Índice

1. Introducción 3

Parte I. ESCENARIO

2. Inicialización Básica de DirectX 6

3. Algoritmo de "División Binaria del Espacio" (BSP) 16

4. Texturas 25

5. Iluminación 28

6. Mejoras en Luces y Texturas 32

7. Compilador de Árboles BSP 39

8. Mapa Selector de Texturas 49

9. Rechazo por Cono de Visión 50

10. Algoritmo de "Conjuntos Posiblemente Visibles" (PVS) 54

11. Técnica de Mapeado MIP (mipmapping) ..... 75{ XE "Usuario" **}**

12. Archivo de registro 77

Parte II. OBJETOS

13. Modelos de Humanoides 81

14. Desplazamiento de los Modelos 87

15. Detección de Colisiones Mediante Esferas 94

16. Inclusión de Múltiples Objetos ..... 98{ XE "Usuario" }

17. Funcionalidad de Disparo 101

Parte III. PERFECCIONAMIENTO

18. Pantalla Completa 105

19. Mejoras en el Rendimiento y Realismo 106

20. Sobreimpresión de Texto en Pantalla 112

21. Sonidos 116

22. Niebla Volumétrica 118

23. Mejora de Sonidos e Inclusión de Música 121

24. Nuevos Movimientos del Protagonista 124

25. Mira Telescópica 125

Parte IV. APÉNDICES

23. Requisitos de ejecución 128

24. Bibliografía 129

Introducción

Introducción al proyecto

El desarrollo de juegos implica la resolución de problemas de carácter diverso puesto que no sólo incluye la utilización de librerías gráficas sino problemas de más alto nivel relativos a la creación de espacios geométricos o controladores de dispositivos de juego, y todo ello en el contexto de una historia con un guión y un estilo que deben ser propios. Por ello este campo nos ha resultado especialmente atractivo para nuestro proyecto.

Descripción del proyecto

Consistirá en el diseño de un juego centrándonos en el aspecto gráfico. El tipo de juego elegido es shoot'em up subjetivo, como por ejemplo "quake" y "doom".

En un primer momento nos centraremos en la creación de un motor gráfico que se adapte a nuestras necesidades, para posteriormente ocuparnos del diseño de personajes, escenarios y guión del juego.

Por último desearíamos implementar sonido y mejoras gráficas como nieblas volumétricas, zooms y mayor nivel de detalle.

Fases

1. Documentación y selección de alternativas: En esta etapa obtendremos documentación sobre las librerías gráficas disponibles (OpenGL y directx), compilador y lenguaje más adecuado (Microsoft, Borland, ...) y herramientas auxiliares de diseño de objetos gráficos. Con esta documentación elegiremos los más convenientes.

2. Toma de contacto con el software elegido para el desarrollo: Esta segunda fase consistirá en desarrollar un prototipo simple de nuestro motor gráfico, para así habituarnos al lenguaje elegido. Este primer prototipo será la base del futuro desarrollo del proyecto.

3. División de tareas: Una vez familiarizados con el entorno procederemos a asignar las diferentes labores que cada uno de los miembros realizará a lo largo del proyecto. Obtendremos un motor gráfico para los personajes y el escenario.

4. Integración de los distintos componentes: Puesta en común de los resultados anteriores juntándolos en lo que sería una primera versión.

5. Diseño de una beta-jugable: Elaborar uno o dos niveles de juego para poder apreciar el conjunto de nuestro trabajo, y poder sobre esta base implementar mejoras adicionales.

6. Mejoras suplementarias: En esta fase incluiríamos detalles de menor importancia que dotarían al juego de un aspecto más elaborado tales como nieblas volumétricas, zooms, tratamiento de luces e incluso efectos sonoros.

PARTE I

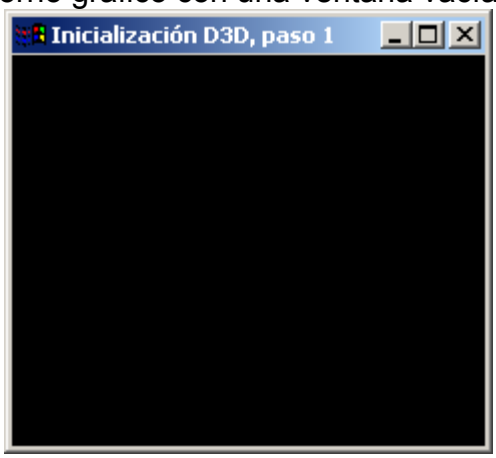
ESCENARIO

Inicialización Básica de DirectX

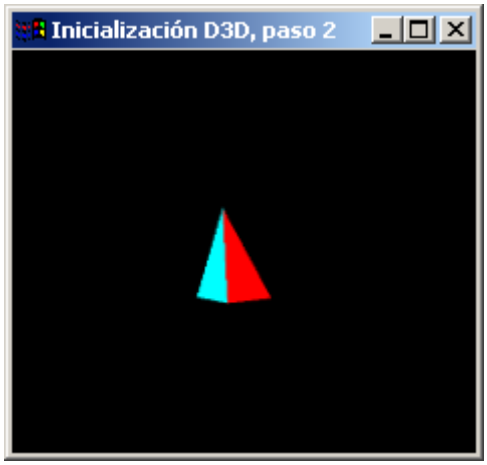
Esta primera versión del proyecto no es más que una toma de contacto con las herramientas de trabajo que vamos a usar, es decir, Visual C++ y DirectX 8.0.

Hemos obtenido 4 ejecutables en los que conseguimos sucesivamente:

1. Inicializar el entorno gráfico con una ventana vacía

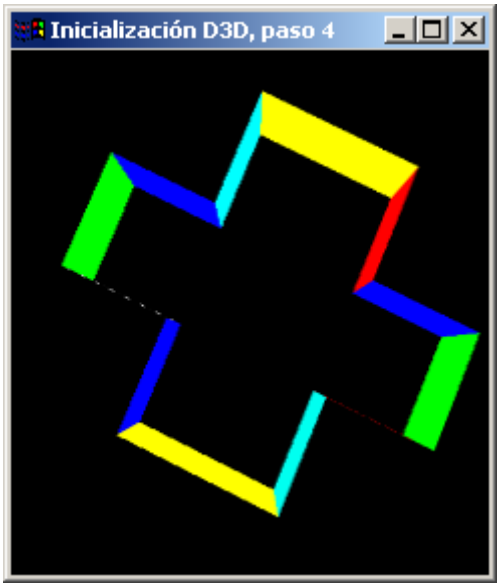


2. Dibujo en 3D de un tetraedro sobre la ventana anterior



3. Lectura de las teclas de cursor aplicando matrices de transformación que permiten el movimiento de la cámara en el espacio.

4. Creación de un mundo (habitación de 12 paredes) en la que se aplica todo lo aprendido en las fases anteriores.



Inicialización de D3D

Para la inicialización de Direct Graphics vamos a necesitar varios pasos:

- Crear un objeto Direct 3D que contenga los parámetros necesarios para su correcta visualización.
- El paso siguiente será coger el modo de pantalla actual para que el objeto D3D pueda visualizarse correctamente.
- Luego podemos crear una variable de parámetros presentes para determinar el comportamiento de nuestra aplicación.
- Finalmente ya podemos crear el device de Direct3D.

Lo primero es crear el objeto de D3D, para ello se hace esto:

```
if( NULL == ( g_pD3D = Direct3DCreate8( D3D_SDK_VERSION ) ) )
    return E_FAIL;
```

Luego se coge el modo actual de visualización del escritorio para pasarlo al objeto D3D

```
D3DDISPLAYMODE d3ddm;
if( FAILED( g_pD3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT,
                    &d3ddm ) ) )
    return E_FAIL;
```

Luego configuramos algunos parámetros que van a determinar el comportamiento posterior y para luego crear el D3Ddevice. De este modo primero limpiamos la memoria, luego lo ponemos en modo ventana que es el que hemos usado, también anulamos la posibilidad de poder cambiar la ventana por motivos de eficiencia.

```
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
```

Finalmente ya podemos crear el D3Device, para ello intentamos usar HAL[1] ya que acelera en algunas tarjetas y usamos MIXED en el proceso de vértices por ser más eficiente, aunque con algunos equipos falla y para ellos usamos SOFTWARE.

```
if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT,
                    D3DDEVTYPE_HAL, hWnd,
                    D3DCREATE_MIXED_VERTEXPROCESSING,
                    &d3dpp, &g_pd3dDevice ) ) )
```

Creación y llenado de un buffer de vértices

Primero hay que darle los valores de los vértices que queremos pintar, en este caso no son más que tres vértices. La estructura de un vértice es {Coordenada X, Coordenada Y, Coordenada Z, Color del vértice}.

```
CUSTOMVERTEX g_Vertices[] =
{
                    { 100.0f,   0.0f,-150.f, 0xff00fff0},
                    { 100.0f, 100.0f,-150.f, 0xff00fff0},
                    { 100.0f, 100.0f,- 50.f, 0xff00fff0},
}
```

Una vez relleno un array con los vértices que queremos dibujar creamos el buffer de vértices reservando suficiente memoria

[1] Los distintos parámetros que podemos usar en esta parte son explicados con más detenimiento posteriormente.

```
if( FAILED( g_pd3dDevice->CreateVertexBuffer( sizeof(g_Vertices) *
                        sizeof(CUSTOMVERTEX), 0,  D3DFVF_CUSTOMVERTEX,
                        D3DPOOL_DEFAULT, &g_pVB ) ) )
```

Ahora hay que llenar el buffer de vértices, para ello hay que "cerrarlo" y una vez metidos ya podemos volverlo a abrir:

```
VOID* pVertices;
   if( FAILED( g_pVB->Lock( 0, sizeof(g_Vertices), (BYTE**)&pVertices,
                    0 ) ) )
        return E_FAIL;
   memcpy( pVertices, g_Vertices, sizeof(g_Vertices) );
   g_pVB->Unlock();
```

Ahora ya tenemos creado y lleno el buffer de vértices. Si ahora queremos pintarlo hay que hacer estos pasos en la función que va a renderizar la escena:

- Fijar la fuente del flujo de vértices.
- Luego indicar qué tipo de vértices nos vamos a encontrar,
- Finalmente renderizar los vértices según nuestro propósito.

Por ejemplo, si queremos pintar un triángulo con los 3 vértices sería así:

```
    g_pd3dDevice->SetStreamSource( 0, g_pVB, sizeof(CUSTOMVERTEX) );
    g_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
    g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLE, 0, 3);
```

Inicialización del entorno y el mundo 3D

Primero hay que definir el mundo, en este caso lo definimos como la identidad, aunque podemos rotarlo:

```
D3DXMATRIX matMundo;
D3DXMatrixIdentity (&matMundo);
g_pd3dDevice->SetTransform(D3DTS_WORLD,&matMundo);
```

Luego definimos el punto de vista, para ello necesitamos un punto de partida, una dirección a la que mirará y otra coordenada que indica algo obvio para nosotros: dónde se encuentra la parte de arriba de nuestro mundo, es decir un vector que aclare lo que es arriba:

```
D3DXMATRIX matVista;
D3DXVECTOR3 puntoDeVista;
D3DXVec3Add(&puntoDeVista, &posicionDeLaCamara, &direccionVista);
D3DXMatrixLookAtLH (&matVista, &posicionDeLaCamara, &puntoDeVista,
                        &D3DXVECTOR3(0.0f,1.0f,0.0f));
```

Ahora necesitamos crear la perspectiva, así como la distancia a la que los objetos se empiezan a renderizar y a la distancia que se dejan de renderizar, así tendremos un intervalo intermedio en el cual podremos visualizar los objetos (en esta caso entre 0 y 300):

```
g_pd3dDevice->SetTransform(D3DTS_VIEW,&matVista);
D3DXMATRIX matProj;
D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 0.0f, 300.0f );
g_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );
```

Inicialización de DirectInput

Primero creamos el objeto DirectInput.

```
if (FAILED (DirectInput8Create (hInst, DIRECTINPUT_VERSION,
                        IID_IDirectInput8, (void**)& lp_DI8,NULL)))
   return (E_FAIL);
```

Luego creamos el "Device" que interactúa con el hardware

```
if (FAILED(lp_DI8->CreateDevice (GUID_SysKeyboard,
                        &lp_DI8Device,NULL)))
   return (E_FAIL);
```

Ahora fijamos el tipo de datos que intercambiamos

```
if (FAILED(lp_DI8Device->SetDataFormat(&c_dfDIKeyboard)))
   return (E_FAIL);
```

Fijamos el modo de interacción, en este caso no exclusivo de la aplicación pero si que va a funcionar cuando no esté activa.

```
if (FAILED(lp_DI8Device->SetCooperativeLevel (hWnd,DISCL_BACKGROUND  |
                        DISCL_NONEXCLUSIVE)))
return (E_FAIL);
```

Finalmente tomamos control sobre el teclado. Como puede fallar, lo intentamos hasta que se nos dé el control.

```
while (FAILED(lp_DI8Device->Acquire())) {}
```

Lectura del teclado

Para leer el teclado usamos el siguiente macro:

```
#define TECLAPULSADA(lista, tecla) (lista[tecla] & 0x80)
```

Una vez definido esto se hace muy fácil la lectura del teclado, ya que hay unas constantes definidas por DirectInput que representan las teclas. Así por ejemplo para la tecla del cursor hacia arriba nuestro código es así:

```
if (TECLAPULSADA(buffer,DIK_UP)) estadoTeclado[0]=1;
```

Problemas solventados en la inicialización básica

La inicialización propuesta en la documentación C++ del SDK, (inicialización implementada en el proyecto del año anterior), genera un mensaje de error

D3DERR_INVALIDCALL en el más antiguo de los equipos utilizados (recordemos que utiliza un chip 3D Rage II+ con limitadas prestaciones gráficas en 3D). Realizando una depuración paso a paso se observa que el error se produce en la instrucción de creación del dispositivo:

```
error = lpd3d8->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hwnd,
                        D3DCREATE_HARDWARE_VERTEXPROCESSING, &pParams,
                        &lpd3ddevice8);

Direct3D8: (INFO) :======================== Hal SWVP device selected
Direct3D8: (ERROR) :This back buffer format is not supported for a
                        windowed device. CreateDevice/Reset Fails
Direct3D8: (ERROR) : Use CheckDeviceType(Adapter, DeviceType, <Current
                        Display Format>, <Desired BackBufferFormat>,
                        TRUE /* Windowed */)
Direct3D8: (ERROR) :Failed to initialize Framework Device.
                        CreateDevice Failed.
```

Probando a cambiar de D3DCREATE_HARDWARE_VERTEXPROCESSING a D3DCREATE_SOFTWARE_VERTEXPROCESSING para que el proceso de vértices se haga por software, a ventana output del depurador muestra:

```
Direct3D8: (ERROR) :Unsupported back buffer format specified.
Direct3D8: (ERROR) :Failed to initialize Framework Device.
                        CreateDevice Failed.
```

Realizando diversas ejecuciones en las que se ha cambiado la profundidad de color de la pantalla (a 256 colores, 16 bits, 24 bits) se obtienen otros errores que siguen impidiendo la ejecución:

```
Direct3D8: (ERROR) :AutoDepthStencilFormat does not match
                        BackBufferFormat because the current Device
                        requires the bitdepth of the zbuffer to match
                        the render-target. See CheckDepthStencilMatch
                        documentation. CreateDevice fails.
Direct3D8: (ERROR) :Failed to initialize Framework Device.
                        CreateDevice Failed.
```

El primero de los errores recomienda el uso de **CheckDeviceType** para comprobar los modos de inicialización compatibles con la tarjeta gráfica. Utilizando esta llamada obtenemos el modo correcto de inicialización, utilizando el parámetro **D3DDEVTYPE_REF** para que utilice el dispositivo de referencia (un dispositivo virtual ofrecido por DirectX que realiza emulación por software de las capacidades hardware que no existan), así como el proceso de los vértices por Software. Obviamente, estos dos parámetros ralentizan la ejecución notablemente en máquinas de mayor potencia, por lo que parece aconsejable realizar como tarea aparte un procedimiento de inicialización robusto que empiece probando una por una todas las capacidades, y asigne los parámetros adecuados para utilizar la aceleración por hardware siempre que sea posible, pero a su vez permita la ejecución mediante emulación

software cuando nos encontremos tarjetas que no dispongan de esa capacidad.

Posteriormente hemos aprendido que el propio SDK suministra una utilidad que informa sobre las diferentes capacidades de proceso hardware del dispositivo gráfico con todo detalle: el Directx Caps Viewer.

Inicialización de diversos modelos y configuraciones de tarjetas

La función de interés en esta versión del proyecto es InitD3D. Esta rutina va probando en cascada diversos modos de inicialización que deberán funcionar en todas las tarjetas objeto de esta prueba. El resto del proyecto solamente tiene que mostrar dos cuadrados paralelos rotando con los que poder comprobar que DirectX funciona correctamente, por lo que puede ser desechado.

El ejecutable obtenido debe obtener la ventana que muestra el objeto rotando, y por tanto, demuestra que la inicialización ha sido correcta, o bien el mensaje de error generado (figura 1).

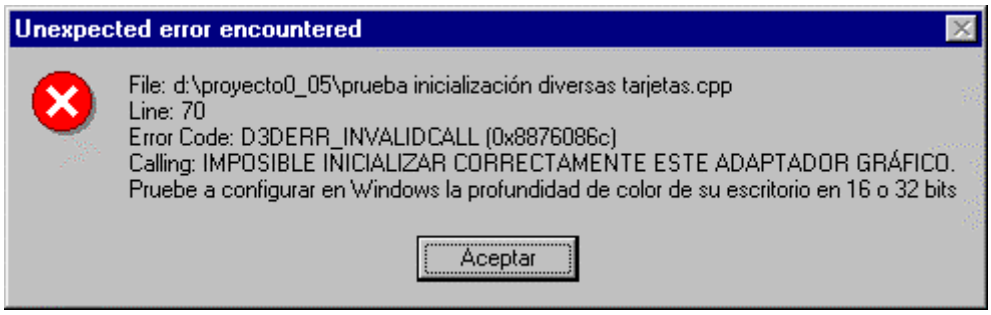


Figura 1.

La función de inicialización debería asignar los parámetros adecuados para conseguir sacar el máximo rendimiento de la tarjeta gráfica. Por ello, se realizan 4 llamadas a CreateDevice, la primera de ellas trata de inicializar el adaptador asignando la mayor parte de las funciones al procesamiento hardware embebido en el chip acelerador de la propia tarjeta. Confiando en la capacidad de proceso del chip gráfico permitiremos la ejecución concurrente de las tareas de dibujo con las encomendadas al procesador Pentium, elevando el rendimiento general del juego. Si la primera llamada resulta imposible de satisfacer por la tarjeta gráfica se realizan sucesivas llamadas en las que las funciones requeridas a la tarjeta son cada vez menores.

El parámetro utilizado para activar el procesamiento hardware en el adaptador por defecto D3DADAPTER_DEFAULT del sistema es D3DDEVTYPE_HAL.

Por otra parte, en la sección dedicada a "Processing Vertex Data" de la documentación C++ del SDK de DirectX 8 se recomienda utilizar el parámetro D3DCREATE_SOFTWARE_VERTEXPROCESSING en la inicialización puesto que así no se limita el número disponible de luces y sin embargo en el caso concreto del procesamiento de vértices los rendimientos hardware y software son similares (al contrario que al dibujar, en el que el rendimiento hardware es muy superior al software).

Asimismo recomienda la posibilidad de inicializar con D3DCREATE_MIXED_VERTEXPROCESSING, para disponer de la opción de procesamiento hardware y software a la vez, pero la ATI Fury no admite la mezcla de procesamiento de vértices, por lo que no se ha considerado ese parámetro en la inicialización.

La primera llamada tiene la forma:

```
if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                                  D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                  &d3dpp, &g_pd3dDevice ) ) )
```

En el caso de que la llamada resulte fallida se intenta una nueva llamada restando capacidad de procesamiento al hardware gráfico, y trasladando la responsabilidad de la ejecución al software de emulación de referencia provisto por DirectX, esto permitirá la ejecución en un amplísimo rango de tarjetas:

```
if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
                                  D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                  &d3dpp, &g_pd3dDevice ) ) )
```

Si el fallo de inicialización persiste, el problema probablemente residirá en que el modo gráfico autodetectado no es correcto, por lo que procedemos a asignar uno de los dos valores de inicialización más comúnmente aceptados (son los que presentan de modo nativo las ATI Fury o Rage, por ejemplo). Empezaremos asignando al Back Buffer[2] el modo de 32 bits (8 bits para cada color, incluyendo el negro) :

```
d3dpp.BackBufferFormat = D3DFMT_X8R8G8B8;
if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
                                   D3DCREATE_SOFTWARE_VERTEXPROCESSING,


&d3dpp, &g_pd3dDevice ) ) )
```

Por último, reducimos al mínimo los requerimientos solicitados, intentando inicializar el Back Buffer a tan sólo 16 bits (5 bits para cada color):

[2] DirectX dispone de uno o varios "Back Buffers" y un "Front Buffer" que se corresponde con la ventana que vemos. No es posible dibujar directamente en el Front, en realidad se dibuja en uno de los Back Buffers y el contenido se vuelca al Front al realizar la invocación del método Present.

```
d3dpp.BackBufferFormat = D3DFMT_X1R5G5B5;
resul_err = g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_REF, hWnd,
                                  D3DCREATE_SOFTWARE_VERTEXPROCESSING,




&d3dpp, &g_pd3dDevice );
```

Finalmente si ha resultado imposible la inicialización del adaptador se genera un cuadro de diálogo con el mensaje de error en el que se recomienda cambiar la profundidad de color de Windows antes de intentarlo de nuevo.

```
if (FAILED(resul_err))
{
      DXTRACE_ERR ("IMPOSIBLE INICIALIZAR CORRECTAMENTE ESTE ADAPTADOR
      GRÁFICO. Pruebe a configurar en Windows la profundidad de color de su
      escritorio en 16 o 32 bits",resul_err);
      return E_FAIL;
}
```

Inicialización del ratón

Se ha introducido una nueva clase llamada ClaseInterfaz que gestiona todos los dispositivos de entrada usados en el proyecto. Ya estaba implementado el uso del teclado, y además hora hemos incorporado el ratón y hemos intentado implementar la entrada por joystick, pero unos problemas imprevistos en la inicialización han retrasado el desarrollo del uso de este dispositivo.

Para el ratón no ha sido muy difícil ya que la inicialización era muy parecida a la del teclado. Este es el código usado para inicializar el ratón:

```
//Creamos el Device que interactua con el hardware
if (FAILED(lp_DI8->CreateDevice(GUID_SysMouse, &g_pMouse, NULL)))
{
                                                                    return
(E_FAIL);
}
else
                                                                    //Fijam
os el tipo de datos que intercambiamos
                                                                    if
(FAILED(g_pMouse->SetDataFormat(&c_dfDIMouse2)))
                                                                    {

                                                                    return
(E_FAIL);
                                                                    }
                                                                    else

                                                                    //Fijam
os el modo de interacción
            if (FAILED(g_pMouse->SetCooperativeLevel(hWnd, DISCL_EXCLUSIVE |
            DISCL_FOREGROUND)))
```

```
                                                                    {

                                                                    return
(E_FAIL);
                                                                    }
                                                                    else

                                                                    return
(S_OK);
}
```

El texto ya viene con sus comentarios y no considero necesario comentarlo por lo dicho antes. Simplemente decir que lp_DI8 es un objeto de Direct Input debidamente inicializado (inicializado antes para en teclado), y que g_pMouse es el dispositivo que interactúa con el ratón.

Luego tenemos que adquirir el control del ratón, y para ello se usa lo mismo que se usó antes para el teclado:

```
g_pMouse->Acquire();
```

Para leer el estado del ratón y poder pasarlo al programa para que la cámara se pueda mover adecuadamente hay que coger este estado del ratón en una variable y luego coger los datos que nos interesen de esta variable:

```
//Cogemos el estado actual del ratón
g_pMouse->GetDeviceState( sizeof(DIMOUSESTATE2), &dims2 );

//Vamos a coger la posición X, Y y Z del ratón
estadoRaton[0]=dims2.lX;
                                                                    estadoR
aton[1]=dims2.lY;
                                                                    estadoR
aton[2]=dims2.lZ;
                                                                    //Vamos
a coger el estado de 11 posibles botones.
                                                                    for(i=3
;i<11;i++) estadoRaton[i]=dims2.rgbButtons[i];
```

Así la variable estadoRaton[] podemos pasarla al programa.

Algoritmo "División Binaria del Espacio" (BSP)

Aunque la anterior versión solamente presentaba un pequeño mundo de 9 polígonos, parece obvio que pronto en el desarrollo de cualquier juego, el número de polígonos necesarios (o el de vértices que los delimitan) se multiplica exponencialmente. Los árboles BSP han llamado nuestra atención debido a que John Carmack los utilizó en Doom y Quake. Para introducirnos en su uso hemos recurrido a la documentación realizada por Gary Simmons.

Tendremos que enfrentarnos a los siguientes problemas:

1. Decidir qué polígonos se van a dibujar (y en qué orden) teniendo en cuenta la actual posición de la cámara. La correcta ordenación de los polígonos en cada momento puede llevar a una fuerte degradación del rendimiento del juego.
2. En el futuro podemos querer incluir el problema de tratar objetos traslúcidos.
3. Detección de colisiones. Al igual que el punto 1, constituye un grave problema de rendimiento realizar un test de colisión respecto de cada polígono por cada movimiento.

BSP precalcula el orden de los polígonos en tiempo de desarrollo, no en tiempo de ejecución, de forma que toda la ordenación de los polígonos está precalculada. Esta preordenación es realizada por un compilador BSP que toma como entrada una lista de polígonos para realizar la ordenación y como salida obtiene un fichero salvado a disco que contiene los polígonos dispuestos en un árbol BSP. Este árbol puede ser cargado en tiempo de ejecución y rápidamente recorrido.

El tener los polígonos preordenados permite que en cada 'frame' se puedan desechar centenares de polígonos que están más allá del límite de visión o están detrás de otros, ahorrando todo el tiempo de proceso asociado a transformar e iluminar todos esos polígonos que jamás se verán. Un test (de colisión o si está tapado por otro polígono o si está más allá del límite de visión) realizado a cada polígono siguiendo el orden determinado por el árbol BSP determinará cuándo podemos parar y desechar el resto de polígonos que queden por procesar.

La justificación del interés por BSP, así como sus principales usos ya han quedado descritos. Ahora, pasaremos a mostrar la implementación y el funcionamiento interno de este algoritmo, así como de algunas mejoras introducidas sobre el algoritmo original.

Un árbol BSP es una subdivisión jerárquica de un espacio de 3 dimensiones en subespacios. Cada nodo se subdivide en nodos que están delante y nodos que están detrás de él. Empezando por el nodo raíz, todas las siguientes inserciones son divididas por el plano del nodo actual, en la parte delantera y trasera.

El objetivo de un árbol BSP es conocer si sus nodos hojas están delante o detrás del nodo padre. Los árboles BSP son muy usados para la interacción en tiempo real con imágenes estáticas, antes de que la escena sea dibujada, se calcula el árbol BSP. Los árboles BSP pueden calcular rápidamente (en tiempo lineal) qué superficie hay que ocultar.

El aspecto de un nodo de árbol BSP sería:

```
struct TNodo
{
plano divisor;
TLista poligono;
TNodo* enfrente;
TNodo* detras;
}
```

Y la creación del árbol a partir de una lista de polígonos:

```
Crea_arbol(TNodo nodo_actual, TLista lista)
{
Poligono raiz=lista.elemento();
nodo_actual->divisor=raiz.plano();
nodo_actual->poligono.añade(raiz);
while((poli=lista.elemento())!=0)
{
resultado=clasificaPoligono(nodo_actual,poli);
switch(resultado){
case ENFRENTE: lista_enfrente.añade(poli);
case DETRAS: lista_detras.añade(poli);}
}
if(!lista_enfrente.vacia()) Crea_arbol(nodo_actual->enfrente,
lista_enfrente);
if(!lista_detras.vacia()) Crea_arbol(nodo_actual->detras,
lista_detras);
}
```

El funcionamiento del algoritmo es el siguiente: Llamamos a la función con dos parámetros, el primero(`TNodo nodo_actual`) es la raíz del subárbol y el nodo a partir del cual vamos a generar las divisiones, el segundo parámetro(`TLista lista`) es una lista de todos los elementos que nos falta por visitar, que parten desde el nodo_actual.

Lo primero que hacemos es asignar a nodo_actual el plano de corte y luego recorremos la lista de los nodos para clasificarlos en dos nuevas listas, los elementos que están delante y los elementos que están detrás del nodo_actual. Después llamamos recursivamente a la función con las listas de los elementos de enfrente y de detrás.

Hay que tener en cuenta dos casos especiales:

- Los planos de las caras de los polígonos coinciden (formando una línea).
- Los planos de las caras de los polígonos intersectan (uno corta al otro).

