Gráficos en Computación

Laura M. Castro Souto

Primer Cuatrimestre
Curso 2000/2001

Índice de cuadros

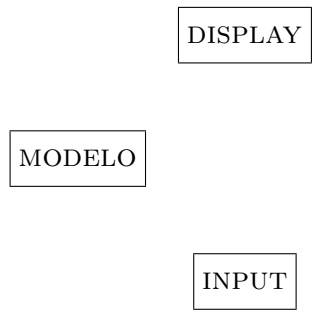
Capítulo 1

Introducción

1.1. ¿Qué significa "Gráficos en Computación"?

Podríamos decir que los **gráficos en computación** se encargan de *crear*, *almacenar*, *manipular* y sobre todo *visualizar* imágenes de objetos a partir de un **modelo** conceptual, independiente, un modelo "a parte". En la práctica, dependerán de la máquina las librerías que se usen después, que harán más rápido, nítido, ... el pintado.

Es importante incluir en esta definición que los Gráficos en Computación son *interactivos*.



Los Gráficos en Computación nacen en los 80, pero no es hasta la década de los 90 cuando empiezan a tener su mayor desarrollo, gracias a la aparición de los PCs y los MAC, que hacen disminuir cuantiosamente el precio del hardware, además de hacer cada vez más solicitados los interfaces gráficos.

1.2. Diferencia entre G.C. y Análisis o Procesado de Imágenes

Tanto en G.C. como en P.I. se trata con imágenes, pero en G.C. lo que se hace es, a partir de un modelo, realizar un dibujo, una visualización en pantalla. El P.I., por el contrario, a partir de una imagen o conjunto, lo que trata es de obtener un modelo (más o menos al revés).

$$\text{Imagen} \longrightarrow \text{Modelo} \longrightarrow \text{Dibujo}$$

El análisis de una imagen puede dividirse en varios pasos:

- ▷ Realización de mejoras en la imagen (aplicación de *filtros*). Con ello se pretende la eliminación de ruido y se realiza la detección de bordes.
- ▷ Extracción de características (y agrupación de las mismas en objetos).

▷ Obtención del modelo de algo que después se podrá pintar.

Hay campos en que G.C. y P.I. parecen seguir líneas cada vez más confluyentes, como pueden ser los relacionados con robots y con realidad virtual.

1.3. Campos de aplicación de los G.C.

Los G.C. han tenido históricamente los siguientes ámbitos de uso:

⋆ Interfaces de usuario (como ya hemos comentado).

⋆ Visualización científica, por ejemplo, de simulaciones de un modo más lento, etc. Dentro de este campo situamos también la "minería de datos" (DATA MINING).

⋆ Modelado, tanto *estático* como *dinámico*, dotado de *interactividad* (poder variar la velocidad, el ángulo, la iluminación,...), *movimiento* (poder "cambiar" o "mover" la cámara[1]), *modificación* de las *características* de los objetos,...

En realidad el segundo punto, el de la visualización científica, puede verse como un modelado de una serie de datos almacenados.

Nosotros en esta asignatura nos centraremos en el **modelado**.

1.3.1. Aplicaciones de los G.C.

El uso de G.C. está presente en:

(1) Las interfaces de usuario, totalmente extendidas.

(1) Simulaciones matemáticas, de física, economía, C.A.D. (diseño asistido por ordenador, *Computer Aided Design*).

(2) Juegos, cartografía, G.I.S. (*Graphical Information System*, necesita máquinas muy potentes), sistemas de control (fábricas, centrales,...).

1.4. Desarrollo histórico del hardware gráfico

1950 <**anterior a**> Tarjetas perforadas, teletipos. No hay procesos de E/S. El M.I.T. (*Massachussets Technological Institute*, Instituto Tecnológico de Massachussets) desarrolla el **C.R.T.** (*tubo de rayos catódicos*).

1963 Iván Sutherland enlaza las estructuras de E/S con estructuras de datos, formalizando la interacción con el usuario.

1964 Desarrollo de los primeros sistemas de C.A.D., a cargo de la General Motors.

<**en adelante**> La llegada de los PCs y los MAC hace que el coste del hardware, hasta ahora muy elevado, se abarate; además, se presentan los dispositivos de E/S tal como los conocemos (se acaba la no portabilidad provocada por la falta de estandarización; el sistema gráfico independiza los desarrollos de las plataformas).

[1] Esto fue crucial en el desarrollo de simuladores; los primeros fueron los de aviación.

Conocemos dos tipos de C.R.T., los **monitores vectoriales**, que cuentan con una lógica (hardware) dentro del monitor con comandos que realizan lo que se requiere, y los **monitores de barrido** (*raster*), con los que estamos más familiarizados, basados en la misma tecnología que la televisión.

Estos dos tipos son fruto de dos filosofías:

En el primer caso, el programa graba en el buffer de instrucciones lo que quiere pintar, una única vez. El procesador gráfico lee el buffer y dibuja en el monitor. Para que la imagen no parpadee, este dibujado debe producirse al menos 30 veces/s. La velocidad con que la imagen se genera depende de su complejidad, es decir, del número de instrucciones en el buffer (cuanto más complejo, más parpadeo y más lentitud).

En el segundo caso, contamos con un buffer orientado a los pixels en pantalla, de modo que la CPU lo único que debe hacer es elegir, según el programa lo requiera, cuáles pone a 1 y cuáles a 0. El controlador de dibujo "traduce" después los pixels al monitor. El tiempo de dibujado es ahora independiente de la complejidad del dibujo, igual que el parpadeo. El limitante es la memoria de video del buffer; si el dibujo es complejo, lo que ocurre es que la CPU tarda en calcular los pixels, y lo que se produce es una pequeña "parada".

Se adoptó la segunda porque no depende de la complejidad, resulta más barata y la estrategia de relleno es importante para formas 2D y 3D. Entre las desventajas que se habrían de solucionar se encuentra la representación discreta (*píxels*): $x_0, y_0 \rightarrow x_1, y_1$, nosotros calculamos los píxels intermedios $\Rightarrow$ visualización de "escaleras" en los dibujos, fenómeno conocido como **aliasing** y que depende en parte de la resolución del monitor. Se han desarrollado técnicas de *antialiasing* por software.

1.4.1. Dispositivos de entrada

En 1968 tuvo lugar una gran revolución con la invención del **ratón**. Actualmente se trabaja, en entornos de realidad virtual[2] y CAD, sobre ratones con 3 grados de libertad, ya no sólo en 2D.

[2]En este campo también supusieron un gran hito los *cascos de realidad virtual*, que permiten un entorno de trabajo mucho mayor; no obstante, aún tienen muchas desventajas.

1.5. Software

Distinguimos entre:

- Software de bajo nivel, cercano al código máquina, muy dependiente del hardware.
- Software de alto nivel, independiente de la máquina, que estandariza el acceso a los dispositivos[3].

1.5.1. Componentes de un sistema de gráficos

Un **sistema de gráficos** tiene básicamente 3 componentes:

Un *Programa de Aplicación*, que trabaja sobre un *modelo* y envía directivas al sistema gráfico de la máquina.

El *modelo* recoge los objetos, sus propiedades y las relaciones entre ellos. Según sea más o menos complejo lo podremos almacenar desde en un fichero hasta en algo similar a una *base de datos*, que es lo normal si la complejidad es grande.

Los formatos estándar a la hora de guardar esta información son **DXF** (AUTOCAD) y **VRML**[4].

1.5.2. Pasos para el funcionamiento de un sistema de gráficos

El "guión" que se sigue está compuesto por los siguientes pasos:

√ El programa cargará o creará un modelo nuevo.

√ Se quedará esperando la interacción del usuario, gracias al llamado *bucle de espera de eventos*[5].

√ Una vez recibida la interacción, de todo el modelo el programa escoge la parte a visualizar, proceso conocido como *recorte*.

√ Se pasa el modelo a formas geométricas (*proyección*). Dependiendo de las directivas que acepte nuestro sistema gráfico particular, podemos tener que pasar esas formas geomátricas a otras más simples.

[3] El primer gran ejemplo fue *Windows*.
[4] Virtual Reality Markup Language, un formato en ASCII fácilmente portable.
[5] Este es un elemento básico en todo programa de Windows.

Capítulo 2

Algoritmos de dibujo

En este capítulo veremos algoritmos de dibujo 2D, de líneas, círculos, curvas, elipses,... y 3D, proyecciones.

2.1. Dibujo de líneas

Las características que buscamos en el dibujado de líneas son fundamentalmente:

→ Líneas de intensidad constante, independientemente de su orientación.

→ Líneas que se dibujen tan rápido como sea posible.

Para ello se utiliza en ocasiones el **degradado** (escala de grises); no se busca la perfección, la nitidez, sino que se aprecie una línea recta "de lejos".

Cada píxel son 3 puntos, uno de cada color (rojo, verde, azul), donde inciden los rayos provenientes del CTR. Existen muy diferentes sistemas, desde los píxels que están completamente separados hasta los que solapan la luz que despiden.

2.1.1. Opciones de dibujado de líneas rectas

Cálculo directo

Dibujar una línea recta calculándola directamente es la alternativa más fácil, pero también la *peor*, como veremos.

Se usa la **función de dibujo de una recta**, es decir, se parte de la **ecuación explícita de la recta**: $y_i = mx_i + B$.

Se toma $x_1 = x_0 + 1$ y se calcula y_1([1]):

Pendiente $m = \dfrac{\partial x}{\partial y} = \dfrac{y_1 - y_0}{x_1 - x_0}$

La pendiente no será un número entero, de modo que para obtener el punto debemos hacer: $(x_i, \mathsf{Round}(y_i))$.

[1] En los ejemplos supondremos que el punto destino está a la derecha del origen.

Cálculo incremental

El método de **cálculo incremental** introduce una variación con respecto al *cálculo directo* con vistas a evitar operaciones:

$$\begin{aligned} y_{i+1} &= mx_{i+1} + B = m(x_i + \Delta x) + B \\ &= mx_i + B + m\Delta x \\ &= y_i + m\Delta x = y_i + m \end{aligned}$$

El último paso lo habilita el hecho de que $\Delta x = 1$. Como la pendiente m no es exacta (no es un número entero), el error se acumula paso a paso. De nuevo tenemos que hacer:

$$(x_{i+1}, \mathsf{Round}(y_{i+1}))$$

aunque, como pretendíamos, hemos reducido el número de operaciones a realizar.

NOTA: Esta forma de cálculo vale entre -1 y 1.
En los otros dos intervalos, $(\infty, 1)$ y $(-1, -\infty)$,
se incrementa en y y se calcula la x:

$$x_{i+1} = x_i + \frac{1}{m}$$

con $\Delta y = 1$.

Debemos, pues, tener en cuenta tres cosas a la hora de poner en práctica este método:

- Dibujamos siempre de izquierda a derecha.
- Debemos tener en cuenta la pendiente para saber qué fórmula utilizar.
- Si la recta a dibujar es *vertical* u *horizontal* no hace falta nada de lo anterior.

Algoritmo del punto medio

Este algoritmo **usa sólo enteros** y fue desarrollado por *Bresenham* en 1965. Se puede aplicar también al dibujo de círculos.

En esta ocasión partimos de la **ecuación implícita de la recta**:

$$F(x, y) = \boxed{\text{a}}x + \boxed{\text{b}}y + \boxed{\text{c}} = 0$$

$$y = \frac{\partial y}{\partial x}x + B$$

$$\boxed{\partial y}x - \boxed{\partial x}y + \boxed{B\partial x} = 0$$

Un punto por debajo hace que $F(x, y) > 0$.
Un punto por encima hace que $F(x, y) < 0$

Se usa d, **variable de decisión**, que toma el valor de la función en el punto medio **M**.

$$d = F(x_{p+1}, y_{p+\frac{1}{2}}) = F(M)$$

Si $d > 0$, quiere decir que **M** cae por debajo de la recta que pretendemos dibujar, de modo que escogemos el punto NE.

Si $d < 0$, **M** queda por encima de nuestra recta, así que escogemos el punto E.

Si $d = 0$, da igual cuál de los dos puntos, NE o E, escojamos.

Como se puede observar, ahora las operaciones consisten tan sólo en una comparación (la de d), se han eliminado los redondeos y productos.

Veremos que $d_{\text{NEW}} = d_{\text{OLD}} + \text{cte}$.

Como hemos dicho,

$$d = F(x_{p+1}, y_{p+\frac{1}{2}}) = a(x_p + 1) + b\Big(y_p + \frac{1}{2}\Big) + c$$

Si se escoge el punto E, el punto medio pasa a ser $(x_{p+2}, y_{p+\frac{1}{2}})$ y así:

$$d_{\text{NEW}} = a(x_p + 2) + b\Big(y_p + \frac{1}{2}\Big) + c$$
$$d_{\text{NEW}} = d_{\text{OLD}} + a = d_{\text{OLD}} + \partial y$$
$$\Delta_E = a = \partial y$$

Si escogemos el punto NE, tenemos como punto medio $(x_{p+2}, y_{p+\frac{3}{2}})$, y:

$$d_{\text{NEW}} = a(x_p + 2) + b\Big(y_p + \frac{3}{2}\Big) + c$$
$$d_{\text{NEW}} = d_{\text{OLD}} + a + b = d_{\text{OLD}} + \partial y - \partial x$$
$$\Delta_{NE} = a + b = \partial y - \partial x$$

Como vemos, tanto Δ_E como Δ_{NE} son *valores constantes* que conocemos desde el principio. La cuestión que nos queda pendiente ahora es: ¿a qué *inicializamos* la variable de decisión d?

Si pretendemos trazar una línea desde (x_0, y_0) hasta (x_1, y_1), podemos desarrollarlo de la siguiente manera:

$$F(x_0 + 1, y_0 + \frac{1}{2}) = a(x_0 + 1) + b\Big(y_0 + \frac{1}{2}\Big) + c$$
$$= \boxed{ax_0 + by_0 + c} + a + \frac{1}{2}b$$
$$= \underbrace{F(x_0, y_0)}_{=0} + a + \frac{1}{2}b = \partial y - \frac{\partial x}{2}$$

La d, si esta en la línea, es nula; de modo que nos queda un término constante, $a + \frac{1}{2}b$. Claro que preferimos tratar con enteros, y aquí tenemos una fracción. Pero podemos solucionarlo multiplicando por dos:

$$2F(x, y) \;\Rightarrow\; \begin{array}{l} d = 2a + b = 2\partial y - \partial x \\ \Delta_E = 2\partial y \\ \Delta_{NE} = 2\partial y - 2\partial x \end{array}$$

No debemos olvidar que todo lo que hemos visto es válido cuando la pendiente de la recta a dibujar se encuentra entre $[0, 1]$. Si la pendiente se hallase entre $[-1, 0]$ escogeríamos entre el punto E y el SE. También conviene recordar que siempre estamos pintando de izquierda a derecha, y que si de este modo, para $d = 0$ escogemos elegir el punto E, pintando a la inversa, de derecha a izquierda, lo equivalente sería elegir el punto SW.

El **problema** de este algoritmo es que *no pinta con la misma intensidad* independientemente de la pendiente. Una recta horizontal se vería más gruesa que una en diagonal. Más adelante veremos cómo se soluciona este problema mediante las técnicas de *antialiasing*.

2.2. Dibujo de círculos

La ecuación del círculo de radio R centrado en el origen es $x^2 + y^2 = R^2$. Aplicaremos siempre esta ecuación, aun en el caso de que queramos centrarlo en otro lugar; simplemente calcularíamos estos puntos y los desplazaríamos.

Despejando, tenemos $y = \pm\sqrt{R^2 - x^2}$ (f. explícita), o bien, en forma polar, $(x, y) = (R\cos\theta, R\sin\theta)$, con $\theta = [0^o, 90^o]$, el caso es que si dibujamos el círculo entre $[0, R]$, a partir de pendiente 1 los pixels que resultan se dispersan demasiado, a parte del gran número y complejidad de los cálculos necesarios.

Así las cosas, *Bresenham*, autor también del algoritmo anterior, el algoritmo del punto medio para dibujado de líneas, esgrime la idea (1977) de que con sólo calcular $\frac{1}{8}$ de la circunferencia es suficiente, pues para tener el resto sólo necesitamos manejar los signos.

De nuevo,

$$F(x, y) = x^2 + y^2 - R^2 \qquad \text{(f. implícita)}$$

Un punto sobre la circunferencia verificará $F(x, y) = 0$; $F(x, y) > 0$ denotará un punto fuera del círculo y $F(x, y) < 0$, un punto dentro del mismo.

Siguiendo la misma filosofía que en el apartado anterior, analizamos la *variable de decisión* para tomar la decisión:

$$\text{Si } d = F(M) > 0, \text{ escogeremos el punto SE}$$

$$\text{Si } d = F(M) < 0, \text{ escogeremos el punto E}$$

De igual modo, buscaremos también ahora una forma de que el **cálculo de** d sea **incremental**:

$$d_{\text{OLD}} = F(x_{p+1}, y_{p-\frac{1}{2}}) = (x_p + 1)^2 + \left(y_p - \frac{1}{2}\right)^2 - R^2$$

Si escogemos E:

$$d_{\text{NEW}} = F(x_{p+2}, y_{p-\frac{1}{2}}) = (x_p + 2)^2 + \left(y_p - \frac{1}{2}\right)^2 - R^2$$

$$d_{\text{NEW}} = d_{\text{OLD}} + 2x_p + 3$$

$$\Delta_E = 2x_p + 3$$

Si escogemos SE:

$$d_{\text{NEW}} = F(x_{p+2}, y_{p-\frac{3}{2}}) = (x_p + 2)^2 + \left(y_p - \frac{3}{2}\right)^2 - R^2$$

$$d_{\text{NEW}} = d_{\text{OLD}} + 2x_p - 2y_p + 5$$

$$\Delta_E = 2x_p - 2y_p + 5$$

Partimos del punto $(0, R)$:

$$F(M) = F\left(1, R - \frac{1}{2}\right) = 1 + \left(R^2 - R + \frac{1}{4}\right) - R^2$$

$$d_{ini} = \frac{5}{4} - R$$

De nuevo tenemos un número fraccionario; cambiamos nuestra variable de decisión d por otra, h, que se define $h = d - \frac{1}{4}$. Consecuentemente, ahora debemos comprobar:

en lugar de $d < 0$, $\Rightarrow h < -\frac{1}{4}$, y entonces elegimos E

en lugar de $d > 0$, $\Rightarrow h > -\frac{1}{4}$, y en ese caso escogemos SE

Claro que como trabajamos con enteros, en realidad comprobar si algo es menor que $-\frac{1}{4}$ equivale a comprobar si es menor que cero; por ello, podemos utilizar d normalmente, teniendo siempre presente el por qué.

Los incrementos, en el algoritmo de dibujado de líneas, son incrementos constantes: $\partial y, \partial x$. En este caso, los *incrementos no son constantes*, dependen de la posición en la que estamos en cada momento, de modo que hay que ir calculándolos paso a paso, lo que repercute en una mayor lentitud del dibujo.

No obstante, se pueden aproximar ambos algoritmos (en rapidez), si se usa un *segundo nivel de incrementos*. Al fin y al cabo, si nos fijamos, son rectas:

$$\Delta_E = 2x_p + 3$$

$$\Delta_{SE} = 2x_p - 2y_p + 5$$

De modo que:

Si el siguiente es E $\Rightarrow$ el punto de evaluación pasa a ser (x_{p+1}, y_p) y el siguiente valor del incremento, Δ_E $2(x_p + 1) + 3$, es decir, $\Delta_{E_{\text{NEW}}} = \Delta_{E_{\text{OLD}}} + 2$.

De manera similar, el siguiente valor de Δ_{SE} es $2(x_p + 1) + 2y_p + 5$, es decir, $\Delta_{SE_{\text{NEW}}} = \Delta_{SE_{\text{OLD}}} + 2$.

