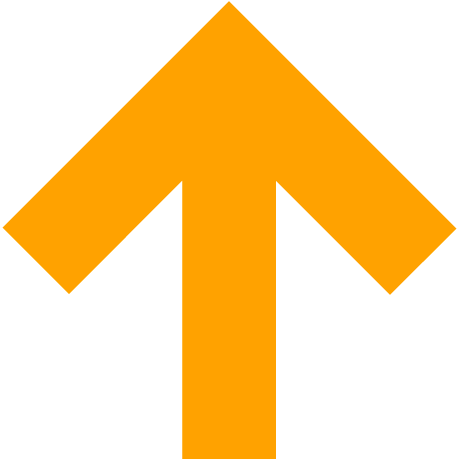
Noviembre 2022

|  |
| --- |
| MANUEL SANTA ISABEL MAYO  RAUL MELGOSA SALVADOR |



|  |  |
| --- | --- |
| [Animación digital – Ingeniería Informática] | [Tema 12: Efectos visuales: Fusiones y niebla] |

Contenido

[2 Introducción 1](#_Toc119429639)

[3 Conceptos básicos sobre OpenGL 2](#_Toc119429640)

[3.1 Creación de una ventana 2](#_Toc119429641)

[3.2 Dibujo de una figura 3](#_Toc119429642)

[3.3 Primer ejercicio de prueba 4](#_Toc119429643)

[3.4 Gestion basica de estados 5](#_Toc119429644)

[3.5 Gestión del redimensionado de la ventana 6](#_Toc119429645)

[4 Fusiones 8](#_Toc119429646)

[4.1 Introducción al concepto de fusión: El modelo RGBA 8](#_Toc119429647)

[4.2 Valores alfa 9](#_Toc119429648)

[4.3 Los factores de origen y destino 9](#_Toc119429649)

[4.4 Primer ejemplo 11](#_Toc119429650)

[4.5 Fusión tridimensional con el buffer de profundidad 17](#_Toc119429651)

[4.6 Conceptos de iluminación básicos 19](#_Toc119429652)

[4.7 Conceptos sobre representación de objetos en 3D 19](#_Toc119429653)

[4.8 Segundo ejemplo 21](#_Toc119429654)

[5 Niebla 27](#_Toc119429655)

[5.1 Introducción al concepto de niebla 27](#_Toc119429656)

[5.2 Declaración de Fog en Opengl 28](#_Toc119429657)

[5.3 Modos de niebla 28](#_Toc119429658)

[5.4 Ejemplo práctico 30](#_Toc119429659)

[6 Reflejos 32](#_Toc119429660)

[6.1 Introducción al concepto de reflejo 32](#_Toc119429661)

[6.2 Stencil Test y Stencil Buffer 33](#_Toc119429662)

[6.3 Ejemplo práctico 35](#_Toc119429663)

[7 Efecto de partículas 37](#_Toc119429664)

[7.1 Introducción al concepto de partículas: Alfa Chanel 37](#_Toc119429665)

[7.2 Ejemplo práctico 38](#_Toc119429666)

[8 Bibliografía 42](#_Toc119429667)

[9 Bibliografía 43](#_Toc119429668)

#### Figuras

[Figura 1. Esquema de transformación de coordenadas en OpenGL 7](#_Toc88272154)

[Figura 3. Fusión de dos triángulos cuando el triángulo amarillo se dibuja primero 15](#_Toc88272155)

[Figura 4. Fusión de 2 triangulos cuando el triángulo azul se dibuja primero 16](#_Toc88272156)

[Figura 5. Fusión de 2 triángulos cuando el triángulo azul se dibuja primero 17](#_Toc88272157)

[Figura 6. Fusión de 2 triágulos cuando el triangulo amarillo se dibuja primero 17](#_Toc88272158)

[Figura 7. Representación de la esfera opaca por delante del cubo translúcido 25](#_Toc88272159)

[Figura 8. Fusión del cubo translúcido con la esfera opaca 26](#_Toc88272160)

[Figura 9. Previsualización del efecto de niebla 27](#_Toc88272161)

[Figura 10. Videojuego Sillent Hills (1999) 28](#_Toc88272162)

[Figura 11. Modos de niebla 29](#_Toc88272163)

[Figura 12. Modo lineal 31](#_Toc88272164)

[Figura 13. Modo exponencial 31](#_Toc88272165)

[Figura 14. Modo exponencial al cuadrado 31](#_Toc88272166)

[Figura 15. Ejemplo con cambio de densidad 32](#_Toc88272167)

[Figura 16. Ejemplo exponencial-exponencial^2 con animación 32](#_Toc88272154)

[Figura 17. Stencil Test y Buffering 33](#_Toc88272155)

[Figura 18. Efecto de reflejo 38](#_Toc88272156)

[Figura 19. Alpha Channel 39](#_Toc88272157)

[Figura 20. Efecto de partículas en OpenGL 42](#_Toc88272158)

#### Tablas

Tabla 1. Tabla de macros para la función glBlendFunc() ………………………………………………..… 10

Tabla 2. Opciones de glStencilFunc()……………………………………………………………………………….. 34

Tabla 3. Opciones de uso de glStencilOp() ………………………………………………………………………. 35

# Introducción

Hola, somos Raul Melgosa Salvador y Manuel Santa Isabel Mayo. En esta práctica vamos a utilizar el entorno API de OpenGL para producir efectos de fusiones (blending) y de niebla. Para hacer uso de esta API, vamos a utilizar la librería glut, la cual se puede descargar gratuitamente en diferentes webs.

Lo primero que queremos destacar en esta introducción es cuál es el entorno de desarrollo elegido para poner en marcha esta práctica. Como se indica en el enunciado de la práctica, vamos a hacer el uso de la API de OpenGL en el lenguaje de programación C++. Para programar en C++ existen distintos entornos de desarrollo, siendo el más conocido el entorno de desarrollo Visual Studio. Sin embargo, nosotros hemos considerado hacer uso del entorno de desarrollo Dev-C++ ya que consideramos que es un entorno de desarrollo más sencillo en el que podemos implementar las funciones de la biblioteca de glut de OpenGL.

Otro punto importante que queremos destacar es la versión de GLUT que vamos a utilizar para el desarrollo de la práctica. En el curso de la asignatura en Studium está disponible un enlace de descarga de la versión 3.7.6. Sin embargo, nosotros hemos descargado la versión 3.0.0. gratuita de glut (freeglut) a través del siguiente enlace:

<https://www.transmissionzero.co.uk/software/freeglut-devel/>

En cuanto al proceso de instalación de GLUT no difiere mucho del proceso utilizado en otras versiones. Básicamente se han realizado 3 acciones:

* Hemos colocado las bibliotecas de funciones de glut (ficheros .a) en la carpeta lib de Dev-C++.
* Hemos colocado los archivos de cabecera de glut (glut.h, etc) en la carpeta include de Dev-C++.
* Hemos colocado la Biblioteca de enlace dinámico (freeglut.dll) en la carpeta C:\Windows\System32 del equipo.

Como último detalle, hemos añadido estas directivas al compilador de Dev-C++ para que nos incluya las funciones y librerías necesarias de GLUT y OpenGL necesarias para poner en marcha nuestra práctica:

-lopengl32

-lfreeglut

-lglu32

# Conceptos básicos sobre OpenGL

## Creación de una ventana

OpenGL es un framework diseñado para renderizar gráficos pero está diseñado para ser independiente de cualquier sistema de ventanas o de cualquier sistema operativo. Debido a esto, OpenGL como tal no tiene comandos para abrir ventanas o leer eventos de teclado o del ratón. Es la biblioteca de funciones GLUT la que se encarga de realizar esta gestión de las ventanas y de los eventos y la realiza de forma simplificada.

Para la gestión de la ventanas en GLUT existen 5 funciones, tareas o rutinas que hay que seguir:

* **glutInit**(int \*argc, char \*\*argv): Este comando se utiliza para inicializar el entorno de GLUT. Este comando debe llamarse antes de cualquier otro.
* **glutInitDisplayMode**(unsigned int mode): Este comando se utiliza para especificar si usar un modelo de color *RGBA* o de índice de color. También puede especificar si desea una ventana de búfer simple o doble. En nuestros ejemplos siempre utilizaremos (a no ser que se indique lo contrario) el modelo de color RGBA y la ventana de buffer simple. Para ello se utilizarán las macros (GLUT\_SINGLE | GLUT\_RGB).
* **glutInitWindowPosition**(int x, int y): Este comando se utiliza para especificar la posición de la ventana con respecto a la esquina superior izquierda.
* **glutInitWindowSize**(int width, int size): Este comando sirve para especificar el tamaño de la ventana.
* int **glutCreateWindow**(char \*string): Este comando crea una ventana con un “contexto” OpenGL (contexto será el parámetro que le pasemos a la función y corresponde con el nombre de la ventana). Hasta que no se llame a la función **glutMainLoop()** que veremos a continuación, la ventana no se mostrará.

Una vez se hayan definido los parámetros de la ventana, lo último que hay que hacer es llamar a la función **glutMainLoop**(void) que muestra todas las ventanas creadas. Con esto tenemos los conceptos básicos para crear y mostrar una ventana.

Una vez explicado esto nos vamos a adentrar en conceptos un poco más avanzados. En la asignatura de Sistemas Operativos vimos lo que eran las funciones de rellamada. En GLUT se utilizan distintas funciones de llamada (callback). La función de rellamada más importante es la función **glutDisplayFunc**(void (\* func )(void)). Cada vez que GLUT determine que el contenido de la ventana debe mostrarse de nuevo, se ejecutará la función manejadora definida en esta función de rellamada **glutDisplayFunc.**

Dentro de esta función manejadora debe incluirse todos los procedimientos para volver a dibujar de nuevo el contenido de la pantalla. Cuando se invoca que la función  **glutMainLoop**(void) que muestra el contenido de la ventana, comenzará la gestión de eventos y se ejecutará la función de rellamada **glutDisplayFunc** (y otras funciones de rellamada en el caso de que las haya) de manera cíclica.

