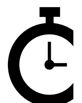
Generación de efectos en tiempo real de procesamiento

PID_00246087

Rafael Pérez Vidal

Tiempo mínimo de dedicación recomendado: 4 horas



Universitat Oberta de Catalunya

Ninguna parte de esta publicación, incluido el diseño general y la cubierta, puede ser copiada, reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea éste eléctrico, químico, mecánico, óptico, grabación, fotocopia, o cualquier otro, sin la previa autorización escrita de los titulares del copyright.

Índice

Introducción ... 5

1. Introducción a los gráficos por computador ... 7
1.1. Modo de vídeo ... 7
1.2. Memoria de vídeo ... 7
1.3. Modos de colores ... 8
1.3.1. Modo indexado o paleta (8 bits de color) ... 8
1.3.2. Modo directo ... 9
1.4. El doble *buffer* ... 9
1.5. La biblioteca SDL ... 10
1.6. Estrellas: ejemplo ... 10

2. Temporización ... 11
2.1. Sistemas de tiempo ... 11
2.2. Tiempo en SDL ... 11
2.3. Fotogramas por segundo ... 11
2.4. El precálculo ... 12
2.4.1. Cálculos en coma fija ... 12
2.4.2. Interpolación ... 13

3. Control del píxel ... 14
3.1. Efectos a pantalla completa ... 14
3.2. Efecto de plasma ... 14
3.2.1. Funciones sinusoidales ... 14
3.2.2. Precálculo ... 15
3.2.3. Colorear ... 16

4. Filtros ... 17
4.1. Matrices en filtros ... 17
4.2. Efecto de fuego ... 18

5. Distorsión de *bitmaps* ... 20
5.1. Distorsión ... 20
5.2. Filtros bilineales ... 20

6. *Bump mapping* ... 23
6.1. La teoría en el *bump mapping* ... 23
6.2. Mapa de luces ... 23

7. *Zoom* fractal ... 26
7.1. Números complejos ... 26

7.2. Mandelbrot 27
7.3. Teoría del *zoom* fractal 28
7.4. *Zoom* a *bitmaps* 28

8. Texturización estática 30
8.1. Concepto de textura 30
8.2. Mapeado 31

9. *Roto-zooming* 32
9.1. Teoría del *roto-zooming* 32
9.2. Rotaciones en 2D 32

10. Sistemas de partículas 34
10.1. El mundo en tres dimensiones. Introducción a matrices 34
10.2. Proyección 2D del mundo 3D 35
10.3. *Antialiasing* 36
10.4. Reescalado de imágenes 36

11. Motores de polígonos 37
11.1. Teoría del renderizado de polígonos por software 37
11.2. Cómo lo hace OpenGL. *Pipeline* 38

12. Texturización con corrección de perspectiva 41
12.1. Método de interpolación lineal 41
12.2. Las nueve constantes 42

13. Música y sincronización 43
13.1. Sincronía 43
13.2. SDL_Audio y SDL_Mixer 45

14. Acabado final 46
14.1. Transiciones 46
14.2. Contenido de la demo 47

Introducción

A largo de la historia de la programación gráfica por ordenador, en los años noventa del siglo pasado y la primera década de 2000, apareció un movimiento llamado *demoscene*. Surgido del entorno del *hacking* (en su definición original creativa), impulsó el reto de extraer el máximo provecho de los procesadores y de mostrar lo que eran capaces de hacer con ellos de manera gráfica y sonora.

En estos retos aprenderemos a hacer los clásicos efectos de la época: plasma, fuego, partículas, campos de estrellas 3D, *scrolls* de texto sinusoidales, *roto-zoomers*, túneles 3D o *blobs*, entre otros. Con esto, comprenderemos la base de ciertos aspectos visuales que los videojuegos de hoy en día han aprovechado.

El *demomaking*, hacer demos, originalmente proviene de la época de los *cracks*, programas que sacaban la protección de juegos que tenían claves de acceso o que permitían copiarlos. Al principio solo eran pantallas con dibujos hechos con texto, pero rápidamente se hicieron impresionantes efectos visuales. Apareció la rivalidad entre los diferentes grupos de *cracking* y se crearon competiciones para ver quién daba la impresión más espectacular, de manera que el tema de hacer el *crack* quedaba en un segundo plano.

Para este reto es necesario que se tengan conocimientos de programación en C ++ y una buena base de conocimientos matemáticos. Hace falta también tener mucha imaginación para que podáis ir construyendo vuestros propios efectos visuales.

Como compilador, trabajaremos sobre una base de proyecto de Visual Studio C++, pero como el código es C++ universal, podéis utilizar cualquier otro compilador de C++.

La biblioteca básica que utilizaremos para realizar gráficos será SDL, Simple DirectMedia Layer.

1. Introducción a los gráficos por computador

1.1. Modo de vídeo

En este reto trabajaremos con píxeles. Por lo tanto, es importante saber el modo de vídeo que utilizaremos. Un modo de vídeo define la resolución de pantalla o ventana que tendremos y la profundidad de color.

Hoy en día quizás no tiene demasiado sentido tener en cuenta la resolución de pantalla, puesto que se trabaja utilizando un modo 3D de representación gráfica a base de polígonos. Pero para nuestro reto nos interesa.

Contenido complementario

Hay muchos formatos de pantalla y resoluciones diferentes, y cada una tiene un nombre propio.

La resolución viene expresada con dos números: la anchura y la altura de píxeles de los que disponemos.

La profundidad de color es la cantidad de *bytes* que utilizaremos para representar cada píxel de pantalla. Existen típicamente dos modos: color indexado (paleta) o color directo.

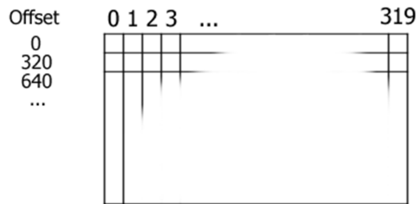
1.2. Memoria de vídeo

Todo modo de vídeo tiene su propia resolución y hay que conocerla para hacer uso de él. Para llevar a cabo nuestros retos, utilizaremos una medida mediana de resolución, típica de formato 4:3, de 640 × 480 píxeles.

La biblioteca SDL funciona de dos maneras a la hora de pintar píxeles en pantalla: mediante *surfaces* o *textures*. En nuestro caso, basaremos los retos usando *surfaces* para poder tener un control más adecuado del píxel. Es cierto que es más lento, pero nos irá mejor para entender ciertos conceptos.

SDL simula el *buffer* de vídeo (*array* de dos dimensiones) en una estructura llamada SDL_Surface. A pesar de que habitualmente tendríamos que pensar que es un espacio bidimensional, lo cierto es que accederemos a él de manera lineal. Es decir, los píxeles están uno al lado del otro línea tras línea y numerados consecutivamente. Aquí presentamos un ejemplo si tuviéramos un modo de vídeo de 320 de ancho.

Imagen 1.



Por lo tanto, cuando accedamos al *buffer* de vídeo de la *surface*, lo haremos siguiendo esta fórmula.

```
Color = posición Y * amplitud de superficie + posición X
```

Veremos que cuando lo hacemos en SDL habrá ciertos cambios en el modo de calcular la anchura de la superficie (*surface*) en la que pintamos, y la profundidad de color también afectará (ved el código ejemplo del capítulo).

Hay que tener cuidado de no salirnos del espacio de memoria asignado por la superficie de pintado, puesto que intentar pintar un píxel fuera del área puede provocar errores o que el programa deje de funcionar. Esto se denomina ***clipping*** y es tan sencillo como que, antes de hacer el pintado, verifiquemos con una condición si nos salimos del espacio que tenemos.

```
if ((x>=0) && (x<=MAXRESX) && (y>=0) && (y<=MAXRESY))
```

1.3. Modos de colores

Tal como hemos mencionado antes, la profundidad de color nos marca cómo tenemos que representar cada píxel de pantalla en la memoria de vídeo del ordenador.

1.3.1. Modo indexado o paleta (8 bits de color)

Es una manera antigua de tratar el color. Se tienen 256 colores diferentes almacenados en una tabla. Cada color tiene un índice asignado e internamente el chip de vídeo interpreta cada índice con el color RGB correspondiente. Hoy en día no tiene uso directamente en las tarjetas actuales, pero se puede simular si queremos representar aspecto *pixelart* de verdad. También nos permite hacer algunos trucos y efectos de buen resultado, como la rotación de paleta, que se usaba para hacer animaciones.

Contenido complementario

La rotación de paleta se usó en muchos videojuegos. Ordenadores como AMIGA o AtariST hacían uso de ella en programas gráficos (videojuegos como Sim City o grandes aventuras gráficas para simular agua o lava volcánica).

Hoy en día es una técnica utilizada, como podemos ver en estos ejemplos de HTML5.

1.3.2. Modo directo

A diferencia del modo indexado, aquí cada píxel tiene toda la información que le hace falta para ser representado. Cada tono de R (rojo), de G (verde) y de B (azul) se pueden almacenar en 15 o 16 bits, antiguamente, por píxel, o usando 24 o 32 bits por píxel. A usar 32 bits (4 *bytes*) por píxel se le denomina true color. Realmente, con 3 *bytes* tendríamos suficiente, puesto que cada tono básico (RGB) puede tener un valor entre 0 y 255. Esto nos daría un total de 16.777.216 colores diferentes. El cuarto *byte* se usa para poner información de canal alfa, es decir, transparencia (habitualmente 0 quiere decir que el píxel es transparente y 255 que es completamente opaco).

Las tarjetas gráficas de hoy en día y los modos de vídeo que usamos son *true color*, de 32 bits.