Si el siguiente es SE $\Rightarrow$ el punto de evaluación pasa a ser (x_{p+1}, y_{p-1}) y el siguiente valor del incremento, Δ_E $2(x_p + 1) + 3$, es decir, de nuevo $\Delta_{E_{\text{NEW}}} = \Delta_{E_{\text{OLD}}} + 2$.

Análogamente, el siguiente valor de Δ_{SE} es $2(x_p + 1) + 2(y_p - 1) + 5$, es decir, $\Delta_{SE_{\text{NEW}}} = \Delta_{SE_{\text{OLD}}} + 4$.

2.3. Dibujo de elipses

En el dibujado de elipses también se usa el algoritmo del punto medio. El mecanismo completo que veremos a continuación se lo debemos a Da Silva (1989).

La ecuación de una elipse centrada en el origen es $F(x,y) = b^2x^2 + a^2y^2 - a^2b^2 = 0$, donde $2a$ es el eje mayor de la elipse y $2b$ el eje menor.

Distinguiremos dos zonas o regiones en el cuarto de la elipse que necesitamos dibujar, separadas ambas por el punto en que la *pendiente* vale -1. En la que llamaremos REGIÓN 1, que discurre entre el punto $(0,b)$ y el de separación, se escogerá entre los puntos E y SE[2], y en la REGIÓN 2, entre el nombrado punto y $(a,0)$, elegiremos entre S y SE.

En cada paso, al empezar, deberemos comprobar, pues, si cambiamos de zona antes de sumar los incrementos y dibujar.

Calculamos

$$\partial F(x,y) = 2b^2x\mathbf{i} + 2a^2y\mathbf{j}$$

(escrito como número imaginario)

Empezamos en $(0,b)$, y el siguiente punto medio se situará en $(x_{p+1}, y_{p-\frac{1}{2}})$. Cada vez que escogemos un punto hay que verificar, como decimos, si:

$$2a^2\left(y_p - \frac{1}{2}\right) \leqslant 2b^2(x_p+1)$$

Cuando esta condición sea cierta, sabremos que cambiamos de zona.

Zona 1

La *variable de decisión* es en principio:

$$d_{\text{OLD}} = F(x_{p+1}, y_{p-\frac{1}{2}}) = b^2(x_p+1)^2 + a^2\left(y_p - \frac{1}{2}\right)^2 - a^2b^2$$

Si se escoge E:

$$d_{\text{NEW}} = F(x_{p+2}, y_{y-\frac{1}{2}}) \quad \rightarrow \quad \Delta_E = b^2(2x_p+3)$$

Si se escoge SE:

$$d_{\text{NEW}} = F(x_{p+2}, y_{p-\frac{3}{2}}) \quad \rightarrow \quad \Delta_{SE} = b^2(2x_p+3) + a^2(-2y_p+2)$$

Zona 2

Variable de decisión:

$$d_{\text{OLD}} = F(x_{p+\frac{1}{2}}, y_{p-1}) = b^2\left(x_p + \frac{1}{2}\right)^2 + a^2(y_p-1)^2 - a^2b^2$$

Si se escoge S:

$$d_{\text{NEW}} = F(x_{p+\frac{1}{2}}, y_{y-2}) \quad \rightarrow \quad \Delta_S = a^2(-2y_p+3)$$

Si se escoge SE:

$$d_{\text{NEW}} = F(x_{p+\frac{3}{2}}, y_{y-2}) \quad \rightarrow \quad \Delta_{SE} = b^2(2x_p+2) + a^2(-2y_p+3)$$

[2]Ver gráficos en transparencias.

$F(x, y) > 0$ significa que el punto cae *fuera* de la elipse

$F(x, y) < 0$ significa que cae *dentro*

En cuanto al *valor inicial* de la variable de decisión, tenemos uno para cada zona:

Zona 1

$$d_{\text{ini}} = F\left(1, b - \frac{1}{2}\right) = b^2 + a^2\left(-b + \frac{1}{4}\right)$$

Zona 2

Si suponemos (x_p, y_p) el último punto de la zona 1,

$$d_{\text{ini}} = F(x_{p+\frac{1}{2}}, y_{p-1})$$

Fijándonos, reparamos en que también en este caso los incrementos son funciones, de modo que se podrían usar *incrementos de segundo orden* para conseguir un mejor rendimiento.

2.4. Dibujo de curvas

El dibujado de algunas curvas puede verse simple, si éstas son parte de círculos o de elipses; podrían calcularse como hemos visto. Sin embargo, en la mayoría de los casos serán más complicadas que esto.

Una solución es aproximar la curva por líneas rectas, el mayor número de ellas suficiente para que no lo note el usuario, claro que pese a la eficiencia de este método, ajustar las rectas sería muy complicado debido al gran número de puntos de intersección que habría que retocar.

Para las líneas hemos utilizado funciones de primer grado, para círculos y elipses, de segundo grado; para el dibujado de curvas se usan *funciones de tercer grado o cúbicas.*

¿Cómo representar esas funciones?

Las *Funciones explícitas* $y = f(x)$ no son adecuadas, puesto que no son invariantes a rotaciones y conllevan demasiados cálculos; además, la tangente de la curva puede ser infinita, lo cual plantea problemas a la hora de unir las mismas.

Las *Funciones implícitas* $F(x, y) = x^2 + y^2 = 0$, donde es difícil calcular la dirección y magnitud de la tangente en los extremos, algo que hace falta también para juntar 2 curvas.

Lo que usaremos serán, pues, **representaciones paramétricas**, en función de un parámetro t, $0 \leqslant t \leqslant 1$:

$$\begin{aligned} x &= x(t) \\ y &= y(t) \\ z &= z(t) \end{aligned}$$

Este tipo de funciones (y representadas de este modo[3]) son más pequeñas, nos dan libertad para ajustar la curva y arrastran una menor carga computacional.

Necesitamos *cuatro parámetros fijos* para las cuatro incógnitas que tendremos (dos puntos y dos de control, cuatro puntos de control,...). Usaremos:

$$\begin{array}{rcrcl} x & = & x(t) & = & a_x t^3 + b_x t^2 + c_x t + d_x \\ y & = & y(t) & = & a_y t^3 + b_y t^2 + c_y t + d_y \\ z & = & z(t) & = & a_z t^3 + b_z t^2 + c_z t + d_z \end{array}$$

Trabajaremos con matrices, así que lo expresamos:

$$\left.\begin{array}{rcrcl} x & = & x(t) & = & a_x t^3 + b_x t^2 + c_x t + d_x \\ y & = & y(t) & = & a_y t^3 + b_y t^2 + c_y t + d_y \\ z & = & z(t) & = & a_z t^3 + b_z t^2 + c_z t + d_z \end{array}\right\} = [t^3 \quad t^2 \quad t \quad 1] \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ d_x & d_y & c_z \\ d_x & d_y & d_z \end{bmatrix} = \mathrm{T} \cdot \mathrm{C}$$

Llamaremos *curva* $\mathrm{Q}(t)$ al conjunto de las tres funciones paramétricas por comodidad.

$$\mathrm{Q}(t) = [x(t)y(t)z(t)] = \mathrm{T} \cdot \mathrm{C} = \mathrm{T} \cdot \mathbf{M} \cdot \mathrm{G} =$$

$$= [t^3 \quad t^2 \quad t \quad 1] \begin{bmatrix} m_{11} & \dots \\ \dots & m_{44} \end{bmatrix} \begin{bmatrix} \mathrm{G}_1 \\ \mathrm{G}_2 \\ \mathrm{G}_3 \\ \mathrm{G}_4 \end{bmatrix}$$

Donde **G** se denomina *vector de geometría* y representa las restricciones, los parámetros conocidos de la curva.

Tanto el vector **G** como la matriz **M** son fijos cuando pintamos la curva; el vector lo es para la curva concreta, y la matriz para el tipo de curva (Hermite, Splines,...). Lo que cambia es el valor de t, $t \in [0, 1]$, $0 \leqslant t \leqslant 1$.

La derivada de $\mathrm{Q}(t)$:

$$\mathrm{Q}'(t) = [3t^2 \quad 2t \quad 1 \quad 0] \cdot \mathrm{C} \qquad \text{(vector tgte. param. de la curva)}$$

Lo más importante, lo que más nos va a preocupar, es que no se noten los "saltos" de una curva a otra al unirlas. Para analizar esto definimos los siguientes conceptos:

Si dos curvas están juntas (tienen un extremo común), se dice que tienen **continuidad geométrica** G^0. En este caso su intersección no será suave, pero es el mínimo exigible.

Cuando las tangentes (vectores tangentes) a las dos curvas en el punto de intersección (unión) tienen la misma dirección (uno es múltiplo del otro), se dice que tienen **continuidad geométrica** G^1.

[3]Para 2D nos quedaríamos con $x(t)$ e $y(t)$.

Continuidad paramétrica

Se da **continuidad paramétrica** C^1 cuando además de tener la misma dirección, las tangentes a las dos curvas en el punto de intersección tienen la misma magnitud.

En general, se da **continuidad paramétrica** C^n si las rectas tangentes a las dos curvas en el punto de intersección tienen la misma dirección y magnitud en la derivada enésima.

Cuanto mayor sea la continuidad paramétrica, la continuidad geométrica entre las curvas, la unión, es más suave.

Veremos tres tipos de curvas:

- Curvas *de Hermite* (que usan 2 puntos - los extremos- y 2 tangentes - los vectores tangentes en ellos -).
- Curvas *de Bézier* (2 puntos extremos, 2 puntos intermedios que controlan las tangentes - de los extremos -).
- *Splines* (4 puntos de control).

El dibujado de las dos primeras es más rápido, pero los splines nos proporcionan continuidad C^1, C^2.

2.4.1. Curvas de Hermite

Como hemos dicho, en el caso de las **curvas de Hermite** las 4 restricciones son dos puntos y dos vectores:

- P_1 punto inicial
- P_4 punto final[4]
- R_1, R_4 vectores tangentes en los puntos 1 y 4

El *vector de geometría* es:

$$G_H = \begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix}$$

La *matriz base* para las curvas de Hermite:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}$$

[4] P_2, P_3 serán puntos intermedios.

A partir de que

$$\mathrm{G_H} = \begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \cdot \mathrm{M_H} \cdot \begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix}$$

se tiene que

$$P_1 =_{t=0} \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} \mathrm{M_H} \begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix}$$

$$P_4 =_{t=1} \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \mathrm{M_H} \begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix}$$

...

$$\mathrm{M_H} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}^{-1} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Si desarrollamos

$$\mathrm{Q}(t) = \mathrm{T} \cdot \mathrm{M_H} \cdot \mathrm{G_H} =$$

$$= \underbrace{(2t^3 - 3t^2 + 1)}\, P_1 + \underbrace{(-2t^3 + 3t^2)}\, P_4 + \underbrace{(t^3 - 2t^2 + t)}\, R_1 + \underbrace{(t^3 - t^2)}\, R_4$$

"funciones de mezcla" (de Hermite), ponderan cada elemento de $\mathrm{G_H}$

También se podría escribir de la forma:

$$x(t) = f(t) = at^3 + bt^2 + ct + d$$

de modo que

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 2P_1 - 2P_4 + R_1 + R_4 \\ -3P_1 + 3P_4 - 2R_1 - R_4 \\ R_1 \\ P_1 \end{bmatrix}$$

La evaluación de esto requeriría 11 productos y 10 sumas por punto. Sin embargo, podemos reducirlo a 9 productos y 10 sumas si calculamos factorizamos:

$$f(t) = ((at + b)t + c)t + d$$

(REGLA DE HORNER)

Para el dibujado de 2 curvas contiguas, buscando el mejor enlace posible, se verifica que:

$$\begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix} \rightarrow \begin{bmatrix} P_4 \\ P_7 \\ k \cdot R_4 \\ R_7 \end{bmatrix}$$

Si $k = 1$, entonces tenemos *continuidad paramétrica* C^1; mientras que si $k \neq 1$ se da *continuidad paramétrica* C^0.

2.4.2. Curvas de Bézier

Especificamos los puntos inicial y final, y *de forma indirecta* sus vectores tangentes, mediante dos puntos que no estarán en la curva, usando los vectores P_1P_2 y P_3P_4, que cumplen:

$$R_1 = Q'(0) = 3(P_2 - P_1)$$
$$R_4 = Q'(1) = 3(P_4 - P_3)$$

Obtendremos la *matriz base de Bézier* a partir de la *matriz de curvas de Hermite*:

$$\mathrm{Q}(t) = \mathrm{T} \cdot \mathrm{M_B} \cdot \mathrm{G} = \mathrm{T}(\mathrm{M_H}\mathrm{M_{HB}})\mathrm{G_B}$$

donde $\mathrm{M_{HB}}$ es la *matriz de paso entre H y B* y verifica:

$$\mathrm{G_H} = \begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \cdot \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}$$
$$\mathrm{G_H} = \mathrm{M_{HB}} \cdot \mathrm{G_B}$$

donde $\mathrm{G_H}$ es el vector de geometría de Hermite y $\mathrm{G_B}$ el vector de geometría de Bézier. La matriz base:

$$\mathrm{M_B} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

La expresión de $\mathrm{Q}(t)$ nos queda:

$$\mathrm{Q}(t) = \underbrace{(1-t^3)}P_1 + \underbrace{3t(1-t^2)}P_2 + \underbrace{3t^2(1-t)}P_3 + \underbrace{t^3}P_4$$

polinomios de Bernstein

Y cuando queramos pintar dos curvas contiguas:

- con continuidad paramétrica C^0 $\longrightarrow$ $P_4 - P_3 = k(P_5 - P_4)$, con $k \neq 1, k > 0$
- con continuidad paramétrica C^1 $\longrightarrow$ $P_4 - P_3 = P_5 - P_4$

En las curvas Bézier tenemos una característica que no teníamos con las de Hermite: la curva queda encerrada dentro del polígono que forman sus 4 puntos de control (***convex hull***[5]).

2.4.3. Splines

La definición de **spline** es algo como una lámina de metal fino que es capaz de adaptarse a una determinada forma.

Distinguiremos los **splines naturales**, que van a tener continuidad paramétrica C^0, C^1 y C^2, un orden más que en los casos anteriores (trazado más suave, aunque cálculo más complejo) y van a interpolar los puntos de control, P_i.

Sin embargo, este tipo de splines no se usan debido a dos principales desventajas: primero porque su cálculo depende de invertir una matriz de orden $(n+1) \times (n+1)$ con $n =$ número de puntos de control. Y segundo porque variando uno de los puntos de control nos varía la forma de toda la curva, cuando en realidad nos interesaría un ajuste más local.

Los que se usan son, en esta misma familia, los **B-splines**, donde las curvas se consideran compuestas por varios segmentos, cada uno de los cuales estará afectado por un conjunto reducido de puntos de control. Como se divide en partes, el cálculo de cada parte es más sencillo. Se mantienen los mismos órdenes de continuidad paramétrica (C^0, C^1, C^2), pero plantean el problema de no interpolar los puntos de control, es decir, no podemos especificar los puntos por los que se tiene que pasar, aunque esto puede conseguirse *duplicando* los puntos de control (que nosotros controlamos y colocamos donde queremos). Si hacemos iguales 3 de los puntos de control, haremos que la curva pase exactamente por ese lugar, con el tributo de que los segmentos anexos se conviertan en rectas, aunque esto no ocurre si se usan *B-splines no uniformes*, de los que hablaremos más adelante.

La continuidad entre los segmentos de los B-splines se consigue, como se deduce de las líneas anteriores, porque comparten los puntos de control:

Si se tienen $n+1$ puntos de control entre $P_0 \dots P_n \rightarrow n-2$ segmentos, entre Q_3 y Q_n. Y si antes teníamos que el parámetro t verificaba $0 \leqslant t \leqslant 1$, ahora tenemos $n-1$ parámetros, de $t_3 \dots t_{n+1}$ con $t_3 = 0, t_4 = 1, t_5 = 2 \dots$ A estos puntos t_i los llamaremos **nodos** (o *nudos*, en inglés *knot*).

Si queremos que una *curva* de este estilo sea *cerrada*, repetiremos la secuencia de los 3 primeros puntos de control al final: $P_0P_1P_2 \dots P_nP_0P_1P_2 \Rightarrow$ B-spline cerrada. Veremos dos tipos de **B-splines cerradas**: **uniformes** (donde la distancia entre t's será uniforme e igual a 1) y **no uniformes** (donde las distancias entre las t's no son uniformes).

B-splines uniformes

Cada segmento de la curva, Q_i, está definido por cuatro puntos de control: $P_{i-3} \dots P_i$. En general, cualquier segmento Q_i empezará cerca de P_{i-2} y finalizará cerca de P_{i-1} (al dibujarlo).

El vector de geometría que definen esos cuatro puntos es:

[5] En inglés, polígono convexo.

$$\mathrm{G_{BS_i}} = \begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix}$$

Cada punto de control, consecuentemente, afecta a 4 segmentos de la curva. En este caso la ecuación para el dibujado de ésta es:

$$\mathrm{Q}_i(t) = \mathrm{T}_i \cdot \mathrm{M_{BS}} \cdot \mathrm{G_{BS_i}}$$

donde

$$\mathrm{M_{BS}} = \frac{1}{6} \cdot \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}$$

y

$$\mathrm{T}_i = [\ (t-t_i)^3 \quad (t-t_i)^2 \quad (t-t_i) \quad 1\ ] \qquad t_i \leqslant t < t_i + 1$$

de suerte que cada valor del vector T_i estará entre 0 y 1.
En función de los puntos:

$$\mathrm{Q}_i(t) = \frac{1}{6}(1-t)^3 P_{i-3} + \frac{1}{6}(3t^3 - 6t^2 + 4)P_{i-2} + \frac{1}{6}(-3t^3 + 3t^2 + 3t + 1)P_{i-1} + \frac{t^3}{6}P_i$$

B-splines no uniformes

La característica distintiva de los B-splines es que las distancias entre t's no es uniforme; la única característica que deben cumplir es ser una secuencia *creciente* (con la posibilidad de que se repitan).

Como ya hemos comentado, una de sus aplicaciones consiste en evitar que los segmentos anexos a los puntos que se hacen coincidir para hacer pasar a una curva por un determinado lugar se vuelvan rectos. Tenemos dos ventajas al utilizarlos: podemos reducir la continuidad paramétrica desde C^2 a C^1, a C^0 e incluso eliminar la continuidad geométrica.

$$C^2 \rightarrow C^1 \rightarrow C^0 \rightarrow \mathrm{G}^0$$

La segunda ventaja es que además de que con continuidad C^0 seguiremos teniendo segmentos curvos, nos es posible interpolar también los extremos sin introducir segmentos lineales.

- ◊ Cuando hacemos coincidir dos puntos, por ejemplo $t_4 = t_5 \rightarrow$ un segmento menos porque Q_4 pasaría a tener longitud 0. Q_3 quedaría contiguo a Q_5, compartirían 2 puntos de control en lugar de tres, de modo que se tendría continuidad C^1.

- ◊ Si se comparten tres, por ejemplo t_4, t_5, t_6 quedarían contiguos $\mathrm{Q}_3 - \mathrm{Q}_6$, que compartirían 1 punto $\Rightarrow$ continuidad C^0 y además se interpolaría el punto P_3.

- ◊ Si se tienen tres t's iguales, se rompe la continuidad, porque entre Q_3 y Q_7 no habría puntos en común $\Rightarrow \mathrm{G}^0$ (2 curvas separadas).

Splines de Catmull-Rom

[6]

Pese a ser mucho más complejos, son interesantes porque permiten, a partir de una serie de puntos, interpolar la curva que pase por ellos. Es decir, dada una secuencia de puntos $P_0 \dots P_m$, conseguiremos una curva que irá entre $P_1 \dots P_{m-1}$.

