

Llib Planar

Versión 0.1

Created by neofar (2005)

Manual de usuario

URL <http://ps2dev.ofcode.com>
Contact neofar@ofcode.com

Introducción

Esta librería se ha desarrollado para trabajar sobre la base de **lalib** de **djhuevo** que se puede descargar en la dirección <http://ps2dev.ofcode.com>

El propósito de la misma es establecer unas funciones básicas para poder trabajar con texturas y para el dibujado de objetos en 2D y en 3D. Inicialmente se quería trabajar únicamente en 2D de ahí venía el nombre de **llibPlanar**, pero se ha extendido para trabajar de forma muy básica con objetos en 3D.

La librería se esta desarrollando de forma muy modular para que se pueda reaprovechar a medida que se optimicen sus partes.

Esta documentación explica de forma general como se ha estructurado la librería y los conceptos genéricos para usarla, pero no debe considerarse una documentación propia que explique el uso de cada función, para esto deberá consultar la documentación del doxygen.

1 - Componentes

- 1.1 - *LlibGS* > Funciones básicas para generar los packets del GS
- 1.2 - *LlibTex* > Carga texturas y las activa en el GS
- 1.3 - *Llib2D* > Dibuja Sprites 2D
- 1.4 - *Llib2DText* > Dibuja Texto
- 1.5 - *Llib3D* > Dibuja Objetos en 3D

2 - Pensando en Z

Cuando estamos trabajando en 2D no tenemos ningún problema en controlar el ZBuffer, nosotros mismos enviamos las primitivas al GS y sabemos en todo momento enviarlas en el orden apropiado. Si trabajamos en 3D tampoco hay problema ya que el propio calculo con las matrices nos va a colocar la Z como debe ser para que se dibuje correctamente. Pero el problema viene cuando queremos mezclar el 2D con el 3D y queremos controlar en todo momento que Sprites se están dibujando por encima del 3D y cuales por debajo.

3 - Mundo 3D

Dentro del apartado 3D tenemos mucho por explicar, todo el desarrollo de esta librería se basa en los ejemplos que vienen con la lib de djhuevo, y sobre esta librería se ha extendido el modelo .3dm desarrollado en estos ejemplos.

Modelo 3D > Simple Model Format [SMF] Versión 0.2

StripModel > Triangles vs TriangleStrip

Addons

En este apartado tenemos códigos que sin ser parte esencial de la librería nos van a ayudar durante el trabajo rutinario

llib3d.utils > Creación de objetos dinámicos

LibGS

Esta es una librería base para enviar los Tags al GS, es una extensión del archivo original "**common/gs.c**" proporcionado por la lib, podemos encontrar las funciones para:

Crear Tags del GS: Son las funciones sacadas directamente de la libreria original de djhuevo

```
__inline__ void addXYZ2(u_short x1, u_short y1, u_int z1);
__inline__ void addRGBAQ(u_char R, u_char G, u_char B, u_char A, float Q);
__inline__ void addST(float S, float T);
__inline__ void addUV(u_int U, u_int V);
__inline__ void addPRIM(u_char prim, u_char iip, u_char tme, u_char fge, u_char abe, u_char aa1, u_char
fst, u_char ctxt, u_char fix);
__inline__ void addSCISSOR_1(u_short SCAX0, u_short SCAX1, u_short SCAY0, u_short SCAY1);
__inline__ void addALPHA_1(u_char A, u_char B, u_char C, u_char D, u_char FIX);
__inline__ void addTEST_1(u_char ATE, u_char ATST, u_char AREF, u_char AFail, u_char DATE, u_char
DATM, u_char ZTE, u_char ZTST);
```

Funciones implementadas

```
__inline__ void gfxInitVideo(int resx,int resy);
__inline__ void gfxSyncV();
__inline__ void gfxSwapBuffers();
__inline__ void gfxRender();
__inline__ void gfxClear();
__inline__ void gfxSetAlphaBlending(int status);
__inline__ void gfxSetScissor(int x1,int y1,int x2,int y2);
__inline__ void gfxClearScissor();
```

Implementamos también las funciones para controlar el BufferZ, se explica su funcionamiento en profundidad en el apartado "**pensando en Z**", así que de momento no tenemos nada que decir

```
void setGSLayerZ(unsigned int v);
void layerZInit(int n);
void layerZStartAt(int n,int minvalue);
void layerZReset();
void layerZActive(int n);
unsigned int layerZGetActive();
unsigned int layerZGetMin(int n);
unsigned int layerZGetMax(int n);
```

Incluimos también algunas funciones simples para dibujar primitivas en 2D

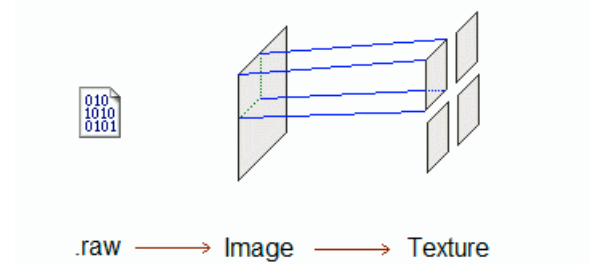
```
__inline__ void gfxSetColor(u_char r, u_char g, u_char b,u_char alpha);
__inline__ void gfxDrawPoint(int x,int y);
__inline__ void gfxDrawLine(int x1,int y1,int x2,int y2);
__inline__ void gfxDrawTriangle(int x1,int y1,int x2,int y2,int x3,int y3);
__inline__ void gfxDrawBox(int x1,int y1,int x2,int y2);
__inline__ void gfxDrawTriangleGoraund(int x1,int y1,u_char r1, u_char g1, u_char b1, int x2,int y2,u_char r2,
u_char g2, u_char b2, int x3,int y3,u_char r3, u_char g3, u_char b3);
__inline__ void gfxDrawBoxGoraund(int x1,int y1,u_char r1, u_char g1, u_char b1, int x2,int y2,u_char r2,
u_char g2, u_char b2, int x3,int y3,u_char r3, u_char g3, u_char b3, int x4,int y4,u_char r4, u_char g4, u_char
b4);
__inline__ void gfxDrawCircle(int x,int y,int z,int radius,int step);
void gfxDrawOval(int x,int y,int z,int radiusX,int radiusY,int step);
```

1.2 LlibTex

En esta librería vamos a reunir todas las funciones que se encargan de administrar las texturas (imágenes en general), subirlas a la VRAM y activarlas en el GS. se ha programado de forma muy estática y todavía no incluye la posibilidad de liberar una textura de memoria, ni creo que se vaya a implementar. Para entender como funcionan las texturas en la PS2 se recomienda mirar el ejemplo "TriangleTextured" que se incluye en los samples de la lib.