En el primer caso, si las caras de los polígonos coinciden hay que tenerlo en cuenta en la construcción del árbol, en cuyo caso se añadiría a la lista de los polígonos del nodo actual. En el switch del código habría que añadir este caso:

```
case COINCIDEN: nodo_actual->poligono.añade(poli);
```

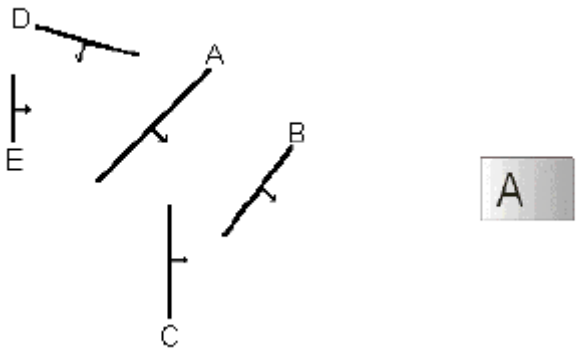
En el segundo caso deberemos partir el polígono en dos, de forma que siempre un polígono (una porción del original) quede delante y el resto detrás. Si no hiciéramos esto, un polígono se encontraría a la vez detrás y delante del otro plano. Para controlarlo habría que añadir al switch:

```
case INTERSECTAN:
poligono p_detras,p_enfrente;
dividir_Poligono(poli,nodo_actual,p_detras,p_enfrente);
lista_detras.añade(p_detras);
lista_enfrente.añade(p_enfrente);
```

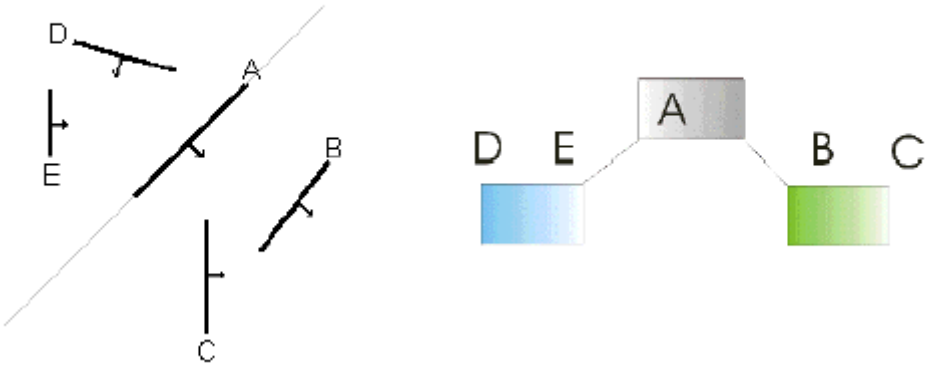
De esta forma se crean dos nuevos polígonos, que son la parte del polígono que se encuentra delante y detrás. Cada uno de ellos se añade a la lista correspondiente.

Podemos ver gráficamente como funciona BSP, en el ejemplo tenemos 5 muros dispuestos en el espacio y etiquetados como A a D, cada uno de ellos tiene una orientación indicada por la flecha asociada:

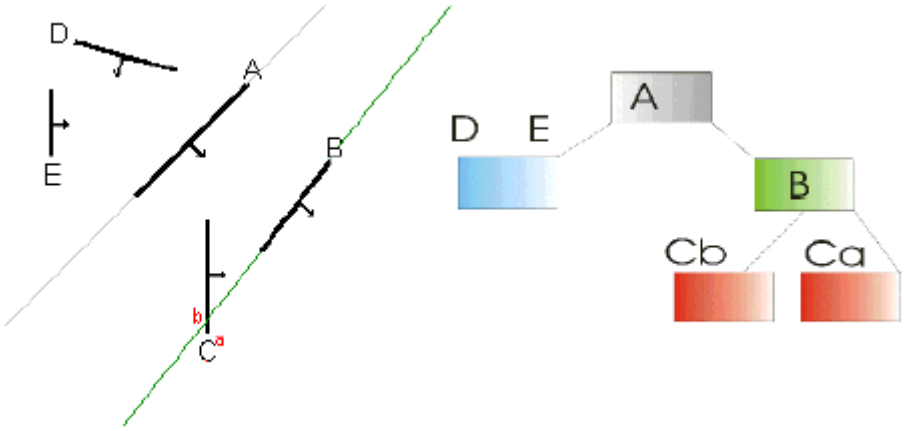
1.- Se selecciona el segmento A como separador del espacio.



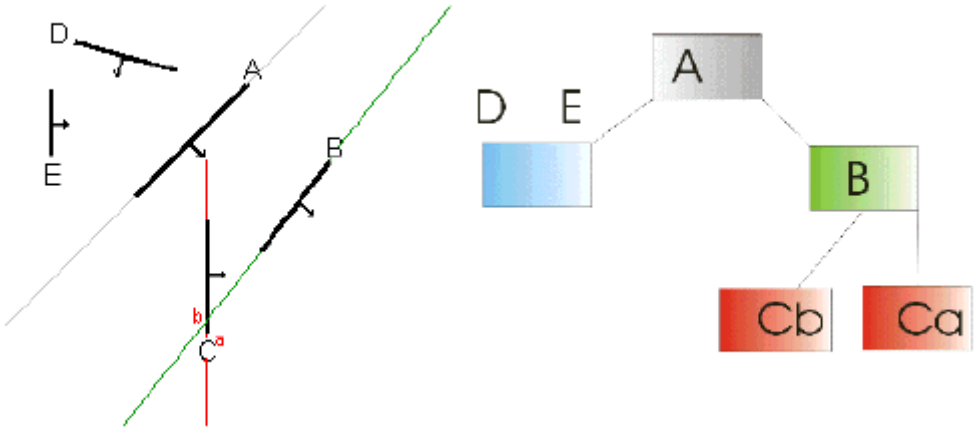
2.- Se expande el plano que pasa por A y divide el espacio entre los que están enfrente de A (B y C) y los que están detrás de A (D y E).



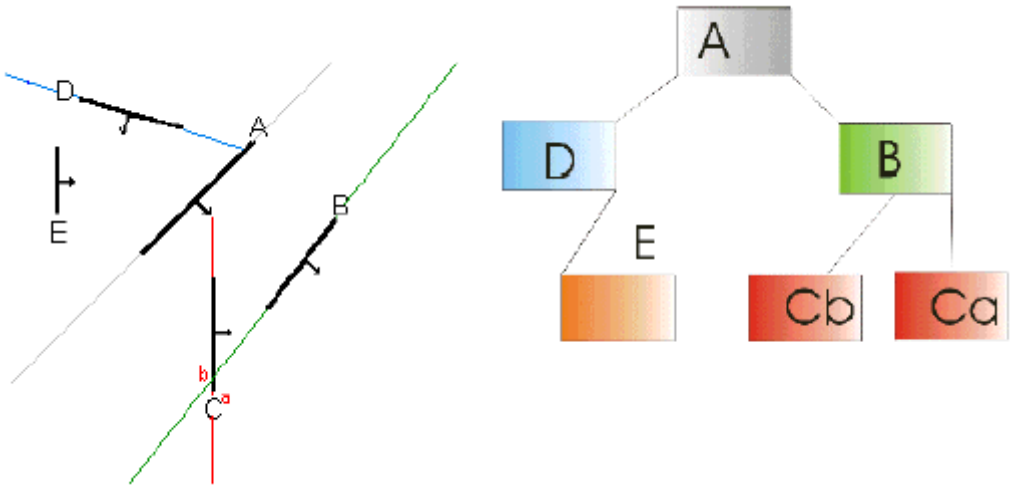
3.- Se introduce el nodo B que traza el plano y divide el segmento C en Ca y Cb, ya que intersecta con él. El Cb pertenecerá a la lista de atrás y el Ca a la lista de delante.



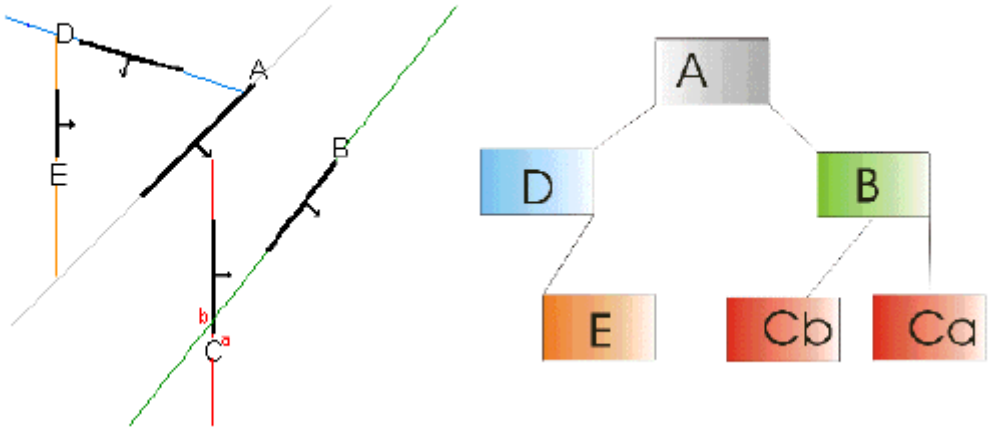
4.- Se crea el plano de Cb y Ca y se introducen en sus respectivos nodos tanto Cb como Ca.



5.- Se crea el plano de D, que deja al segmento E enfrente de él y por lo tanto en esa lista.



6.- Se crea el plano de E y se introduce en su respectivo nodo.



Al algoritmo clásico BSP, se le han introducido algunas mejoras para tratar de forma más eficaz las estructuras sólidas (como las delimitadas por 4 paredes y por tanto no accesibles desde las zonas de paso del juego).

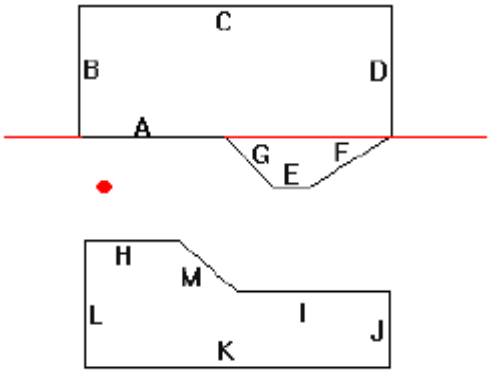
Los planos del ejemplo anterior tenían el problema de ser poco creíbles dentro de un nivel de juego. Lo habitual es encontrarse varios de estos planos juntos formando paredes, columnas, etc. La estructura que formen estos planos será sólida: necesitamos representar sólidos.

Son necesarias unas modificaciones en el tipo de nodo y en el funcionamiento del algoritmo:

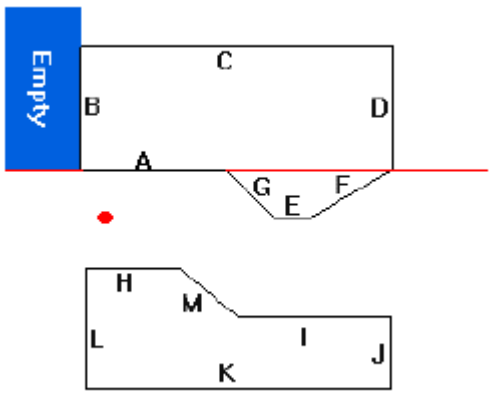
```
TNodoB
{
plano divisor;
TLista poligono;
TNodo* enfrente;
TNodo* detras;
BOOL esHoja;
BOOL esSolido;
}
```

Ahora el funcionamiento de BSP será el siguiente:

Seleccionamos el plano de A para dividir el espacio, poniendo el resto de los polígonos de la B-M en sus respectivas listas. Ahora el algoritmo selecciona el muro B para realizar la siguiente división.

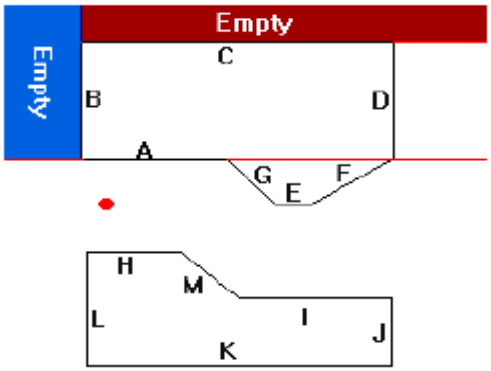


Ponemos C y D en su lista y a continuación, en vez de poner la lista del frente a NULL creamos un nuevo nodo y lo ponemos en la lista de delante de B, este nodo no tiene polígono porque es una hoja. Se actualizan las nuevas

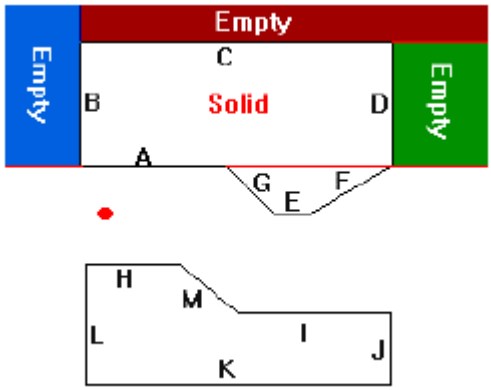


variables esHoja a true y esSolido a false.

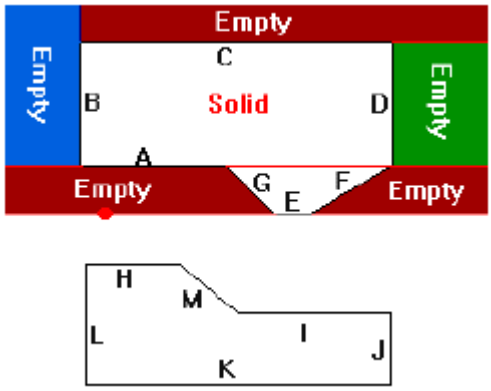
Se selecciona el plano de C para dividir el espacio, como enfrente del plano no hay nada se crea un nodo nuevo y se pone su variable esHoja a true y como está en la lista de detrás de su padre la variable esSolido se pone a false. En la lista de los que están detrás de C se pone D.



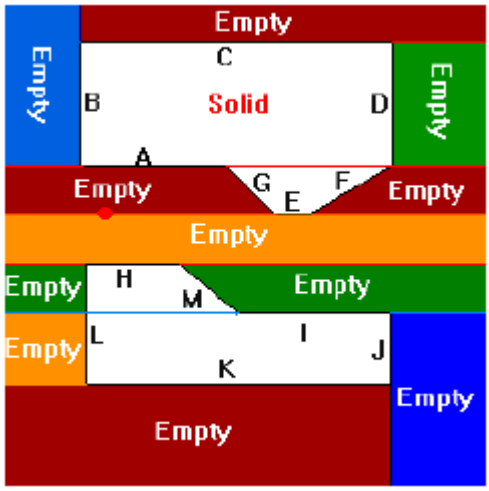
Se selecciona el plano D para realizar la división, como no hay nada en la lista de enfrente se crea un nuevo nodo y se pone su variable esHoja a true y esSolido a false. Como la lista de los elementos que están detrás está vacia se vuelve a crear un nuevo nodo, con la variable esHoja a true y la variable esSolido también a true porque pertenece a la lista de detrás.



La función continúa recursivamente, utilizando ahora el plano de E. Separando los que están detrás (G y F) y los que están delante (H-M). Selecciona a continuación G y después F y procede como en los casos anteriores.



Las mismas reglas se aplicarían para el cálculo del árbol BSP asociado al segundo bloque sólido del ejemplo hasta obtener:



Texturas

El objetivo inicial de esta nueva entrega es familiarizarnos con el uso de texturas. Las texturas pueden ser consideradas como los dibujos que adornan paredes, techos y suelos, y dotan a estas superficies de mayor realismo y credibilidad que los colores de la anterior entrega del proyecto. En esta figura vemos la textura utilizada en esta nueva versión del trabajo:

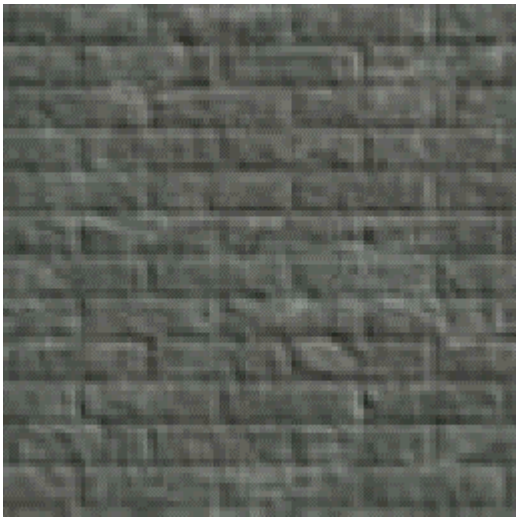


Figura 1.

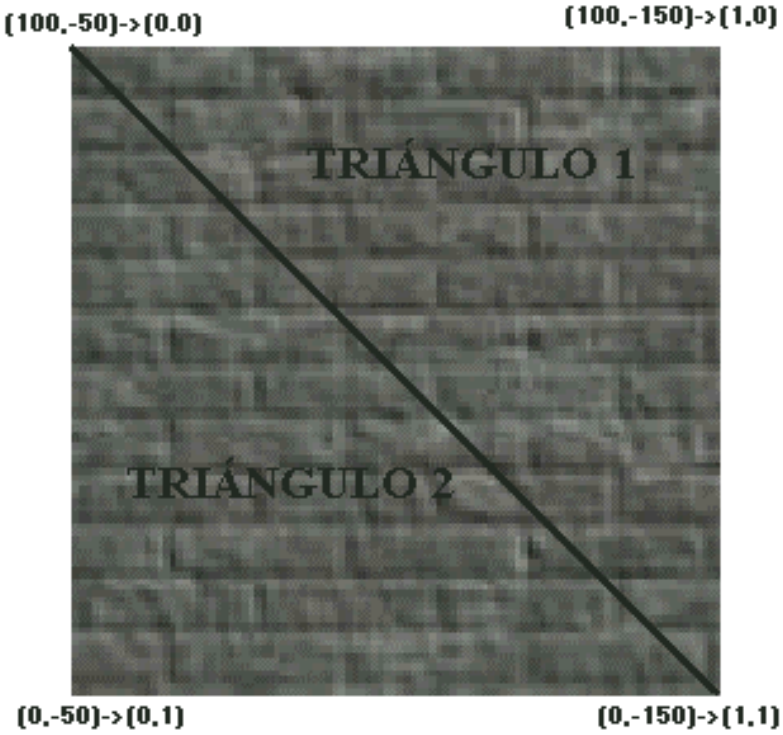
Para la inclusión de texturas debemos realizar unas pequeñas modificaciones en el código del programa. La primera de estas modificaciones está en el vértice, ya que ahora, la estructura que representa al vértice, no va a tener como componentes el color y la posición en el espacio, sino que el atributo color es sustituido por dos coordenadas para la textura: tu y tv, en el ejemplo.

Estas coordenadas de la textura varían de 0.0 a 1.0, y representan la posición de la textura en el eje X e Y, respectivamente. Es decir, los valores (0.0, 0.0) de tu y tv, representan la esquina superior izquierda de la textura y los valores (1.0, 1.0), la esquina inferior derecha. DirectX utiliza estas coordenadas para saber como colocar la textura en el polígono.

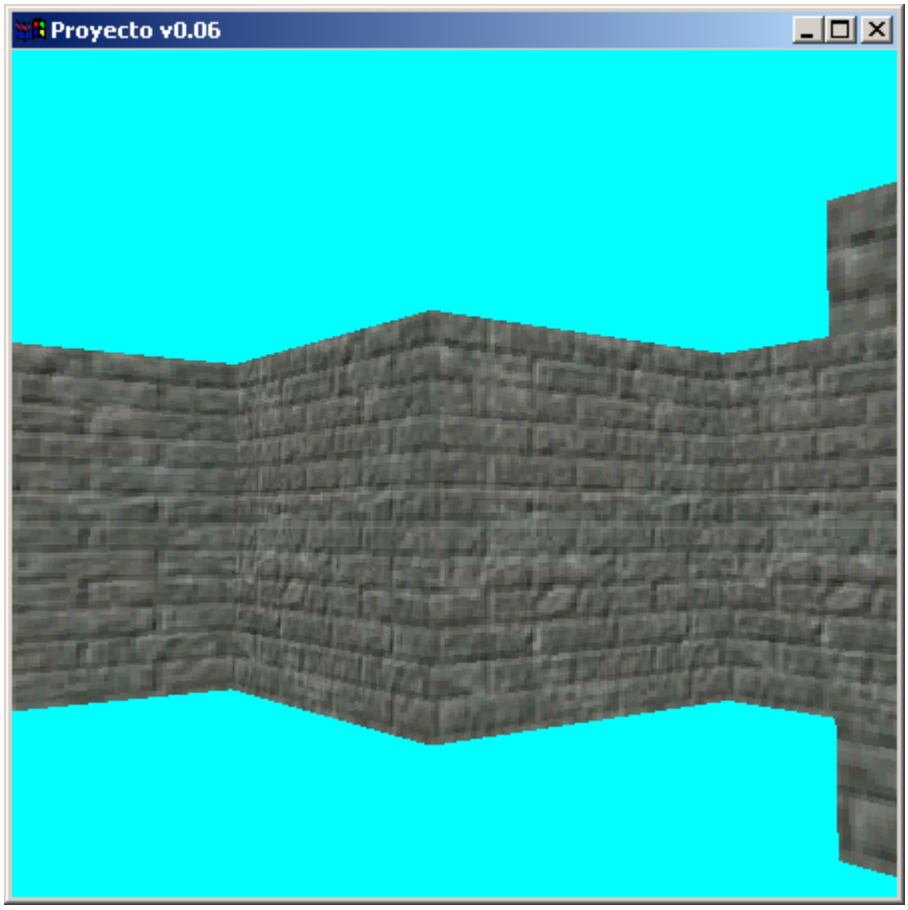
Vamos a explicar más claramente la teoría anterior mediante un ejemplo: la primera pared que se dibuja en nuestro escenario del proyecto. Toda pared está formada por dos triángulos rectángulos adyacentes por una de sus aristas. Las coordenadas Y y Z (al ser todos los puntos coplanarios despreciamos la coordenada X) de los tres vértices de cada uno de estos triángulos son los siguientes:

TRIÁNGULO 1: (0, -150), (100, -150), (100, -50)
TRIÁNGULO 2: (100, -50), (0, -50), (0, -150)

La siguiente figura representa todos los elementos anteriores: los dos triángulos rectángulos que forman la pared, las coordenadas de los 4 vértices y las coordenadas de la textura:



El resultado final de la aplicación de texturas a nuestro escenario generado a partir del árbol BSP es el siguiente:


Proyecto v0.06

Iluminación

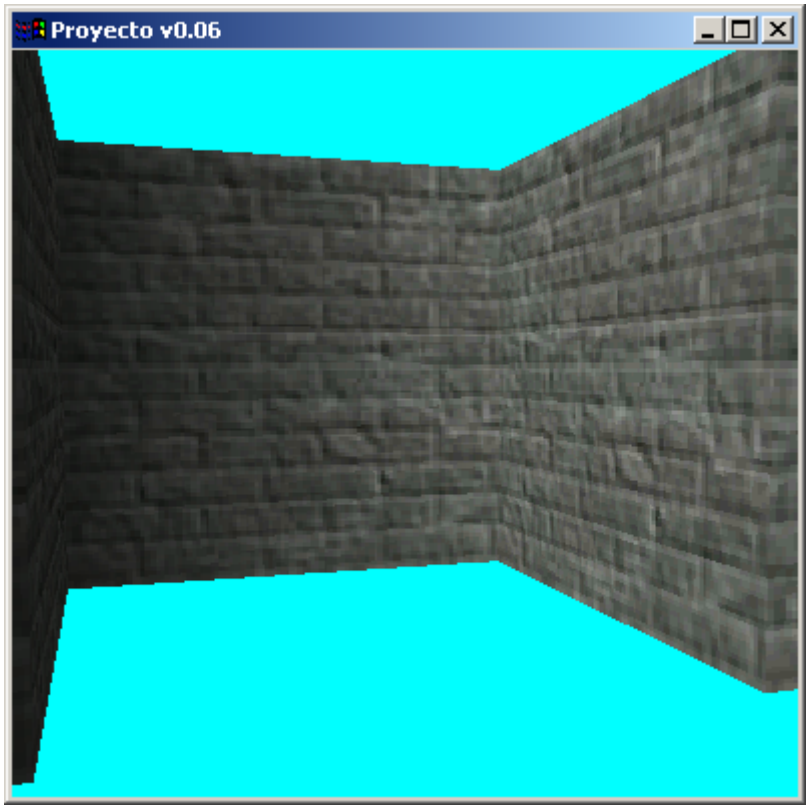
También se ha comenzado a tratar el tema de las luces, aunque al ser un apartado tan amplio y con tantas posibilidades, su estudio pormenorizado va a ser pospuesto hasta más adelante.

El uso de luces en nuestro proyecto consiste, básicamente, en añadir un nuevo procedimiento, llamado SetupLights(), que gestiona la configuración de las luces. En esta primera toma de contacto con la iluminación de DirectX, hemos optado por utilizar un único punto de luz (D3DLIGHT_POINT) que irradia en todas las direcciones, algo así como un pequeño sol.

DirectX se encarga de realizar todos los cálculos matemáticos necesarios para crear el efecto de iluminación sobre una superficie. No obstante, nosotros debemos indicar cual es el vector normal (perpendicular) a la superficie en la que se encuentra cada uno de los vértices. De este modo, DirectX calcula el grado de luz que incide sobre vértice del polígono, y usa esta información para realizar una interpolación lineal en cada uno de los puntos en el interior del polígono.

Claro, que para que DirectX sea capaz de calcular el grado de luz incidente sobre cada vértice de una superficie, debe conocer el "material" de que está hecho dicha superficie. Por ello, es necesario que el procedimiento SetupLights(),también genere y fije un tipo cualquiera de material y todos los atributos de color y ambiente necesarios.

Los resultados de aplicar el uso de texturas además de las luces en esta nueva versión del proyecto se pueden ver en la figura siguiente:



Implementación de la iluminación

Como hemos dicho anteriormente, esta nueva versión del proyecto, incluye tanto la utilización de texturas, como el uso de puntos de luz. Por ello, la explicación de las modificaciones realizadas incluye ambos temas.

El primer cambio en el código tiene lugar en el formato de los vértices, que ya no tiene la componente del color. En cambio, la nueva estructura almacena dos coordenadas para la textura, tu y tv, y el vector normal a la superficie. La componente restante es la posición espacial del punto. La estructura queda así:

```
	struct CUSTOMVERTEX
	{
		D3DXVECTOR3 posicion; // La posición espacial del vértice
		D3DXVECTOR3 normal;   // El vector normal a la superficie
		                                                    FLOAT
tu, tv;   // Las coordenadas para la textura
	};
```

Al usar las texturas, lo primero que tenemos que hacer es inicializar cada uno de los vértices en el buffer con los valores correctos. Esto se realiza en el procedimiento InitVB(), repasando todas posiciones del buffer:

```
        CUSTOMVERTEX g_Vertices[36];
                                                                    for(int
i=0;i<6;i++) {
                                                                    g_Verti
ces[i*6].tu=1;
                                                                    g_Verti
ces[i*6].tv=1;
                                                                    g_Verti
ces[i*6+5].tu=1;
                                                                    g_Verti
ces[i*6+5].tv=1;
                                                                    g_Verti
ces[i*6+1].tu=1;
                                                                    g_Verti
ces[i*6+1].tv=0;
                                                                    g_Verti
ces[i*6+2].tu=0;
                                                                    g_Verti
ces[i*6+2].tv=0;
                                                                    g_Verti
ces[i*6+3].tu=0;
                                                                    g_Verti
ces[i*6+3].tv=0;
                                                                    g_Verti
ces[i*6+4].tu=0;
                                                                    g_Verti
ces[i*6+4].tv=1;
                                                                    }

           g_Vertices[0].posicion=D3DXVECTOR3(100.0f,   0.0f,-150.0f);
           g_Vertices[0].normal=D3DXVECTOR3(0.0f,   0.0f,-150.0f);
                                    ...
                                    ...
```

Estas dos últimas instrucciones se repiten para todos los vértices para asignar correctamente las posiciones y normales de cada punto.

Para cargar la textura que vamos a usar, y que en este caso representa una pared de ladrillos, hacemos la siguiente llamada dentro del procedimiento InitVB():

```
if(  FAILED(  D3DXCreateTextureFromFile(  g_pd3dDevice,  "wall2.bmp",
&g_pTexture ) ) )
```

Por último dentro del procedimiento Render() realizamos las siguientes modificaciones:

1º. Adjudicar la textura a la escena actual:

```
g_pd3dDevice->SetTexture( 0, g_pTexture );
```

2º. Cambiar los estados de la escena de la textura:

```
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,
D3DTOP_MODULATE );

g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1,
D3DTA_TEXTURE );

g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2,
D3DTA_DIFFUSE );

g_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP,
D3DTOP_DISABLE );
```

El segundo punto a tener en cuenta es el de las luces. Para incorporar las luces al proyecto se ha creado un nuevo procedimiento llamado SetupLights(), que se encarga de colocar y configurar el punto de luz empleado en el escenario.