1.4. El doble *buffer*

Las pantallas se actualizan en cada fotograma y se pintan con el contenido de la memoria de vídeo. La frecuencia con la que se actualiza depende del monitor y la tarjeta de vídeo. Habitualmente suele ser de unas sesenta veces por segundo, 60 fotogramas (*frames*). Esto nos da un tiempo de espera entre fotograma y fotograma de unos 17 milisegundos.

En el código ejemplo del capítulo, el control de tiempo y la sincronía de pintado los haremos con este código:

```
#define FPS 60
unsigned int lastTime = 0, currentTime, deltaTime;
float msFrame = 1/(FPS/1000.0f);
...
currentTime = SDL_GetTicks();
deltaTime = currentTime-lastTime;
if (deltaTime < (int)msFrame){
        SDL_Delay ( (int)msFrame – deltaTime);
}
lastTime = currentTime;
```

Mantendremos una constante de espera en función del tiempo que haya pasado entre cada paso de pintado de nuestro código sobre la base de los fotogramas por segundo que queramos mantener.

SDL ya controla por sí solo el tema del doble *buffer*. Pero intentaremos entender qué es esto.

Contenido complementario

¿Realmente, hace falta el doble *buffer*? Sí. Se usa tanto hoy en día como cuando había tarjetas gráficas VGA de 256 colores.

Cuando pintamos píxeles en pantalla, realmente no lo hacemos en el *buffer* que se está visualizando, sino en un segundo *buffer* clónico de medidas y formato que el que se muestra. En cada fotograma, se intercambia el *buffer* que se tiene que mostrar, con lo que se evita un temblor o distorsiones de imágenes que se pudieran dar. La función de SDL SDL_UpdateWindowSurface ya nos hace este trabajo.

1.5. La biblioteca SDL

Antes que nada, hace falta que preparemos un proyecto de C++ que nos inicie la ventana y que tengamos vinculados los ficheros de biblioteca SDL correctamente.

Reto 1

Utilizando la biblioteca SDL y siguiendo el tutorial de la web Lazy Foo, vamos a crear una ventana y el contexto de OpenGL necesario.

1.6. Estrellas: ejemplo

Vamos a representar un campo de estrellas del espacio. Cada estrella se representa con un píxel en pantalla y tiene una coordenada X,Y. Hay que mover las estrellas horizontalmente, y si una cae ya fuera de la pantalla, se regenera de nuevo al principio de la misma aleatoriamente en posición Y.

Reto 2

Recread el campo de estrellas con tonos de color gris en función de las velocidades de cada estrella.

Este efecto es muy básico. Siempre seguiremos el patrón de inicializar, actualizar y pintar (init, *update* y *render*).

Probad de experimentar en el código haciendo cambios, como por ejemplo, mover las estrellas en otras direcciones, cambiar los colores o la medida. Como sugerencia, si hiciéramos caer las estrellas de arriba abajo y les diéramos un movimiento aleatorio pequeño de derecha a izquierda, como un zarandeo, optimizaríamos un efecto de nieve que cae.

2. Temporización

2.1. Sistemas de tiempo

Cuando ejecutamos en un ordenador, no siempre podemos tener asegurado que el tiempo de ejecución sea el mismo en todos. Cada ordenador funciona a velocidades diferentes. Hay otros factores, como la tarjeta gráfica, que pueden hacer que vaya más o menos rápido. Como desarrolladores de efectos visuales gráficos, tenemos que procurar que el tiempo no sea un problema y que nuestro código siempre se ejecute mostrando el mismo.

Una de las maneras de hacer esto es controlar la ejecución dentro de unos intervalos regulares, como un reloj, y hacer que todos los cálculos de nuestro efecto estén en relación con este factor de tiempo. De este modo, el efecto siempre se verá igual en todos los *frame rates* ('velocidades de ejecución por fotograma').

2.2. Tiempo en SDL

Como hacemos uso de SDL, esta biblioteca nos proporciona un *wrapping* contra el sistema operativo en el que estemos con una función que nos da el tiempo que ha pasado en milisegundos desde el inicio de la aplicación. Se trata de la función SDL_GetTicks (ved ejemplo en la unidad 1).

Haciendo uso de esta función, podemos montar un sistema de espera de ejecución para simular el tiempo de *frame rate* y fijarlo a los FPS (*frames per second*) que queramos.

Hay otras maneras de hacer sincronía, como por ejemplo, mediante V-sync, la sincronía vertical de la tarjeta de vídeo. SDL puede hacerlo, pero solo cuando usamos texturas, es decir, con aceleración gráfica. En nuestro caso, en que lo hacemos por *surface*, no es posible.

2.3. Fotogramas por segundo

FPS es una unidad de medida de la cantidad de fotogramas por segundo que nuestro sistema es capaz de hacer.

A la hora de hacer videojuegos, se suelen exigir unos 60 fotogramas por segundo. Esto nos deja un tiempo de cálculo por cada fotograma de unos 17 milisegundos, que es muy poco, pero que hoy en día se puede conseguir.

Para nuestros objetivos, ir a 30 o incluso a 25 es aceptable. Pensad que estamos haciendo unos efectos visuales que no requieren interacción con el usuario, solo el visionado. Por lo tanto, no sufráis si os queda un efecto de menos de 60 FPS. Seguramente no se notará.

2.4. El precálculo

A la hora de hacer un efecto visual, es muy importante realizar ciertas partes de él como precálculo; justamente por lo que acabamos de comentar de hacer la ejecución dentro de unos FPS.

Hay que ser crítico con el tema del precálculo. Hoy en día no hay que realizar un precálculo de todo lo que pasará en el efecto. Por ejemplo, antiguamente hacer cálculos de trigonometría de senos y cosenos tenía un coste muy elevado en términos de tiempo de CPU. Actualmente no es muy necesario. En cambio, para otras cosas, como puede ser un filtro nuestro inventado, donde por cada píxel hubiera una serie de números que nos da un rango de valores fijos, podemos hacer un precálculo de estos valores. Lo haríamos previamente al principio del efecto, y los guardaríamos en una tabla. Así después el acceso a estos datos simula el efecto del cálculo. Siempre es más económico un acceso a un dato en una posición que realizar todo el cálculo.

Lo mejor es experimentar. Primero intentaremos hacerlo todo en tiempo real, y si vemos que el efecto sufre consumo de tiempo y caen los FPS, miraremos qué podemos precalcular.

2.4.1. Cálculos en coma fija

Una de las fuentes de pérdida de velocidad son los cálculos con decimales que podemos hacer por segundo. Las CPU actuales son rápidas al hacer cálculos, pero todavía es más rápido si los cálculos se hacen con números enteros. El problema viene cuando necesitamos tener decimales de alguna manera.

Existe una técnica llamada coma fija que nos permite expresar el dato de decimales en un formato entero. La idea es reservar una cantidad de cifras para el valor entero y otra para el valor decimal.

Imaginad que queréis guardar un valor que fuera de –128 a 127 y que pudiese tener decimales. En este caso, usaríamos un tipo de dato contenedor más grande, por ejemplo, un entero de 16 bits, de los cuales destinaríamos 8 al valor entero y 8 al valor decimal. Entonces, para extraer la parte decimal o la parte entera del número, haríamos las siguientes operaciones:

```
ValorEnComaFija = (int)( ValorConDecimal * 256.0f )
ValorConDecimal = (float)(NumeroEnComaFija) / 256.0f
```

Evidentemente, perdemos precisión.

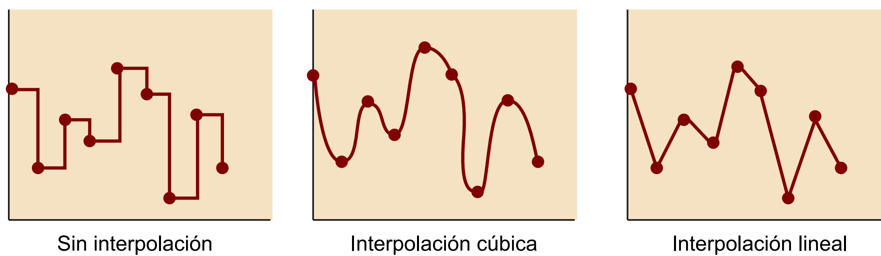
Si tenemos otros números guardados también en coma fija, la suma y la resta son directas. Por lo tanto, el coste es mínimo. En el caso de tener que multiplicar o dividir valores guardados en coma fija, hay que restaurar la proporción de parte entera y parte decimal haciendo desplazamientos de bit.

```
ResultadoEnComaFija = ( Valor1ComaFija * Valor2ComaFija ) >> 8
ResultadoEnComaFija = ( Valor1ComaFija << 8 ) / Valor2ComaFija
```

2.4.2. Interpolación

La interpolación es una técnica para averiguar valores intermedios de un conjunto de datos dados. Se puede hacer una interpolación de diferentes maneras (por ejemplo, de modo lineal o de modo cúbico).

Gráfica 1.



La interpolación cúbica es la más precisa, pero más costosa de hacer. Habitualmente realizaremos interpolaciones lineales, que dan unos resultados bastante favorables.

Describimos la interpolación lineal con la siguiente fórmula:

```
C = A + (B-A) *K
```

Supongamos que tenemos dos valores A y B y que buscamos un valor intermedio C. Necesitamos un coeficiente de mezcla *k*. Este coeficiente estará siempre entre 0 y 1. Si vale 0, el punto resultante será cercano a A, y si es 1, será cercano a B.

3. Control del píxel

3.1. Efectos a pantalla completa

En la unidad 1, el efecto de estrellas no supone un gran coste a la CPU para realizarlo, puesto que solo hacíamos un pintado de los puntos de las estrellas en sí. La mayoría de los efectos que queremos generar serán a pantalla completa, y esto implica pintar todos los puntos de la pantalla.

Hacerlo utilizando una rutina como *putpixel* puede ser demasiado lento debido a la cantidad de condiciones internas por píxel que hay. En vez de esto, lo que haremos es recorrer la matriz de puntos de pantalla con un doble bucle, y efectuaremos la función de nuestro efecto en cada píxel.