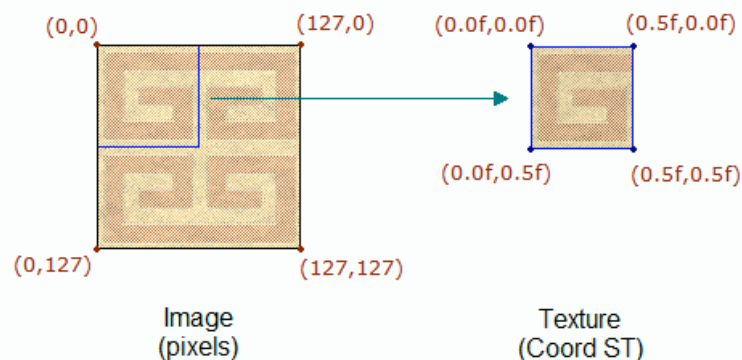
0.- Introducción

Al inicializar esta librería le daremos la dirección de memoria que tenemos libre en la Vram, la librería se encarga de ir cargando imágenes e ir metiéndolas en la Vram una detrás de otra, para la librería eso se considera Imágenes que viene a ser el propio archivo gráfico, a partir de ese archivo gráfico podemos recortar zonas que corresponde a las Texturas en si. De esta forma podremos subir una Imagen de 256x256 pixeles que tenga en su interior todas las texturas que usemos



Esta forma de trabajar con las texturas hace que por un lado se mas rápido acceder a distintas texturas, ya que todas se alojan en la misma imagen y podremos activar esa imagen en el GS con una única instrucción. Luego el recorte de la textura se encargará de coger la zona que le interesa.

Pero por otro lado puede dar muy mal resultado si tenemos las texturas muy mal repartidas en distintas imágenes, la idea seria que se activase la imagen[0] y luego se enviasen todos los objetos que usen esta imagen, luego cambiaríamos a la imagen[1] y enviaríamos los objetos que usasen esa textura. Esto para el proceso en 2D es muy sencillo de calcular, aunque para el renderizado en 3D se puede complicar un poco más.



Cuando cargamos una imagen en formato RAW la vamos a tratar en Pixeles, a partir de ahí cuando saquemos un trozo de la imagen para generar una textura vamos a tener las coordenadas en ST, no será necesario realizar ningún calculo, a la librería le decimos el trozo que queremos recortar en pixeles y ya se encargará de convertir eso a las coordenadas ST y guardarlas en memoria.

1.- Inicialización

```
void setVideoBaseDir(int w,int h,int ztest,int zpsm);
```

Este método es el que le dice a la librería que configuración de pantalla tenemos, de esta forma calcula cuanto ocupa nuestra pantalla en la VRAM y nos pone el PunteroBase a partir de esa posición, una vez hecho esto ya esta lista para empezar a subir imágenes.

En principio seria mucho más útil preguntarle esto a la lib, pero como todavía no está implementado lo tenemos que inicializar así.

```
// (ScreenWidth * ScreenHeight * BytesDel_PSM * ZBuffer) / 256  
// (640 * 480 * 4 * 3) / 256 = 14400 (pantalla 640x480 32 bits con zBuffer)
```

```
void texturePrepareArray(int NumImages, int NumTextures);
```

Como hemos explicado en la introducción esta librería no es dinámica, de momento no tiene una lista de punteros en su estructura, y desde un principio le debemos decir cuantas Imágenes vamos a leer, y cuantas Texturas vamos a tener en total para que pida la memoria necesaria para guardar esta información.

```
void textureFixDir();
```

Mediante este método lo que hacemos es fijar las texturas que ya tenemos cargadas para que no se liberen nunca (mientras no reiniciemos la librería). Esto es para leer en primer lugar aquellas texturas que serán comunes durante todo el programa, como puede ser por ejemplo la textura con el charset

```
texturePrepareArray(4, 4); // preparamos espacio para 4 imágenes y 4 texturas  
id = imageLoadRaw("Fuente_256.raw", 256, 256, LIB_GS_PSMCT32); // cargamos la imagen 0  
textureCreate(id,0,0,256,256); // y la textura 0  
textureFixDir(); // A partir de este momento, la imagen[0] esta protegida y no se libera al hacer un reset();
```

```
void textureReset();
```

Como hemos indicado en la introducción, esta librería no cuenta con una forma para liberar una textura en concreto, en su lugar tenemos este método que reinicia totalmente la carga de texturas, y devuelve el puntero de carga al inicio de la memoria libre.

2 - Cargar Imágenes y crear Texturas

```
int ImageLoadRaw(char *filename,int w,int h,int psm);
```

Este método carga una imagen en formato *raw*, en principio no se ha implementado ningún otro método para cargar una imagen, por eso en este mismo método le tenemos que decir cuanto ocupa la imagen en píxeles y el *Pixel Storage Method* que usa.

El método nos devuelve la posición que ocupa esta imagen dentro de nuestra estructura, no será necesario guardar estos valores ya que van a ser secuenciales.

```
imageLoadRaw("imagen.raw", 256, 256, LIB_GS_PSMCT32);
```

- Faltaría por implementar la carga de ficheros IIF (con una cabecera), con lo que bastaría darle únicamente la imagen
- En el directorio [/Tools/](#) de la librería se encuentra una utilidad para convertir archivos en formato TGA a .raw y a .iif el código fuente de esta utilidad se puede conseguir en ps2dev.ofcode.com

```
int textureCreate(unsigned int nImage,int x1,int y1,int x2,int y2);
int textureCreateST(unsigned int nImage,float x1,float y1,float x2,float y2);
```

Este método crea una textura a partir de una imagen, en los parámetros (x1-x2-y1-y2) se le especifica el cuadrado que recorta de la imagen para generar esta textura.

Supongamos que en el método anterior hemos cargado la imagen de 256x256 píxeles. Al leer una única imagen sabemos que se encuentra en el índice=0

```
textureCreate(idImagen, 0, 0,128,128);
textureCreateST(idImagen, 0.0f,0.0f,1.0f,1.0f);
```

Resulta bastante cómodo hacer el recorte trabajando en píxeles, y la propia librería hace la conversión a ST. Tenemos también la correspondiente función que trabaja con los valores en ST.