Así, lo primero que tenemos que hacer es crear un objeto de tipo D3DMATERIAL8, del que van a estar "formadas" las paredes. Es importante fijar materiales para las superficies, ya este tipo de objetos contienen las características para el reflejo de la luz y, en general, para la iluminación. En nuestro caso concreto hemos creado un material con estas características:

```
D3DMATERIAL8 mtrl;

                                                        ZeroMem
ory( &mtrl, sizeof(D3DMATERIAL8) );

                                                        mtrl.Di
ffuse.r = mtrl.Ambient.r = 1.0f;

                                                        mtrl.Di
ffuse.g = mtrl.Ambient.g = 1.0f;

                                                        mtrl.Di
ffuse.b = mtrl.Ambient.b = 1.0f;

                                                        mtrl.Di
ffuse.a = mtrl.Ambient.a = 1.0f;
g_pd3dDevice->SetMaterial( &mtrl ); //Fija el material
                                    //en nuestro "mundo"
```

El segundo paso es crear un objeto que contenga la luz. Este objeto es del tipo D3DLIGHT8, y es sobre este objeto sobre el que vamos a realizar todos los cambios para incorporar la luz. Para ello, primero reservamos y limpiamos memoria suficiente para el objeto, y, a continuación, definimos los parámetros que vayamos a usar para nuestro escenario. Estos parámetros son el tipo de luz, el color, el rango y la posición del foco o punto emisor.

```
ZeroMemory( &light, sizeof(D3DLIGHT8) );
light.Type       = D3DLIGHT_POINT;
light.Diffuse.r  = 1.0f;
light.Diffuse.g  = 1.0f;
light.Diffuse.b  = 1.0f;
light.Range      = 10000.0f;
light.Position   = D3DXVECTOR3(0.0f,50.0f,0.0f);
```

Finalmente sólo tenemos que incorporar el nuevo foco de luz a nuestro escenario y activarlo:

```
g_pd3dDevice->SetLight( 0, &light );
g_pd3dDevice->LightEnable( 0, TRUE );
```

Por último, no debemos olvidar que para poder incorporar luces a nuestro decorado, debemos indicar a DirectX que hay un foco de luz activo. Para ello, debemos configurar el estado de renderización de este modo:

```
g_pd3dDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
```

Mejoras en Luces y Texturas

El objetivo de esta versión era profundizar más en el uso de texturas y luces. Para ello principalmente se han consultado dos fuentes: el manual de la SDK de DirectX 8, y artículos que se pueden encontrar en http://www.gamedev.net .

En cuanto a las luces, si en la versión anterior habíamos incluido un punto de luz que irradiaba en todas las direcciones; ahora ya hemos quitado esta fuente de luz ya que cuantas menos fuentes de luz usemos en una aplicación, menos recursos del sistema consumirá. De este modo para lo que nos proponemos en un futuro a medio plazo la luz ambiente cumple con nuestro objetivo a la perfección y sólo incluiremos otras fuentes de luz para casos concretos como puede ser la inclusión de un objeto linterna o algo parecido.

En lo referente a las texturas, nuestro objetivo era incluir más texturas en la aplicación. Esto lo hemos conseguido con una técnica bastante sencilla a simple vista, que consiste en unir varias texturas en un mismo archivo para luego dentro del programa elegir la textura por medio de las coordenadas explicadas anteriormente.

Figura 1.

Para explicar mejor esto, podemos ver la figura 1, que es un archivo donde almacenamos las cuatro texturas que vamos a poder usar en esta versión de la aplicación, y se trata de cuatro texturas unidas en un mismo archivo, así si antes un textura tenía 256x256 pixels, la nueva textura con cuatro texturas en su interior tendrá 512x512 pixels (ver figura 2).

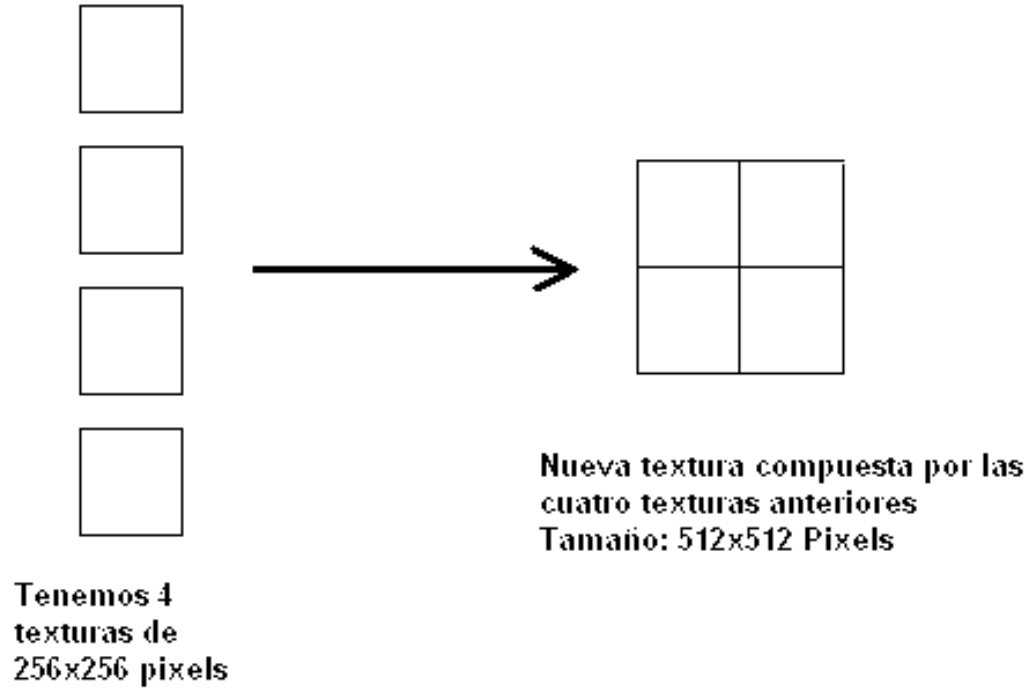


Figura 2.

Ahora sólo queda explicar cómo se accede a las texturas, los cual resulta bastante sencillo. Recordemos que antes para definir en un vértice a qué lugar de la textura correspondía usábamos dos coordenadas u y v, las cuales iban desde 0 hasta 1, indicando el (0,0) la esquina superior izquierda, y el (1, 1) la esquina inferior derecha. Pues bien, ahora va a ser igual, ya que para acceder a la textura necesitamos las mismas coordenadas, pero, si usáramos las mismas coordenadas que en la versión anterior en una pared se visualizarían las cuatro texturas a la vez, por lo que si sólo queremos poner la textura superior izquierda, habrá que limitar las coordenadas u y v a 0.5. De este modo conseguimos la visualización de la primera textura, y para las siguientes es muy similar.

El resultado obtenido se puede ver en la siguiente figura.

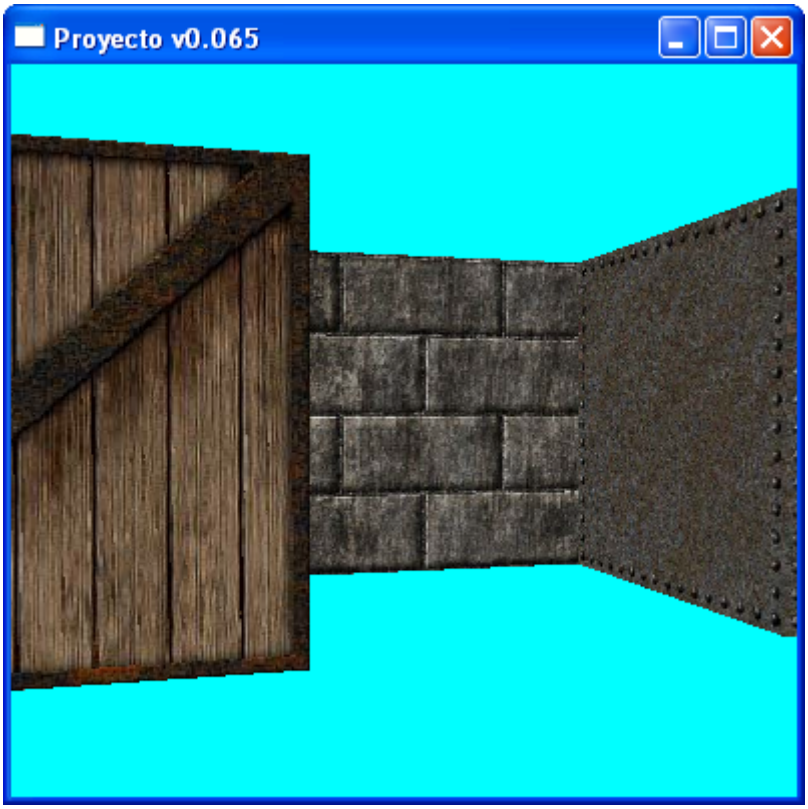


Figura 3.

En la figura 3 se ven sólo 3 texturas, y de hecho sólo hemos empleado 3 de las 4 texturas posibles, pero se podrían emplear las 4, y se podría extender a un número indeterminado, aunque con alguna pega que vamos a exponer a continuación.

Tamaño de las texturas

Existe un pequeño problema con el tamaño de las texturas, ya que la carga en memoria y el uso posterior de las texturas depende de la memoria existente en el sistema.

De este modo el tamaño del la memoria presente en la tarjeta gráfica así como el tamaño de la memoria cache de segundo nivel, va a ser determinante para el buen funcionamiento de la aplicación. Así un tamaño menor de las texturas (128x128 pixels) va a ser más eficiente ya que va a poder ser conservado en la memoria caché en la mayoría de los casos.

Entonces nos podemos preguntar: ¿por qué hasta ahora no hemos usado este tamaño de texturas?. La respuesta es que ese tamaño de texturas están muy poco definidas y sólo nos servirían para repetirlas muchas veces a modo de mosaico, por ejemplo en las baldosas del suelo. Esto es cierto, pero también influyen otros factores, y es que el tamaño de 256x256 pixels es el

más rápido para las tarjetas gráficas y para DirectX, y reduce el intercambio de texturas. De hecho DirectX recomienda el uso de la técnica que hemos explicado anteriormente para juntar 4 texturas de 128x128 pixels en una de 256x256, pero también advierte que no abusemos de esta técnica porque a menor tamaño de texturas también aumenta la eficiencia por el lado de la memoria caché.

Parece evidente que lo que hemos hecho con las texturas en esta prueba no es lo más eficiente, y ya que estamos cuidando la eficiencia desde el primer momento (por ejemplo en las luces), pues vamos a plantear la solución más sencilla e intuitiva.

El usar una textura de 512x512 pixels no es muy normal, ya que por lo explicado anteriormente las texturas más eficientes son las de 256x256 pixels de tamaño. De este modo vamos a intentar usar este último tamaño de texturas y sólo aplicar la técnica de unir texturas para las pequeñas (128x128 pixels). Para esto, si queremos meter más texturas sólo hay que intercambiar la textura actual cada vez que queramos dibujar algo con una textura diferente con la primitiva `SetTexture`.

Esto plantea un nuevo problema, y es que cambiar la textura implica un retardo por el intercambio de memoria que implica. La recomendación que se hace en el manual de DirectX es la de intentar juntar el mayor número de figuras que tengan la misma textura para dibujarlas a la vez, y así intercambiar el menor número de veces la textura empleada.

Sin embargo esta recomendación no sabemos si la vamos a poder cumplir, ya que el dibujado de objetos y figuras en nuestra aplicación va a ser gestionada por el algoritmo BSP, y éste no se preocupa de la eficiencia en cuanto a cargas de texturas.

Hemos desarrollado una nueva aplicación que implementa el cambio de texturas. Así ahora cambiamos la textura a usar en la cuarta pared, y luego la volvemos a cambiar para pintar las 2 paredes restantes.

Figura 4.

Estamos ante un problema de difícil solución, ya que en el fondo no es más que una relación inversa entre rendimiento y calidad. Una solución simple es la de desarrollar la aplicación, y luego ya preocuparnos por la eficiencia, para ver los requerimientos mínimos. También hay que tener en cuenta que estos problemas de rendimiento se van a dar al usar muchas texturas y muchos objetos y polígonos, por lo que en un principio no van a aparecer hasta que tengamos grandes mapas y muchos enemigos y objetos móviles.

En principio vamos a tomar esta solución ya que lo que más nos preocupa ahora es desarrollar un algoritmo BSP eficiente y la inclusión de otros objetos móviles.

Implementación de las mejoras

El código para esta nueva versión no ha variado mucho, ya que en cuanto a las luces, hemos quitado el punto de luz y hemos subido la luz ambiente para compensar la pérdida de luz.

```
g_pd3dDevice->SetRenderState( D3DRS_AMBIENT, 0x00f0f0f0 );
```

En las texturas no ha habido grandes variaciones, ya que simplemente hemos cambiado la carga de texturas para que usemos dos archivos de texturas para posteriormente poder cambiar la textura actual.

Para cargar las dos texturas:

```
                                                                    if(
    FAILED( D3DXCreateTextureFromFile( g_pd3dDevice, "textura.jpg",
    &g_pTexture ) ) )
                                                                    return
    E_FAIL;
                                                                    if(
    FAILED( D3DXCreateTextureFromFile( g_pd3dDevice, "floor.jpg",
    &g_pTexture2 ) ) )
                                                                    return
    E_FAIL;
```

Y el cambio de la textura actual se hace como sigue:

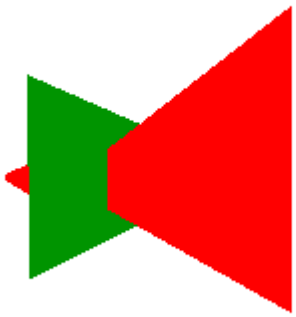
```
                                                                    //Carga
mos la primera textura
      g_pd3dDevice->SetTexture( 0, g_pTexture );
                                                                    //Ahora
dibuja la primera lista de triangulos
                                                                    g_pd3dD
evice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 6);
                                                                    //Cambi
amos la textura para dibujar otra pared
                                                                    g_pd3dD
evice->SetTexture( 0, g_pTexture2 );
                                                                    //Dibuj
a una sola pared con la textura anterior
                                                                    g_pd3dD
evice->DrawPrimitive( D3DPT_TRIANGLELIST, 18, 2);
                                                                    //Volve
mos a cambiar la textura
                                                                    g_pd3dD
evice->SetTexture( 0, g_pTexture );
                                                                    //Dibuj
amos las dos paredes restantes
                                                                    g_pd3dD
evice->DrawPrimitive( D3DPT_TRIANGLELIST, 24, 4);
```

Compilador de Árboles BSP

En esta nueva versión del proyecto hemos desarrollado un compilador de árboles BSP, una utilísima representación de escenarios tridimensionales que comenzó a usarse en “Doom” (uno de los primeros shoot’em up subjetivos y, quizá, el más famoso) y todavía hoy, 9 años más tarde, sigue empleándose en los últimos lanzamientos.

Durante el desarrollo de un motor 3D para un juego, nos encontramos el grave problema de dibujar los polígonos en orden, de tal modo que los polígonos situados en un plano posterior no se superpongan a aquellos que están más cerca del punto de vista. Este problema puede ser solventado utilizando un Z-buffer, pero sólo en el caso de que nuestro escenario no contenga objetos transparentes, porque estos objetos deben ser renderizados antes que los polígonos situados detrás. Además la utilización del Z-buffer tampoco resuelve situaciones como la siguiente, en la que los dos polígonos están a la vez cerca y lejos del punto de vista:

Imposible ordenar con un Z-Buffer



Por si esto fuera poco, cuando sea necesario crear una rutina para el control de colisiones (no atravesar paredes, comprobar si las balas impactan...), habría que recorrer todos los polígonos del escenario comprobando, para cada uno de ellos, si hay colisión o no.

Todo lo anterior nos hace pensar en las ventajas que obtendríamos si consiguiéramos una buena ordenación de los polígonos de nuestro escenario. Esto es precisamente de lo que se encargan los algoritmos BSP.

La utilización de árboles y algoritmos BSP supone una ordenación en tiempo de

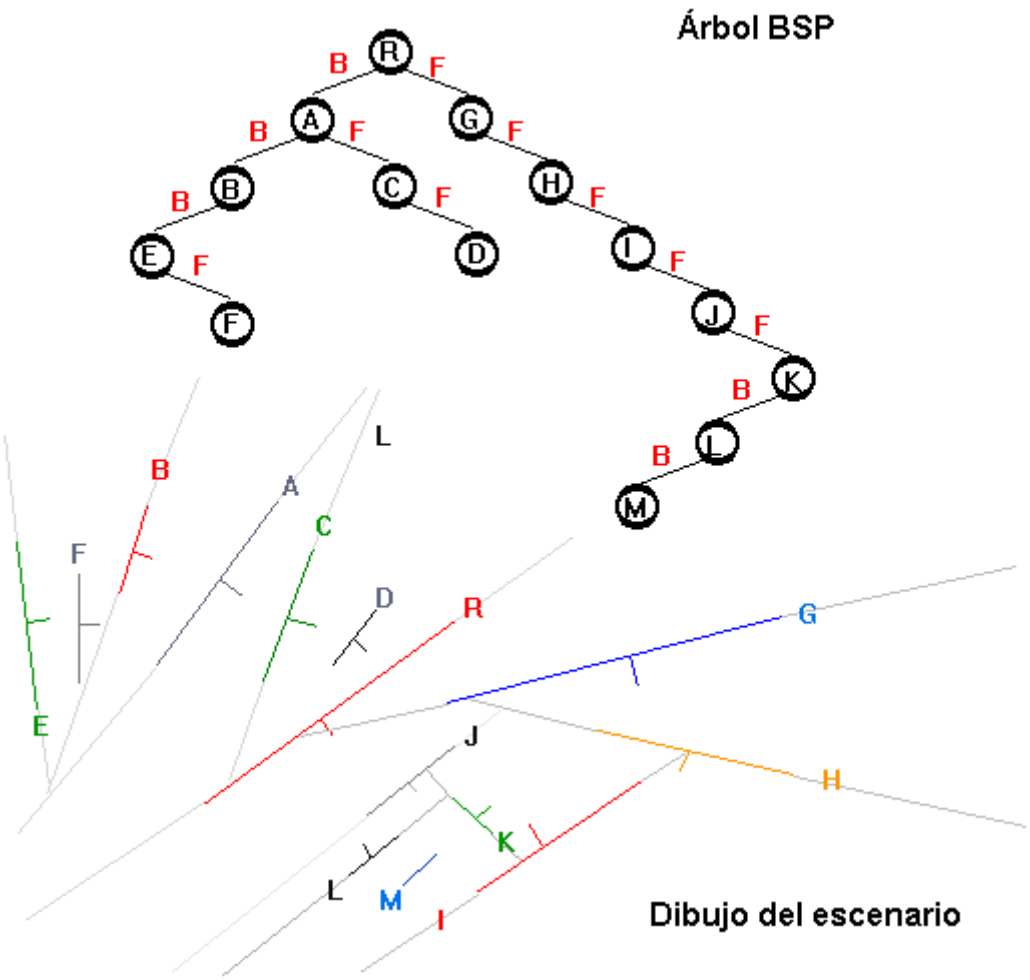
compilación (no en tiempo de juego) de todos los polígonos, lo que implica, no sólo el correcto dibujo de todos los polígonos del escenario, sino también el rechazo de todos aquellos triángulos que se encuentren fuera del plano de visión, así como un recorrido mucho más eficiente de la lista de polígonos. Queda clara la ganancia en velocidad que se obtiene cuando, con unas comprobaciones simples, podemos rechazar unos cuantos miles de polígonos de nuestro escenario por encontrarse fuera del rango de visión.

Pero vamos a explicar con un poco más de detalle en qué consisten y cómo se construyen estos árboles BSP.

Los árboles BSP son árboles binarios muy parecidos a los árboles que se construyen, por ejemplo, al hacer una ordenación in-orden de una lista de números. La diferencia estriba en que, en el caso de BSP, los números son polígonos y la función de ordenación es la posición relativa de cada polígono, delante o detrás, con respecto al padre. Cada nodo BSP debe almacenar, además del polígono que representa, dos punteros: uno a un polígono situado enfrente de éste y otro, a uno situado detrás.

Al construir un árbol BSP, pasamos, inicialmente, la lista de polígonos que forman nuestro escenario. El algoritmo para crear el árbol BSP elige un polígono para que actúe como divisor, y distribuye el resto en dos listas diferentes: una con todos los polígonos situados delante del divisor y otra que contiene a todos aquellos situados detrás. Si el plano divisor se cruza con algún polígono, éste último es cortado por el plano divisor y cada una de las partes es situada en su lista correspondiente y tratada como un polígono nuevo. Después, el algoritmo hace una llamada recursiva con cada una de las listas y repite el proceso hasta que todos los polígonos ocupan uno y sólo uno de los nodos del árbol BSP.

En la siguiente figura se puede apreciar el funcionamiento del algoritmo BSP. Las letras de la A a la M representan todos los polígonos de la lista inicial y la letra R, el primer plano tomado como divisor. Tras la primera llamada a la función la lista inicial se separa en dos, una de la A a la F y otra de la G a la M. A partir de ahí, los divisores son elegidos en orden alfabético y le proceso se repite hasta construir el árbol completo.



En la documentación correspondiente a la entrega 0.051 se detalla, mediante un ejemplo en 6 pasos, la progresiva construcción del árbol BSP y las decisiones que toma en cada momento el algoritmo.

Claro que todo lo dicho anteriormente es solo una pequeña introducción a los algoritmos BSP, ya que en un nivel clásico de un shoot'em up subjetivo, los muros no son simples polígonos que por un lado se ven y por otro no (recordemos el "culling" de DirectX), sino que van a ser espacios sólidos formados por al menos 4 polígonos como este:

Estructuras de datos de la implementación BSP.

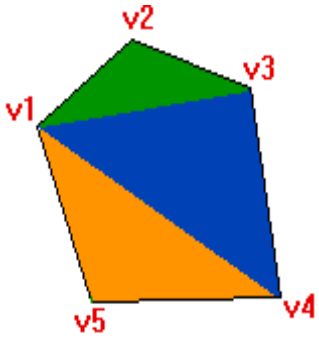
La estructura de datos CUSTOMVERTEX ,estructura facilita por DirectX 8 para ser adaptada a las necesidades del usuario, contiene los datos relativos a un vértice definido por el usuario del SDK. En este caso se adecúa a nuestras necesidades almacenando la posición (en formato estándar D3DXVECTOR3 de DirectX), el color y las coordenadas de la textura (el significado de estas coordenadas se detalló en la entrega correspondiente al uso de texturas).

```
struct CUSTOMVERTEX
{
                                                    D3DXVEC
TOR3 posicion;
      DWORD
                                                    color;
                                                    FLOAT
tu, tv;
};
```

La estructura de datos polígono es fundamental en todo el desarrollo, por ello explicaremos con detalle cada uno de sus atributos:

```
struct Poligono {
                                                    CUSTOMV
ERTEX listaVertices[10];
                                                    D3DXVEC
TOR3 normal;
                                                    int
numVertices;
                                                    int
numIndices;
                                                    int
listaIndices[24];
                                                    Poligon
o *sig;
};
```

Esta estructura Polígono almacena una figura poligonal que puede tener hasta 10 vértices. En el ejemplo se muestra un polígono de 5 vértices:



A partir de la lista de vértices podemos dividirlo en los triángulos que lo forman, resultando 3 triángulos que, en realidad, son los que se van a dibujar mediante las primitivas de dibujo de DirectX. Si mandamos dibujar 3 triángulos por separado deberíamos enviar 9 vértices en el caso del ejemplo, pero esto no es necesario si mantenemos una lista de índices en la que se diga en qué orden se dibujan los vértices, en el ejemplo la lista de índices sería: v1,v2,v3, v1,v3,v4, v1,v4,v5. De acuerdo a la documentación del SDK de DirectX, el uso de vértices compartidos acelera el tratamiento de los mismos, además de facilitarnos el dibujo de las paredes (y en general cualquier forma poligonal). Por ejemplo, sólo es necesario darle los vértices de un rectángulo para que la función adecuada calcule los 2 triángulos que forman el polígono. Para n vértices el número necesario de índices a guardar es (n-2)*3, el campo listaÍndices se ocupa de guardar los índices asociados a listaVértices. Guardamos también el número de vértices y de índices implicados en este polígono.

El polígono debe guardar su normal, para poder conocer su orientación, además de facilitar la comprobación de si un punto (u otro polígono) está delante o detrás.

Por último un puntero al siguiente polígono para formar la lista enlazada de polígonos que se entregará a la función encargada de la construcción del árbol BSP.

Funciones en la implementación BSP.

```
CUSTOMVERTEX creaVertice(D3DXVECTOR3 vec,DWORD color, FLOAT tu,
                                    FLOAT tv);
```

Descripción:

Construye la estructura de datos que contendrá un vértice.

Parámetros de entrada:

- Vec: La posición en el formato estándar de DirectX
- Color: Color del vértice en formato RRGGBB (8 bits para cada color)
- Tu: coordenada x de la textura
- Tv: coordenada y de la textura

Parámetros de salida:

- Retorno de la función: La estructura de datos customvertex tal cómo ha sido escrita en las estructuras de datos.

```
int clasificarPunto (D3DXVECTOR3 *punto,Poligono *plano);
```

Descripción:

Eligiendo un vértice cualquiera del polígono, calculamos el vector dirección de la recta que une el vértice anterior y el punto a clasificar. Si hallamos el producto vectorial de la dirección calculada y la normal del polígono, el ángulo resultante determina si está delante o detrás. Si es

mayor de 90º el punto está detrás del polígono, si es menor que 90º el punto está enfrente. Si ningún caso se cumple, es decir, el resultado es cero o extremadamente cercano a 0 (hemos de pensar que tratamos con tipos float de C++), entonces el punto está en el plano.

Parámetros de entrada:

- Punto: El punto a clasificar.
- Plano: El polígono (que si lo extendemos hasta el infinito podríamos ver como un plano).

Parámetros de salida:

- Retorno de la función: La clasificación del punto definida por los valores (enteros): CP_COINCIDENTES 0, CP_DELANTE 1, CP_DETRAS 2.

```
int clasificarPoligono (Poligono *poligono,Poligono *plano);
```

Descripción:

Clasificamos un polígono respecto a otro (o al plano que forma, si se prefiere). Para ello, se realiza la operación anterior clasificarpunto, pero para cada vértice del polígono, de forma que podemos tener 4 casos. Que el número de vértices delante sea igual al número de vértices del polígono, lo que significa que el polígono está delante. Igual para el caso de que esté completamente detrás. Igual es el tratamiento i el número de vértices coincidentes es igual al número de vértices del polígono (dicho de otra forma, todos los vértice son coincidentes), en este caso los polígonos son coincidentes. Finalmente, si no se cumple ninguno de los casos anteriores sabemos que los polígonos se cortan (o intersectan).

Parámetros de entrada:

- Polígono: El polígono a clasificar.
- Plano: El otro polígono (o plano, como se prefiera) contra el que se va a clasificar.

Parámetros de salida:

- Retorno de la función: : La clasificación del polígono definida por los valores (enteros): CP_COINCIDENTES 0, CP_DELANTE 1, CP_DETRAS 2 o CP_CORTAN 4.

```
void inicializarPoligonos(NodoBSP *raízArbolBSP);
```

Descripción:

Inicializar Polígonos es en realidad un sencillo editor de niveles de juego. Llama a creavertice, anyadirPoligono y finalmente a construyeArbolBSP para la construcción del árbol BSP. A partir de un nivel de juego “dibujado” en una matriz de 20 x 40 en la que un ‘1’ indica un bloque sólido, ‘0’ el camino libre y ‘2’ y ‘3’ bloques sólidos dispuestos en diagonal para permitir construir paredes orientadas al noreste y noroeste respectivamente, se construyen las listas de vértices que se entregarán a añadirPolígono para construir la lista de polígonos que, posteriormente será procesada por construyeArbolBSP.

Parámetros de entrada:

- Ninguno.

Parámetros de salida:

- RaízArbolBSP: Puntero al nodo raíz del árbol BSP construido.
- Retorno de la función: Ninguno.

```
Poligono *anyadirPoligono(Poligono *padre,CUSTOMVERTEX *listaVertices,
                          int numVertices);
```

Descripción:

Construye un polígono nuevo a partir de una lista de vértices y su número añadiéndolo como polígono hijo del polígono padre suministrado. Se crea la lista de índices tal como se ha descrito en la estructura de datos polígono y se rellenan el número de vértices y el número de índices con la fórmula (nvértices-2)*3. A partir de tres puntos del plano se calculan dos vectores que, a su vez, sirven para determinar la normal al polígono.

Parámetros de entrada:

- Padre: Puntero al polígono padre del polígono a crear.
- ListaVertices: Lista enlazada de vértices del polígono a construir.
- NumVertices: EL número de vértices del polígono.

Parámetros de salida:

- Retorno de la función: Puntero al polígono construido.

```
void dividirPoligono(Poligono *poligono, Poligono *plano,
                     Poligono *parteDelantera, Poligono *parteTrasera);
```

Descripción:

Realiza la partición del polígono que es cortado por un plano (otro polígono), dividiéndolo en dos (parteDelantera y parteTrasera). Esto impide que en un árbol BSP se de el problema con el que se abre esta entrega de documentación, en la que el polígono rojo (el más grande) está a la vez delante y detrás del verde que lo corta. Se determina si cada uno de los vértices del polígono debe pertenecer a los vértices que quedan delante o a los de detrás utilizando la función ya comentada clasificarPunto. Cada una de las aristas del polígono se trata como una recta separada. Para realizar el corte se ayuda de la función interseccionRectaPlano, con la que se determina la intersección entre la recta correspondiente a cada una de las aristas del polígono con el plano de corte. De esta forma hemos determinado que vértices deben formar parte del polígono parteDelantera y los que deben formar el polígono parteTrasera.

Parámetros de entrada:

- Polígono: El polígono a cortar.
- Plano: El plano de corte (el polígono contra el que se quiere cortar).

Parámetros de salida:

- ParteDelantera: Nuevo polígono creado con la mitad del polígono que queda delante.
- ParteTrasera: Igual para la otra mitad.
- Retorno de la función: Ninguno.

```
Poligono* seleccionarMejorDivisor(Poligono *listaPoligonos);
```

Descripción:

Como es habitual en todos los árboles binarios, el rendimiento óptimo se obtiene cuando se logra un árbol equilibrado. Por ello debemos intentar elegir bien el primer polígono que efectuará la primera división del espacio. Esta función recorre una vez la lista de polígonos evaluando cada uno de ellos contra el resto. A cada uno evaluado se le asigna un número que es la diferencia entre los que están enfrente y los que quedan detrás, obviamente cuánto menor sea este número más equilibrado estará el árbol, por lo que nuestro objetivo será elegir el polígono con el valor mínimo. Por otra parte, es mejor que los polígonos no resulten cortados, puesto que el número de polígonos se multiplica por dos. Penalizando en el valor calculado que cada polígono evaluado realice muchos cortes limitamos el número de cortes a realizar. En este caso se ha elegido una función heurística en la que se multiplica por 8 el número de cortes realizados, impidiendo (ya que obtendría un valor alto) que un polígono que corta a muchos sea elegido como divisor.