De esta manera, cada vez que se produzca el *evento* en el que GLUT determina que el contenido de la ventana debe cambiar, se ejecutará la función *manejadora* que se ha registrado en la *función de rellamada*.

En ocasiones, si nuestro propio programa de OpenGL cambia el contenido de la ventana, tendremos que redibujar el contenido de la ventana utilizando el conjunto de comandos que se encuentra en la manejadora definida en **glutDisplayFunc**.Para ello, se hará uso del comando **glutPostRedisplay**() el cual provocará que el **glutMainLoop()** llame a la manejadora definida en **glutDisplayFunc** para el redibujado de la pantalla.

Básicamente con esto conseguiremos “forzar” un evento donde el manejo de este evento implica la ejecución de un porción de código específica donde en este caso esa porción de código estará destinada a volver a dibujar el contenido de la ventana.

Otras funciones de rellamada:

* **glutReshapeFunc**(void (\*func)(int w, int h)): Esta función de rellamada se utilizará para registrar una función manejadora que se ejecute cuando cambie el tamaño de la ventana.
* **glutKeyboardFunc**(void (\*func)(unsigned char key, int x, int y)): Esta función de rellamada se utilizará para registrar una función manejadora que se ejecute cuando se presiona una tecla del teclado.
* **glutMouseFunc**(void (\*func)(int button, int state, int x, int y)): Esta función de rellamada se utilizará para registrar una función manejadora que se ejecute cuando se presiona una tecla del ratón.
* **glutMotionFunc**(void (\*func)(int x, int y)): Esta función de rellamada se utilizará para registrar una función manejadora que se ejecute cuando se mueva el ratón al mismo tiempo que se presiona una tecla del ratón.

También existe una función de rellamada **glutIdleFunc** (void (\* func )(void)) que se utiliza para registrar una función manejadora que se ejecute cuando no haya eventos pendientes de atender.

## Dibujo de una figura

Una vez sabemos, crear una ventana y registrar funciones que se ejecuten cuando ocurra un determinado evento, el siguiente paso es empezar a dibujar cosas en esta ventana. Para ello vamos a ver un conjunto de comando básicos de OpenGl cuya funcionalidad se va a explicar a continuación:

* **glClearColor() y glClear()**: Estos comandos se utilizan para borrar la pantalla. Con **glClearColor()** se establece el color con el que se borra la pantalla y con **glClear()** se hace efectivo ese borrado. Una vez se ha establecido el color de limpieza, cada vez que se llame a **glClear()** la ventana se borra con ese color hasta que se diga lo contrario.
* **glColor3f()**: Este comando se utiliza para establecer el color en el que se van a dibujar las distintas figuras y objetos. Una vez definido este color, todas las figuras y objetos se dibujarán con este color hasta que se diga lo contrario.
* **glOrtho()**: Este comando se utiliza para establecer el sistema de coordenadas que debe usar OpenGL para dibujar la imagen
* **glBegin()** y **glEnd()**: Estos comandos sirven para indicar el comienzo y el fin del bloque de código utilizado para definir el objeto o figura a dibujar. Con el comando **glBegin()** específicas con una macro el tipo de figura a crear (línea, polígono, puntos separados, etc)
* **glVertex3f()**: Este comando permite definir las coordenadas de los distintos puntos que forman un polígono o figura. Si por ejemplo se desea dibujar un triángulo habrá que definir las coordenadas de los 3 puntos que forman ese triángulo.
* **glFlush()**: Este comando se utiliza para ejecutar los comandos de dibujo en vez de que se almacene en un buffer a la espera de otros comandos OpenGL adicionales.

Estas funciones tienen distintos parámetros los cuales no se han comentado en profundidad ya que el objetivo era comentar brevemente qué hace cada comando. Para ver el significado de los distintos parámetros de cada comando, simplemente hay que buscar su documentación en internet. En el siguiente enlace se puede obtener fácilmente la documentación de las distintas funciones de OpenGL:

<https://learn.microsoft.com/es-es/windows/win32/opengl/gl-functions>

## Primer ejercicio de prueba

Con esto ya tenemos una base teórica suficiente para realizar un primer ejercicio en nuestro entorno de OpenGL con el apoyo de nuestras bibliotecas GLUT. Este primer ejemplo consisteira en dibujar un rectangulo de color blanco sobre una panatala de fondo negro. El código para realizar esta tarea será el siguiente:

#include <GL/glut.h>

**void** display(**void**)

{

    glClear (GL\_COLOR\_BUFFER\_BIT);

    glColor3f(1.0, 1.0, 1.0);

    glBegin(GL\_POLYGON);

        glVertex3f(0.25, 0.25, 0.0);

        glVertex3f(0.75, 0.25, 0.0);

        glVertex3f(0.75, 0.75, 0.0);

        glVertex3f(0.25, 0.75, 0.0);

    glEnd();

    glFlush ();

}

**void** init (**void**)

{

    glClearColor(0.0, 0.0, 0.0, 0.0);

    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);

}

**int** main(**int** argc, **char**\*\* argv)

{

    glutInit(&argc, argv);

    glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

    glutInitWindowSize(800, 800);

    glutInitWindowPosition(0, 0);

    glutCreateWindow("Ejemplo rectángulo básico");

    init();

    glutDisplayFunc(display);

    glutMainLoop();

**return** 0;

}

Obsérvese cómo en este ejemplo se han hecho uso de todas las funciones explicadas anteriormente para la creación de un rectángulo blanco sobre un fondo negrto. Lo primero que se ha hecho es iniciar el entorno de GLUT con **glutInit()** y después se han definido el tipo, el tamaño y la posición de la ventana con las funciones **glutInitDisplayMode()**, **glutInitWindowSize()** y **glutInitWindowPosition()**. Una vez se han definido los parámetros de la ventana se procede a la creación de la misma a través del comando **glutCreateWindow()**.

Una vez hemos creado la ventana a partir de las funciones de biblioteca de GLUT, llamaremos a una función init() que contendrán una serie de comandos que “prepararan” la ventana para poder dibujar sobre ella. En nuestro caso, la función init() contendrá 2 comandos **glClearColor()** y **glOrtho()** que definirán el color de borrado de la pantalla y el espacio de coordenadas en el que se realizarán los dibujos.

Por último, definimos una función manejadora (llamada display) a través de la función de rellamada **glutDisplayFunc()** en el que cada vez que se produzca el evento de cambiar el dibujo, se ejecutar un serie de coanndo que realair el dibujo del cuadrado correspondiente.

Esta función manejadora consta de un comando **glClear()** para limpiar el contenido de la ventana, un comando **glColor3f()** para establecer el color de la figura, los comandos **glBegin()** y **glEnd()** en los que se define los puntos que van a formar el cuadrado utilizando el comando **glVertex3f()** para definir cada punto y el cuando **glFlush()** para dibujar la figura inmediatamente y que no se quede almacenada en los buffers.

## Gestion basica de estados

Un objeto se puede dibujar o renderizar con iluminación, texturizado, eliminación de superficies ocultas, fusión, niebla, etc o algún otro tipo de estado que modifique la apariencia del objeto. Cada uno de estos estados digamos que está controlada por una variable de estado asociada. Inicialmente todos estados están desactivadas pero tenemos la opción de activar y desactivar distintos estados a través de 2 comandos:

* *void* ***glEnable****(GLenum* cap*):* Este comando permite activar un estado el cual se indicará como parámetro.
* *void* ***glDisable****(GLenum* cap*):* Este comando permite desactivar un estado el cual se indicará como parámetro.

Existen más de 40 estados que se pueden pasar como parámetro a ***glEnable()*** y ***glDisable()*** pero en nuestro caso solo nos interesan los estados *GL\_BLEND* y *GL\_FOG* los cuales controlan la combinación de valores RGBA y la niebla respectivamente y los cuales vamos a tratar en este documento.

También existe otro comando llamado ***glIsEnabled()*** el cual nos permite verificar si un determinado estado está habilitado o deshabilitado. En el caso de que esté habilitado devolverá *GL\_TRUE*y en caso contrario devolverá *GL\_FALSE*.

## Gestión del redimensionado de la ventana

Ahora, vamos a imaginar que tenemos un determinado dibujo en la ventana (por ejemplo el cuadrado que hemos dibujado en el ejemplo anterior) y en un determinado momento el usuario reajusta el tamaño de la ventana. En este momento, las propiedades del dibujo en cuanto a su forma pueden alterarse de una manera que no es la deseada.

Para evitar que el redimensionado de la ventana altere el contenido de la ventana de una manera no deseada vamos a utilizar un conjunto de comandos que nos van a ayudar en la gestión del redimensionado. Los comandos son los siguientes:

**glViewport()**: Este comando se utiliza para ajustar las coordenadas del dispositivos normalizadas *x* e *y* a las coordenadas de la ventana. Con este comando se define el lugar donde se renderiza el contenido de la ventana. Para calcular esta coordenadas de la ventana se van a proporcionar 4 parámetros.

Los 2 primeros parámetros hacen referencia a las coordenadas x e y de la esquina inferior izquierda de la ventana (suelen tener los valores 0, 0 ya que se suele utilizar esta esquina inferior izquierda como origen de coordenadas).

Los 2 siguientes parámetros hacen referencia al alto y ancho de la ventana redimensionada.

**glMatrixMode()**: Antes de realizar una explicación de este comando vamos a explicar el proceso que utiliza OpenGl para renderizar los distintos objetos en la pantalla:

OpenGL realiza un conjunto de transformaciones para transformar la coordenadas de un objeto cualquiera (de 4 dimensiones) en coordenadas de ventana (2 dimensiones). Para realizar esta transformación se utilizan 3 transformaciones llamadas transformación del modelo de vista, transformación de la proyección y transformación del viewport.

En la siguiente imagen (Figura 1) se adjunta un esquema de este sistema de transformación de coordenadas que estamos explicando:

Diagrama

Descripción generada automáticamente

1. **Esquema de transformación de coordenadas en OpenGL**

Como se puede observar en la imagen, el primer paso es convertir las coordenadas del objeto en coordenadas del ojo. La coordenadas del ojo es un sistema de coordenadas en que se define la escena respecto el punto de vista de la cámara. Para obtener estas coordenadas del ojo se van a multiplicar estas coordenadas del objeto por una matriz de puntos de vista.