Será muy común que cuando estamos depurando trabajemos con imágenes de menor resolución para agilizar los tiempos de carga, en estas condiciones si recortamos la textura a partir del ST no deberemos modificar nada, ya que estas coordenadas ST son proporcionales a la resolución de la Imagen. Si estuviéramos trabajando con Píxeles tendríamos que estar ajustando estos valores constantemente.

```
int TextureUse(libGsTexEnv *base, int n);
```

Este método es el que activa la textura en el GS

```
gsPacketCount += TextureUse((libGsTexEnv *)&gsPacket[gsPacketCount], indexTexture);
```

Esta librería trabaja sobre la base de *lilibgs*, así que podremos liberarnos luego del parámetro *libGsTexEnv*

3 - Propiedades de la imagen: Tfx & Filter

Si miramos el header de esta librería veremos enseguida que tenemos 2 estructuras `libtex_Image` y `libtex_Texture`, accediendo directamente a estas estructuras podemos modificar cualquier parametro de estas. Por ejemplo para activar el filter de una imagen cargada bastaria con esto:

```
int IndexImage = ImageLoadRaw("image_64.raw", 64, 64, LIB_GS_PSMCT32);
```

```
void imageSetTfx(unsigned int nImage,int tfx);  
// #define LIB_TFX_MODULATE 0  
// #define LIB_TFX_DECAL 1  
// #define LIB_TFX_HIGHLIGHT 2  
// #define LIB_TFX_GHLIGHT2 3
```

```
void imageSetFilter(unsigned int nImage,int filter);  
// #define LIB_FLT_NEAREST 0  
// #define LIB_FLT_LINEAR 1  
// #define LIB_FLT_NEAREST_MIPMAP_NEAREST 2  
// #define LIB_FLT_NEAREST_MIPMAP_LENEAR 3  
// #define LIB_FLT_LINEAR_MIPMAP_NEAREST 4  
// #define LIB_FLT_LINEAR_MIPMAP_LINEAR 5
```

* Esta configuracion afecta a la imagen en sí, y por tanto a todas las texturas que estén en esta imagen

Estas funciones se tendrán que implementar para evitar que alguien acceda a índices incorrectos de la estructura, se supone que al final la estructura `texInfo` no debería ser accesible desde fuera de la librería para evitar manipulaciones.

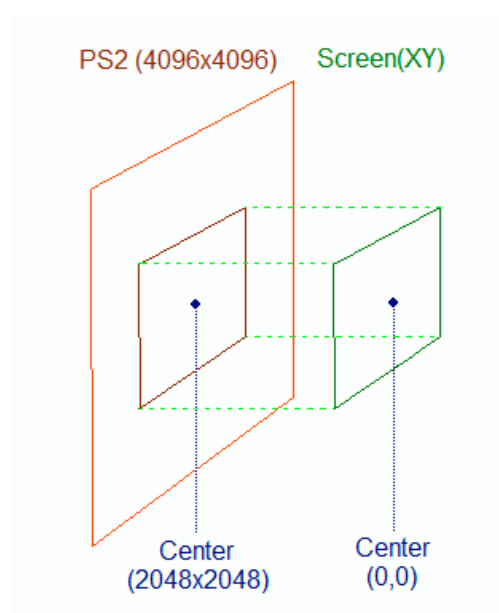
1.3 Llib2D

La finalidad de esta librería es únicamente la de dibujar sprites en 2D, a pesar de usar el termino Sprite se debe entender que esta librería únicamente se encarga del dibujado, no controla en ningún momento lo que corresponde a un sprite real (concepto de objeto).

Se le ha añadido la posibilidad de pivotar puntos dentro del Sprite en 2D para calcular las transformaciones que se le aplican al sprite también sobre estos puntos, de esta forma podremos localizar un píxel del sprite original una vez renderizado.

0 - Introducción

Para unificar criterios entre en motor 2D y el motor 3D vamos a trabajar con un centro de coordenadas (0,0) situado en el centro de nuestra pantalla. Para el GS nuestra pantalla mide 4096x4096 pixeles, de esa forma nuestro centro lógico estará en 2048x2048 que corresponde con el centro de nuestra pantalla (independientemente de la resolución que tengamos puesta).



1- Sprites

```
void spriteSetTexture(int indexTexture);
```

Este método setea la textura que vamos a usar para el próximo Sprite a renderizar, luego el método `Render()` será el que la envíe al GS.

```
void spriteReset();
```

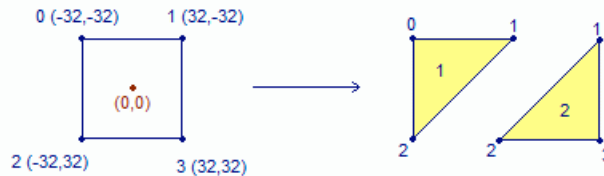
Reinicia todos los valores del Sprite

```
spriteSetSize(0,0);
spriteSetColor(0x80,0x80,0x80);
spriteSetAlpha(0xFF);
```

```
void spriteSetSize(int x, int y);
```

Establece el tamaño del sprite que vamos a dibujar. Este método es el primero que debemos usar y lo que hace es directamente colocar las 4 coordenadas XY a esos valores. A partir de esas 4 coordenadas el sistema acabará por dibujar los 2 triángulos que forman este Sprite tal como se indica en la imagen. Esta forma de repartir las 4 coordenadas nos va a facilitar el dibujado al enviar un TriStrip al GS

Recuerda que el sprite va a aparecer centrado en pantalla (supongamos un sprite de 64x64 pixeles)



```
int midx = x/2;
int midy = y/2;

sprite.x[0] = -midx; sprite.y[0] = -midy;
sprite.x[1] = +midx; sprite.y[1] = -midy;
sprite.x[2] = -midx; sprite.y[2] = +midy;
sprite.x[3] = +midx; sprite.y[3] = +midy;
```

```
void spriteTranslate(int x,int y);
void spriteScale(float x,float y);
void spriteRotate(float z);
void spriteRotateHex(unsigned char z);
```

Estos métodos trabajan directamente con las coordenadas XY que se han generado al darle tamaño al Sprite, no necesitan ninguna explicación, solamente indicar que estamos trabajando en 2D, y el método `Rotate` tiene un único parámetro porque siempre rotamos los sprites en el eje Z.

Estos 3 métodos se aplican directamente en el orden en el que se ejecutan, es decir no es lo mismo un `Rotate+Translate` que un `Translate+Rotate`.

Por defecto el Sprite ha sido centrado en (0,0) al crearse, por lo que viene perfecto para rotarlo directamente sobre su centro, si queremos rotarlo sobre otro punto, tendremos que hacer el `translate` antes que nada.