Parámetros de entrada:

- ListaPolígonos: La lista enlazada de polígonos del escenario.

Parámetros de salida:

- Retorno de la función: El polígono elegido.

```
void construyeArbolBSP(NodoBSP *nodoActual,Poligono *listaPoligonos);
```

Descripción:

Construye recursivamente, el árbol binario de nodos polígono siguiendo las reglas descritas en la documentación de la entrega 0.051, en la que se muestra en varios ejemplos paso a paso la construcción del árbol.

Parámetros de entrada:

- NodoActual: Apunta al nodo raíz del árbol a crear.
- Polígono: Lista enlazada de polígonos del escenario.

Parámetros de salida:

- Retorno de la función: Ninguno.

```
bool interseccionRectaPlano (D3DXVECTOR3 *inicioRecta,
                        D3DXVECTOR3 *finRecta, D3DXVECTOR3 *puntoPlano,
                        D3DXVECTOR3 *normalPlano,
                        D3DXVECTOR3 *interseccion, float *porcentaje);
```

Descripción:

Calcula el punto de intersección, si lo hay, entre un plano, representado por un punto y el vector normal y una recta, representada por dos puntos. Para ello se calcula el tamaño del vector que va de inicioRecta a finRecta, si este vector está en el mismo lado del plano entonces no hay corte. Si no es así, calculamos un vector que una cualquier punto del plano con el punto inicial de la recta, y hallamos su tamaño, si la distancia al plano es menor que la longitud de la recta entonces se cortan. Dividiendo la distancia al plano por la longitud de la recta

obtenemos el índice que variando de 0 a 1 nos dice dónde se cortan la recta y el plano, por ejemplo la distancia al plano es 1 y la longitud de la recta es 2, se cortarán en el 0.5 (a la mitad de la recta). Podemos calcular el punto de corte escalando el vector dirección de la recta al porcentaje calculado y sumándolo al punto inicial de la recta.

Parámetros de entrada:

- InicioRecta: El punto de inicio de la recta.
- FinRecta: El punto de final de la recta.
- PuntoPlano: Un punto del plano.
- NormalPlano: La normal del plano.

Parámetros de salida:

- Intersección: El punto de intersección plano-recta.
- Porcentaje: Varía de 0 (inicio de la recta) a 1 (fin de la recta) y determina el punto de la recta en el que ha ocurrido la intersección. Resulta útil cuando se está realizando el corte de un polígono, (recordemos que el corte de polígono hace varias llamadas a esta función), para determinar cómo distribuir una textura de forma que quede exactamente igual que si el polígono nunca hubiera sido separado en dos. Si no lo tuviéramos en cuenta la textura aparecería "rota" al cortar el polígono, justo en el comienzo del segundo polígono obtenido tras el corte.
- Retorno de la función: Verdadero si hay intersección, Falso en caso contrario.

```
void recorrerArbolBSP(NodoBSP *nodoActual,D3DXVECTOR3 *posicionCamara,
                      CUSTOMVERTEX *buffer,int *tamanyo);
```

Descripción:

Realiza recursivamente el recorrido del árbol de polígonos, partiendo del punto focal determinado por nuestra posición (la posición de la cámara), y llenando el buffer con los vértices a dibujar.

Parámetros de entrada:

- NodoActual: La raíz del árbol de polígonos del escenario.
- PosicionCamara: El punto en el que nos encontramos.

Parámetros de salida:

- Buffer: Puntero al inicio del buffer en el que se guardan todos los vértices customvertex que deben ser dibujados.
- Tamaño: Tamaño del buffer anterior en número de vértices. Nos indicará cuando parar de dibujar, puesto que el número de primitivas triángulo que pondremos en el buffer de vértices para que DirectX las dibuje será el tamaño/3.
- Retorno de la función: Ninguno.

Detección de colisiones

La detección de colisiones la realiza la función lineaDeVision. Esta función determina si existe un sólido entre el punto de inicio y el punto de fin (si hay una línea de visión entre el inicio y el fin). Para ello utiliza el atributo

esSolido de cada nodo hoja del árbol BSP, devolviendo verdadero si esSolido es falso, y devuelve falso si encuentra esSolido a verdadero.

```
bool lineaDeVision (D3DXVECTOR3 *inicio, D3DXVECTOR3 *fin,
                                NodoBSP *nodoActual);
```

Parámetros de entrada:

- Inicio: El punto de inicio.
- Fin: El de fin de la línea de visión (lo que podemos ver o no).
- NodoActual: El nodo raíz del árbol BSP.

Parámetros de salida:

- Retorno de la función: Verdadero si no hay obstáculos sólidos, falso en caso contrario.

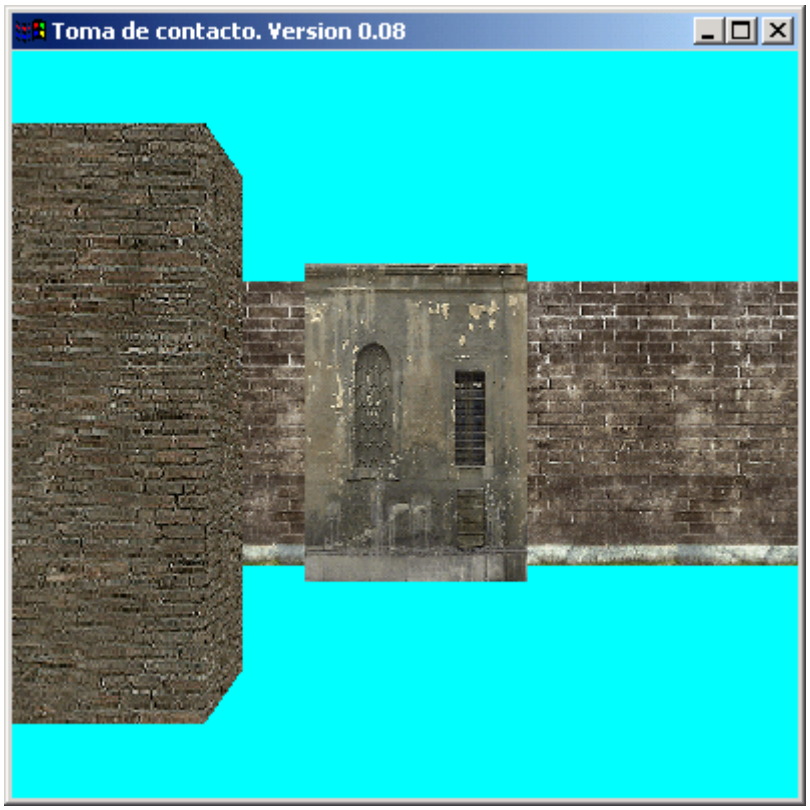
Cuando se recoge del teclado una petición de movimiento de avance, primero se comprueba si la nueva posición de la cámara resultante de esa petición de avanzar tiene línea de visión respecto del punto que ocupamos actualmente. Si hay línea, es decir, dando el punto de inicio y el punto al que queremos ir, la función responde verdadero, entonces el movimiento se efectúa, y no hace nada en caso contrario.

Mapa Selector de Texturas

Hemos incluido un mapa selector de texturas que indica la textura que corresponde a cada muro. Este mapa se "dibuja" en una matriz de 20 x 40 en la que un número entero indica la textura seleccionada para el muro que reside en la posición correspondiente.

El método que usamos para el cambio de texturas consiste en seleccionar la textura a usar en cada momento. Este método no es el más eficiente, ya que no se preocupa de agrupar los polígonos con texturas iguales para representarlos a la vez. Esto ocurre porque el algoritmo BSP no se preocupa de la eficiencia en la selección de las texturas, a cambio presenta otras ventajas esenciales, como la detección de colisiones y la correcta presentación de los polígonos.

Pantalla de muestra del uso de diversas texturas:

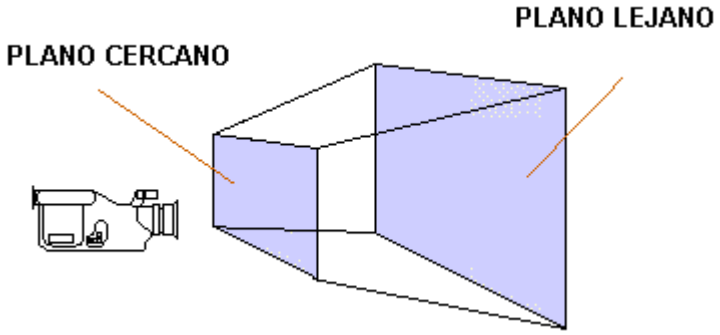


Rechazo por Cono de Visión

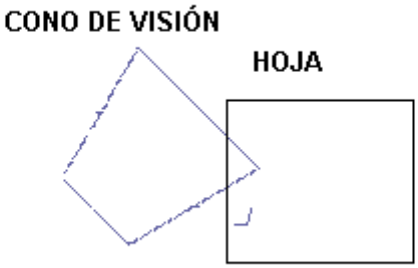
La anterior entrega de este proyecto consistía en aprovechar las ventajas de los árboles BSP para crear un nuevo tipo de árbol, los árboles sólidos con hojas, que reducían el número de polígonos que se mandan a dibujar. La idea era clasificar los polígonos en hojas, de forma que cuando la cámara se encontraba en una de esas hojas, sólo se mandaban a dibujar los polígonos de las hojas visibles desde la posición de la cámara.

En esta nueva entrega se ha reducido aún más el numero de polígonos que se dibujan rechazando todas las hojas que se van a dibujar y que se encuentren fuera del cono de visión.

Este proceso consta de dos partes bien definidas: cálculo de los planos que forman el cono de visión y rechazo de las hojas que se encuentren fuera del mismo. En la primera parte del proceso lo único que hay que hacer es calcular los seis planos que conforman el cono de visión, a saber, el plano cercano, lejano, superior, inferior, derecho e izquierdo.

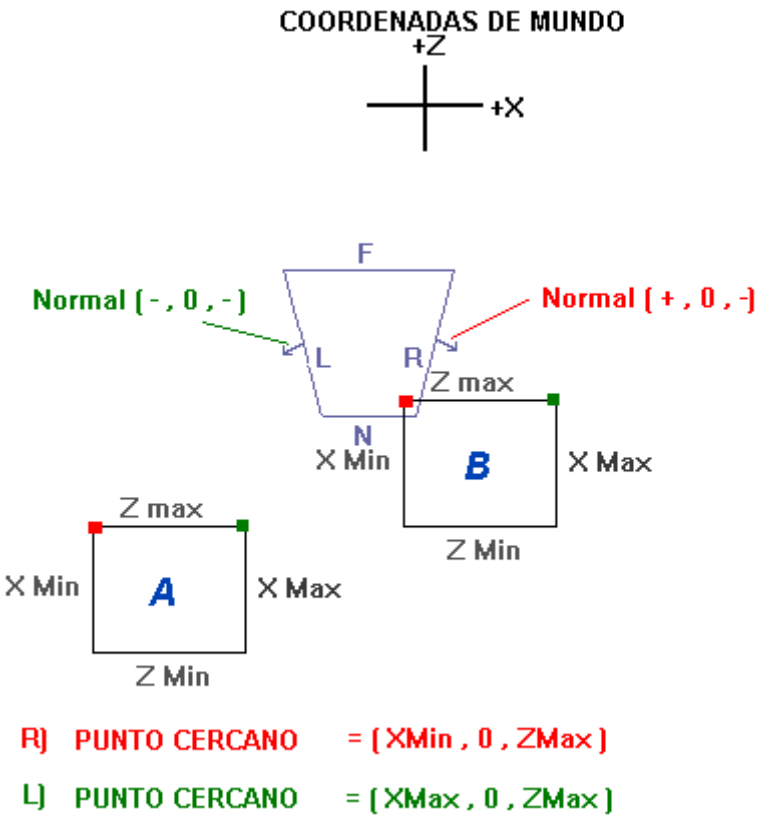


La segunda parte del proceso consiste en comprobar si la hoja que va a ser dibujada se encuentra completamente fuera del cono de visión, para así decidir si se dibuja o no. Si por el contrario, la hoja se encuentra total o parcialmente dentro de la hoja, se mandará a dibujar. Este proceso de decisión no consiste simplemente en comprobar si alguno de los vértices que definen los polígonos de la hoja se encuentra dentro del cono, como demuestra la figura siguiente, en la que podemos ver que todos los vértices de la hoja se encuentran fuera del cono y aún así la hoja debe ser dibujada:



La solución a este problema es sencilla y consiste en calcular el vértice de la caja minimal de la hoja que esté más cerca del cono de visión. Si este punto se encuentra delante de alguno de los planos del cono, la hoja estará fuera del cono.

La siguiente figura explica como funciona el método:



En la figura observamos que el punto más cercano al cono de la caja minimal A es el punto verde. Como este punto se encuentra delante del plano L del cono, deducimos que la hoja A esta fuera del cono. En cambio, para la hoja B,

encontramos que el punto más cercano al cono (el rojo) se encuentra detrás de TODOS los planos del cono, por lo que podemos afirmar que la hoja B se encuentra dentro del cono.

Implementación del rechazo por cono de visión

Para implementar el anterior método hemos creado una nueva estructura, los planos de visión, que representan un plano del cono de visión almacenando la normal del plano y la distancia de éste al centro de coordenadas.

```
struct PLANOSVISION
{
                                                    float distancia;
      D3DXVECTOR3 normal;
};
```

El siguiente paso es calcular los planos del cono de visión. Para ello creamos el siguiente método:

```
bool hojaEnConoDeVision (long hoja);
```

En esta función combinamos las matrices de mundo y de proyección y obtenemos todos los datos necesarios para crear los seis planos.

Con todos estos datos, solo basta comprobar si la hoja está o no dentro del cono, para lo que usaremos la siguiente función:

```
bool hojaEnConoDeVision (long hoja)
```

Esta función lo único que hace es calcular el punto más cercano de la caja minimal al cono y luego comprobar si ese punto se encuentra delante de alguno de los planos del cono

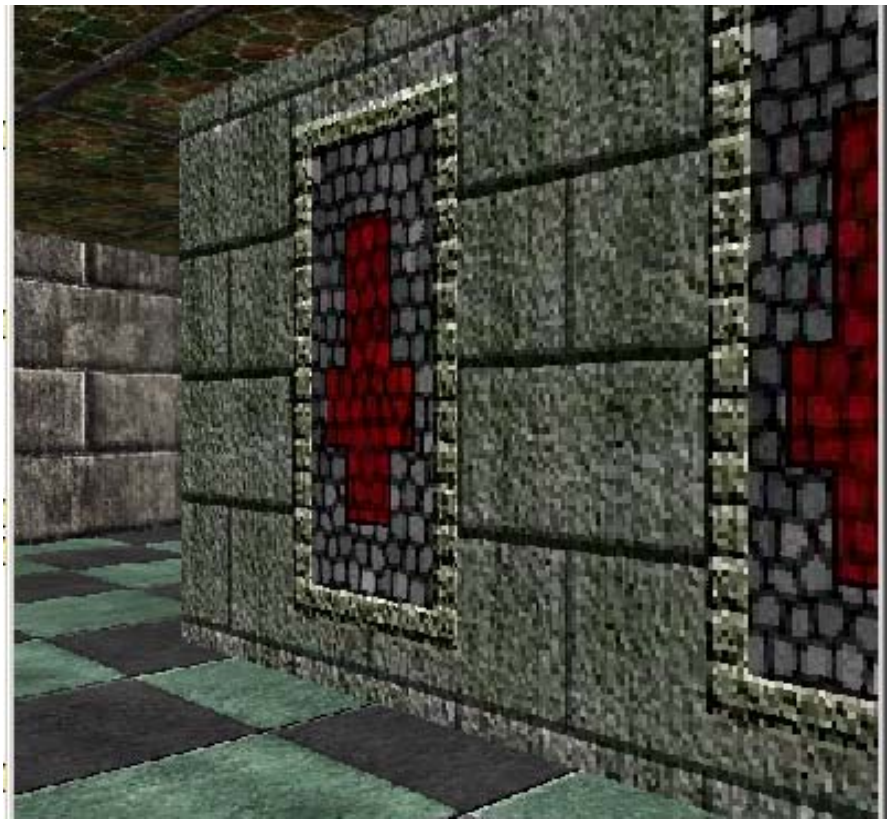
```
                                                    //Calculamos el punto más cercano
                                                    puntoCercano.x = (listaPlanosVision[i].normal.x > 0.0f) ? limiteMinimo.x : limiteMaximo.x;
                                                    puntoCercano.y = (listaPlanosVision[i].normal.y > 0.0f) ? limiteMinimo.y : limiteMaximo.y;
                                                    puntoCercano.z = (listaPlanosVision[i].normal.z > 0.0f) ? limiteMinimo.z : limiteMaximo.z;

      //Comprobamos si el punto más cercano está fuera del cono
```

```
if (D3DXVec3Dot(&listaPlanosVision[i].normal,&puntoCercano)
+listaPlanosVision[i].distancia > 0.0f) return false;
```

Este código se repite seis veces, una para cada plano del cono.

Lo único que queda ahora es añadir una instrucción que compruebe, para cada hoja que va a ser dibujada, si se encuentra no dentro del cono de visión.

El resultado de la aplicación del rechazo por cono de visión en conjunción con el mapa selector de texturas se puede apreciar en la siguiente figura:



Algoritmo de "Conjuntos Posiblemente Visibles" (PVS)

En la anterior versión del proyecto ya estaba completamente implementado el uso de árboles BSP, que nos había dado varias ventajas como ordenación de polígonos sin usar Z-Buffer, control de colisiones... Sin embargo seguimos teniendo un problema, y es que si aumentábamos las dimensiones de nuestro escenario, el rendimiento se vería muy afectado. Esto es debido a que los el recorrido de los árboles BSP dibujaba todos aquellos polígonos cuya cara frontal se encontrara mirando hacia la posición de la cámara. Por ello, para esta nueva versión del proyecto, se ha sustituido la anterior estructura BSP, por un nuevo tipo de árbol llamado árboles BSP de hojas sólidas (Solid Leaf Tree), que permite la implementación del algoritmo de conjuntos posiblemente visibles PVS ("Posibly Visible Set").

La ventaja de los PVS es que guardan información para cada hoja del nuevo árbol de todas las hojas visibles desde esa posición, de forma, que a la hora de dibujar el escenario, el número de polígonos manejados se reduce drásticamente. Sin embargo antes de explicar esto en más detalle vamos a introducir brevemente en que consisten los árboles BSP sólido de hojas

Los árboles BSP, muy usados en juegos como Quake o Unreal, que usábamos antes eran bastante ineficientes porque la escasa información que almacenaban, su estructura y su forma de construirlos, obligaba a que fueran recorridos siempre que el escenario era dibujado. En este tipo de árboles, una vez que un polígono era usado como divisor, no volvía a ser utilizado, de forma que todos los polígonos eran independientes entre sí.

La gran diferencia en este nuevo tipo de árboles, radica en que, cuando un polígono es usado como divisor, se añade a la lista de polígonos situados delante del plano divisor y se marca como usado. De este modo, una vez que todos los polígonos de una rama han sido usados como divisor, se construye una hoja, que no es más que un conjunto convexo de polígonos.

El siguiente gráfico ayudará a comprender mejor esta diferencia:

Fig A) Árbol de nodos sólidos BSP

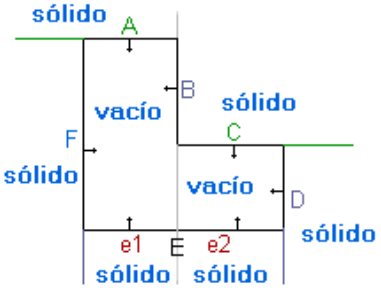
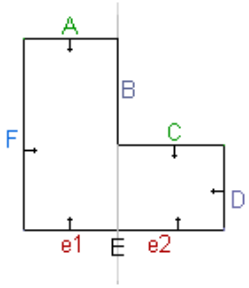
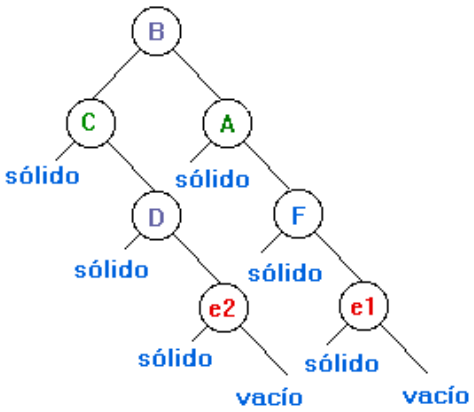
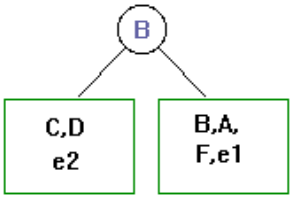


Fig B) Árbol de Hojas







Las hojas son listas de polígonos guardados juntos y que son convexos

Ahora, como se puede ver en el gráfico las hojas van a ser listas de polígonos, y éstos podrán ser dibujados en cualquier orden siempre que se aplique el 'back culling' y el Z-Buffer esté activo.

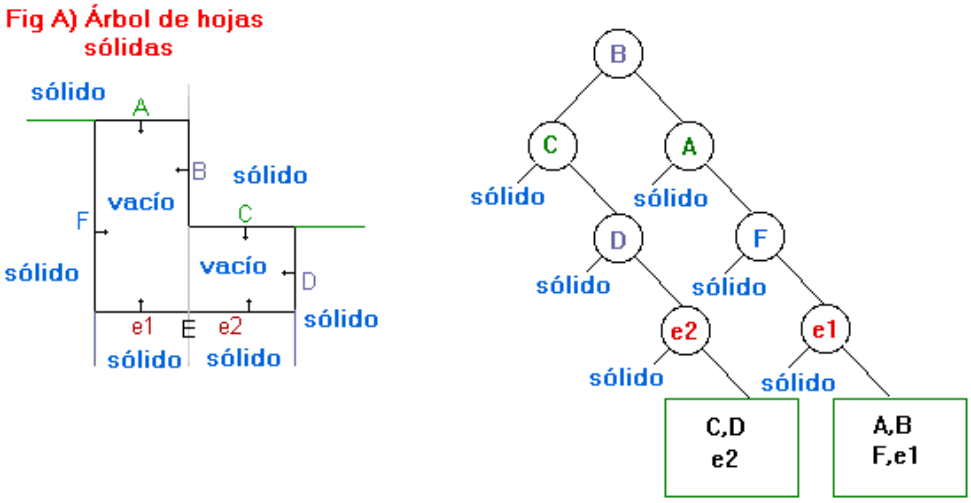
Esto puede parecer una desventaja con respecto a la anterior versión, pero en realidad es muy útil desde el momento en que esta nueva información almacenada en las hojas, que son regiones del espacio próximas entre sí, nos permite construir con relativa facilidad los PVS mencionados con anterioridad.

De todas formas, no vamos a centrarnos mucho en esta clase de árboles ya que tienen un gran problema, y es que no tiene información acerca de espacios sólidos y vacíos necesaria para el control de colisiones.

La solución a este problema son los árboles de hojas sólidas ('Solid Leaf BSP Tree'), que no es más que una mezcla entre los árboles que hemos comentado hasta ahora.

El funcionamiento de un árbol de hojas sólidas es muy parecido al del árbol sólido, solo que cuando una rama no tiene hijo trasero se le añade una que indica que el espacio situado detrás de la hoja padre es un espacio sólido por el cual no se permite caminar.

Para verlo mejor vamos a ver el siguiente dibujo:



Las hojas contienen conjuntos convexos de polígonos y también forman un espacio vacío. Las hojas siempre están en el nodo frontal y el espacio sólido en el nodo de detrás.

Sin embargo, por lo que vemos en el dibujo este árbol es tan grande como el que usábamos en la versión anterior, sin embargo ahora vamos a tener una gran ventaja, y es que este árbol no hay que recorrerlo entero, sino que una correcta representación de esta estructura de árbol mediante arrays de nodos, de hojas y de polígonos, permitirá acceder simplemente a las posiciones precisadas de la estructura del árbol. No obstante, la gran ventaja de este tipo de árboles es la misma que hemos mencionado antes: la facilidad para construir los datos de los PVS.

Los datos PVS no son ni más ni menos que la representación, en binario, del conjunto de hojas del árbol visible desde una hoja determinada y son propios de cada hoja y allí son almacenados. La idea es que los datos PVS son un conjunto de bits, en principio uno para cada hoja del árbol, de forma que cuando el bit i-ésimo vale 1 indica que la hoja i-ésima es visible, por ejemplo, si el primer byte de los datos PVS para una hoja cualquiera es 07, en realidad lo que esta diciendo es que las hojas 0, 1 y 2 son visibles mientras que el resto no, porque la representación binaria de 07 es 0000 0111 con los bits 0, 1 y 2 activos. De esta forma, cuando el árbol vaya a ser dibujado, bastará con recorrerlo una sola vez hasta llegar a la hoja en la que se encuentra la cámara, y desde ahí y gracias a los datos PVS, dibujar solo las hojas que sean visibles. No debemos olvidar que la representación de la estructura de árbol se realiza mediante arrays, de forma que bastará hacer un acceso a la posición de la hoja

en cuestión para obtener todos sus datos, incluida la lista de polígonos a dibujar.

Sin embargo podemos llegar a tener un problema de espacio si queremos mantener en cada hoja del árbol tantos bits como hojas. Es por esto por lo que vamos a usar un modo de almacenamiento que reduce el tamaño de las hojas, de forma que, si bien al construir inicialmente los datos PVS sí se reserva un bit para cada hoja, una vez construidos todos los datos PVS, son comprimidos con la técnica Zero Length Run.

La idea de esta técnica es la siguiente:

- Si nos encontramos un byte distinto de 0 lo trasladamos tal cual.
- Si es 0, ponemos un 0, y a continuación el número de 0's que vayan seguidos contando el anterior.

Un ejemplo:

	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	.....
Valor	15	0	16	0	8	0	0	0	0
Hojas	0-7	8-15	16-23	24-31	32-39	40-47	48-55	56-63	.....

Suponiendo que el resto del array son 0's, la información que vamos a guardar es la siguiente

15, 0, 1, 16, 0, 1, 8, 0, 121

Esto tiene muchas ventajas ya que normalmente desde una hoja, la cantidad de hojas vecinas que sean visibles va a ser muy pequeña, de forma que ahorramos mucha memoria escribiendo un 0 y a continuación el número de 0's que siguen. Otra gran ventaja es que todo esto se va a guardar en un mismo array, y las hojas sólo van a tener que almacenar un puntero que indica la posición inicial de sus datos PVS en el array maestro.

Aunque esto pueda parecer que es mucho tiempo de cálculo, sólo lo vamos a tener que realizar una sola vez, ya que se va a calcular antes de la ejecución del juego.

En las siguientes secciones, van a ser explicados con más detenimiento dos aspectos importantes de esta nueva versión como son la representación de nuestro árbol mediante arrays y los portales, que son unos planos invisibles

situados entre las hojas y que son necesarios para calcular correctamente los datos PVS.

Representación del árbol de hojas sólidas

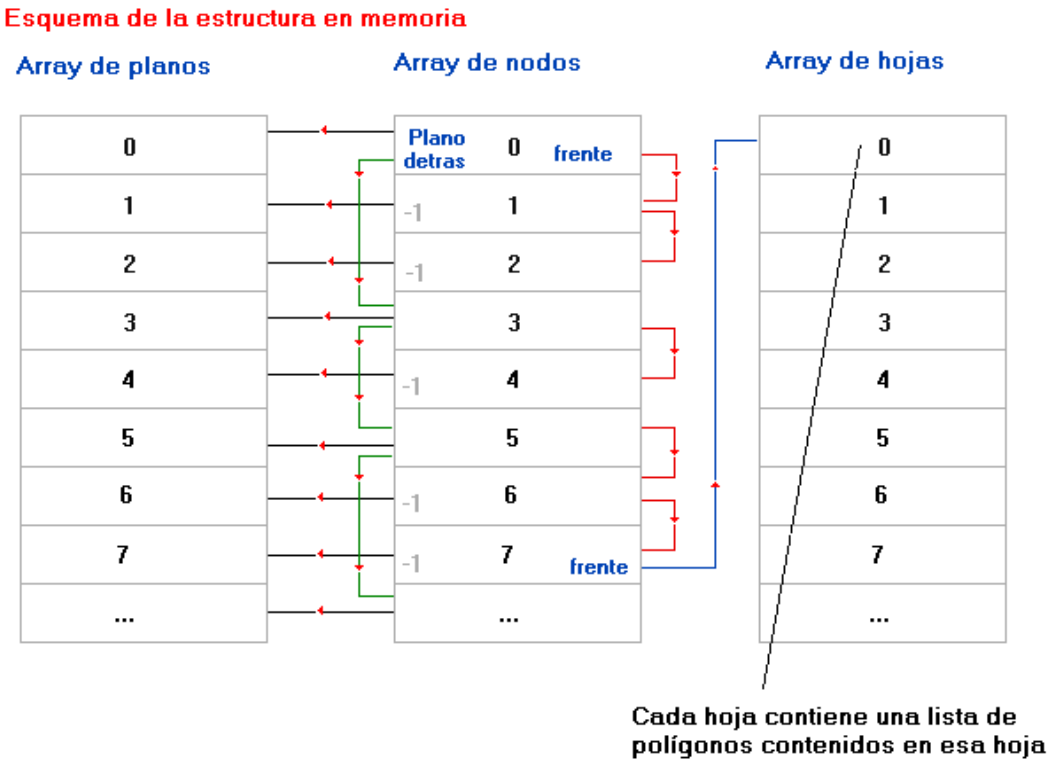
Para representar el árbol, lo primero que tenemos que solucionar es que ahora ya no podemos implementar el árbol con enlaces, ya que PVS requiere que podamos acceder a varias hojas desde otra hoja, y las alternativas son, o tener enlaces a todas las hojas, o recorrer el árbol hasta llegar a cada una de las hojas. Además para en un futuro poder guardar la información compilada del árbol para luego poder cargarla y no tener que calcularla lo que vamos a hacer es guardar el árbol en un array, y así también podremos acceder a cada hoja con sólo acceder a una componente del array.