En este caso no se da la característica del *convex hull*, y se tiene:

$$\mathrm{Q}_i(t) = \mathrm{T} \cdot \mathrm{M}_{\mathrm{CR}} \cdot \mathrm{G}_{\mathrm{B}} =$$
$$= \frac{1}{2}\mathrm{T} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix}$$

2.4.4. Subdivisión de curvas

Cuando necesitamos *más control* para ajustar las curvas, lo que se puede hacer es:

↪ *Aumentar el grado de la curva* que estamos dibujando (en lugar de cúbica, de grado 4, 5,...). Problema: mucho más cómputo, redibujado mucho más lento.

↪ Inclusión de *más puntos de control* (lo más corriente cuando se quiere ajuste fino en determinadas zonas).

Subdivisión sobre curvas Bézier

Se definen 2 trozos de curva (se "parte"):

$$\begin{array}{lll} \text{Antes} & & 0 \leqslant t < 1 \\ \text{L} & \Longrightarrow & 0 \leqslant t < 0'5 \\ \text{R} & \Longrightarrow & 0'5 \leqslant t < 1 \end{array}$$

De suerte que, si originalmente teníamos puntos $\mathrm{P}_1 \dots \mathrm{P}_4$ ahora tenemos[7]:

$$\mathrm{L}_1 = \mathrm{P}_1 \qquad \mathrm{R}_2 = \frac{\mathrm{H} + \mathrm{R}_3}{2}$$
$$\mathrm{L}_4 = \mathrm{R}_1 = \frac{\mathrm{L}_3 + \mathrm{R}_2}{2}$$
$$\mathrm{L}_2 = \frac{\mathrm{P}_1 + \mathrm{P}_2}{2} \qquad \mathrm{R}_3 = \frac{\mathrm{P}_3 + \mathrm{P}_4}{2}$$
$$\mathrm{H} = \frac{\mathrm{P}_2 + \mathrm{P}_3}{2}$$
$$\mathrm{L}_3 = \frac{\mathrm{P}_2 + \mathrm{H}}{2} \qquad \mathrm{R}_4 = \mathrm{P}_4$$

[6] Ver transparencias.
[7] Ver transparencias.

Y se verifica:

$$\mathrm{G_B^L} = \underbrace{\mathrm{D_B^L}}_{\text{matriz de división}} \cdot \underbrace{\mathrm{G_B}}_{\text{vector geométrico}} = \frac{1}{8}\begin{bmatrix} 8 & 0 & 0 & 0 \\ 4 & 4 & 0 & 0 \\ 2 & 4 & 2 & 0 \\ 1 & 3 & 3 & 1 \end{bmatrix} \cdot \underbrace{\begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}}_{\text{se obtiene de lo anterior}}$$

$$\mathrm{G_B^R} = \mathrm{D_B^R} \cdot \mathrm{G_B} = \frac{1}{8}\begin{bmatrix} 1 & 3 & 3 & 1 \\ 0 & 2 & 4 & 2 \\ 0 & 0 & 4 & 4 \\ 0 & 0 & 0 & 8 \end{bmatrix} \cdot \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}$$

Subdivisión sobre curvas de Hermite

Puesto que hemos visto cómo "pasar" de Hermite a Bézier, es aplicable el apartado anterior.

Subdivisión sobre B-Splines uniformes

Se aplica la misma división en 2 partes, siendo las relaciones en este caso:

$$\mathrm{G_{BS}^L} = \mathrm{D_{BS}^L} \cdot \mathrm{G_{BS}} = \frac{1}{8}\begin{bmatrix} 4 & 4 & 0 & 0 \\ 1 & 6 & 1 & 0 \\ 0 & 4 & 4 & 0 \\ 0 & 1 & 6 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}$$

$$\mathrm{G_{BS}^R} = \mathrm{D_{BS}^R} \cdot \mathrm{G_{BS}} = \frac{1}{8}\begin{bmatrix} 1 & 6 & 1 & 0 \\ 0 & 4 & 4 & 0 \\ 0 & 1 & 6 & 1 \\ 0 & 0 & 4 & 4 \end{bmatrix} \cdot \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}$$

NOTA

Para el caso de los *B-Splines no uniformes* no existe matriz de división y el proceso es recursivo y por tanto más complejo que los que hemos visto aquí.

2.4.5. Dibujo de forma óptima

La primera ecuación para el dibujado de curvas que vimos, en forma paramétrica, fue $f(t) = at^3 + bt^2 + ct + d$, de donde hemos derivado todo lo que hemos visto en este punto 2.4.

Esta ecuación se obtenía también a partir del producto $\mathrm{T} \cdot \mathrm{M} \cdot \mathrm{G}$, como vimos con detalle para el primer caso, las curvas de Hermite.

Vimos la mejora que suponía presentar $f(t)$ en la forma de la *regla de Horner*, $f(t) = ((at + b)t + c)t + d$, que nos reducía los cálculos a 9 productos y 10 sumas.

Por supuesto, nos interesará quedarnos sólo con sumas para simplificar el cómputo, de modo que analizaremos los incrementos, de la misma manera que ya hicimos con rectas, etc.

El incremento de un punto a otro viene dado por:

$$\Delta f(t) = f(t+\delta) - f(t)$$

es decir,

$$f_{n+1} = f_n + \Delta f_n$$

Utilizando la ecuación original:

$$\Delta f(t) = a(t+\delta)^3 + b(t+\delta)^2 + c(t+\delta) + d - (at^2 + bt^2 + tc + d)$$

$$= 3at^2\delta + t(3a\delta^2 + 2b\delta) + a\delta^3 + b\delta^2 + c\delta$$

Como se puede observar, el orden de la función se ha reducido de 3 a 2, así que se ha ganado, pero aún hay productos que realizar; proseguimos, pues, haciendo las *diferencias de segundo orden*:

$$\Delta^2 f(t) = \Delta f(t+\delta) - \Delta f(t)$$

Desarrollando:

$$\Delta^2 f_n = \Delta f_{n+1} - \Delta f_n \qquad \Longrightarrow \qquad \Delta f_n = \Delta f_{n-1} + \Delta^2 f_{n-1}$$

Ya sustituyendo de nuevo nos quedaría:

$$\Delta^2 f(t) = 6a\delta^2 t + 6a\delta^3 + 2b\delta^2$$

que es de orden 1 (una recta). Pero en las rectas también habíamos utilizado el desarrollo de las diferencias, así pues:

$$\Delta^3 f(t) = \Delta^2 f(t+\delta) - \Delta^2 f(t)$$

es decir,

$$\Delta^3 f_n = \Delta^2 f_{n+1} - \Delta^2 f_n \qquad \Longrightarrow \qquad \Delta^2 f_{n-1} = \Delta^2 f_{n-2} + \Delta^3 f_{n-3}$$

Y nos queda:

$$\Delta^3 f(t) = 6a\delta^3$$

Volviendo a la expresión que teníamos al principio:

$$f_{n+1} = f_n + \Delta f_n = f_n + \Delta f_{n-1} + \Delta^2 f_{n-1}$$

Ya que partimos de $f(t) = at^3 + bt^2 + ct + d$, si hacemos $t = 0$:

$$f_0 = d \qquad \Delta f_0 = a\delta^3 + b\delta^2 + c\delta \qquad \Delta^2 f_0 = 6a\delta^3 + 2b\delta^2 \qquad \Delta^3 f_0 = 6a\delta^3$$

Como se puede observar, $\Delta^3 f_0$ es *fijo*, para todos los puntos es igual. Estos valores son la **inicialización**; son $3 \times 3 = 9$ sumas para cada punto además de la inicialización, que se hace una sola vez.

Capítulo 3

Dibujo 3D

El dibujado en 3D resulta más complejo debido a dos razones:

$\rightsquigarrow$ Se tiene una dimensión más (eje z).

$\rightsquigarrow$ Los medios (dispositivos) de visulización son 2D.

Se introduce una fase más en el proceso de dibujo. Hasta ahora hacíamos:

$$\boxed{\text{Modelo 2D}} \longrightarrow \boxed{\text{Recorte}} \longrightarrow \boxed{\text{Scan-Dibujo}}$$

Aunque la etapa de **recorte** no la hemos visto. Se ocupa de escoger la parte el modelo que queda "dentro de la pantalla" y de mostrar sólo eso.

Para dibujar objetos 3D:

$$\boxed{\text{Modelo 3D}} \longrightarrow \boxed{\text{Recorte}} \longrightarrow \boxed{\text{Proyección}} \longrightarrow \boxed{\text{Scan-Dibujo}}$$

El paso que se añade, de **proyección**, es necesario para *traducir* de coordenadas 3D a coordenadas 2D, de forma que la fase de dibujo (*scan*[1]) sea como la bidimensional. Es, pues, una fase clave, en la que más nos detendremos.

Otras técnicas que veremos serán la detección de superficies visibles, el sombreado, etc.

3.1. Proyecciones

Definición

Una **proyección**[2] es una *transformación* de unos puntos que están en un sistema de *coordenadas de dimensión n* a un sistema de *coordenadas de dimensión menor* que n.

Definición

Llamamos **proyector** a la línea recta que sale del *centro de proyección*, pasa por cada punto del objeto e interseca con el plano de proyección.

[1] Algoritmos de dibujo.

[2] Cuando no especifiquemos nada más, nos estaremos refiriendo a *proyecciones geométricas planas.*

Definición

El **centro de proyección** es el punto de vista desde donde se mira la figura.

Así pues, usando *proyecciones geométricas planas*, proyectamos sobre un *plano de proyección* y los proyectores que usamos son rectas. Como consecuencia de estas dos cosas, al proyectar una línea recta, sólo tenemos que proyectar sus extremos y posteriormente unirlos.

Tenemos dos tipos de proyecciones: **proyecciones de perspectiva** y **proyecciones paralelas**. La diferencia entre ambas nos la da el *centro de proyección*: en las *proyecciones de perspectiva* el centro de proyección está a una *distancia finita* del *plano de proyección*, mientras que en las *proyecciones paralelas* está a una *distancia infinita* de dicho plano. Debido a esto, mientras que las proyecciones de perspectiva se especifican o definen por su centro de proyección, las proyecciones paralelas lo hacen por la *dirección de proyección*, un vector que nos dice hacia dónde tenemos que mirar.

La proyección de perspectiva tiende a parecer más realista, queda mejor en pantalla, pero no es buena para medir distancias, no se mantienen los ángulos en las figuras y en general las líneas paralelas de los objetos no se van a mantener paralelas.

Por el contrario, en una proyección paralela el aspecto no es tan realista pero las líneas paralelas quedan paralelas, se mantienen las distancias, y los ángulos en las superficies paralelas al plano de proyección.

3.1.1. Proyecciones de Perspectiva

Existen tres tipos de proyecciones de perspectiva, dependiendo del número de ejes que corte el plano de proyección. En estas proyecciones todas las líneas que no sean paralelas al plano de proyección van a tender a un **punto de fuga**. Tenemos un *punto de fuga* por cada eje que corta el plano de proyección.

3.1.2. Proyecciones Paralelas

Distinguimos dos tipos, dependiendo de la relación entre la dirección de la proyección (DOP) y la *normal al plano de proyección* (VPN, view plane normal). Si ambas direcciones coinciden, tenemos **proyecciones paralelas ortogonales**; sino, hablamos de **proyecciones paralelas oblicuas**.

Proyecciones Paralelas Ortogonales

Las clases más comunes de proyecciones ortogonales son:

Planta o *elevación superior*, *elevación de plano* (top elevation).

Alzado o *elevación frontal* (front elevation).

Perfil o *elevación lateral* (side elevation).

Las ventajas de estas proyecciones son que distancias y ángulos pueden ser medidos a partir de ellas, pero en ocasiones puede ser difícil reconocer un objeto, ya que en cada caso sólo se muestra una de sus caras.

Otro tipo de proyección ortogonal es la **axonométrica**, en la que la normal del plano no se corresponde con ningún eje principal (en las listadas líneas arriba sí lo hace); se ven por tanto varias caras al mismo tiempo. Se conserva el paralelismo de las líneas, pero no los ángulos, y las distancias se pueden medir sobre cualquiera de los ejes principales con *factores de escalamiento*.

La proyección **isonométrica** es una proyección axonométrica muy usada, en la que el ángulo que forma el plano de proyección con cada eje es el mismo (el factor de escalamiento es igual para los 3 ejes principales). Hay 8 visiones isonométricas de una figura, una en cada octante.

Proyecciones Paralelas Oblicuas

En ellas el plano de proyección es normal a un eje principal. Se distingue entre:

Caballeras, en las que la diferencia entre la dirección del plano y la dirección de proyección del plano es de 45^o. Las líneas perpendiculares al plano de proyección mantienen su tamaño.

Cabinet, donde el ángulo es de $63'4^o$. Las líneas perpendiculares al plano de proyección reducen su tamaño a la mitad.

3.1.3. Nomenclatura

Necesitamos definir los siguientes conceptos:

El **plano de proyección**, determinado por

un punto en él, *punto de referencia* o View Reference Point (VRP)
y una perpendicular a él, *normal* o View Plain Normal (VPN)

Tendremos también la "parte" del plano de proyección o *plano de vista* que realmente se va a visualizar (en la pantalla, en un formulario Delphi...): la **ventana** (W), y para ella (se define relativa a) un **sistema de coordenadas de referencia de vista** (VRC), cuyo centro es el VRP y con ejes (u, v, n): n viene dado por la normal al plano de proyección VPN, v por un vector definido respecto a la normal, VUP, y u de forma que los tres formen un sistema de coordenadas RIGHT-HANDED. Una vez definido el VRC podemos determinar las coordenadas $(u_{\min}, v_{\min}), (u_{\max}, v_{\max})$, puntos mínimo y máximo para la ventana.

El *modelo* a pintar estará ubicado en un **sistema de coordenadas del mundo**, (x, y, z), respecto al cual se situarán los objetos.

El *centro de proyección* y la *dirección de proyección* (DOP) se definen con un **punto de referencia de proyección**[3] (Proyection Reference Point, PRP), un punto "donde empezar", indicador del tipo de proyección. Si la proyección es de perspectiva, el PRP es el centro de proyección. Si es paralela, la *dirección de proyección* (DOR) se define como el vector entre el PRP y el *centro de la ventana*[4] (CW), que suele ser distinto del VRP.

Volumen de vista

Limita la "porción de mundo" que se recortará y proyectará en el plano de vista. Para una proyección de perspectiva es una pirámide truncada (o no) con ápice en el PRP y definida por los proyectores (aristas) que pasan por las esquinas de la ventana de proyección. Para una paralela, es un paralelogramo con los lados paralelos a la dirección de proyección (del PRP al centro de la ventana).

Estos volúmenes pueden ser *infinitos* si no les ponemos una limitación; normalmente se coloca un límite al volumen de vista para restringir lo que pintaremos (**recorte**) mediante dos planos paralelos al plano de vista: **Front Clipping Plane** (F) y **Back Clipping Plane** (B), planos de *recorte anterior / posterior*. Estos planos se especifican mediante una distancia (F, B) relativa al plano de proyección.

3.2. Definición de Modelos 3D

Una vez que ya hemos visto todos estos conceptos, lo que necesitamos saber es: *¿Qué valores debemos conocer para poder hacer una proyección de cualquier tipo?*

Tenemos un sistema de coordenadas del mundo dado, a partir de las que definimos el VRP que con la normal determina el plano de proyección.

Definíamos después un vector hacia arriba, VUP, y junto con lo anterior, tenemos el llamado sistema de coordenadas de vista (u, v, n).

Determinados $(u_{\min}, v_{\min})$ y $(u_{\max}, v_{\max})$, tenemos entonces la ventana.

El VRP y el tipo de proyección nos definen nuestra proyección concreta y por último tendremos el volumen de vista, infinito a menos que establezcamos F y B, dos distancias que lo limitan.

Los pasos en el dibujado 3D, como ya hemos vislumbrado, son:

PASOS

1. Definir los *parámetros* de la vista.
2. Tendremos un *modelo* con objetos en 3D.
3. Aplicar el tipo de *proyeción* definida en el primer punto para *traducir las coordenadas* 3D del modelo a coordenadas 2D en el plano de proyección.

Nos ocuparemos en este apartado del segundo paso.

[3]Este punto se define en el VRC, no en el sistema de coordenadas del mundo. Su posición no cambia al variar el VUP ni el VRP, pero ello puede hacer más difícil moverlo para obtener diferentes vistas.

[4]El centro de la ventana no tiene por qué coincidir con el centro de coordenadas de vista.

Existen tres tipos de definición de modelos 3D:

- Definición de superficiones de polígonos.
- Superficies curvas.
- Modelado de sólidos.

Veremos con detenimiento cada uno de ellos:

3.2.1. Superficies de polígonos

En muchos sistemas se usa este primer tipo de definición de modelos 3D porque los polígonos son superficies de muy baja complejidad y tienen como ventaja que son rápidos de dibujar (las ecuaciones de los polígonos son ecuaciones lineales). Como desventaja, su dibujo no es tan realista como en los otros tipos de definiciones, tanto visualmente (que serán mejores las superficies curvas) como en cuanto a las características del objeto (que será mejor el modelado de sólidos).

Tendremos 3 tablas de datos en este modelo (vértices, aristas y superficies[5]), aunque dependiendo de sobre lo que se vaya a utilizar puede complicarse más o menos.

A partir de los vértices, calcularemos la *ecuación implícita* del plano donde está el polígono, $\mathrm{A}x + \mathrm{B}y + \mathrm{C}z + \mathrm{D} = 0$ (usaremos tres puntos que no estén en línea dentro del polígono, tres vértices, para determinar esos $\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}$).
La normal del plano viene dada por un vector con componentes $\mathrm{N} = (\mathrm{A}, \mathrm{B}, \mathrm{C})$ y la usaremos para saber qué es *hacia dentro* y qué *hacia fuera* del polígono.
Además, cualquier punto (x, y, z) sustituido en la ecuación del plano, si:

$$\begin{array}{rcl} = 0 & \Longrightarrow & \text{el punto está en el plano} \\ < 0 & \Longrightarrow & \text{el punto está hacia dentro de la superficie} \\ > 0 & \Longrightarrow & \text{el punto está hacia fuera de la superficie} \end{array}$$

3.2.2. Superficies curvas

A partir de superficies 2D se definen sus "extensiones" al espacio tridimensional:

$$\text{círculo} \implies \text{esfera: } x^2 + y^2 + z^2 = r^2 \ (^6)$$

$$\text{elipse} \implies \text{elipsoide: } \left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1 \ (^7)$$

$$\text{toro / toroide}$$

Estas son *curvas cuadráticas*; un poco más complejas son las llamadas *curvas supercuadráticas*, que son una generalización de las anteriores[8], formadas a partir de la incorporación de parámetros adicionales: uno para las curvas y dos para superficies.

[5] Ver transparencia.
[8] Ver transparencias.

Otro tipo más de superficies curvas son las superficies definidas por cubas cúbicas, llamadas *curvas paramétricas bicúbicas*, que se basan en las curvas paramétricas cúbicas que ya vimos (Hermite, Bézier, Splines) y cuya ecuación general en forma matricial era:

$$\mathrm{Q}(t) = \mathrm{T} \cdot \mathrm{M} \cdot \mathrm{G}$$

Este tipo de superficie va a ser un conjunto de curvas basadas en un vector S, una matriz M (que, como antes, va a depender del tipo de curva: Hermite, Bézier,...) y un vector de geometría G:

$$\mathrm{Q}(s) = \mathrm{S} \cdot \mathrm{M} \cdot \mathrm{G}$$

donde G antes era fijo: $\begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix}$ para Hermite, $\begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}$ para Bézier...