Rotate versus RotateHex: el método `Rotate()` recibe como parámetro el ángulo en radianes para rotar el Sprite, pero en determinados casos vamos a necesitar rotar un ángulo determinado y a veces moverse con radianes resulta bastante pesado, así que tenemos el método `RotateHex` que recibe como parámetro un **unsigned char** y corresponder a un vuelta completa de 256 pasos. Es

decir si queremos rotar el objeto 90° corresponde de $256/4 = 64$, del mismo modo media vuelta es 128. Este método lo único que hace es convertir el valor que le damos a radianes y llamar a la función Rotate().

```
void spriteSetColor(u_char r,u_char g,u_char b);
void spriteSetAlpha(u_char a);
```

Establecemos el color y la transparencia del sprite que vamos a dibujar. por defecto el color debe ser (0x80,0x80,0x80) y la transparencia como 0xFF.

El método Reset() ya se encarga de setear estos valores por defecto.

```
void spriteRender();
```

Este método es el que, por fin, envía los *Tags* al *GS* en este orden:

```
// Setea la Textura
// envia addPRIM
// addRGBAQ
// addST (coordenada 0)
// addXYZ2(coordenada 0)
// [] seguimos con los ST y XYZ de las otras coordenadas
```

Esta librería tiene una variable `spriteLayerZ` que corresponde al plano Z donde esta enviando este sprite, cada vez que dibuja un sprite nuevo lo incrementa, para que el próximo Sprite se dibuje encima del anterior y pueda evaluar correctamente el alpha.

Todavía no se ha programado ningún método para controlar esta variable `spriteLayerZ`, de momento se ha añadido un método `spriteResetZ()` para respetar esta variable.

2 - Pivots

```
lLib2D_Pivot *spritePivotCreate(int n);
int spriteAddPivot(lLib2D_Pivot *pivot,int x,int y);
```

Mediante este método generaremos una estructura `lLib2D_Pivot`, al inicializarla le decimos el numero de puntos que queremos controlar, y mediante el método `spriteAddPivot()` vamos añadiendo coordenadas a la estructura. Este método nos devuelve el índice que le da a este punto dentro de la estructura.

```
lLib2D_Pivot *mypivot;

mypivot = spritePivotCreate(2); // -> creamos la estructura para 2 pivotes
spriteAddPivot(mypivot,10,10); // -> id = 0
spriteAddPivot(mypivot,12,16); // -> id = 1
```

```
void spriteSetPivot(lLib2D_Pivot *pivotInfo);
```

```
int spriteGetPivotX(lLib2D_Pivot *pivot,int i);
int spriteGetPivotY(lLib2D_Pivot *pivot,int i);
```

1.4 Llib2DText

0 - Introducción

Aquí vamos a reunir las funciones típicas para dibujar texto en pantalla, a pesar del nombre de la librería ninguna función depende de Llib2D esta librería se encontraría a un mismo nivel que el 2D ya que prácticamente es la misma funcionalidad, pero con funciones optimizadas para dibujar Sprites en lugar de triángulos.



1 – Inicialización

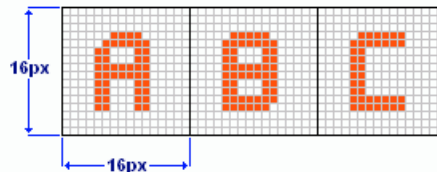
```
void textSetFont(unsigned int n, int nFonts, int charSizeX, int charSizeY);
```

Mediante esta función le pasamos a la librería la referencia de la textura donde se encuentra la fuente, y le indicamos en esta textura:

nFonts -> numero de fuentes (bloques verticales)

charSizeX -> Tamaño horizontal en pixeles del carácter

charSizeY -> Tamaño vertical en pixeles del carácter



```
imageLoadRaw("Font_256.raw", 256, 256, LIB_GS_PSMCT32); // cargamos la imagen
textureCreate(0,0,0,256,256); // generamos la textura
textSetFont(0, 1, 16, 16); // 1 unica Fuente, con un caracter de 16x16 pixels
```

```
void textSetFontActive(int n);
```

Establecemos la fuente activa dentro de la textura, por ahora podemos enviar un máximo de 8 fuentes en un mismo archivo gráfico, este tope se ha definido en el código.. tampoco creo que se vaya a necesitar tantas fuentes, y a nivel de pruebas únicamente he usado 2 fuentes, a partir de ahí todo es desconocido.

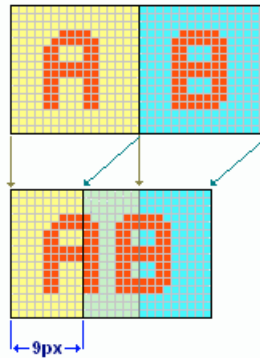
A la hora de renderizar cada caracter, el sistema busca las coordenadas XY dentro de la fuente, y luego aplica un desplazamiento vertical en función de la fuente activa:

```
texto += textInfo.fontActive * (textInfo.charSizeY * 8);
```

```
void textSetPadding(int index,int padding);
```

Establecemos el padding para cada una de las fuentes.

Tenemos que cada carácter dentro de la fuente se encuentra en una caja de X pixels (definidos al iniciar la fuente), pero a la hora de ir concatenando los caracteres para escribir el texto es muy posible que esa separación en pixeles sea muy grande. Mediante esta función indicamos que separación debemos emplear al enviar los caracteres.



```
textSetFont(0, 1, 16, 16); // Activamos una fuente de 16x16 pixels
textSetPadding(0,9); // Fuente 0 -> padding de 9
```

```
void textSetColor(u_char r,u_char g, u_char b);
void textSetAlpha(u_char a);
```

Mediante esta función establecemos el color del texto, para ello hay que tener en cuenta que la fuente original debe estar en color blanco, ya que esta función trabaja sobre el tint. La otra función establece el alpha.

```
void textSize(float factor);
```

Mediante esta función podremos aplicar una escala sobre la fuente, por defecto la fuente se va a imprimir a un tamaño igual a los pixeles que definen el tamaño del carácter (CharSizeX y CharSizeY respectivamente).

```
textSize(0.5f); // una fuente a la mitad de tamaño
textSize(2.0f); // el doble de tamaño
```

```
int textGetHeight();
int textGetWidth(char *str);
```

Estos dos métodos nos devuelve el alto de la fuente y el ancho de String en concreto. En apariencia estos valores son muy sencillos de calcular, pero estas funciones tienen en cuenta el factor de escalado de la fuente y el padding que tiene el charset activo.

```
void textPrintXY(int x, int y, char *str);
void textPrintCenter(int y, char *str);
```

descripcion

1.5 Llib3D

Criterios iniciales

Partimos de que el usuario conoce los términos básicos para hacer un render en 3D, en esta documentación no se va a explicar como se dibuja algo en 3D ni nada nuevo que no sepamos sobre los vectores y matrices, no vamos a entrar a explicar como se calcula la incidencia de la luz sobre un triángulo. La librería ya hace todo esto de forma automática, solo deberemos conocer los conceptos de lo que estamos haciendo.