El segundo paso consiste en convertir las coordenadas del ojo en coordenadas de recorte. Las coordenadas de recorte hacen referencia a las coordenadas del volumen de visualización normalizado.. Para obtener estas coordenadas de recorte se van a multiplicar estas coordenadas del ojo por una matriz de proyección.

El tercer paso es realizar una proyección de perspectiva. Este paso no es demasiado importante para esta práctica porque simplemente lo mencionamos. El resultado es la obtención de las coordenadas del dispositivo. Estas coordenadas del dispositivo son el resultado de la selección de las coordenadas que caen dentro del volumen de visualización normalizado.

En último paso consiste en transformar las coordenadas del dispositivo en coordenadas de ventana. Estas coordenadas de ventana corresponden con las coordenadas de la pantalla del ordenador. Estas coordenadas de pantalla son el resultado de aplicar la transformación de viewport a las coordenadas de dispositivo.

Una vez explicado más o menos el proceso de transformación de coordenadas que utiliza OpenGL para la representación de objetos en una ventana vamos a explicar para qué se utiliza este comando. Básicamente este comando se utiliza para especificar con qué tipo de matriz queremos trabajar.

Podemos trabajar con 3 matrices distintas. La matriz del modelo de la vista, la matriz de proyección y la matriz de textura aunque esta última no la vamos a utilizar.

**glLoadIdentity()**: Este comando se utiliza para asignar a la matriz seleccionada a  través del comando **glMatrixMode()** la matriz identidad. Como ya sabemos cualquier matriz “origen” multiplicada por la matriz identidad tendrá como resultado esa misma matriz “origen”. Si establecemos una matriz de transformación como una matriz identidad esa transformación sería nula ya que el efecto de multiplicar una matriz “origen” por la matriz identidad es nulo.

**gluOrtho2D()**: Este comando se utiliza para definir una matriz de proyección ortográfica 2D. Esta matriz de proyección ortográfica 2D se multiplicará por la matriz de proyección actual. Los 4 parámetros que recibe son las coordenadas de las rectas de recortado. Con estas 4 rectas de recortados podemos recortar la proyección del objeto final.

# Fusiones

## Introducción al concepto de fusión: El modelo RGBA

Con esta introducción, ya podemos entender los conceptos básicos que suele utilizar un programa que utiliza las funciones de OpenGL y de la biblioteca GLUT. De esta manera,

ya estamos capacitados para aprender a realizar efectos de fusión o bending en OpenGL no sin antes aprender ciertos contenidos teóricos sobre la fusión en OpenGL.

Como ya sabemos, la pantalla de un ordenador está formada por un conjunto de pequeños “cuadraditos” llamados píxeles. El hardware del ordenador hace que cada píxel emite diferentes cantidades de luz roja, luz azul y luz verde obteniéndose así un color determinado. De esta manera, para representar una imagen cada píxel va a utilizar una combinación de estas 3 luces para representar el color correspondiente de la imagen.

Este modelo en el que se utilizan 3 luces de colores rojo, verde y azul se conoce como modelo RGB. A menudo se incluye un 4 valor llamado alfa o A obteniéndose así el modelo RGBA. De esta manera, se almacenarán los valores R, G, B y A para cada pixel o bien en una estructura centralizada llamada “mapa de color” o bien en estructuras individuales para cada pixel.

A modo de resumen, los valores R, G y B nos van a permitir representar los distintos colores pero es el valor A o alfa el que nos va a interesar para realizar los efectos de fusión o blending. Ahora explicaremos que es este valor A en el modelo RGBA y porque es tan importante para realizar estos efectos de fusión.

## Valores alfa

Cuando el estado de mezcla (*GL\_BLEND*) está activado, este valor alfa se utiliza para combinar el color del fragmento que se está procesando con el valor del color del pixel que ya está almacenado en el buffer de fotogramas. Esta fusión se produce cuando el objeto ya está rasterizado (se tienen los valores R, G y B del fragmento ya calculados) pero antes de que los píxeles del buffer de fotogramas sean dibujados.

Cuando el estado de mezcla (*GL\_BLEND*) no está activado, cada fragmento sobrescribe los valores del color del buffer de fotogramas. En este caso, no existe ningún tipo de combinación o mezcla. De esta manera, con este valor alfa se define la proporción o porcentaje del color existente en los buffers de fotograma deben combinarse con los valores del nuevo fragmento que se está procesando.

Para entender esto que acabamos de explicar vamos a poner un ejemplo sencillo:

Supongase que estamos viendo un objeto de color naranja por ejemplo a traves de un vidrio translucido verde. Si este vidrio deja pasar el 80% por ciento de la luz que incide sobre él (tiene una opacidad del 20%), el color del objeto que se verá será una combinación del 20% del verde del video con un 80% del naranja del objeto.

En este caso, el valor alfa de este color del vidrio es del 20 por ciento y este valor alfa depende de la opacidad de la botella. Es por esto por lo que muchas veces se piensa que el valor A del modelo RGBA se asocia a la opacidad aunque ya hemos visto que hemos definido este valor A de una manera más “exquisita”.

## Los factores de origen y destino

Como hemos dicho, en la fusión los valores del fragmento que se está procesando (*origen*) se combinan con los valores de los píxeles almacenados en los buffers de fotogramas (*destino*). Esta fusión se realiza en 2 etapas:

En primer lugar, se especifica cómo calcular los factores del origen y destino. Estos factores son una cuadrupla de valores R, G, B, A que se obtienen por la multiplicación de los componentes RGBA del origen y el destino respectivamente.

En segundo lugar, se calcula el valor RGBA resultante de la mezcla a partir de los factores de origen y destino y los valores RGBA de origen y destino siguiendo la siguiente fórmula:

(RsSr+RdDr, GsSg+GdDg, BsSb+BdDb, AsSa+AdDa)

Donde:

* Factor de origen: (Sr, Sg, Sb, Sa)
* Factor de destino: (Dr, Dg, Db, Da)
* RGBA origen: (Rs, Gs, Bs, As)
* RGBA destino: (Rd, Gd, Bd, Ad)

**Nota**: Cada valor R, G, B y A tiene valores que oscilan entre 0 y 1 (ambos incluidos).

Una vez entendido cómo se realiza la fusión, vamos a ver cómo se realiza esta fusión en OpenGL. Como ya mencionamos anteriormente, para renderizar un objeto con el efecto de fusión, tenemos que activar este estado de fusión utilizando la función **glEnable()** y la macro GL\_BLEND para indicar que queremos activar el estado de fusión.

Una vez hemos activado el estado de fusión, para realzar el efecto de fusión utilizaremos la función ***glBlendFunc****(GLenum* sfactor*, GLenum* dfactor*)* la cual utiliza 2 parámetros que son 2 especificaciones para calcular el índice de origen y el índice de destino respectivamente.

En la siguiente tabla (Figura 2) se adjunta las posibles macros que se pueden utilizar como parámetros de la función ***glBlendFunc()*** junto con el doctor de mezcla calculado al aplicar esa macro.

|  |  |  |  |
| --- | --- | --- | --- |
| **Constant** | **Relevant Factor** | **Computed Blend Factor** |  |
| GL\_ZERO | Source or destination | (0,0,0,0) |  |
| GL\_ONE | Source or destination | (1,1,1,1) |  |
| GL\_DST\_COLOR | Source | (Rd,Gd,Bd,Ad) |  |
| GL\_SRC\_COLOR | Destination | (Rs,Gs,Bs,As) |  |
| GL\_ONE\_MINUS\_SRC\_COLOR | Source | (1,1,1,1)- (Rd,Gd,Bd,Ad) |  |
| GL\_ONE MUNUS\_SRC\_COLOR | destination | (1,1,1,1)- (Rs,Gs,Bs,As) |  |
| GL\_SRC\_ALPHA | Source or destination | (As,As,As,As) |  |
| GL\_ONE\_MINUS\_SRC\_ALPHA | Source or destination | (1,1,1,1)- (As,As,As,As) |  |
| GL\_DST\_ALPHA | Source or destination | (Ad,Ad,Ad,Ad) |  |
| GL\_ONE\_MINUS\_DST\_ALPHA | Source or destination | (1,1,1,1)- (Ad,Ad,Ad,Ad) |  |
| GL\_SRC\_ALPHA\_STATURATE | source | (f,f,f,1); f= min (As,1-Ad) |  |

**Tabla 1.     Tabla de macros para la función glBlendFunc()**

También indica en la tabla si la macro se puede utilizar para calcular el factor de origen, el de destino o el de ambos.

## Primer ejemplo

Una vez hemos visto cuales son los fundamentos teóricos y cuales son las funciones de OpenGL básicas para la realización del efecto de fusión vamos a realizar un primer ejemplo relativamente sencillo que va a utilizar este efecto de fusión.

Este ejemplo va a consistir en el dibujado de 2 triángulos con distintos colores (azul y amarrilo) donde estos 2 triángulos van a compartir un área en común, en la que queremos que se fusionen los colores de ambos triángulos. Además, vamos a crear un mecanismo rudimentario (a través de la pulsación de un botón del teclado) en el que podamos cambiar este efecto de fusión de manera que en ocasiones predomine más el color amarillo y en otras predomine más el color azul.