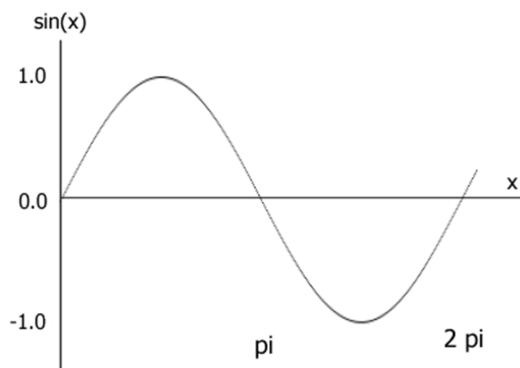
Como estamos pintando cada píxel de pantalla, nos podemos ahorrar también limpiar el *buffer* de vídeo en cada fotograma, y así ganamos más tiempo para el efecto.

3.2. Efecto de plasma

Un plasma se define como una función habitualmente hecha de senos que evolucionan con el tiempo. En cada fotograma, definimos una función que toma como parámetros dos valores de posición X e Y y nos devuelve un número real. Calculamos el resultado de cada píxel y le aplicamos un color resultante. Desmembremos algo más esta explicación.

3.2.1. Funciones sinusoidales

Gráfica 2.



Esta es la representación gráfica de la función de seno. Esta función tiene un periodo de 2 pi que se repite infinitamente. Este 2 pi se mide en radianes. El resultado de una función seno devuelve siempre un valor entre –1 y 1.

La función del coseno es similar, a excepción de que está trasladada a pi/2 radianes. No empieza en 0, sino en 1. Se podría decir que deriva de la función de seno y hay una relación directa.

```
cos(X) = sin (x+PI/2)
```

3.2.2. Precálculo

En el efecto de plasma, el precálculo es extremadamente útil. Realizar funciones sinusoidales en cada píxel puede ser muy costoso. El secreto se basa en precalcular tanto como sea posible. Una manera de conseguir velocidad es precalculando el plasma dentro de un *buffer* que sea cuatro veces más grande que el *buffer* de píxeles de pantalla, es decir, el doble de largo y el doble de ancho.

Entonces en vez de calcular cada píxel, lo que hacemos es desplazarnos como en una ventana por encima de este *buffer*. En cada fotograma seleccionamos qué parte del *buffer* queremos ver. Podemos añadir un segundo *buffer* para que nuestro plasma no sea tan monótono y el resultado sea la suma de los dos *buffers*.

Imagen 2.

El primer *buffer* responde a la siguiente función:

```
64+63*sin(hypot(SCREEN_HEIGHT-j,SCREEN_WIDTH-i)/16 )))
```

El segundo *buffer* responde a la siguiente función:

```
64+63*sin(i/(37+15*cos(j/74)))*cos(j/(31+11*sin(i/57))) )
```

Experimenta con valores para conseguir efectos diferentes. Mezcla senos y cosenos. Todos los valores están calculados entre 0 y 255. Hay que hacer un subproceso para dar color a estos valores.

3.2.3. Colorear

El proceso de dar color dependerá de una supuesta paleta virtual que precalcularemos y haremos. La idea es que tenga unos colores de transición suaves. Quedará también mejor si hacemos que los últimos colores lleguen al mismo tono que el inicial. De este modo, podemos hacer un efecto cíclico muy interesante.

Imagen 3.

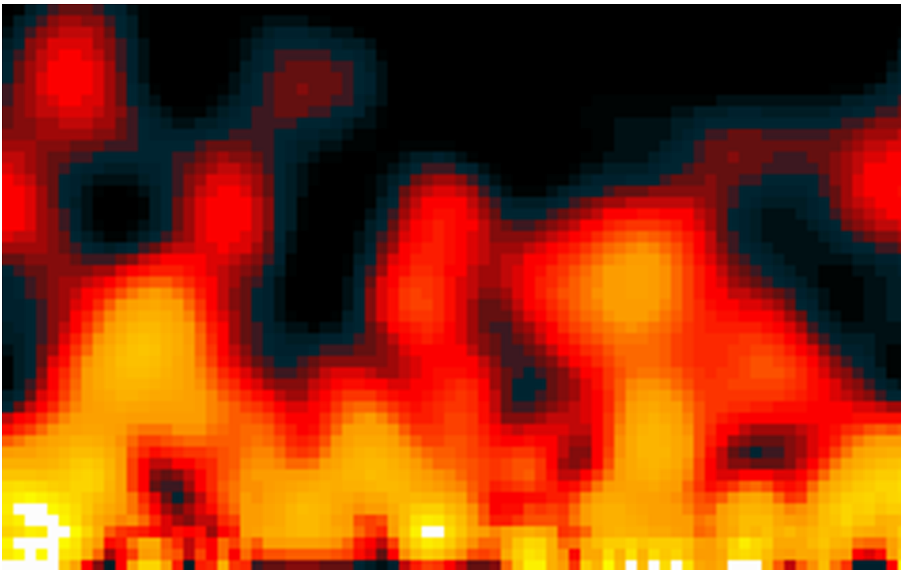
Reto 3.

Haced un efecto de plasma a pantalla completa con colores llamativos y psicodélicos.

4. Filtros

El siguiente paso que daremos una vez visto el efecto de plasma, es ver una técnica de generación de efectos basada en filtros. La idea es utilizar el fotograma anterior para generar el siguiente, y usaremos unas matrices numéricas a modo de filtros. Para probar la técnica, haremos un típico efecto de fuego.

Imagen 4.



Contenido complementario

El efecto de fuego fue inventado por Javier Arévalo Baeza, conocido como Jare, del grupo de demoscene Iguana. Apareció en la demo Inconexia en 1993 (versión en vídeo). Para la celebración de los veinte años de esta demo, Jare hizo un puerto del código a HTML5.

4.1. Matrices en filtros

Un filtro se puede aplicar a cualquier imagen para generar otra. Un filtro se puede describir como una matriz, habitualmente de 3 × 3 o de 5 × 5, que contiene números en coma flotante.

Imagen 5.

Matrix:

-0.5	0.0	1.0	0.0	-0.5
0.0	1.0	1.5	1.0	0.0
1.0	1.5	2.0	1.5	1.0
0.0	1.0	1.5	1.0	0.0
-0.5	0.0	1.0	0.0	-0.5

Divisor: 16

Para aplicar un filtro a una imagen, tomamos cada píxel de la imagen y centramos la matriz en este píxel. Entonces, por cada píxel que entra dentro de la matriz de alrededor, multiplicamos los componentes de color (RGB) del píxel por el valor que le corresponda de la casilla de la matriz. Sumamos todos los resultados y lo dividimos entre el valor divisor que asignamos a la matriz. Finalmente, guardamos el nuevo dato como valores del píxel de la imagen de destino.

Los resultados de los filtros pueden dar origen a muchos efectos típicos, como podéis ver en este enlace con ejemplos de código.

Para una explicación más matemática, consultad este enlace.

En este enlace podéis ver ejemplos de matrices de efectos utilizadas en el programa de retoque de imágenes GIMP.

4.2. Efecto de fuego

En el efecto de fuego, lo que queremos es que el color, las llamas, suban hacia arriba y vayan perdiendo intensidad. Tenemos que pensar los valores de un filtro que hagan este efecto.

Imagen 6.

Matrix:

0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0
0.0	1.0	0.0	1.0	0.0
0.0	1.0	1.0	1.0	0.0
0.0	1.0	1.0	1.0	0.0

Divisor: 8

Con este método calculamos el siguiente fotograma a partir del anterior, pero necesitamos un fotograma inicial, el primero. Lo que haremos es inicializar unos puntos de color en la parte baja del fuego y el «calor» se propagará hacia arriba. También podemos añadir algunos puntos aleatorios mientras las llamas van hacia arriba.

Reto 4

Implementad vuestro propio efecto de fuego haciendo uso de esta técnica de filtros.

5. Distorsión de *bitmaps*

5.1. Distorsión

Una vez vistos los efectos basados en filtros, vamos a ver otros de modificación de gráficos. Hasta ahora hemos cogido la información de un *buffer* y la hemos trasladado a otro de manera directa. Vamos a hacer una distorsión en el *buffer* de destino, es decir, pasaremos cada píxel del *buffer* original por una fórmula o función que produzca un cambio en el píxel. De este modo haremos una distorsión.

Igual que hicimos con el plasma, aquí podemos precalcular bastante. En el caso de que las funciones de desplazamiento que queramos usar sean constantes siempre, en vez de hacer el cálculo por cada píxel, lo que haremos será un desplazamiento de los datos entre *buffers*.

Hay que tener cuidado con el rango del *buffer*. Al hacer estos desplazamientos y distorsiones, es fácil que nos salgamos del *buffer* y provoquemos que el programa no funcione. Habría que añadir algún tipo de condición de recorte (*clipping*) para no salir de rango.

Reto 5

Cargando una imagen con SDL, aplicad un filtro de distorsión para dar un efecto como de olas. Consultad el tutorial de LazyFoo para entender cómo cargar una *surface* en memoria. SDL de serie carga solo BMP. Hay que usar la extensión SDL Image para cargar otros formatos como PNG o JPG. Mirad este otro tutorial de LazyFoo como referencia. No utilicéis las funciones de SDL de copia directa del *buffer* de la imagen cargada en el *buffer* de pantalla. Como lo que queremos es generar una distorsión, tendremos que recorrer el *buffer* manualmente, como hemos hecho hasta ahora.