La parte en 3D, a pesar de estar tratada de forma muy simple, es la más extensa de todas y todo apunta a que va a seguir creciendo. En la versión actual estamos trabajando con modelos 3DM en su versión 2. Pero tenemos planeado a corto plazo evolucionar un paso más este modelo y optimizar la maya a TriStrips, aunque a fin de cuentas esto será un cambio que afectará únicamente a la forma interna de dibujar el objeto y no se apreciará ningún cambio desde el uso de la librería.

Dentro del campo 3D vamos a manipular estos 3 conceptos

Camera

0 - Introducción

1 - LayerZ

2 - Creamdo y activando camaras

Light

0 - Introduccion

1.I - Lights

1.II - Ambient & Diffuse

1.III- Specular

2 - Coding

Objects

1.5 Llib3D - Camaras

En esta primera versión las posibilidades de la cámara son mínimas, todavía no tenemos ninguna libertad para posicionarnos en el mundo. La librería en general esta diseñada para trabajar con una cámara que siempre esta mirando en la dirección $\langle 0,0,-1 \rangle$ y donde solo podremos indicar la profundidad del plano.

Creando Cámaras

```
ObjectCamera *cameraCreateZ(int factorZ);
```

Crea una cámara especificando la profundidad de la misma en el plano Z.

```
ObjectCamera *camara;  
camara = cameraCreateZ(-200); // la profundidad de la camara
```

```
ObjectCamera *cameraCreateForLayer(int factorZ,int layer);
```

Crea una cámara especificando una profundidad, y diciéndole en que layerZ queremos que nos renderice los objetos

```
layerZInit(3); // Vamos a trabajar con 3 Capas *  
  
layerZStartAt(0,0); // capa 0 -> (0x00,0xFF)  
layerZStartAt(1,0xFF); // capa 1 -> (0xFF, 0xFFFF - 0xFF)  
layerZStartAt(2,0xFFFF - 0xFF); // capa 2 -> ( 0xFFFF - 0xFF, 0xFFFF)  
  
// ahora podemos crear la camara, diciendole cual es la capa donde debe renderizar los objetos  
  
ObjectCamera *camara;  
camara = cameraCreateForLayer(-200,1);  
  
// la camara ya prepara la matriz para que renderice los objetos en el bufferZ  
// que especificamos en esa Layer,  
// de esa forma tendremos una Layer 0 que estara SIEMPRE por debajo del 3D  
// y unca Layer 2 que estará SIEMPRE por encima del 3D
```

* El concepto de LayerZ se explica en profundidad en el apartad 2 “Penzando en Z”

```
ObjectCamera *cameraCreateZRange(int factorZ,unsigned minz, unsigned int maxz);
```

Crea la cámara especificándole el rango mínimo y máximo en el bufferZ donde va a renderizar los objetos. Viene a ser lo mismo que el método anterior con la diferencia que aquí no necesitamos crear las capas en el bufferZ.

```
ObjectCamera *camara;  
camara = cameraCreateForLayer(-200,256,1024);
```

```
void cameraSetActive(ObjectCamera *camara);
```

Activa una cámara. Desde este momento cualquier objeto en 3D que rendericemos se dibujara según los parámetros de esta cámara.

1.5 Llib3D - Lights

Desde el principio parecía que el campo de las luces era donde se podía explorar o implementar nuevas cosas, para una luz estaba claro que solo íbamos a necesitar un vector dirección y un color para calcular la incidencia con cada polígono, pero a base de experimentar y probar cosas nuevas hemos llegado a esto. He descartado por el momento las luces puntuales (por el coste de calcular la dirección para cada vértice) y de momento solo tenemos luces direccionales.

1.I - Lights

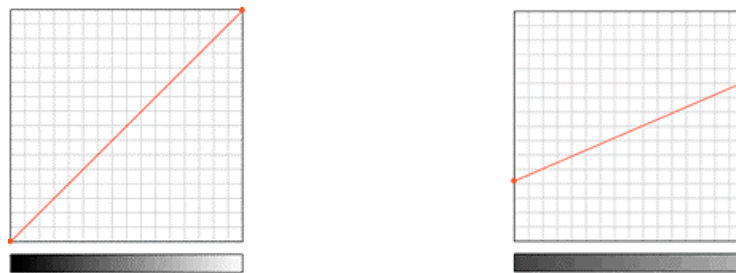
Para calcular la incidencia de la luz sobre un polígono nos bastará con hacer el producto vectorial de la luz con la normal del polígono (o normal del vértice según el tipo de sombreado que queramos utilizar), este producto vectorial nos tiene que dar valores entre 1 y -1.

Los valores que estén por debajo de 0 (los negativos) los debemos ignorar ya que matemáticamente esas normales no están mirando hacia la luz, es decir el polígono no recibe nada de luz, pero para los valores positivos deberemos multiplicar ese valor por cada componente RGB de la luz.

Esto nos vendría a dar una escala de color lineal, pero con unas diferencias de color que van a variar desde el negro absoluto al blanco... algo que en la realidad no nos interesaría conseguir, visualmente tendremos muchísima diferencia de color en un objeto (además que para el color los valores por encima del 0x80 ya son exagerados).

1.II - Ambient & Diffuse

En un calculo normal nuestra escala de color debería ser como se indica en la imagen (1), con valores que irían desde el negro al blanco, pero vamos a definir un color mínimo (llamado ambient) y un color máximo (llamado diffuse). Nuestra escala de color se va a establecer dentro de esa rampa de color como se muestra en la imagen:

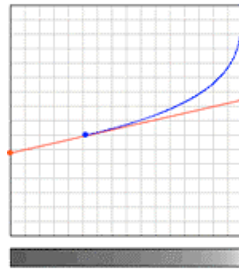


- (1) Escala de color de negro a blanco
- (2) Escala de color dando un valor mínimo y un valor máximo (ambient y diffuse)

1.III - Specular

Matemáticamente nos va a resultar muy sencillo calcular la rampa de color si solo tuviéramos valores de ambient y diffuse, es una simple formula lineal, pero nos interesa añadir un valor specular.