La superficie bicúbica la definimos como una *función de dos variables*:

$$\mathrm{Q}(s,t) = \mathrm{S} \cdot \mathrm{M} \cdot \begin{bmatrix} \mathrm{G}_1(t) \\ \mathrm{G}_2(t) \\ \mathrm{G}_3(t) \\ \mathrm{G}_4(t) \end{bmatrix}$$

donde G está en función de t, cada valor es una función.

Recorriendo el índice $t = 0 \ldots 1$ dibujamos una serire de curvas que nos darán la superficie:

Cada una de las funciones $\mathrm{G}_i(t)$ tiene la forma

$$\mathrm{G}_i(t) = \mathrm{T} \cdot \mathrm{M} \cdot \mathrm{G}_i$$

de modo que la ecuación de la superficie bicúbica puede expresarse:

$$\mathrm{Q}(s,t) = \mathrm{S} \cdot \mathrm{M} \cdot \begin{bmatrix} g_{11} & g_{12} & g_{13} & g_{14} \\ g_{21} & \ddots & & \vdots \\ g_{31} & & \ddots & \\ g_{41} & \cdots & & g_{44} \end{bmatrix} \cdot \mathrm{M}^{\mathrm{T}} \cdot \mathrm{T}^{\mathrm{T}}$$

donde la matriz M depende de si queremos una superficie bicúbica de Hermite, Bézier,... (nos sirven las mismas matrices que ya vimos) y la *matriz de geometría* (ya no vector) se calcula a partir de los puntos de control de la superficie.

Una superficie de Hermite o Bézier tendrá *continuidad* C^1, mientras que una superficie de Splines tendrá *continuidad* C^2. Recordemos que para obtener continuidad paramétrica C^1 es necesario que se superpongan puntos de control de las dos curvas.

3.2.3. Modelado de sólidos

En las dos secciones anteriores hemos visto la definición de modelos 3D que nos proporcionan "la parte de fuera" de los objetos, superficies de polígonos y superficies curvas, que se visualizan realizando un *render*.

Es tiempo ahora de que analicemos la simulación de otros parámetros, como la textura por ejemplo, que nos permitirán un modelado completo de los objetos, de toda su estructura interior, algo necesario para aplicaciones como simuladores de choques de coches, simuladores de robots, ...

Hay tres técnicas de representación de modelos:

Técnicas de extrusión ó revolución (barridos traslacionales / rotacionales)

Se tiene una forma 2D que se utiliza por traslación para construir el objeto 3D.

Se define una trayectoria y se hace avanzar la forma por la misma. Esa trayectoria puede ser normal o bien oblicua al plano de la figura. Se habla de *revolución* cuando se hace girar la figura en torno a un eje.

Representación de objetos por división (partición) espacial

En este tipo de representación el objeto 3D complejo se va a construir mediante la unión sin intersección de primitivas muy simples. Estas primitivas pueden ser varias, de distinto tamaño, de distinta orientación,...

Esta forma de representación es útil para trabajar con deformaciones, por ejemplo.

Geometría sólida constructiva

También en este caso tenemos un conjunto de primitivas simples, y un conjunto de operadores booleanos. Se construye un árbol (como si fuese la representación de un algoritmo) del proceso que nos lleva a a figura compleja.

Almacenar una figura de este modo ocupa muy poco espacio, puesto que sólo almacenamos las primitivas y las operaciones que hacemos con ellas (especie de formato VRML). Esos operadores son los típicos $\pm, \cup$ y además *rotación, traslación* y *escalado*.

3.3. Generación de vistas (proyecciones)

Este es el último paso previo a la visualización efectiva en pantalla de nuestro objeto. Hasta ahora hemos tratado con las coordenadas 2D y 3D típicas, usuales, a saber (x, y) y (x, y, z). A partir de este momento le añadimos una coordenada más, que denotaremos por w, a cada sistema, quedándonos (x, y, w) para 2D y (x, y, z, w) para 3D.

Se hace notar que:

↬ Los puntos $(1, 1, 2)$ y $(2, 2, 4)$ son *puntos equivalentes* en el espacio bidimensional.

↬ Por lo menos uno de los valores (coordenada) ha de ser distinto de cero, es decir, $(0, 0, 0)$ no es un punto permitido ni válido. Lo normal es que $w \neq 0$ y lo que se hace cuando esto se cumple es dividir todas las coordenadas por w: $\left(\frac{x}{w}, \frac{y}{w}, 1\right)$, denominadas **coordenadas cartesianas**.

↬ Se usan para que todas las transformaciones que apliquemos a los puntos se realicen de la misma forma, con *matrices de transformación*, de tamaño 3×3 para 2D y 4×4 para 3D.

3.3.1. El paso de 2D a 3D

$$\underbrace{\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}}_{\text{punto sobre el plano de proyección}} = \text{M} \cdot \underbrace{\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}}_{\text{punto en el espacio 3D}}$$

Las coordenadas 2D obtenidas vienen dadas por el par (x_p, y_p), que será lo que pintaremos en pantalla.

EJEMPLO 1. La **matriz de proyección o transferencia** para *proyecciones en perspectiva.*

Restricciones: $\mathsf{COP} = (0, 0, 0)$

$z = z_p$

(plano de proy. perp. al eje z a una distancia d del origen)

Se va a cumplir que

$$\frac{x}{z} = \frac{x_p}{d} \qquad \Longrightarrow \qquad \mathbf{x_p} = \frac{x \cdot d}{z} = \frac{\mathbf{x}}{\mathbf{z/d}} \qquad \text{distancia } d \text{ factor de escala}$$

(proyec. de eltos. más lejanos más pequeña) por similitud de triángulos:

Del mismo modo

$$\frac{y}{z} = \frac{y_p}{d} \qquad \Longrightarrow \qquad \mathbf{y_p} = \frac{y \cdot d}{z} = \frac{\mathbf{y}}{\mathbf{z/d}}$$

Con esto construimos la M_{transf}:

$$\text{M}_{\text{prpers}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

Operando se consigue:

$$\begin{bmatrix} x/\frac{z}{d} \\ y/\frac{z}{d} \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix} = \mathrm{M} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

donde el primer vector columna contiene la proyección del punto y el segundo las coordenadas homogéneas.

Ejemplo 2. La **matriz de proyección o transferencia** para *proyecciones paralelas.*

Ahora el plano de proyección estará en $z = 0$ y el punto de vista COP sigue estando en la coordenada z pero a una distancia $-d$: $(0, 0, -d)$

Igual que antes, por similitud de triángulos, ahora se cumple:

$$\frac{x}{z+d} = \frac{x_p}{d} \qquad \Longrightarrow \qquad \mathbf{x_p} = \frac{d \cdot x}{z+d} = \frac{\mathbf{x}}{(\mathbf{z}/\mathbf{d}) + \mathbf{1}}$$

$$\frac{y}{z+d} = \frac{y_p}{d} \qquad \Longrightarrow \qquad \mathbf{y_p} = \frac{d \cdot y}{z+d} = \frac{\mathbf{y}}{(\mathbf{z}/\mathbf{d}) + \mathbf{1}}$$

En este caso la matriz:

$$\mathrm{M}_{\text{prpers'}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

Nótese que hemos obtenido un resultado para proyecciones en perspectiva, todavía.

¿Cuál es la ventaja sobre la anterior? ¿Qué pasa con COP si aumentamos d? En la proyección paralela el punto de vista está en el infinito: si aumentamos d al infinito obtenemos $\mathrm{M}_{\text{prparal}}$:

$$\mathrm{M}_{\text{paral}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Realmente lo que se hace es eliminar la componente z del punto y quedarse con la(s) otra(s) coordenada(s).

Forma más general para deducir la matriz M para cualquier tipo de proyección

El plano de proyección va a estar en la posición $(0, 0, z_p)$ y es perpendicular al eje z (igual que en los dos casos que acabamos de ver).

El centro de proyección lo definimos a partir del punto a partir del cual se define el plano $(0, 0, z_p)$:

$$\mathsf{COP} = (0, 0, z_p) + \underbrace{\mathrm{Q}}_{\text{distancia}} \cdot \underbrace{(d_x, d_y, d_z)}_{\text{vector de dirección}}$$

Calculamos la ecuación del proyector (línea que une el COP y P), podemos hacer:

$$\mathrm{COP} + t(\mathrm{P} - \mathrm{COP})$$

con

$$0 \leqslant t \leqslant 1$$

(igual que en las curvas paramétricas).

Para $t = 0$ el punto que tenemos es justamente COP, y para $t = 1$, P. Cualquier otro punto de la recta vale:

$$\begin{array}{lcl} \text{en } x & \rightarrow & x' = \mathrm{Q}d_x + (x - \mathrm{Q}d_x)t \\ \text{en } y & \rightarrow & y' = \mathrm{Q}d_y + (y - \mathrm{Q}d_y)t \\ \text{en } z & \rightarrow & z' = (z_p + \mathrm{Q}d_z) + (z - (z_p + \mathrm{Q}d_z))t \end{array}$$

Lo que queremos es conocer el punto en que se produce la intersección.

Sabemos que $z' = z_p$, así que despejando podemos averiguar t:

$$t = \frac{[z_p - (z_p + \mathrm{Q}dz)]/ - \mathrm{Q}dz}{[z - (z_p + \mathrm{Q}dz)]/ - \mathrm{Q}dz} = \frac{z_p - (z_p + \mathrm{Q}dz)}{z - (z_p + \mathrm{Q}dz)} = \boxed{\frac{1}{\dfrac{z_p - z}{\mathrm{Q}dz} + 1}}$$

De modo que los "valores definitivos" son:

$$x_p = \frac{x - z\dfrac{dx}{dz} + z_p\dfrac{dx}{dz}}{\dfrac{z_p - z}{\mathrm{Q}dz} + 1}$$

$$y_p = \frac{y - z\dfrac{dy}{dz} + z_p\dfrac{dy}{dz}}{\dfrac{z_p - z}{\mathrm{Q}dz} + 1}$$

$$z_p = z_p\frac{\dfrac{z_p - z}{\mathrm{Q}dz} + 1}{\dfrac{z_p - z}{\mathrm{Q}dz} + 1} = \frac{-z\dfrac{z_p}{\mathrm{Q}dz} + \dfrac{z_p + z_p\mathrm{Q}dz}{\mathrm{Q}dz}}{\dfrac{z_p - z}{\mathrm{Q}dz} + 1}$$

Y la matriz de transformación general:

$$\mathrm{M}_{\mathrm{general}} = \begin{bmatrix} 1 & 0 & -\frac{dx}{dz} & z_p\frac{dx}{dz} \\ 0 & 1 & -\frac{dy}{dz} & z_p\frac{dy}{dz} \\ 0 & 0 & -\frac{z_p}{\mathrm{Q}dz} & \frac{z_p^2}{\mathrm{Q}dz} + z_p \\ 0 & 0 & -\frac{1}{\mathrm{Q}dz} & \frac{z_p}{\mathrm{Q}dz} + 1 \end{bmatrix}$$

que se configura según el tipo de proyección sustituyendo los parámetros por[9]:

	z_p	Q	dx	dy	dz
$\mathrm{M}_{\mathrm{per}}$	d	d	0	0	-1
$\mathrm{M}_{\mathrm{per'}}$	0	d	0	0	-1
$\mathrm{M}_{\mathrm{par}}$	0	∞	0	0	-1
$\mathrm{M}_{\mathrm{Caballera}}$	0	∞	$\cos\alpha$	$\sin\alpha$	-1
$\mathrm{M}_{\mathrm{Cabinet}}$	0	∞	$\frac{\cos\alpha}{2}$	$\frac{\sin\alpha}{2}$	-1

[9]Ver transparencias.

Capítulo 4

Transformaciones geométricas

Hasta ahora tanto el dibujo de rectas, como de círculos, etc. ha sido *estático*. Pero tras realizar el dibujo puede darse interacción con el usuario, con la posibilidad de que éste modifique el modelo (color, tamaño,...) o el punto de vista, debiendo redibujarse el original para plasmar los cambios.

De cómo realizar *transformaciones* sobre objetos ya definidos trataremos en este tema. Las principales son:

- **Traslación**.
- **Escalado**.
- **Rotación**.
- **Otras**.

Ninguna de ellas es demasiado compleja. Comenzaremos con su aplicación en 2D y al final del capítulo veremos cómo se hace extensible al espacio tridimensional.

4.1. Traslaciones

En una **traslación** tenemos una diferencia en x y en y que queremos aplicar a todos los puntos de una figura:

$$(dx, dy) \quad \Rightarrow \quad \begin{array}{rcl} x' & = & x + dx \\ y' & = & y + dy \end{array}$$

Usaremos notación matricial:

$$P' = \begin{bmatrix} x' \\ y' \end{bmatrix} \qquad P = \begin{bmatrix} x \\ y \end{bmatrix} \qquad T = \begin{bmatrix} dx \\ dy \end{bmatrix}$$

$$P' = P + T$$

4.2. Escalados

En este caso queremos modificar el tamaño de la figura:

$$\begin{array}{rcl} x' & = & x \cdot S_x \\ y' & = & y \cdot S_y \end{array} \qquad \begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}}_{S} \begin{bmatrix} x \\ y \end{bmatrix} \qquad \Longleftrightarrow \qquad P' = S \cdot P$$

El caso es que esta operación, realizada tal como acabamos de describir, además de escalar, *mueve*:

- Si $S_x, S_y < 1$, el escalado hace la figura más pequeña pero, además, la coloca más cerca del origen de coordenadas.
- Si $S_x, S_y > 1$, el escalado hace la figura más grande y la sitúa más lejos del origen de coordenadas.

Lo que se debe hacer para obtener el resultado deseado (el escalado de la figura, pero en la posición en que se encuentra) es trasladar el objeto al origen aplicando $\mathrm{M}_{\mathrm{traslac}}$, aplicarle luego el escalado y por último devolverla a su sitio:

$$[\underbrace{T(x,y)}_{\text{regreso}} \cdot \underbrace{S}_{\text{escalado}} \cdot \underbrace{T(-x,-y)}_{\text{al origen}}]$$

Si $S_x = S_y$ (los dos factores son iguales) se habla de **escalado uniforme**; en otro caso el escalado se dice **diferencial** (hace variar las proporciones del objeto).

4.3. Rotaciones

Giramos la figura un numero θ de grados:

$$\begin{array}{rcl} x' &=& x\cos\theta - y\sin\theta \\ y' &=& x\sin\theta + y\cos\theta \end{array} \qquad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad \Longleftrightarrow \qquad P' = R \cdot P$$

También en este caso nos ocurre lo mismo que con el escalado: si aplicamos lo anterior tal cual vamos a rotar la figura, pero con respecto al origen de coordenadas, que normalmente no es lo que pensamos en hacer cuando hablamos de rotar un objeto.

Si lo que queremos hacer es rotar con respecto a un punto de la propia figura, la llevaremos al origen, la rotaremos el número de grados θ y la devolveremos a su posición:

$$[T(x,y) \cdot R(\theta) \cdot T(-x,-y)]$$

Los *ángulos positivos* dan como resultado una rotación en la dirección contraria al giro de las agujas del reloj, mientras que los *ángulos negativos* la producen en la misma dirección en que éstas giran.

4.4. Transformaciones en coordenadas homogéneas

Las transformaciones son, por tanto, como acabamos de ver, *operaciones con matrices*:

$$\begin{array}{ll} P' = T + P & \text{traslación} \\ P' = S \cdot P & \text{escalado} \\ P' = R \cdot P & \text{rotación} \end{array}$$

Claro que podrían unificarse aún más si consiguiésemos que fuesen el mismo tipo de operación.

Si usamos *coordenadas homogéneas* podríamos unificarlas y expresar todas las transformaciones como **productos de matrices**.

Traslación en coordenadas homogéneas

$$P' = T(d_x, d_y) \cdot P \qquad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} x + dx \\ y + dy \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \cdot \underbrace{\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}}_{\substack{\text{pto. en coord.} \\ \text{homogéneas}}}$$

Escalado en coordenadas homogéneas

$$P' = S(S_x, S_y) \cdot P \qquad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} xS_x \\ yS_y \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Rotación en coordenadas homogéneas

$$P' = R(\theta) \cdot P \qquad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

4.5. Composición de transformaciones

Traslación sobre traslación

La *traslación* verifica la *aditividad*, de modo que podemos hacer:

$$\begin{aligned} P' &= T(dx_1, dy_1) \cdot P \\ P'' &= T(dx_2, dy_2) \cdot P' \end{aligned}$$

$$P'' = T(dx_2, dy_2)[T(dx_1, dy_1) \cdot P] = \begin{bmatrix} 1 & 0 & dx_1 + dx_2 \\ 0 & 1 & dy_1 + dy_2 \\ 0 & 0 & 1 \end{bmatrix}$$

Escalado sobre escalado

Para el *escalado* se verifica algo similar, se *multiplican* los factores:

$$\begin{aligned} P' &= S_1 \cdot P \\ P'' &= S_2 \cdot P' \end{aligned}$$

$$P'' = S_2(S_1 \cdot P) = \begin{bmatrix} S_{x_1}S_{x_2} & 0 & 0 \\ 0 & S_{y_1}S_{y_2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotación sobre rotación

Y, por último, para la *rotación* lo que se haría sería *sumar los ángulos.*

$$\begin{aligned} P' &= R(\theta) \cdot P \\ P'' &= R(\phi) \cdot P' \end{aligned}$$

$$P'' = R(\phi)[R(\theta) \cdot P] = \begin{bmatrix} \cos(\theta+\phi) & -\sin(\theta+\phi) & 0 \\ \sin(\theta+\phi) & \cos(\theta+\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Es decir, cualquier combinación de transformaciones nos da una matriz:

$$\begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

4.6. Otras transformaciones

Las tranformaciones vistas hasta ahora se engloban bajo el nombre de **transformaciones afines**, que se caracterizan porque conservamos el paralelismo entre líneas, pero no las distancias (longitudes) ni los ángulos.

Otras transformaciones posibles, las últimas que veremos, son las de **Shear**[1] y las **Reflexiones**.

4.6.1. Shear

El *esquilado* también es una transformación afín, y puede realizarse sobre el eje x o sobre el eje y. La matriz de transformación es:

$$\mathrm{M}_x = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = [\ x+ay \quad y \quad 1\ ]^t$$

para el eje x. Como se puede observar, depende del parámetro a (cte. de proporcionalidad) y de la distancia en el eje y.

[1] *Esquilado, sesgo.*

En el eje y:

$$\mathrm{M}_y = \begin{bmatrix} 1 & 0 & 0 \\ b & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

4.6.2. Reflexiones

En las transformaciones de *reflexión* vamos a cambiar la figura como si se reflejara en un espejo. El "espejo" será lo que marcaremos como *eje de reflexión*.

Reflexión sobre el eje x

Simplemente cambiamos las coordenadas y de la figura (por sus opuestas), manteniendo las x:

$$Re_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Reflexión sobre el eje y

En este caso cambiamos las coordenadas x por sus opuestas, manteniendo las y:

$$Re_y = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Reflexión sobre el origen

Si queremos reflejar sobre el origen tomaremos como eje de reflexión la recta $x = -y$:

$$Re_{(0,0)} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Reflexión sobre cualquier punto

No obstante, no será lo más usual que queramos reflejar sobre un eje. Veamos cómo se hace usando cualquier línea recta como eje.

Por ejemplo, supongamos que queremos reflejar usando como eje de reflexión la recta $x = y$. Podemos girar la figura 45^o con respecto al origen, para hacer coincidir el eje de reflexión con el x ó el y. Entonces aplicaríamos Re_x ó Re_y y por último, rotaríamos de nuevo -45^o:

$$R(-45^o) \cdot Re_x \cdot R(45^o)$$

Lo que intentaremos hacer es, pues, combinar las técnicas ya estudiadas. Para un eje de reflexión que no pase por el origen, $y = mx + b$ seguiremos los siguientes pasos:

1. Trasladaremos la recta al origen.
2. Calcularemos los grados que forma con uno de los ejes.

De esta manera ya estamos en el caso anterior y aplicamos:

$$[T(0,b)R(-\theta) \cdot Re_x \cdot R(\theta)T(0,-b)] \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Es decir, trasladamos (en y) el objeto b posiciones hasta el origen, lo rotamos θ grados (se sacan de la pendiente) para ajustar a uno de los ejes, aplicamos la reflexión y luego deshacemos los pasos anteriores.