Vamos a ver cómo va a ser la estructura de cada nodo:

```
struct NODE {
      bool esHoja;
      unsigned long plano;
      unsigned long frente;
      signed long detras;
                                                                  };
```

Así, tenemos que para apuntar al nodo de delante de detrás, sólo tenemos que almacenar la posición en el array, y si detrás marca –1 significa que es sólido. Si esHoja es cierto, tenemos que es una hoja y entonces el índice corresponde al array de hojas, y si es falso no, y por lo tanto frente índice será correspondiente al array de nodos. Por otro lado el plano no es más que un índice en un array de planos y que va a indicar la posición y la normal.

Para comprenderlo mejor el siguiente dibujo:



Mientras tanto, la estructura de las hojas es la siguiente:

```
struct HOJA {
    long poligonoComienzo;
    long poligonoFinal;
    long listaIndicesPortales[50];
    long numeroPortales;
    long indicePVS;
};
```

Hay 2 variables pertenecientes a esta estructura que no voy a explicar ahora, ya que lo trataremos luego, y son las referentes a los portales.

La variable indicePVS va a indicar donde comienza la información PVS para esa hoja, como ya comentamos anteriormente.

Las variables poligonoComienzo y poligonoFinal indican la lista de polígonos correspondientes a la hoja, ya que los polígonos al ser tratados por

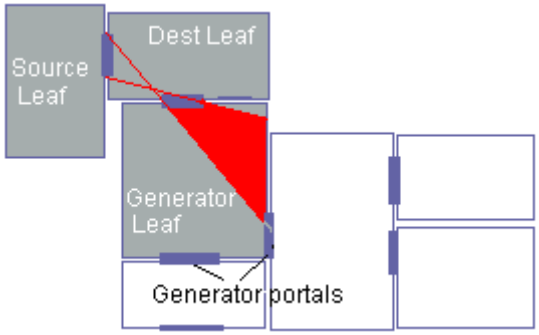
este nuevo compilador BSP van a quedar ordenados de forma que esto sea posible.

Portales

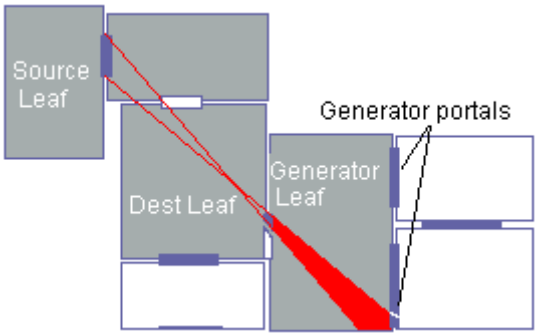
Como ya hemos dicho anteriormente, los portales son una herramienta necesaria para poder calcular correctamente los datos PVS. Los portales no son ni más ni menos que unos planos invisibles situados entre dos hojas adyacentes y que guardan información sobre las dos hojas entre las que se encuentran.

Para que todo quede más claro, vamos a explicar el proceso de la construcción de los datos PVS con ayuda de unos cuantos diagramas.

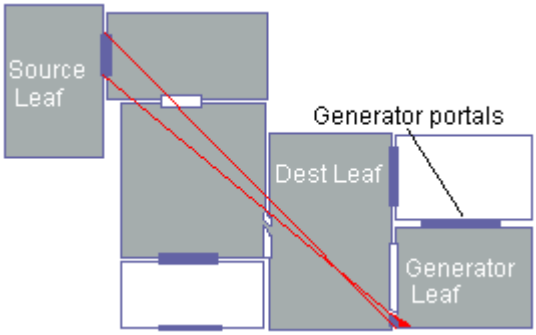
Cuando los datos PVS van a ser calculados, se toman todos los portales que se encuentran en la misma hoja de la cámara, que llamaremos hoja fuente, y se añaden al conjunto de hojas posiblemente visibles todas aquellas hojas situadas al otro lado de cada uno de esos portales. A esas nuevas hojas las llamaremos hojas destino. En este momento, hallaremos, entre todos portales de la hoja destino, aquellos que sean visibles desde la hoja fuente, y entre estos últimos calcularemos la otra hoja adyacente, a la que llamaremos hoja generadora. Estas hojas generadoras también se añaden a los datos PVS.



Desde este momento repetimos el proceso anterior considerando como nuevas hojas generadoras a aquellas adyacentes a cada uno de los portales generadores, aunque con una excepción ya que solo consideraremos la parte del portal generador visible desde el portal fuente.



El proceso termina cuando todas las hojas generadoras han sido creadas y estudiadas.



Para calcular los datos PVS de todas las hojas de nuestro árbol bastará repetir este proceso considerando como hoja fuente a cada una de las hojas del árbol.

Estructuras de datos utilizadas

La clase ClaseEscenario posee los siguientes atributos privados:

Atributos para guardar la cantidad máxima de nodos, polígonos, planos, hojas y portales que contienen las respectivas listas:

```
long maxNumNodos

long maxNumPoligonos

long maxNumPlanos
```

```
                                                                long
maxNumHojas

                                                                long
maxNumPortales
```

Listas que almacenan los nodos, polígonos, hojas, planos, datos de los conjuntos potencialmente visibles (PVS), portales y polígonos con textura que forman el árbol BSP:

```
                                                                POLIGON
O *listaPoligonos

                                                                NODO
*listaNodos

                                                                HOJA
*listaHojas

                                                                PLANO
*listaPlanos

                                                                BYTE
*listaDatosPVS

                                                                PORTAL
**listaPortales

                                                                POLIGON
O *listaPoligonosConTextura[25]
```

Array global para almacenar las 25 texturas que se utilizarán en el escenario:

```
                                                                LPDIREC
T3DTEXTURE8 arrayTexturas[25]
```

Atributos que contienen los índices actuales para las diferentes listas:

```
                                                                long
indPoligono

                                                                long
indNodo

                                                                long
indHoja

                                                                long
indPlano

                                                                long
indPortal
```

Almacena el numero de bytes que cada conjunto potencialmente visible (PVS) necesita:

```
long bytesPorConjunto
```

Implementación de PVS y portales

ClaseEscenario()
Descripción:

Constructor de la clase escenario. Se establece 100 como tamaño inicial máximo de nodos, polígonos, planos, hojas y portales del árbol BSP. Posteriormente se redimensionan las listas hasta el tamaño necesario para acoger los datos que se van insertando. Comenzamos realizando una reserva de memoria inicial de 100 elementos para cada una de las listas implicadas. Se limpia la memoria y se inicializan a 0 los índices actuales de cada lista.

Parámetros de entrada:

- Ninguno

Parámetros de salida:

- Retorno de la función: Ninguno

```
void inicializarPoligonos(LPDIRECT3DDEVICE8 lpDevice)
```

Descripción:

Crea la lista de polígonos iniciales a partir del fichero que contiene el escenario correspondiente al nivel del juego. Se leen las texturas que forman parte del nivel con cargarTexturas, y crea el árbol BSP correspondiente con una llamada a construirArbolBSP. Seguidamente se construye la lista de portales (las puertas del escenario) con una llamada a construirPortales y se calculan los datos de los conjuntos potencialmente visibles para cada hoja con la llamada a calcularPVS.

Parámetros de entrada:

- LpDevice: El dispositivo Directx del juego al que se asociarán las texturas.

Parámetros de salida:

- Retorno de la función: Ninguno

```
POLIGONO *cargarNivel(char *nombreArchivo)
```

Descripción:

Crea la lista de polígonos que representa cada escenario del juego. Para ello lee del fichero de descripción del escenario los datos correspondientes a los polígonos, llamando a añadirPolígono (anyadirPolígono) para insertar la posición espacial de los vértices leídos, su color, la componente especular (que determinará el brillo especular del polígono cuando una luz directa – no la luz ambiente - se proyecte sobre él) y los componentes de las texturas.

Parámetros de entrada:
- NombreArchivo: el nombre del archivo que contiene el escenario del nivel del juego.

Parámetros de salida:
- Retorno de la función: La lista de polígonos generada.

```
POLIGONO *anyadirPoligono
                                                              (POLIGO
NO *poligonoPadre,
                              CUSTOMVERTEX *listaVertices,
                              int indiceTextura,
                              int numVertices)
```

Descripción:

Construye un polígono nuevo a partir de una lista de vértices y su número añadiéndolo en la posición correcta de la lista de polígonos. Se crea la lista de índices tal como se hacía en la antigua implementación del árbol BSP y se rellenan el número de vértices y el número de índices con la fórmula (nvértices-2)*3. A partir de tres puntos del plano se calculan dos vectores que, a su vez, sirven para determinar la normal al polígono. Se guarda el índice de la textura que tendrá el polígono.

Parámetros de entrada:
- PoligonoPadre: El polígono padre del polígono a crear.
- ListaVértices: Lista enlazada de vértices del polígono a construir.
- ÍndiceTextura: Número de la textura del polígono.
- NumVértices: El número de vértices del polígono.

Parámetros de salida:
- Retorno de la función: El polígono construido.

```
void cargarTexturas(LPDIRECT3DDEVICE8 lpDevice)
```

Descripción:

Carga las 25 texturas que se van a utilizar en el escenario mediante la primitiva D3DXCreateTextureFromFileA proporcionada por DirectX.

Parámetros de entrada:
- LpDevice: El dispositivo Directx creado por el juego al que se asociarán las texturas.

Parámetros de salida:
- Retorno de la función: Ninguno

```
void construirArbolBSP(long nodo, POLIGONO *listaPoligonosInicial)
```

Descripción:

Construye recursivamente el árbol BSP a partir de una lista enlazada de polígonos y un numero de nodo. Inicialmente la llamada se hace con la lista de polígonos que hemos obtenido con cargarNivel y el número de nodo 0.

Parámetros de entrada:
- Nodo: El número de nodo a partir del que se crea el árbol.
- ListaPoligonosInicial: La lista de polígonos a incluir en el árbol.

Parámetros de salida:
- Retorno de la función: Ninguno

```
long seleccionarMejorDivisor(POLIGONO *listaPoligonosInicial)
```

Descripción:

Selecciona el mejor plano divisor de una lista de polígonos de acuerdo a una heurística en la que se penaliza que se realicen muchos cortes en cada polígono evaluado y favorece que el árbol sea lo más equilibrado posible (igual número de polígonos delante que detrás).

Parámetros de entrada:
- ListaPoligonosInicial: La lista enlazada de polígonos del escenario.

Parámetros de salida:
- Retorno de la función: El índice indplano que determina el plano que ha resultado como mejor divisor posible.

```
void calcularCajaMinimal     (CAJAMINIMAL *cajaMinimal,
                              POLIGONO *listaPoligonosInicial)
```

Descripción:

Calcula la caja minimal ("bounding box") que contiene completamente a un conjunto dado de polígonos. Las dimensiones de esta caja minimal serán las necesarias para abarcar todos los polígonos de una hoja.

Parámetros de entrada:
- ListaPoligonosInicial: Lista de polígonos que deberá abarcar.

Parámetros de salida:
- CajaMinimal: Estructura con las dimensiones de la caja cuadrangular que encierra a los polígonos.
- Retorno de la función: Ninguno

```
void calcularLímitesPoligono (POLIGONO *poligono,
                              D3DXVECTOR3 *maximo, D3DXVECTOR3 *minimo)
```

Descripción:

Calcula los límites que contienen completamente a un polígono dado. Para ello se inicializan las componentes de la solución a valores imposibles (extremadamente grandes) y recorremos todos los vértices del polígono guardando los valores máximos y mínimos que encontramos.

Parámetros de entrada:
- Polígono: El polígono del que se pretenden calcular sus límites.

Parámetros de salida:
- Máximo: Vector 3D de directX con las componentes máximas para los ejes x,y,z.
- Mínimo: Vector 3D de directX con las componentes mínimas para los ejes x,y,z.
- Retorno de la función: Ninguno

```
void borrarPoligono(POLIGONO *poligono)
```

Descripción:

Borra un polígono y todos sus atributos asociados.

Parámetros de entrada:

- Polígono: El polígono a borrar.

Parámetros de salida:

- Retorno de la función: Ninguno

`void incrementarIndNodo()`

Descripción:

Incrementa el contador del array de nodos (atributo indNodo) y, en caso de que supere el máximo número de nodos, aumenta el tamaño del array volviendo a solicitar otros 100 nodos más (o el valor establecido en maxNumNodos) para listaNodos mediante `realloc`.

Parámetros de entrada:

- Ninguno

Parámetros de salida:

- Retorno de la función: Ninguno

`void incrementarIndPoligono()`

Descripción:

Incrementa el contador del array de polígonos y, en caso de que supere el número máximo, aumenta el tamaño del array volviendo a solicitar otros 100 polígonos más.

Parámetros de entrada:

- Ninguno

Parámetros de salida:

- Retorno de la función: Ninguno

`void incrementarIndPlano()`

Descripción:

Incrementa el contador del array de planos y, en caso de que supere el número máximo, aumenta el tamaño del array volviendo a solicitar otros 100 planos más.

Parámetros de entrada:

- Ninguno

Parámetros de salida:

- Retorno de la función: Ninguno

`void incrementarIndHoja()`

Descripción:

Incrementa el contador del array de hojas y, en caso de que supere el número máximo, aumenta el tamaño.

Parámetros de entrada:

- Ninguno

Parámetros de salida:

- Retorno de la función: Ninguno

```
void incrementarIndPortal()
```

Descripción:

Incrementa el contador del array de portales y, en caso de que supere el número máximo, aumenta el tamaño.

Parámetros de entrada:

- Ninguno

Parámetros de salida:

- Retorno de la función: Ninguno

```
int clasificarPoligono(POLIGONO *poligono, PLANO *plano)
```

Descripción:

Clasificamos un polígono respecto a otro (o al plano que forma, si se prefiere). Eligiendo un vértice cualquiera del polígono, calculamos el vector dirección de la recta que une el vértice anterior y el punto a clasificar. Si hallamos el producto vectorial de la dirección calculada y la normal del polígono, el ángulo resultante determina si está delante o detrás. Si es mayor de 90º el punto está detrás del polígono, si es menor que 90º el punto está enfrente. Si ningún caso se cumple, es decir, el resultado es cero o extremadamente cercano a 0 (hemos de pensar que tratamos con tipos float de C++), entonces el punto está en el plano. Esta operación se realiza para cada vértice del polígono, de forma que podemos tener 4 casos. Que el número de vértices delante sea igual al número de vértices del polígono, lo que significa que el polígono está delante. Igual para el caso de que esté completamente detrás. Igual es el tratamiento i el número de vértices coincidentes es igual al número de vértices del polígono (dicho de otra forma, todos los vértice son coincidentes), en este caso los polígonos son coincidentes. Finalmente, si no se cumple ninguno de los casos anteriores sabemos que los polígonos se cortan (o intersectan).

Parámetros de entrada:

- Polígono: El polígono a clasificar.
- Plano: El otro polígono (o plano, como se prefiera) contra el que se va a clasificar.

Parámetros de salida:

- Retorno de la función: La clasificación del polígono definida por los valores (enteros): CP_COINCIDENTES 0, CP_DELANTE 1, CP_DETRAS 2 o CP_CORTAN 4.

```
void dividirPoligono   (POLIGONO *poligono, PLANO *plano,
                        POLIGONO *divisionDelante,
                        POLIGONO *divisionDetras)
```

Descripción:

Realiza la partición del polígono que es cortado por un plano (otro polígono), dividiéndolo en dos (divisiónDelante y divisiónDetrás). Esto impide que un polígono esté a la vez delante y detrás de otro que lo

corta. Se determina si cada uno de los vértices del polígono debe pertenecer a los vértices que quedan delante o a los de detrás utilizando la función clasificarPunto. Cada una de las aristas del polígono se trata como una recta separada. Para realizar el corte se ayuda de la función interseccionRectaPlano, con la que se determina la intersección entre la recta correspondiente a cada una de las aristas del polígono con el plano de corte. De esta forma hemos determinado que vértices deben formar parte del nuevo polígono de delante y los que deben formar el nuevo polígono de detrás.

Parámetros de entrada:

- Polígono: El polígono a cortar.
- Plano: El plano de corte (el polígono contra el que se quiere cortar).

Parámetros de salida:

- ParteDelantera: Nuevo polígono creado con la mitad del polígono que queda delante.
- ParteTrasera: Igual para la otra mitad.
- Retorno de la función: Ninguno.

```
bool interseccionRectaPlano  (D3DXVECTOR3 *inicioRecta,
                              D3DXVECTOR3 *finRecta,
                              D3DXVECTOR3 *puntoPlano,
                              D3DXVECTOR3 *normalPlano,
                              D3DXVECTOR3 *interseccion,
                              float *porcentaje)
```

Descripción:

Calcula el punto de intersección, si lo hay, entre un plano, representado por un punto y el vector normal y una recta, representada por dos puntos. Para ello se calcula el tamaño del vector que va de inicioRecta a finRecta, si este vector está en el mismo lado del plano entonces no hay corte. Si no es así, calculamos un vector que una cualquier punto del plano con el punto inicial de la recta, y hallamos su tamaño, si la distancia al plano es menor que la longitud de la recta entonces se cortan. Dividiendo la distancia al plano por la longitud de la recta obtenemos el índice que variando de 0 a 1 nos dice dónde se cortan la recta y el plano, por ejemplo la distancia al plano es 1 y la longitud de la recta es 2, se cortarán en el 0.5 (a la mitad de la recta). Podemos calcular el punto de corte escalando el vector dirección de la recta al porcentaje calculado y sumándolo al punto inicial de la recta.

Parámetros de entrada:

- InicioRecta: El punto de inicio de la recta.
- FinRecta: El punto de final de la recta.
- PuntoPlano: Un punto del plano.
- NormalPlano: La normal del plano.

Parámetros de salida:

- Intersección: El punto de intersección plano-recta.
- Porcentaje: Varía de 0 (inicio de la recta) a 1 (fin de la recta) y determina el punto de la recta en el que ha ocurrido la intersección. Resulta útil

cuando se está realizando el corte de un polígono, (recordemos que el corte de polígono hace varias llamadas a esta función), para determinar cómo distribuir una textura de forma que quede exactamente igual que si el polígono nunca hubiera sido separado en dos. Si no lo tuviéramos en cuenta la textura aparecería "rota" al cortar el polígono, justo en el comienzo del segundo polígono obtenido tras el corte.

- Retorno de la función: Verdadero si hay intersección, Falso en caso contrario.

```
int clasificarPunto (D3DXVECTOR3 *punto, PLANO *plano)
```

Descripción:

Esta operación es idéntica a la descrita en clasificar polígono, sólo que aplicada a un solo punto. Eligiendo un vértice cualquiera del polígono que actuará como plano de corte, calculamos el vector dirección de la recta que une el vértice anterior y el punto a clasificar. Si hallamos el producto vectorial de la dirección calculada y la normal del polígono, el ángulo resultante determina si está delante o detrás. Si es mayor de 90º el punto está detrás del polígono, si es menor que 90º el punto está enfrente. Si ningún caso se cumple, es decir, el resultado es cero o extremadamente cercano a 0 (hemos de pensar que tratamos con tipos float de C++), entonces el punto está en el plano.

Parámetros de entrada:

- Punto: El punto a clasificar.
- Plano: El polígono (que si lo extendemos hasta el infinito podríamos ver como un plano).

Parámetros de salida:

- Retorno de la función: La clasificación del punto definida por los valores (enteros): CP_COINCIDENTES 0, CP_DELANTE 1, CP_DETRAS 2.

```
void recorrerArbol(D3DXVECTOR3 camara, LPDIRECT3DDEVICE8 lpDevice)
```

Descripción:

Recorre el árbol de nodos hasta llegar a la hoja en la que se encuentra la cámara. Desde ahí sólo dibuja los polígonos visibles desde esa hoja. Se pueden dar 3 casos diferenciados dependiendo de si la posición de la cámara está delante, detrás o es coincidente con el nodo. Si la cámara está delante del nodo actual y es un nodo hoja se llama a dibujarHoja sobre el dispositivo lpDevice, si aún no es hoja avanzamos por la rama delantera del nodo en el que estamos. Otra posibilidad es que la cámara esté detrás del nodo y sea un nodo hoja, en cuyo caso la función no dibuja y acaba puesto que el movimiento es ilegal, en caso contrario se avanza por la rama trasera del nodo actual. Por último si son coincidentes, la cámara está sobre el nodo y se trata igual que si estuviera delante (dibujando si es hoja, o avanzando por la lista de delante si aún no es una hoja).

Parámetros de entrada:

- Cámara: La posición de la cámara (nuestra posición en el juego).

- LpDevice: El dispositivo directX sobre el que se dibujará.

Parámetros de salida:

- Retorno de la función: Ninguno

```
void dibujarHoja(long hoja, LPDIRECT3DDEVICE8 lpDevice)
```

Descripción:

Recorre todos los polígonos de una hoja para dibujarlos comprobando la visibilidad para todas las hojas respecto de la hoja original, de forma que sólo vamos dibujando las que resultan visibles. Para ello nos desplazamos por la lista de datos PVS hasta los correspondientes a la hoja original, analizando seguidamente bit a bit dentro del byte PVS si la hoja i-ésima es visible o no. Si la hoja es potencialmente visible guardamos cada polígono de esa hoja en la lista que le corresponde de acuerdo a su textura (de las 25 listas de polígonos con textura disponibles, una por cada textura que aparece en el escenario). Finalmente se dibuja la lista de triángulos `(D3DPT_TRIANGLELIST)` de cada polígono con la textura que corresponde mediante la primitiva directX `DrawIndexedPrimitiveUP`.

Parámetros de entrada:

- Hoja: La hoja original a dibujar.
- LpDevice: El dispositivo directX sobre el que se dibujará.

Parámetros de salida:

- Retorno de la función: Ninguno

```
bool lineaDeVision
                                                    (D3DXVE
CTOR3 *ptoInicio,
                        D3DXVECTOR3 *ptoFin, long nodo)
```

Descripción:

Calcula recursivamente, recorriendo el árbol de nodos, si hay una línea de visión entre dos puntos dados respecto al plano del nodo que se suministra. Para ello realiza una clasificación de cada punto con respecto al plano llamando a la función clasificarPunto.

Parámetros de entrada:

- PtoInicio: Vector 3D con el punto inicial.
- PtoFin: Vector 3D con el punto final.
- Nodo: El plano del nodo actual.

Parámetros de salida:

- Retorno de la función: Verdadero si no hay obstáculos sólidos, falso en caso contrario.

```
PORTAL *calcularPortalInicial(long nodo)
```

Descripción:

Crea el portal inicial para un nodo determinado. Para ello se calcula el centro geométrico del portal, sus límites y la distancia al plano divisor de ese nodo. Seguidamente se obtendrá el tamaño correcto de los vectores

x e y que delimitan los extremos del portal, calculando la distancia desde el centro del portal a los lados de la caja minimal (bounding box) que se obtuvo para ese nodo. De esta forma tendremos ya todos los datos necesarios para determinar las posiciones de los 4 vértices que lo delimitan, información que insertaremos junto a los 6 índices que permiten delimitarlo rapidamente mediante dos triángulos y la normal obtenida de la normal del plano que actúa como divisor para ese nodo.

Parámetros de entrada:

- Nodo: El índice del nodo sobre el que se creará el portal.

Parámetros de salida:

- Retorno de la función: El portal creado.

```
PORTAL *recortarPortal(PORTAL *portal, long nodo)
```

Descripción:

Recorta un portal de tal forma que se ajuste a la región entre dos hojas. Recorre la lista de portales para determinar los portales que deben ser divididos con respecto al plano del nodo que se entrega como parámetro, para ello utiliza dividirPortal.

Parámetros de entrada:

- Portal: Lista de portales.
- Nodo: El índice del nodo respecto del que se determinan los recortes.

Parámetros de salida:

- Retorno de la función: Lista de portales con las divisiones ya realizadas.

```
void dividirPortal     (PORTAL *portal, PLANO *plano,
                        PORTAL *divisionDelantera,
                        PORTAL *divisionTrasera)
```

Descripción:

Corta un portal con un plano y devuelve las dos partes resultantes. Resuelve el problema que ya tratamos en dividirPolígono pero aplicado al caso de un portal. De esta forma obtendremos un nuevo portal con la división delantera y otro con la trasera, al igual que con dividirPolígono.

Parámetros de entrada:

- Portal: El portal a recortar.
- Plano: El plano de corte (el polígono contra el que se quiere cortar).

Parámetros de salida:

- DivisiónDelantera: Nuevo portal creado con la mitad del portal que queda delante.
- DivisiónTrasera: Igual para la otra mitad.

Retorno de la función: Ninguno

`INDICE *recorridoInOrden(long nodo)`

Descripción:

Hace un recorrido "in-orden" del árbol BSP para evitar la recursión en construirPortales. El funcionamiento es el habitual recorrido "in-orden" a partir del número de nodo dado como parámetro.

Parámetros de entrada:

- Nodo: El índice del nodo a partir del que se realiza el recorrido.

Parámetros de salida:

- Retorno de la función: La lista de índices a los nodos que indica el recorrido correcto "in-orden".

`void borrarPortal(PORTAL *portal)`

Descripción:

Borra un portal y todos sus atributos.

Parámetros de entrada:

Portal: El portal a borrar con `delete`.

Parámetros de salida:

- Retorno de la función: Ninguno

`void construirPortales()`

Descripción:

Construye los portales de nuestro escenario y los recorta para que sean del tamaño adecuado. Además comprueba si están repetidos y si son portales válidos. Para ello comienza obteniendo la lista de índices a los nodos de recorridoInOrden. Recorre toda la lista de modo que para cada nodo se calcula el portal inicial con calcularPortalInicial, se recorta con recortarPortal pasándolo por todos los nodos del árbol. Se comprueba si es un portal válido y si no está duplicado (con comprobarPortalDuplicado). Si es válido se añade a la lista principal de portales del escenario.

Parámetros de entrada:

- Ninguno

Parámetros de salida:

- Retorno de la función: Ninguno

`void eliminarPortalLista(PORTAL *portal)`

Descripción:

Elimina un portal de la lista enlazada de portales del escenario ocupándose de cruzar los punteros para que la lista siga enlazada y liberar la memoria llamando a borrarPortal.

Parámetros de entrada:

- Portal: Portal a eliminar.

Parámetros de salida:

- Retorno de la función: Ninguno

```
bool comprobarPortalDuplicado(PORTAL *portal, long *indicePortal)
```

Descripción:

Comprueba si hay un portal duplicado recorriendo la lista de portales del escenario y en caso de no encontrarse devuelve el índice donde debería ir colocado.

Parámetros de entrada:

- Portal: El portal que se comparará con los existentes en la lista de portales del objeto escenario.

Parámetros de salida:

- ÍndicePortal: En caso de que la función se evalúe a falso, índicePortal contendrá la posicion en la que debe colocarse el portal de entrada.
- Retorno de la función: Verdadero si ya existía el portal que se pretende insertar en la lista de portales. Falso si no existe, en cuyo caso se devuelve la posición adecuada en el índice.

```
long calcularPVS()
```

Descripción:

Recorre todas las hojas calculando los datos PVS para cada una de ellas y devuelve el tamaño actual de la lista de datos PVS. Comenzamos por fijar a 1 el bit correspondiente a la hoja actual puesto que cada hoja se "ve" a sí misma. A continuación, y para cada hoja, recorremos su lista de portales y para cada portal obtenemos la hoja situada al otro lado del portal y fijamos su bit a 1 indicando que es visible.

Parámetros de entrada:

- Ninguno

Parámetros de salida:

- Retorno de la función: Índice que marca la siguiente posición a escribir de la lista de datos PVS.

```
void fijarBitPVS(BYTE *hojaPVS, long hojaDestino)
```

Descripción:

Utilizada solamente para fijar el bit de visibilidad correcto para un búffer de bits PVS y una hoja determinada.

Parámetros de entrada:

- HojaDestino: Índice a la hoja destino.

Parámetros de salida:

- HojaPVS: Byte con el bit correspondiente a la hoja destino puesto a 1.
- Retorno de la función: Ninguno

```
PLANO calcularPlanoPortal (PORTAL *portal)
```

Descripción:

Dado un portal, genera el plano que lo contiene. El plano obtiene la normal del propio portal, y el primero de los vértices del portal será uno de los puntos del plano.

Parámetros de entrada:

- Portal: El portal sobre el que se pretende obtener el plano.

Parámetros de salida:

- Retorno de la función: El plano que contiene al portal.

```
void calcularDatosPVS                                          (long
hojaFuente, PORTAL *portalFuente,
                        long hojaDestino, PORTAL *portalDestino,
                        BYTE *hojaPVS)
```

Descripción:

Calcula recursivamente los datos de visibilidad (PVS) necesarios dados una hoja y portal fuente y hoja y portal destino. Mediante clasificarPunto clasificamos los centros geométricos de ámbos portales y desechamos los casos triviales, como que sean el mismo, si no es el caso se llama a recortarConAntiPenumbra para los portales fuente, destino y el generador obtenido a partir de la hoja generadora calculada en función de la hoja destino. Llamando a fijarBitPVS ponemos a 1 el bit correspondiente a la hoja generadora de cada una de las llamadas recursivas.