Mediante este valor podremos conseguir que la luz incida de forma mas directa en la parte mas iluminada, ya no tenemos una rampa lineal, y empezamos a complicar un poco el calculo (al final esta rampa de color va a estar precalculada en un array, así que no tendremos ningún problema).



Esta forma de calcular la luz ya la estuve probando en el sample coolShading incluido con la snapshot 0.2 de la lib de djhuevo, y el efecto que se conseguía era bastante bueno, además en ese ejemplo no calculaba la curva como ahora, simplemente aumentaba la pendiente de la recta en los últimos puntos.

Ahora que sabemos que vamos a calcular la rampa de color y la vamos a mantener de forma estática en un array ya no hay nada que nos pare a calcular cualquier tipo de luz, y lo primero que vamos a añadir es un valor de specular pero en la parte 0... que falta de nombre la vamos a llamar contraSpecular, y vamos a conseguir luces de este estilo:



...esto va tomando forma...

2 - Coding

En mi primera versión de luces, era suficiente con hacer esto:

```
ObjectLight *luz = lightCreate(0,0,10); // Creamos la luz indicando las coordenadas donde se encuentra
lightSetAmbient(luz,0.2f,0.2f,0.2f,1.0f); // Color ambient (R,G,B, intensidad)
lightSetDiffuse(luz,0.8f,0.8f,0.8f,1.0f); // Color diffuse (R,G,B intensidad)
```

Y con esto se definían los valores que toman la luz para la formula lineal, pero ahora se complica un poco si queremos especificar los valores de specular.

Para crear la curva de specular necesitamos decirle, en primer lugar, a que color tendemos en el specular:

```
lightSetSpecular(luz,1.0f,1.0f,1.0f,1.0f); // Color Specular (R,G,B, intensidad)
```

Y a continuación en que punto (entre 0.0f y 0.1f respecto de la línea ambient-diffuse) comienza la curva del specular, aunque en los gráficos se ha dibujado en color gris (con el R=G=B) esta curva puede empezar en un punto distinto para cada canal RGB.

```
lightSetSpecularNode(luz,0.8f,0.8f,0.6f); // Punto donde comienza el Specular (canal R,G,B)
```

Del mismo modo tenemos las funciones para el ContraSpecular

```
lightSetCSpecular(luz,0.0f,0.0f,0.0f,1.0f); // Color CSpecular (R,G,B, intensidad)
lightSetCSpecularNode(luz,0.2f,0.2f,0.2f); // Punto donde TERMINA el CSpecular (canal R,G,B)
```

Finalmente, una vez que hemos establecido todos los parámetros RGB debemos llamar a un método que calcula la rampa de color:

```
lightCreateColorRamp(luz);
```


3 - More coding

Bien, crear luces es ahora algo mas complicado por los metodos que nos lleva, asi que para agilizar un poco este proceso vamos a guardar las luces en un archivo para poder leerlas de forma comoda.

```
ObjectLight *luz;  
  
luz = LoadLight("File.light"); // archivo con la definicion RGB  
lightCreateColorRamp(luz); // Este metodo es el que calcula la rampa de color
```

Y esto es todo, en el directorio /Tools/ se incluye una utilidad para compilar estos archivos.light a partir de un archivo txt con los valores de la luz

1.5 Llib3D – Objects

3D... solo son 2 letras, pero hay tantos conceptos por explicar dentro de este campo que vamos a cortar por lo sano y centrarnos en lo que hacemos, sin explicar como lo hacemos.

1 - Cargando modelos y creando instancias

Empezamos por aclarar un concepto genérico dentro del campo en 3D, por un lado debemos tener una geometría original, el objeto base que hemos leído donde se guardan sus vértices y todos sus polígonos (indistintamente de la forma en que este guardado este objeto), y por otro lado tenemos una instancia de ese objeto que es la que vamos a manipular constantemente aplicando las transformaciones para dibujarla a nuestro antojo. En nuestra Librería estos dos elementos se llaman:

Model3DM -> Geometría original cargada de un archivo 3DM.

Object3D -> Objeto 3D manipulable (es una instancia de un Model3DM y siempre hace referencia a la geometría de su generador).

```
Model3DM *LoadModel3DM(char * filename)
```

Mediante este método vamos a leer un modelo en formato 3DM:

```
Model3DM *modelo3DM;  
modelo3DM = LoadModel3DM("models/hexagon.3dm");
```

En el directorio **/Tools/** de la librería se encuentra una utilidad para convertir archivos en formato ASE a .3dm , el código fuente de esta utilidad se puede conseguir en ps2dev.ofcode.com

```
void model3DMScale(Model3DM *obj,float x,float y,float z)  
void model3DMScaleTo(Model3DM *modelo3DM,float x,float y,float z) (sin implementar)
```

Sobre un objeto 3DM no tenemos nada que hacer, simplemente es una geometría inicial de un objeto, el objeto maestro a partir del cual vamos a trabajar.

Lo que si es posible es que este objeto tenga unas proporciones que en principio nos sea difícil de tratar, aunque posteriormente será posible escalar a nuestro antojo el objeto en 3D, al leer distintos modelos puede darse el caso que necesitemos escalar la geometría original en lugar de ir arrastrando esa transformación a cada modelo que la implemente.

Nota: el método *model3DMScaleTo* está todavía sin implementar.

```
Object3D *ObjectCreate(Model3DM *modelo3DM)
```

Mediante este método crearemos una instancia de un objeto original 3DM:

```
Object3D *o1, *o2;  
o1 = ObjectCreate(modelo3DM); // Creamos una instancia del objeto  
o2 = ObjectCreate(modelo3DM); // y otra mas
```

Para crear un objeto basta con decirle cual es el Modelo3DM del que vamos a tomar la geometría, en este momento la librería dimensiona el objeto para poder renderizar esa geometría.

Nota importante: El Objeto3D guarda una referencia al modelo que lo genera, para poder tomar de ahí los vértices y toda la geometría. Por lo que un Modelo3DM NO Debe liberarse de memoria mientras se tenga algún objeto referenciándolo. La librería no controla esos fallos.