El código para realizar esta tarea (obviando la creación de la ventana) será el siguiente:

#include <GL/gl.h>

#include <GL/glu.h>

#include <GL/glut.h>

#include <stdlib.h>

**static** **int** primeroIzquierda = GL\_TRUE;

**static** **void** init(**void**)

{

   glEnable(GL\_BLEND);

   glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

   glClearColor(0.0, 0.0, 0.0, 0.0);

}

**static** **void** dibujarTrianguloIzquierdo()

{

   glBegin(GL\_TRIANGLES);

      glColor4f(1.0, 1.0, 0.0, 0.75);

      glVertex3f(0.1, 0.9, 0.0);

      glVertex3f(0.1, 0.1, 0.0);

      glVertex3f(0.7, 0.5, 0.0);

   glEnd();

}

**static** **void** dibujarTrianguloDerecho()

{

   glBegin(GL\_TRIANGLES);

      glColor4f(0.0, 1.0, 1.0, 0.75);

      glVertex3f(0.9, 0.9, 0.0);

      glVertex3f(0.3, 0.5, 0.0);

      glVertex3f(0.9, 0.1, 0.0);

   glEnd();

}

**void** display(**void**)

{

   glClear(GL\_COLOR\_BUFFER\_BIT);

**if** (primeroIzquierda) {

      dibujarTrianguloIzquierdo();

      dibujarTrianguloDerecho();

   }

**else** {

      dibujarTrianguloDerecho();

      dibujarTrianguloIzquierdo();

   }

   glFlush();

}

**void** reshape(**int** w, **int** h)

{

   glViewport(0, 0, (GLsizei) w, (GLsizei) h);

   glMatrixMode(GL\_PROJECTION);

   glLoadIdentity();

**if** (w <= h)

      gluOrtho2D(0.0, 1.0, 0.0, 1.0\*(GLfloat)h/(GLfloat)w);

**else**

      gluOrtho2D(0.0, 1.0\*(GLfloat)w/(GLfloat)h, 0.0, 1.0);

}

**void** keyboard(**unsigned** **char** key, **int** x, **int** y)

{

**switch** (key) {

**case** 't':

**case** 'T':

         primeroIzquierda = !primeroIzquierda;

         glutPostRedisplay();

**break**;

**case** 27: *//el 27 hace referencia al carácter de escape*

         exit(0);

**break**;

      default:

**break**;

   }

}

Obsérvese cómo en este caso, en la función de iniciación **init()** además de establecer el color de borrado de la pantalla, se ha activado del estado de fusión a través de la función ***glEnable()*** y posteriormente se ha especificado la manera de calcular los factores de fusión de origen y de destino a través de la función ***glBlendFunc()***.

En este caso, se han utilizado las macros GL\_SRC\_ALPHA y GL\_ONE\_MINUS\_SRC\_ALPHA para especificar el cálculo de los factores de fusión de origen y destino. De esta manera, vamos a utilizar el valor de la saturación de origen (GL\_SRC\_ALPHA) y su inverso (GL\_ONE\_MINUS\_SRC\_ALPHA) para calcular estos factores de origen y destino.

Por otro lado, en las funciones **drawLeftTriangle()** y **drawRightTriangle()** dibujamos los trangulos de color amarillo (izquierda) y azul (derecha) respectivamente. Obsérvese cómo en este caso, tenemos la definición 3 de puntos en vez de 4 dentro de las funciones **glBegin()** y **glEnd()** y como la función **glBegin()** recibe como parámetro la macro GL\_TRIANGLES indicando la creación de un triángulo.

Obsérvese como ahora para definir los colores de ambos triángulos se utiliza una versión de **glColor()** que recibe 4 parámetros en vez de 3 ya que en este caso nos interesa darle un valor a alfa. En nuestro ejemplo utilizaremos un valor alfa de 0.75.

Por otra parte, vamos a ver como se ha gestionado el redimensionamiento de la ventana. Esto no interviene para nada en el proceso de fusión pero ya que hemos incidido en el proceso de representación de objetos en función del tamaño de la ventana anteriormente, vamos a comentarlo brevemente.

En primer lugar, se ha utilizado el comando **glViewport()** con el objetivo de definir el nuevo espacio de visualización (ventana) en el que se van a representar los objetos. En segundo lugar, se ha seleccionado la matriz de proyección a partir de la función **glMatrixMode()** y se le ha asignado el valor de la matriz identidad a través de la función **glLoadIdentity()**.

En último lugar, se ha utilizado el comando **gluOrtho2D()** para definir un volumen de visualización en 2D utilizando rectas de recortado. Si la pestaña es más alta que ancha se recortará por abajo y si la pestaña es más ancha que alta se recortará por la derecha.

En cuanto a la función **display** que gestiona el dibujado en la pantalla no hay mucho que comentar. Se utiliza la función **glClear()** para borrar la pantalla y la función **glFlush()** para forzar la representación de los objetos. Obsérvese que existe una variable booleana que determina el orden de dibujado de los triángulos.

En cuanto la función **keyboard** que gestiona los eventos del teclado, básicamente se evalúa si la tecla pulsada es una T (tanto mayúscula como minúscula) y en el caso de que se pulse esta tecla T se cambiará el contenido de la variable booleana y se forzara el redibujado a través de la función **glutPostRedisplay()**. De esta manera, cada vez que se pulse la tecla T cambiara el orden el que se dibujaran los triángulos lo cual tiene un impacto directo sobre el efecto de fusión el cual vamos a comentar a continuación.

En el dibujado de ambos triángulos hemos definido un valor alfa de 0.75. Si dibujamos primero el triangulo izquierdo (amarillo) y despues el derecho (azul) tenemos que el factor de origen es de 0.75 mientras que el factor de destino es del 0.25 (1 - 0.75).

De esta manera, en la fusión vamos a tener un 75 % del color del segundo rectangulo que es el azul (figura origen) y un 25 % del color del primer rectángulo que es el amarillo (figura destino) por lo que el color azul del segundo triangulo predominará sobre el color amarillo del primer triangulo.

Si dibujamos primero el triángulo derecho (azul) y después el izquierdo (amarillo) tenemos que el factor de origen es de 0.75 mientras que el factor de destino es del 0.25 (1 - 0.75).

De esta manera, en la fusión vamos a tener un 75 % del color del segundo rectangulo que es el amarillo (figura origen) y un 25 % del color del primer rectángulo que es el azul (figura destino) por lo que el color amarillo del segundo triangulo predominará sobre el color azul del primer triangulo.

En las siguientes capturas se mostrarán 2 ejemplos de ejecución para cuando se dibuja primero el triángulo izquierdo (amarillo) (Figura 3) como cuando se dibuja primero el triángulo derecho (azul) (Figura 4):

Icono

Descripción generada automáticamente con confianza baja

1. **Fusión de dos triángulos cuando el triángulo amarillo se dibuja primero**

Un conjunto de letras blancas en fondo azul

Descripción generada automáticamente con confianza media

1. **Fusión de 2 triángulos cuando el triángulo azul se dibuja primero**

Si queremos variar la predominancia de un color sobre otro en la mezcla simplemente debemos cambiar el valor de alfa al definir el color de los triángulos al gusto. Si por algún motivo se desea que la mezcla entre los colores de ambos triángulos sea “equitativa” (50% cada uno) se asignará a cada triángulo un valor alfa de 0.5.

Veamos cómo sería esta mezcla “equitativa” utilizando nuestro ejemplo (Figuras 5 y 6):

Un conjunto de letras blancas en fondo azul

Descripción generada automáticamente con confianza mediaUn conjunto de letras blancas en fondo azul

Descripción generada automáticamente con confianza media

1. **Fusión de 2 triángulos cuando el triángulo azul se dibuja primero**
2. **Fusión de 2 triángulos cuando el triángulo amarillo se dibuja primero**

Como se puede observar en la imagen, ahora la mezcla del triángulo amarillo y el triangulo azul da un color verdoso que se encuentra a medio camino de ambos. Sin embargo, al usar un valor alfa de 0.5 el color de ambos triángulos es más transparente debido a esta bajada del valor alfa en ambos triángulos.

Sin embargo, aunque se establezca este valor del 50 % ciento el valor de la mezcla se va a ver afectada ya que visualmente predomina el color del último triángulo dibujado.

## Fusión tridimensional con el buffer de profundidad

Como se pudo observar al final del ejemplo anterior, el orden en el que se dibujan los triángulos afecta en gran medida a la mezcla resultante. Esto para los objetos en 3D sucede de la misma manera ya que al realizar mezcla entre objetos translúcidos tridimensionales, esta mezcla también va a depender si el dibujado de estos objetos se realiza de atrás hacia adelante o de adelante hacia atrás. Además para los objetos en 3D hay que tener en cuenta el efecto del buffer de profundidad.

Este buffer de profundidad realiza un seguimiento de la distancia entre el punto de vista y la parte del objeto que ocupa un píxel determinado en una ventana de la pantalla. Cuando aparece otro color candidato para este píxel de la ventana de la pantalla, se dibuja solo el objeto que está más cerca del punto de vista. Este valor de distancia o profundidad se almacena en el buffer de profundidad. De esta manera, las partes oscurecidas u ocultas no se dibujan necesariamente y por tanto no se utilizan para fusionar.

Si se tiene en la misma escena un objeto opaco y un objeto translúcido, entonces es interesante utilizar el buffer de profundidad para eliminar la superficie oculta de cualquier objeto que se encuentre detrás de un objeto opaco. Pero aquí surge algo interesante. Si el objeto translúcido se encuentra detrás del objeto opaco, queremos que el buffer de profundidad elimine este objeto translúcido ya que se encuentra oculto y no se ve. Sin embargo, si el objeto translúcido está por delante del objeto opaco, queremos que el color de este elemento translúcido se mezcle con el color del objeto opaco que se encuentre detrás.

Si la escena es “rígida” y los objetos no se mueven es fácil determinar qué objeto está delante del otro pero si estos objetos se mueven o el punto de vista se mueve este proceso se complica. Sin embargo, hay una solución relativamente sencilla a este problema.

Esta solución consta de los siguientes pasos:

1. Habilitar el buffer de profundidad y dibujar todos los objetos opacos.
2. Cambiar el modo de funcionamiento del buffer de profundidad para que sea de solo lectura.
3. Dibujar los objetos translúcidos, donde los valores de profundidad de estos objetos translúcidos se compara con los valores de profundidad de los objetos opacos, los cuales se encuentran en el buffer de profundidad.
4. Si en esa comparación de profundidad, este objeto translúcido se encuentra detrás de algún objeto opaco entonces no se dibuja
5. Si el objeto translúcido está por delante de los objetos opacos, este si se dibuja y se mezcla con los objetos opacos ya que estos valores del buffer de profundidad no pueden cambiar.