5.2. Filtros bilineales

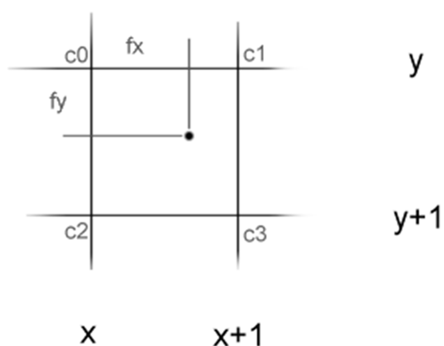
El efecto que hacemos de distorsión trata las coordenadas como datos de números enteros. Esto hace que tengamos un redondeo al color más cercano (*nearest filter*). Este tipo de aproximación tan directa puede producir resultados no deseados cuando el resultado está cerca de cero.

Una solución para esto es aplicar un algoritmo de filtro bilineal. Este tipo de filtro, hoy en día, ya lo llevan incorporado las tarjetas gráficas. Pero vamos a ver cómo es la algoritmia, y lo haremos nosotros por software.

En tarjetas gráficas se habla de téxeles en vez de píxeles cuando se hace referencia a las coordenadas de textura de un polígono. Estas coordenadas se expresan en coma flotante, con decimales. Lo que hacemos es redondear estos valores y mirar el téxel más cercano. Dejándolo aquí, esto es rápido de hacer, pero no tenemos la precisión de color que querríamos.

Para hacerlo mejor visualmente, lo que hacemos es coger los 4 téxeles que estén más cercanos al punto al que hemos ido a mirar y hacemos una interpolación lineal de los valores de color que hay en los cuatro, y ya lo tenemos.

Imagen 7.



La ecuación resultante sería esta. No está del todo optimizada, pero así se entienden mejor todos los pasos:

```
color = c0*(1-fx)*(1-fy) + c1*fx*(1-fy) + c2*(1-fx)*fy + c3*fx*fy
```

- (x,y) son la parte entera de las coordenadas de un téxel.
- (fx,fy) son la parte decimal de las coordenadas de un téxel.
- c0 es el color del téxel en la posición (x,y).
- c1 es el color del téxel en la posición (x+1,y).
- c2 es el color del téxel en la posición (x,y+1).
- c3 es el color del téxel en la posición (x+1,y+1).

Como ya hemos dicho, este proceso aumenta muchísimo la calidad del resultado, pero frena mucho el pintado. Tenemos que cargar cuatro veces más *bytes* en memoria, multiplicar los valores entre ellos, etc. Incluso el hecho de efectuarlo con coma fija podría todavía hacerlo ir lento.

Reto 6

Implementad vuestro propio filtro bilineal con el código anterior. Haced que con una tecla podamos activarlo y desactivarlo. Notad si hay frenada de FPS. Si usáis Visual Studio, podéis ver el consumo de CPU en tiempo real.

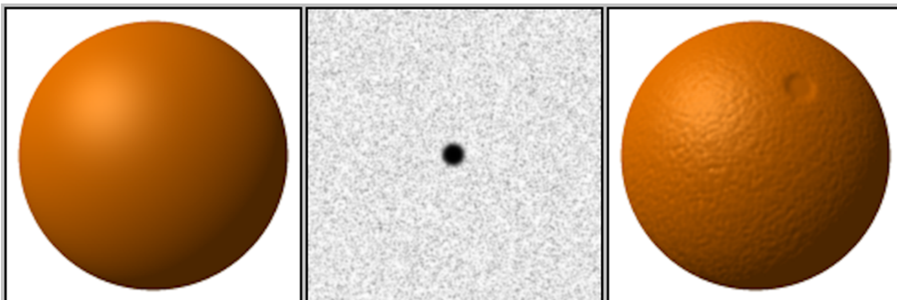
6. *Bump mapping*

Bump mapping es una técnica que hace que una superficie plana se vea como si tuviese pequeñas deformaciones, a manera de protuberancias sobre ella. La mayoría de motores de juegos 3D implementan esta técnica, y hoy en día se ha superado con el uso de mapas de normales, de los que ya hablaremos más adelante.

6.1. La teoría en el *bump mapping*

La idea principal del truco es que al mismo tiempo que guardamos la textura de color, lo que se denomina *diffuse texture*, tenemos también otra textura en tonos de gris, que corresponde al *bump map*. Esta textura indica qué elementos deben tener el efecto de protuberancia más marcado. De este modo, cuanto más claro sea un píxel, más se supone que tiene que sobresalir este punto de la textura. De hecho, podemos considerar que un *bump map* es un mapa de alturas, pero muy pequeñas.

Imagen 8.



Pensando en 3D, podemos decir que con este mapa, esta textura de tonos de gris, podemos saber el vector normal de cada píxel y ver cómo le incide la luz. Para hacerlo, necesitaremos saber la posición de la luz, hacia dónde está mirando y la orientación de la superficie, es decir, su vector normal. Como tenemos el mapa de tonos grises, de aquí sacaremos el valor de la normal de cada píxel.

Como lo estamos haciendo en 2D, podemos aplicar una serie de trucos para no hacer demasiados cálculos.

6.2. Mapa de luces

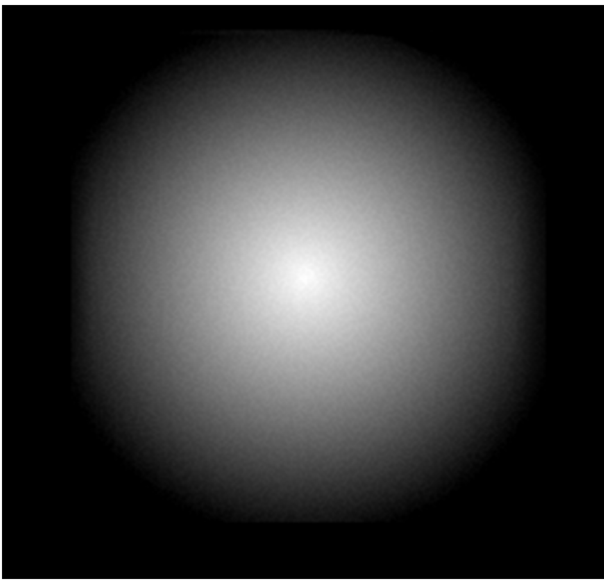
Para hacer que este algoritmo vaya rápido como efecto 2D, necesitamos evitar todas estas operaciones con vectores que hemos mencionado.

La idea básica es la siguiente.

Podemos sombrear la textura dado el vector normal. Sin embargo, podemos calcular el vector con la pendiente de la superficie en ese punto. Y conseguir la pendiente es solo una cuestión de cálculo de la diferencia entre los píxeles vecinos al *bump map*. Así pues, en resumen, podemos sombrear el píxel en el mapa de color dando la diferencia calculada a partir del mapa en relieve.

Por lo tanto, dadas dos diferencias, según el eje X y el eje Y, podemos encontrar la intensidad del píxel actual. Para esto, precalculamos y construimos un *buffer* de luz, tal como muestra la figura.

Imagen 9.



Para generar este *buffer* de luz, hacemos una simple ecuación que calcula la distancia desde el centro a cada píxel. La intensidad de la luz se va desvaneciendo linealmente desde el origen. Se podría añadir un poco de color.

Con estos dos *buffers*, el *buffer* de luz y el *bumb map*, ya podemos sombrear cada píxel con las dos diferencias, aquí expresadas como deltas. Veamos la idea con un pequeño pseudocódigo.

```
primera columna del buffer no se pinta
for (j=1; j<MAXRESY; j++) {
       primer pixel de la linea no se pinta
       for (i=1; i<MAXRESX; i++) {
               deltaX = bump_map(i,j) - bump_map(i-1,j)
           deltaY = bump_map(i,j) - bump_map(i-1,j)
             if deltaX I deltaY son en rango de pantalla {
               intensity = light_map( 128+deltaX, 128+deltaY )
             }
             pixel = colour_map(i,j) * intensity;
             pinta el pixel
```

```
        }
}
```

Reto 7

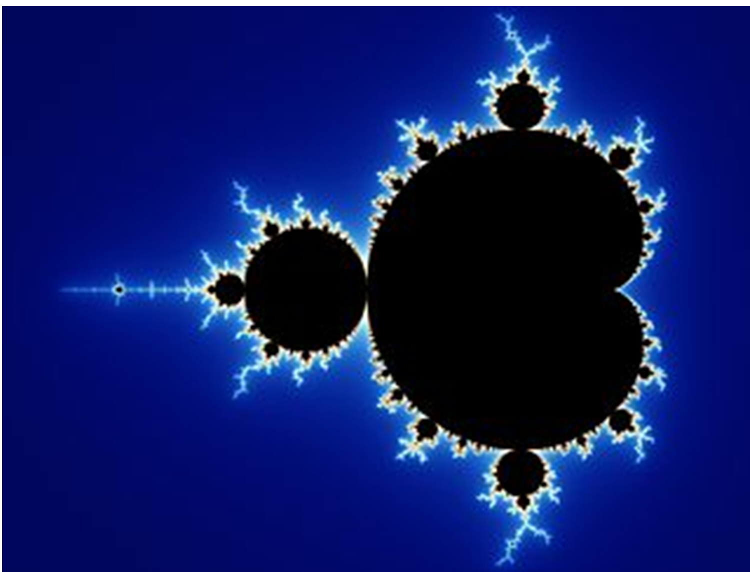
Implementad el efecto de *bump map* sobre una imagen. Construid un fichero gráfico *bump map* de tonos de gris. Y por cálculo algorítmico, generad el *buffer* de luz. Moved el *buffer* de luz por la superficie de la imagen con movimientos sinusoidales para dar la impresión de que da vueltas.

7. *Zoom* fractal

Un fractal es un objeto matemático de gran complejidad definido por algoritmos simples. El más conocido es el de Mandelbrot.