Parámetros de entrada:

- HojaFuente: La hoja de inicio del cálculo de los datos de visibilidad.
- PortalFuente: El portal de inicio.
- Hojadestino: La hoja destino del cálculo.
- PortalDestino: El portal correspondiente al destino.

Parámetros de salida:

- HojaPVS: El búffer de bits con los bits de visibilidad adecuados puestos a 1.
- Retorno de la función: Ninguno

```
PORTAL *recortarConAntiPenumbra                                (PORTAL
*portalFuente,
                                    PORTAL *portalDestino,
                                    PORTAL *portalGenerador)
```

Descripción:

Genera los planos divisores y examina el portal generador con todos esos planos recortándolo o incluso borrándolo si es necesario. Para ello creamos todos los planos de corte válidos que vayan del portal fuente al destino y viceversa generando las dos aristas necesarias para calcular la normal del plano de recorte. Una vez generados todos los planos de corte recortamos el portal generador con cada uno de esos planos

Parámetros de entrada:

- PortalFuente: El portal inicio del cálculo de los planos de corte.
- PortalDestino: El portal destino.
- PortalGenerador: El portal a recortar respecto de los fuente y destino.

Parámetros de salida:

- Retorno de la función: El portal generador recortado.

En la siguiente figura se observa la inclusión de portales en el escenario:

Técnica de mapeado MIP (mipmapping)

El término MIP proviene del latín Multi in Parvum (muchos en poco) debido a que en esta técnica de mapeado de texturas se usan múltiples versiones de cada mapa de texturas, cada una a diferente nivel de detalle. Cuando el objeto se acerca o se aleja del observador, se aplica la textura apropiada. Esto hace que los objetos tengan un alto grado de realismo y acelera el tiempo de proceso, permitiendo al programa dibujar de forma más simple (con mapas de textura menos detallados) cuando los objetos se alejan.

Además, la técnica del mipmapping reduce la distorsión que se produce en las texturas de los objetos 3D cuando se alejan mucho de la cámara, ya llega un momento en el número de pixels que forman la textura es mayor que la propia resolución de la tarjeta gráfica en el polígono que encierra la textura, lo que obliga a desechar algunos de ellos para poder dibujarla produciendo, a veces, distorsiones indeseables. De esta forma podemos controlar la apariencia de la textura a diferentes distancias.

Obviamente la utilización de mipmapping requiere un poco más de esfuerzo al confeccionar el gráfico de textura puesto que es preciso hacer mapas diferentes según la distancia. En el siguiente gráfico se muestra un ejemplo clásico de generación de mapas de diferentes resoluciones a partir de la misma imagen:

Sin embargo DirectX soporta de modo nativo la técnica de mipmapping por lo que se suministra la herramienta DirectX Texture para la generación automática de los “mip maps” a partir de un archivo gráfico (.bmp, .dds o .tga).

Adicionalmente es posible activar el mipmapping en el dibujo de todas las texturas del escenario con la función `SetTextureStageState` y el parámetro D3DTSS_MIPFILTER aplicado al dispositivo 3D creado en DirectX.

El resultado final del uso de esta técnica en el juego se puede observar en cualquier textura dibujada en el escenario del juego:

Archivo de registro

Hemos incorporado un archivo de registro en el que se irá volcando la información que el juego genera durante la inicialización (sistema operativo, tarjeta y drivers detectados, creación de los objetos más importantes del juego, lectura de los archivos externos, etc.). Obligamos a la escritura física en el fichero después de cada línea para saber exactamente en qué punto se ha detenido la ejecución del programa.

Progresivamente se irá incluyendo más información sobre la ejecución del nuevo código que se vaya implementando. Actualmente el aspecto del actual registro es el siguiente:

```
-----------------------------------------------------
----------------- Archivo de Registro ---------------
-----------------------------------------------------
                                                Jueves 17 de Enero de 2002   11:39:59

Objeto clase escenario creado

Sistema Operativo en ejecución: Microsoft Windows 2000 Professional   Service Pack 2
DirectX8 detectado.

Tarjeta gráfica detectada: 16MB ATI Rage 128 Ultra
Librería del controlador: ati2dvaa.dll
Versión del controlador: 5.13.1.3217

Resolución del modo gráfico: 392 x 373
Memoria libre disponible en la tarjeta: 41 Mb

Escenario cargado
Arbol BSP construido
Portales construidos
Datos PVS calculados
Inicialización de Direct3D completada.
-----------------------------------
                                                11:42:46
```

Implementación del archivo de registro

`Registro::Registro()`

Descripción:

Constructor del archivo de registro “registro.txt”, inicializa el fichero y escribe la fecha (en español independientemente de la localización de la versión del sistema operativo utilizado) y la hora de comienzo de la ejecución.

Parámetros de entrada:

- Ninguno.

Parámetros de salida:

- Retorno de la función: Ninguno.

`Registro::escribeLinea(void *linea)`

Descripción:

Escribe el parámetro entregado (un número, por ejemplo) en el archivo de registro, realizando la conversión apropiada, como una línea de texto.

Parámetros de entrada:

- Línea: Puntero a lo que deseamos escribir.

Parámetros de salida:

- Retorno de la función: Ninguno.

`Registro::escribeLinea(char *linea)`

Descripción:

Escribe la cadena de carácteres en el archivo.

Parámetros de entrada:

- Línea: Puntero a la línea de texto.

Parámetros de salida:

- Retorno de la función: Ninguno.

`Registro::cerrar()`

Descripción:

Anota la hora del final de la ejecución y cierra el fichero.

Parámetros de entrada:

- Línea: Ninguno

Parámetros de salida:

- Retorno de la función: Ninguno.

Funciones auxiliares para la obtención de información del sistema:

`char *getTarjeta(LPDIRECT3D8 d3d8)`

Descripción:

Construye una línea de texto con la información de la tarjeta gráfica detectada (marca y modelo) y los drivers en uso.

Parámetros de entrada:

- D3d8: El dispositivo DirectX asociado a la tarjeta.

Parámetros de salida:

- Retorno de la función: Cadena de caracteres conteniendo la información obtenida del sistema.

```
char *getControlador(LPDIRECT3D8 d3d8)
```

Descripción:

Construye una línea de texto con la versión de los drivers detectados.

Parámetros de entrada:

- D3d8: El dispositivo DirectX asociado a la tarjeta.

Parámetros de salida:

- Retorno de la función: Cadena de caracteres con los cuatro dígitos que determinan completamente la versión del driver gráfico.

```
char *getResolucion(HWND hWnd)
```

Descripción:

Construye una cadena con la resolución gráfica de la ventana (sea en modo ventana o a pantalla completa) en la que se está ejecutando el juego.

Parámetros de entrada:

- hWnd: La ventana en la que se dibuja.

Parámetros de salida:

- Retorno de la función: Cadena de caracteres con la resolución en ancho x alto detectada.

```
char *getMemoria(LPDIRECT3DDEVICE8 d3d8)
```

Descripción:

Construye una cadena con la información relativa a la cantidad de memoria disponible en la tarjeta para texturas. Habitualmente será la memoria física libre disponible en la propia tarjeta, pero en algunos aparatos en los que se habilitado en la configuración BIOS el uso de una porción de la memoria central como memoria gráfica será la suma de ambas.

Parámetros de entrada:

- D3d8: El dispositivo DirectX asociado a la tarjeta.

Parámetros de salida:

- Retorno de la función: Cadena de caracteres con la memoria disponible en MegaBytes.

```
char *getPlataforma()
```

Descripción:

Detecta el sistema operativo Windows en uso, y en el caso de Windows 2000, la versión y los service packs aplicados.

Parámetros de entrada:

- Ninguno.

Parámetros de salida:

- Retorno de la función: Cadena de caracteres con el sistema operativo detectado (Windows 95, 98, 2000, etc.).

PARTE II

OBJETOS

Modelos de Humanoides

Al abordar la introducción de figuras de humanoides en el escenario encontramos dos posibilidades a la hora de importar el diseño gráfico de la figura:

- Obtener el diseño de 3Dstudio en formato .3ds
- Obtener el diseño de los modelos quake en alguno de los formatos disponibles .md

La principal ventaja de utilizar modelos de 3DStudioMax es que DirectX proporciona una utilidad (`conv3ds.exe`) que realiza la conversión de formato .3ds a formato .x automáticamente. Sin embargo, en su contra juega la menor disponibilidad de modelos gráficos realizados para juegos. Por otra parte no tenemos información suficiente sobre cómo han de integrarse las animaciones de modelos predefinidas por 3Dstudio con las animaciones que pretendemos para nuestro juego.

En cuanto a los modelos de quake, su principal ventaja es la abundancia de modelado gráfico disponible libremente en internet y el hecho de que el formato de los ficheros sea público. La desventaja de esta alternativa reside en el hecho de que no existe un conversor de ningún formato de modelo quake en el paquete de utilidades de DirectX.

Después de estudiar el formato de ficheros .md2 hemos decidido asumir esta alternativa y por tanto la construcción de un cargador de .md2 a nuestro juego, puesto que vemos más oportunidades de realizar las animaciones si controlamos directamente los datos de los vértices leídos del fichero, para de esta forma, insertarlos en nuestras propias estructuras de datos de polígonos, que formarán la figura del humanoide.

Formato del fichero de modelo quake 2 (.md2)

La estructura de datos en la que se guarda la cabecera del fichero .md2 definida en ClaseObjeto.h queda como sigue:

```
struct CABECERASMD2
{
                                                    int
identificador;
                                                    int
version;
                                                    int
anchoTextura;
```

```
                                                    int
largoTextura;
                                                    int
tamFrame;
                                                    int
numTexturas;
                                                    int
numXYZ;
                                                    int
numST;
                                                    int
numTriangulos;
                                                    int
numGLCMDS;
                                                    int
numFrames;
                                                    int
despTexturas;
                                                    int
despST;
                                                    int
despTriangulos;
                                                    int
despFrames;
                                                    int
despGLCMDS;
                                                    int
despFin;
};
```

Ahora, veamos qué propósito tiene cada componente de la cabecera del fichero:

- identificador – El número "mágico" (con la misma utilidad que el número mágico de los ficheros en UNIX) que nos dice si un fichero es realmente un .md2, este número es 844121161 si el .md2 es válido.
- version – El número de versión del fichero, debe ser siempre 8.
- anchoTextura – Nos dice el ancho de la textura utilizada para el modelo.
- altoTextura – Igual para la altura.
- tamFrame – El tamaño de cada uno de los frames (tramas o marcos) del modelo. Posteriormente esperamos obtener la sensación de movimiento mediante la rápida superposición de estas tramas o marcos uno sobre otro (técnica de animación mediante 'sprites').
- numTexturas – El número de texturas utilizadas en el modelo.
- numXYZ – Dice cuántos vértices tiene el modelo. Resulta útil, por ejemplo, para volcar la información a un fichero de registro, en el que se dé cuenta del nº de vértices que tienen los objetos cargados en el escenario.

- numST – El nº de coordenadas de textura que tiene el modelo (se asocian dos coordenadas de textura a cada posición de un vértice).
- numTriangulos – Informa sobre el nº de triángulos que tiene el modelo. Es un campo de más utilidad que numXYZ o numST.
- numGLCMDS – Informa sobre si el modelo fué optimizado para listas de triángulos (strips) en las que se comparten algunos de los vértices, o para ramilletes de triángulos (fans) en las que todos los triángulos del ramillete comparten un vértice. No es muy utilizado por los editores de modelos .md2.
- numFrames – EL número de frames que tiene el modelo.

Los siguientes campos determinan el desplazamiento (medido en bytes) respecto del principio del fichero, con la información referente a:

- despTexturas – El lugar dónde encontraremos las texturas
- despST – Las coordenadas de texturas
- despTriangulos – Los vertices de los triángulos
- despFrames – Los frames del modelo
- despGlCMDS – La información referente a la optimización
- despFin – El final del fichero.

Pasemos a cada una de las porciones que nos encontramos después de la cabecera:

Cada 'frame' del modelo (puede haber hasta 198) tiene la cabecera:

```
struct ALIASFRAME
{
                                                    float
escala[3];
                                                    float
translacion[3];
                                                    char
nombre[16];
                                                    VERTIC
ESTRIANGULO vertices[1];
};
```

En ella se guardan la escala y traslación del frame (posteriormente nosotros debemos escalarlo a una décima parte, puesto que los modelos aparecen demasiado grandes en los escenarios DirectX), así como un nombre descriptivo y el primero de los vértices del frame.

La estructura fundamental que debemos leer son los vértices en sí del modelo, lo que hacemos con la estructura:

```
struct VERTICESTRIANGULO
{
                                                    BYTE
vertices[3];
```

```
                                                                    BYTE
indiceLuzNormal;
      };
```

Cada byte leído se corresponde con una vértice del triángulo. El otro campo es un índice a una tabla interna de quake referente a iluminación, por lo que en nuestro caso simplemente se desprecia. Asimismo deberemos tener en cuenta que el sistema de coordenadas de quake no es el mismo que en DirectX puesto que la 'z' de quake es la 'y' en DirectX.

Pretendemos obtener un sistema flexible de inclusión de objetos en el escenario por lo que toda la información leída del fichero se guardará en el objeto instanciado a la clase:

```
      class ClaseObjeto
      {
                                                                    privat
e:

                                                                    //Solo
se usan al cargar el modelo
                                                                    ELEMLI
STAINDICES*
                                                                    listaI
ndices;
                                                                    ELEMLI
STAFRAMES*
                                                                    listaF
rames;

                                                                    //Nume
ro de frames,triangulos y vertices
                                                                    long
numFrames,numTriangulos,numVertices;

                                                                    //Sirv
e para guardar todos los datos del modelo
                                                                    DATOSM
ODELO
                                                                    listaD
atos[MAX_FRAMES];

                                                                    //Alma
cena la textura del objeto para no tener que
           //crearla cada vez que dibujamos

                                                                    LPDIRE
CT3DTEXTURE8 textura;
```

La estructura de datos fundamental de esta clase es:

```
      DATOSMODELO
                                                                  listaD
atos[MAX_FRAMES];
```

ListaDatos almacena para cada una de las 'frames' del modelo el número de vértices que posee y una lista enlazada a los vértices de los triángulos que la forman. De esta forma para dibujar un 'frame' simplemente pasaremos el puntero de esta lista como puntero al buffer de triángulos que DirectX debe dibujar con la primitiva de dibujo de triángulos.

DatosModelo se ha definido:

```
      struct DATOSMODELO
      {
                                                                  long


                                                                  numVer
tices;
                                                                  CUSTOM
VERTEX*
                                                                  listaV
ertices;
      };
```

CustomVertex almacena la misma información que para el resto de vértices que componen el escenario (posición, color, componente especular de la iluminación y coordenadas de las texturas).

Implementación de los Objetos

```
ClaseObjeto::ClaseObjeto()
```

Descripción:

Constructor del objeto. Inicializa los atributos privados.

Parámetros de entrada:

- Ninguno.

Parámetros de salida:

- Retorno de la función: Ninguno.

```
bool ClaseObjeto::cargarModelo                                    (char*
nombreFichero,
                                  LPDIRECT3DDEVICE8 lpDevice)
```

Descripción:

Realiza toda la carga del fichero .md2, empezando por la cabecera, continuando con las texturas asociadas y dando tantas pasadas de lectura de información de los vértices como 'frames' tenga el fichero.

Parámetros de entrada:
- NombreFichero: El nombre del fichero .md2
- LpDevice: El dispositivo DirectX al que se asociarán las texturas leídas.

Parámetros de salida:
- Retorno de la función: Éxito (verdadero) o fallo (falso) en la lectura.

```
bool ClaseObjeto::dibujaFrame(int numFrame,
                              LPDIRECT3DDEVICE8 lpDevice)
```

Descripción:

Dibuja el nº de frame del modelo que deseemos. Mediante repetidas llamadas a esta operación conseguimos la animación mediante superposición de los diferentes 'frames'.

Parámetros de entrada:
- Línea: Número de frame a dibujar de entre los disponibles.
- LpDevice: El dispositivo DirectX al que se manda a dibujar el buffer de triángulos.

Parámetros de salida:
- Retorno de la función: Verdadero si el número de frame solicitado existe en ese modelo, falso en caso contrario.

```
void ClaseObjeto::destruirModelo()
```

y

```
ClaseObjeto::~ClaseObjeto()
```

Descripción:

Destructores de ClaseObjeto. Liberan la memoria dinámica solicitada al cargar el modelo.

Parámetros de entrada:
- Ninguno

Parámetros de salida:
- Retorno de la función: Ninguno.

Adicionalmente se utilizan las siguientes funciones desarrolladas en la propia definición de la clase y de significado obvio:

```
inline int devCuentaFrame() {return numFrames;}
```

Devuelve el numero total de frames. Al ser funciones sencillas se definen como un macro.

```
inline int devCuentaVertices() {return numVertices;}
```

Igual para los vértices.

```
inline int devCuentaTriangulos() {return numTriangulos;}
```

Igual para los triángulos.

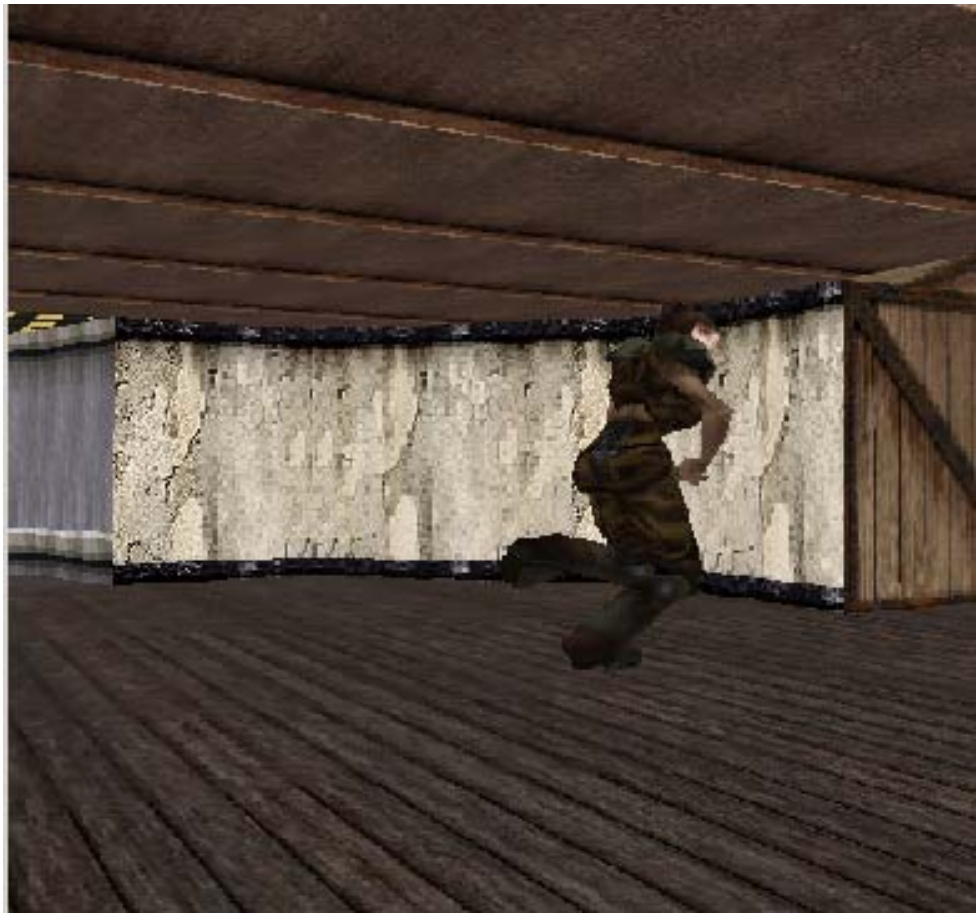
La carga del modelo se realizará mediante una llamada como:

```
objeto.cargarModelo("Objetos/TRIS.MD2",g_pd3dDevice);
```

y se dibujará un frame diferente en cada llamada automática (realizada internamente por DirectX) a `Render()`:

```
if (num_frame < MAX_FRAME)
++num_frame;
      else
num_frame = 0;
objeto.dibujaFrame((int)numFrame,g_pd3dDevice);
```

En la figura contigua aparece el modelo introducido:



Desplazamiento de los Modelos

Una vez conseguida la introducción de modelos de humanoides en el escenario y la animación del mismo permaneciendo en el mismo punto mediante la técnica de superposición de frames descrita en la anterior entrega, se plantea la necesidad de abordar el problema del desplazamiento de la figura por el escenario.

En un primer intento, adoptamos la estrategia de mover la figura moviendo cada uno de los triángulos que lo componen. De esta forma, leíamos por teclado el desplazamiento deseado (con la misma técnica de tratamiento del interfaz de teclado que ya utilizamos para las flechas de dirección), y se efectuaba el desplazamiento del dibujo sumando (o restando) al eje x o y un valor adecuado para conseguir el efecto deseado. Esta operación suponía extraer del buffer de dibujo y para cada una de las tramas (puede haber hasta 198) todos los triángulos que componían la figura y seguidamente sumar un desplazamiento (número en punto flotante) al eje de cada uno de los vértices, lo que genera un enorme número de operaciones a realizar.

Esta opción fue desechada al obtener el primer resultado en pantalla. El modelo se movía a golpes y el teclado dejaba de responder durante unos segundos. Examinando el rendimiento del juego mediante un medidor independiente de frames por segundo (fps), el resultado al dibujar solamente el escenario se mantiene de manera sostenida por encima de los 300 fps, bajando rara vez de 250 fps, lo que se puede calificar de resultado excelente.

Hemos incluido, mediante la tecla F2, la posibilidad de activar/desactivar el rechazo de hojas que están fuera del cono de visión, observando un ligero aumento de rendimiento cuando se activa el rechazo.

Los problemas aparecen al incluir el bucle de orden $O(N^2)$ de sumas en punto flotante que provoca que el rendimiento se divide por 10 cayendo hasta 30 fps. Esta cifra podría ser justificable si se tratara del rendimiento final esperado para el juego, pero dado que deben aparecer muchos más modelos diferentes en el escenario, el módulo de inteligencia artificial también se llevará su parte de capacidad de proceso, así como otras posibles mejoras relativas a iluminación, sonido,... optamos por abandonar esta técnica por su inaceptable rendimiento.

Solución seleccionada

Partimos de la base de que es necesario tratar la lista de triángulos a dibujar como un

todo, puesto que las operaciones individuales sobre cada uno de ellos son excesivamente costosas.

Todas las librerías gráficas proporcionan operaciones primitivas para el cálculo de las matrices aplicables a las 3 operaciones básicas: rotación, traslación y escalado. En DirectX la matriz de traslación a un punto se obtiene mediante:

```
D3DXMatrixIdentity(&matrizAux);
float x = this->posicionFinal.x;
float y = this->posicionFinal.y;
float z = this->posicionFinal.z;
D3DXMatrixTranslation(&matrizAux,x,y,z);
```

De esta forma tenemos en matrizAux la matriz que aplicada a un punto (mediante una simple multiplicación de matrices) nos da el punto desplazado.

Cada dispositivo Direct3DDevice8 definido tiene asociadas matrices de transformación que determinan la localización espacial, su rotación y tamaño así como el punto de vista, dirección y perspectiva del mundo en uso. Es posible obtener la matriz de transformación del estado actual mediante:

```
lpDevice->GetTransform(D3DTS_WORLD,&matrizMundo);
```

Ahora, tendremos la matriz de transformación (para nuestro dispositivo DirectX) para el estado del mundo actual en matrizMundo.

Seguidamente mediante multiplicación de matrices aplicamos la matriz de traslación a la matriz de transformación del mundo:

```
                                                          D3DXM
atrixMultiply(&matrizMundoTrans,&matrizAux,
                          &matrizMundo);
```

Asimismo es posible establecer la matriz actual de transformación del mundo a la nueva matriz obtenida:

```
lpDevice->SetTransform(D3DTS_WORLD,&matrizMundoTrans);
```

Inmediatamente después se manda a dibujar el buffer de triángulos mediante la primitiva habitual DrawPrimitiveUp (merece la pena observar que no se ha realizado ninguna operación específica sobre los vértices de los triángulos):

```
lpDevice->DrawPrimitiveUP(D3DPT_TRIANGLELIST,
  this->listaDatos[numFrame].numVertices/3,
 (BYTE**)&this->listaDatos[numFrame].listaVertices[0],
  sizeof(CUSTOMVERTEX));
```

Finalmente se vuelve al estado anterior al dibujo del modelo estableciendo la matriz de transformación original que tenía el mundo, dejando las cosas como estaban:

```
lpDevice->SetTransform(D3DTS_WORLD,&matrizMundo);
```

Mediante estas operaciones encadenadas de manipulación de matrices de transformación hemos logrado el desplazamiento del modelo sin caída del rendimiento general del juego, por lo que en lo sucesivo aplicaremos esta solución al resto de modelos que se incluyan en el escenario.

Con objeto de mantener la información relativa a la posición que ocupa el personaje en el escenario (así como la posición a la que se desplaza) y la dirección de la mirada, o lo que es lo mismo, el frente del personaje, se han incluido en ClaseObjeto los atributos:

```
                                                          D3DXV
ECTOR3
                                                          posic
ionObjeto;

                                                          D3DXV
ECTOR3
                                                          posic
ionFinal;
```

```
                                                    D3DXV
ECTOR3
                                                    direc
cionMirada;
```

Selección de las animaciones del personaje

Para el control de las diferentes animaciones que debe presentar un personaje se han incluido los atributos `flagMuerto` y `flagCorre` que indican el estado en el que se encuentra el personaje (por el momento, sólo contemplaremos 3 estados correspondientes a "correr", "muerto" y "en espera"). Si ninguno de los indicadores (flags) es verdadero consideraremos que está en espera.

Para cada una de las animaciones contempladas deberemos guardar el frame con la que comienza, su terminación y la que actualmente estamos mostrando en la secuencia de superposiciones que generan la animación.

Los atributos añadidos a ClaseObjeto son:

```
                                                    int
inicioEspera,finEspera,cuentaEspera;
                                                    int
inicioAtaque,finAtaque,cuentaAtaque;
                                                    int
inicioCorre,finCorre,cuentaCorre;
                                                    int
inicioMuerto,finMuerto,cuentaMuerto;

                                                    bool
flagCorre;
                                                    bool
flagMuerto;
```

Cada vez que el dispositivo DirectX llama automaticamente al procedimiento Render() se genera una llamada a dibujaModelo(), pero dado que el rendimiento del juego es alto se produce un parpadeo del modelo por el constante redibujado, por lo que hemos decidido redibujar el personaje una de cada 20 veces que se genera la llamada automática al Render(). De esta forma conseguimos un modelo estable y perfectamente perfilado, incluso al desplazarse por el escenario, a la vez que reservamos capacidad de proceso para el futuro.

La inicialización de los atributos se hace con valores apropiados para realizar el dibujo en solamente una de cada 20 pasadas:

```
                                                        this-
>inicioEspera = this->cuentaEspera = 0;
                                                        this-
>finEspera = 800;
                                                        this-
>inicioAtaque = this-> cuentaAtaque = 920;
                                                        this-
>finAtaque = 1080;
                                                        this-
>inicioCorre = this-> cuentaCorre = 800;
                                                        this-
>finCorre = 900;
                                                        this-
>inicioMuerto = this-> cuentaMuerto = 3800;
                                                        this-
>finMuerto = 3960;

                                                        //Ini
cializamos los flags
                                                        this-
>flagCorre = false;
                                                        this-
>flagMuerto = false;
```

Ahora las teclas ‘W’ y ‘S’ activan el flag correspondiente a corriendo, de forma que la próxima vez que se dibuje empezaremos por el frame correspondiente a esa animación, entrando en un bucle que llevará a la animación una y otra vez de principio a fin, hasta que se deje de pulsar la tecla de movimiento.

La tecla ‘X’ activa el flag de la animación correspondiente a la muerte del personaje.

En caso de que ninguno de estos flags esté activado se entra en el bucle de animaciones del estado en espera:

```
if (this->flagMuerto)
{
      this->cuentaMuerto = (this->cuentaMuerto < this->finMuerto-1) ?
                        this->cuentaMuerto + 1 : this->finMuerto-1;
      dibujaFrame(this->cuentaMuerto / 20,lpDevice);
      return;
}

if (this->flagCorre)
{
```

```
      this->cuentaCorre = (this->cuentaCorre < this->finCorre-1) ?
                        this->cuentaCorre + 1 : this->inicioCorre;
      dibujaFrame(this->cuentaCorre / 20,lpDevice);
      this->flagCorre = false;
      return;
}

this->cuentaEspera = (this->cuentaEspera < this->finEspera-1) ?
                        this->cuentaEspera + 1 : this->inicioEspera;
dibujaFrame(this->cuentaEspera / 20,lpDevice);
```

DibujaFrame() realiza las operaciones de transformación descritas en el apartado dedicado al desplazamiento de modelo.

La carga del modelo ahora incluye un parámetro que permite seleccionar la textura a aplicar, por lo que la llamada queda:

```
      objeto.cargarModelo("Objetos/tris/tris.md2",
                          "Objetos/tris/yo2.bmp",g_pd3dDevice);
```

Esto nos permitirá cargar un sólo modelo que sirva para varios personajes que presentarán distintas apariencias en pantalla con tan sólo cambiar de textura.

Ya que podemos mover libremente el objeto por el escenario lo situamos en su posición de partida con el método moverObjeto():

```
      objeto.moverObjeto(D3DXVECTOR3(0.0f,2.9f,0.0f));
```

Modificaciones en los métodos de ClaseObjeto

`ClaseObjeto::ClaseObjeto()`

Descripción:

Constructor del objeto. Se han incluido inicializaciones para la posición del modelo y las secuencias de cada animación que posea el personaje.