Para controlar el modo de funcionamiento del buffer de profundidad en OpenGl se utiliza el comando **glDepthMask()** el cual permite habilitar y deshabilitar la escritura sobre los buffers de profundidad. Si el parámetro de este comando es GL\_FALSE la escritura en el buffer de profundidad estará desactivada y si el parámetro es GL-TRUE la escritura en el buffer de profundidad estará activada. Por defecto, la escritura en el buffer de profundidad estará habilitada.

## Conceptos de iluminación básicos

Para el próximo ejemplo que vamos a realizar más adelante vamos a fusionar un objeto opaco y uno transparente. Para hacer más interesante la mezcla, vamos a jugar un poco con la iluminación de los objetos para que el resultado de la mezcla sea más interesante.

Para poder jugar con la iluminación, en primer lugar tendremos que introducir una serie de comandos básicos de OpenGl que nos permiten gestionar la iluminación. Estos comandos son los siguientes:

**glMaterialfv()**: Este comando se utiliza para especificar parámetros de material para el modelo de iluminación. Recibe 3 parámetros que son la cara del objeto 3D a la que se desea aplicar estos parámetros del material, el tipo de propiedad que se desea especificar del material y sus valores correspondientes.

**glLightfv()**: Este comando se utiliza para crear y especificar los distintos parámetros de las luces. Recibe 3 parámetros que son el identificador de la luz que se está creando, el tipo de propiedad que se desea especificar de la luz y sus valores correspondientes.

Una vez se han definido las propiedades de los materiales y se ha creado una luz (GL\_LIGHT) hay que activar la iluminación a través del comando glEnable(GL\_LIGHTING). Además hay que activar la luz específica que hemos creado a través del comando glEnable(GL\_LIGHT0).

## Conceptos sobre representación de objetos en 3D

Como ya hemos adelantado antes, el siguiente ejemplo de mezcla vamos a realizar a partir de varios objetos 3D para buscar esa fusión tridimensional de la que hemos hablado en el apartado anterior. Para eso será necesario crear objetos en 3D lo cual se hace de una manera diferente a como se realiza en 2D.

Para poder crear objetos en 3D vamos a utilizar las llamadas listas de visualización. Para realizar este manejo de las listas de visualización tenemos las siguientes comandos:

* **glGenLists()**: Este comando se utiliza para solicitar un índice (o identificador entero) el cual se va a utilizar para posteriormente crear una lista de visualización. Cada lista de visualización se asocia con un índice entero por lo que antes de crear la lista se debe solicitar un índice. Este sistema se utiliza para evitar usar o eliminar listas no deseadas.

* **glNewList() y glEndList()**: Estos comandos se utilizan para definir el inicio y fin de una lista de visualización. En glNewList() se pasa como parámetro el índice obtenido con el comando **glGenLists()** que identifica nuestra lista.

* **glutSolidSphere()** y **glutSolidCube()**: Estas funciones específicas de la biblioteca GLUT se utilizan para generar una esfera y un cubo respectivamente. A través de los parámetros de estas funciones se pueden especificar las dimensiones de la esfera y el cubo.

* **glCallList()**: Este comando se utiliza para invocar la lista de visualización especificada como parámetro. Para especificar qué lista de visualización se desea ejecutar se utiliza el índice asociado a esa lista de visualización.

Con estos comandos básicos ya podemos dibujar una esfera y un cubo en la ventana correspondiente. Ahora, tenemos que colocar el cubo y la esfera en la posición deseada y con la configuración de colores deseada. Esto se va a realizar de forma muy diferente a como se realizaba en 2D ya que esto se va a realizar a partir de transformaciones.

Las transformaciones básicas que se pueden realizar en OpenGL son las siguientes:

* **glTranslatef()**:Este comando se utiliza para establecer las coordenadas xyz en las que se va a situar el objeto correspondiente.
* **glRotatef()**: Este comando se utiliza para establecer el ángulo de rotado y las coordenadas xyz del vector de rotación del objeto correspondiente.
* **glMaterialfv()**: Este comando se utiliza para especificar parámetros de material para el modelo de iluminación.

Al realizar una transformación se multiplica el valor de la matriz del modelo de vista (que está seleccionada por defecto) por la matriz de la transformación correspondiente. De esta manera, si queremos realizar transformaciones distintas sobre objetos diferentes debemos tener un mecanismo que nos permite recuperar el valor inicial de la matriz para que así no se mezclen las transformaciones.

Para ello OpenGL utiliza una pila de matrices como método para guardar el estado anterior de la matriz. Para operar en esta pila de matrices se tienen 2 comandos básicos:

* **glPushMatrix()**: Este comando guarda la matriz actual en la pila de matrices (en la primera posición).
* **glPushMatrix()**: Este comando extrae la primera matriz de la pila de matrices y le asigna esta valor a la matriz actual.

De esta manera, si queremos realizar transformaciones distintas para 2 objetos distintos se seguirá el siguiente esquema:

glPushMatrix(); // Establecer matriz actual en la pila

//tranformaciones1

//dibujado del objeto1

glPopMatrix(); // Extraiga la matriz antigua sin las transformaciones.

glPushMatrix(); // Establece la matriz actual en la pila

//tranformaciones2

//dibujado del objeto2

glPopMatrix(); // Extraiga la matriz antigua sin las transformaciones.

El objetivo básicamente es guardar el estado de la matriz inicial, realizar las transformaciones, dibujar el objeto y repercusiones el valor de la matriz inicial y así repetir para cada objeto al que se desea aplicar transformaciones distintas.

## Segundo ejemplo

Una vez hemos visto los conceptos básicos sobre representación de figuras en 3D y la utilización del buffer de profundidad para hacer mezclas vamos a realizar un segundo ejemplo más complejo en el que se van a hacer uso del buffer de profundidad para hacer mezclas.

Este ejemplo va a consistir en el dibujado de 2 objetos 3D (esfera y cubo) con distintos colores (azul y amarrilo) donde la esfera será un objeto opaco y el cubo un objeto translúcido. La idea de este ejemplo es colocar estos dos objetos uno delante del otro para ver cual es el efecto que se produce. Con un par de botones del teclado podemos cambiar el orden en el que se posicionan los objetos en la ventana.

Como ya vimos en el apartado teórico del buffer de profundidad, si la esfera opaca se coloca delante del cubo translúcido, la parte del cubo translúcido que se encuentra detrás no se dibujara ya que no se va a ver. Sin embargo, si el cubo translúcido se encuentra delante de la esfera opaca, se producirá un efecto de mezcla entre la parte del cubo y la parte de la esfera que se encuentran “superpuestos”.

El código para realizar esta tarea (obviando la creación de la ventana) será el siguiente:

#include <stdlib.h>

#include <stdio.h>

#include <GL/gl.h>

#include <GL/glu.h>

#include <GL/glut.h>

#define MAXZ 8.0

#define MINZ -8.0

**static** **float** solidoZ = MAXZ;

**static** **float** transparenteZ = MINZ;

**static** GLuint listaEsfera, listaCubo;

**static** **void** init(**void**)

{

   GLfloat mat\_especular[] = { 1.0, 1.0, 1.0, 1.0 };

   GLfloat mat\_shininess[] = { 100.0 };

   GLfloat posicion[] = { 0.5, 0.5, 1.0, 0.0 };

   glMaterialfv(GL\_FRONT, GL\_SPECULAR, mat\_especular);

   glMaterialfv(GL\_FRONT, GL\_SHININESS, mat\_shininess);

   glLightfv(GL\_LIGHT0, GL\_POSITION, posicion);

   glEnable(GL\_LIGHTING);

   glEnable(GL\_LIGHT0);

   glEnable(GL\_DEPTH\_TEST);

   listaEsfera = glGenLists(1);

   glNewList(listaEsfera, GL\_COMPILE);

      glutSolidSphere(0.4, 16, 16);

   glEndList();

   listaCubo = glGenLists(1);

   glNewList(listaCubo, GL\_COMPILE);

      glutSolidCube(0.6);

   glEndList();

}

**void** display(**void**)

{

   GLfloat mat\_solido[] = { 0.75, 0.75, 0.0, 1.0 };

   GLfloat mat\_cero[] = { 0.0, 0.0, 0.0, 1.0 };

   GLfloat mat\_transparente[] = { 0.0, 0.8, 0.8, 0.6 };

   GLfloat mat\_emision[] = { 0.0, 0.3, 0.3, 0.6 };

   glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

   glPushMatrix();

      glTranslatef(-0.15, -0.15, solidoZ);

      glMaterialfv(GL\_FRONT, GL\_EMISSION, mat\_cero);

      glMaterialfv(GL\_FRONT, GL\_DIFFUSE, mat\_solido);

      glCallList(listaEsfera);

   glPopMatrix();

   glPushMatrix();

      glTranslatef(0.15, 0.15, transparenteZ);

      glRotatef(15.0, 1.0, 1.0, 0.0);

      glRotatef(30.0, 0.0, 1.0, 0.0);

      glMaterialfv(GL\_FRONT, GL\_EMISSION, mat\_emision);

      glMaterialfv(GL\_FRONT, GL\_DIFFUSE, mat\_transparente);

      glEnable(GL\_BLEND);

      glDepthMask(GL\_FALSE);

      glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE);

      glCallList(listaCubo);

      glDepthMask(GL\_TRUE);

      glDisable(GL\_BLEND);

   glPopMatrix();

   glFlush();

}

**void** reshape(**int** w, **int** h)

{

   glViewport(0, 0, (GLint) w, (GLint) h);

   glMatrixMode(GL\_PROJECTION);

   glLoadIdentity();

**if** (w <= h)

      glOrtho (-1.5, 1.5, -1.5\*(GLfloat)h/(GLfloat)w,

             1.5\*(GLfloat)h/(GLfloat)w, -10.0, 10.0);

**else**

      glOrtho (-1.5\*(GLfloat)w/(GLfloat)h,

             1.5\*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);

   glMatrixMode(GL\_MODELVIEW);

   glLoadIdentity();

}

**void** keyboard(**unsigned** **char** key, **int** x, **int** y)

{

**switch** (key) {

**case** 'a':

**case** 'A':

         solidoZ = MINZ;

         transparenteZ = MAXZ;

         glutPostRedisplay();

**break**;

**case** 'r':

**case** 'R':

         solidoZ = MAXZ;

         transparenteZ = MINZ;

         glutPostRedisplay();

**break**;

**case** 27:

        exit(0);

    }

}

Como se puede observar, en este ejemplo la función **init()** tiene muchas más tareas que en el ejemplo anterior. En primer lugar, a partir de la función **glMaterialfv()** se va a definir la reflectancia y el brillo de la cara frontal de todos los materiales que se van a crear. En este caso, la reflectancia de los objetos será máxima (todos los valores RGBA a 1) y el brillo será bastante alto (el valor máximo es 128 y en nuestro caso es 100).