Los fractales se crearon para explicar la geometría de ciertos objetos de la naturaleza. Aparte de sus aplicaciones científicas, resultan de mucha belleza en una representación gráfica. Una de las características de un fractal es que si ampliamos un área y rehacemos el cálculo, nos puede dar una ampliación infinita.

Imagen 10.



Para comprender cómo hacerlo, antes que nada necesitamos tener una base de números complejos.

7.1. Números complejos

Hagamos un recordatorio de números complejos.

Los números complejos se pueden dividir en dos componentes: la parte real y la parte imaginaria. Si la parte imaginaria es cero, entonces el número es como un número real normal. Los números complejos tienen esta notación:

```
Z = a + i*b
```

Donde a es la parte real, b es la parte imaginaria e i es la raíz cuadrada de –1.

A la hora de representar gráficamente un número complejo, se usa el eje X para representar la parte real, y el eje Y, de ordenadas, para la parte imaginaria.

7.2. Mandelbrot

Como ya hemos dicho, la magia de los fractales es que pueden ser generados en cualquier nivel de zoom con una sencilla ecuación. En el caso que estudiaremos, la ecuación de Mandelbrot es la siguiente:

```
Z(0) = C
Z(n+1) = Z(n)^2 + C (1)
```

Hagamos una pequeña explicación: un número complejo C se dice que pertenece al fractal de Mandelbrot si Z (*n*) se mantiene dentro de la distancia finita del origen cuando *n* es infinitamente grande. Así pues, básicamente hay dos tipos de números complejos: los que pertenecen al conjunto de Mandelbrot y aquellos que no lo hacen. Lo que tenemos que hacer es determinar de manera eficiente si un número complejo está en el conjunto de Mandelbrot o no. Aquí está el truco.

Tan pronto como Z (*n*) va más allá del círculo en torno al origen de radio 2, a continuación se ha demostrado que cuando *n* se hace infinitamente grande, también lo hará Z (*n*). Así pues, todo lo que tenemos que hacer es repetir el cálculo de la ecuación (1) hasta que la distancia entre Z (*n*) y el origen sea más de 2. Por supuesto, no puede suceder, por lo que tenemos que añadir un simple contador de nuestro bucle, y rescatar si el contador llega a un cierto límite.

Como solo hay dos tipos de números complejos, ya sea dentro o fuera del sistema del fractal, de manera visual solo podríamos representar con dos colores el fractal. Lo que haremos es utilizar el valor del contador para hacer referencia a un color. Asignaremos un color basado en el número de iteraciones que necesitemos para determinar si C estaba dentro.

Pintaremos imágenes usando un intervalo de (–1,5, –1) a (0,5, 1). Rápidamente veremos la forma del escarabajo, también conocido como cardioide, característica del conjunto fractal de Mandelbrot. Cuanto más se acerque, más cardioides se encuentran.

Reto 8

Prueba de implementar el fractal de Mandelbrot como imagen estática en pantalla.

7.3. Teoría del *zoom* fractal

Podríamos calcular el fractal ampliándolo en cada fotograma, pero nos llevaría demasiado tiempo de cálculo. Hacer el cálculo con pocos píxeles, a baja resolución, puede ser una solución, pero no se vería muy bonito. Por lo tanto, haremos un *zoom* de un área del fractal, y mientras estamos mostrando el *zoom*, calcularemos de nuevo el fractal en otro nivel de *zoom*.

7.4. *Zoom* a *bitmaps*

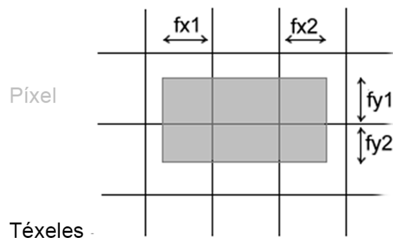
El método que usaremos para hacer este *zoom* no tiene nada de nuevo que no hayamos visto ya cuando hemos hablado del filtro bilineal y los téxeles.

Cuando hiciéramos los *zooms*, seguramente notaríamos que algunos téxeles hacen unos patrones irregulares y extraños. Esto es culpa del cálculo que hacemos, y se nota más cuando los colores que hay son muy diferentes entre ellos.

Mipmapping es justamente la técnica que hace falta para prever este problema; aunque haciéndolo en un fractal que se recalcula en cada *frame*, sería demasiado costoso. Lo que podemos hacer es un *mipmapping* dinámico, es decir, ir calculando los *mipmaps* en diferentes *zooms* y no siempre.

Todo lo que tenemos que hacer es encontrar qué área de cada téxel está cubierta por el píxel, y calcular una media ponderada de los téxeles que cubre.

Imagen 11.



La parte difícil es que un píxel puede cubrir 2 × 2, 2 × 3, 3 × 2 o 3 × 3 téxeles, de manera que o se cargan los nueve téxeles cada vez o se implementa una simple condición para gestionar cada caso por separado. Esto evitaría cargar téxeles innecesariamente, pero el código sería muy malo si lo hiciéramos así, teniendo en cuenta todas las posibilidades.

Mostramos aquí el código para el caso 2 × 3. Los otros son bastante sencillos a partir de este.

```
fy1 = 0x100-((py>>8)&0xff)
 fx2 = ((px+deltax)>>8)&0xff
 fy2 = ((py+deltay)>>8)&0xff
  c = ( bitmap[y*MAXRESX+x] * fx1 * fy1
```

```
 + bitmap[y*MAXRESX+(x+1)] * 256 * fy1
 + bitmap[y*MAXRESX+(x+2)] * fx2 * fy1
 + bitmap[(y+1)*MAXRESX+x] * fx1 * fy2
 + bitmap[(y+1)*MAXRESX+(x+1)] * 256 * fy2
 + bitmap[(y+1)*MAXRESX+(x+2)] * fx2 * fy2 ) / pixelsize;
1 = 0x100-((px>>8)&0xff)
```

Pixelsize es una constante para cada píxel. Solo tiene que calcular una vez por llamada al procedimiento de zoom. Es una variable de punto fijo de 16,16 (16 bits de entero y 16 de decimales).

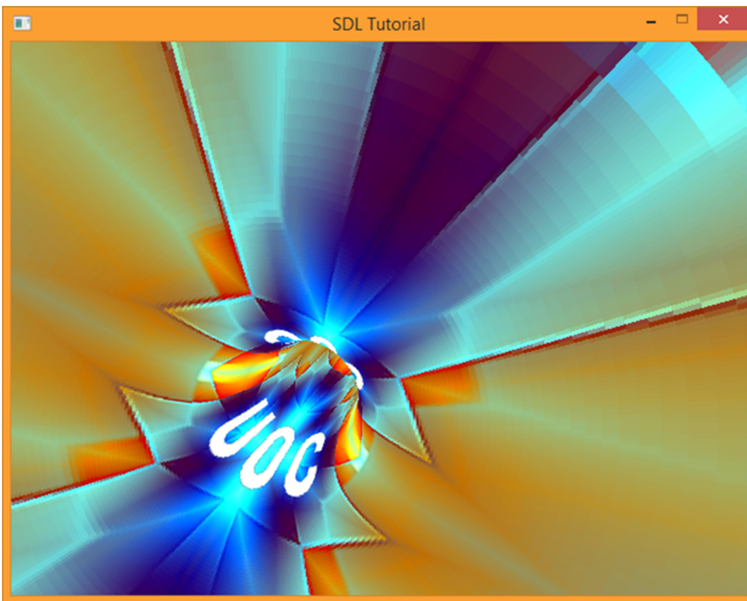
Reto 9

Aplicad la técnica del *zoom* en el dibujado del fractal de Mandelbrot.

8. Texturización estática

Hasta ahora hemos estado utilizando *buffers* de píxeles para conseguir diferentes efectos. Técnicamente, hemos hecho mapeados de texturas. Vamos a aplicar este concepto de texturización a un efecto típico 3D, pero basado en una algoritmia 2D, el efecto túnel.

Imagen 12.



8.1. Concepto de textura

Una textura se define como un mapa o *buffer* de píxeles cuadrado, es decir, igual de ancho que de alto. Sus medidas han de estar en base 2, o sea, 64, 128, 256, 512, 1.024...

Podremos aplicar texturas a cualquier cosa una vez vista la técnica. Otra característica que tiene que cumplir una textura es que sea repetitiva por todos lados.

Habitualmente a las texturas se les asignan las siglas de textura U,V. Esto es así para distinguirlas de las coordenadas de pantalla o de dibujo. Las texturas trabajan con téxeles. Las coordenadas de texturas se expresan en tanto por 1, es decir, de 0 a 1. Si obtuviéramos una coordenada superior a 1, habría que normalizarla, y obtendríamos vueltas sobre la textura.

8.2. Mapeado

La técnica se basa en precalcular el par U,V de cada píxel y en tiempo real; solo hay que sumar un desplazamiento de textura (du,dv) que hace que se mueva. Esta técnica se puede usar para crear túneles texturizados.

Reto 10

Probad de implementar un efecto túnel.

9. *Roto-zooming*

9.1. Teoría del *roto-zooming*

Como en otros efectos que hemos tratado, el *rotozoomer* es sencillo una vez que se entiende cómo funciona. La idea es que se selecciona un área de la textura con la cual queremos llenar un espacio de pantalla. El área de textura elegida puede tener cualquier forma, pero si se hace en forma de polígono de cuatro lados, es más fácil. Y todavía más si el área es ortonormal, es decir, un área rectangular.

Imagen 13.

Contenido complementario

Esta imagen de *roto-zooming* pertenece a la demo SecondReality.

Se trata de una demo para PC mítica, de visionado obligatorio y que hizo que toda una generación quisiera dedicarse a los gráficos en tiempo real, al ver que esto funcionaba en un 486 a 16 MHz. (vídeo)

Un área ortonormal es la más fácil de hacer, y un área arbitraria de cuatro lados de medidas diferentes. Todo depende del efecto que queramos dar. Si la textura contiene un texto que hay que leer, mejor que la distorsión sea ortonormal. Hacer un área arbitraria hace tambalear más el efecto, no de manera realista, pero bastante bonita. Para un área ortonormal, solo hacen falta tres puntos en la textura.