Además, siempre que las matrices contengan 1's, el objeto se reflejará a la misma distancia. Si usamos un número *mayor que -1* el reflejo quedará más cerca del eje, y si usamos un número *menor que -1*, quedará más lejos.

4.7. Optimización

Evidentemente, nos interesa que todo lo que venimos viendo hasta ahora (su dibujado) se lleve a cabo lo más rápido posible. Veamos algunas estrategias para conseguirlo.

4.7.1. Optimización de la composición de transformaciones

Ya vimos cómo agrupar varias transformaciones del mismo tipo. Veremos ahora que las transformaciones comunes, las que se ejecutan un mayor número de veces, deberíamos calcularlas directamente para una mayor eficiencia. Estas operaciones incluyen la aplicación de varias matrices de transformación cada vez que las ejecutamos.

Traslación + escalado

Es un escalado sobre un punto que no es el origen.

$$T(x_1, y_1) \cdot S(S_x, S_y) \cdot T(-x_1, -y_1) = \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} S_x & 0 & x_1(1 - S_x) \\ 0 & S_y & y_1(1 - S_y) \\ 0 & 0 & 1 \end{bmatrix} = S_{(x_1, y_1)}(S_x, S_y)$$

Rotación + traslación

Es una rotación desde un punto distinto del origen.

$$T(x_1, y_1) \cdot R(\theta) \cdot T(-x_1, -y_1) = \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & x_1(1-\cos\theta) + y_1 \sin\theta \\ \sin\theta & \cos\theta & y_1(1-\cos\theta) - x_1 \sin\theta \\ 0 & 0 & 1 \end{bmatrix} = R_{(x_1,y_1)}(\theta)$$

Construir estas matrices (que serán probablemente las que nos proporcionen el resultado deseado) nos hará ganar en rapidez, ya que aplicamos una sola en lugar de tres[2].

4.7.2. Optimización del cálculo

Cualquier matriz de transformación, como hemos dicho, es de la forma:

$$\mathrm{M} = \begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

Aplicada sobre un punto en coordenadas homogéneas, $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$, son 9 productos y 6 sumas para obtener cada punto "transformado".

Sin embargo, si nos damos cuenta, la última fila nunca varía, de suerte que lo único que realmente deberíamos calcular sería:

$$x' = xr_{11} + yr_{12} + t_x$$

$$y' = xr_{21} + yr_{22} + t_y$$

Es decir, 4 productos y 4 sumas.

Otra forma de optimizar, para *rotaciones de un número de grados muy pequeño*[3], teniendo en cuenta que $\cos 0^o = 1$ y aproximando, aceleramos el dibujo si calculamos:

$$x' = x\cos\theta - y\sin\theta \simeq x - y\sin\theta$$

$$y' = x\sin\theta + y\cos\theta \simeq x\sin\theta + y$$

Que es mucho más rápido porque se eliminan casi la mitad de las operaciones sobre ángulos, claro que cualquier tipo de aproximación siempre acumula error, de modo que debe recordarse aplicarla siempre sobre los datos originales, no unas aproximaciones sobre otras, calculando de vez en cuando el valor real.

[2] ¡En el caso de la reflexión podrían ser hasta 5 matrices!

[3] Por ejemplo, cuando el usuario examina un objeto girándolo 360^o.

4.8. Transformaciones en 3D

Como sabemos, los puntos en 3D se representan en coordenadas homogéneas: $\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$

Consecuentemente, tendremos ahora *matrices de transformación* 4×4.

Traslación

$$\mathrm{M}_{(t_x,t_y,t_z)} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Escalado

$$S_{(S_x,S_y,S_z)} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotación

La rotación es un poco más complicada. Distinguiremos un tipo de rotación para cada uno de los tres ejes:

ROTACIÓN EN EJE Z: Consiste en rotar sobre el plano xy:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

ROTACIÓN EN EJE X: Rotar sobre el plano yz:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

ROTACIÓN EN EJE Y: Rotación sobre el plano zx:

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

El resto de consideraciones son análogas a las que hacíamos para 2D:

- Se compondrán las matrices siempre que sea posible para optimizar el cálculo.
- Las mismas consideraciones de eficiencia: no aplicar directamente el producto de matrices (recordar que la última fila nunca varía) y en rotaciones aproximar los cosenos sin olvidar controlar la acumulación del error.
- Una matriz de transformación tiene siempre la forma:

$$\mathrm{M} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Capítulo 5

Luz monocroma y color

El tema del color es un tema subjetivo y difícil, debido a que en el color y en nuestra apreciación del mismo influyen distintos factores. Un objeto puede ser de un color, pero la iluminación puede hacernos cambiar la idea sobre éste, el fondo contra el que lo estamos viendo también, nuestro sistema visual lo mismo, incluso las características (material) del objeto cambian la apreciación que tenemos de su color. Todas estas características, ajenas, externas al color en sí, le afectan; hay que tenerlas en cuenta, pero es difícil implementarlas.

5.1. Luz monocroma

Antes de meternos con el color a fondo veremos un poco de **luz monocroma**. Cuando trabajamos con aplicaciones que la usan, el factor que nos interesa o vamos a manejar es la **intensidad** o **luminosidad** (0 ≡ negro, 1 ≡ blanco). Ahora bien, estos valores hay que *discretizarlos* y traducirlos a un número de bits ⇒ grises.

$$2^n = m \textbf{ valores distintos}$$

donde n es el número de bits. El rango [0,1] se reparte entre esos m valores. Por ejemplo, si $n = 8$, repartimos [0,1] entre 256.

La primera idea que se nos ocurre para realizar este reparto es una *división uniforme* de ese rango disponible. Sin embargo, el sistema visual humano no funciona así, los incrementos en los valores de grises si se hacen por una cantidad fija no se aprecia un gradiente homogéneo. Para ver un gradiente homogéneo se necesita aumentar siempre la misma proporción, no el mismo valor.

$$\Delta = 0'01 \qquad 0 \quad 0'01 \quad 0'02 \quad 0'03 \quad \dots \quad 1$$

$$0'1 \xrightarrow{10\,\%} 0'11 \qquad \Delta = 0'01$$

$$0'5 \xrightarrow{10\,\%} 0'55 \qquad \Delta = 0'05$$

Es decir, en lugar de un *incremento lineal*, haremos un *incremento logarítmico*. Siguiendo con nuestro ejemplo de 8 bits:

$$I_0 \rightsquigarrow I_1 = r \cdot I_0 \rightsquigarrow I_2 = r \cdot I_1 = r^2 \cdot I_0$$
$$\dots$$
$$I_{255} = r \cdot I_{254} = r^2 I_{253} = \dots = r^{255} I_0 = 1$$

Donde I_0 es un *valor de inicio*, el más cercano al negro, pero no exactamente el 0. Lo que debemos es calcular r:

$$r = \left(\frac{1}{I_0}\right)^{\frac{1}{255}}$$

Así, cualquier valor de intensidad es:

$$I_j = r^j I_0 = \left(\frac{1}{I_0}\right)^{\frac{j}{255}} \cdot I_0 = I_0^{\frac{(255-j)}{255}} \qquad \text{con } 0 \leqslant j \leqslant 255$$

Generalizando para $n+1$ intensidades en lugar de 256:

$$\boxed{r = \left(\frac{1}{I_0}\right)^{\frac{1}{n}}} \qquad \boxed{I_j = I_0^{\frac{(n-j)}{n}}}$$

para $0 \leqslant j \leqslant n$.

Dentro de un monitor la luminosidad nunca es cero, el mínimo que se consigue es $I_0 = \left[\frac{1}{200}, \frac{1}{4}\right]$ de la $I_{\text{máx}} = 1$. La relación entre la *luminosidad máxima* y la *luminosidad mínima* se denomina **rango** (intervalo) **dinámico**:

$$\text{rango dinámico } = \frac{1}{I_0}$$

Es una característica técnica de los monitores, y será mejor cuanto más grande sea[1].
Una vez que sabemos el número de bits, pues, definiremos nuestras intensidades.

La intensidad también se puede expresar en función del número de electrones que se lanzan contra la pantalla, aunque la relación no es lineal sino a través de dos parámetros específicos del monitor CRT:

$$I_j = kN^\gamma$$

Como decimos, tanto k como γ son específicos para cada monitor; lo que se hace para calcular esta **corrección gamma** es definir tablas que relacionan el número de electrones N con la intensidad I_j.

Una vez que hemos conseguido que la diferencia entre un nivel de gris y otro sea la misma, la siguiente cuestión es ¿cuál es el número mínimo de bits para conseguir *no detectar los saltos entre dos niveles de gris* consecutivos? Pues esto se consigue si el valor $r \leqslant 1'01$.
De hecho, r nunca podría ser $r < 1$ (es su límite inferior), ya que $I_{j+1} = rI_j$.

[1]El rango dinámico, la resolución y la profundidad del color son factores determinantes para casos, por ejemplo, como los diagnósticos de radiografías sobre monitor.

Establecido el incremento r, calcularemos el número de intensidades de que dispondremos despejando en

$$r = \left(\frac{1}{I_0}\right)^{\frac{1}{\mathbf{n}}}$$

para obtener:

$$1'01 = \left(\frac{1}{I_0}\right)^{\frac{1}{\mathbf{n}}} \qquad \Rightarrow \qquad \mathbf{n} = \log_{1'01}\left(\frac{1}{I_0}\right)$$

de modo que, como se puede ver, el número de intensidades **n** depende del rango dinámico $\frac{1}{I_0}$; a mayor rango dinámico (mayor calidad del monitor), mayor número de bits se requieren para un mayor número de intensidades ó niveles de gris.

La siguiente tabla relaciona estos factores:

	$\frac{1}{I_0}$	n
CRT	50-200	400-550
Impresoras	10-100	250-500

En el caso de las impresoras, el número n es un número de intensidades teórico, que disminuye en la práctica (según la calidad del papel, tinta, el modo de impresión,...) de forma que a partir de 64-100 ya no se distiguen los saltos.

5.1.1. Implementaciones para los niveles de gris

Half-Tone

El **half-tone** es una primera aproximación, útil para imágenes con pocos niveles de gris.

En ella, el fondo es blanco y los puntos son negros, sin grises, pero no tienen todos el mismo diámetro. Ahora bien, si sólo tenemos dos intensidades (B/N) ¿cómo definimos una gama de gris? Lo que se hace es explotar una característica del sistema visual, la **integración espacial**. Según esta característica, nosotros, cuando miramos algo, si está lejos, tendemos a integrar la información de color de un área pequeña a un solo valor.

Esto en el ordenador se simula haciendo una **división de la resolución espacial**. No podemos poner píxeles más o menos pequeños / grandes, y si sólo disponemos de B/N (computadores bitono 0/1), lo que se hace es agrupar varios píxeles de forma que los pensamos como si fueran uno[2].

Según el número de píxeles que agrupemos tendremos un número de tonalidades de gris. Por ejemplo, si los agrupamos en matrices de 2×2, tendremos 5 tonos de gris. En general,

$$n \times n \quad \longrightarrow \quad n^2 + 1 \text{ tonos de gris}$$

[2]Es la técnica usada en la impresión de fotografías en los periódicos.

La forma de definir la matriz es importante, ya que nos da una *ordenación* de los píxeles que tenemos que ir encendiendo para pasar de un gris a otro sucesivamente. Para el caso 3×3 un ejemplo sería:

$$\begin{bmatrix} 6 & 8 & 4 \\ 1 & 0 & 3 \\ 5 & 2 & 7 \end{bmatrix}$$

Las **normas** que han de seguirse a la hora de definir una matriz con esta finalidad son:

1ª) *Evitar artefactos visuales.* Por ejemplo, si tenemos la matriz:

$$\begin{bmatrix} 6 & 8 & 4 \\ 1 & 2 & 3 \\ 5 & 0 & 7 \end{bmatrix}$$

el rellenar una zona amplia con puntos de nivel de gris 3 nos hará ver una serie de rayas horizontales[3].

2ª) *Una secuencia creciente de píxeles en los niveles de gris*, es decir, que los píxeles que cubran una intensidad estén presentes en la siguiente, añadiéndoseles uno más[4].

3ª) La secuencia tiene que tener un *crecimiento hacia afuera* para simular el efecto de la intensidad (para simular ese punto que crece).

4ª) *Los píxeles han de estar agrupados.*

La idea del *half-tonning* se puede seguir usando en dispositivos con pocos niveles de gris cuando queremos ampliar la profundidad del color. De este modo, a los m niveles de gris del dispositivo se añadirán los proporcionados por la matriz $n \times n$ del half-tonning, teniéndose los tonos de gris totales[5]:

$$n^2 * (m-1) + 1$$

Por ejemplo, en un dispositivo con 4 niveles de gris, utilizando una matriz half-tonning 2×2 se obtendrían $(2 \times 2) * (4-1) + 1$.

El problema que presenta el hacer esto es que *reducimos la resolución del dispositivo a la mitad*[6]. Sin embargo, para mejorar esto, podemos utilizar la técnica de la **difusión de error**[7], que consiste en calcular la *diferencia* entre lo que realmente pretendemos representar y la capacidad del dispositivo:

$$\text{DIF} = \text{REAL} - \text{DISPOSIT.}$$

Dicha diferencia se "reparte" entre los píxeles contiguos que quedan por pintar de la forma que se indica a continuación, donde el valor indicado en la casilla es la fracción de la diferencia que se le suma al valor real del pixel correspondiente.

[3] Ver figura 13.11 en las fotocopias.
[4] Ver figura 13.12.
[5] Ver figura 13.13.
[6] En general, si la matriz es de $n \times n$, la reducimos en un factor $\frac{1}{n}$.
[7] Algo que la mayoría de las impresoras admiten como una opción más.

Los cuatro errores que se difunden deben sumar exactamente 1, no se pueden consentir errores de redondeo. Para ello, pueden calcularse tres de ellos y el ultimo como lo que reste entre 1 y su suma. Se obtienen mejores resultados aún si se alterna la dirección de barrido (izquierda-derecha, derecha-izquierda).

5.2. Luz coloreada

En luz monocroma sólo tenemos un parámetro en cuenta, la intensidad. Ya vimos las particularidades que presentaba el ojo humano con respecto a la percepción de ésta y no es descaminado pensar que también se presentarán "curiosidades" en este caso.

Los parámetros que manejaremos ahora son:

- **Hue-matiz** (*tinte*), distinción de un color de otro.
- **Saturación**, que indica la distancia de un color con respecto al gris correspondiente (por la intensidad). Los *colores primarios* están totalmente saturados, mientras que los *colores pastel* están poco saturados, incluyen más luz.
- **Luminosidad ó intensidad**, correspondiente al parámetro de intensidad en la luz monocroma.

La apreciación de los colores es también (y sobretodo) subjetiva.

Existe una Rama de la Física llamada **Colorimetría** que se dedica al estudio de los colores, analizando algunos parámetros más a la hora de definir cada uno de ellos:

- **Longitud de onda** dominante.
- **Pureza** de excitación, cantidad de luz blanca del color (en cierto modo equivalente a la *saturación*). Los colores puros no tienen luz blanca (están muy saturados), mientras que el resto de la gama cada vez tiene más.
- **Luminancia**, cantidad o intensidad de luz.

Recordemos una vez más que tenemos que tener en cuenta el sistema visual humano, que cuenta con 3 tipos de receptores (3 tipos de *conos*) para el color en la retina.

Los monitores CRT tienen también 3 tipos de fósforo (P).

El sistema RGB no permite visualizar todos los colores posibles[8]. Por ello, en 1931 la Comisión Internacional de l'Ecleriage (CIE) elaboró un nuevo sistema de *respuesta a los colores*, definiendo tres colores primarios estándar, (x, y, z), para reemplazar a rojo, verde y azul. La combinación lineal de estos colores primarios sí que permite la representación de todos los colores visibles[9].

Existe una transformación de colores (x, y, z) a RGB.

5.2.1. Modelos de color

Un **modelo de color** es una especificación de un sistema tridimensional de coordenadas de colores. Todos los *modelos de color* que vamos a ver son subconjuntos dentro de los colores CIE, van a representar unas coordenadas 3D de color dentro de ese subconjunto. Todos y cada uno de ellos sólo es capaz de representar un subconjunto de los colores visibles.

Distinguimos *modelos de color orientados al hardware*, como:

- **RGB**, el más conocido, usado en los monitores CRT.
- **CMY**, usado en dispositivos de impresión en color.
- **YIQ**, sistema para la transmisión de TV en color.

Ninguno de ellos es muy sencillo de usar, pues no se relacionan directamente con los conceptos intuitivos de matiz, saturación y brillo. Se desarrollaron otros con este fin: **HSV**, **HLS**.

También veremos cómo convertir entre unos y otros.

Modelo RGB

Este modelo está basado en los tres *colores primarios*: **rojo**, **verde** y **azul**. Además, es un modelo ***aditivo***, se basa en la suma de los mencionados colores primarios. Si al sumarlos, dos colores dan blanco, se denominan **complementarios**.

El subconjunto de interés es el cubo unidad. El negro es el $(0, 0, 0)$, el blanco el $(1, 1, 1)$ y la diagonal entre ellos representa los niveles de gris.

Aunque todos los monitores usen RGB, la gama de colores que van a poder visualizar (el "cubo") va a ser levemente distinta porque depende de las características físicas del mismo, como por ejemplo de las propiedades del fósforo que tengan en las pantallas.

Modelo CMY

Este otro modelo está basado en la *resta* de colores, en este caso de **cyan**, **magenta** y **yellow**, que son los *complementarios* del rojo, el verde y el azul respectivamente (por eso se denominan ***primarios sustractivos***), en lugar de en la suma.

Se usa con respecto a labores de impresión, y eso justifica su *modus operandi*, ya que en pantalla en principio partimos de un fondo negro, que es nuestro origen y a partir de ahí sumamos para obtener distintos colores. Sobre papel, partimos del fondo blanco, que es ahora el origen de nuestro cubo (el subconjunto del sistema de coordenadas cartesianas

[8] Remitimos una vez más a las fotocopias.

[9] Sin ponderaciones negativas.

para este modelo es el mismo que para RGB, sólo que está al revés: el blanco en el $(0,0,0)$ y el negro en el $(1,1,1)$), y debemos *restar* para conseguir los diferentes colores:

$$\begin{bmatrix} \mathrm{C} \\ \mathrm{M} \\ \mathrm{Y} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} \mathrm{R} \\ \mathrm{B} \\ \mathrm{B} \end{bmatrix}$$

Del mismo modo:

$$\begin{bmatrix} \mathrm{R} \\ \mathrm{G} \\ \mathrm{B} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} \mathrm{C} \\ \mathrm{M} \\ \mathrm{Y} \end{bmatrix}$$

Una variación de este modelo que se emplea en el proceso de impresión, en la reproducción impresa de los colores, es el **CMYB**[10], donde **B** denota el negro, con el que se consigue un color más verdadero.

Modelo YIQ

El modelo **YIQ** se emplea en la transmisión de televisión en color en Estados Unidos, que usa el estándar NTSC[11]. Se trata de una recodificación de RGB para obtener mayor eficiencia de transmisión y sobre todo para tener compatibilidad con la televisión en blanco y negro.

La componente **Y** de YIQ corresponde a la *luminancia*, se define como el primario ***y*** CIE. Las componentes **I** y **Q** codifican el color. Las televisiones en blanco y negro sólo representarán la componente Y de la señal.