En segundo lugar, se crea y establece la posición de la luz a partir de la función **glLightfv()**. En nuestro caso, colocaremos la luz en la esquina superior derecha. Una vez hecho esto, hay que activar la iluminación a través del comando **glEnable(GL\_LIGHTING)**. Además hay que activar la luz específica que hemos creado a través del comando **glEnable(GL\_LIGHT0)**. También se utiliza el comando **glEnable(GL\_DEPTH\_TEST)** para activar el uso del buffer de profundidad.

En último lugar, se definen 2 listas de visualización para especificar la creación de la esfera y el cubo respectivamente ya que estos objetos no se dibujaran hasta que se llame a la función **glCallList()**. Como ya hemos visto, en los aspectos teóricos, primero se obtiene un índice entero para esa lista a través de la función **glGenLists()** y después se define la lista de visualización con **glNewList()** (utilizando el índice de la lista como parámetro) y **glEndList()** especificando en el cuerpo cual va a ser el contenido de esta lista de visualización.

En cuanto a la función **display** (manejadora de los eventos de dibujado) se harán las acciones necesarias para dibujar los 2 objetos en la ventana con las características y transformaciones adecuadas.

En primer lugar, en el comando **glClear()** en el que se especifica el borrado de la pantalla, se incluya la macro GL\_DEPTH\_BUFFER\_BIT para inciar los valores del buffer de profundidad al valor más lejano.

En segundo lugar, se definen las transformaciones para la esfera y el cubo utilizando los comandos **glPushMatrix()** y **glPopMatrix()** para evitar que las transformaciones se mezclen tal como explicamos antes.

En la esfera se realizan 2 transformaciones: La primera es para desplazar la esfera ligeramente hacia la esquina inferior izquierda. La segunda transformación es para establecer la intensidad de luz emitida y la reflectancia difusa de la cara frontal de la esfera. Obsérvese como el valor alfa en los 2 casos es 1 lo que nos indica que el material es opaco.

En el cubo se realizan muchas transformaciones: La primera es para desplazar la esfera ligeramente hacia la esquina superior derecha. La segunda transformación sirve para rotar el cubo ligeramente para que se puedan ver 3 de las caras del cubo. La tercera transformación es para establecer la intensidad de luz emitida y la reflectancia difusa de la cara frontal de la esfera. Obsérvese como el valor alfa en los 2 casos es 0.6 lo que nos indica que el material es translúcido.

Antes de realizar el dibujo del cubo, se activa el estado de mezcla y se establecen los buffers de profundidad en solo lectura. Esto se realiza para comparar la profundidad de este objeto translúcido con el opaco de forma que si el objeto translúcido se encuentra detrás del opaco no se dibuja y si el objeto translúcido se encuentra delante del opaco entonces se produce un efecto de mezcla utilizando para esta mezcla los factores de mezcla especificados en la función **glBlendFunc()**. En nuestro caso el factor de destino es la esfera opaca (GL\_ONE) y el factor de origen es el cubo translúcido (GL\_SRC\_ALPHA). Una vez se ha dibujado el cubo, se vuelve a permitir la escritura el el buffer de profundidad y se desactiva el estado de mezcla.

Por otra parte, en cuanto a la gestión del redimensionado de la ventana, la función manejadora reshape es muy parecida al ejemplo anterior. El único cambio significativo es la utilización del comando **glOrtho()** para definir un volumen de visualización ahora en 3D utilizando rectas de recortado. En 2D sólo se utilizaban 4 rectas de recortado pero ahora al tener 3 dimensiones se utilizan 6 rectas de recortado.

Si la pestaña es más alta que ancha se recortará por arriba y abajo y si la pestaña es más ancha que alta se recortará por la derecha y la izquierda.

En cuanto la función **keyboard** que gestiona los eventos del teclado, básicamente se evalúa si la tecla pulsada es una A o una R (tanto mayúscula como minúscula).

En el caso de que se pulse la tecla R se definirá la profundidad de forma que la esfera opaca será menos profunda (z = 8) que el cubo transparente (z = -8) y se forzara el redibujado a través de la función **glutPostRedisplay()**.

En el caso de que se pulse la tecla A se definirá la profundidad de forma que la cubo translúcido será menos profundo (z = 8) que la esfera opaca (z = -8) y se forzara el redibujado a través de la función **glutPostRedisplay()**.

Como nota adicional, observe como en la creación de la ventana con la función **glutInitDisplayMode()** se ha especificado la macro GLUT\_DEPTH para invocar una ventana con buffer de profundidad.

En las siguientes capturas se adjuntará el resultado del ejercicio cuando la esfera está por delante del cubo (Figura 7) y cuando el cubo está por delante de la esfera (Figura 8):

Imagen que contiene Logotipo

Descripción generada automáticamente

1. **Representación de la esfera opaca por delante del cubo translúcido**

Logotipo

Descripción generada automáticamente

1. **Fusión del cubo translúcido con la esfera opaca**

Como se puede observar, cuando la esfera opaca está por delante del cubo translúcido, la parte del cubo que está superpuesta con la esfera no se ve y además no se habrá dibujado por que el buffer de profundidad ha determinado que el cubo es más profundo que la esfera por lo que solo dibuja en esa parte la esfera.

En cambio, cuando el cubo translúcido está delante de la esfera opaca, se producirá una fusión entre la parte del cubo y la parte de la esfera que está superpuesta. Como los buffers de profundidad se congelaron (modo de solo lectura) en el dibujado del cubo entonces no se borra la parte de la esfera que está superpuesta con el cubo ya que el buffer de profundidad no se ha podido actualizar.

# Niebla

## Introducción al concepto de niebla

El uso de la niebla en Opengl tiene un concepto muy simple, vamos a trabajar con escalas grises para generar este efecto. Para hacer esto posible, se usarán distintas funciones que se aplicarán a cada píxel dependiendo de su distancia respecto al ojo. Un píxel el cual esté más alejado de la cámara tendrá menos visibilidad que uno que esté más cerca. Ahí es donde entran las escalas grises antes mencionadas. Cuando un píxel sea más lejano, se le aplicará un tono gris más denso. Al contrario, cuanto más cerca esté el píxel, más ligero será el tono de gris.

También debemos aclarar que no es obligatorio usar solo el color gris, por ejemplo, si queremos hacer un efecto de noche en un bosque, podríamos usar el color negro en vez del gris para crear este otro tipo de atmosferas. El color es un valor que se puede modificar.

Otro factor que se puede modificar es la densidad de la niebla. Así podemos determinar la distancia a la que un objeto le afecta este efecto.

Gráfico, Gráfico de burbujas

Descripción generada automáticamente

1. **Previsualización del efecto de niebla**

En este pequeño ejemplo (Figura 9) se ve cómo un mismo objeto le afecta distinto peso del color gris por culpa de la distancia respecto al punto de vista.

La niebla es un recurso muy utilizado y que es una herramienta muy útil ya que nos permite poner un límite a la distancia a la cual podríamos dibujar un objeto. Esto ayuda al rendimiento y crea programas menos pesados de ejecutar. Un ejemplo muy famoso en el que se usó la niebla para cumplir este objetivo es el videojuego Silent Hills (Figura 10).

Una persona caminando en la calle

Descripción generada automáticamente con confianza media

1. **Videojuego Sillent Hills (1999)**

## Declaración de Fog en Opengl

Lo primero que debemos hacer para incluir el efecto de niebla es habilitar la función con **glEnable(GL\_FOG).**  Ya habilitado, podemos declarar los siguiente parámetros claves:

* El color de la niebla (**GL\_FOG\_COLOR**). Este se declara como un array de floats como ya hemos visto anteriormente con los colores.  En el caso del gris usaremos 0.5,0.5 y 0.5.
* La función con la que se aplica la niebla (**GL\_FOG\_MODE**). Explicaré este concepto y sus modos un poco más adelante.
* La distancia a la que los pixeles se les empieza a aplicar la niebla (**GL\_FOG\_START**) y a la distancia a la que la niebla a su valor máximo (**GL\_FOG\_END**).
* Densidad de la niebla (**GL\_FOG\_DENSITY**) es un valor que se asigna a la niebla y afecta a dos de los tres modos que veremos a continuación.

glEnable(GL\_FOG);

   {

      GLfloat fogColor[**4**] = {**0.5**, **0.5**, **0.5**, **1.0**};

      fogMode = GL\_EXP;

      glFogi (GL\_FOG\_MODE, fogMode);

      glFogfv (GL\_FOG\_COLOR, fogColor);

      glFogf (GL\_FOG\_DENSITY, **0.35**);

      glHint (GL\_FOG\_HINT, GL\_DONT\_CARE);

      glFogf (GL\_FOG\_START, **1.0**);

      glFogf (GL\_FOG\_END, **5.0**);

   }

## Modos de niebla

Se nos ofrecen tres tipos de niebla que corresponden a cómo se va a aplicar el efecto dependiendo de la distancia.

Gráfico, Gráfico de líneas

Descripción generada automáticamente

1. **Modos de niebla**

**GL\_Linear** → corresponde con una función lineal muy simple. El factor f (el cual se aplicará más tarde al peso del color gris) se calcula con la división del punto final de niebla menos el píxel respecto al punto de vista dividido entre  el punto final menos el punto inicial.

***F= end-z / end-start***

Este es el modo más simple y por tanto al aplicarse crea un efecto poco dinámico. Para corregirlo, se usarán fórmulas las cuales contienen el factor de densidad de niebla.

**GL\_EXP** → usaremos una función exponencial que contenga los valores de la distancia del píxel y de la densidad de la niebla.