9.2. Rotaciones en 2D

El área rectangular de la textura la querremos hacer rotar y ampliar para conseguir el efecto que buscamos. Por lo tanto, hemos de tener clara la teoría de la rotación de puntos 2D.

Dados un radio y un ángulo de rotación, podemos saber las nuevas coordenadas del punto que estamos haciendo rotar.

```
X = radio * cos (angulo)
```

```
Y = radio * sin (angulo)
```

Con esto es suficiente para dar la impresión de *roto-zoom*.

Reto 11

Probad de implementar un *rotozoomer* con una imagen.

10. Sistemas de partículas

A partir de este punto del PLAN, pasemos a hacer el salto a las tres dimensiones.

Para empezar, haremos un repaso del conocimiento necesario para entender la programación del mundo 3D.

Tenemos que hacer un repaso del tema de matrices y después llevaremos a cabo un sistema de partículas, que son puntos en el espacio tridimensional.

Añadiremos a este sistema un efecto de filtro *antialias* y usaremos la imagen del fotograma anterior, a la que le haremos un reescalado para dar un efecto de estrella/túnel.

Imagen 14.

10.1. El mundo en tres dimensiones. Introducción a matrices

Antes que nada, tenemos que definir nuestro espacio tridimensional a partir de unos ejes. Hay diferentes convenciones para orientar los ejes, pero solo uno tiene sentido. Haced el siguiente ejercicio. Extended la mano izquierda ante vosotros, apuntando hacia delante. Haced que el pulgar señale hacia arriba y el dedo medio hacia la derecha. Por convención, el dedo medio apunta al eje X positivo; el índice, hacia el Z positivo, y el pulgar, hacia el Y positivo.

En el universo tridimensional, definiremos vértices y puntos tridimensionales, cada uno con tres componentes (x,y,z). Los vértices no tienen dirección. Los vectores, por otro lado, son, por definición, una dirección. Los vectores están hechos de la diferencia de dos vértices. Los vectores van de un vértice a otro. No tienen un punto fijo en un origen, sino que se pueden aplicar a cualquier punto del espacio. A todo esto, los vectores se pueden representar

con los tres componentes de un vértice (x,y,z), asumiendo que su origen es el (0,0,0). Así pues, con los tres componentes, tanto podemos representar vértices como vectores.

Los vértices se definen generalmente en mundo-universo. Lo que quiere decir es que están en un sistema de coordenadas predefinido, que se escogió cuando se creó el mundo-universo. El objetivo de una aplicación 3D es encontrar las coordenadas de estos vértices en el espacio de la cámara, girando y trasladando cuando sea necesario. Después, desde el espacio de cámara, hay que convertir el espacio de la pantalla mediante la proyección de los vértices.

Llegados a este punto, hemos de recuperar nuestros conocimientos sobre matrices matemáticas. Se recomienda la lectura de este enlace para hacer un repaso rápido:

Los puntos importantes que hemos de recordar son los siguientes:

- Las matrices son *arrays* de dos dimensiones: de 3 × 3 o de 4 × 4.
- Las matrices de 4 × 4 permiten, aparte de trabajar con los tres componentes de coordenadas (x,y,z), añadir el origen del sistema de coordenadas. Esto las hace interesantes para realizar transformaciones del espacio tridimensional (escalado, rotación y traslación).
- Es importante tener presente el concepto de matriz identidad, diagonal con 1 y el resto de casillas con ceros.
- La operación de multiplicar matrices no es conmutativa, por lo que el orden es importante.

Nota

Se os hace entrega de par de .h que definen operaciones sobre matrices y sobre vectores. Ello os facilitará el trabajo. También se redefinen operadores con este tipo de datos. Consultad el código fuente para más detalles.

10.2. Proyección 2D del mundo 3D

El paso final de trabajar con un universo 3D es la proyección a pantalla 2D, es decir, representar los puntos del espacio sobre un plano bidimensional.

```
xs = x * zs / z
ys = y * zs / z
```

zs es la distancia que hay desde la supuesta cámara 3D hasta el plano de proyección que representa la pantalla. z es la distancia que hay de la cámara a la proyección ortogonal del punto que buscamos sobre el eje Z.

Como se puede ver, es un simple tema de proporción y reducción. Nos faltaría añadir unas constantes para centrar nuestro sistema de coordenadas en mitad de la pantalla de dibujo. Básicamente, es sumar la mitad de la medida de pantalla en las dos fórmulas.

Reto 11

Cread un *array* de puntos 3D y plasmadlos contra el *buffer* de pantalla. Movedlos y hacedlos rotar para ver que son puntos tridimensionales. También podríais dar tonos de color en función de la profundidad que tengan en Z los puntos.

10.3. *Antialiasing*

Cuando pintemos en pantalla nuestras partículas, las veremos muy pequeñas. Por lo tanto, podríamos darles más grosor, pero no pintando solo el mismo tono de color, sino produciendo un cierto efecto borroso a su alrededor. Es lo que conocemos como efecto *antialiasing*.

Se puede hacer mediante un color aproximado del tono que tiene más el tono que ya hay. Su base es hacer medias de los cuatro puntos de alrededor. Es una técnica para evitar lo que se conoce como *aliasing* o dientes de sierra.

10.4. Reescalado de imágenes

Esta es una técnica sencilla que se ve muy bien. La idea es utilizar el fotograma anterior y copiar una pequeña porción, que escalamos y desenfocamos. Después dibujamos el fotograma actual por encima. Esto puede producir algunos efectos de luz muy bonitos, como cuando un rayo de luz brilla a través de la niebla.

En relación con el código, la base es similar a la proyección de 3D a 2D. Por esto los dos procedimientos están relacionados.

Tened presente que todos los ejemplos pueden ser modificados con algún movimiento sinusoidal o similar para generar un efecto nuevo. Recordad que la mayoría de efectos más vistosos son una combinación de 2D y 3D.

Reto 12

Implementad este efecto de visión borrosa en una función aparte y aplicadla al código de partículas hecho en el reto anterior para que las partículas dejen rastro. Tendréis que utilizar dos *buffers* para hacer mezcla.

11. Motores de polígonos

Ya hemos entrado en el universo 3D. Vamos a ver los fundamentos básicos de un motor de pintado de polígonos. Mucho del trabajo del que aquí hablaremos, hoy en día lo hacen las aceleradoras gráficas. Estas tarjetas de vídeo utilizan ya algoritmos en su interior para hacer el pintado de los polígonos con solo darles un conjunto de datos que son las coordenadas de los vértices y unos cuantos parámetros más. Vamos a adentrarnos en cómo funcionan por dentro para comprender mejor el proceso.

Ved también

Existen muchas bibliotecas que nos ayudan a crear gráficos 3D.
Aquí tenéis una lista.

11.1. Teoría del renderizado de polígonos por software

Haremos uso de la CPU y de la memoria RAM principal para realizar nuestro propio motor de pintado de polígonos 3D. Una de las formas más rápidas de pintar polígonos es utilizar líneas horizontales para ir pintando cada triángulo como si fuera un conjunto de líneas. Es lo que se conoce como algoritmo de *tri fill* ('llenado de triángulos').

Lo primero que tenemos que señalar es que la técnica se limita a polígonos convexos, lo que significa que todos los ángulos tienen que ser de menos de 180 grados. Nuestro pintado a base de líneas horizontales hace que este solo se cruce con dos bordes del triángulo. Si se necesita un motor que sea capaz de hacer polígonos cóncavos, se tiene que dividir este polígono en polígonos convexos pequeños.

Así pues, antes de pintar, necesitaremos la información apropiada. En la mayoría de los casos, para cada línea Y, esta información se compone de las coordenadas X y Z y las coordenadas de textura (TX, TY). Todo esto se almacena en lo que denominamos una tabla de límites.

La tabla de límites tiene dos entradas por línea: una para el inicio de la línea y otra para el final. Ahora, antes poderlo dibujar, tenemos que obtener la información correcta en la tabla de límites. Esto se hace simplemente mediante la interpolación a lo largo de cada límite. Se busca la X, los valores Z, TX y TY al recorrer estos límites para cada línea. Ahora bien, dada la tabla de límites completa, podemos interpolar fácilmente nuestra información a lo largo de las líneas horizontales mientras vamos dibujando.

Todo irá bien mientras nuestro polígono esté dentro de la pantalla de dibujo, mientras todos sus vértices estén dentro de pantalla. En el momento en que uno de los vértices se salga de pantalla, nuestro programa dejará de funcionar.

Para evitarlo, lo que hacemos es una técnica llamada *clipping*. La idea es descartar todo aquello que quede fuera de pantalla o de una región y que no se pinte. Ya hemos hablado de esta técnica en capítulos anteriores del PLAN. Hay varias maneras de hacerlo y varios puntos donde podemos mirar de hacer *clipping*.

Antes que nada, en relación con el pintado de líneas horizontal de nuestro *tri fill*, vigilaremos que cuando pintemos cada punto estemos dentro de pantalla. En el momento en que ya no estemos ahí, dejaremos de pintar y romperemos el bucle. En segundo término, podemos verificar si el polígono que intentamos pintar está completamente fuera de pantalla, cosa que hará que ya no tenga ni que entrar en el bucle de pintado.

Toda esta técnica nos produciría un pintado de polígono de color plano. Fijémonos en el siguiente enlace para ver otras técnicas de pintado, como el *Gouraud*, el texturizado o el *environment mapping*.