La correspondencia RGB-YIQ se define:

$$\begin{bmatrix} \mathrm{Y} \\ \mathrm{I} \\ \mathrm{Q} \end{bmatrix} = \begin{bmatrix} 0'299 & 0'587 & 0'114 \\ 0'569 & -0'275 & -0'321 \\ 0'212 & -0'528 & 0'311 \end{bmatrix} \cdot \begin{bmatrix} \mathrm{R} \\ \mathrm{B} \\ \mathrm{B} \end{bmatrix}$$

Las cantidades reflejan la importancia relativa del verde y el rojo, así como la poca importancia relativa del azul en la brillantez. La inversa de la matriz RGB a YIQ se usa para la conversión YIQ a RGB:

$$\begin{bmatrix} \mathrm{R} \\ \mathrm{G} \\ \mathrm{B} \end{bmatrix} = \begin{bmatrix} 1 & 0'956 & 0'620 \\ 1 & -0'272 & -0'647 \\ 1 & -1'108 & 1'705 \end{bmatrix} \cdot \begin{bmatrix} \mathrm{Y} \\ \mathrm{I} \\ \mathrm{Q} \end{bmatrix}$$

El modelo YIQ aprovecha dos propiedades útiles de nuestro sistema visual: somos más sensibles a cambios en la luminancia que a cambios en el matiz o la saturación y además los objetos que cubren una parte pequeña de nuestro campo visual producen una sensación de color limitada. Ambos datos sugieren que pueden y deben usarse más bits de ancho de banda para representar Y que para la representación de I y Q. De hecho, la codificación NTSC proporciona doble ancho de banda para el primero que para los otros dos.

[10] **CMYK** en inglés.

[11] En Europa se usa el sistema PAL.

Otros modelos

Los modelos RGB, CMY e YIQ están orientados al hardware, como ya hemos dicho. Por el contrario, existen otros modelos *orientados a la percepción del color*:

- **HSV**, *hue* (matiz), *saturation*, *value*.
- **HLS**, *hue*, *lightness*, *saturation*.

5.2.2. Interpolación de color

La **interpolación de color** se refiere al paso gradual desde un color A hasta otro color distinto B recorriendo la gama entre ellos. Es importante para:

√ Sombreados en 3D.

√ Antialiasing.

√ Fundido de imágenes.

Los resultados de la interpolación dependen del modelo de colores que se escoja, así que debemos hacerlo con cuidado. Si la conversión de un modelo de colores a otro transforma una línea recta (que representa la trayectoria de interpolación) en un modelo de colores a una línea recta en el otro modelo, entonces los resultados de la interpolación lineal en ambos modelos serán iguales. Esto ocurre entre RGB, CMY, YIQ y CIE. Sin embargo, una línea recta en el modelo RGB generalmente no se transformará en una línea recta en el modelo HSV o HLS.

5.2.3. Reproducción impresa de los colores

Como ya hemos comentado, mientras que el modelo RGB se utiliza para los monitores, es el modelo CMY el utilizado para la impresión del color. Normalmente suele emplearse la variación de la que hablamos, el CMYB, y suele realizarse además una *reducción de niveles* de color al imprimir, ya que las impresoras mezclan el color intentando imitar la forma en que el SVH[12] los integra, y por otra parte estos dispositivos tienen menos profundidad de color.

Las opciones que se manejan son la interpolación de valores ($+$RG, $-$B), dando menos resolución al azul, o la utilización de una *paleta de colores* más reducida.

Es difícil tener un color en pantalla y que se obtenga un fiel reflejo por la impresora. Incluso entre dos impresoras, si son de distinta tecnología.

5.2.4. Consideraciones sobre la utilización del color

En general no se suele pensar sobre el uso del color en un programa, pero debería prestársele más atención.

Las consideraciones a tener en cuenta son, entre otras:

↬ Cualquier programa debe ser validado por usuarios (recordemos la subjetividad del color).

[12] Abreviatura de Sistema Visual Humano.

- Existen una serie de **reglas** para combinar colores:
  - Debe seguirse un *criterio*.
  - Cuando en un gráfico, por ejemplo, hay pocos colores, el fondo puede ser el complementario de uno de ellos. En caso contrario (muchos colores), es más conveniente un fondo neutro: gris.
  - Hay que tener en cuenta que algunos colores están asociados a ciertas ideas ("números rojos").
  - Tampoco debemos olvidar que suelen asociarse los objetos que tienen el mismo color (por eso los objetos de control estándares suelen ser de colores neutros).
- Además de la *psicología* también debe tenerse en cuenta la **fisiología** humana, cómo funciona el SVH:
  - Nosotros percibimos el color como la luz con diferentes *longitudes de onda* λ que van del azul al rojo. Cada longitud de onda está asociada a un nivel de energía distinto, y esto se asocia con el hecho de que colores distintos en la misma posición parezca sin embargo que están a distinta distancia (algo rojo parece más grande).
  - El ojo humano es más sensible a variaciones de intensidad que de color, de modo que la líneas, texto o detalles que deban destacar deben ir en color blanco o negro.
  - Somos poco sensibles a los cambios en el azul, de modo que no se deben poner juntos colores que sólo cambien en su tonalidad azul, que sólo se diferencien en la cantidad de azul que contienen. Tampoco se recomienda colocar juntos colores con intensidades muy parecidas (blanco-amarillo, negro-azul,...).

Capítulo 6

Mejoras en la visualización

En este tema trataremos tres aspectos que nos pueden ayudar a mejorar considerablemente el aspecto y el acabado final de nuestros objetos:

- *Relleno* (de figuras).
- *Antialiasing*.
- *Recorte*.

6.1. Relleno de figuras

El **relleno de figuras** se usa sobre todo para dar una apariencia más realista a los objetos que estamos dibujando. El proceso se articula en dos pasos:

a) Escoger los pixels que tendremos que rellenar (pintar), es decir, detectar qué pixels quedan dentro de la figura que pretendemos pintar (rellenar).

b) Escoger el color que le vamos a asignar a cada pixel, algo que tiene mucho que ver con la iluminación y el sombreado (que veremos en el tema siguiente), así como también con el *antialiasing*[1].

Hay dos tipos o *técnicas de relleno*:

√ Por *barrido*.

√ Por *inundación*.

En el primero se supone que conocemos la estructura del objeto que queremos rellenar, ya sean sus vértices, las ecuaciones de las líneas que lo forman,... Tenemos una *línea de barrido* que cortará a la figura en cada instante en determinados puntos; algunos segmentos se rellenarán y otros no[2]. En caso de que no se conozca la figura, se utiliza el segundo método, en el que a partir de un punto nos "expandimos" hasta llenarla.

[1] Cerca de los bordes se darán ciertos matices,...

[2] Ver figura 3.22 en transparencias.

6.1.1. Relleno por barrido

Relleno de rectángulos

El relleno de rectángulos no supone problema alguno, ya que son figuras totalmente regulares, todas las líneas de relleno van a ser iguales. El principio de *coherencia espacial* tiene en ellos su máxima expresión.

Principio de Coherencia Espacial

> Normalmente, al pasar de una línea de barrido a la siguiente, lo que estamos rellenando va a cambiar muy poco, sea cual sea la figura que nos ocupe[3].

Lo que se hace para que los algoritmos de relleno funcionen mejor es buscar los *cambios en las figuras*. En los rectángulos, el principio y el final simplemente. Entre cambios, se puede maximizar el rendimiento evitando cambiar pixels de uno en uno (se puede hacer en bloque).

Relleno de polígonos

También en este caso tenemos una línea de barrido que recorre la figura. Usaremos algún algoritmo incremental o de punto medio para calcular los bordes al moverla a lo largo del polígono que se va a rellenar por dentro. Se podría hacer incluso sin la figura pintada en pantalla (en memoria).

Se siguen tres pasos:

1. **Búsqueda de las intersecciones de cada uno de los bordes.**
   Se usa un *método de paridad* (utilizando un bit con ese fin) para minimizar la lógica del programa. Inicializada la línea y el bit en cuestión, al pasar por una intersección la paridad cambiará, de suerte que si comenzamos en *par*, pintaremos cuando su valor sea *impar* y no rellenaremos cuando sea *par*[4].

   Deben tenerse en cuenta las siguientes consideraciones:

   - Si usamos un algoritmo incremental, de manera que tengamos $x_{i+1} = x_i + \frac{1}{m}$, el valor real $\frac{1}{m}$ puede hacer que tengamos que pintar el pixel inexistente $10'16$, por ejemplo.

     Se adopta el convenio de redondear hacia abajo al cruzar un borde de salida y redondear hacia arriba al cruzar un borde de entrada.
   - En los vértices podemos considerar que tenemos dos pixels, uno por cada linea que se cruza en ese punto. Por convenio de nuevo, sólo se cuenta aquél que es posición mínima (y_{min}) de una de las líneas.
   - Las líneas horizontales no cuentan para cambiar la paridad.
   - Como consecuencia de las dos anteriores, las líneas horizontales inferiores se pintan y las superiores no.

[3] Este principio es análogo al *principio de localidad en memoria*, etc.

[4] Es un ejemplo, el caso contrario sería también perfectamente válido.

Estos métodos tienen, no obstante, problemas con polígonos muy estrechos (slivers, *astillas*), que sólo tienen líneas de borde (problemas de resolución).

2. **Ordenación de dichas intersecciones en el eje X.**

   Como hemos mencionado, se usará un algoritmo incremental o de punto medio para ir siguiendo los bordes[5], teniendo en cuenta que sólo algunas aristas nos interesan para una línea de rastreo y que las aristas intersecadas por una línea de rastreo i también suelen ser intersecadas por la línea de rastreo $i+1$ (*coherencia de aristas*), de la siguiente manera:

   - Se crea una *tabla de bordes* en la que tenemos una lista de posiciones en el eje Y e información sobre cada uno de los bordes ordenados de acuerdo con su coordenada y menor (se mantienen también ordenados con respecto a la coordenada x creciente del punto extremo inferior):
     - Dónde acaba (y_{max}).
     - Dónde empieza (x_{min}).
     - La inversa de su pendiente.
   - *Tabla de bordes activos* (TBA): ordena las aristas de acuerdo con el valor de sus intersecciones en x para que se puedan rellenar los tramos. Está inicialmente vacía.
   - En la TBA se busca entre los bordes el de $y_{min} = y_{\text{del barrido}}$.
   - Iterativamente se rellenan los pixels limitados por los bordes en la línea de rastreo.
   - Se borran los bordes para los que $y_{max} = y_{\text{del barrido}}$ (aristas que no están comprendidas en la siguiente línea de rastreo).
   - Se incrementa y: $y = y + 1$ (coordenada de la siguiente línea de rastreo, se rellena de abajo a arriba).
   - Se actualizan los valores de las x para las aristas no verticales que permanezcan en la TBA, se incluyen los bordes nuevos en la TBA y se retorna al tercer punto hasta que nos quedemos sin bordes activos[6].

   Este *algoritmo de línea de rastreo* lleva el control del conjunto de aristas que interseca y los puntos de intersección de cada línea de rastreo.

3. **Relleno ó coloreado de los pixels.**

Relleno de curvas

El relleno de figuras como arcos, círculos y elipses sigue las mismas directrices que el relleno de polígonos en general, pero puede hacerse más eficiente aprovechando sus especiales características de simetría y redondeo.

[5] Ver algoritmo 3.26 en transparencias.
[6] Ver figura 3.27 en transparencias.

Relleno con patrones

En este tipo de relleno se cuenta con una *matriz* $N \times M$, que es lo que denominamos **patrón** y que, repitiéndose varias veces en la figura, nos ayudará a calcular el valor adecuado para cada píxel.

Las cuestiones que nos planteamos en el ámbito de esta técnica son: (1) ¿Cómo pintar? y (2) ¿Dónde pintar?

Para la primera de ellas tenemos dos opciones: en *modo opaco*, es decir, sustituyendo el valor del píxel que hay "debajo" del patrón por el nuevo valor, o bien *con distintas formas o tipos de transparencia*, de manera que uno de los colores del *bitmap* se considere transparente[7] y para los demás se hace la media o cualquier otra combinación entre los colores + fondo.

En cuanto a la segunda, se puede:

- Fijar una posición dentro de la figura y comenzar a repetir el patrón para rellenar a partir de ella. En este caso, si movemos la figura, nuestro relleno no cambia. El inconveniente es que en figuras complejas no es sencillo encontrar ese punto de inicio.
- Fijar el patrón a las coordenadas de la pantalla, calculando el pixel con respecto a dicho origen. Consecuentemente, si movemos la figura el relleno quedará fijo y no se moverá con ella.

Dado un punto de coordenadas (x, y), para saber qué color del patrón le corresponde se calcula:

$$\begin{aligned} x_p &= [x \bmod N] \\ y_p &= [y \bmod N] \end{aligned}$$

donde (x_p, y_p) será la posición del patrón de la cual recuperamos el color a adoptar por el pixel en cuestión.

6.1.2. Relleno por inundación

Los **algoritmos de relleno por inundación** se usan cuando no conocemos la estructura de las figuras y sólo contamos con un *bitmap* con pixels/posiciones de un color y pixels/posiciones de otro.

Estos algoritmos se estructuran en tres pasos:

a) Un *método de propagación*, que nos dice qué pixels inundamos en la siguiente iteración. En él reside la complejidad del algoritmo.

b) Un *procedimiento de comprobación*, que nos dice si un pixel está dentro de la región que tenemos que inundar.

[7] Esta técnica se usa en los GIF.

c) Un *procedimiento de cambio de valor del pixel*, que calculará el color que tiene que tener ese pixel.

Dentro de los algoritmos de inundación, necesitamos definir el concepto de **regiones** o **vecindades** (en inglés, neighbourhood), que son los pixels que están alrededor del que estamos estudiando. Distinguiremos dos tipos en cada una:

- *Vecindad 4-conectada*, en la que los "vecinos" son los pixels situados arriba, abajo, a la izquierda y a la derecha del pixel sobre el que nos encontramos.
- *Vecindad 8-conectada*, en la que los "vecinos" son todos los pixels de alrededor, los de la 4-conectada más los de las diagonales.

★ *Región 4-conectada*, aquélla a la que se puede llegar desde un píxel con movimientos a través de vecindades 4-conectadas.
Por ejemplo, en la siguiente situación tenemos dos regiones 4-conectadas diferentes:

★ *Región 8-conectada*, aquélla a la que se puede llegar desde un píxel con movimientos a través de vecindades 8-conectadas.
En el ejemplo anterior, hay una sola región 8-conectada.

Usaremos un tipo u otro de regiones en los algoritmos de inundación.

Dado un pixel P, también se tiene:

Región definida por el fondo La región conectada más grande en la que todos los pixels tienen el mismo valor que P.

Región definida por el borde La región conectada más grande a partir de dicho punto P en la que los valores de los pixels son distintos del valor de P.

El primer caso es difícil de aplicar en dibujos o fotografías reales, ya que lo que se tiene en realidad son áreas en las que la tonalidad de un color va cambiando gradualmente. Lo que se hace es añadir un *valor de tolerancia* y asumir que la región está formada por los vecinos que se mantengan dentro del rango marcado por éste (por ejemplo, con diferencias inferiores al 5 %).

Distintos Algoritmos de Inundación

Directo Es el más sencillo de todos. Consiste simplemente en, dado un punto inicial P escogido por el usuario, comprobamos a partir de él la vecindad (4 u 8) y sobre ella volvemos a aplicar el mismo algoritmo (es un procedimiento recursivo).

Por barrido Su programación es un poco más compleja. Dado un punto P, se inunda por barrido la línea en la que se encuentra, hasta llegar a un borde o bien hasta que el color de fondo deja de ser el mismo (*condición de parada*). Luego estudiamos las líneas superior e inferior a la actual (desde cada posición examinamos las que quedan arriba y abajo) e introducimos en una pila todos los *puntos de inicio* (los que están más a la derecha, más a la izquierda de un borde,...). Repetimos el proceso hasta que la pila esté vacía.

Inundación con bordes difusos (*soft filling*) Suele usarse en zonas próximas a líneas con las que se ha utilizado alguna técnica de *antialiasing*. Suponemos que tenemos un color de línea conocido L, frente a un color de fondo F que queremos cambiar. En la zona difusa el color de pixel es una *combinación lineal* de los dos colores:

$$P = tF + (1-t)L \qquad \text{con } 0 \leqslant t \leqslant 1$$

Sea N el nuevo color de fondo y tengamos algun algoritmo que nos diga qué pixels tenemos que cambiar, como los ya vistos. Entonces, como conocemos los valores anteriores P, F y L, podemos despejar t en alguna de las componentes RGB en las que la participación de F y L sea distinta, para que no se anulen (no puede ocurrir que se anulen todas porque de lo contrario $F = L$). Una vez obtenido, se calcula P con t y el nuevo N:

$$P_1 = tN + (1-t)L$$

6.1.3. Grosor

Veremos cuatro tipos de algoritmos para implementar las directivas con grosor. A parte del método, hay otras cosas a tener en cuenta, como por ejemplo la **brocha**, ¿qué forma ha de tener? Podría ser rectangular, cuadrada, circular,... Esta última es la que da mejores resultados, aunque también es la que conlleva más cálculos. En el caso de la rectangular habría que considerar la orientación que se le da en cada momento; debería girarse dependiendo de la tangente de la directiva sobre la que pintamos en cada punto para que quede igual siempre. Lo que ganamos en sencillez se pierde, pues, en estos cálculos, de modo que usaremos las brochas circulares.

Primer Método: Duplicación de Pixels

[8]

Consiste en escribir varios pixels por cada pixel calculado, lo que funcionará bastante bien con líneas, donde se siguen unos sencillos pasos:

[8] Ver figura 3.31 en transparencias.

1. Pintamos la primitiva.

2. *a*) Si la pendiente $|m| < 1$, añadimos pixels en vertical.

   b) Si la pendiente $|m| > 1$, añadimos pixels en horizontal.

   Es decir, si

   $$-1 < \text{ pendiente } < 1$$

   se duplican pixels en las columnas y en las filas para los demás casos.

Los *problemas* con que nos encontramos al aplicar este método son varios:

- Deben realizarse más cálculos en los vértices, donde se unen dos líneas, para que éstos queden bien.

- Con esta técnica el grosor cambia dependiendo del ángulo (en función de la pendiente m): las líneas horizontales y verticales se verán más gruesas que las que tienen cierta inclinación.

  $$0^o \quad \Rightarrow \quad \text{grosor } t$$

  $$45^o \quad \Rightarrow \quad \text{grosor mínimo } \frac{t}{\sqrt{2}}$$

- ¿Cómo pintaremos un grosor impar?

Segundo Método: Algoritmos de Brocha

En este caso contamos con un *bitmap* (rectangular, circular) que se desplaza por la trayectoria de la figura que estamos dibujando. Aunque se elimina el problema de los vértices en líneas con distinta inclinación, también ahora nos encontraremos con algunos contratiempos:

- El problema del grosor es opuesto a la situación anterior, también depende de la pendiente, pero

  $$0^o \quad \Rightarrow \quad t$$

  $$45^o \quad \Rightarrow \quad t \cdot \sqrt{2}$$

- Han de tenerse en cuenta *criterios de eficiencia*, pues en muchas ocasiones no hará falta pintar todos los pixels.

Tercer Método: Duplicación y Relleno de Directivas

La estrategia consiste en pintar dos directivas desplazadas $\frac{t}{2}$ y $-\frac{t}{2}$ de la posición en que realmente queremos dibujarla y acto seguido, rellenamos el espacio entre ambas. Así pues, para dibujar una línea, dibujaríamos un rectángulo:

Este método soluciona por completo los problemas con el grosor, incluso el grosor impar. También funciona con círculos[9]: si queremos centrar un círculo en (x, y) dibujaremos dos en ese punto, con radios $radio+\frac{t}{2}$ y $radio-\frac{t}{2}$.