***F= e^-(density\*z)***

Este valor es mucho más preciso que el que nos ofrecía la función lineal. Aún así OpenGL nos ofrece un modo aún más refinado.

**GL\_EXP2** → Prácticamente idéntico al anterior solo que en este caso la densidad\*distancia se eleva al cuadrado.

***F= e^-(density\*z)^2***

Poniéndolo todo en práctica tenemos los siguientes valores de f si ponemos la distancia entre [1-5] ,vamos a valorar un objeto que está  a distancia 3 y con una densidad de niebla de 0.35

***f (GL\_linear)= 0.5***

***f (GL\_Exp) = 0.35***

***f(GL\_Exp2 )=  0.332***

Este factor se puede aplicar de dos maneras distintas, dependiendo de cómo tratemos los colores.

Con RGB primero se debe hacer la suma entre el color del pixel que estamos tratando\* f más la el color de la niebla \* 1-f. De esta forma se suman los dos colores para formar el resultado aplicado con niebla.

***C = f Ci + (1 - f ) Cf***

La segunda forma es con índices de colores los cuales se deben aplicar anteriormente tanto a los objetos como a la niebla con el modo ***GL\_FOG\_INDEX .*** Aunque está forma de utilizar niebla es interesante, me centraré mejor en el modo RGB , ya que es el formato más común de usar colores.

## Ejemplo práctico

A continuación pondremos un ejemplo en cual aplicamos los distintos modos de niebla para ver cómo afectan a un mismo objeto que se repite en distintas posiciones.

glEnable(GL\_FOG);

   {

      GLfloat fogColor[**4**] = {**0.5**, **0.5**, **0.5**, **1.0**};

      fogMode = GL\_EXP;

      glFogi (GL\_FOG\_MODE, fogMode);

      glFogfv (GL\_FOG\_COLOR, fogColor);

      glFogf (GL\_FOG\_DENSITY, **0.35**);

      glHint (GL\_FOG\_HINT, GL\_DONT\_CARE);

      glFogf (GL\_FOG\_START, **1.0**);

      glFogf (GL\_FOG\_END, **5.0**);

   }

Lo primero que hacemos es declarar todos los valores del ejemplo anterior.

void keyboard(unsigned char key, int x, int y)

{

   switch (key) {

      case ‘f’:

      case ‘F’':

**if** (fogMode == GL\_EXP) {

            fogMode = GL\_EXP2;

            printf ("Fog mode is GL\_EXP2**\n**");

         }

**else** **if** (fogMode == GL\_EXP2) {

            fogMode = GL\_LINEAR;

            printf ("Fog mode is GL\_LINEAR**\n**");

         }

**else** **if** (fogMode == GL\_LINEAR) {

            fogMode = GL\_EXP;

            printf ("Fog mode is GL\_EXP**\n**");

         }

         glFogi (GL\_FOG\_MODE, fogMode);

         glutPostRedisplay();

**break**;

      case **27**:

         exit(**0**);

**break**;

      default:

**break**;

   }

}

Lo siguiente que haremos es crear una función la cual cada vez que pulsemos la tecla “F” cambiará su modo de niebla.

 El resultado que obtenemos es el siguiente:

Una imagen de un videojuego

Descripción generada automáticamente con confianza baja

1. **Modo lineal**

Un dibujo de una persona

Descripción generada automáticamente con confianza media

1. **Modo exponencial**

Un dibujo de una persona

Descripción generada automáticamente con confianza media

1. **Modo exponencial al cuadrado**

Podemos observar que las diferencias son muy sutiles ya que estamos en unos ejemplos muy concretos en los cuales los distintos modos no nos dejan ver muchas diferencias. Antes de ver esto con otros ejemplos, vamos a variar la propiedad de  densidad de niebla.



Imagen que contiene alimentos

Descripción generada automáticamente

1. **Ejemplo con cambio de densidad**

Podemos ver que al incrementar la densidad, la niebla es mucho más acentuada.

Ahora veremos un segundo ejemplo , en el que no solo usaremos un mismo objeto a diferentes distancias, sino que usaremos un objeto en 3d el cual está animado.

Imagen que contiene estacionaria, sobres

Descripción generada automáticamente Una caricatura de una persona

Descripción generada automáticamente con confianza media

1. **Ejemplo exponencial-exponencial^2 con animación**

En este segundo ejemplo, podemos afirmar como realmente los cambios no son tan visibles con los dos modos exponenciales. Así que realmente para proyectos grandes en los que usemos niebla, los tres modos pueden ser igual de válidos.

# Reflejos

## Introducción al concepto de reflejo El efecto de reflejo en Opengl consistirá en jugar con dos ideas. La primera será redibujar el objeto u objetos que queremos que se reflejen en una superficie en concreto. La segunda idea es usar los efectos de fusión ya explicados anteriormente.

Empezaremos explicando dos nuevos conceptos para entender cómo redibujar el objeto reflejado: Stencil Test y Stencil Buffer.

## Stencil Test y Stencil Buffer

A la hora de renderizar un objeto con Opengl , se le pueden aplicar como ya hemos visto múltiples procesos. Después de pasar el objeto por el Fragment Shader, se le puede aplicar el testeo Stencil para descartar una serie pixeles los cuales se encuentran marcados en el Stencil Buffer.

Dentro de este buffer nos encontramos un conjunto de 1 y 0. Al renderizar el Stencil test solo se renderizan los píxeles de la imagen original que coincidan con las casillas que tengan un 1 en el stencil buffer.

Interfaz de usuario gráfica

Descripción generada automáticamente

1. **Stencil Test y Buffering**

Para usar el test y el buffer debemos seguir los siguientes pasos:

* Limpiar el buffer de posibles usos anteriores.
* Habilitar la escritura del buffer
* Dar nuestras medidas y lo que queremos que aparezca en el buffer
* Deshabilitar la escritura del buffer
* Renderizar los objetos usando el testeo.

Para habilitar y deshabilitar el testeo usaremos  **glEnable(GL\_STENCIL\_TEST)** y

**glDisable(GL\_STENCIL\_TEST)**, los cuales ya los hemos visto y utilizado múltiples veces.

También usaremos **glClear(...GL\_STENCIL\_BUFFER\_TEST)** para limpiar el buffer en un primer momento.  
  
OpenGL nos da una diversa cantidad de acciones con las que nuestro objeto interactúa con el Stencil Buffer. Para manejar dichas acciones, tendremos dos funciones :  **glStencilFunc** y **glStencilOp**.

La primera función ,**glStencilFunc(GLenum func, GLint ref, GLuint mask),** contiene los 3 siguientes parámetros:

* func : Es el tipo de función que se aplica dentro del testeo. Podemos ver las distintas funciones en la siguiente tabla.
* ref: valor que se da como referencia para el test.
* mask : máscara que se aplica al final de func  y ref como un AND.

|  |  |  |
| --- | --- | --- |
|  | **Enum** | **Condición** |
|  | GL\_NEVER | Siempre falla el test |
|  | GL\_LESS | (ref&mask)<(stencilbuff&mask) |
|  | GL\_LESS | (ref&mask)>(stencilbuff&mask) |
|  | GL\_GREATER | (ref&mask)>(stencilbuff&mask) |
|  | GL\_EQUAL | (ref&mask)=(stencilbuff&mask) |
|  | GL\_ALWAYS | Siempre se supera el test |
|  | GL\_LEQUAL | (ref&mask)≤(stencilbuff&mask) |
|  | GL\_GEQUAL | (ref&mask)≥(stencilbuff&mask) |
|  | GL\_NOTEQUAL | (ref&mask)≠(stencilbuff&mask) |

1. **Opciones de glStencilFunc()**

Si por ejemplo queremos usar el test para que pasen los píxeles que coinciden con el buffer en las posiciones que haya un 1 usaremos la siguiente orden: **glStencilFunc(GL\_EQUAL, 1, 1);**

La segunda función, **glStencilOp(GLenum sfail, GLenum dpfail, GLenum dppass),** se encargará más sobre las opciones del buffer y que acciones tomar sobre el test. La función contiene los 3 siguientes parámetros:

* dpfail : Acción a tomar si el  Stein Test falla
* dpfial : Acción a tomar si el Depth Test falla, pero el stein acierta.
* dppass: Acción a tomar si se aciertan tanto el Stein como el Depth test.

Las opciones que podemos usar son las siguientes:

|  |  |  |
| --- | --- | --- |
|  | **Acción** | **Descripción** |
|  | GL\_KEEP | El testeo no hace ningún cambio. |
|  | GL\_ZERO | El testeo transforma el valor a 0 del buffer. |
|  | GL\_REPLACE | El valor del buffer se cambia por ref, el cual se declaró en la función de glStencilFunc |
|  | GL\_INCR | El valor del buffer aumenta por uno sino es el máximo posible. |
|  | GL\_INCR\_WRAP | Igual que el anterior, solo que cuando se llega al máximo, el valor cambia a 0. |
|  | GL\_DECR | El valor de buffer disminuye por uno hasta un mínimo posible. |
|  | GL\_DECR\_WRAP | Igual que el anterior, solo que cuando se llega al mínimo, el valor cambia al máximo. |
|  | GL\_INVERT | Invierte los valores del buffer a nivel de bit |
|  | GL\_KEEP | El testeo no hace ningún cambio. |

1. **Opciones de glStencilOP()**

## Ejemplo práctico

Veremos ahora cómo podemos aplicar todos estos conceptos adquiridos en un ejemplo práctico. Vamos a usar el código del segundo ejemplo de niebla, el del  cubo que tiene una animación de girar, para aplicarle un reflejo. Lo primero será definir igual que un cubo , el suelo en el que queremos que queremos que se refleje.

Lo siguiente es dibujar la escena con las funciones que hemos visto anteriormente:

void drawScene() {

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT | GL\_STENCIL\_BUFFER\_BIT);

/….