Aparte del color que le queramos dar al triángulo, hay otras técnicas que se pueden aplicar para conseguir una mejor calidad y hacer polígonos que imiten la luz y la realidad lo más esmeradamente posible. Algunas de ellas ya las hemos comentado en este plan (*antialiasing*, *mip-mapping*, *bilinear filtering*...).

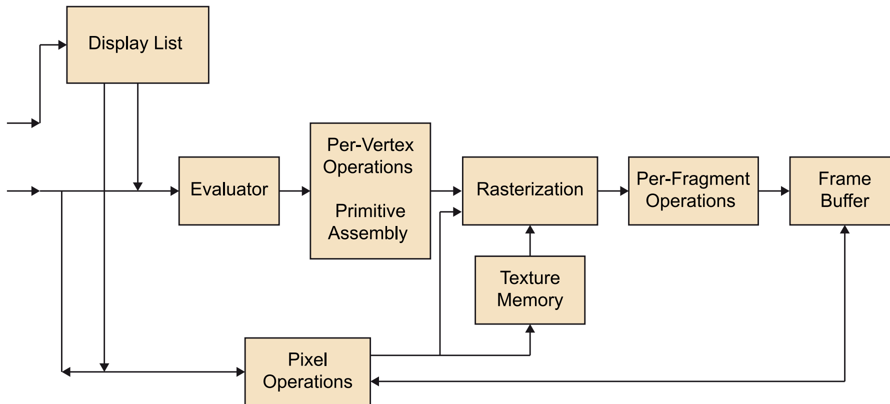
Reto 13

Por algoritmia, cread una función capaz de situar en un *array* los conjuntos de puntos necesarios para pintar la figura simple de un toroide. Haced también un *array* que guarde los triángulos que lo forman. Y aplicando la técnica del *tri fill*, haced una función capaz de recorrer la información generada y dibujar el toroide.

11.2. Cómo lo hace OpenGL. *Pipeline*

El código que pedimos para el reto nos mostrará un solo objeto, pero cuando implementamos un motor gráfico 3D hemos de tener un código capaz de manejar muchos objetos. Hay que hacer una buena planificación de código y ordenar el hecho del pintado y el cálculo de manera óptima. Es lo que se denomina *render pipeline*, de pintado.

Gráfica 3.



Hoy en día son OpenGL y las tarjetas gráficas los se encargan de este proceso. Se puede decir que nosotros preparamos los datos y OpenGL hace el trabajo sucio. Pero centrémonos en intentar explicar cómo funciona un *render pipeline*. Hará falta que ordenemos el código haciendo uso de diferentes clases.

Veamos.

Cada objeto que se ha de pintar se define por vértices, vértices normales y polígonos. Cuando hablamos de vértices normales, hablamos de un concepto que matemáticamente no es correcto del todo. Un vértice en sí no tiene una normal. El vértice normal define una normal teórica si hubiera una superficie en este punto, un plano. Se utiliza para hacer el cálculo de sombras y luces sobre el punto en concreto, y haciendo uso de todos los puntos, nos queda un objeto sombreado.

También tenemos polígonos. Estos contienen los números de los vértices que los definen, el vértice central del plano que definen y el vector normal. Este vector normal, aparte de por el tema de la luz y las sombras, es para saber si está mirando a cámara o no.

Entonces, en cada fotograma hay que volver a calcular algunos datos en relación con la posición del objeto y la cámara. La cámara generalmente no se mueve, puesto que es el objeto quien lo hace; los objetos simplemente giran ante ella. En primer lugar, hacemos girar todos los puntos y sus normales correspondientes. Después hay que limpiar la pantalla y el ZBuffer.

Una vez hecho esto, para cada polígono, comprobamos si es visible marcando el producto escalar de su normal con el vector de la cámara para el vértice central. Si es visible, lo dibujamos y dejamos que, haciendo uso del *ZBuffer*, se resuelva cualquier problema de profundidad.

OpenGL se encarga del trabajo sucio, del pintado, pero hay que especificarle todo lo que queremos. En el OpenGL moderno, se trabaja en lo que se llama VAO (*vertex attribute objects*). Este tipo de objeto contiene la información del

vértice que se ha de pintar, de cómo están unidos entre ellos, de las coordenadas de textura de cada vértice y de cómo está colocada la información en un solo *array* de datos.

OpenGL recibe esta información y la hace pasar a través de unos pequeños programas que podemos hacer nosotros. Son los llamados *shaders*. Estos *shaders* tenemos que especificarlos nosotros mismos y construirlos.

Antes que nada, procesa los datos con el *vertex shader*, que se encarga de unir los vértices del VAO con líneas. Optativamente existe un *shader* secundario, denominado *geometry shader,* que puede generar más polígonos a base de funciones matemáticas. Con esta información se generan los téxeles o *fragmentos*, en argot de OpenGL. Recordemos que los téxeles son píxeles de pantalla, pero con más información aparte del color o la posición. También tienen información 3D.

Contenido complementario

Si os interesa adentraros ya en este concepto, podéis visitar la web del grupo Khronos. Es la asociación de empresas del sector que gestiona OpenGL, entre otras API gráficas.

Estos *fragmentos* son procesados a través del *fragment shader*, que se encarga de dar el color final de píxel. Aquí es donde se procesa la luz y las sombras. Y ya finalmente se realiza un trabajo de *blending* o mezcla, que da a los objetos pintados su transparencia u ordenamiento dentro de pantalla.

En otro plan de la asignatura, veremos más a fondo cómo funciona OpenGL.

12. Texturización con corrección de perspectiva

Cuando utilizamos texturas en la representación 3D del pintado de polígonos, hemos de aplicarles a estas un factor correctivo de desplazamiento para simular la perspectiva, tal como pasa en la visión del mundo real. Veamos cómo funciona esta técnica. Le haremos algunas modificaciones para conseguir más velocidad del proceso. Pensamos en un plan infinito que se estira en perspectiva.

Imagen 15.

El error que se comete habitualmente es pensar que la aplicación de coordenadas de textura UV se ha de hacer de manera lineal. Si queremos simular perspectiva, esto no se tiene que hacer así. Para hacerlo bien, debemos tener en cuenta la coordenada Z. Hay dos técnicas para hacer esto. Lo que llevaremos a cabo es un tratamiento del píxel de dentro del polígono. En la primera técnica, nos encontraremos con un proceso lento, puesto que necesitaremos una división por cada píxel. Lo que haremos es realizar el cálculo cada 16 píxeles. Utilizaremos interpolación lineal.

12.1. Método de interpolación lineal

La primera técnica implica la interpolación lineal. Aunque U, V y Z no pueden ser interpolados linealmente en el espacio de pantalla, U/Z, V/Z y 1/Z sí que pueden. Así, cuando estamos recorriendo el largo de la línea *limit* horizontal de los polígonos que dibujamos, interpolamos estos valores a su lugar. Después, cuando necesitamos las coordenadas U y V, simplemente calculamos con la siguiente ecuación:

```
(U/Z) / (1/Z) = U/Z * Z = U
(V/Z) / (1/Z) = V/Z * Z = V
```

Fijémonos en que la división hace más lento el proceso.

12.2. Las nueve constantes

Esta técnica parece mágica. Utiliza nueve constantes. Es un poco compleja de entender, de manera que no entraremos en cómo llegar a estos valores mediante derivación. El proceso de cálculo de las nueve constantes es el siguiente:

```
// Tenemos 3 vertices que definen el poligono
        V1 V2 V3
// Marcamos uno como base.
        Bp = V1
// Definimos los vectores que definen el espacio de textura
        Up = V2-V1
        Vp = V3-V1
// Se calculan las 9 constantes utilizando el producto vectorial
     sZ = Up ^ Vp
     sU = Vp ^ Bp
     sV = Bp ^ Up
// Desarrolladas quedarian asi
        sZ.x = Up[1] * Vp[2] - Vp[1] * Up[2]
        sZ.y = Vp[0] * Up[2] - Up[0] * Vp[2]
        sZ.z = Up[0] * Vp[1] - Vp[0] * Up[1]
        sU.x = Vp[1] * Bp[2] - Bp[1] * Vp[2]
        sU.y = Bp[0] * Vp[2] - Vp[0] * Bp[2]
        sU.z = Vp[0] * Bp[1] - Bp[0] * Vp[1]
        sV.x = Bp[1] * Up[2] - Up[1] * Bp[2]
        sV.y = Up[0] * Bp[2] - Bp[0] * Up[2]
        sV.z = Bp[0] * Up[1] - Up[0] * Bp[1]
```

Y ¿qué hacemos con estas constantes? Pues a partir de aquí construimos un ecuación que nos permite derivar de nuevo 1/Z, U/Z y V/Z respeto a las nueve constantes y un vértice de pantalla en una posición (i,j).

```
1/Z = sZ.z + sZ.y * j + sZ.x * i
U/Z = sU.z + sU.y * j + sU.x * i
V/Z = sV.z + sV.y * j + sV.x * i
```

La ventaja de este método es que podemos calcular instantáneamente tanto U y V para los píxeles en el polígono sin interpolación.

Reto 14

Simulad un plano infinito con una textura aplicada y corregida de perspectiva.

13. Música y sincronización

Hemos hecho un repaso a lo largo de los capítulos de la parte visual de la *demoscene*. Nos queda la parte sonora, tan importante como la imagen. No diremos aquí cómo tiene que ser la música de la demo, pero sí que habría que destacar el tema de la sincronización. La sincronización es importante para transmitir lo que queremos con nuestra demo. Los efectos han de aparecer cuando toca, a ritmo, y esto nos da el *feeling* que tiene una buena demo.

13.1. Sincronía

Hay varias formas de hacer sincronía. Todo depende del formato de fichero sonoro que se use. Consultad el PLAN «Manipulación del sonido en videojuegos» para más información sobre los formatos sonoros y cómo están hechos por dentro.

