3f( 1.0f,-1.0f, 1.0f); // Bottom Right Of The Quad
(Front)
        glColor3f(1.0f,1.0f,0.0f); // Set The Color To
Yellow
        glVertex3f( 1.0f,-1.0f,-1.0f); // Top Right Of The
Quad (Back)
        glVertex3f(-1.0f,-1.0f,-1.0f); // Top Left Of The
Quad (Back)
        glVertex3f(-1.0f, 1.0f,-1.0f); // Bottom Left Of The
Quad (Back)
        glVertex3f( 1.0f, 1.0f,-1.0f); // Bottom Right Of The Quad
(Back)
        glColor3f(0.0f,0.0f,1.0f); // Set The Color To
Blue
        glVertex3f(-1.0f, 1.0f, 1.0f); // Top Right Of The Quad (Left)
        glVertex3f(-1.0f, 1.0f,-1.0f); // Top Left Of The
Quad (Left)
        glVertex3f(-1.0f,-1.0f,-1.0f); // Bottom Left Of The
Quad (Left)
        glVertex3f(-1.0f,-1.0f, 1.0f); // Bottom Right Of
The Quad (Left)
        glColor3f(1.0f,0.0f,1.0f); // Set The Color To
Violet
        glVertex3f( 1.0f, 1.0f,-1.0f); // Top Right Of The Quad
(Right)
        glVertex3f( 1.0f, 1.0f, 1.0f); // Top Left Of The Quad (Right)
        glVertex3f( 1.0f,-1.0f, 1.0f); // Bottom Left Of The Quad
(Right)
        glVertex3f( 1.0f,-1.0f,-1.0f); // Bottom Right Of
The Quad (Right)
        glEnd(); //
Done Drawing The Quad

        rtri+=0.2f;
        // Increase The Rotation Variable For The Triangle ( NEW )
        rquad-=0.15f; //
Decrease The Rotation Variable For The Quad ( NEW )

```

```

        return TRUE; //
Keep Going
}

GLvoid KillIGLWindow(GLvoid) // Properly
Kill The Window
{
    if (fullscreen) //
Are We In Fullscreen Mode?
    {
        ChangeDisplaySettings(NULL,0); // If So Switch Back
To The Desktop
        ShowCursor(TRUE); //
Show Mouse Pointer
    }

    if (hRC) //
Do We Have A Rendering Context?
    {
        if (!wglMakeCurrent(NULL,NULL)) // Are We Able To
Release The DC And RC Contexts?
        {
            MessageBox(NULL,"Release Of DC And RC Failed.,"SHUTDOWN ERROR",MB_OK |
MB_ICONINFORMATION);
        }

        if (!wglDeleteContext(hRC)) // Are We Able To
Delete The RC?
        {
            MessageBox(NULL,"Release Rendering Context Failed.,"SHUTDOWN ERROR",MB_OK
| MB_ICONINFORMATION);
        }
        hRC=NULL;
        // Set RC To NULL
    }

    if (hDC && !ReleaseDC(hWnd,hDC)) // Are We Able To Release The
DC
    {
        MessageBox(NULL,"Release Device Context Failed.,"SHUTDOWN ERROR",MB_OK |
MB_ICONINFORMATION);
        hDC=NULL;
        // Set DC To NULL
    }

    if (hWnd && !DestroyWindow(hWnd)) // Are We Able To Destroy The
Window?
    {
        MessageBox(NULL,"Could Not Release hWnd.,"SHUTDOWN ERROR",MB_OK |
MB_ICONINFORMATION);
        hWnd=NULL;
        // Set hWnd To NULL
    }

    if (!UnregisterClass("OpenGL",hInstance)) // Are We Able To Unregister Class
    {
        MessageBox(NULL,"Could Not Unregister Class.,"SHUTDOWN ERROR",MB_OK |
MB_ICONINFORMATION);
        hInstance=NULL; //
Set hInstance To NULL
    }
}

/* This Code Creates Our OpenGL Window. Parameters Are: *
* title - Title To Appear At The Top Of The Window *
* width - Width Of The GL Window Or Fullscreen Mode *
* height - Height Of The GL Window Or Fullscreen Mode *
* bits - Number Of Bits To Use For Color (8/16/24/32) *
* fullscreenflag - Use Fullscreen Mode (TRUE) Or Windowed Mode (FALSE) */

BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag)
{

```

```

        GLuint          PixelFormat;                // Holds The Results After Searching For A Match
        WNDCLASS        wc;                        // Windows Class Structure
        DWORD           dwExStyle;                // Window Extended Style
        DWORD           dwStyle;                  // Window Style
        RECT            WindowRect;              // Grabs Rectangle Upper Left / Lower
Right Values
        WindowRect.left=(long)0;                // Set Left Value To 0
        WindowRect.right=(long)width;          // Set Right Value To Requested Width
        WindowRect.top=(long)0;                // Set Top Value To 0
        WindowRect.bottom=(long)height;        // Set Bottom Value To Requested Height

        fullscreen=fullscreenflag;              // Set The Global Fullscreen Flag

        hInstance       = GetModuleHandle(NULL); // Grab An
Instance For Our Window
        wc.style         = CS_HREDRAW | CS_VREDRAW | CS_OWNDC; // Redraw On Size,
And Own DC For Window.
        wc.lpfnWndProc   = (WNDPROC) WndProc; // WndProc
Handles Messages
        wc.cbClsExtra    = 0;
        // No Extra Window Data
        wc.cbWndExtra    = 0;
        // No Extra Window Data
        wc.hInstance     = hInstance; //
Set The Instance
        wc.hIcon         = LoadIcon(NULL, IDI_WINLOGO); // Load The Default
Icon
        wc.hCursor       = LoadCursor(NULL, IDC_ARROW); // Load The
Arrow Pointer
        wc.hbrBackground = NULL; //
No Background Required For GL
        wc.lpszMenuName  = NULL;
        // We Don't Want A Menu
        wc.lpszClassName = "OpenGL"; //
Set The Class Name

        if (!RegisterClass(&wc)) //
Attempt To Register The Window Class
        {
            MessageBox(NULL,"Failed To Register The Window
Class.", "ERROR", MB_OK|MB_ICONEXCLAMATION);
            return FALSE;
            // Return FALSE
        }

        if (fullscreen)
            // Attempt Fullscreen Mode?
        {
            DEVMODE dmScreenSettings;
            // Device Mode
            memset(&dmScreenSettings,0,sizeof(dmScreenSettings)); // Makes Sure Memory's
Cleared
            dmScreenSettings.dmSize=sizeof(dmScreenSettings); // Size Of The Devmode
Structure
            dmScreenSettings.dmPelsWidth  = width; // Selected Screen
Width
            dmScreenSettings.dmPelsHeight = height; // Selected Screen
Height
            dmScreenSettings.dmBitsPerPel = bits; // Selected
Bits Per Pixel
            dmScreenSettings.dmFields=DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;

            // Try To Set Selected Mode And Get Results. NOTE: CDS_FULLSCREEN Gets Rid Of Start Bar.
            if
(ChangeDisplaySettings(&dmScreenSettings,CDS_FULLSCREEN)!=DISP_CHANGE_SUCCESSFUL)
            {
                // If The Mode Fails, Offer Two Options. Quit Or Use Windowed Mode.
                if (MessageBox(NULL,"The Requested Fullscreen Mode Is Not Supported By\nYour
Video Card. Use Windowed Mode Instead?","NeHe GL",MB_YESNO|MB_ICONEXCLAMATION)==IDYES)
                {
                    fullscreen=FALSE; // Windowed Mode Selected. Fullscreen = FALSE
                }
            }

```

```

        else
        {
            // Pop Up A Message Box Letting User Know The Program Is Closing.
            MessageBox(NULL,"Program Will Now
Close.", "ERROR", MB_OK|MB_ICONSTOP);
            return FALSE;
        }
    }

    if (fullscreen)
        // Are We Still In Fullscreen Mode?
    {
        dwExStyle=WS_EX_APPWINDOW;
        // Window Extended Style
        dwStyle=WS_POPUP;
        // Windows Style
        ShowCursor(FALSE);
        // Hide Mouse Pointer
    }
    else
    {
        dwExStyle=WS_EX_APPWINDOW | WS_EX_WINDOWEDGE;           // Window
Extended Style
        dwStyle=WS_OVERLAPPEDWINDOW;
        // Windows Style
    }

    AdjustWindowRectEx(&WindowRect, dwStyle, FALSE, dwExStyle);    // Adjust Window To True
Requested Size

    // Create The Window
    if (!hWnd=CreateWindowEx(    dwExStyle,
// Extended Style For The Window