2 - Render modes + Color + Textures + Lights

```
void objectSetRenderMode(Object3D *obj, unsigned int mode)
void objectSetRenderColor(Object3D *obj, unsigned int mode)
void objectSetRenderMap(Object3D *obj, unsigned int mode)
```

Estos 3 métodos nos definirán el modo de representación para el Objeto:

```
objectSetRenderMode
    RENDER_POINT
    RENDER_WIREFRAME
    RENDER_FLAT
    RENDER_GORAUD
    RENDER_GORAUD3
```

```
objectSetRenderColor
    RENDER_COLOR
    RENDER_TEXTURE
```

```
objectSetRenderMap
    RENDER_ENV
```

```
void objectSetColor(Object3D *obj, float r, float g, float b)
void objectSetTexture(Object3D *obj, int textureIndex)
void objectSetTextureMap(Object3D *obj, int textureIndex)
```

Mediante estos métodos vamos a definir el color del Objeto o la textura que se le aplica, estos métodos se deberán usar en función del modo de Render o modo de Color que le estemos aplicando:

```
objectSetColor
    Valor RGB del color del Objeto. Este color se usa siempre en los metodos de render Point y Wireframe. Y es el color que tiene el modelo en el resto de modos de render siempre que se active objectSetRenderColor(RENDER_COLOR)
```

```
objectSetTexture
    Textura del modelo. Unicamente se aplica en los modos FLAT y GORAUD, y siempre que se active objectSetRenderColor(RENDER_TEXTURE)
```

```
objectSetTextureMap
    Textura del Environment Map, se aplica unicamente si se activa el objectSetRenderMap(RENDER_ENV)
```

```
void objectSetLight(Object3D *obj, int index, ObjectLight *luz)
```

Mediante este método enlazamos una luz a un objeto, por defecto la librería tiene un tope de 3 luces por objeto, en este método debemos indicar el índice de la luz.

Si queremos eliminar una luz del objeto bastará con asignarle un NULL.

3 - Moviendo objetos

```
void objectReset(Object3D *obj);
```

Este método es el que se encarga de resetear la matriz de transformación del objeto

```
void objectTranslate(Object3D *obj,float x,float y,float z)  
void objectRotate(Object3D *obj,float x,float y,float z)  
void objectRotateHex(Object3D *obj,u_char x,u_char y,u_char z)
```

Traslada y Rota el objeto en el Mundo en 3D, estas operaciones se aplican directamente sobre la matriz de transformación por lo que influye en orden en que se ejecutan:

Del mismo modo en llib2D se incluye un método para rotar el objeto en un ángulo pasado en Hexadecimal, para entender esta función consulta aquel apartado, que no es cuestión de explicar lo mismo 2 veces.

```
void objectTranslate2D(Object3D *obj,float x,float y)  
void objectScale2D(Object3D *obj,float x,float y)
```

Estos métodos tiene un poco mas de explicación, se llaman metodos2D y tienen 2 parametros XY porque se aplican a nivel de transformación en pantalla, es decir:

Translate 2D: traslada el render del objeto para moverlo a la coordenada determinada de pantalla, nuestra cámara en 3D va a renderizar el objeto en el centro de pantalla, a menos que lo hayamos trasladado en el mundo 3D

Scale2D: escala la proyección de los vértices en pantalla, no escala en ningún momento el objeto original ni los vértices del modelo.

Nota: Estas 2 operaciones se calculan en la matriz CameraToScreen para luego generar la matriz final de transformación. Es la matriz de proyección quien aplica estas transformaciones al modelo.

4 - Accion !

```
void objectRender(Object3D *obj)
```

Necesita esto alguna explicación?

2 – Pensando en Z

0 - Introducción

Tenemos algo pendiente que no hemos explicado en el proceso 2D ni en el 3D, y es la forma en la que vamos a tratar el bufferZ.

Para nuestro trabajo en 2D tenemos una variable (que normalmente no deberemos tocar y es administrada desde llibGS) para controlar el plano Z, cada vez que dibujamos un sprite nuevo esa variable se va incrementando para que el siguiente sprite se dibuje por encima del anterior, de esta forma garantizamos que las transparencias se calculen de forma correcta.

En el trabajo en 3D ocurre algo similar, es la propia geometría del objeto y la posterior proyección de los vértices lo que ordena los polígonos en el bufferZ, en el mundo 3D no tenemos ningún control para cambiar el valor Z sobre el cual se dibuja el objeto, esa Z es calculada por la proyección y viene determinado por la cámara.

Pero el mayor problema lo íbamos a tener al mezclar elementos 2D con 3D. Queremos controlar el rango sobre el cual estamos dibujando los objetos en 3D para poder dibujar sprites por detrás o por encima, o incluso puede que nos interese tener elementos en 3D que estén en el fondo, siempre por detrás de elementos 3D principales.... sería bastante engorroso estar trabajando con las posiciones espaciales reales para evitar que un objeto del fondo empiece a solaparnos.

Un ejemplo muy sencillo de ver puede ser los valores de score de cualquier juego, estos texto son simples Sprites Planos que siempre estarán por encima de todo, debemos tener reservado algún plano para poder dibujar estas cosas, y garantizar que nunca se dibuje nada en esos planos.

Para este trato vamos a crear lo que llamaremos LayerZ.

La misión del LayerZ es la de dividir nuestro BufferZ en distintas capas de trabajo y activarlas cuando nos interese, de esa forma podemos controlar en que capa se dibujan los objetos y controlar de forma fiable las superposiciones.

1- Uso

El LayerZ es de uso muy sencillo, y su configuración también es muy sencilla

```
void layerZInit(int n);
void layerZStartAt(int n,int minvalue);
void layerZActive(int n);
```

Empezamos por iniciar el sistema diciéndole únicamente en cuantas capas vamos a dividir el bufferZ. Vamos a suponer que nos interesa dividir el espacio en esta estructura, queremos crear 3 Capas, tal y como se representa:

```
LayerInit(3); // empezamos por indicar el numero de capas
```

Esto nos da que vamos a tener 3 capas (numeradas como 0,1 y 2), ahora solo nos faltaría configurar el rango de cada capa, para ello tenemos el método **layerZStartAt** que únicamente tiene un parámetro, el punto donde empieza (por defecto siempre termina donde empieza la siguiente).

```
// 0x0000.....0xFFFF
// |         |                               |         |
// | LAYER 0 |                               | LAYER 2 |
// |         |                               |         |
// |         |                               |         |
// |         |                               |         |
// 0         256                               (65536-256)       65536

LayerZStartAt(0,0x0000); // Capa 0
LayerZStartAt(1,0x00FF); // Capa 1
LayerZStartAt(2, 0xFFFF - 0x00FF); // Capa 2
```

Ahora solo nos falta ver en un ejemplo como activamos cada una de las capas, y como dibujaríamos sobre cada capa:

```
gfxSyncV(); // Esperamos al Retazo
gfxSwapBuffers(); // cambiamos el buffer de dibujado
gfxClear(); // y limpiamos la pantalla

// Dibujamos en LAYER 0
layerZActive(0);
DibujaElementosFondo();

// Dibujamos en LAYER 1
layerZActive(1);
DibujaElementosActivos();

// Dibujamos en LAYER 2
layerZActive(2);
DibujaElementosPanelSuperior();

gfxRender(); // enviamos todo el flujo al GS
```

Nota: El método `gfxClear()` tiene poco que ver con un clear de la pantalla, en realidad lo que hace es limpiar aquellos registros temporales que se hayan modificado anteriormente, pero también resetea el LayerZ activando la capa principal, por lo que no sería necesario activar la capa(0), aunque se haga por claridad de código.