Distinguiremos dos tipos de sincronía: por acontecimiento o aviso o por factor de tiempo.

Hacer sincronía por acontecimiento solo nos lo podemos permitir cuando estamos trabajando con un formato de sonido que marca partitura, como pueden ser MIDI o MOD. El tradicional en *demoscene* es MOD. Este formato permite disponer dentro de la partitura unas marcas, que desde el código podemos leer o saber cuándo aparecen mediante bibliotecas de reproducción de MOD, como pueden ser Mikmod o Midas Sound System.

Contenido complementario

Actualmente, cuesta de encontrar la biblioteca Midas Sound System e información sobre esta. Está ciertamente desfasada. Pero es interesante para ver cómo está hecha y cómo se estructura.

Imagen 16.

```
Konsole
-= MikMod 3.2.0 =-
Driver: Open Sound System, 16 bit normal stereo, 44100 Hz, no reverb
File: sniper.it
Name: "I am the sniper"
Type: Compressed ImpulseTracker 2.14p4, Periods: XM type, linear
pat:007/024 pos:18 spd: 6/130 vol:100%/100% time: 0:54 chn:20/39+15->35
[ 34 Instruments ] -- H:[help] S:[samples] L:[playlist] C:[config]
  0 Airy Strings                         17 Cowbell
  1 D50 High Strings                     18 Crash Cymbal
  2 So Mellow                            19 Handclap
  3 Saw Pad                              20 Kick1
  4 EPF Stringpad                        21 Openhihat
  5 Fantasia                             22 Pedal Hihat
  6 Bass                                 23 Snare
  7 SH101 Sub Bass                       24 Tamborine
  8 Acoustic Bass                        25 Timbalelo
  9 Weee                                 26 Drum
 10 TG Vibes                             27 Snare
 11 Orgel                                28 Kick2
 12 Powerlead                            29 --------
 13 TG Moot                              30 Pack set together by
 14 TG Analog                            31 Zionic
 15 Bongo                                32  ntitled
 16 Closed Hihat                         33  ntitled
```

Este método es el ideal, puesto que no tenemos que estar pendientes de activar un reloj o de contar el tiempo. Simplemente, cuando aparece la marca o acontecimiento, hacemos el cambio que queremos.

El concepto fue inventado por *demomakers*, específicamente para demos, pero las ventajas de este formato de archivo son tan obvias que algunos juegos utilizan el mismo principio. El módulo completo es básicamente una lista de pistas, que se pueden reproducir en prácticamente cualquier orden. Una pista es una lista de canales, en los que las muestras se reproducen con varios efectos diferentes. Esto es muy similar a MIDI, excepto en el hecho de que las muestras son, en realidad, diferentes para cada módulo, y por lo tanto, tienen que ser almacenadas. Esta es la razón principal por la que los módulos son más grandes que los archivos MIDI, puesto que una buena cantidad de datos de onda tiene que ser anexada al archivo. Las instrucciones para cada canal de cada pista también necesitan ser almacenadas, pero en general ocupan mucho menos espacio.

La sincronización es extremadamente fácil con los módulos. En general, los reproductores de módulos tienen una función de sincronización, de forma que cuando el reproductor se encuentra con un módulo de instrucción especial en el patrón, se llama a un procedimiento de *callback* definido por el usuario. Este procedimiento puede, a su vez, hacer lo que quiera, como fijar un coeficiente de disipación de la imagen, intercambiar una textura, imprimir un texto...

Si lo hacemos por factor de tiempo, tenemos que ir con mucho cuidado de empezar a contar el tiempo en el mismo momento en que se dispara la música. Se podrían producir algunos desajustes sonoros si, por lo que sea, el reloj que usamos se quedase sin actualizar en algún fotograma o si la música se dejase de reproducir. Esta es la manera más actual de hacerlo. Quizás es un poco más pesada, puesto que tendremos que tener una larga lista de puntos de tiempo donde queramos hacer algo o introducir algún cambio.

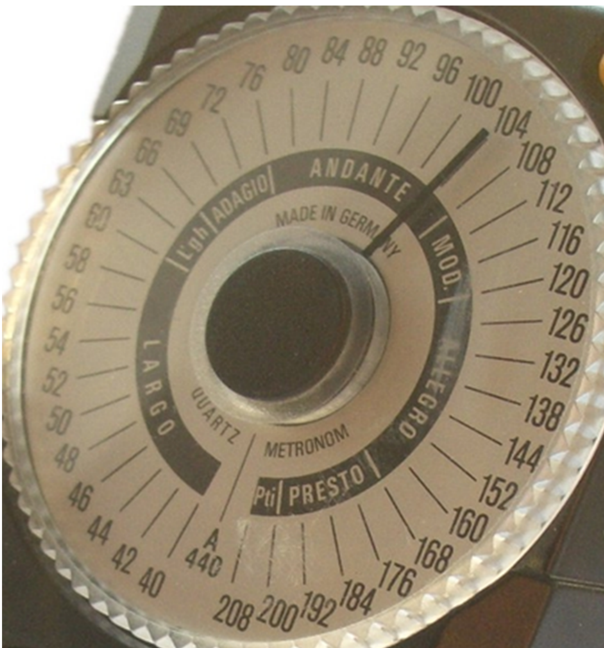
Cuando trabajamos con ficheros de formato MP3, WAV u OGG, no nos queda más remedio que hacerlo de este modo, puesto que al fichero de sonido no le podemos añadir ninguna marca de tiempo ni clave de sincronía.

13.2. SDL_Audio y SDL_Mixer

Reto 15

Investigad sobre la biblioteca SDL_Mixer, que es una extensión de SDL. Podéis usar la información del PLAN «Manipulación del sonido en videojuegos». Construid un programa que cargue una canción, la ponga en marcha y contad el tiempo desde el momento en que empecemos a lanzar la música. Contaremos milisegundos. Si sabéis el ritmo de la canción, los BPM (*beats per minute*), haced parpadear la pantalla a ritmo.

Imagen 17.



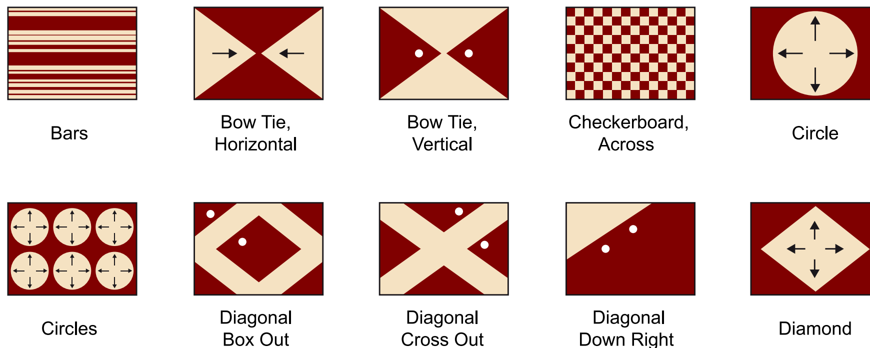
14. Acabado final

Ya disponemos de todos los elementos para hacer nuestra demo en tiempo real. Tenemos diferentes efectos visuales recreados en tiempo real y ya sabemos cómo realizar el lanzamiento de un sonido y sincronizarlo todo. Tenemos que darle ahora un acabado final bonito y atractivo. Enlazaremos ahora todos los efectos que hemos creado en un solo ejecutable, en un solo código. Para hacerlo más agradable, trabajaremos las transiciones.

14.1. Transiciones

Cuando uno piensa en la demo ideal, imagina que esta se compone de varias escenas diferentes. Como lo que queremos es que se fijen en nuestros efectos que hemos creado, intentaremos no ponerlos todos a la vez, sino que iremos mostrándolos de uno en uno. Por otro lado, tampoco nos interesa mostrar ciertos efectos junto con otros, puesto que el tiempo de cálculo quizás no deja sobreponer un efecto sobre otro. Por lo tanto, para que la demo dé una sensación de suavidad, haremos transiciones que nos lleven de un efecto a otro.

Imagen 18.



Lo más simple sería hacer una disipación progresiva hacia blanco o hacia negro, o desde pantalla en negro hacer aparecer el efecto. También podemos añadir *flashes* en tono blanco para marcar los golpes de ritmo. Asimismo, podríamos hacer una disipación progresiva de un efecto y hacer otro al mismo tiempo mientras se funden las imágenes,

La parte difícil con este tipo de transiciones es hacerlas rápidas. En general, se tiene que ejecutar el algoritmo del efecto anterior, el algoritmo de transición y el algoritmo del efecto siguiente. Todo esto porque es extremadamente intenso en términos de tiempo de computación. Pero en la mayoría de los casos, se puede tener una transición a una pantalla en negro, o una imagen estática, y después otra transición al efecto siguiente, lo que reduce drásticamente el cálculo.

El otro secreto de buenas transiciones es hacerlas originales. A continuación, apuntamos algunas ideas que pueden constituir un buen ejercicio (aunque la mayoría de ellas ya se han hecho antes):

- Disipación sencilla de entrada/salida.
- Pantalla dividida en muchos polígonos y hacer que revienten.
- Movimiento de objetos en 3D fuera de la pantalla.
- *Zoom* de muy cerca y disipación.

Contenido complementario

Una buena fuente de inspiración es ver otras demos. La página web pouet.net es la base de datos más grande de *demoscene* que hay. Es la sede central. Echadle un vistazo.

14.2. Contenido de la demo

Ahora es cosa vuestra demostrar todo aquello de lo que sois capaces para hacer una demo de cosecha propia. Explorad los efectos explicados; cambiad números; generad vuestros propios efectos; mezclad diferentes efectos para hacer uno nuevo; añadid una música; sincronizad los efectos que queráis mostrar utilizando transiciones espectaculares. El límite es vuestra imaginación.

Reto 16

Haced vuestra propia demo.