/… Dibujamos el cubo

drawCube(\_textureId, \_angle);

glPopMatrix();

De momento solo hemos limpiado el stencil buffer para comenzar desde 0. Además dibujaremos el primer cubo (el cual no es el reflejo).

glEnable(GL\_STENCIL\_TEST);

glColorMask(0, 0, 0, 0); //Disable drawing colors to the screen

glDisable(GL\_DEPTH\_TEST); //Disable depth testing

glStencilFunc(GL\_ALWAYS, 1, 1); //Make the stencil test always pass

//Make pixels in the stencil buffer be set to 1 when the stencil test passes

glStencilOp(GL\_KEEP, GL\_KEEP, GL\_REPLACE);

//Set all of the pixels covered by the floor to be 1 in the stencil buffer

drawFloor(\_textureId);

 Nos interesa poner en nuestro suelo todo lleno de 1 para que se pueda aplicar el Stencil Test sobre el segundo cubo. Para ello seguimos los siguientes pasos.Primero, vamos a habilitar **GL\_STENCIL\_TES**T, y deshabilitamos tanto  el **DEPTH\_TEST** como los colores de la pantalla(solo queremos dibujar el stencil buffer , no nos interesa pintar nada de colores ni de la pantalla).

Lo siguiente es usar las dos opciones de Stencil : **glStencilFunc** (se pondrá que el test se supera siempre y ponemos el valor de referencia y de máscara a 1) y **glStencilOp(**pondremos que solo se sustituyan el valor del stencil buffer por 1 cuando se superen tanto el depth\_test como stencil\_test).   
El último paso será dibujar el suelo. Al hacerlo vamos a ir rellenando el stencil buffer con 1 en todos los espacios que hayamos pintado. Ya tenemos preparado el espacio donde va a ser representado el reflejo.

glColorMask(1, 1, 1, 1); //Enable drawing colors to the screen

glEnable(GL\_DEPTH\_TEST); //Enable depth testing

//Make the stencil test pass only when the pixel is 1 in the stencil buffer

glStencilFunc(GL\_EQUAL, 1, 1);

glStencilOp(GL\_KEEP, GL\_KEEP, GL\_KEEP); //Make the stencil buffer not change

//Draw the cube, reflected vertically, at all pixels where the stencil

//buffer is 1

glPushMatrix();

glScalef(1, -1, 1);

glTranslatef(0, BOX\_HEIGHT, 0);

drawCube(\_textureId, \_angle);

glPopMatrix();

Lo segundo que vamos a hacer es dibujar el segundo cubo (que representa el reflejo del primero). Vamos a activar tanto la mascara de color como el depth test ya que los necesitaremos para poder dibujar el cubo. A la hora de usar el stencil test , marcaremos que solo vamos a dibujar los lugares donde el valor del buffer sea 1. Por tanto pondremos en glStencilFunc el valor de GL\_EQUAL. En el glStencilOp mantendremos todos los valores iguales ya que no nos interesa cambiar el buffer. Ya con esto dibujamos el segundo cubo, el cual será igual menos en escala para  que de el efecto de reflejo (hay una distancia entre el espejo y el cubo).

glDisable(GL\_STENCIL\_TEST); //Disable using the stencil buffer

//Blend the floor onto the screen

glEnable(GL\_BLEND);

glColor4f(1, 1, 1, 0.7f);

drawFloor(\_textureId);

glDisable(GL\_BLEND);

glutSwapBuffers();

}

Por último vamos a dibujar el suelo, el cual no nos interesa aplicarle nada de las operaciones de stencil. Por tanto solo vamos a deshabilitar esta opción. A su vez vamos a aplicarle el valor 0.7 de alfa para que se pueda fusionar el suelo con el cubo y así poder tener el efecto de reflejo.

El resultado de este ejemplo es el siguiente :

Una caricatura de una persona

Descripción generada automáticamente con confianza baja

1. **Efecto de reflejo**

# Efecto de partículas

## Introducción al concepto de partículas: Alfa Chanel

 Para crear el efecto de partículas se pueden utilizar muchos recursos y formas. La que vamos a explorar y la que tiene mucho que ver con los valores alfa es Alpha Channel.   
La idea será crear una imagen la cual al pasarla por este canal, a cada píxel se le aplicará un valor de alfa equivalente a la del canal. Por así decirlo, será como aplicar un filtro de alfa a cada pixel de la imagen. Este es un método para generar el efecto de partícula.

 El canal alfa se puede encontrar en la propia imagen (en el formato png) o en dos imágenes separadas como veremos en el formato .bmp.

## Ejemplo práctico

Vamos a incluir dentro de nuestro proyecto dos imágenes bmp, una que represente el color puro y otra que represente el canal alfa.

Imagen que contiene Texto

Descripción generada automáticamente

1. **Alpha Channel**

Podemos ver como solo se utilizan los colores blanco, negero y diferentes tonos de gris. Estos colores representan el valor alfa, siendo el blanco un valor 1f (completamente opaco) y el negro el valor 0f (completamente transparente). Con esta idea entendemos que si un objeto se aleja más del centro , su valor alfa decrece.

La forma de hacerlo es parecida a como cargamos una imagen con el bitmap en OpenGL, vamos a ir recorriendo la imagen pixel a pixel y pondremos en un array de color cual es su RGB. Ahora, además de ese primer recorrido, vamos a comparar cada casilla del array con el Alpha Channel para darle el valor alfa correspondiente segundo su escala de grises.   
Al tratarse de una escala de grises podemos coger el valor R, G o B sin diferencias.

Vamos a ver cómo se aplica esto dentro del código.

**char**\* addAlphaChannel(Image\* image, Image\* alphaChannel) {

    char\* pixels = new char[image->width \* image->height \* 4];

    for(int y = 0; y < image->height; y++) {

        for(int x = 0; x < image->width; x++) {

            for(int j = 0; j < 3; j++) {

                pixels[4 \* (y \* image->width + x) + j] =

                    image->pixels[3 \* (y \* image->width + x) + j];

            }

            pixels[4 \* (y \* image->width + x) + 3] =

                alphaChannel->pixels[3 \* (y \* image->width + x)];

        }

    }

    return pixels;}

En esta parte del código vemos cómo se van a recorrer las dos imágenes para primero darle el valor RGB de la imagen original y después ver que valor alpha tiene cada pixel respecto al alphaChannel. En este caso podemos ver como coje el valor R de la imagen del AlphaChannel para saber cual es el valor de alfa.

Lo siguiente será cargar dicha imagen con el canal de alfa. Para ello vamos a crear una nueva función de loadTexture.

GLuint loadAlphaTexture(Image\* image, Image\* alphaChannel) {

    char\* pixels = addAlphaChannel(image, alphaChannel);

    GLuint textureId;

    glGenTextures(1, &textureId);

    glBindTexture(GL\_TEXTURE\_2D, textureId);

    glTexImage2D(GL\_TEXTURE\_2D,

                 0,

                 GL\_RGBA,

                 image->width, image->height,

                 0,

                 GL\_RGBA,

                 GL\_UNSIGNED\_BYTE,

                 pixels);

    delete pixels;

    return textureId;

}

Esta función es muy similar a los ladtexture anteriores, solo que aquí vamos a llamar antes a la función anterior de asignar el RGBA a cada pixel.   
Al estar usando blending, vamos a tener que habilitar como ya sabemos en la función de initRendering, **glEnable(GL\_BLEND).**

Vamos a explorar un poco ahora el comportamiento de cómo funcionan las partículas. En el caso de nuestro código, vamos a crear una estructura para cada partícula la cual contiene el color, la velocidad, la posición y dos timers (uno de tiempo de vida y otro que sea el máximo tiempo que puede vivir). Con los timers estableceremos cómo se irá viendo la partícula con el paso del tiempo.

Lo siguiente serán las funciones de rotación de la partícula. Nosotros queremos que aparezcan las partículas en forma de fuente , así que lo que queremos es que roten. En este caso vamos a rotar la cámara en una posición de 30º. Además, podemos ver la segunda función que nos devolverá la posición de la partícula.

Vec3f rotate(Vec3f v, Vec3f axis, float degrees) {

    axis = axis.normalize();

    float radians = degrees \* PI / 180;

    float s = sin(radians);

    float c = cos(radians);

    return v \* c + axis \* axis.dot(v) \* (1 - c) + v.cross(axis) \* s;

}

Vec3f adjParticlePos(Vec3f pos) {

    return rotate(pos, Vec3f(1, 0, 0), -30);

}

Haremos uso de la posición de las partículas cada vez que detectemos que una partícula A se encuentra detrás de una partícula B.  
Por último hablaremos un poco de las variables globales de cómo funciona la animación de estas partículas. Estos parámetros son los siguientes:

const float GRAVITY = 3.0f;

const int NUM\_PARTICLES = 1000;

const float STEP\_TIME = 0.01f;

const float PARTICLE\_SIZE = 0.05f;

No me centraré en contar cómo funciona la animación , ya que simplemente lo que se hace es combinar las posiciones con la gravedad para así ir calculando con cada step(cada frame del programa) la posición de la partícula.   
También destaco como el numero de particulas es constante, esto se debe a que cuando una particula llega a su limite, vease llega al color negro en el alpha channel, debe morir para así crear una nueva particula.

La clase fundamental que usaremos será la de ParticleEngine, la cual contiene todo lo necesario para generar una partícula, asignarle una textura según el canal de alfa, también asignarle un color según el tiempo del step. Además de muchos cálculos que no creo que deban explicarse ya que son fórmulas matemáticas complejas para entender como la partícula va a moverse según la física.

Al final los conceptos más importantes que se deben de entender es como hemos asignado un tiempo de vida y un máximo de partículas que veremos en la pantalla.

Dicho todo esto , el resultado del código es el siguiente:

Imagen que contiene verde, cuarto

Descripción generada automáticamente

1. **Efecto de partículas en OpenGL**

# Bibliografía

<https://www.cs.buap.mx/~hilario_sm/slide/graphics-2018/opengl-Part-I.pdf>

<http://openglenfichas.uji.es/ejemplos/transf.pdf>

<http://openglenfichas.uji.es/ejemplos/ventana2d.pdf>

<https://eduarmandov.files.wordpress.com/2017/05/graphicscomputer-vistaopengl.pdf>

<http://www.videotutorialsrock.com/opengl_tutorial/fog/text.php>