                                // Class Name
                                "OpenGL",
                                // Window Title
                                title,
                                // Defined Window Style
                                dwStyle |
                                // Required Window Style
                                WS_CLIPSIBLINGS |
                                // Required Window Style
                                WS_CLIPCHILDREN,
                                // Window Position
                                0, 0,
                                Calculate Window Width
                                WindowRect.right-WindowRect.left, //
                                Calculate Window Height
                                WindowRect.bottom-WindowRect.top, //
                                NULL,
                                // No Parent Window
                                NULL,
                                // No Menu
                                hInstance,
                                // Instance
                                NULL)))
        // Dont Pass Anything To WM_CREATE
    {
        KillIGLWindow(); // Reset
        The Display
        MessageBox(NULL,"Window Creation Error.", "ERROR", MB_OK|MB_ICONEXCLAMATION);
        return FALSE; // Return
    }
    FALSE
}

    static PIXELFORMATDESCRIPTOR pfd= // pfd Tells Windows How We
Want Things To Be
    {
        sizeof(PIXELFORMATDESCRIPTOR), // Size Of This Pixel Format
Descriptor

```

```

        1,
        // Version Number
        PFD_DRAW_TO_WINDOW | // Format
Must Support Window
        PFD_SUPPORT_OPENGL | // Format Must
Support OpenGL
        PFD_DOUBLEBUFFER, // Must
Support Double Buffering
        PFD_TYPE_RGBA, // Request
An RGBA Format
        bits, //
Select Our Color Depth
        0, 0, 0, 0, 0, 0, // Color Bits Ignored
        0,
        // No Alpha Buffer
        0,
        // Shift Bit Ignored
        0,
        // No Accumulation Buffer
        0, 0, 0, 0, //
Accumulation Bits Ignored
        16,
        // 16Bit Z-Buffer (Depth Buffer)
        0,
        // No Stencil Buffer
        0,
        // No Auxiliary Buffer
        PFD_MAIN_PLANE, //
Main Drawing Layer
        0,
        // Reserved
        0, 0, 0 //
Layer Masks Ignored
    };

    if (!hDC=GetDC(hWnd)) // Did We Get A
Device Context?
    {
        KillIGLWindow(); // Reset
The Display
        MessageBox(NULL,"Can't Create A GL Device
Context. ","ERROR",MB_OK|MB_ICONEXCLAMATION);
        return FALSE; // Return
FALSE
    }

    if (!(PixelFormat=ChoosePixelFormat(hDC,&pdf)) // Did Windows Find A Matching Pixel Format?
    {
        KillIGLWindow(); // Reset
The Display
        MessageBox(NULL,"Can't Find A Suitable
PixelFormat. ","ERROR",MB_OK|MB_ICONEXCLAMATION);
        return FALSE; // Return
FALSE
    }

    if(!SetPixelFormat(hDC,PixelFormat,&pdf) // Are We Able To Set The Pixel Format?
    {
        KillIGLWindow(); // Reset
The Display
        MessageBox(NULL,"Can't Set The PixelFormat. ","ERROR",MB_OK|MB_ICONEXCLAMATION);
        return FALSE; // Return
FALSE
    }

    if (!hRC=wglCreateContext(hDC)) // Are We Able To Get A Rendering
Context?
    {
        KillIGLWindow(); // Reset
The Display
        MessageBox(NULL,"Can't Create A GL Rendering
Context. ","ERROR",MB_OK|MB_ICONEXCLAMATION);

```

```

        return FALSE; // Return
FALSE
    }

    if(! wglMakeCurrent(hDC,hRC)) // Try To Activate The
Rendering Context
    {
        KillGLWindow(); // Reset
The Display
        MessageBox(NULL,"Can't Activate The GL Rendering
Context. ","ERROR",MB_OK|MB_ICONEXCLAMATION);
        return FALSE; // Return
FALSE
    }

    ShowWindow(hWnd,SW_SHOW); // Show The Window
    SetForegroundWindow(hWnd); // Slightly Higher
Priority
    SetFocus(hWnd); // Sets
Keyboard Focus To The Window
    ReSizeGLScene(width, height); // Set Up Our Perspective GL
Screen

    if (!InitGL()) // Initialize
Our Newly Created GL Window
    {
        KillGLWindow(); // Reset
The Display
        MessageBox(NULL,"Initialization Failed. ","ERROR",MB_OK|MB_ICONEXCLAMATION);
        return FALSE; // Return
FALSE
    }

    return TRUE; // Success
}

LRESULT CALLBACK WndProc(   HWND   hWnd,
                           UINT   uMsg, // Handle For This Window // Message
                           WPARAM  wParam, //
                           LPARAM lParam) //
{
    switch (uMsg) // Check
For Windows Messages
    {
        case WM_ACTIVATE: // Watch For
Window Activate Message
        {
            if (!HIWORD(wParam)) // Check Minimization
State
            {
                active=TRUE; // Program
Is Active
            }
            else
            {
                active=FALSE; // Program
Is No Longer Active
            }
        }

        return 0; // Return To
The Message Loop
    }

    case WM_SYSCOMMAND: // Intercept
System Commands
    {
        switch (wParam) // Check
System Calls
        {

```