No obstante, con elipses se nos presenta un inconveniente: si dibujamos con radios $a + \frac{t}{2}$, $a - \frac{t}{2}$ y $b + \frac{t}{2}$, $b - \frac{t}{2}$, nos encontraremos con que las dos elipses resultantes no son *confocales* (no tienen los focos en la misma posición), de modo que no obtendremos el resultado deseado.

Cuarto Método: Aproximación de Directivas con Polilíneas

Suele aplicarse sobre todo con curvas. Se trata de dividir la directiva en un conjunto de rectas y aplicar cualquiera de los algoritmos de grosor que hemos visto.

Deben tenerse en cuenta dos matices o consideraciones:

- En aquellos lugares de la directiva que pretendamos dibujar donde haya mucha variación en la pendiente debe subdividirse más la curva (en un mayor número de rectas).
- Debemos cuidar los vértices de las intersecciones para que el paso de una línea a otra sea lo suficientemente suave.

6.2. Antialiasing

Ya hemos visto cómo el dibujo en dispositivos con dos niveles de gris produce un fenómeno de visionado de "escaleras" en la trayectoria de las directivas que pintamos, que hemos denominado *aliasing*. Veremos a continuación cómo subsanar en parte este incómodo efecto.

Una solución posible podría consistir en *duplicar* los pixels para obtener mayor resolución. No obstante, si hacemos esto, el problema no se solucionará, sólo se aplazará, seguirá realmente ahí. Por otra parte, el uso de un mayor numero de pixels requerirá, además de más potencia de cálculo, más memoria de vídeo, lo que repercutirá en un hardware más caro y de todos modos no habremos arreglado nada.

[9]Ver figura 3.36 en transparencias.

La verdadera solución reside en usar *más niveles de gris* y utilizar los *pixels adyacentes*. Combinando estos dos principios y dependiendo de la figura usada para hacer la ponderación, aparecen dos tipos de técnicas: de **muestreo SIN** y **CON ponderación**.

Muestreo SIN ponderación

[10]

Supongamos que nuestra directiva tiene un grosor t. La idea consiste en dar una intensidad i a todos los pixels que queden debajo del "rectángulo" de la directiva[11], dependiendo dicha intensidad del área del pixel que quede debajo de la directiva, de modo que si el pixel queda casi totalmente debajo del rectángulo, será más oscuro, tendrá una mayor intensidad, y viceversa.

La línea ha de cumplir una serie de propiedades:

- La intensidad i del pixel disminuye al incrementarse la distancia entre el centro del pixel y la línea.
- La primitiva no influye en la intensidad de los pixels que no toca.
- Áreas iguales de solapamiento en los pixels suponen intensidades iguales para los pixels, independientemente de la parte por la que se solape, ya que la figura usada para la ponderación es un *cubo*.
- Es un buen compromiso entre complejidad y rendimiento.

Muestreo CON ponderación

[12]

En este caso definimos un *círculo* en el que está metido el pixel. Sobre él "colocaremos" un *cono* que ponderará la intensidad que se le otorgará al pixel según por dónde pase la directiva sobre él.

Las diferencias con el **muestreo SIN ponderación** son:

- Una línea que no pase por encima del pixel puede afectar su nivel de gris.
- Con líneas de grosor 1 no se consigue una intensidad o nivel de gris 0.

[10] Ver figura 3.57 en transparencias.
[11] Fig. 3.55 en transparencias.
[12] Ver figura 3.56 en transparencias.

- Es el único método que iguala la intensidad independientemente de la pendiente de la línea, obtiene siempre líneas de intensidad homogénea, aunque conlleve cálculos más complejos (las verticales y horizontales no son completamente negras, por lo mencionado en el punto anterior).

Por cada tipo de ponderación se construye a priori una tabla que en función de la distancia y el grosor calcula los valores de las intensidades de pixel. Para cada pixel calculamos la distancia entre su centro y la línea que pintamos usando el **algoritmo de Gupta-Sproull** (una modificación del algoritmo del punto medio)[13]:

Cada línea toca entre un mínimo de 1 y un máximo de 5 pixels, aunque lo normal son 3.

Igual que en el algoritmo del punto medio, se tiene una variable de decisión que llamamos $d = F(M) = F(x_{p+1}, y_{p+\frac{1}{2}})$.

$$F(x, y) = 2(ax + by + c) = 0$$

$$v = y - y_p$$

$$\left.\begin{array}{l} D = v \cdot \cos\theta \\ dx = \sqrt{dx^2 + dy^2} \cdot \cos\theta \end{array}\right\} \quad \boxed{D = \frac{v \cdot dx}{\sqrt{dx^2 + dy^2}}}$$

donde el denominador es una constante, ya que desde el principio conocemos dx y dy; lo que cambia es v. Intentaremos poner el numerador, pues, en función de d, que calculamos como entero en el algoritmo del punto medio normal:

$$F(x, y) = 2(ax + by + c) = 0 \quad \Rightarrow \quad y = \frac{ax + c}{-b}$$

y sustituyendo en

$$v = y - y_p$$

tenemos

$$v = \frac{a(x_{p+1})c}{-b} - y_p$$

multiplicando por $-b$

$$-bv = a(x_p + 1) + by_p + c$$

y como $b = -dx$([14])

[13] Ver pseudocódigo en transparencias.

[14] Ver apuntes sección 2.1.1 (página 10).

$$vdx = \frac{F(x_{p+1}, y_p)}{2}$$

Operando...

$$\begin{aligned} 2vdx &= F(x_{p+1}, y_p) \\ 2vdx &= 2a(x_{p+1}) + 2by_p + 2c \\ &= 2a(x_{p+1}) + 2b(y_{p+\frac{1}{2}}) + 2\frac{b}{2} + 2c \\ &= \mathbf{d + dx} \end{aligned}$$

Entonces

$$D_E = \frac{2vdx}{2\sqrt{dx^2 + dy^2}} = \frac{d + dx}{2\sqrt{dx^2 + dy^2}}$$

donde d ya se tiene del algoritmo del punto medio, dx se calcula sólo una vez, al principio, igual que el denominador. De modo que simplemente hay que sumar y dividir entre una constante cada vez.

ARRIBA $(1 - v)$

Sustituimos en la ecuación y trabajamos con $2(1 - v)dx$, que es

$$2(1 - v)dx = 2dx - 2vdx$$

donde $2dx$ es una constante y $2vdx$ ya va siendo calculado como hemos visto.
Se obtiene $D_{E\text{arriba}}$.

ABAJO $(1 + v)$

Igual que antes...

$$2(1 + v)dx = \underbrace{2dx}_{\text{cte.}} + \underbrace{2vdx}_{\text{ya calculado}}$$

Se obtiene $D_{E\text{abajo}}$.

En el caso de escoger el punto NE:

$$2vdx = F(x_{p+1}, y_{p+1})$$

y

$$v = y - y_{p+1}$$

Operando llegamos a...

$$\begin{aligned} 2vdx &= F(x_{p+1}, y_{p+1}) \\ 2vdx &= 2ax_{p+1} + 2by_{p+1} + 2c \\ &= 2ax_{p+1} + 2by_{p+\frac{1}{2}} + 2\frac{b}{2} + 2c \\ &= \mathbf{d} + \mathbf{b} \\ &= \mathbf{d} - \mathbf{dx} \end{aligned}$$

Ahora

$$D_{NE} = \frac{d - dx}{2\sqrt{dx^2 + dy^2}}$$

Igual que en el caso anterior.

Calculada D, se acude a las mencionadas tablas y con el grosor que queramos obtenemos la intensidad de gris con que pintaremos el pixel.

D	t	Intensidad
0	1	0 (negro)
0'1	"	⋮
0'2	"	
⋮	"	⋮
$D_{\max}$	"	255 (blanco)

Cuadro 6.1: Ejemplo de tabla SIN ponderación.

6.3. Recorte

El proceso de **recorte** selecciona los objetos que van a quedar delante de la ventana de vista.

Cuando el modelo es complejo o muy grande, el *recorte* nos hará más eficiente su dibujado final, ya que no se calculará la proyección de todos sus puntos si al final no se van a ver a través de la ventana.

El paso de recorte se inserta:

Normalmente tendremos un **rectángulo de recorte** que tendrá como coordenadas significativas que usaremos en los cálculos las siguientes:

Del recorte de un rectángulo siempre se va a obtener un rectángulo. Del recorte de un polígono, uno o más polígonos. Del de círculos y elipses, como mucho cuatro arcos. Y del de una línea, siempre otra línea.

6.3.1. Recorte en cuanto a vértices o puntos extremos

Este es un recorte clave, ya que lo usaremos después en el resto de figuras. Consiste en, dado un vértice situado en (x, y), para ver si lo pintamos o no simplemente se comprueba:

$$x_{\min} \leqslant x \leqslant x_{\max}$$
$$y_{\min} \leqslant y \leqslant x_{\max}$$

Si no se cumple alguna de las cuatro desigualdades, el vértice (x, y) queda fuera del rectángulo de recorte.

6.3.2. Recorte de líneas

En este tipo de recorte hay que comprobar la posición de los extremos de la línea:

- Si los dos están dentro, se pinta la línea entera, pues quiere decir que queda por entero dentro del rectángulo de recorte.
- Si uno está dentro y el otro está fuera, hay un segmento de línea que hay que pintar. Lo que hay que hacer es buscar la intersección de la línea con los bordes del rectángulo.
- El caso más problemático es en el que los dos vértices están fuera, pues puede entonces ocurrir que la línea esté completamente fuera o puede ser que haya un segmento que esté dentro[15]. Para este caso, se hacen una serie de tests con el fin de comprobar si hay o no que pintar la línea. Existen tres algoritmos posibles (en orden creciente de eficiencia):
  - *Cálculo directo* o *cálculo de las intersecciones del segmento con los bordes.*
  - *Algoritmo de Cohen-Sutherland.*
  - *Algoritmo de Cyrus-Beck.*

[15]Ver todos estos ejemplos en fig. 3.38 de transparencias.

Cálculo directo

Usaremos la ecuación paramétrica de la recta.

Tenemos una recta entre dos puntos P_0 (de coordenadas (x_0, y_0)) y P_1 (de coordenadas (x_1, y_1)). Cualquier punto en ella estará representado por:

$$\begin{array}{l} x = x_0 + t(x_1 - x_0) \\ y = y_0 + t(y_1 - y_0) \end{array} \qquad t \in [0, 1]$$

Este enfoque pretende intersecar la línea con las cuatro aristas del rectángulo de recorte para ver si alguno de los puntos de intersección está en esas aristas, pues de ser así indicará que la línea cruza el rectángulo de recorte y se encuentra parcialmente dentro de él. Por tanto, para cada línea y cada uno de los cuatro bordes del rectángulo se toman las dos líneas matemáticamente infinitas que las contienen y se intersecan. Después se determina si el punto de intersección está dentro del rectángulo de recorte y de la línea, es decir, se hayan los valores de la t en que se produce la intersección:

línea	$x = x_0 + t_l(x_1 - x_0)$	$y = y_0 + t_l(y_1 - y_0)$
bordeizq	$x = x_{\min}$	$y = y_{\min} + t_b(y_{\max} - y_{\min})$

Con $x = x_{\min}$ sustituimos en $x = x_0 + t_l(x_1 - x_0)$ y obtenemos t_l. Sabiendo t_l podemos obtener y de la ecuación $y = y_0 + t_l(y_1 - y_0)$; y por último, conociendo y, obtenemos t_b de $y = y_{\min} + t_b(y_{\max} - y_{\min})$.

Conocidos t_l y t_b, para saber que se produce la intersección donde a nosotros nos interesa tenemos que comprobar que sus valores están entre 0 y 1:

Algoritmo de Cohen-Sutherland

Se basa en dividir el espacio en 9 regiones alrededor del rectángulo de recorte[16]. En 3D se hace de forma similar, pero se tiene un cubo de recorte y 27 regiones, pero ya lo veremos más adelante.

En primer lugar se hacen pruebas para rechazar/aceptar directamente las líneas: por ejemplo, si los dos extremos quedan dentro se pinta. ¿Cómo se hace el rechazo? Si tenemos los dos extremos fuera pero caen ambos a la izquierda, ambos a la derecha, ambos arriba o ambos abajo del rectángulo de recorte, no puede haber ninguna parte de la línea que quede dentro, de modo que se rechaza.

Si tenemos los dos extremos fuera pero no ocurre lo anterior, se extienden los segmentos del rectángulo de recorte y se consiguen las citadas 9 zonas, a las que vamos a asignar (el orden da igual) un código de 4 bits según el cual sabremos qué bordes tenemos que atravesar para pintar la línea.

En el ejemplo (figura 3.39 de las transparencias):

- El primer bit indica si estamos por encima del rectángulo de recorte.

[16] Ver figura 3.39 en transparencias.

- El segundo bit indica si estamos por debajo.
- El tercero, si estamos a la derecha.
- El cuarto, si estamos a la izquierda.

Una vez definidas las zonas, hemos de etiquetar los extremos de las líneas que queremos recortar con el código seleccionado, dependiendo de la zona en la que caigan.

Con los extremos etiquetados, se pueden hacer las comprobaciones de forma más rápida:

- Un código 0000 significa que estamos dentro.
- Si hacemos un AND bit a bit de dos extremos y obtenemos un 1 (algún bit nos da 1), se puede rechazar, pues estamos en el caso en el que los dos están a la derecha, a la izquierda, arriba o abajo.
- Con el resto de líneas llevamos a cabo un proceso iterativo en el que subdividimos cada línea que no se adapta a ninguna de las dos condiciones anteriores y vemos si el segmento que queda las cumple; en caso contrario, lo volvemos a dividir...

  ¿Cómo es este proceso? Para cada extremo se calcula la intersección con el borde que indique el primer bit que tenga a 1 su código de región. Eso nos dará el nuevo segmento, después recalculamos su código y volvemos a comprobar si la línea está dentro, la podemos rechazar o de nuevo tendremos que subdividirla. El bucle se repite hasta que se pueda tomar una decisión.

El problema de este algoritmo es que, al ser iterativo, normalmente se van a realizar operaciones que no son eficientes (se calcularán intersecciones que no nos valdrán). No obstante, funciona bien en dos casos: si el *rectángulo de recorte* es *grande* (se aceptan casi todos los objetos) y si es *muy pequeño*[17] (casi todos los objetos se van a rechazar). Es decir, funciona bien cuando se realizan las dos comprobaciones directas y no se ejecuta el proceso iterativo.

Algoritmo de Cyrus-Beck o de Recorte Paramétrico

Este algoritmo es posterior y mucho mejor que el algoritmo de Cohen Sutherland cuando hay muchas líneas u objetos que recortar, ya que cuenta con la ventaja de no ser iterativo.

Se llama de *recorte paramétrico* porque usaremos las ecuaciones paramétricas de las líneas. De forma que si tenemos una recta entre P_0 y P_1, cualquier punto entre ellos que pertenezca a la misma tendrá una expresión:

$$P(t) = P_0 + (P_1 - P_0)t \qquad \text{con } t \in [0, 1]$$

Calcularemos el *valor de t de la línea*[18] para la intersección con los cuatro bordes del rectángulo de recorte.

Para cada borde también calcularemos la *normal* N_i, que nos indicará la parte exterior del rectángulo de recorte.

[17] Se usa para simular el área de click del ratón.

[18] No como en el algoritmo de cálculo directo, que calculábamos la t para la línea y para los bordes.

$$v_1 \cdot v_2 = v_{1x} \cdot v_{2x} + v_{1y} \cdot v_{2y}$$

Cuadro 6.2: Producto escalar de dos vectores.

Si definimos un punto del borde izquierdo en cualquier lugar entre el propio borde izquierdo y la línea, su *producto escalar* dará cero, puesto que serán perpendiculares[19]. De este producto nulo despejaremos los valores de t.

$$N_i \cdot [\underbrace{P(t)} - P_{E_i}] = 0$$

$$N_i \cdot [P_0 + (P_1 - P_0)t - P_{E_i}] = 0$$

$$N_i \cdot [P_0 - P_{E_i}] + N_i \cdot [P_1 - P_0]t = 0$$

$$t = \frac{N_i \cdot [P_0 - P_{E_i}}{-N_i \cdot D} \qquad \text{donde} \quad D = [P_1 - P_0]$$

Esta fórmula, aunque parezca complicada, proporciona unos resultados muy sencillos[20]. Lo único que se debe controlar es el denominador, es decir, que la normal N_i no sea nula (por error), que $D \neq 0$ (o sea, que $P_0 \neq P_1$) y que ambos no sean paralelos, pues en tal caso no existiría intersección.

El primer paso, pues, es calcular los 4 t_l (uno con respecto a cada borde del rectángulo de recorte.

El segundo consiste en desestimar aquellos valores de $t \notin [0, 1]$, ya que representan intersecciones que quedan fuera del rectángulo de recorte:

En tercer lugar, ordenaremos el resto de t's y las etiquetaremos como *potencialmente salientes* o *potencialmente entrantes*.

Una intersección se dirá:

Potencialmente entrante (PE) Cuando pasamos del lado de fuera al lado de dentro del rectángulo de recorte con respecto a cada borde[21].

Potencialmente saliente (PS) Cuando pasamos del lado de dentro al lado de fuera del rectángulo de recorte con respecto a cada borde.

[19] Ver figura 3.42 de transparencias.

[20] Ver transparencias, tabla 3.1.

[21] Tomando como sentido para la ínea el $P_0 \rightarrow P_1$. Ver figura 3.43.

Se cumple que:

$$\begin{array}{l} \text{PS si el ángulo entre } N_i \text{ y } P \text{ es } < 90^o \\ \text{PE si el ángulo entre } N_i \text{ y } P \text{ es } > 90^o \end{array} \quad \Leftrightarrow \quad \begin{array}{l} \text{PS si } N_i \,.\, P > 0 \\ \text{PE si } N_i \,.\, P < 0 \end{array}$$

Lo cual nos beneficia porque el producto $N_i.P$ ya lo tenemos calculado del primer paso.

Por último, nos quedaremos con el PE de mayor valor de t y con el PS de menor valor de t. Estas dos intersecciones nos darán el segmento recortado, el que queda dentro del rectángulo de recorte.

Si resultase que sólo tenemos puntos PS, entonces nos quedaríamos con P_0 y el PS de menor t. Si sólo tuviésemos puntos PE, tomaríamos P_1 y el PE de mayor t. Estos casos remiten a situaciones como:

Si toda la línea resultase quedar fuera del rectángulo de recorte[22], los valores de t de los PS serían mayores que los de los PE, lo que nos indicaría que debemos desestimar la línea entera.

6.3.3. Recorte de círculos

Para proceder al recorte de un círculo, calcularemos el cuadrado que lo rodea, que se conoce como **extensión** y que nos servirá para hacer una primera aproximación, pues hallaremos sus intersecciones con el rectángulo de recorte.

Si no se cortan, ignoraremos el círculo (entendiendo que en ese caso queda completamente fuera del rectángulo de recorte). En caso de que se detecte alguna intersección entre ellos, se divide el círculo en cuadrantes y se vuelve a comprobar de la misma forma para ver dónde está la intersección. Este procedimiento podría seguirse con los octantes, calculando después ya la intersección real entre círculo y rectángulo de recorte para saber de dónde a dónde tenemos que pintar.

La estrategia para las elipses es similar.

6.3.4. Recorte de polígonos

Del recorte de un polígono, como ya comentamos, podemos pasar a tener n polígonos. El recorte de un polígono puede:

- Añadir bordes nuevos.
- Eliminar bordes que ya existen.
- Segmentar otra parte de ellos.

[22]Línea 2 en figura 3.43 de transparencias.