Parámetros de entrada:

- Ninguno.

Parámetros de salida:

- Retorno de la función: Ninguno.

```
bool ClaseObjeto::cargarModelo                                    (char*
nombreFichero,
```

```
                                                        char*
nombreTextura,
                                    LPDIRECT3DDEVICE8 lpDevice)
```

Descripción:

Se ha añadido un segundo parámetro para poder seleccionar la textura a aplicar al modelo. Esto permitirá dotar de diversas apariencias a un mismo modelo. Se vuelcan al archivo de registro el nombre y secuencia de animaciones del modelo cargado.

Parámetros de entrada:

- NombreFichero: El nombre del fichero .md2
- NombreTextura: El nombre del fichero .bmp
- LpDevice: El dispositivo DirectX al que se asociarán las texturas leídas.

Parámetros de salida:

- Retorno de la función: Éxito (verdadero) o fallo (falso) en la lectura.

```
bool ClaseObjeto::dibujaFrame(int numFrame,
                              LPDIRECT3DDEVICE8 lpDevice)
```

Descripción:

Esta operación ha sido completamente rediseñada incorporando la secuencia de manipulación de matrices de transformación descrita en el apartado “desplazamiento del modelo”.

Parámetros de entrada:

- Línea: Número de frame a dibujar de entre los disponibles.
- LpDevice: El dispositivo DirectX al que se manda a dibujar el buffer de triángulos.

Parámetros de salida:

- Retorno de la función: Verdadero si el número de frame solicitado existe en ese modelo, falso en caso contrario.

```
void ClaseObjeto::dibujaModelo(LPDIRECT3DDEVICE8 lpDevice)
```

Descripción:

Método añadido en esta versión. Se ocupa de decidir la animación a dibujar en función del estado de los atributos flagMuerto y flagCorre, realizando la llamada a dibujaFrame() con el parámetro adecuado. Recordemos que el estado de estos atributos cambia en función de las pulsaciones de tecla (W, S o X) leído por teclado tal como se describe en “selección de las animaciones del personaje”.

Parámetros de entrada:

- LpDevice: El dispositivo DirectX al que se manda a dibujar el buffer de triángulos.

Parámetros de salida:

- Retorno de la función: Ninguno.

```
void ClaseObjeto::moverObjeto(D3DXVECTOR3 posicionNueva)
```

Descripción:

Método añadido en esta versión. Modifica el atributo privado posicionFinal para que el personaje se desplace a la punto determinado por posicionNueva, siempre que el personaje no haya muerto.

Parámetros de entrada:

- posicionNueva: El punto objetivo del desplazamiento.

Parámetros de salida:

- Retorno de la función: Ninguno.

```
void ClaseObjeto::giroDerecha()
```

Descripción:

Método añadido en esta versión. Modifica el atributo privado direccionMirada de forma que el personaje girará su frente a la derecha en incrementos de 0.01 radianes. Esta operación se realiza en respuesta a la pulsación de la tecla 'D'.

Parámetros de entrada:

- Ninguno.

Parámetros de salida:

- Retorno de la función: Ninguno.

```
void ClaseObjeto::giroIzquierda()
```

Descripción:

Método añadido en esta versión. Igual que el anterior pero hacia la izquierda en decrementos de 0.01 radianes con la pulsación de la tecla 'A'.

Parámetros de entrada:

- Ninguno.

Parámetros de salida:

- Retorno de la función: Ninguno.

```
void ClaseObjeto::destruirModelo()
```

y

```
ClaseObjeto::~ClaseObjeto()
```

Descripción:

Destructores de ClaseObjeto. Se ha solucionado un error en el proceso de liberación de memoria dinámica de la versión anterior.

Parámetros de entrada:

- Ninguno

Parámetros de salida:

- Retorno de la función: Ninguno.

Detección de Colisiones Mediante Esferas

Se han contemplado todos los casos de colisión (paredes, objetos y protagonista) que se pueden dar en el escenario.

Objetos animados.

Incluimos en el método inicializarObjeto de ClaseObjeto el cálculo de la caja minimal (bounding box) que contiene completamente cada una de los frames de la animación del personaje. Esta caja que rodea al personaje se calcula igual que en el método calcularCajaMinimal ya utilizado en ClaseEscenario, se inicializan las posiciones mínimas y máximas a valores imposibles (para x,y,z) y se van comparando contra cada vértice en el momento en que se carga el personaje. De esta forma podemos mantener la posición máxima y la mínima, y por tanto lo que nos interesa, que es el centro del personaje.

En un primer momento consideramos la posibilidad de mantener un único centro para cada personaje, pero dado que la secuencia concreta en la que se encuentre el personaje en cada momento puede hacer variar enormemente este punto (si salta por ejemplo) finalmente hemos incluido cálculo de un punto central para cada frame, es decir para cada uno de los momentos de la animación. Pensamos que, aunque esta elección introduce mucha más complejidad en la detección de colisiones, éstas serán mucho más realistas que si sólo se toma un punto central aproximado del objeto animado.

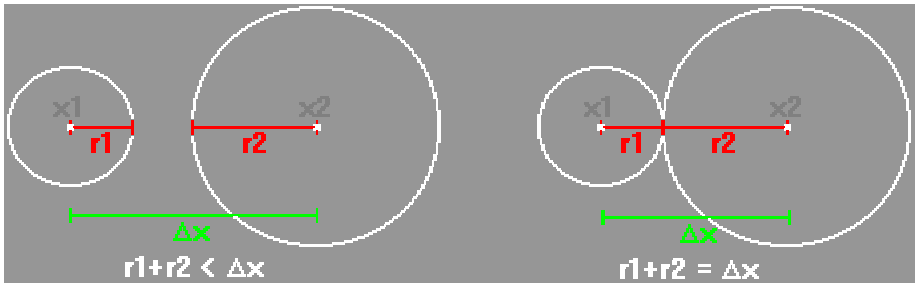
En torno al centro calculado se traza una esfera imaginaria, de forma que cada vez que se dibuja el escenario se realiza el movimiento previsto del desplazamiento (aún sin dibujarlo) y se comprueba la colisión con detectarColision para una esfera de radio dado, si no existe inconveniente el movimiento se realiza, en caso contrario se retrocede a la posición inicial.

DetectarColision (para el caso de los objetos) comprueba que no exista colisión ni con el escenario (con una llamada a colisionEsferas), ni con el resto de personajes/objetos (con una llamada a colision2Esferas) incluyendo entre ellos al protagonista, es decir, la posición de la cámara.

Para detectar la colisión entre la esfera de un objeto y el escenario, se clasifica el punto central de la esfera con el plano del polígono que tenemos en la dirección del movimiento (el recorrido recursivo del árbol BSP en busca de colisión partiendo de la raíz –nodo 0- nos dice cuál es), para ello hacemos uso

de la función clasificarPunto que ya existía en ClaseEscenario (recordemos que devolvía Delante, Detrás o Coincidentes). Seguidamente se calcula el punto de la esfera más cercano al plano (a partir de la normal del plano y el centro de la esfera) y se clasifica también este punto. Si ambos puntos están delante del plano no hay colisión, si los puntos quedan en lados contrarios se devuelve colisión.

La detección de la colisión entre la esfera de un objeto y otro objeto se realiza simplemente calculando el vector que une los centros y comprobando si su longitud es mayor que la suma de sus radios, en ese caso sabemos que no hay colisión:




```
bool ClaseEscenario::colision2Esferas (D3DXVECTOR3 centro1,
                 float radio1,D3DXVECTOR3 centro2,float radio2)
{
                                                          D3DXVE
CTOR3 vectorCentros = centro2 - centro1;
                                                          float
moduloVector = D3DXVec3Length(&vectorCentros);
                                                          return
!(radio1 + radio2 < moduloVector);
}
```

Protagonista.

Hasta ahora habíamos tratado la posición de la cámara como una variable global del juego que determinaba lo que se veía, lo que no, etc. Pero al plantearnos las colisiones con la cámara hemos decidido crear una nueva clase con atributos que mantengan todos los aspectos relativos al comportamiento del protagonista en el juego. Además el hecho de manejar al protagonista con métodos que existen igualmente para los objetos, puede darnos la posibilidad en el futuro de hacer que el protagonista juegue de modo automático utilizando el módulo de inteligencia artificial.

La detección de colisiones en este caso reside igualmente en comprobar que la esfera que rodea al punto central (lo que hasta ahora hemos llamado posición de la cámara), no colisiona ni con el escenario ni con el resto de objetos. Para ello se utilizan las mismas funciones expuestas anteriormente, cambiando solamente el parámetro que indica que se trata de la cámara/protagonista:

```
if (modo == COL_CAMARA)
{
                                                        result
adoEscenario = this-colisionEsferas(&centro,radio,0);
                                                        result
adoResto = this->colision2Esferas(centro,radio,
                    this->objeto.devPosicionObjeto(),1.0f);
                                                        return
resultadoEscenario || resultadoResto;
}

if (modo == COL_OBJETO)
{
     resultadoEscenario = this->colisionEsferas
          (&this->objeto.devPosicionObjeto(),1.0f,0);
                                                        result
adoResto = this->colision2Esferas(centro,radio,
          this->objeto.devPosicionObjeto(),1.0f);
                                                        return
resultadoEscenario || resultadoResto;
}
```

Implementación de ClaseProtagonista

Se han incluido atributos privados referentes a la vista en primera persona:

```
                                                        D3DXV
ECTOR3 posicionDeLaCamara;

                                                        D3DXV
ECTOR3 direccionVista;

                                                        float
anguloCabeza;

                                                        float
anguloMirada;
```

Con unos métodos incluidos en la propia definición para acceder a los dos de uso frecuente desde el exterior:

```
inline D3DXVECTOR3 devPosicionCamara()
          {return this->posicionDeLaCamara;}
inline D3DXVECTOR3 devDireccionVista()
          {return this->direccionVista;}
```

Se suministran dos constructores alternativos para inicializar los atributos:

```
ClaseProtagonista::ClaseProtagonista()
```

y

```
ClaseProtagonista::ClaseProtagonista(D3DXVECTOR3
posicionDeLaCamara, D3DXVECTOR3 direccionVista, float
anguloMirada, float anguloCabeza)
```

Métodos para cada movimiento del protagonista (movimiento ya implementado varias versiones atrás), cada uno con su propio método, de forma que el protagonista va a responder a las mismas órdenes que los objetos animados del escenario:

```
//Metodo que hace avanzar al protagonista
void avanza();
//Metodo que hace retroceder al protagonista
void retrocede();
//Metodo que desplaza a la derecha al protagonista
void desplazamientoDerecha();
//Metodo que desplaza a la izquierda al protagonista
void desplazamientoIzquierda();
//Metodo que mueve la cabeza del protagonista verticalmente
void mirarVertical(float coordenadas);
//Metodo que mueve la cabeza del protagonista
horzontalmente
void mirarHorizontal(float coordendas);
```

La estructura de todo el programa (especialmente el principal) ha sido adaptada para que las llamadas a esta nueva clase partan desde el escenario (ClaseEscenario) como ya ocurría con los modelos animados. Esta disposición es más razonable y estructurada si pensamos que tanto el protagonista como los objetos animados evolucionan dentro del escenario.

Inclusión de múltiples objetos

El único objeto de ClaseObjeto ha sido sustituido por un vector de 5 posiciones de objetos de ClaseObjeto. Dado que desde el primer momento tuvimos la previsión de que todos los detalles relativos a los personajes animados ya estuvieran encapsulados dentro de su clase, empezar a manejar varios a la vez sencillamente consiste en instanciarlos. El resto del código de inicialización es trivial:

```
void ClaseEscenario::inicializarObjetos(LPDIRECT3DDEVICE8 lpDevice)
{
                                                                    listaOb
jetos[0].cargarModelo("Objetos/Tris/tris.md2","Objetos/Tris/yo2.bmp",l
pDevice);
                                                                    listaOb
jetos[1].cargarModelo("Objetos/Civil/tris.md2","Objetos/Civil/Civil.bm
p",lpDevice);
                                                                    listaOb
jetos[2].cargarModelo("Objetos/Ripper/tris.md2","Objetos/Ripper/Ripper
.bmp",lpDevice);
                                                                    listaOb
jetos[3].cargarModelo("Objetos/Hobgoblin/tris.md2","Objetos/Hobgoblin/
Hobgoblin.bmp",lpDevice);
                                                                    listaOb
jetos[4].cargarModelo("Objetos/Blade/tris.md2","Objetos/Blade/Blade.bm
p",lpDevice);

                                                                    listaOb
jetos[0].colocarObjeto(10.0f,-10.0f);
                                                                    listaOb
jetos[1].colocarObjeto(-10.0f,-10.0f);
                                                                    listaOb
jetos[2].colocarObjeto(10.0f,10.0f);
                                                                    listaOb
jetos[3].colocarObjeto(-10.0f,10.0f);
                                                                    listaOb
jetos[4].colocarObjeto(0.0f,0.0f);
```

Para tratar cada uno de ellos aplicamos la acción al elemento adecuado del array, como por ejemplo en el desplazamiento:

```
if (this->detectarColision(
            this->listaObjetos[indiceObjeto].devPosicionObjeto(),
            1.0f,COL_OBJETO))
```

De esta forma ha sido necesario rescribir los métodos inicializarObjetos y moverObjeto (que ahora posee un parámetro entero con el índice del objeto animado a desplazar), pero no se ha tocado la implementación de ClaseObjeto.

Implementación de múltiples objetos

```
void ClaseEscenario::moverObjeto(int indiceObjeto)
```

Descripción:

Realiza el desplazamiento del personaje que se pasa por parámetro si del avance o retroceso no resulta una colisión. Como está pendiente de realizar un módulo de inteligencia artificial que se ocupe de determinar sus movimientos, por el momento, sencillamente avanzan hasta que chocan con una pared, en ese momento retroceden.

Parámetros de entrada:

- IndiceObjeto: El número de personaje que queremos desplazar

Parámetros de salida:

- Retorno de la función: Ninguno.

```
void ClaseEscenario::inicializarObjetos(LPDIRECT3DDEVICE8 lpDevice)
```

Descripción:

Realiza la carga del modelo y lo sitúa en el escenario.

Parámetros de entrada:

- LpDevice: El dispositivo DirectX sobre el que se dibuja.

Parámetros de salida:

- Retorno de la función: Ninguno.

A continuación mostramos diversos modelos introducidos conjuntamente en el escenario:

Funcionalidad de disparo

Se han añadido los métodos:

- Disparar en la claseProtagonista
- DispararProtagonista, dibujaPuntoMira y ColisionRayoEsfera en la ClaseEscenario
- Atacado en la ClaseObjeto

Una nueva clase llamada ClaseArma.

Un atributo arma instanciado a ClaseArma en ClaseProtagonista, para representar el arma que porta el protagonista.

```
void ClaseProtagonista::disparar(D3DXVECTOR3 *inicioRayo,
                        D3DXVECTOR3 *finRayo,int *danyo)
```

Descripción:

"Realiza" el disparo, calculando el final del disparo en función de la posición actual, el alcance del arma y la dirección a la que se está mirando. Asimismo calcula el daño en función del arma utilizada.

Parámetros de entrada:

- Ninguno

Parámetros de salida:

- InicioRayo: La posición actual del protagonista.
- FinRayo: La posición final calculada para el disparo.
- Danyo: El daño que causa ese disparo.
- Retorno de la función: Ninguno.

```
bool ClaseEscenario::colisionRayoEsfera(D3DXVECTOR3 centro,
            float radio,D3DXVECTOR3 inicioRayo,D3DXVECTOR3 finRayo)
```

Descripción:

Determina si una esfera es alcanzada en algún punto por el disparo. Para ello se calculan los vectores que unen el punto de inicio del disparo con el centro de la esfera y con el final del disparo, podemos hacer que estos dos vectores formen un triángulo rectángulo de forma que podemos calcular la longitud del segundo cateto. Existe colisión si la longitud del cateto calculado es menor (cae dentro) que el radio de la esfera.

Parámetros de entrada:

- Centro: El centro de la esfera
- Radio: El radio de la esfera

- InicioRayo: La posición actual del protagonista.
- FinRayo: La posición final calculada para el disparo.

Parámetros de salida:

- Retorno de la función: Verdadero si el rayo impacta la esfera, falso en caso contrario.

```
void ClaseEscenario::dispararProtagonista()
```

Descripción:

Llama a protagonista.disparar para determinar los parámetros del disparo y la posición actual. Seguidamente, para cada uno de los objetos en el escenario determina si hay colisión entre el rayo del disparo y la esfera de cada uno. Si la hay reduce la vida del objeto atacado.

Parámetros de entrada:

- Ninguno.

Parámetros de salida:

- Retorno de la función: Ninguno.

```
void ClaseEscenario::dibujaPuntoMira(D3DXVECTOR3 posicionDeLaCamara,
                        LPDIRECT3DDEVICE8 lpDevice)
```

Descripción:

Calcula los 4 puntos que formarán el objetivo del punto de mira, dibujándolo con la primitiva de dibujo de listas de líneas en color rojo.

Parámetros de entrada:

- PosicionDeLaCamara: La posición actual
- LpDevice: El dispositivo DirectX sobre el que se dibuja.

Parámetros de salida:

- Retorno de la función: Ninguno.

```
void ClaseObjeto::atacado(int danyo)
```

Descripción:

Reduce la vida del personaje en función del daño pasado por parámetro, poniendo el atributo flagMuerto a verdadero en caso de que la vida sea igual o menor que 0. Esto hará que en la próxima pasada de dibujo se active la animación correspondiente a la muerte del personaje.

Parámetros de entrada:

- Danyo: El daño causado por el arma utilizada.

Parámetros de salida:

- Retorno de la función: Ninguno.

Por el momento la ClaseArma es muy básica, de forma que sólo contempla los siguientes atributos privados:

```
                                                        //Indic
a el alcance que tiene el arma
```

```
                                                                    float
alcance;

                                                                    //Indic
a el daño que hace el arma
                                                                    int
danyo;
```

Y los métodos de significado obvio:

```
                                                                    //Const
ructor de la clase
                                                                    ClaseAr
ma();
                                                                    //Const
ructor de la clase alternativo
                                                                    ClaseAr
ma(float alcance,int danyo);
                                                                    //Devue
lve el alcance del arma
                                                                    inline
float devAlcance() {return this->alcance;}
                                                                    //Devue
lve el daño que hace el arma
                                                                    inline
int devDanyo() {return this->danyo;}
```

PARTE III

PERFECCIONAMIENTO

Pantalla completa

Se ha modificado la inicialización de la ventana de juego con el parámetros WS_POPUP que crea la ventana a pantalla completa obteniendo las coordenadas máximas de la pantalla con GetSystemMetrics:

```
HWND hWnd = CreateWindow( "Proyecto", Version, WS_POPUP, 0, 0,
                                                          GetSyst
emMetrics(SM_CXSCREEN),
                                                          GetSyst
emMetrics(SM_CYSCREEN),
                                  NULL, NULL, wc.hInstance, NULL );
```

Por el momento no ha sido posible controlar el cambio pantalla completa/ventana de Windows mediante una tecla, por lo que queda como una mejora pendiente.

Se ha incluido la profundidad de color R5G6B5 (5 bits para el rojo, 6 para el verde y 5 el azul), ya que aunque ya estaba contemplada una profundidad de color de 16 bits (X1R5G5B5, con un bit desechado), algunas tarjetas gráficas utilizan esta otra posibilidad:

```
d3dpp.BackBufferFormat = D3DFMT_R5G6B5;
```

Mejoras en el rendimiento y realismo

El cuerpo principal de nuestro programa no es más que un bucle infinito en el que se realizan las siguientes operaciones:

- Se capturan los eventos relacionados con Windows: es decir, todo lo referente a las ventanas y demás acciones relacionadas con el sistema operativo.
- Dibujamos la pantalla: con la llamada a render vamos a generar todo el escenario y todos los objetos y los vamos a dibujar en pantalla.
- Hacemos un barrido de teclado y ratón: con esto comprobamos todo lo relacionado con la entrada de datos al juego, ya sea por ratón o por teclado.

El problema que surge es que este bucle infinito se va a ejecutar tantas veces como sea capaz de realizar la máquina sobre la que esté corriendo, por lo que vamos a consumir todos los recursos de proceso del sistema.

Esto nos lleva a que, en una máquina con mucha capacidad de proceso el juego va a ejecutarse de forma satisfactoria y a una velocidad aceptable. Sin embargo en una máquina con poca capacidad el juego se verá muy lento, casi desesperante. Esto se debe a que las constantes de movimiento que nosotros damos al juego son muy pequeñas, pero como en una máquina rápida se ejecutan muchas veces conseguimos un movimiento extremadamente suave, pero que para una máquina lenta se convierte en un movimiento muy lento.

Resumiendo, tenemos dos problemas: el bucle infinito y las diferencias entre máquinas. Vamos a plantear soluciones para estos dos problemas:

Solución al bucle infinito:

Hasta ahora, todos los ejemplos que hemos visto utilizan este método para funcionar, sin embargo también es cierto que para aplicaciones comerciales esta solución no parece la más correcta.

La mejor solución sería la de cambiar el bucle infinito por interrupciones que se generaran cada cierto tiempo, y así que se pintara la pantalla y se leyera el ratón y el teclado.

Esta solución no la hemos implementado todavía ya que pensamos que es algo secundario, teniendo en cuenta que nos queda mucho por hacer dentro del apartado gráfico y del juego en sí. Así que en principio lo dejaremos a un lado por el momento.

Solución para las diferencias entre máquinas:

La primera solución que se nos ocurre es muy simple, y se trata de tener algún tipo de función o algoritmo que nos diga el rendimiento aproximado de la máquina sobre la que se está ejecutando el juego, y de este modo poder adaptar el juego a la máquina.

Esta solución ha sido implementada en esta versión de la siguiente forma:

Cuando nosotros realizamos la primera llamada a render, calculamos el tiempo en milisegundos que tarda en hacerlo y luego ya comprobamos el ratón y el teclado aparte.

Con este tiempo calculamos un número de veces en el que el render no se va a dibujar en las siguientes ocasiones. Este número de veces va a ser el tiempo en milisegundos/10. Vamos a explicarlo más detenidamente con un ejemplo:

1. Calculamos el tiempo que tarda en hacer un render: 30 ms.
2. Dividimos este tiempo entre 10 para calcular el número de veces que no se va a pintar el render: 30/10 = 3 veces.
3. En las siguientes 3 pasadas no se va a realizar el render, pero sí la lectura del ratón y del teclado.
4. Una vez que se han realizado estas 3 pasadas volvemos a hacer el render y a calcular el tiempo que tarda en hacerlo, y por lo tanto repetimos el proceso.

Así lo que vamos a conseguir es que aunque no se actualice la pantalla en todas las pasadas del bucle, sí que vamos a seguir leyendo el ratón y el teclado, por lo que el movimiento sí que se va a realizar.

Por lo tanto en máquinas lentas, en las que se tardará mucho en hacer el render, vamos a conseguir movimientos más rápidos, y en máquinas rápidas en las que se tarde poco en hacer el render, se harán más render, aunque mucho más rápidamente que en una máquina lenta.

Con esto, hemos conseguido que el movimiento del protagonista sea parecido en máquinas lentas y rápidas, sin embargo en máquinas muy lentas vamos a encontrarnos con que es posible que el movimiento se realice a saltos, aunque como siempre pasa, hay que poner un límite de requisitos mínimos, y eso lo tendremos que probar.

Un inconveniente que tiene esta solución con la versión actual, es que el movimiento de objetos animados (enemigos), ahora mismo se realiza dentro

del render, y por lo tanto en máquinas lentas al hacer menos llamadas a render, el movimiento de los enemigos es muy lento.

Esto se puede solucionar haciendo que el movimiento de los enemigos (en un futuro cercano controlado por inteligencia artificial), se haga a la vez que se realiza la lectura del ratón y el teclado o simplemente adecuando el número de veces que se hace el render al movimiento de los personajes.

Este problema sigue estando presente en esta versión, pero se solucionará en siguientes versiones.

Implementación de las mejoras

Declaramos las variables locales para medir el tiempo:

```
SYSTEMTIME tiempo1;
SYSTEMTIME tiempo2;
```

Con esto calculamos el tiempo que tarda en hacer un render, metiendo el tiempo inicial y el final en las variables anteriores. La variable ***dibuja*** indica si se tiene que hacer el render, y ***pasadas*** es el contador que nos indica el número pasadas que nos quedan por realizar sin hacer render:

```
GetLocalTime(&tiempo1);
if (dibuja)
{
  Render(tiempo);
}
else
{
  pasadas = pasadas-1;
  if (pasadas<0)
  {
    dibuja = true;
    Render(tiempo);
  }
}
GetLocalTime(&tiempo2);
```

Se lee el teclado y el ratón:

```
salir = LeeTeclado(interfaz);
LeeRaton(interfaz);
```

Calculamos el número de veces que se va a dejar de hacer el render:

```
if (dibuja && tiempo1.wSecond == tiempo2.wSecond)
{
  tiempo = tiempo2.wMilliseconds-tiempo1.wMilliseconds;
  pasadas = tiempo/10;
  dibuja = false;
}
return salir;
```

Mejora del realismo en la muerte de los enemigos:

Un problema que teníamos en versiones anteriores es que en la muerte de un enemigo, éste quedaba flotando en el aire.

Este enemigo al morir está flotando en el aire

Esto se debe a que nosotros hasta ahora dibujábamos el modelo según su centro de todos sus vértices, pero sólo en la primera ocasión, y además siempre lo dibujábamos a la misma altura.

Ahora lo que hacemos es calcular en todo momento (frame del objeto a dibujar) el centro de todos los vértices usando la caja minimal.

La caja minimal del frame nos devuelve 2 vértices, el primero con las coordenadas máximas de todos los vértices de ese frame, y el segundo con las coordenadas mínimas de todos los vértices del frame.

Con los vértices de la caja minimal es fácil calcular el centro del objeto en cada frame: (vértice máximo + vértice mínimo)/2. Así ya tenemos el centro, y una vez que tenemos el centro del frame ya podemos calcular la altura del centro y del frame, por lo que ahora sólo nos queda posicionar el frame en todo momento a la altura que hemos calculado.

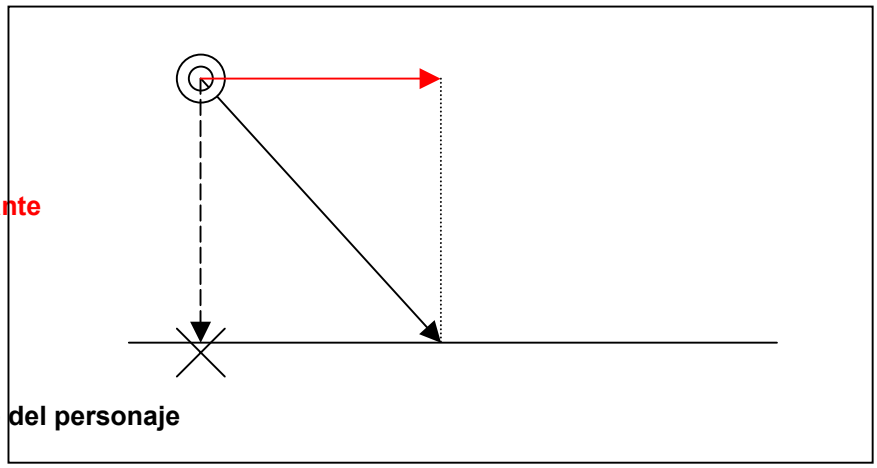
Gracias a esto conseguimos que el realismo a la hora de morir un enemigo sea mayor:

Este enemigo muere con más realismo, a la altura del suelo

Mejora del algoritmo de control de colisiones:

En versiones anteriores ya hemos modificado el control de colisiones con éxito, incluyendo el control de colisiones con esferas contra el escenario y contra otros objetos.

Sin embargo en control de colisiones sigue teniendo un pequeño defecto, y es que si el protagonista se choca contra un obstáculo, se para sin más. Nosotros hemos pensado que lo más lógico es que si el protagonista incide contra una pared en un ángulo agudo, el movimiento hecho contra la pared sea descartado y usando la componente paralela a la pared nos siguiéramos moviendo.



Nos hemos aprovechado de que nuestro algoritmo del escenario guarda la normal de las paredes, sin embargo aunque hemos conseguido que en la mayoría de las paredes esto funcione, falla en algunos puntos, por lo que dejamos a un lado esta implementación en espera de encontrar una solución al problema.

Sobreimpresión de texto en pantalla

El SDK de Directx proporciona, aparte de las librerías gráficas en sí, diversas utilidades y ejemplos adicionales para facilitar o explicar su uso. Inicialmente hemos tratado de utilizar el paquete d3dxfont.h/d3dxfont.cpp para lograr la sobreimpresión de caracteres en la pantalla del juego. Sin embargo, el sistema de dibujo de las fuentes ha resultado ser incompatible con nuestro sistema de dibujo del escenario. D3dfont dibuja las fuentes mediante triángulos con el sistema de buffer de vértices bloqueado (lock) que nosotros utilizamos en las primeras versiones del juego. Actualmente utilizamos un sistema de buffer de vértices mucho más perfeccionado, que ni siquiera necesita bloquearse para introducir los datos de los triángulos. El bloqueo de un tipo de buffer es incompatible con la definición del otro buffer. Por otra parte, d3dfont toma parte de sus parámetros de los definidos para el dibujo del escenario por lo que el resultado no es el de una sobreimpresión, sino que se integra en el escenario tomando incluso las texturas utilizadas para los objetos y muros.