```

Screensaver Trying To Start?           case SC_SCREENSAVE:           //
Trying To Enter Powersave?           case SC_MONITORPOWER:       // Monitor
From Happening                         return 0;                     // Prevent
Exit                                   }                               //
                                     break;
Receive A Close Message?             case WM_CLOSE:               // Did We
Quit Message                           {                             // Send A
Back                                   PostQuitMessage(0);         // Jump
                                     return 0;
Being Held Down?                     case WM_KEYDOWN:            // Is A Key
TRUE                                   {                             // If So, Mark It As
Back                                   keys[wParam] = TRUE;
                                     return 0;                     // Jump
Key Been Released?                   case WM_KEYUP:              // Has A
FALSE                                  {                             // If So, Mark It As
Back                                   keys[wParam] = FALSE;
                                     return 0;                     // Jump
The OpenGL Window                     case WM_SIZE:               // Resize
Back                                   {                             // Jump
                                     ReSizeGLScene(LOWORD(lParam),HIWORD(lParam)); // LoWord=Width, HiWord=Height
                                     return 0;
}
// Pass All Unhandled Messages To DefWindowProc
return DefWindowProc(hWnd,uMsg,wParam,lParam);
}

int WINAPI WinMain(                   HINSTANCE   hInstance,           // Instance
Line Parameters                       HINSTANCE   hPrevInstance,       // Previous Instance
                                     LPSTR       lpCmdLine,           // Command
Window Show State                     int          nCmdShow)           //
{
    MSG      msg;
Windows Message Structure              //
    BOOL     done=FALSE;             // Bool
Variable To Exit Loop

    // Ask The User Which Screen Mode They Prefer
    if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?", "Start
Fullscreen?",MB_YESNO|MB_ICONQUESTION)==IDNO)
    {
        fullscreen=FALSE;           // Windowed Mode
    }

    // Create Our OpenGL Window
    if (!CreateGLWindow("NeHe's Solid Object Tutorial",640,480,16,fullscreen))

```

```

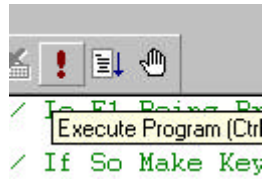
    {
        return 0; // Quit If
Window Was Not Created
    }

    while(!done) // Loop That
Runs While done=FALSE
    {
        if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Is There A Message Waiting?
        {
            if (msg.message==WM_QUIT) // Have We Received
A Quit Message?
            {
                done=TRUE; //
If So done=TRUE
            }
            else //
If Not, Deal With Window Messages
            {
                TranslateMessage(&msg); // Translate The
                DispatchMessage(&msg); // Dispatch The
                Message
                Message
            }
        }
        else //
If There Are No Messages
        {
            // Draw The Scene. Watch For ESC Key And Quit Messages From DrawGLScene()
            if ((active && !DrawGLScene()) || keys[VK_ESCAPE]) // Active? Was There A Quit
Received?
            {
                done=TRUE; //
ESC or DrawGLScene Signalled A Quit
            }
            else //
Not Time To Quit, Update Screen
            {
                SwapBuffers(hdc); // Swap
                Buffers (Double Buffering)
            }
            if (keys[VK_F1]) // Is F1 Being
Pressed?
            {
                keys[VK_F1]=FALSE; // If So
                KillIGLWindow(); // Kill Our
                Current Window
                fullscreen=!fullscreen; // Toggle Fullscreen /
                Windowed Mode
                // Recreate Our OpenGL Window
                if (!CreateGLWindow("NeHe's Solid Object Tutorial",640,480,16,fullscreen))
                {
                    return 0; // Quit If
                }
Window Was Not Created
            }
        }
    }

    // Shutdown
    KillIGLWindow(); // Kill The
Window
    return (msg.wParam); // Exit The Program
}

```

Y pulsar el icono de la exclamación para compilar y ejecutar.



Y ESTO ES TODO... SI QUEREIS MÁS INFORMACIÓN PODEIS ENTRAR EN CONTACTO CONMIGO EN ifgstorm@chips.uvigo.es O iagofg@hotmail.com