El algoritmo más frecuentemente utilizado es el **algoritmo de Sutherland-Hodgman**. Este algoritmo reparte o divide el trabajo del recorte en 4 pasos, haciendo el recorte individualmente por cada uno de los bordes[23]:

1) Todo lo que quede a la derecha de la línea del borde derecho (no sólo lo que quede a la derecha del propio borde) se elimina.

1) Se elimina todo lo que quede por debajo de la línea del borde inferior.

1) Todo lo que se tenga a la izquierda de la línea del borde izquierdo también se elimina.

1) Y por último también se elimina todo lo que quede por encima de la línea del borde superior.

Se trata, pues, de una estrategia *divide y vencerás*, que resuelve una serie de problemas sencillos e idénticos que al combinarse resuelven el problema general (el problema sencillo es el recorte con respecto a una sola arista).

Una vez hecho esto, se analizan los vértices de la figura que estamos recortando con respecto a cada uno de los bordes sucesivamente. Nótese que esto es lo que lo diferencia principalmente del *algoritmo de Cohen-Sutherland*, ya que recortamos todas las líneas con respecto a todos los bordes, mientras que con *Cohen-Sutherland* es al contrario: según la posición de la línea elegimo el borde con respecto al cual la recortaremos.

Empezando en uno cualquiera, se recorren consecutivamente para ver si están en el semiplano[24] interior o en el semiplano exterior (respecto al rectángulo de recorte que con las forma en su intersección con las otras tres líneas). Para esto se puede evaluar el signo del producto escalar de la normal a la línea y la arista del polígono, como se veía en la sección 6.3.2 (página 73).

Los vértices que quedan dentro se guardan en una *lista de vértices*, los que quedan fuera los desechamos y cuando pasamos de dentro a fuera (o viceversa)[25] se genera un vértice nuevo para la lista[26]. Esta lista de vértices contendrá al final los puntos que deberemos unir para trazar nuestro(s) nuevo(s) polígono(s), que sí se verá(n).

6.3.5. Recorte 3D

En 3D en lugar de un *rectángulo de recorte* tenemos un **volumen de vista** que delimitará lo que veremos proyectado en la pantalla. Se pintará lo que quede dentro de ese volumen[27]; haremos un recorte de todos los objetos de nuestro modelo de forma que sólo dibujamos lo que vamos a ver:

Modelo 2D ⟶ Recorte ⟶ Dibujo	Pasos para 2D.
Modelo 3D ⟶ Recorte ⟶ Proyección ⟶ Dibujo	Pasos para 3D.

[23] Ver figura 3.47 en transparencias.
[24] Cada línea que contiene a un borde divide el plano de vista en dos semiplanos.
[25] Lo detectamos porque un vértice resulta estar dentro y el siguiente consecutivo, fuera.
[26] Ver figura 3.48.
[27] Ver transparencias página 262.

En el caso de 3D la operación de *recorte* es una operación costosa, que se puede simplificar no obstante si usamos **volúmenes de vista normalizados**[28]:

Antes de aplicar el recorte, por tanto, haremos una *normalización* de todas nuestras figuras, luego recortaremos (entonces será más sencillo) y posteriormente proyectaremos y dibujaremos. Cada vez que modifiquemos el volumen de vista para que sea normal, las modificaciones se aplicarán también a todos los objetos de nuestro mundo.

NORMALIZACIÓN DEL VOLUMEN DE VISTA

Se pretende hacer coincidir las coordenadas de vista y las del mundo:

$$T(u, v, n \longrightarrow x, y, z)$$

$$R \cdot T(u, v, n \longrightarrow x, y, z)$$

Puede ser necesario aplicar una operación de *share*:

$$SH \cdot R \cdot T(u, v, n \longrightarrow x, y, z)$$

Por último, se realiza un escalado tanto del volumen de vista como de los objetos, para obtener un volumen normalizado:

$$S \cdot SH \cdot R \cdot T(u, v, n \longrightarrow x, y, z)$$

Todas las matrices pueden componerse y obtener una única matriz.

Los pasos son similares tanto para proyecciones paralelas como para proyecciones en perspectiva, sólo que el volumen en este caso es una pirámide.

Una vez tenemos el volumen normalizado, podemos aplicarle el recorte. Se usan los mismos algoritmos que para 2D, adaptados a 3D:

- Algoritmo de Cohen-Sutherland.
- Algoritmo de Cyrus-Beck.

Algoritmo de Cohen-Sutherland

En lugar de tener 4 bits como en 2D, en 3D vamos a tener dos bits más a mayores, esto es, un total de 6 bits, que nos representarán las 27 zonas, cada una con su código, en que queda dividido el espacio.

[28] Trasparencias figura 3.45.

De nuevo, si tenemos los dos vértices de una línea en la zona en la que el código es todo ceros, estarán dentro y la pintaremos por completo. Si tenemos un extremo dentro y otro fuera realizaremos el recorte y si están los dos fuera, sucesivos recortes iterativos siguiendo la misma filosofía que ya vimos en su momento (sección 6.3.2, página 71).

El valor de los bits se indica a continuación según el tipo de proyección utilizada:

Proyecciones paralelas

b_1 $y > 1$ indica que estamos sobre el volumen de recorte
b_2 $y < -1$ indica que estamos bajo el volumen de recorte
b_3 $x > 1$ indica que estamos a la derecha
b_4 $x < -1$ indica que estamos a la izquierda
b_5 $z < -1$ indica que estamos detrás
b_6 $z > 0$ indica que estamos delante

Proyecciones perspectiva

b_1 $y > -z$ = estamos sobre el volumen de recorte
b_2 $y < z$ = estamos bajo el volumen de recorte
b_3 $x > -z$ = estamos a la derecha
b_4 $x < z$ = estamos a la izquierda
b_5 $z < -1$ = estamos detrás
b_6 $z > z_{\min}$ = estamos delante

donde $z_{\min} < 0$ es la distancia a la que está el plano delantero *front plane*.

Algoritmo de Cyrus-Beck

Su funcionamiento es análogo. Utilizaremos la expresión:

$$t = \frac{N_i \cdot (P_0 - P_{E_i})}{-N_i \cdot D}$$

que al desarrollar da 6 ecuaciones en lugar de 4, pero igual de sencillas.

Ahora cualquier punto de $P(t)$ se expresará:

$$\begin{aligned} x &= x_0 + t(x_1 - x_0) \\ y &= y_0 + t(y_1 - y_0) \\ z &= z_0 + t(z_1 - z_0) \end{aligned}$$

Se calcularán 6 valores para t, de los cuales descartaremos los que no estén entre $[0, 1]$, ordenaremos el retos, los etiquetaremos PE o PS y escogeremos el de t mayor de entre los PE y el de t menor de entre los PS para dibujar el segmento entre ellos.

Capítulo 7

Tratamiento de figuras 3D

En este tema trataremos dos cuestiones fundamentales en el tratamiento de figuras 3D, como son la **detección de superficies visibles** y la **Iluminación**. Estos dos puntos se introducen entre la fase de *recorte* y la de *proyección*, puesto que en ésta perdemos la información de profundidad que hace falta para la iluminación, por ejemplo.

7.1. Determinación de superficies visibles

Es el proceso de selección de líneas o superficies que son visibles desde el centro de proyección / en la dirección de proyección (dependiendo de la proyección usada).

¿Por qué hacer detección de superficies visibles?

Por dos razones principales:

- Buscamos realismo.
- Por eficiencia: se va a invertir un trabajo en seleccionar unas superficies, pero se va a ahorrar en las que no se pintarán a consecuencia de ello.

Aproximaciones: ¿cómo hacerlo?

Hay dos tipos de estrategias:

- Se puede hacer por pixels.
- Se puede hacer por objetos.

El primer caso consistiría en analizar, para cada pixel, qué objetos se "ven" a través de él, qué vemos más cerca (para poner ese color en pantalla). Limitamos la visión a ese pixel. Si se tienen p pixels y n objetos, el poder de computación que esto requiere es del orden de $p \times n$, lo que supone un trabajo bastante costoso.

En el segundo caso veríaos qué objetos se tienen y, cogiéndolos de dos en dos, estudiaríamos cuáles quedan delante de cuáles. En este caso el trabajo son n^2 operaciones de comparación, algo que puede ser más eficiente, ya que normalmente $n \lll p$, pero ello

dependerá de la complejidad de los propios objetos.

En general, si n es pequeño y los objetos son regulares, se usará la estrategia de aproximación por objetos, y en caso contrario, la aproximación por pixels.

Consideraciones

Además de señalar que la detección de superficies visibles ha de llevarse a cabo antes de la proyección, haremos otra serie de matizaciones:

√ Dados dos puntos $P_1(x_1, y_1, z_1)$ y $P_2(x_2, y_2, z_2)$, ¿cuál está delante? Si la proyección es paralela, si $x_1 = x_2$ e $y_1 = y_2$ ambos están en el mismo *proyector*, de modo que estarán uno delante del otro dependiendo de su z. Si la proyección es perspectiva, estarán en el mismo proyector si $\frac{x_1}{z_1} = \frac{x_2}{z_2}$ y $\frac{y_1}{z_1} = \frac{y_2}{z_2}$.

√ Por razones de eficiencia, puede ser interesante buscar la *extensión* de las figuras[1] (en el caso de que haya pocas figuras), ya que si sabemos dónde está un objeto no tenemos que recorrer el resto de las líneas:

√ También por eficiencia puede ser interesante realizar una previa *detección de caras ocultas*: calculando el producto escalar de la dirección de proyección y la normal cada cara de la figura, $\mathsf{DOP}\boldsymbol{.}N$, y analizando su valor:

- ▷ Si es $= 0 \Rightarrow$ perpendiculares, de modo que esa cara no la veremos.
- ▷ Si es $> 0 \Rightarrow$ es una cara que "apunta hacia atrás", así que tampoco la veremos.
- ▷ Si es $< 0 \Rightarrow$ es una cara que "apunta hacia delante", aunque nada nos asegura que la vayamos a visualizar. Por ello, este paso previo es útil para descartar, no para seleccionar.

Algoritmos

A continuación veremos una serie de diferentes algoritmos que se ocupan de la detección de superficies visibles:

(1) Determinación de líneas visibles.

(2) Z-Buffer.

(3) Lista de prioridad.

(4) Líneas de barrido.

(5) Subdivisión de áreas.

(6) Trazado de rayos.

[1] Se refiere a la estrategia que empleábamos para el recorte de círculos.

7.1.1. Algoritmos de Determinación de Líneas Visibles.

Su salida van a ser las líneas o segmentos de línea que hay que pintar como líneas visibles. Un ejemplo es el **algoritmo de Appel**:

- ↦ Sólo se consideran las caras frontales.
- ↦ Se define una variable, cantidad de visibilidad, con la que se va a caracterizar cada segmento de una línea. Si la línea se ve, su valor es cero. Cada vez que pasa por detrás de una cara frontal se incrementa en 1. Lo que se hace es mirar si pasa por detrás de un borde y el tipo de éste (transparencias figura 15.19).
- ↦ Es un algoritmo orientado a polígonos que no se insertan uno en otro.

7.1.2. Algoritmo Z-Buffer

Vamos a definir un *buffer de pantalla* (podría ser nuestro propio rectángulo de recorte), donde en cada posición vamos a almacenar un color, el que vamos a ver. Asimismo, definiremos otro buffer, al que llamaremos *z-buffer*, que será una matriz con las mismas dimensiones que el anterior, que almacenará los valores de profundidad (valores en el eje z, de ahí su nombre).

El z-buffer se inicializa a la distancia más lejana y el buffer de pantalla al color de fondo. A continuación se pintan todos los objetos; si al ir a pintar un punto (color) correspondiente a un objeto ya hay otro almacenado cuya z correspondiente es menor (lo que significará que está delante), no lo pondremos.

Entre las ventajas de este algoritmo está la posibilidad de pintar cualquier tipo de objeto sin que, además, haga falta calcular intersecciones. Sin embargo, acarrea un consumo doble de memoria, algo que por otra parte hoy en día no supone un problema.

7.1.3. Algoritmos de Lista de Prioridad

En estos algoritmos se busca ordenar los objetos en cuanto a su coordenada z, de forma que cuando esté hecha esta ordenación empecemos a pintar por los que están más lejos, sin preocuparnos, pues, de qué partes no pintamos, etc.

Los problemas que se presentan se ven claramente en la figura 15.24 de transparencias (objetos solapados cíclicamente, por ejemplo). En estos casos, además de la ordenación, hay que subdividir los objetos. Veremos una forma de afrontar esto:

Árboles BSP (Binary Space Partitioning)

Son árboles de división binaria del espacio. Se usan cuando los objetos son fijos e interesa cambiar el punto de vista, por ejemplo.

Se trata de dividir el conjunto de objetos del modelo de forma que en esa división los objetos de delante de esa división (los de un lado) siempre oculten a los del otro lado (los de detrás), y que los que etiquetamos como de "detrás" nunca queden por delante de ninguno de "delante".

Elegiremos uno de los objetos como raíz, dividirá en dos el modelo. Seguidamente hay que establecer qué es delante y qué es detrás (no hay una norma preestablecida). Los objetos que queden en medio de la división se fragmentan[2] y se reetiquetan, una parte de un lado y otra parte del otro. El proceso sigue hasta que sólo queda uno delante y otro detrás (nodos de un elemento).

Este árbol se construye una sola vez. Después podemos cambiar el punto de vista con el mismo árbol donde queramos y éste nos dice cómo pintar los objetos. Dado un punto de vista, se mira dónde estamos respecto al raíz. Si por ejemplo estamos delante pintaremos en este orden: objetos detrás, raíz, objetos delante, y así sucesivamente para cada vértice a pintar.

7.1.4. Algoritmos de Líneas de Barrido

Estos algoritmos se basan en la misma técnica que vimos para inundación por barrido. Se mantienen una *tabla de bordes* (ET), una *tabla de bordes activos* (AET) y a mayores una *tabla de polígonos* (PT) donde guardamos información del plano, de cómo sombrearlo (color de los puntos) y de si el barrido está dentro o fuera de cada uno en cada momento. Cuando se está pintando dentro de dos, se calcula la distancia, cuál está delante, y se pinta con su información de sombreado.

7.1.5. Algoritmos de Subdivisión de áreas

La estrategia de estos algoritmos consiste en subdividir la pantalla en partes en las que sea más fácil la tarea de detectar la superficie visible.

Se trata de una división recursiva. Mientras se tengan problemas, se sigue haciendo la división.

El más simple de estos algoritmos es el de **Warnock**, que contempla cuatro posibilidades de interacción de los polígonos con las partes de la pantalla (figura 15.42):

1. Que el polígono quede totalmente alrededor de la pantalla.
2. Que la interseque.
3. Que quede totalmente dentro de ella.
4. Que quede totalmente fuera.

El algoritmo lleva a cabo, pues, para cada parte de la pantalla, cuatro comprobaciones:

1. Si todos los polígonos son disjuntos (están fuera); en ese caso no hay problema y se pinta el color de fondo.

[2]Como el caso del 5a, 5b en la figura 15.31.

2. Si tiene un sólo polígono que interseca; entonces se pinta el fondo y la parte del polígono que corresponda.

3. Si hay un polígono que rodea dicha parte de la pantalla; tampoco supone un problema, se pinta el color del polígono.

4. Si conviven varios polígonos, de forma que uno de ellos rodea y queda por encima del resto; se pinta ese polígono.

Si no se da ninguna de las anteriores situaciones, entonces dividimos, porque el problema es complejo, y volvemos a estudiar las cuatro subdivisiones (figura 15.44).

El problema con que nos encontramos deriva de que trabajamos con un bitmap que tiene una resolución finita (por ejemplo, 1024×860), de modo que llegará un momento en el que alcanzaremos el nivel de pixel tras hacer varias subdivisiones recursivas, y sin que se nos de ninguna de las cuatro situaciones "no problemáticas" anteriores. Lo que se hace es calcular para dicho pixel la intersección con los objetos, ver cuál está más cerca y ponerle su color.

Entre otros algoritmos de subdivisión de áreas, el más usado es el de **Weiler-Atherton**, que se basa en la misma idea pero en lugar de subdividir la pantalla subdivide los propios objetos que dibujamos (se segmentan unos objetos respecto a otros). Esto evita el problema de la resolución, pero hace necesario un algoritmo de recorte y búsqueda de intersecciones mucho más complejo (figura 15.48).

7.1.6. Algoritmos de Trazado de Rayos

En estos algoritmos, lo que se hace es, desde un punto de vista y trabajando con un plano de proyección, *disparar* una especie de "rayo" (en realidad un proyector) hacia los objetos de nuestro modelo. De esta manera conoceremos qué objetos están en la trayectoria de ese rayo y nos quedaremos con el que esté más cerca del punto de vista. Se repite para cada pixel del plano de proyección.

Es un tipo de algoritmo que se desarrolló para sombreado e iluminación, pero que es perfectamente aplicable a detección de superficies visibles.

7.2. Iluminación

En esta sección veremos algunas técnicas que permiten calcular el nivel de color para cada punto de una figura 3D, con vistas a proporcionarle un mayor realismo. Para hacer esto se tienen diferentes **modelos**.

7.2.1. Modelos de iluminación

El más sencillo es el denominado **luz ambiente** o **luz ambiental**. En él no tenemos ninguna fuente de luz y los objetos van a tener un color plano (definimos un color que es el que se le pone a todos los pixels de un objeto), sin efectos de sombreado,...

Otro modelo es la **reflexión difusa**, donde tenemos un foco puntual de luz y el color de cada pixel se calcula en función del ángulo que forma el rayo de luz con el plano del pixel y en función de la distancia al foco de luz.

El modelo de **atenuación atmosférica** intenta dar una apariencia más real teniendo en cuenta condiciones de atenuación en la distancia debidas a la presencia de una atmósfera.

La **reflexión especlar** es un modelo que reproduce el efecto que se produce cuando se mira un objeto brillante y pulido bajo una luz: hay una parte del mismo que se ve del color de la luz y no del color del objeto.

Estos son modelos de iluminación por pixel. Pero nuestros objetos se componen normalmente de polígonos y es impensable aplicar esto a cada uno de los pixels que los componen. Lo que se hace es un cálculo para el polígono o sus partes de y extenderlo a todo él. Las dos aproximaciones que más se utilizan son:

Sombreado constante Se calcula el color para un punto de un polígono del objeto con alguno de los modelos anteriores y se le da ese color a todos los demás polígonos. Es la aproximación más rápida, pero también la que peores resultados reporta.

Sombreado interpolado Necesita más cálculo, pero sus resultados son notablemente mejores. Se calcula el color para los vértices del polígono, y se interpolan los valores para el resto de puntos.

Lo que acabamos de describir es sencillo en figuras simples. En figuras más complejas, normalmente aproximadas por polígonos, la aplicación de estas estrategias provocará que veamos los bordes de dichos polígonos, de modo que no nos valen. Los que se usa son:

Sombreado de Gouraud Para cada vértice se calcula la normal como una media de las normales de las caras adyacentes.

Con respecto a esas normales se calcula el color que les corresponde a los vértices y después se interpola para todo el polígono.

Sombreado de Phong En este caso se interpolan directamente los valores de las normales traducidos a colores. Conlleva más cálculo pero, lógicamente, queda mejor.

7.2.2. Modelos de iluminación global

Se tienen dos modelos principales de iluminación global:

Raytracing Es un proceso recursivo que recrea el fenómeno natural que se produce con la luz, que cuando llega a una superficie se refleja, llega a otra, y así sucesivamente hasta que se le acaba la energía. Es largo y complejo, pero se puede detener en cualquier momento (cuanto más se itere, más real quedará).

Radiosity Se basa más en la parte física de la energía: calcula qué parte de la misma se refleja y cuál se absorbe, qué parte de ésta se vuelve a emitir, etc. La ventaja es que se calcula una vez y si se mantienen las condiciones de iluminación, puede verse el modelo desde cualquier sitio sin recalcularlo de nuevo.

Índice general