A la vista de estas dificultades, hemos decidido realizar nuestra propia definición de fuentes a partir del interfaz ID3DXFont que poseen las librerías de DirectX. De esta forma es posible definir un puntero a este interface, una estructura de datos normalizada para la creación de fuentes LOGFONT (que explicamos en el código) y unirlos al dispositivo DirectX que se ocupa de dibujar en pantalla mediante una llamada a la primitiva D3DXCreateFontIndirect. Esta llamada devuelve un puntero a la fuente conteniendo el bitmap de cada uno de los caracteres, construido en función de los parámetros suministrados a partir de la definición de la fuente en formato true type (.ttf de la carpeta windows\fonts del sistema operativo). De esta forma obtenemos una gran flexibilidad en la creación de las fuentes, ya que potencialmente disponemos de todas las fuentes cargadas, y por tanto disponibles en el sistema operativo.

Finalmente, para mostrar los caracteres deseados en pantalla, sólo es necesario añadir dentro de la primitiva Render(), que DirectX llama automáticamente para dibujar la pantalla, la definición de un rectángulo que contendrá el texto, un buffer con el propio texto y una llamada a la primitiva DrawText del interfaz antes mencionado.

Al definir el color en la llamada a DrawText hemos incluido, aparte de los tres canales habituales (rojo, verde, azul) un canal que define la transparencia (en la terminología de DirectX al byte que define transparencia se le conoce como canal alpha), con lo que obtenemo un interesante resultado, permitiéndonos definir niveles de transparencia a elección del programador para cada tipo diferente de fuente, desde una leve sombra cercana a la transparencia total hasta la opacidad completa habitual de los textos sobreimpresos. De esta

forma el parámetro de color se ha definido como 0xAARRGGBB. Adicionalmente, la inclusión de estas características no ha penalizado el rendimiento.

Para mejorar la cohesión, se ha creado una nueva ClaseFuente que contiene todas las definiciones necesarias, en la que el método dibujaFuente se ocupa de realizar la labor de impresión de todos los textos. Tan sólo es necesario incluir una llamada a este método en el Render() entre el comienzo de la escena BeginScene() y su fin EndScene(), para que los datos de dibujo del texto estén disponibles en el momento de hacer el Present() que volcará el backbuffer al frontal de la pantalla para mostrarla.

Definición de fuentes: Interface ID3DXFont

```
void ClaseFuente::inicializarFuentes(LPDIRECT3DDEVICE8 g_pd3dDevice)
```

Descripción:

Inicializa la fuente a utilizar, para ello llama a D3DXCreateFontIndirect con el puntero al dispositivo, el puntero a la fuente y el puntero a la siguiente estructura:

```
                                                                LOGFONT
definicion1={

                                                                   32,
//alto

                                                                   0,
//ancho

                                                                   0,
// estos dos parámetros definen la orientación

                                                                   0,
// medida en décimas de grado

FW_BOLD, // grosor

FALSE, // itálica

FALSE, // subrayado

FALSE, // efecto hueco

DEFAULT_CHARSET, // conjunto de caracteres por defecto

OUT_DEFAULT_PRECIS, //precisión al dibujar
```

```
CLIP_DEFAULT_PRECIS, // precisión de recorte

ANTIALIASED_QUALITY,// calidad de impresión

DEFAULT_PITCH,// pitch por defecto

"LCD" // nombre de fuente
                                                                    }
```

Dejamos el conjunto de caracteres en la selección por defecto, puesto que los europeos ya están incluidos.

Igualmente ponemos la precisión al dibujar (por defecto es máxima precisión) y el recorte (por si las letras se salen de la pantalla) en los parámetros por defecto.

Activamos la función de AntiAliasing para mejorar el aspecto de la fuente en pantalla. No obstante, a pesar de tener esta función activada, al orientar diagonalmente el texto aparecen los molestos efectos "de escalera", si bien es cierto, que ese efecto mejora al aumentar la resolución del backbuffer.

El nombre de la letra, es el nombre que aparece dentro del fichero true type que contiene la letra, escrito tal y como aparece definido en el sistema operativo.

Parámetros de entrada:

- `g_pd3dDevice`: LpDevice: El dispositivo DirectX sobre el que se dibuja.

Parámetros de salida:

- Retorno de la función: Ninguno.

`void ClaseFuente::dibujaFuente(void)`

Descripción:

Vuelca cada uno de los textos, asociándoles uno de los tipos de letra anteriormente definidos, al backbuffer. Es posible seleccionar un color de dibujo y un nivel de transparencia como ya se ha explicado. El lugar de dibujo se controla definiendo un rectángulo de tipo RECT con los parámetros deseados, que contendrá el texto. Dado que el texto se transfiere primero a un buffer, es posible incluir variables en el texto (vida, tiempo, posición,…) utilizando los parámetros de formato de texto de sprintf.

Parámetros de entrada:

- Ninguno.

Parámetros de salida:

- Retorno de la función: Ninguno.

`void ClaseFuente::liberaFuente(void)`

Descripción:

Libera la memoria utilizada para la clase.

Parámetros de entrada:

- Ninguno

Parámetros de salida:

- Retorno de la función: Ninguno.

El resultado de la inclusión de texto semitransparente sobreimpreso en pantalla aparece en la siguiente figura:

Sonidos

Procederemos a la inclusión de sonidos, con las siguientes características:

- Dos sonidos disponibles.
- Si el evento que activa un sonido se produce una vez, y vuelve a activarse antes de que acabe de sonar el anterior, el sonido volverá a empezar (también ocurre esto si es otro sonido, pero esta dentro del mismo objeto).
- Música de fondo en formato MIDI.

Vamos a analizar el código para verlo más a fondo:

Lo primero que haremos será crear los interfaces necesarios para que funciones el sonido:

```
IDirectMusicLoader8*      g_pCargador                          = NULL;
IDirectMusicPerformance8* g_pEjecutor                          = NULL;
IDirectMusicSegment8*     g_pSegmento                          = NULL;
```

Luego inicializamos el COM (Component Object Model, usado por DirectX) y creamos el objeto cargador, y el objeto ejecutor:

```
CoInitialize(NULL);

CoCreateInstance(CLSID_DirectMusicLoader, NULL,


CLSCTX_INPROC, IID_IDirectMusicLoader8,


(void**)&g_pCargador);

CoCreateInstance(CLSID_DirectMusicPerformance, NULL,


CLSCTX_INPROC, IID_IDirectMusicPerformance8,


(void**)&g_pEjecutor );
```

A continuación procedemos a inicializar el audio, nosotros lo hemos hecho como se ve a continuación:

```
g_pEjecutor->InitAudio(
      NULL, // IDirectMusic interface, no es necesario.
                                                              NULL,
// IDirectSound interface, no es necesario.
```

```
                                                                    NULL,
// Window handle.
                                                                    DMUS_AP
ATH_SHARED_STEREOPLUSREVERB, // Tipo audiopath por defecto.
                                                                    64,


                                                                    //
Numero de canales.
                                                                    DMUS_AU
DIOF_ALL,                                                           //
Caracteristicas del sintetizador.
                                                                    NULL


                                                                    //
Parametros de audio, por defecto.
);
```

Para cargar un archivo de audio, que luego vamos a poder hacer sonar a través de la tarjeta de sonido se usa lo siguiente:

```
g_pCargador->LoadObjectFromFile(
      CLSID_DirectMusicSegment,   // Identificador de clase.
                                                                    IID_IDi
rectMusicSegment8,   // ID del interfaz deseado.
                                                                    ruta,



                                                                        //
Nombre del archivo.
                                                                    (LPVOID
*) &g_pSegmento      // Puntero que recibe el interfaz.
)
```

Así, ya tenemos el archivo cargado en memoria y sólo nos queda hacerlo sonar cuando queramos, de eso se encarga lo siguiente. Primero asignaremos el camino de audio (Audio Path) por donde va a sonar, y luego lo hacemos sonar:

```
g_pSegmento->Download( g_pEjecutor );
g_pEjecutor->PlaySegmentEx(
      g_pSegmento, // Segmento que va a sonar.
      NULL,        // Usado para canciones, ahora no lo usamos.
      NULL,        // Usado para transiciones.
      0,           // Flags.
      0,           // Tiempo de comienzo, 0 es inmediato.
                                                                    NULL,
// Puntero que recibe el estado del segmento.
                                                                    NULL,
// Objeto a parar.
                                                                    NULL
// Audiopath, si no esta, coge por defecto.
);
```

Niebla Volumétrica

Los efectos atmosféricos volumétricos se han utilizado en la programación de juegos con objeto de dar una sensación de mayor realismo al escenario dibujado. Para ello se crea un sistema de partículas que simula el "volumen" del humo, fuego, agua, niebla, etc. En nuestro caso hemos optado por incorporar niebla volumétrica (aunque de hecho, el efecto sería el mismo para el agua, tan sólo cambiando el color). Aumentando la densidad de las partículas en función de la distancia desde el punto de vista se crea un ambiente de bruma a tu alrededor que te envuelve como si estuvieras dentro.

En DirectX el efecto de niebla se implementa mediante la mezcla del color de los objetos en una escena con un determinado color escogido para la niebla, basándose en la distancia de un objeto en la escena (su distancia al punto de vista). A medida que los objetos se distancian, sus colores originales se van diluyendo más y más en el color escogido para la niebla, creando la ilusión de que el objeto está siendo oscurecido por finas partículas flotando en la escena.

Implementación de la niebla

A continuación pasamos a explicar los pasos a dar y el interfaz de programación que ofrece DirectX para este efecto:

Activamos el efecto "fog" para el dispositivo DirectX que hemos creado:

```
g_pd3dDevice->SetRenderState (D3DRS_FOGENABLE, TRUE);
```

Existen dos tipos de niebla, "pixel fog" y "vertex fog", dado que la niebla "por píxel" utiliza las fórmulas de cálculo propias del dispositivo adaptador, preferimos seleccionar la niebla "por vértices" para evitar que la falta de soporte de la tarjeta gráfica de una determinada característica (existen diferentes fórmulas para niebla exponencial, lineal,etc.) impida su uso:

```
g_pd3dDevice->SetRenderState (D3DRS_FOGVERTEXMODE,D3DFOG_NONE);
```

Establecemos el color de la niebla (se desecha el canal alpha mediante la llamada XRGB):

```
g_pd3dDevice->SetRenderState (D3DRS_FOGCOLOR, D3DCOLOR_XRGB (255, 255,
255));
```

Como ya se ha expuesto, la mezcla de colores se realiza en función de la distancia al punto de vista. Existen varias fórmulas de cálculo, pero las principales son las de distancia exponencial y las de distancia lineal. En la niebla de distancia exponencial, a medida que aumenta la distancia la densidad

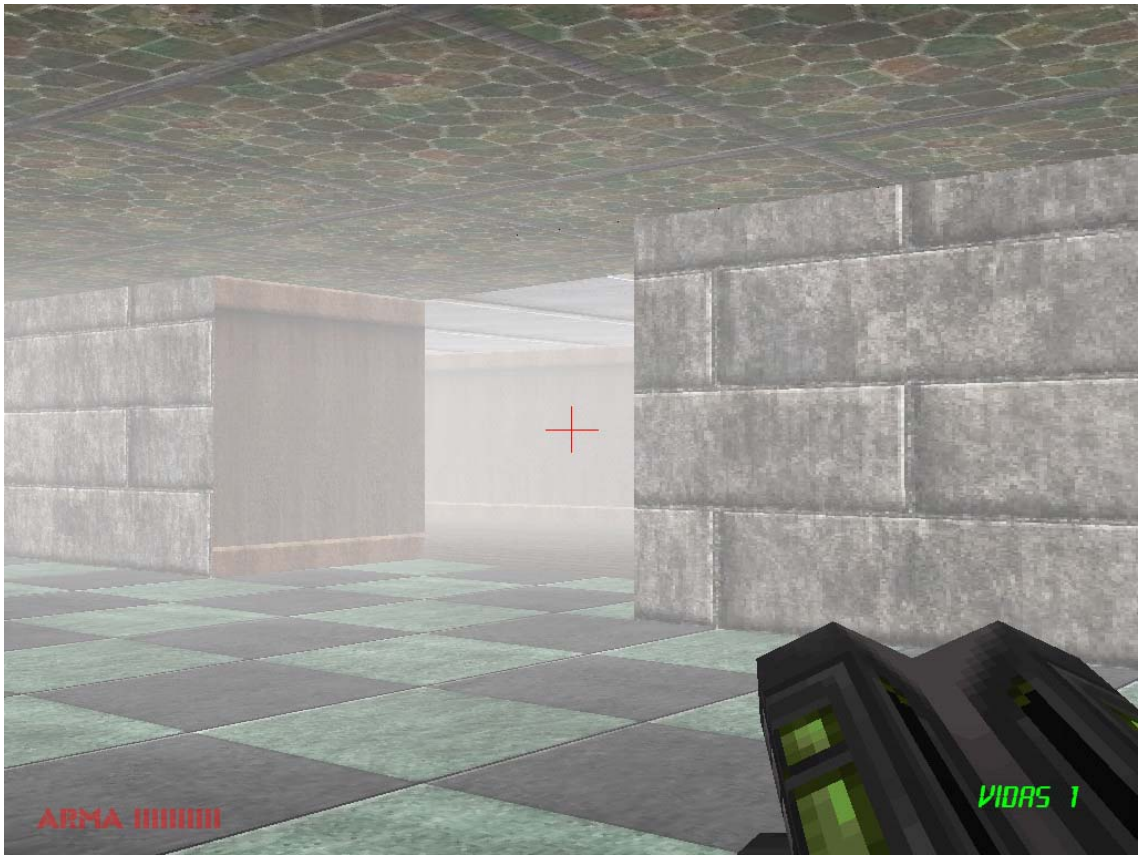
de las partículas aumenta de modo exponencial, resultando un efecto de fina niebla cercana al espectador, mientras que resulta muy saturada en los puntos lejanos (en principio la densidad se acumula hasta el infinito). En cambio, la niebla de distancia lineal tiene puntos concretos de comienzo de final, en este intervalo la densidad de la niebla es homogénea, de forma que la relación entre distancia y saturación percibida de la niebla es más uniforme. En nuestro caso hemos definido una distancia exponencial con un valor de densidad (obtenido empíricamente) de 0.05.

```
g_pd3dDevice->SetRenderState (D3DRS_FOGTABLEMODE,D3DFOG_EXP);

float nieblaValor = 0.05f;

g_pd3dDevice->SetRenderState
                  (D3DRS_FOGDENSITY,*((DWORD*)&nieblaValor));
```

Aquí acabarían los pasos necesarios, no obstante con objeto de poder comparar el efecto de niebla lineal, hemos definido también las sentencias necesarias para este otro método (hemos probado en el intervalo de 0.3 a 30):

```
// ejemplo de niebla de distancia lineal

g_pd3dDevice->SetRenderState (D3DRS_FOGTABLEMODE,D3DFOG_LINEAR);

float nieblaValor = 0.3f;

g_pd3dDevice->SetRenderState
                  (D3DRS_FOGSTART,*((DWORD *) &nieblaValor));

nieblaValor = 30.0f;

g_pd3dDevice->SetRenderState (D3DRS_FOGEND,*((DWORD *) &nieblaValor));
```

La inclusión del efecto de niebla se observa en:



Mejora de Sonidos e Inclusión de Música

Ya en la entrega anterior habíamos incluido el sonido de una forma primitiva, ya que podíamos hacer sonar un archivo de sonido (en nuestro caso era de formato de onda ".wav").

Sin embargo ya se advertía que tenía una gran limitación, ya que no podíamos reproducir dos sonidos a la vez sin que se superpusieran, ya que no se mezclaban los sonidos, simplemente dejaba el primero de sonar, para empezar el segundo sonido.

La solución tomada en esta versión no es más que un parche para no meternos muy a fondo con el tema, ya que el tiempo de proyecto se nos está acabando. Así la solución que hemos elegido ha sido la de crear nuevas estructuras para contener a otros sonidos, y de esta forma podemos hacer sonar hasta tres sonidos a la vez mientras los mezcla la tarjeta de sonido en la salida.

Así en la versión actual podemos encontrar tres sonidos que pueden darse a la vez, el primero es de tipo midi, y los dos restantes de tipo "wav".

La ejecución de archivos "wav" ya fue explicada, de todas formas ha habido ligeros cambios en la forma de hacerlos sonar, con ciertas mejoras, así que voy explicar como funcionan los métodos relacionados con la reproducción del fichero midi.

Implementación de las mejoras

Lo primero que vamos a necesitar con las estructuras necesarias para la ejecución del sonido:

```
IDirectMusicLoader8*      g_pCargador                    = NULL;
IDirectMusicPerformance8* g_pEjecutor                    = NULL;
IDirectMusicSegment8*     g_pSegmento                    = NULL;
```

Así, usaremos un cargador, un ejecutor y el segmento que vamos a hacer sonar.

Lo siguiente va a ser inicializar estas estructuras que hemos creado, ya que ahora mismo apuntan a null:

```
CoInitialize(NULL); //Esto siempre se necesita poner

CoCreateInstance((REFCLSID)CLSID_DirectMusicLoader, NULL,
                 CLSCTX_INPROC, (REFIID)IID_IDirectMusicLoader8,
                 (void**)&g_pCargador);
```

```
CoCreateInstance((REFCLSID)CLSID_DirectMusicPerformance, NULL,
                 CLSCTX_INPROC, (REFIID)IID_IDirectMusicPerformance8,
                 (void**)&g_pEjecutor );
```

Una vez creado el ejecutor y el cargador, inicializamos el ejecutor:

```
g_pEjecutor->Init (NULL, // direccion al objeto DirectMusic

                                                                NULL,
// direccion al objeto DirectSOund

                                                                NULL);
// window handle
```

Luego añadimos un puerto de la tarjeta de sonido al ejecutor:

```
g_pEjecutor->AddPort (NULL)
```

Luego también hemos fijado el directorio de búsqueda de los archivos de sonido de la siguiente forma:

```
WCHAR ruta[MAX_PATH];
MultiByteToWideChar( CP_ACP, 0, ".\\audio\\", -1,



ruta, MAX_PATH );

g_pCargador->SetSearchDirectory(

GUID_DirectMusicAllTypes,

                                                                ruta,









FALSE);
```

Ahora ya hemos hecho todas la inicializaciones necesarias, excepto el segmento de sonido que queremos que suene, así que lo vamos a cargar de la siguiente forma:

```
// Este objeto sirve para identificar lo que vamos a cargar
DMUS_OBJECTDESC obj_desc;
// String en unicode q contiene el nombre del fichero
```

```
WCHAR w_archivo[_MAX_PATH];

// Borramos la memoria del descriptor
ZeroMemory (&obj_desc, sizeof (obj_desc));
obj_desc.dwSize = sizeof (obj_desc);
// Cambiamos el nombre del archivo a wide string
MULTI_TO_WIDE(w_archivo,archivo);
// Le damos el identificador de la clase CLSID_DirectMusicSegment
obj_desc.guidClass = CLSID_DirectMusicSegment;
// Asignamos el nombre del archivo
wcscpy (obj_desc.wszFileName,w_archivo);
// Asignamos los miembros del objeto que vamos a usar
obj_desc.dwValidData = DMUS_OBJ_CLASS | DMUS_OBJ_FILENAME;

// Aquí cargamos el segmento
g_pCargador->GetObject (&obj_desc,




(REFIID)IID_IDirectMusicSegment,



(LPVOID *) &g_pSegmento);

// Le indicamos el tipo de archivo que ha cargado
g_pSegmento->SetParam ((REFGUID)GUID_StandardMIDIFile,


                                                        -1,


                                                        0,


                                                        0,


                                                        (LPVOID
)g_pEjecutor);

// Cargamos los instrumentos necesarios para ejecutar el sonido
g_pSegmento->SetParam ((REFGUID)GUID_Download,1,0,0,
(LPVOID)g_pEjecutor);
```

Si todo lo anterior se ha realizado con éxito (nosotros lo comprobamos con los parámetros que devuelve indicando error o éxito), entonces tendremos bien cargado el segmento midi y ya estará listo para hacerlo sonar.

Para hacer sonar un segmento lo hacemos mediante el método PlaySegment:

```
g_pEjecutor->PlaySegment (g_pSegmento,

                                                              0,

                                                              0,

NULL);
```

Si en algún momento queremos hacer que pare este segmento (ya que los archivos midi son muy largos), usaremos él método Stop:

```
g_pEjecutor->Stop(NULL, NULL, 0,  0);
```

Para finalizar debemos liberar todos los objetos creados para hacer que suene nuestro archivo midi de la siguiente forma:

```
g_pEjecutor->CloseDown();

g_pCargador->Release();
g_pEjecutor->Release();
g_pSegmento->Release();

CoUninitialize();
```

Nuevos Movimientos del Protagonista

En un juego de este tipo son fundamentales ciertos movimientos, como el salto y la posibilidad de agacharse, que mejoran la jugabilidad.

Así, y dado que hemos mejorado el control de colisiones en versiones anteriores, hemos decidido implementar estos nuevos movimientos. De esta forma, ahora vamos a poder sortear de un salto al enemigo pequeño que ya tenemos puesto en el escenario, ya que las cajas minimales del objeto y la nuestra no se van a chocar mientras saltamos.

Para el caso de la función de agacharse es muy sencillo, ya que sólo tenemos que cambiar nuestro punto de vista de la cámara para hacerlo más bajo, y para levantarse lo contrario.

Sin embargo para el salto ya no es tan sencillo, ya que antes podíamos hacer la transición del agacharse nula sin problemas (mucho juegos lo hacen), pero ahora, al saltar y subir el punto de vista, tenemos que hacerlo gradualmente hasta un punto máximo, para luego volver a caer (simulando la fuerza de la gravedad).

Así cuando el protagonista salta hacemos lo siguiente:

```
this->saltando = true;
SYSTEMTIME tiempo;
GetLocalTime(&tiempo);
this->tiempoSalto = tiempo.wMinute*3600*1000 + tiempo.wSecond*1000 +
                        tiempo.wMilliseconds;
```

De esta forma activamos el salto asignándole un tiempo de salto.

Lo siguiente será ir modificando la posición de la cámara en función del tiempo para que simule la gravedad, y cuando llegue a la posición inicial que pare de saltar:

```
SYSTEMTIME tiempo;
GetLocalTime(&tiempo);
int milisegundos = tiempo.wMinute*3600*1000 + tiempo.wSecond*1000 +
            tiempo.wMilliseconds - this->tiempoSalto;
this->posicionDeLaCamara.y = 3.0f + (7.0f*milisegundos)/1000.0f -
            (0.5f*9.8f*milisegundos*milisegundos)/1000000.0f;
if (this->posicionDeLaCamara.y < 3.0f)
{
                                                                this-
>saltando = false;
```

```
                                                                this-
>posicionDeLaCamara.y = 3.0f;}
```

Mira Telescópica

En muchos juegos actuales podemos ver como entre las armas que dispone el jugador se encuentra un rifle de francotirador con mira telescópica.

Nosotros hemos querido incluir un rifle de este tipo para probar el uso de texturas con Alpha Channel y el efecto del Zoom.

El efecto del rifle de francotirador lo podemos ver en la siguiente captura de pantalla:

Y en la siguiente captura podemos ver una captura similar, desde el mismo punto, pero con más Zoom:

Implementación de la mira

Las ideas principales sobre lo que se basa esta arma de rifle de francotirador son, como ya hemos apuntado, la textura con Alpha Channel y el efecto Zoom.

Para el efecto Zoom, que tal vez parece lo más complicado de implementar, sólo hay que cambiar un parámetro de la matriz de perspectiva aplicada a la matriz de proyección principal.

Así en D3DXMatrixPerspectiveFovLH tenemos que cambiar el segundo parámetro (que tenemos inicialmente a 1.0f). Este campo es el que indica el Campo de Visión (en inglés Field Of View o FOV). De esta forma si reducimos el campo de visión vamos a conseguir un efecto de Zoom.

Por otro lado, para hacer el efecto del punto de mira con una cruz (como se ha podido ver en las pantallas anteriores), lo que haremos será poner una textura justo delante de nosotros que tenga Alpha Channel, y además tenerlo activado.

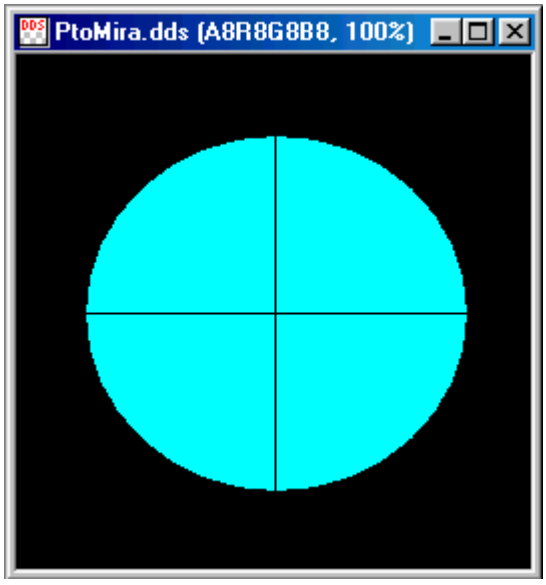
Para colocar siempre el punto de mira es necesario que esté a una distancia variable, para que la textura ocupe toda la pantalla sea cual sea la profundidad del Zoom. La fórmula matemática es la siguiente:

```
posicionPlano = posicionDeLaCamara +
((0.19f*tanf(0.35f))/tanf(zoom/2))*direccionNormalizada;
```

De esta forma calculamos la posición del plano donde se ubica la textura del punto de mira, donde "zoom" es la variable que lleva el grado de Zoom actual.

Para la textura, hemos dicho que hemos usado el Alpha Channel, que no es más que una textura en escala de grises asociada a la textura principal, en la que cada texel define el grado de transparencia según sea más claro o más oscuro en la escala de grises.

La textura principal nuestra es una textura totalmente negra, a la que le vamos a aplicar una segunda textura para definir la transparencia.



En esta captura (sacada del editor de texturas de DirectX 8.0) podemos observar como la graduación en azul celeste es el Alpha Channel, y va a se la parte que se vea transparente a la hora de ver la textura en el juego.

Por último para que funcione bien, tenemos que activar el Alpha Channel:

```
g_pd3dDevice->SetRenderState(D3DRS_ALPHABLENDENABLE,TRUE);
```

```
g_pd3dDevice->SetRenderState(D3DRS_SRCBLEND,D3DBLEND_SRCALPHA);
g_pd3dDevice->SetRenderState(D3DRS_DESTBLEND,D3DBLEND_INVSRCALPHA);
```

Esta es la forma que hemos usado nosotros para activar el Alpha Channel según nuestras necesidades.

Requisitos de ejecución

Es imprescindible la instalación del runtime de usuario Directx 8.0, de distribución gratuita y disponible, entre otros sitios, en www.microsoft.com/directx.

Plataformas de ejecución probadas

Pentium III 800 MHz

- Windows 2000 Professional Service Pack 2
- DirectX 8.0a
- ATI Rage Fury con chip Rage 128 PRO

Pentium Celeron 266 MHz

- Windows 98 Second Edition
- DirectX 8.0a
- ATI 3DCharger con chip 3D Rage II+

Pentium 4 1'4GHz

- Windows 2000 Professional Service Pack 2
- DirectX 8.0a
- nVIDIA GeForce 2 MX 200 con chip GeForce 2 MX

Amd K7 XP 1700+

- Windows 98
- Windows XP
- DirectX 8.0a
- Hercules Prophet 4500 con chip Kyro II

Pentium 4 1'7GHz

- Windows 2000 Professional Service Pack 2
- DirectX 8.0a
- ATI Rage 128 Ultra AGP 16 Mb con chip Rage 128 Pro II

Bibliografía

- **DirectX 8.0 Programmer's Reference.** Publicación electrónica : Microsoft Corporation, 1995-2000
- **MSDN Library Visual Studio 6.0. Sistema de programación Microsoft® Visual Studio™ 6.0.** Publicación electrónica : Microsoft Corporation, 1991-1998
- **3D game engine design : a practical approach to real-time computer graphics / David H. Eberly.** San Francisco [etc.]: Morgan Kaufmann, cop. 2001
- **3D games : real-time rendering and software technology / Alan Watt and Fabio Policarpo.** New York : ACM Press ; Harlow, England [etc.] : Addison-Wesley ; 2001
- **Game programming gems / edited by Mark DeLoura.** Rockland, Massacusetts : Charles River Media, cop. 2000
- **Game programming Gems 2 / Edited by Mark A. Deloura.** Hingham, Mass. : Charles River Media cop. 2001
- **Introduction to Computer Game Programming with DirectX 8.0.** Texas : Wordware, 2001
- **Learn computer game programming with DirectX 7.0 / Ian Parberry.** Texas : Wordware, 2000
- **Computer graphics : principles and practice / James D. Foley ... [et al.]** Reading, Mass. ; Wokingham : Addison-Wesley, c1990.
- **3D computer graphics / Alan Watt.** Harlow, England ; New York : Addison-Wesley, 2000
- **The C++ programming language / Bjarne Stroustrup.** Boston [etc.] : Addison-Wesley, 2001
- **Microsoft Visual C++ 6.0 : Manual de referencia / Chris H. Pappas, William H. Murray, III.** Madrid [etc.] : McGraw-Hill, cop. 1999
- **Object-oriented programming in C++ / Robert Lafore.** Indianapolis, Indiana : SAMS, cop. 1999

Documentación Web

- **http://www.gamedev.net**
- **http://www.flipcode.com**
- **http://nexe.gamedev.net/**
- **http://www.zen-x.net/DirectX8**
- **http://www.programmersheaven.com**
- **Foro de www.mr-gamemaker.com**