Vamos a ver un ejemplo de uso algo mas completo, esta vez queremos tener objetos en 3D que se estén renderizando en planos totalmente separados, ya que por en medio necesitamos tener una capa para dibujar algunos sprites. Total.... nada mas sencillo que hacer esto:

En primer lugar empezamos por declarar las 2 cámaras de trabajo

```
// Vamos a tener 2 Camaras
ObjectCamera *CameraLayer1;
ObjectCamera *CameraLayer3;
```

Configuramos el LayerZ, nos preparamos los espacios de trabajo, podemos ver el rango que dejamos para la cámara que trabaje sobre el Layer1 y el Layer3

```
//
// 0x0000.....0xFFFF
// | LAYER 0 | LAYER 1 | LAYER 2 | LAYER 3 | LAYER4 |
// | 0 | FF | 7FFF-FF | 7FFF+FF | FFFF-FF | FFFF |

layerZInit(5);

layerZStartAt(0,0x0000);
layerZStartAt(1,0x00FF); // <-- aqui crearemos una camara 3D
layerZStartAt(2,0x7FFF - 0xFF);
layerZStartAt(3,0x7FFF + 0xFF); // <-- aqui la otra camara 3D
layerZStartAt(4,0xFFFF - 0xFF);
```

Creamos ambas cámaras, indicando la profundidad de la misma y la Layer sobre la que queremos que renderice los objetos:

```
// Preparamos 2 camaras, cada una de ellas va a renderizar los objetos
// para encajarlos en la capa seleccionada
CameraLayer1 = cameraCreateForLayer(-200,1);
CameraLayer3 = cameraCreateForLayer(-200,3);
```

Ahora solo nos falta ver como dibujar

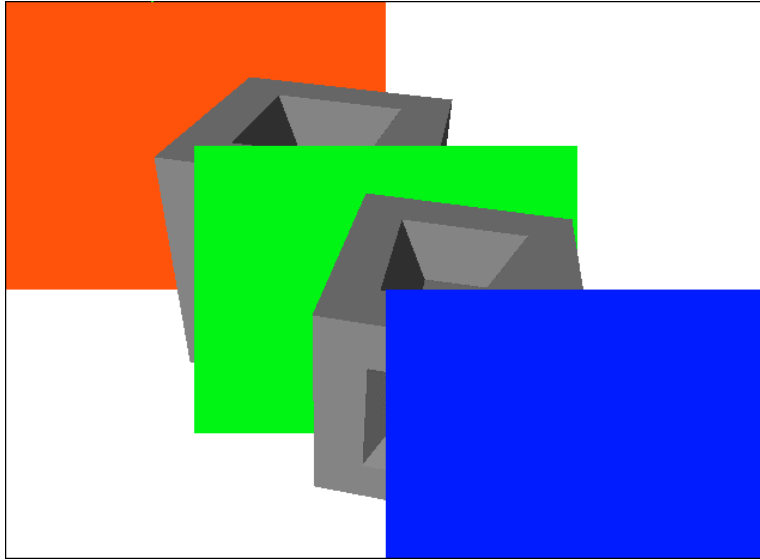
```
// LAYER 0
layerZActive(0);
gfxSetColor(0xFF,0x00,0x00,0x80); // red
gfxDrawBox(-320,-240,0,0);

// LAYER 1
layerZActive(1);
cameraSetActive(CameraLayer1); // <- Activamos la camara
objectReset(object);
objectRotate(object,x,y,z);
objectTranslate(object,-20,20,0);
objectRender(object);

// LAYER 2
layerZActive(2);
gfxSetColor(0x00,0xFF,0x00,0x80); // green
gfxDrawBox(-160,-120,160,120);

// LAYER 3
layerZActive(3);
cameraSetActive(CameraLayer3); // <- Activamos la camara
objectReset(object);
objectRotate(object,x,y,z);
objectTranslate(object,20,-20,0);
objectRender(object);

// LAYER 4
layerZActive(4);
gfxSetColor(0x00,0x00,0xFF,0x80); // blue
gfxDrawBox(0,0,320,240);
```



2 - Como funciona realmente la camara

Una de las matrices que se genera para operar con la proyección final es la CameraToScreen, esta matriz es la que define las transformaciones del punto de vista de la pantalla (entre otros factores podemos ver que tiene el escalado de la imagen y la posición en pantalla).

```
libVu0ViewScreenMatrix(
    cameraToScreen,
    512.0f,
    1.0f, 1.0f,           // factor de escalado x,y
    2048.0f, 2048.0f,     // centro de la pantalla x,y
    minz * 1.0f, maxz * 1.0f, // zmin, zmax
    64.0f, 65536.0f);    // nearz, farz
```

Al multiplicar esta matriz por el WorldToCamara (que es realmente la matriz que define la posición y dirección de cámara) es cuando se obtiene la matriz final para proyectar los vértices.

Bueno, a lo que vamos, en esa matriz tenemos 2 parámetros (zmin y zmax) que definen el valor en el bufferZ donde se van a dejar los vértices una vez proyectados, sabiendo esto es muy sencillo crear una cámara específica para que renderice dentro de una Layer en concreto. Para ello el LayerZ tiene 2 métodos que te devuelven el valor zmin y zmax de una Layer indicada:

```
unsigned int layerZGetMin(int n);
unsigned int layerZGetMax(int n);
```

Internamente la cámara para una Layer se crea mediante el siguiente código

```
ObjectCamera *cameraCreateForLayer(int factorZ, int layer)
{
    return cameraCreateZRange(factorZ, layerZGetMin(layer), layerZGetMax(layer));
}
```

Una vez que tenemos las cámaras creadas simplemente hay que activarlas cuando queramos que entren en uso.